Domain theory and denotational semantics of functional programming

Martín Escardó

School of Computer Science, Birmingham University

MGS 2007, Nottingham, version of April 17, 2007

What is denotational semantics?

Very abstract answer:

types are objects of a category,

programs are morphisms of this category.

Concrete examples:

1. category of sets (when it works).
2. categories of domains.
3. realizability toposes.
4. categories of games.

Why denotational semantics?

1. Mathematical models aid program verification.
2. They guide the construction of programming languages.
3. Sometimes they allow one to discover new algorithms.

   Games. (Un)decidability of observational equivalence.

   Domains. (Un)decidability of function equality.
4. $\langle$Fill in your favourite answer here.$\rangle$

Why various kinds of denotational semantics?

Different mathematical aspects are addressed/emphasized:

Domains. Finite approximation of infinite objects.

Realizability. Constructive logic and computability.

Games. Interaction, sequentiality.

Operational versus denotational semantics

Operational semantics tells you how your programs are run.

Denotational semantics tells you what your programs compute.

Operational versus denotational semantics

Definition, to be made precise:

Adequacy. For observable types, the two agree.

Full abstraction. Operational and semantic equivalence agree.

Universality. All computable elements are programmable.

Universality $\implies$ full abstraction $\implies$ adequacy.

The converses fail.

One would like

Types are sets.

Programs are functions.

Life would be much simpler if this were always possible

(but perhaps less exciting).

(Synthetic domain theory rescues this wish.)

When do plain sets work?

E.g.

1. Gödel's system T: typed λ-calculus with primitive recursion.
2. Martin-Löf type theory.
3. Typed λ-calculus with (co)inductive types.

(But, for all I know, full abstraction for these may fail.)

When plain sets don't work?

E.g.

1. Function recursion.
2. Type recursion, e.g. $D \cong (D \to \texttt{Bool})$.
3. Certain total functionals.
   a. Fan functional.
   b. Bar recursion.

Dana Scott (1969, 1972) proposed to use domains.

Ershov independently (motivation higher-type computability).

Precursors of domain theory

Kleene's recursion theorem.

Can find f such that $f = F(f)$.

Myhill–Shepherdson theorem.

Computable functions $(\mathbb{N} \to \mathbb{N}) \to (\mathbb{N} \to \mathbb{N})$ are continuous.

Rice–Shapiro theorem.

Semidecidable subsets of $\mathcal{P}\,\mathbb{N}$ are Scott open.

Platek's approach to Kleene–Kreisel higher-type computability.

E.g. which $((\mathbb{N} \to \mathbb{N}) \to \mathbb{N}) \to \mathbb{N}$ are computable?

What is a domain?

A set, with concrete, finite elements,

together with ideal, infinite elements such that

ideal elements are uniquely determined by

their concrete approximations.

This can be made precise in a number of ways.

Example

Consider programs (in any suitable language) that output bits either for ever, or else until they get stuck (in an infinite loop).

E.g.

```
(a) while (true) {  |  (b) while (true) {   |  (c) print 0;
        print 0;    |      }                 |      print 1
        print 1;    |                        |      while (true) {
    }               |                        |      }
```

Domain-theoretic denotations:

(a) $(01)^\omega$, (b) ϵ, (b) 01.

Example continued

See whiteboard for a picture of the Cantor tree.

The runs of such programs correspond to paths in the Cantor tree.

E.g. (1) corresponds to the path

$$\underbrace{\epsilon}_{\alpha_0}, \underbrace{0}_{\alpha_1}, \underbrace{01}_{\alpha_2}, \underbrace{010}_{\alpha_3}, \underbrace{0101}_{\alpha_4}, \underbrace{01010}_{\alpha_5}, \ldots .$$

Concrete versus ideal

Using notation to be made precise later:

$$\underbrace{(01)^\omega}_{\text{what you imagine}} = \underbrace{\bigsqcup_{i \geq 0}}_{\text{glue together}} \underbrace{\alpha_i}_{\text{what you see}}$$

Terminologies for this operation: join, supremum, least upper bound.

Making this example precise

The set is $D = \underbrace{\{0,1\}^*}_{\text{nodes of the tree}} \cup \underbrace{\{0,1\}^\omega}_{\text{infinite paths}}$.

For $\alpha, \beta \in D$, write

$$\alpha \sqsubseteq \beta$$

to mean that α is a prefix of β.

Making this example precise

This is a *partial order*:

Reflexivity. $\alpha \sqsubseteq \alpha$.

