Semantics

for the lazy functional programmer

Martín Escardó

School of Computer Science, Birmingham University, UK

MGS 2009, Leicester, UK, 30th March — 3rd April

Programming language semantics

Officially, it investigates the "meaning" of (types and) programs.

A systematic collection of mathematical tools for reasoning about programs and programming languages, in a "compositional" way.

Two main kinds of programming language semantics

1. Operational.

   How programs are run (or executed or evaluated).

2. Denotational.

   What types and programs are thought of.
   E.g.
   1. Types are sets, programs are functions.
   2. Types are domains, programs are continuous functions.
   3. Types are game arenas, programs are strategies.
   4. Types are objects of a category, programs are morphisms.

What they have in common

Both are concerned with program *equivalence* and *correctness*.

When are two programs to be considered the same
for all purposes, except time and space efficiency?

(But some kinds of semantics consider efficiency too.)

These lectures: a blend of the two

1. Operational semantics based on ideas from denotational semantics.
2. Particularly from domain theory.
3. Domain theory itself relies on order and topology.
4. We'll develop both, but directly in operational semantics.

Some references:

Our starting point is recorded in:
1. Pitts. Operationally-Based Theories of Program Equivalence.

We follow this up:
2. Escardo and Ho. Operational domain theory and topology of sequential programming languages.

Recommended book for denotational semantics:
3. T. Streicher. Domain-theoretic foundations of functional programming. World Scientific, 2006, 120 pages.

Unfortunately, I won't have time to develop denotational semantics.

But I intend to give a good perspective for you to approach it on your own.

Some reasons for considering program semantics

1. Specification of the language compiler. Is it correct?
2. Understand rigorously what programs do.
3. Design programs to do what we want.
4. Be able to say rigorously what we want.
5. Be able to prove that programs (don't) do what we want.
6. Discover (sometimes counter-intuitive) facts about programs.
7. Design new languages, improve/redesign existing languages.
8. Scientific curiosity.

Example 1

Consider "minimal Haskell": Booleans, integers, finite products, function types, lazy lists, full recursion, λ-calculus.

Define:

```
or :: Bool -> Bool -> Bool
or x y = if x then True else y
```

Then:

```
or False False = False
or False True  = True
or True  False = True
or True  True  = True
```

But

```
or True bot  = True = if True then True else bot
or bot  True = bot  = if bot then True else True
```

Here

```
bot :: a -> a
bot = bot
```

Hence

$$\mathtt{or}\, x\, y \neq \mathtt{or}\, y\, x$$

in general.

Example 2

Claim. There is no "parallel or" function

```
por :: Bool -> Bool -> Bool
```

such that

```
por False False = False
por x True = por True y = True
```

for all x and y of type `Bool`.

Questions.

1. Why, intuitively?
2. How does one actually rigorously prove this?
3. Can the language be extended to accomodate this?
4. Do we want this?

Exercise

How many or-like functions

```
f :: Bool -> Bool -> Bool
```

can you define in Haskell that satisfy

```
f False False = False
f False True  = True
f True  False = True
f True  True  = True
```

but behave differently at `bot`?

Example 3

Define

```
orL, orR :: Bool -> Bool -> Bool

orL x y = if x then True else y
orR x y = if y then True else x
```

Then, although

$$\texttt{orL} \neq \texttt{orR}$$

they are equivalent,

$$\texttt{orL} \sim \texttt{orR},$$

in the sense that they behave in the same way on total elements.

Questions.

1. How does one make this precise?
2. How should totality be defined?

Example 4

Define

```
type Z = Integer
type Cantor = Z -> Bool
```

Claim. There is a total program

```
equal :: (Cantor -> Z) -> (Cantor -> Z) -> Bool
```

such that for all total `f, g :: Cantor -> Z`

```
equal f g = True
```

iff f and g are equal on total inputs.

Question. Really? How can this be? What is the program?

Example 5

Define

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x : xs) = f x : map f xs

iterate :: (a -> a) -> a -> [a]
iterate f x = x : iterate f (f x)
```

Intuitively,

```
iterate f x = [x, f x, f(f x), f(f(f x)), ...]
```

This is an infinite list.

Claim.

