Tutorial on Semantics
Part II
Domain Theory

Prakash Panangaden

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

Fields Institute, Toronto: 20th June 2011
Outline

1. Cartesian closed categories
2. Approximation and continuous domains
3. Categories of algebraic domains
4. Denotational semantics of PCF
5. Adequacy of the denotational semantics
6. Full abstraction
What we need

- We saw that we needed fixed-point theory at all types.
- We therefore need to define models of data types that support this.
- We also need functions between data types to be data types.
- Since we are looking at properties of all data types together we need to look at the category of data types.
A **category** \( C \) consists of two collections: \( C_0 \) **objects** and \( C_1 \) **morphisms**. There are functions \( \text{dom}, \text{cod} : C_1 \to C_0 \) and a partial function \( \circ : C_1 \times C_1 \to C_1 \) called **composition**.

The function \( g \circ f \) is defined if and only if \( \text{cod}(f) = \text{dom}(g) \) and when it is defined \( \text{dom}(g \circ f) = \text{dom}(f), \text{cod}(g \circ f) = \text{cod}(g) \).

For every \( X \in C_0 \) there is a unique morphism \( \text{id}_X \) which is an identity for composition.

Composition is associative.
Some categorical concepts

- The collection of objects can be a set: small category.
- The collection of morphisms between two objects can be a set: locally small category. We write $\text{Hom}(A, B)$ or $C(A, B)$: homset.
- A **functor** $\mathcal{F}$ relates two categories: it maps objects to objects and morphisms to morphisms and it preserves identities and composition.

$$
\begin{array}{ccc}
X & \xrightarrow{f} & Y \\
\downarrow{g} & \nearrow{g \circ f} & \downarrow{Z} \\
\mathcal{F}(X) & \xrightarrow{\mathcal{F}(f)} & \mathcal{F}(Y) & \xrightarrow{\mathcal{F}(g)} & \mathcal{F}(Z)
\end{array}
$$

$\mathcal{F}(g \circ f) = \mathcal{F}(g) \circ \mathcal{F}(f)$
A category may or may not have products.
This makes the concept of homset (or function space) *internal*; i.e. there are objects that behave like the homsets.
Terminal Objects

Definition
An object in a category is **terminal** if there is a unique morphism to it from every object.

Definition
An object in a category is **initial** if there is a unique morphism from it to every object.
A CCC has finite products,
a terminal object
and exponentials.
We want our domains to form a CCC.
Basic properties of domains

- Domains should capture the idea of *partial information*.
- This is expressed *qualitatively* through a domain.
- A domain should be a poset with a least element.
- A directed set $X \subseteq D$ satisfies: $\forall x, y \in X \exists z \in X$ with $x, y \leq z$. It represents a *consistent* collection of data. Every directed set should have a least upper bound ($\text{sup}$, $\lor$).
- Such posets are called **dcpos** for directed-complete posets.
- Henceforth, all domains will be dcpos; more conditions later.
- Functions between domains should be monotone.
- Functions between domains should preserve sups of directed sets: continuity.
We want some concept of “piece of information”.

We say that $b$ is an essential approximation of $y$ if whenever there is a directed set $X$ with $y \leq \bigvee X$ then for some $x \in X$ we have $b \leq x$; we write $b \ll y$.

Any limiting process that passes $y$ must pass $b$ at some finite stage.

Example: consider the domain consisting of subsets of the integers. Then an essential approximation of the set of positive even numbers is $\{2, 6, 8\}$ but the set of positive powers of two is an approximation but not an essential approximation.

We will write $\downarrow(x)$ for the set of essential approximations to $x$. 
We would like to have a collection of “tractable” elements that allow one to represent everything in the domain.

A **basis** $B$ for a domain $D$ is a (countable) family of elements such that for every $d \in D$ the set of elements $B_d = B \cap \downarrow(d)$ is directed and $\bigvee B_d = d$.

A domain with a (countable) basis is said to be (ω-)**continuous**.