Transitivity. $\alpha \sqsubseteq \beta \sqsubseteq \gamma \implies \alpha \sqsubseteq \gamma$.

Anti-symmetry. $\alpha \sqsubseteq \beta \;\&\; \beta \sqsubseteq \alpha \implies \alpha = \beta$.

Making this example precise

For any path

$$\alpha_0 \sqsubseteq \alpha_1 \sqsubseteq \alpha_2 \sqsubseteq \cdots \sqsubseteq \alpha_i \sqsubseteq \cdots$$

there is $\beta \in D$ such that

1. $\alpha_i \sqsubseteq \beta$ for all i.
2. If, for another $\beta' \in D$,

   1'. $\alpha_i \sqsubseteq \beta'$ for all i,

   then $\beta \sqsubseteq \beta'$.

Making this example precise

For any path

$$\alpha_0 \sqsubseteq \alpha_1 \sqsubseteq \alpha_2 \sqsubseteq \cdots \sqsubseteq \alpha_i \sqsubseteq \cdots \sqsubseteq \beta \sqsubseteq \beta'$$

there is $\beta \in D$ such that

1. $\alpha_i \sqsubseteq \beta$ for all i. (β is an upper bound of the sequence α_i.)
2. If, for another $\beta' \in D$,

   1'. $\alpha_i \sqsubseteq \beta'$ for all i, (β' is an other upper bound.)

   then $\beta \sqsubseteq \beta'$. (So β is the *least* upper bound.)

Making this example precise

For any path

$$\alpha_0 \sqsubseteq \alpha_1 \sqsubseteq \alpha_2 \sqsubseteq \cdots \sqsubseteq \alpha_i \sqsubseteq \cdots$$

there is $\beta \in D$ such that

1. β is an upper bound of the sequence α_i.
2. β is below any other upper bound β'.

This β is unique. Why?

We write $\beta = \bigsqcup_i \alpha_i$.

Summary

D is an ω-complete poset.

Sometimes domain is taken to mean ω-complete poset with a least element $\perp$.

In this example, $\perp$ is the empty sequence ϵ.

Another example: lazy lists in Haskell

For any type σ, there is a type $[\sigma]$ of finite and infinite lists.

It has the following elements:

1. The bottom sequence "$[$".

1'. More generally, "$[x_1, x_2, \ldots, x_n$" with $x_i \in d$.

2. Their terminated versions "$[x_1, x_2, \ldots, x_n]$".

3. Infinite sequences "$[x_1, x_2, \ldots, x_n, \ldots$"

and nothing else

Order: To be added. Board for the moment.

Simpler examples

The type `Bool` in Haskell. Has three elements: `True`, `False`, $\perp$.

Order: `True` and `False` are maximal, $\perp$ is minimal.

The type Integer in Haskell. Has all the integers plus $\perp$.

Order: Integers are maximal, $\perp$ is minimal.

All paths are trivial. The orders are ω-complete.

Semantics of programs and of function types

If two types σ and τ are interpreted as domains D and E, then the function type $(\sigma \to \tau)$ is interpreted as a domain $(D \to E)$.

Question. What $(D \to E)$ should/can be?

1. All functions?

2. The computable functions?

Answer. Something in between.

3. The continuous functions.

Why? Answer postponed until we see some examples.

Continuity — computational motivation

A function $f\colon D \to E$ is continuous if finite parts of $f(x)$ depend only on finite parts of x.

Continuity — a special case first

Consider $D = \{0,1\}^* \cup \{0,1\}^\omega$ ordered by prefix.

Definition. $f\colon D \to D$ is monotone if

$$\alpha \sqsubseteq \beta \implies f(\alpha) \sqsubseteq f(\beta).$$

If you supply more input, you get more output.

Definition. f is of finite character if

whenever $\beta \sqsubseteq f(\alpha)$,

there is $\alpha' \sqsubseteq \alpha$ finite such that already $\beta \sqsubseteq f(\alpha')$.