```
map f (iterate f x) = iterate f (f x).
```

Intuitively, this is so because both sides are equal to

```
[f x, f(f x), f(f(f x)), f(f(f(f x))), ...]
```

Question. How does one prove claims such as this?

The programming language PCFL

I'll consider a subset of the language Haskell.

1. Base types for booleans and integers.
2. Finite products.
3. Function types.
4. Recursion.
5. Lazy lists.

A precise description of the syntax is given in:
Pitts. Operationally-Based Theories of Program Equivalence.

Also look at:
Plotkin. LCF considered as a programming language.

Main concepts

1. Terms, raw syntax of the language.
2. Types.
3. Contexts.
4. Terms in context.
5. Rules for type assignment.

Syntax

$$
\begin{aligned}
L, M, N \quad ::= \quad & \textit{syntactical variables } x, y, z, f, g, \ldots \\
& \mid \lambda x.M \\
& \mid MN \\
& \mid \mathsf{fix}\, x.M \\
& \mid \mathsf{True} \mid \mathsf{False} \\
& \mid \mathsf{if}\, L\, \mathsf{then}\, M\, \mathsf{else}\, N \\
& \mid 0 \mid 1 \mid 2 \mid \ldots \mid n \mid \ldots \\
& \mid M\, \mathsf{op}\, N \\
& \mid (M, N) \\
& \mid \mathsf{fst}\, M \mid \mathsf{snd}\, M \\
& \mid \mathsf{nil} \mid M : N \\
& \mid \mathsf{case}\, L\, \mathsf{of}\, \{\mathsf{nil} \to M \mid x : xs \to N\}
\end{aligned}
$$

From Haskell recursion to PCFL recursion

Syntax in Haskell:

```
double :: Natural -> Natural
double 0 = 0
double (n+1) = 2 + double n
```

Step 1. Rewrite as a single equation:

```
double = \n -> if n == 0 then 0 else 2 + double(n-1)
```

Step 2. Make `double` into a variable, and define, without recursion:

```
f :: (Natural -> Natural) -> (Natural -> Natural)
f double = \n -> if n == 0 then 0 else 2 + double(n-1)
```

Then `double` satisfies the equation in Step 1 iff

```
double = f(double)
```

Syntax in PCFL:

```
fix double -> \n -> if n == 0 then 0 else 2 + double(n-1)
```

Types

Ground types:

$$\gamma ::= \mathsf{Integer} \mid \mathsf{Bool}\,.$$

General types:

$$\begin{aligned} \sigma, \tau \quad ::= \quad & \gamma \\ & \mid \sigma \to \tau \\ & \mid \sigma \times \tau \\ & \mid [\sigma] \end{aligned}$$

Type assertions

See Pitts Figure 2 in page 247, and Lambda Calculus lectures.

1. $\Gamma \vdash M : \sigma$.
2. Γ is a function from a set of variables to types.
3. $\mathsf{Exp}_\sigma(\Gamma)$ is the set of terms that can have type σ in the type assignment Γ.
4. Exp_σ the set of closed terms of type σ.
5. By an abuse of notation, we often write Exp_σ as σ.
   E.g.
   1. $3 + 5 \in \mathsf{Integer}$.
   2. $\lambda x.x + 3 \in (\mathsf{Integer} \to \mathsf{Integer})$

Program evaluation

Main concepts:

1. Canonical forms (or values) v.
2. Algorithmic rules defining the evaluation relation $M \Downarrow v$.
3. Basic properties (determinacy, subject reduction).

Canonical forms (or values)

$$
\begin{array}{rcl}
v, w & ::= & \mathsf{True} \mid \mathsf{False} \\
& & \mid 0 \mid 1 \mid 2 \mid \ldots \mid n \mid \ldots \\
& & \mid \lambda x.M \\
& & \mid (M, N) \\
& & \mid \mathsf{nil} \mid M : N
\end{array}
$$

Idea: When we reach a canonical form, we stop evaluation.

Evaluation relation (big-step style)

(Small-step style given in the functional programming course.)
Inductively defined relation "term $\Downarrow$ canonical form".

$$\frac{}{v \Downarrow v} \qquad \frac{L \Downarrow \mathsf{True} \quad M \Downarrow v}{\mathsf{if}\ L\ \mathsf{then}\ M\ \mathsf{else}\ N \Downarrow v} \qquad \frac{L \Downarrow \mathsf{False} \quad N \Downarrow w}{\mathsf{if}\ L\ \mathsf{then}\ M\ \mathsf{else}\ N \Downarrow w}$$

$$\frac{M \Downarrow m \quad N \Downarrow n}{M\ \mathsf{op}\ N \Downarrow m\ \mathsf{op}\ n} \qquad \frac{L \Downarrow \lambda x.N \quad N[x := M] \Downarrow v}{LM \Downarrow v} \qquad \frac{M[x := \mathsf{fix}\, x.M] \Downarrow v}{\mathsf{fix}\, x.M \Downarrow v}$$

$$\frac{L \Downarrow (M, N) \quad M \Downarrow v}{\mathsf{fst}\ L \Downarrow v} \qquad \frac{L \Downarrow (M, N) \quad N \Downarrow w}{\mathsf{snd}\ L \Downarrow w}$$

$$\frac{L \Downarrow \mathsf{nil} \quad M \Downarrow v}{\mathsf{case}\ L\ \mathsf{of}\{\mathsf{nil} \to M \mid x : xs \to N\} \Downarrow v} \qquad \frac{L \Downarrow H : T \quad N[x := H, xs := T] \Downarrow w}{\mathsf{case}\ L\ \mathsf{of}\{\mathsf{nil} \to M \mid x : xs \to N\} \Downarrow w}$$

Program equivalence, intuitively

Haskell programmer's version.

Two programs of integer type are equivalent iff they both evaluate to the same number, or both diverge.

Two programs of functional type are equivalent if they produce equivalent outputs for equivalent inputs.

(But how about, e.g., lazy lists?)

Program equivalence, intuitively

Compiler-writer's version.

Two terms are equivalent if when one is substituted for the other in the same program of observable type, no behavioural change can be observed in the resulting program.

Important for code optimization.

1. The programmer writes down a recipe for some behaviour.
2. Any optimization the compiler performs is allowed to change the internal behaviour, but not the external observable behaviour.

Program equivalence, intuitively

Denotational semantics minded person's version.

Two programs are equivalent when they denote the same element of the model.

E.g.

1. The same number.
2. The same lazy list.
3. The same function.

Which notion of equivalence is right?

All of them have their uses!

In any case, all of them agree for terms of observable type.

So, any difference arises only on terms of non-observable type.

My contention is that we are free to play with the meaning of terms of non-observable type, provided our choices don't interfere with the sacred meaning of terms of observable type.

In these lectures I chose to adopt the compiler writer's view. (But this doesn't matter much, as I'll explain as we proceed.)

Contextual equivalence

Main concepts:

1. Ground contexts.
2. (Variable capturing) context substitution.
3. Typed contexts.
4. Contextual equivalence and preorder.

Ground contexts

A context is a term with a missing subterm.

We can supply the missing term M in a context C obtaining a term $C[M]$.
(See Pitts' notes for a formal definition.)

We are interested in contexts of ground type.

Contextual equivalence

See Pitts page 252.

Definition

We write $M = N$ to mean that for every ground context C capturing all free variables of M, N:

$$C[M] \Downarrow v \iff C[N] \Downarrow v.$$

Other terminologies:

1. Observational equivalence.
2. Operational equivalence.

This is reflexive, transitive and symmetric.

Contextual preorder

Definition

We write $M \sqsubseteq N$ to mean that for every ground context C capturing all free variables of M, N:

$$C[M] \Downarrow v \implies C[N] \Downarrow v.$$

Other terminologies:

1. Observational preorder.
2. Operational preorder.

This is reflexive and transitive.

Examples/exercises

Define $\bot = \mathsf{fix}\, x.x$, say for $x : \mathsf{Integer}$.

1. There is no v such that $\bot \Downarrow v$.
2. $\bot \sqsubseteq 3$.
3. $\bot \sqsubseteq M$ for any M.
4. $(\bot, 3, \bot) \sqsubseteq (2, 3, \bot)$.
5. $(2, \bot, \bot) \sqsubseteq (2, 3, \bot)$.
6. The least upper bound of $(\bot, 3, \bot)$ and $(2, \bot, \bot)$ is $(2, 3, \bot)$.

Examples/exercises

Define