We say that $e$ is **finite** (compact) if $e \ll e$.

Sometimes we do not have enough finite elements but we can often find enough essential approximations.

Example: $[0, 1]$ with the usual order has only one finite element but the rational form a nice countable basis.
Examples of continuous domains

- The set of all subsets of positive integers, ordered by inclusion. Take the *finite* subsets as the basis. These are actually *finite elements*; which partly explains the terminology.

- The set of all partial functions from a countable set to itself ordered by inclusion of graphs.

- The set of all subprobability distributions on a finite set, ordered pointwise.

- A countable basis is given by all the distributions that assign rational weights to each point.

- Continuous domains arise whenever one is dealing with real numbers: probabilistic systems, real-time systems, computing with real numbers.
One wants to relate the denotational semantics with the operational semantics; one needs to work with “syntactically representable elements” as a way of forging this connection.

It usually happens that this connection is mediated by finite elements.

A continuous domain in which all the basis elements are finite (not finite in number!) is called an algebraic domain.

For the traditional semantic applications algebraic domains are very important. For more recent applications to real-time, hybrid and probabilistic systems continuous domains are necessary.

Whence comes this name “algebraic”?

The collection of finitely generated subgroups in the lattice of subgroups of a given group forms an algebraic dcpo. Many examples in algebra come from finitely generated meaning “finite”.

Panangaden (McGill University) Tutorial on Semantics Part II LICS Toronto June 2011
What are general functions spaces?

If $D$ and $E$ are dcpos then we define $[D \rightarrow E]$ to be the poset of \textit{continuous} functions from $D$ to $E$ with the following order:

$$f \leq g \text{ iff } \forall d \in D, \ f(d) \leq_E g(d).$$

It is not hard to show that $[D \rightarrow E]$ is itself a dcpo.

We can define $D \times E$ as $\{(x, y) | x \in D, y \in E\}$ with the order $(x, y) \leq (x', y')$ iff $x \leq_D x'$ and $y \leq_E y'$.

If we define \textbf{Dcpo} to be the category with dcpos as objects and continuous functions as morphisms we get a cartesian closed category.
If we adopt $\omega$-algebraicity as a basic requirement for our domains we need to ensure that the function spaces are also $\omega$-algebraic.

However, we cannot take domains to be arbitrary $\omega$-algebraic dcpos.

There are three famous examples due to Gordon Plotkin of $\omega$-algebraic dcpos $D$ with $[D \to D]$ not $\omega$-algebraic.
Plotkin’s first example

This is not too bad, it is algebraic but not $\omega$-algebraic.
Plotkin’s second example
These last two are really terrible!
A pair of elements \( x, y \) in a dcpo are said to be **bounded** or **consistent** if there is some \( z \) such that \( x, y \leq z \).

**Definition**

A **Scott domain** is an \( \omega \)-algebraic dcpo such that every non-empty finite set of elements has a least upper bound.

They are also called bounded-complete dcpos or consistently-complete dcpos.
Easy to see that Plotkin’s examples are all ruled out.

Easy fact: if $e_1, e_2$ are compact and $e_1 \sqcup e_2$ exists, then it is also compact; hence, same is true for finite sets of compact elements.

If $D$ and $E$ are Scott domains and the finite elements are denoted \{d_i\} and \{e_j\} respectively, then the following are compact elements of the function space

\[
d_i \xrightarrow{e_j(x)} = \begin{cases} 
  e_j, & \text{if } d_i \leq x; \\
  \bot, & \text{otherwise.}
\end{cases}
\]

They are called step functions.