Continuity — a special case first

Theorem. For $f : D \to D$ monotone, TFAE:

1. f is of finite character.
2. For every path

$$\alpha_0 \sqsubseteq \alpha_1 \sqsubseteq \alpha_2 \sqsubseteq \cdots \sqsubseteq \alpha_i \sqsubseteq \cdots$$

   with

$$\alpha_\infty = \bigsqcup_i \alpha_i$$

   one has

$$f(\alpha_\infty) = \bigsqcup_i f(\alpha_i).$$

Continuity — a special case first

Theorem. For $f : D \to D$ monotone, TFAE:

1. f is of finite character.
2. For every path

   $$\alpha_0 \sqsubseteq \alpha_1 \sqsubseteq \alpha_2 \sqsubseteq \cdots \sqsubseteq \alpha_i \sqsubseteq \cdots$$

   one has

   $$f(\bigsqcup_i \alpha_i) = \bigsqcup_i f(\alpha_i).$$

Proof. Exercise. Hint. First show that α' is finite iff whenever $\alpha' \sqsubseteq \bigsqcup_i \alpha_i$ for α_i ascending, there is i such that already $\alpha' \sqsubseteq \alpha_i$. $\square$

Continuous function

We make the previous theorem into a definition:

Definition. A function of domains is continuous iff

1. it is monotone, and
2. it preserves joins of ascending chains.

Interpretation of function types

If two types σ and τ are interpreted as domains D and E, then the function type $(\sigma \to \tau)$ is interpreted as the domain $(D \to E)$.

Definition. $(D \to E) =$ set of continuous functions $D \to E$ ordered pointwise.

This means: $f \sqsubseteq g$ iff $f(x) \sqsubseteq g(x)$ for all $x \in D$.

Theorem. If D and E are domains, then so is $(D \to E)$.

Some examples.

Use board.

Products

If two types σ and τ are interpreted as domains D and E, then the product type $(\sigma \times \tau)$ is interpreted as the domain $(D \times E)$.

Definition. $(D \times E) =$ cartesian product ordered coordinatewise.

This means: $(x, y) \sqsubseteq (x', y')$ iff $x \sqsubseteq x'$ and $y \sqsubseteq y'$.

Theorem. If D and E are domains, then so is $(D \times E)$.

Note on products

Haskell requires a slightly different interpretation of the product.

Finite elements in general

Let D be a poset with joins of ascending sequences.

Definition. $b \in D$ is called finite if whenever $b \sqsubseteq \bigsqcup_i x_i$ for some ascending sequence x_i, there there is x_i such that already $b \sqsubseteq x_i$.

Definition. D is called ω-algebraic if every element of D is the join of an ascending sequence of finite elements.

Closure properties of algebraic domains

Characterization of continuity

Let D and E be ω-algebraic posets.

Theorem. For $f : D \to E$ monotone, TFAE:

1. f is of finite character.
2. For every path

   $$\alpha_0 \sqsubseteq \alpha_1 \sqsubseteq \alpha_2 \sqsubseteq \cdots \sqsubseteq \alpha_i \sqsubseteq \cdots$$

   one has

   $$f(\bigsqcup_i \alpha_i) = \bigsqcup_i f(\alpha_i).$$

Slides incomplete from now on.

Interpretation of recursion

An example of application of all we have seen so far.

Usually one meets the recursive definition of factorial.

This is rather boring.

All consider a surprising program, due to Ulrich Berger (1990).

Berger's functional — preliminaries

```
type Z = Integer
type Baire = [Z]
```

The specification of Berger's functional, to be given below, talks about infinite sequences of bits.

Let `Cantor` denote this subset of `Baire`.

We say that $p \in (\texttt{Baire} \to \texttt{Bool})$ is defined on `Cantor` if $p(\alpha) \neq \bot$ for all $\alpha \in \texttt{Cantor}$.

Specification of Berger's functional

```
berger :: (Baire -> Bool) -> Baire
```

For every $p \in (\texttt{Baire} \to \texttt{Bool})$, if p is defined on `Cantor` then

1. $\texttt{berger}(p) \in \texttt{Cantor}$, and
2. $p(\texttt{berger}(p)) = \texttt{True}$ iff $p(\alpha) = \texttt{True}$ for some

$\alpha \in \texttt{Cantor}$.

Application: exhaustive search over infinite sets

```
forsomeC, foreveryC :: (Baire -> Bool) -> Bool
forsomeC p = p(berger p)
foreveryC p = not(foreveryC(\a -> not(p a)))

equalC :: (Baire -> Z) -> (Baire -> Z) -> Bool
equalC f g = foreveryC(\a -> f a == g a)
```

Theorem. For f and g defined on `Cantor`, `equalC f g = True` iff f, g agree on `Cantor`.

Berger's functional

```
berger :: (Baire  -> Bool) -> Baire
berger p = if p(0 : berger(\a -> p(0 : a)))
           then 0 : berger(\a -> p(0 : a))
           else 1 : berger(\a -> p(1 : a))
```

Theorem. This satisfies the above specification.

Proof. Postponed.

Intuition. See board discussion.