```
bot :: a -> a
bot = bot

f :: Integer -> Integer -> Integer
f  0     n = bot
f (k+1) 0 = 1
f (k+1) n = n * f k (n-1)
```

Write this without pattern matching, using fix.
Show that:

1. $f\ k\ n = \bot \iff k < n$.
2. $f\ n\ n = n!$.
3. If $f\ k\ n \sqsubseteq f\ (k+1)\ n$.

Properties of contextual equivalence and preorder

1. Adequacy.
2. (In)equational logic.
3. β-rules.
4. Extensionality.
5. Recursion unfolding.
6. Rational completeness and continuity.

Adequacy

Recall that γ ranges over ground types.

If $M : \gamma$ is a closed term, then

$$M = v \iff M \Downarrow v.$$

If $M, N : \gamma$ are closed then $M = N$ iff $M \Downarrow v \iff N \Downarrow v$.

(In)equational logic

See Pitts page 254.

1. $M \sqsubseteq N$ implies $M[x := L] \sqsubseteq N[x := L]$.
2. $M = N$ implies $M[x := L] = N[x := L]$.
3. $x \sqsubseteq x$.
4. If $x \sqsubseteq y$ and $x \sqsubseteq y$ then $x \sqsubseteq y$.
5. $x \sqsubseteq y$ and $x \sqsubseteq y$ iff $x = y$.
6. $x = y$ implies $y = x$.
7. If $x = y$ and $y = z$ implies $y = z$.
8. $y \sqsubseteq y'$ implies $\lambda x.y \sqsubseteq \lambda x.y'$.
9. $y \sqsubseteq y'$ implies $\mathsf{fix}\, x.y \sqsubseteq \mathsf{fix}\, x.y'$.
10. $l \sqsubseteq l'$ and $y \sqsubseteq y'$ and $z \sqsubseteq z'$ implies

$$\begin{array}{rl} & \mathsf{case}\, l\, \mathsf{of}\, \{\mathsf{nil} \to y \mid x : xs \to z\} \\ \sqsubseteq & \mathsf{case}\, l'\, \mathsf{of}\, \{\mathsf{nil} \to y' \mid x : xs \to z'\}. \end{array}$$

β-rules

1. $(\lambda x.M)N = M[x := N]$.
2. $\mathsf{if}\ \mathsf{True}\ \mathsf{then}\ x\ \mathsf{else}\ y = x$.
3. $\mathsf{if}\ \mathsf{False}\ \mathsf{then}\ x\ \mathsf{else}\ y = y$.
4. $m\ \mathsf{op}\ n = m\,\mathsf{op}\,n$.
5. $\mathsf{fst}(M, N) = M$.
6. $\mathsf{snd}(M, N) = N$.
7. $\mathsf{case}\ \mathsf{nil}\ \mathsf{of}\{\mathsf{nil} \to M \mid x : xs \to N\} = M$.
8. $\mathsf{case}(H : T)\ \mathsf{of}\{\mathsf{nil} \to M \mid x : xs \to N\} = N[x := H, xs = T]$.

Extensionality

1. $\vec{x}\colon\vec{\sigma}\vdash L = M$ iff $L[\vec{x} := \vec{N}] = M[\vec{x} := \vec{N}]$ for all closed $\vec{N}\colon\vec{\sigma}$.
2. If $f, g\colon\sigma\to\tau$ then $f = g$ iff $f\,x = g\,x$ for all $x\colon\sigma$.
3. If $p, q\colon\sigma\times\tau$ then $p = q$ iff $\mathsf{fst}\,p = \mathsf{fst}\,q$ and $\mathsf{snd}\,p = \mathsf{snd}\,q$.
4. For all $l, l'\colon[\sigma]$, we have $l = l'$ iff
   1. $l = \mathsf{nil}$ implies $l' = \mathsf{nil}$, and
   2. $l = x : xs$ implies $l' = x : xs$,

Order extensionality

