Domain theory and denotational semantics of functional programming

Martín Escardó

School of Computer Science, Birmingham University

MGS 2007, Nottingham, version of April 20, 2007 17:26

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:

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

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`, $\bot$.

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

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

Order: Integers are maximal, $\bot$ 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 $\beta \sqsubseteq f(\alpha)$ for β finite,

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 sequences.

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 Haskell products and function types

Interpreted as $(D \times E)_\perp$ and $(D \to E)_\perp$.

E.g. the following two functions are *different* (using `seq`):

```
f,g :: a -> b
f = f
g x = g x
```

Then

```
seq f True
```

diverges, but

```
seq g True
```

evaluates to `True`.

Note on Haskell products and function types

So,

1. Haskell products are not categorical, and
2. Haskell is not cartesian or monoidal closed.

In particular, $\mathrm{curry}(\mathrm{uncurry}(f)) \neq f$ in general.

Interaction between products and function spaces

Theorem. Continuous functions of domains form a cartesian closed category.

This amounts to:

1. The evaluation function $\mathrm{eval}\colon (D \to E) \times D \to E$ defined by $\mathrm{eval}(f, x) = f(x)$ is continuous.
2. If $f\colon C \times D \to E$ is continuous, then so is the function $\bar{f}\colon C \to (D \to E)$ defined by $\bar{f}(x) = \lambda y.f(x, y)$.

Consequence of cartesian closedness

Theorem. If a function is λ-defined from continuous functions, then it is itself continuous.

Application. Consider a functional programming language based on the simply typed λ-calculus, with some primitive functions that are continuous.

Then all functions definable in this language are automatically continuous.

E.g. Haskell, PCF.

Interpretation of recursion — introduction

A recursive definition of a function $f\colon D \to E$ can always be written in the form

$$f(x) = F(f, x).$$

for a suitable continuous $F\colon (D \to E) \times D \to E$.

Equivalently,

$$f = \lambda x.F(f, x)$$

or, with $G = \bar{F}$,

$$f = G(f).$$

Interpretation of recursion — example

```
fact :: Integer -> Integer
fact n = if n == 0 then 1 else n * fact(n-1)
```

Take the opportunity to give the semantics of primitive functions.

Now define

```
G :: (Integer -> Integer) -> (Integer -> Integer)
G f = \n -> if n == 0 then 1 else n * f(n-1)
```

Then the above definition is equivalent to

```
fact = G(fact)
```

Interpretation of recursion — questions

Conversely, given any $G\colon (D \to E) \to (D \to E)$, one can recursively define $f\colon D \to E$ by

$$f = G(f).$$

Questions.

1. Is there any continuous function f such that $f = G(f)$.

   If not, we are in trouble.

2. Is there more than one?

3. If so, which one do we choose?

Interpretation of recursion — another example

One can recursively define elements too.

```
naturals :: [Integer]
naturals = 0 : map (1+) naturals
```

Moreover, for any $g\colon C \to C$ one can recursively define $x \in C$ by

$$x = g(x).$$

Interpretation of recursion — is it possible?

Summary:

1. For any continuous $G\colon (D \to E) \to (D \to E)$

   there must be $f \in (D \to E)$ such that $f = G(f)$.

2. For any continuous $g\colon C \to C$

   there must be $x \in C$ with $x = g(x)$.

But

3. With $C = (D \to E)$ and $g = G$ and $x = f$,

   requirement (2) is a particular case of (1).

Interpretation of recursion — theorem

If D is an ω-complete poset with a least element $\perp$, then any continuous $f\colon D \to D$ has a least fixed point.

That is:

1. There is $x \in D$ such that $x = f(x)$.
2. If $y = f(y)$ then $x \sqsubseteq y$.

Proof sketch. Take $x = \bigsqcup_n f^n(\perp)$ and show that this choice works.

Interpretation of recursion — adequacy

Question. Why is it sensible to interpret recursive definitions as least fixed points?

Answer. This is justified by a theorem called *computational adequacy*, to be given later.

Roughly, this says that, with this interpretation, the denotational and the operational semantics agree at observable types.

Continuity of the fixed-point operator

Theorem. The function $\mathrm{fix}\colon (D \to D) \to D$ that sends a continuous function to its least fixed point is continuous.

Proof. Trick.

Let $E = ((D \to D) \to D)$ and define $\Phi\colon E \to E$ by $\Phi(F) = \lambda f. f(F f)$.

Then Φ has a least fixed point F, which is a continuous function.

But $F = \bigsqcup_n \Phi^n(\bot) = \bigsqcup_n \lambda f. f^n(\bot) = \lambda f. \bigsqcup_n f^n(\bot)$.