Do reasonable sups of these things always exist?
Sups of step functions

- When should $d_1 \rightharpoonup e_1$ and $d_2 \rightharpoonup e_2$ be consistent?
- When $d_1$ and $d_2$ are consistent then $e_1$ and $e_2$ should be consistent.
- In that case $e = e_1 \sqcup e_2$ exists, because of bounded completeness!
- Then we can define

$$\left(d_1 \rightharpoonup e_1 \sqcup d_2 \rightharpoonup e_2\right)(x) = \begin{cases} 
  e_1, & \text{if } d_1 \leq x \text{ but } d_2 \not\leq x; \\
  e_2, & \text{if } d_2 \leq x \text{ but } d_1 \not\leq x; \\
  e, & \text{if } d_1 \leq x \text{ and } d_2 \leq x; \\
  \bot, & \text{otherwise}. 
\end{cases}$$

- Now we can get a basis for the function space by taking sups of all bounded (consistent) finite collections of step functions.
- The category of Scott domains is cartesian closed.
Is this the best one can do?

- Gordon Plotkin defined a larger category – the SFP domains – which ruled out his three examples and showed that this gives a CCC of $\omega$-algebraic domains. He needed it for his work on powerdomains and nondeterministic computation.

- Mike Smyth showed that this is the largest CCC of $\omega$-algebraic domains.

- Carl Gunter showed that the Scott domains are the largest first-order axiomatizable CCC of $\omega$-algebraic domains.

- Achim Jung showed that there were exactly 4 maximal CCCs of algebraic domains.

- Why do we need more CCCs if Scott domains are good enough for PCF?

- We need them when we add new features – nondeterminism, probability – to the language and need to model them.
Basic domains for PCF

The “flat” domain of naturals: $\mathbb{N}_\bot$

The flat domain of booleans: $\mathbb{B}_\bot$
The ground types

\[ [Nat] = \mathbb{N}_\bot; \quad [Bool] = \mathcal{B}_\bot. \]

The higher types

\[ [[\sigma \times \tau]] = [[\sigma]] \times [[\tau]]; \quad [[\sigma \to \tau]] = [[[[\sigma]] \to [[\tau]]]. \]

The constants, pairing, projection, plus, equals and conditionals are interpreted the obvious way.

The \( \lambda \)-calculus part is interpreted in the manner we have already indicated. We need to show various things are continuous.

It only remains to explain \( \text{fix} \).
Given $D$ a Scott domain (any dcpo with $\bot$ will do); define $\text{fix}_D : [D \rightarrow D] \rightarrow D$ by

$$\text{fix}_D(f) = \bigvee \{\bot, f(\text{bot}), \ldots, f^{(n)}(\bot), \ldots\}.$$ 

This is itself a continuous function.

The family $\text{fix}_D$ is the *unique* family satisfying the following uniformity condition. If $h$ is strict ($h(\bot) = \bot$) and the diagram

$$\begin{array}{ccc}
D & \xrightarrow{f} & D \\
\downarrow{h} & & \uparrow{h} \\
E & \xrightarrow{g} & e
\end{array}$$

commutes, then $h(\text{fix}_D(f)) = \text{fix}_E(g)$.

$$\langle \text{fix}(M) \rangle = \text{fix}(\langle M \rangle).$$
Subject reduction and soundness

**Theorem**

If $\Gamma \vdash M : \tau$ is a valid typing judgment and $M \xrightarrow{*} N$ then $\Gamma \vdash N : \tau$ is a valid typing judgment.

**Theorem**

If $\Gamma \vdash M : \tau$ and $M \xrightarrow{*} N$ then $\llbracket M \rrbracket = \llbracket N \rrbracket$. 
A **context** in PCF is essentially a term with a “hole” in it into which another term of the appropriate type can be plugged in.

For example $\lambda x.\langle 2, x[\cdot]\rangle$. If we put a term of the right type in the hole, we will get a PCF term.

A semantics is *compositional* if $[M] = [N]$ implies that for all contexts $C[\cdot]$ (of the right type) $[C[M]] = [C[N]]$.

The denotational semantics of PCF based on domains (the standard model) is compositional.