1. $\vec{x} : \vec{\sigma} \vdash L \sqsubseteq M$ iff $L[\vec{x} := \vec{N}] \sqsubseteq M[\vec{x} := \vec{N}]$ for all closed $\vec{N} : \vec{\sigma}$.
2. If $f, g : \sigma \to \tau$ then $f \sqsubseteq g$ iff $f\,x \sqsubseteq g\,y$ for all $x \sqsubseteq y : \sigma$.
3. If $p, q : \sigma \times \tau$ then $p \sqsubseteq q$ iff $\mathsf{fst}\,p \sqsubseteq \mathsf{fst}\,q$ and $\mathsf{snd}\,p \sqsubseteq \mathsf{snd}\,q$.
4. For all $l, l' : [\sigma]$, we have $l \sqsubseteq l'$ iff
   1. $l = \mathsf{nil}$ implies $l' = \mathsf{nil}$, and
   2. $l = x : xs$ implies $l' = x' : xs'$ for some $x' \sqsupseteq x$ and $xs' \sqsupseteq xs$,

η-rules

$$f = \lambda x.f\ x.$$

$$b = \text{if } b \text{ then True else False}\,.$$

$$p = (\text{fst } p, \text{snd } p).$$

$$l = \text{case } l \text{ of} \{\text{nil} \longrightarrow \text{nil} \mid x : xs \longrightarrow x : xs\}.$$

Recursion unfolding

$$\mathsf{fix}\, x.M = M[x := \mathsf{fix}\, x.M]$$

We write

$$\mathsf{fix}\, F = \mathsf{fix}\, x.F\, x$$

where x doesn't occur free in F.

Then the above can be expressed as

$$\mathsf{fix}\, f = f(\mathsf{fix}\, f).$$

That is, $\mathsf{fix}\, f$ is a fixed point of f.

Least (pre-)fixed points

For $f : \sigma \to \sigma$ and $x : \sigma$

$$f\,x \sqsubseteq x \implies \mathsf{fix}\,f \sqsubseteq x.$$

In particular

$$f\,x = x \implies \mathsf{fix}\,f \sqsubseteq x.$$

This says that $\mathsf{fix}\,f$ is the least fixed-point of f.

Syntactic bottom

The term $\bot = \mathsf{fix}\, x.x$ acts as a least element:

$$\bot \sqsubseteq x.$$

Rational completeness

$$\bot \sqsubseteq f(\bot) \sqsubseteq f^2(\bot) \sqsubseteq \cdots f^n(\bot) \sqsubseteq f^{n+1}(\bot) \sqsubseteq \cdots$$

Moreover:

$$\mathsf{fix}\, f \sqsubseteq x \iff f^n(\bot) \sqsubseteq x \text{ for all } n.$$

That is, fixed points are least upper bounds:

$$\mathsf{fix}\, f = \bigsqcup_n f^n(\bot).$$

Rational continuity

More generally:

$$g(\bot) \sqsubseteq g(f(\bot)) \sqsubseteq g(f^2(\bot)) \sqsubseteq \cdots g(f^n(\bot)) \sqsubseteq g(f^{n+1})(\bot) \sqsubseteq \cdots$$

and

$$g(\mathrm{fix}\, f) \sqsubseteq y \iff g(f^n(\bot)) \sqsubseteq y \text{ for all } n.$$

That is:

$$g(\mathrm{fix}\, f) \quad = \quad \bigsqcup_n g(f^n(\bot)).$$

Rational chains

Definition

A rational chain is a sequence of the form

$$x_n = g(f^n(\bot)).$$

The above says that every rational chain has a least upper bound, namely

$$g(\text{fix}\, f).$$

However, not every increasing chain has a least upper bound.

Domain theory and topology

We'll introduce two new types to PCFL, which in Haskell can be defined as