That is, $F = \mathrm{fix}$, and so fix is continuous. Q.E.D.

The functional language PCF

Scott (1969), Plotkin (1977). Streamed-down version of Haskell.

Simply typed λ-calculus with base types for natural numbers and booleans, and with recursion. Call-by-name evaluation.

Primitive operations: basic arithmetic and comparisons, conditional, fixed-point functionals.

It comes with a program logic, called LCF.

The domain-theoretic interpretation of PCF validates the axioms of the logic. We'll come back to this.

Sample application

I'll consider a surprising program, due to Ulrich Berger (1990).

It performs a seemingly impossible task:

Given a predicate p defined on infinite sequences of bits,

it checks whether or not p holds for all infinite sequences of bits.

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

the program `berger` finds $\alpha \in \texttt{Cantor}$ such that $p(\alpha)$ holds, if such an α exists, always returning an element of `Cantor`.

Specification of Berger's functional — bis

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

Corollary: exhaustive search over infinite sets

```
forsomeC, foreveryC :: (Baire -> Bool) -> Bool
forsomeC p = p(berger p)
foreveryC p = not(forsomeC(\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` if f, g agree on `Cantor`, and `equalC`$(f)(g) =$ `False` otherwise.

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. See board discussion.

Back to finite elements

The proof of the above correctness result relies on finite elements.

Want notion of finite element for other domains, e.g. function spaces.

This leads to interesting and useful proof principles.

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 (1) it has countably many finite elements and (2) every element of D is the join of an ascending sequence of finite elements.

Examples

For $D = \{0,1\}^* \cup \{0,1\}^\omega$ ordered by prefix, the finite elements in this abstract sense are the finite elements in the concrete sense.

Hence this domain is ω-algebraic.

Examples

Let $\mathcal{N}$ be $\mathbb{N} \cup \{\bot\}$ ordered by $x \sqsubseteq y$ iff $x = \bot$ or $x = y$.

(Bottom is minimal, natural numbers are maximal.)

This is called the flat domain of natural numbers.

This is trivially algebraic: all elements are finite.

Examples

Let $D = (\mathcal{N} \to \mathcal{N})$.

Exercise. The elements of D are precisely the monotone functions. (Continuity trivializes.)

The following elements of D are finite:

1. The constant functions.
2. The functions $f \in D$ such that the set $\{n \in \mathbb{N} \mid f(n) \neq \bot\}$ has finite cardinality.

Deduce that this domain is algebraic.

Functions of finite character

Let D and E be algebraic.

Definition. $f \in (D \rightarrow E)$ is of finite character if

whenever $c \sqsubseteq f(x)$ for $c \in E$ finite and any $x \in D$,

there is $b \sqsubseteq x$ finite such that already $c \sqsubseteq f(b)$.

Characterization of continuity

Let D and E be algebraic.

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

1. f is of finite character.
2. f is continuous.

Algebraic producs

The product of two algebraic domains is algebraic.

Exercise. Show that $(x, y) \in D \times E$ is finite iff x and y are finite.

Algebraic function spaces

It is not the case that if D and E are algebraic then so is $(D \to E)$.

However, under additional assumptions on D and E, the conclusion holds.

Definition. A Scott domain is a poset that has

1. joins of ascending sequences,
2. a least element,
3. joins of upper bounded finite sets.

Examples. All domains we have seen so far are Scott domains.

Scott domains form a cartesian closed category

It is enough to show that if D and E are Scott domains then so are $(D \times E)$ and $(D \to E)$.

Exercise. Do the $D \times E$ case yourself.

Function spaces of Scott domains

I may add a slide here. If not I'll use the board.

The language PCF (Streicher's book version)

$$\frac{}{\Gamma, x : \sigma, \Delta \vdash x : \sigma} \qquad \frac{\Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash \lambda x.M : \sigma \to \tau} \qquad \frac{\Gamma \vdash M : \sigma \to \tau \quad \Gamma \vdash N : \sigma}{\Gamma \vdash M N : \tau}$$

$$\frac{}{\Gamma \vdash \mathtt{zero} : \mathtt{nat}} \qquad \frac{\Gamma \vdash M : \mathtt{nat}}{\Gamma \vdash \mathtt{succ}\, M : \mathtt{nat}} \qquad \frac{\Gamma \vdash M : \mathtt{nat}}{\Gamma \vdash \mathtt{pred}\, M : \mathtt{nat}}$$

$$\frac{\Gamma \vdash L : \mathtt{nat} \quad \Gamma \vdash M : \mathtt{nat} \quad \Gamma \vdash N : \mathtt{nat}}{\Gamma \vdash \mathtt{if}\, L\, \mathtt{then}\, M\, \mathtt{else}\, N : \mathtt{nat}} \qquad \frac{\Gamma \vdash M : \sigma \to \sigma}{\Gamma \vdash \mathtt{Y} M : \sigma}$$

Big-step style operational semantics

$$\frac{}{x \Downarrow x} \qquad \frac{}{\lambda x.M \Downarrow \lambda x.M} \qquad \frac{M \Downarrow \lambda x.L \qquad L[N/x] \Downarrow V}{M\,N \Downarrow V}$$

