What we have

In the last lecture we showed how to construct complex CPO's from simpler CPO's.

- if $D_1, D_2, \ldots, D_n$, are CPO's then so is $(D_1 \times D_2 \times \ldots \times D_n)$,
- if $D_1, D_2, \ldots, D_n$, are CPO's then so is $(D_1 + D_2 + \ldots + D_n)$,
- if D is a CPO, then $D_\perp = \{\lfloor d \rfloor \mid d \in D\} \cup \{\perp\}$ is a CPO.
- if E is a CPO, then the space $D \to E$ of all continuous functions from D to E is a CPO.

Moreover, $D_\perp$ is pointed, and if E and $D_1, \ldots, D_n$ are pointed CPO's, then so are $(D_1 \times D_2 \times \ldots \times D_n)$ and $D \to E$. We can form expressions like this in our metalanguage:

- constants $u \in U$, $\mathsf{true}, \mathsf{false} \in T$, where $T = U + U$, $0, 1, 2, \ldots \in \mathcal{Z}$
- lifting $\lfloor n \rfloor$
- tupling $\langle \cdot, \ldots, \cdot \rangle$ and projection π_i
- injection $\mathrm{in}_i(\cdot)$
- application $\cdot(\cdot)$, composition $\cdot \circ \cdot$
- (continuous) functions curry and fix.

One more tool we need is to be able to use abstraction – form functions from open terms using the λ operator.

Abstraction

We would like to use the construct $\lambda x \in D.\ e$. Thus we'd appreciate a theorem saying that if the expression e is continuous (in some sense), then $\lambda x \in D.\ e$ is also continuous. In order to do this we first need to define what does it mean for an expression e which possibly contains free variables to be continuous.

Definition. Expression e is continuous in variable $x \in D$ iff for arbitrary values of all other variables, the function $\lambda x \in D.\ e$ is continuous (we need to worry only about variables that are free in e). Expression e is continuous iff it is continuous in all variables (again, only variables free in e matter).

Theorem. If e is an expression in our metalanguage built from constants, continuous functions and variables using tuple construction, application and abstraction, then e is continuous in all variables.

We prove by induction on structure of e that e is continuous in its variables. Assume that all subexpressions of e are continuous in all their variables. If e is a ...

- continuous function like curry, fix or π_i, there is nothing to prove.
- constant c: $\lambda x \in D.\ c$ is a constant function, and thus continuous
- variable $y = x$: $\lambda x \in D.\ x$ is identity on D
- variable $y \neq x$: $\lambda x \in D.\ y$ is constant function
- tuple $\langle e_1, \ldots, e_n \rangle$: From induction hypothesis we know that $\lambda x \in D.\ e_i$ is continuous for $i = 1, \ldots, n$. $\rangle$From the previous lecture we know that $\lambda x \in D.\ \langle e_1, \ldots, e_n \rangle$ is continuous in x as long as each of $\lambda x \in D.\ e_i$ is continuous in x.

- application $c(e')$, where c is a continuous function: By the induction hypothesis, $\lambda x \in D.\ e'$ is continuous. Thus, $\lambda x \in D.\ c(e') = c \circ \lambda x \in D.\ e'$ is continuous, since composition of continuous functions is continuous.

- abstraction $\lambda y \in E.\ e'$, where $y = x$: $\lambda x \in D \lambda y \in E.\ e' = \lambda x \in D \lambda x \in D.\ e'$ is a constant function, thus continuous

- abstraction $\lambda y \in E.\ e'$, where $y \neq x$: From induction hypothesis, e' is continuous, thus $e'\{\pi_1 p/x\}$ is continuous and also $e'\{\pi_1 p/x\}\{\pi_2 p/y\}$ is continuous in its variables. Since $\lambda x \in D.\ \lambda y \in E.\ e' = \mathsf{curry}(\lambda p \in D \times E.\ e'\{\pi_1 p/x\}\{\pi_2 p/y\})$ and curry maps continuous functions to continuous functions, $\lambda x \in D.\ \lambda y \in E.\ e'$ is also continuous.

Note that application of e_1 to e_2 is $e_1(e_2) = \mathsf{apply}(\langle e_1, e_2 \rangle)$ and fix e is an application of a continuous function fix to e, so both are covered by the cases above.

REC

Let's apply the metalanguage to define the semantics of a simple language REC. A program in REC consists of a *declaration* of functions:

$$d := f_1(x_1, x_2, \ldots, x_{a_1}) = e_1,\ \ldots\ , f_n(x_1, x_2, \ldots, x_{a_n}) = e_n$$

where each expression on the right-hand side of each function definition has the form

$$e := n \mid x \mid e_0 \oplus e_1 \mid \textbf{ifz}\ e_0\ \textbf{then}\ e_1\ \textbf{else}\ e_2 \mid f_i(e_1, \ldots, e_{a_i})$$

and an expression e. Thus, a program is a pair (d, e).

An Example

We can write a REC program for computing the next prime number after 1000 (note: true=0, false=1)

$$\begin{aligned} f_1(n,m) &= \textbf{ifz}\ m*m > n\ \textbf{then}\ 0\ \textbf{else}\ \textbf{ifz}\ n\%m\ \textbf{then}\ 1\ \textbf{else}\ f_1(n, m+1) \\ f_2(n) &= \textbf{ifz}\ f_1(n,2)\ \textbf{then}\ n\ \textbf{else}\ f_2(n+1) \\ f_2(1000) & \end{aligned}$$

Thus REC is expressive enough to handle recursive functions and we can code up loops of them.

