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 $D_\bot = \{ \lfloor d \rfloor \mid d \in D \} \cup \{ \bot \}$ is a CPO.
- if $E$ is a CPO, then the space $D \rightarrow E$ of all continuous functions from $D$ to $E$ is a CPO.

Moreover, $D_\bot$ 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 $D \rightarrow E$. We can form expressions like this in our metalanguage:

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

One more tool we need is to be able to use abstraction – form functions from open terms using the $\lambda$ 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 $\text{curry}$, $\text{fix}$ or $\pi_i$, 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 $i = 1, \ldots, n$. 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 $y = x$: $\lambda x \in D. e' \circ y = \lambda x \in D. e'$ is a constant function, thus continuous.

- abstraction $\lambda y \in E. e'$, where $y \neq x$: From induction hypothesis, $e'$ is continuous, thus $e'\{\pi_1p/x\}$ is continuous in its variables. Since $\lambda x \in D. \lambda y \in E. e' = \text{curry} (\lambda y \in E. e')$ 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 $e_1(e_2) = \text{apply}(\langle e_1, e_2 \rangle)$ 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

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

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)

$$f_1(n, m) = \text{ifz } m \ast m > n \text{ then } 0 \text{ else ifz } n \% m \text{ then } 1 \text{ else } f_1(n, m + 1)$$

$$f_2(n) = \text{ifz } f_1(n, 2) \text{ then } n \text{ else } f_2(n + 1)$$

$$f_2(1000)$$

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:

$$(d, n_1 \oplus n_2) \rightarrow (d, n) \quad n = n_1 \oplus n_2$$

$$(d, \text{ifz } n \text{ then } e_1 \text{ else } e_2) \rightarrow (d, e_1) \quad n = 0$$

$$(d, \text{ifz } n \text{ then } e_1 \text{ else } e_2) \rightarrow (d, e_2) \quad n \neq 0$$

$$(d, f_i(n_1, \ldots, n_{a_i})) \rightarrow (d, e_i(n_1/x_1, \ldots, n_{a_i}/x_{a_i}))$$

Note that we need no rule for $(d, x) \rightarrow ?$, 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

\[ ρ ∈ \text{Env} = (\text{Var} → \mathbb{Z}) \]
\[ φ ∈ \text{Fenv} = (\mathbb{Z} a_1 → \mathbb{Z}_⊥) × \ldots × (\mathbb{Z} a_n → \mathbb{Z}_⊥) \]
\[ C[e] ∈ \text{Denotation} = \text{Fenv} → \text{Env} → \mathbb{Z}_⊥ \]

Then we can define the meaning \( C[e]φρ \) of an expression \( e \) inductively as follows:

\[
\begin{align*}
C[n]φρ &= \lfloor n \rfloor \\
C[x]φρ &= \lfloor ρ(x) \rfloor \\
C[e_0 + e_1]φρ &= C[e_0]φρ + C[e_1]φρ \\
C[\text{if} z \; e_0 \; \text{then} \; e_1 \; \text{else} \; e_2]φρ &= \text{let } n = C[e_0]φρ. \text{if } n = 0 \text{ then } C[e_1]φρ \text{ else } C[e_2]φρ \\
C[f_i(e_1, \ldots, e_{a_i})]φρ &= \text{let } n_i = C[e_1]φρ. \ldots \text{let } n_{a_i} = C[e_{a_i}]φρ. \pi_i(\pi_i(\ldots(\pi_i(C[e_1]φρ), \ldots, C[e_{a_i}]φρ)) \ldots ) \\
\end{align*}
\]

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

\[ \pi_i φ = λy_1, \ldots, y_{a_i} ∈ \mathbb{Z}. C[e_i]φρ[x_1 ↦ y_1, \ldots, x_{a_i} ↦ y_{a_i}] \]

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

For every \( ρ \), this defines an equation

\[ φ = (λv ∈ Z^{a_1}. C[e_1]φρ[x_1 ↦ π_1 v, \ldots, x_{a_1} ↦ π_{a_1} v], \ldots , λv ∈ Z^{a_n}. C[e_n]φρ[x_1 ↦ π_1 v, \ldots, x_{a_n} ↦ π_{a_n} v]) \]

The \( ϵ \) is a variable environment with no bindings – no variable is defined. We take \( ρ = ϵ \) and find a fixed point:

\[ δ = \text{fix } λφ ∈ (Z^{a_1} → Z_⊥) × \ldots × (Z^{a_n} → Z_⊥). \]
\[ \{ λv ∈ Z^{a_1}. C[e_1]φρ[x_1 ↦ π_1 v, \ldots, x_{a_1} ↦ π_{a_1} v], \ldots , λv ∈ Z^{a_n}. C[e_n]φρ[x_1 ↦ π_1 v, \ldots, x_{a_n} ↦ π_{a_n} v] \} \]

Since for a fixed expression \( e \), \( C[e_n] \) is built using only allowed operations, it is continuous. The domain \((Z^{a_1} → Z_⊥) × \ldots × (Z^{a_n} → Z_⊥)\) is pointed, thus we are guaranteed to find the least fixed point \( δ \). We may thus define the meaning of an expression to be \( C[e]δϵ \).