$$\frac{M\,(\mathtt{Y}M) \Downarrow V}{\mathtt{Y}M \Downarrow V}$$

Big-step style operational semantics

For $n \in \mathbb{N}$, write $\underline{n} = \texttt{succ}^{\,n}(\texttt{zero})$.

$$\frac{}{\underline{0} \Downarrow \underline{0}} \qquad \frac{M \Downarrow \underline{n}}{\texttt{succ}\, M \Downarrow \underline{n+1}} \qquad \frac{M \Downarrow \underline{0}}{\texttt{pred}\, M \Downarrow \underline{0}} \qquad \frac{M \Downarrow \underline{n+1}}{\texttt{pred}\, M \Downarrow \underline{n}}$$

$$\frac{L \Downarrow \underline{0} \quad M \Downarrow V}{\texttt{if}\, L\, \texttt{then}\, M\, \texttt{else}\, N \Downarrow V} \qquad \frac{L \Downarrow \underline{n+1} \quad N \Downarrow V}{\texttt{if}\, L\, \texttt{then}\, M\, \texttt{else}\, N \Downarrow V}$$

The Scott model of PCF

This denotational semantics associates

to each type σ,

a domain $D_\sigma = [\![\sigma]\!]$, and

to each term $x_1 : \sigma_1, \ldots, x_n : \sigma_n \vdash M : \tau$,

a continuous function $[\![M]\!] \colon [\![\sigma_1]\!] \times \cdots \times [\![\sigma_n]\!] \to [\![\tau]\!]$.

Interpretation of types

By induction on types:

$$D_{\mathtt{nat}} = \mathbb{N}_{\perp} \qquad\qquad D_{\sigma\to\tau} = (D_\sigma \to D_\tau)$$

Interpretation of terms

On the above domains, define

$$\bot - 1 = \bot, \qquad 0 - 1 = 0, \qquad \bot + 1 = \bot,$$

$$\mathrm{if}\,\bot\,\mathrm{then}\,x\,\mathrm{else}\,y = \bot,$$

$$\mathrm{if}\,0\,\mathrm{then}\,x\,\mathrm{else}\,y = x,$$

$\mathrm{if}\,n\,\mathrm{then}\,x\,\mathrm{else}\,y = y$ for $n \neq \bot$ positive.

This gives continuous functions

$$(\text{---} + 1) : \mathbb{N}_\bot \to \mathbb{N}_\bot \qquad (\text{---} - 1) : \mathbb{N}_\bot \to \mathbb{N}_\bot$$

$$(\mathrm{if}\,\text{---}\,\mathrm{then}\,\text{---}\,\mathrm{else}\,\text{---}) : \mathbb{N}_\bot \times \mathbb{N}_\bot \times \mathbb{N}_\bot \to \mathbb{N}_\bot$$

Interpretation of terms

$$[\![\Gamma \vdash \mathtt{zero}]\!](\vec{d}) = 0$$

$$[\![\Gamma \vdash \mathtt{succ}\, M]\!](\vec{d}) = [\![\Gamma \vdash M]\!](\vec{d}) + 1$$

$$[\![\Gamma \vdash \mathtt{pred}\, M]\!](\vec{d}) = [\![\Gamma \vdash M]\!](\vec{d}) - 1$$

$$[\![\Gamma \vdash \mathtt{if}\, L\, \mathtt{then}\, M\, \mathtt{else}\, N]\!](\vec{d}) = \\ \text{if } [\![\Gamma \vdash L]\!](\vec{d}) \text{ then } [\![\Gamma \vdash M]\!](\vec{d}) \text{ else } [\![\Gamma \vdash N]\!](\vec{d})$$

Interpretation of terms

$$[\![x_1 : \sigma_1, \ldots, x_n : \sigma_n \vdash x_i]\!](d_1, \ldots, d_n) = d_i$$

$$[\![\Gamma \vdash \lambda x.M : \sigma \to \tau]\!] = \overline{[\![\Gamma, x : \sigma \vdash M : \tau]\!]}$$

$$[\![\Gamma \vdash M\,N]\!](\vec{d}) = [\![\Gamma \vdash M]\!](\vec{d})\left([\![\Gamma \vdash N]\!](\vec{d})\right)$$

$$[\![\Gamma \vdash \mathtt{Y} M]\!](\vec{d}) = \mathrm{fix}\left([\![\Gamma \vdash M]\!](\vec{d})\right)$$

Computational adequacy

Theorem. For every term M of ground type with no free variables, and every $n \in \mathbb{N}$,

$$[\![M]\!] = n \iff M \Downarrow \underline{n}.$$

(Hence if $[\![M]\!] = \bot$ then there is no n such that $M \Downarrow n$.)

Proof. See Chapter 4 of Streicher's book.

The logic LCF

The following principles are validated by the Scott model:

1. $M \sqsubseteq_{\sigma\to\tau} M' \wedge N \sqsubseteq_{\sigma} N' \implies MN \sqsubseteq_{\tau} M'N'$

1'. $M =_{\sigma\to\tau} M' \wedge N =_{\sigma} N' \implies MN =_{\tau} M'N'$

2. $\lambda x.M \sqsubseteq_{\sigma\to\tau} \lambda x.M' \implies \forall x:\sigma.M \sqsubseteq M'$.

2'. $\lambda x.M =_{\sigma\to\tau} \lambda x.M' \implies \forall x:\sigma.M = M'$.

3. $(\lambda x.M)N =_{\tau} M[N/x]$

4. $\lambda x:\sigma.Mx = M$ provided x is not free in M.

The logic LCF — recursion principles

5. $\mathtt{Y}M = M(\mathtt{Y}M)$.

6. $\forall x : \sigma . Mx \sqsubseteq x \implies \mathtt{Y}M \sqsubseteq x$.

7. $P(\bot) \land (\forall x : \sigma P(x) \implies P(Mx)) \implies P(\mathtt{Y}M)$.

For (7) we require that

x is not free in M and

$P(x)$ is a predicate built from atomic formulas using $\forall, \land, \lor$ and $A \implies (-)$ where A is an arbitrary formula without free occurrences of x.

Operational equivalence

A.k.a. contextual equivalence, observational equivalence.

Two terms of higher type are equivalent if they produce the same answer when put in any context of ground type.

$M =_{\mathrm{op}} N$ iff for all ground contexts $C[-]$,
$C[M] \Downarrow \underline{n} \iff C[N] \Downarrow \underline{n}$.

Operational preorder:

$M \sqsubseteq_{\mathrm{op}} N$ iff for all ground contexts $C[-]$,
$C[M] \Downarrow \underline{n} \implies C[N] \Downarrow \underline{n}$.

Failure of full abstraction of the Scott model

Proposition $M = N$ implies $M =_{\mathrm{op}} N$.

However, there are $M =_{\mathrm{op}} N$ with $M \neq N$.

I'll write the counter-example in Haskell.

Counter-example to full abstraction (Plotkin 1977)