If $C[\cdot]$ is such that $C[M]$ is of ground type, we say $C$ is a ground context.
Observations

- We cannot test terms of all types for equality, only ground types.
- We can observe a ground term by seeing to what value it reduces.
- We write $M \Downarrow m$ if the term $M : Nat$ eventually reduces to the number $m$.
- What can we observe about higher type terms?
- We say $M, N$ are **observationally equivalent** if for all ground contexts $C[\cdot]$ for $M$ and $N$, $C[M] \Downarrow v$ if and only if $C[N] \Downarrow v$; we write $M \equiv_{obs} N$.
- We write $M \Downarrow \perp$ to mean $\forall v. \neg(M \Downarrow v)$.
- We would like our denotational semantics to be a good guide to observational equivalence.
**Adequacy**

**Definition**

We say a semantics is **adequate** if

\[
[M] = [N] \Rightarrow M \equiv_{\text{obs}} N.
\]

This is equivalent to

**Theorem**

\[
[M] = [v] \iff M \Downarrow v.
\]

**Proof sketch**

Assume \([M] = [N]\) and the proposition holds. Let \(C[\cdot]\) be a ground context and \(v\) a value such that \(C[M] \Downarrow v\). Thus \([C[M]] = [C[v]] = [C[N]]\), where we have used compositionality. Thus, \(C[N] \Downarrow v\).
Theorem

The denotational semantics of PCF is adequate.
How can we reason about higher type languages?

- We use both the term structure and the type structure.
- Terms of simple structure – like variables – can have arbitrarily complicated types.
- Therefore the induction arguments are not just nicely nested.
- Furthermore, we have to deal with substitutions into open terms.
- The main technique uses logical relations invented by Tait in 1967 to prove strong normalization of simply-typed $\lambda$-calculus.
- We will illustrate logical relations with the proof of adequacy.
- For simplicity, I will forget about products.
The computability predicate

- If $M : Nat$ is closed it is said to be **computable** if $[M] = [v]$ implies $M \Downarrow v$.
- If $M : \tau \rightarrow \tau'$ is closed it is computable if, for every closed computable term $N : \tau$, $MN : \tau'$ is computable.
- If $M$ has free variables $\{x_1, \ldots, x_k\}$ then it is computable if for every substitution $M[N_1/x_1, \ldots, N_k/x_k]$ of closed computable terms for the free variables we get a computable term.
- We call such a substitution computable.
- We write $\sigma$ for a substitution and $\sigma[M]$ for the term resulting from the substitution.
We claim every PCF term is computable. Induction on structure of terms and types.

\( M = x : \tau \); a computable substitution will certainly produce a computable term.

Cases where \( M \) is a conditional or \textit{plus} are easy structural induction cases.
\[ M = \lambda x. Q : \tau_1 \rightarrow \tau_2. \] Let \( \sigma \) be a computable substitution and let \( \vec{T} \) be a sequence of closed computable terms such that \( \sigma[M] \vec{T} \) is of ground type and that \( \llbracket \sigma[M] \vec{T} \rrbracket = \llbracket v \rrbracket \).

\[ \sigma[M] \vec{T} = \sigma[\lambda x. Q] T_1 T_2 \ldots T_k = (\lambda x. \sigma[Q]) T_1 T_2 \ldots T_k \]

\[ = (\sigma[Q][T_1/x_1]) T_2 \ldots T_k. \]

Now the term \( (\sigma[Q][T_1/x_1]) = Q[T_1/x_1, S_1/y_1, S_2/y_2, \ldots] \) is just another substitution instance of \( Q \) by a computable substitution \( \sigma' \). Hence, by the induction hypothesis it is computable.

Thus \( \llbracket \sigma[M] \vec{T} \rrbracket = \llbracket \sigma'[Q] T_2 \ldots T_k \rrbracket = \llbracket v \rrbracket \) implies that \( \sigma'[Q] T_2 \ldots T_k \Downarrow v \).

Hence \( \sigma[M] \vec{T} \Downarrow v \).

One can prove the application case with similar arguments.
Here we need another theorem: approximation.