```
data Sierp = Top
data OmegaBar = S OmegaBar
```

1. `Sierp` has two elements, `Top` and `bot`.
2. `OmegaBar` has elements

   ```
   bot, S bot, S(S bot), S(S(S bot)), ..., infty
   ```

   where

   ```
   infty = S infty
   ```

PCFL + S + $\bar{\omega}$

Add two new base types.

Sierpinski space: S.

Vertical natural numbers: $\bar{\omega}$.

PCFL + S + $\bar{\omega}$

Terms:

1. $\top : \mathsf{S}$ is a term.
2. If $M : \mathsf{S}$ and $N : \sigma$ are terms then $(\text{if } M \text{ then } N) : \sigma$ is a term.
3. If $M : \bar{\omega}$ is a term then so are $(M + 1) : \bar{\omega}$ and $(M - 1) : \bar{\omega}$ and $(M > 0) : \mathsf{S}$.

Remarks:

1. Notice that there is no "else" clause in the above construction.
2. The only value (or canonical form) of type S is $\top$.
3. The values of type $\bar{\omega}$ are the terms of the form $M + 1$.
4. The role of zero is played by divergent computations.
5. A term $(M > 0)$ can be thought of as a convergence test.

PCFL + S + $\bar{\omega}$

Big-step operational semantics:

$$\frac{M \Downarrow \top \quad N \Downarrow v}{(\text{if } M \text{ then } N) \Downarrow v} \qquad \frac{M \Downarrow N+1 \quad N \Downarrow v}{M-1 \Downarrow v} \qquad \frac{M \Downarrow M'+1}{M>0 \Downarrow \top}$$

The Sierpinski type captures observational equivalence

$M \sqsubseteq N$ iff for any context $C[-]\colon \mathsf{S}$ that captures the free variables of M and N, if $C[M] = \top$ then $C[N] = \top$,

Elements of a type

1. An *element* of a type is a closed term of that type.
2. We adopt usual set-theoretic notation.
3. E.g. we write $x \in \sigma$ and $f \in (\sigma \to \tau)$.
4. We occasionally refer to elements of function types as *functions*.

Observational equivalence and order for elements

The above definitions and observations specialize to:

1. $x = y$ in σ iff for every $p \in (\sigma \to \mathsf{S})$,

$$p(x) = \top \iff p(y) = \top.$$

2. $x \sqsubseteq y$ in σ iff for every $p \in (\sigma \to \mathsf{S})$,

$$p(x) = \top \implies p(y) = \top.$$

The elements of S

1. The elements $\bot$ and $\top$ of S are contextually ordered by
$$\bot \sqsubseteq \top.$$
2. They are contextually inequivalent.
3. Any element of S is equivalent to one of them.
4. We think of S as a type of outcomes of observations or semi-decisions.
5. $\top$ is "observable true" and $\bot$ is "unobservable false".

The elements of $\bar{\omega}$

1. $\infty = \mathsf{fix}\, x.x + 1 \in \bar{\omega}$.
2. By an abuse of notation, for $n \in \mathbb{N}$ we write n to denote the element $\mathrm{succ}^n(\bot)$ of $\bar{\omega}$, where $\mathrm{succ}(x) = x + 1$.
3. The elements $0, 1, 2, \ldots, n, \ldots, \infty$ of $\bar{\omega}$ are all contextually inequivalent, and any element of $\bar{\omega}$ is contextually equivalent to one of them.
4. They are contextually ordered by

$$0 \sqsubseteq 1 \sqsubseteq 2 \sqsubseteq \ldots \sqsubseteq n \sqsubseteq \ldots \sqsubseteq \infty.$$

5. E.g. we have $0 - 1 = 0$ and $(x + 1) - 1 = x$ and $(0 > 0) = \bot$ and $(x + 1 > 0) = \top$ and $\infty - 1 = \infty$ and $(\infty > 0) = \top$.

The generic rational chain

Lemma

The sequence $0, 1, 2, \ldots, n, \ldots$ *in* $\bar{\omega}$ *is a rational chain with least upper bound* ∞*, and, for any* $l \in (\bar{\omega} \to \sigma)$*,*

$$l(\infty) = \bigsqcup_n l(n).$$

Proof.

$n = \mathrm{succ}^n(\bot)$ and $\infty = \mathsf{fix}\,\mathrm{succ}$.
Hence we can take $g = l$ and $f = \mathrm{succ}$. □