```
testpor :: Int -> (Bool -> Bool -> Bool) -> Int

testpor n f = if (not (f False False)) &&
                     (f bot True) &&
                     (f True bot)
              then n
              else bot
```

Now $\text{testpor } 0 =_{\text{op}} \text{testpor } 1$ but $\text{testpor } 0 \neq \text{testpor } 1$ in Scott's model.

Rescuing full abstraction

Parallel-or is not definable in PCF: add it!

Then all finite elements become definable.

This implies full abstraction.

(So operational equivalence changes when you add parallel-or.)

Universality (Plotkin 1977)

An element (or function!) is computable iff it is the join of an r.e. ascending sequence of finite elements.

There is a computable function $\exists : (\mathbb{N}_\perp \to \mathbb{B}_\perp) \to \mathbb{B}_\perp$ which is not PCF definable.

$$\exists(p) = \texttt{False} \text{ if } p(\perp) = \texttt{False},$$

$$\exists(p) = \texttt{True} \text{ if } p(n) = \texttt{True} \text{ for some } n \in \mathbb{B}.$$

Theorem. PCF extended with parallel-or and exists is universal.

Universality (Normann 1998)

Theorem. Every total computable functional is PCF definable.

Recursive types

Domain equations. Language FPC. (Much closer to Haskell.)

Similar programme has been developed.

References

Streicher. Domain-Theoretic Foundations of Functional Programming. World Scientific Publishing (2006).

Abramsky and Jung. Domain Theory. In Handbook of Logic in Computer Science, Vol. III, Clarendon Press, (1994).

Plotkin. Pisa notes on domains (1983).

Escardó and Ho. Operational domain theory and topology of a sequential programming language. LICS'2005.

Escardó. Infinite sets that admit fast exhaustive search. LICS'2007.