Operational semantics of REC

We define a configuration of a program to be a pair (d, e), where d are the function definitions and e is an expression.

Interesting cases of rules:

$$\begin{array}{rcll} (d, n_1 \oplus n_2) & \to & (d, n) & n = n_1 \oplus n_2 \\ (d, \textbf{ifz}\ n\ \textbf{then}\ e_1\ \textbf{else}\ e_2) & \to & (d, e_1) & n = 0 \\ (d, \textbf{ifz}\ n\ \textbf{then}\ e_1\ \textbf{else}\ e_2) & \to & (d, e_2) & n \neq 0 \\ (d, f_i(n_1, \ldots, n_{a_i})) & \to & (d, e_i\{n_1/x_1, \ldots, n_{a_i}/x_{a_i}\}) & \end{array}$$

Note that we need no rule for $(d, x) \to ?$, since we always substitute away free variables.

CBV Denotational semantics

Suppose we are given values of variables (ρ) and meaning of functions (ϕ) appearing our language. Formally, we have

$$\begin{array}{l} \rho \in \mathbf{Env} = (\mathbf{Var} \to \mathcal{Z}) \\ \phi \in \mathbf{Fenv} = (\mathcal{Z}^{a_1} \to \mathcal{Z}_\perp) \times \ldots \times (\mathcal{Z}^{a_n} \to \mathcal{Z}_\perp) \\ \mathcal{C}[\![e]\!] \in \mathbf{Denotation} = \mathbf{Fenv} \to \mathbf{Env} \to \mathcal{Z}_\perp \end{array}$$

Then we can define the meaning $\mathcal{C}[\![e]\!]\phi\rho$ of an expression e inductively as follows:

$$\begin{array}{rcl} \mathcal{C}[\![n]\!]\phi\rho & = & \lfloor n \rfloor \\ \mathcal{C}[\![x]\!]\phi\rho & = & \lfloor \rho(x) \rfloor \\ \mathcal{C}[\![e_0 \oplus e_1]\!]\phi\rho & = & \mathcal{C}[\![e_0]\!]\phi\rho \oplus_\perp \mathcal{C}[\![e_1]\!]\phi\rho \\ \mathcal{C}[\![\mathbf{ifz}\ e_0\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2]\!]\phi\rho & = & \mathsf{let}\ n = \mathcal{C}[\![e_0]\!]\phi\rho.\mathsf{if}\ n = 0\ \mathsf{then}\ \mathcal{C}[\![e_1]\!]\phi\rho\ \mathsf{else}\ \mathcal{C}[\![e_2]\!]\phi\rho \\ \mathcal{C}[\![f_i(e_1, \ldots, e_{a_i})]\!]\phi\rho & = & \mathsf{let}\ n_1 = \mathcal{C}[\![e_1]\!]\phi\rho \ldots \mathsf{let}\ n_{a_i} = \mathcal{C}[\![e_{a_i}]\!]\phi\rho. \\ & & \quad (\pi_i\phi)(\langle \mathcal{C}[\![e_1]\!]\phi\rho, \ldots, \mathcal{C}[\![e_{a_i}]\!]\phi\rho \rangle) \end{array}$$

Of course, we would like to find ϕ such that its i-th component has the same meaning as e_i:

$$\pi_i\phi = \lambda y_1, \ldots, y_{a_i} \in \mathcal{Z}.\ \mathcal{C}[\![e_i]\!]\phi\rho[x_1 \mapsto y_1, \ldots, x_{a_i} \mapsto y_{a_i}]$$

for every $i = 1, \ldots, n$ and every ρ.

For every ρ, this defines an equation

$$\begin{array}{rl} \phi = & \langle \lambda v \in \mathcal{Z}^{a_1}.\ \mathcal{C}[\![e_1]\!]\phi\rho[x_1 \mapsto \pi_1 v, \ldots, x_{a_1} \mapsto \pi_{a_1} v], \\ & \ldots \\ & \lambda v \in \mathcal{Z}^{a_n}.\ \mathcal{C}[\![e_n]\!]\phi\rho[x_1 \mapsto \pi_1 v, \ldots, x_{a_n} \mapsto \pi_{a_n} v] \rangle \end{array}$$

The $\emptyset$ is a variable environment with no bindings – no variable is defined. We take $\rho = \emptyset$ and find a fixed point:

$$\begin{array}{rcl} \delta & = & \mathsf{fix}\ \lambda\phi \in (\mathcal{Z}^{a_1} \to \mathcal{Z}_\perp) \times \ldots \times (\mathcal{Z}^{a_n} \to \mathcal{Z}_\perp). \\ & & \langle \lambda v \in \mathcal{Z}^{a_1}.\ \mathcal{C}[\![e_1]\!]\phi\rho[x_1 \mapsto \pi_1 v, \ldots, x_{a_1} \mapsto \pi_{a_1} v], \\ & & \ \ldots \\ & & \ \lambda v \in \mathcal{Z}^{a_n}.\ \mathcal{C}[\![e_n]\!]\phi\rho[x_1 \mapsto \pi_1 v, \ldots, x_{a_n} \mapsto \pi_{a_n} v] \rangle \end{array}$$

Since for a fixed expression e, $\mathcal{C}[\![e_n]\!]$ is built using only allowed operations, it is continuous. The domain $(\mathcal{Z}^{a_1} \to \mathcal{Z}_\perp) \times \ldots \times (\mathcal{Z}^{a_n} \to \mathcal{Z}_\perp)$ is pointed, thus we are guaranteed to find the least fixed point δ. We may thus define the meaning of an expression to be $\mathcal{C}[\![e]\!]\delta\emptyset$.