The generic rational chain

Lemma

A sequence $x_n \in \sigma$ is a rational chain if and only if there exists $l \in (\bar{\omega} \to \sigma)$ such that for all $n \in \mathbb{N}$,

$$x_n = l(n),$$

and hence such that $\bigsqcup_n x_n = l(\infty)$.

Proof.

($\Rightarrow$): Given $f \in (\tau \to \tau)$ and $g \in (\tau \to \sigma)$ with $x_n = g(f^n(\bot))$, recursively define

$$h(y) = \text{if } y > 0 \text{ then } f(h(y-1)).$$

Then $h(n) = f^n(\bot)$ and hence we can take $l = g \circ h$.
($\Leftarrow$): Take $g = l$ and $f(y) = y + 1$. □

Rational continuity revisited

Proposition

If $f \in (\sigma \to \tau)$ and x_n is a rational chain in σ, then

1. *$f(x_n)$ is a rational chain in τ, and*
2. *$f(\bigsqcup_n x_n) = \bigsqcup_n f(x_n)$.*

Proof.

Take $l \in (\bar{\omega} \to \sigma)$ such that $x_n = l(n)$.
Then the definition $l'(y) = f(l(y))$ shows that $f(x_n)$ is rational.
Now calculate:
$f(\bigsqcup_n x_n) = f(l(\infty)) = l'(\infty) = \bigsqcup_n l'(n) = \bigsqcup_n f(l(n)) = \bigsqcup_n f(x_n)$.

Particular case

Corollary

For any rational chain f_n in $(\sigma \to \tau)$ and any $x \in \sigma$,

1. *$f_n(x)$ is a rational chain in τ, and*
2. $(\bigsqcup_n f_n)(x) = \bigsqcup_n f_n(x)$.

Proof.

Apply the above to the evaluation functional $F \in ((\sigma \to \tau) \to \tau)$ defined by $F(f) = f(x)$. □

Topology

Definition

We say that a set U of elements of a type σ is *open* if there is $\chi_U \in (\sigma \to \mathsf{S})$ such that for all $x \in \sigma$,

$$\chi_U(x) = \top \iff x \in U.$$

Exercise. The subset $\{\top\}$ of S is open, but $\{\bot\}$ is not. Enumerate all open subsets.

Axioms for topology

Definition

We say that a sequence of open sets in σ is a rational chain if the corresponding sequence of characteristic functions is rational in the type $(\sigma \to \mathsf{S})$.

Proposition

For any type, the open sets are closed under the formation of

1. *finite intersections and*
2. *rational unions.*

Proof.

Programming exercise. □

(Unless the language has parallel features, the open sets don't form a topology in the classical sense.)

Topological continuity

Proposition

For any $f \in (\sigma \to \tau)$ and any open subset V of τ, the set $f^{-1}(V) = \{x \in \sigma \mid f(x) \in V\}$ is open in σ.

Proof.

If $\chi_V \in (\tau \to \mathsf{S})$ is the characteristic function of the set V then $\chi_V \circ f \in (\sigma \to \mathsf{S})$ is that of $f^{-1}(V)$. □

Specialization order

Lemma

For $x, y \in \sigma$, the relation $x \sqsubseteq y$ holds iff $x \in U$ implies $y \in U$ for every open subset U of σ.

Hence $\uparrow x \stackrel{\text{def}}{=} \{y \in \sigma \mid x \sqsubseteq y\} = \bigcap\{U \text{ open in } \sigma \mid x \in U\}$.

Proof.

This is a reformulation of a property stated above.
The conclusion follows from the definition of intersection. □

Open sets are Scott open

Proposition

For any open set U in a type σ,

1. *if $x \in U$ and $x \sqsubseteq y$ then $y \in U$, and*
2. *if x_n is a rational chain with $\bigsqcup x_n \in U$, then there is $n \in \mathbb{N}$ such that already $x_n \in U$.*

Proof.

(1): By monotonicity of χ_U.
(2) By rational continuity of χ_U : If $\bigsqcup x_n \in U$ then $\top = \chi_U(\bigsqcup_n x_n) = \bigsqcup \chi_U(x_n)$ and hence $\top = \chi_U(x_n)$ for some n, i.e., $x_n \in U$. □

Next

1. Finite elements.
2. Continuity in terms of finite elements.
3. Proof principles based on finite elements.