Imagine the recursion unwound to some depth and then wherever `fix` occurs we replace it with `⊥`.

The collection of partial unwindings are the *syntactic approximants*.

We can show that the denotational semantics of the syntactic approximants give a directed set with least upper bound the meaning of the original term.

We can show that if any of the approximants applied to closed computable terms converges to `ν` then so does the original term.

We prove by induction on the depth of the unwinding that the unwindings are computable.

Putting all this together we can complete the argument.
A perfect match?

- We would like our denotational semantics to be a perfect match with observational equivalence.

\[
[M] = [N] \iff M \equiv_{obs} N.
\]

- Unfortunately, it is not!

- Consider the function “parallel or” with the following table

<table>
<thead>
<tr>
<th>por</th>
<th>⊥</th>
<th>tt</th>
<th>ff</th>
</tr>
</thead>
<tbody>
<tr>
<td>⊥</td>
<td>⊥</td>
<td>tt</td>
<td>⊥</td>
</tr>
<tr>
<td>tt</td>
<td>tt</td>
<td>tt</td>
<td>tt</td>
</tr>
<tr>
<td>ff</td>
<td>⊥</td>
<td>tt</td>
<td>ff</td>
</tr>
</tbody>
</table>

- This function cannot be defined in PCF; proved by Plotkin in 1977.

- This function is “listening in parallel” to two inputs and will use whichever one converges first.

- However, the operational semantics of PCF is sequential.
The problem with parallel or

Call this term $T$.

Call this term $F$.

Consider the terms $(\lambda f. T)\, \text{por} = \text{tt}$ and $(\lambda f. F)\, \text{por} = \text{ff}$. So it is definitely the case that $[T] \neq [F]$. However, no PCF definable term will ever see the difference.
What can be done about this?

- Add parallel or to the language or some other parallel construct.
- Various extended languages were shown to have fully abstract domain models.
- Key step in proving full abstraction: all the finite elements are definable.
- Construct a fully abstract model from the syntax: Milner 1977.
- All fully abstract models are isomorphic, so the question is one of presenting a fully abstract model in an insightful way. The domain model gives insight into the nature of computation that is not just mimicking the operational semantics.
- Try to characterize sequential computation mathematically.
Berry introduced a new restriction and a new order on functions – the stable order – and introduced stronger finiteness conditions.

In Scott domains a finite (i.e. compact) element can be above infinitely many elements! This does not happen in stable domain theory.

PCF can be given an adequate semantics with stable domains.

Parallel or does not appear in stable domains.

Unfortunately, other more complicated examples can be given – discovered by Berry himself – that show that full abstraction fails.
Berry and Curien started the study of sequential *algorithms* on concrete data structures.

Girard invented linear logic in the mid 1980s and this made a huge impact on the semantics community by making resource sensitivity an integral part of logic and proof theory.

Abramsky and Jagadeesan developed full completeness results for linear logic based on dialogue games.

Abramsky, Jagadeesan and Malacaria and simultaneously and independently Hyland and Ong and also independently Nickau developed fully abstract games models for PCF.

O’Hearn and Riecke gave domain theoretic fully abstract models but they were also based on intensional ideas.
Loader’s result

- Ralph Loader showed that observational equivalence of even finitary PCF is undecidable.
- This means that no fully abstract model can be effectively presented.
The basic idea is to model data types as dialogue games and programs as strategies: there is no notion of winning or losing.

Remarkably different programming paradigms appear as different restrictions on allowed strategies.

Two important restrictions needed for modelling PCF are called *innocence* and *bracketing*. Loosening these restrictions yields fully abstract models of extensions of PCF!

\[
\begin{align*}
 PCF + \text{control} & \quad \text{---} \quad PCF + \text{control} + \text{state} \\
 PCF & \quad \text{---} \quad PCF + \text{state} \\
 G_i & \quad \text{---} \quad G \\
 G_{ib} & \quad \text{---} \quad G_b
\end{align*}
\]