Programming Computable Functions **PCF**

PCF is a model programming language with higher types, in the sense that it allows the construction of types using a function space constructor. Functions become "first-class" values and can be supplied as inputs to functions or delivered as outputs.

Again, we will be forced with a choice as to whether evaluation should proceed in a call-by-value or call-by-name fashion. The first choice will lead to a language behaving much like the eager language Standard ML, the second to one closely similar in behavior to lazy languages Miranda, Orwell or Haskell.

1 Language Definition

1.1 Syntax

We will use the same syntactic domains and metavariables associated with the recursive programming language **REC**.

As in the language **REC** we have terms in **PCF** which evaluate to natural numbers. Such terms can be built up using numerals, variables, conditionals and arithmetic operations and will yield natural numbers as values or diverge. However in addition there will be terms which can yield pairs or even functions as values.

The main constructions on types considered are products and function space. The syntax of types in **PCF** are given by

$$\tau \ ::= \ \mathbf{N} \mid \mathbf{T} \mid \tau_1 \times \tau_2 \mid \tau_1 \rightarrow \tau_2$$

where **N** and **T** are the *ground types* integer and boolean and the others are *constructed types.*

Accordingly, the syntax of terms in **PCF**
is given by

$$\begin{aligned} t \ ::= \ & x \mid n \mid t_1 + t_2 \mid \mathbf{true} \mid \mathbf{false} \mid t_1 = t_2 \mid \mathbf{if}\ t\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2\ \mathbf{fi} \mid \\ & (t_1, t_2) \mid \mathbf{fst}(t) \mid \mathbf{snd}(t) \mid \\ & \lambda x.t \mid (t_1\ t_2) \mid \\ & \mathbf{let}\ x \Leftarrow t_1\ \mathbf{in}\ t_2 \mid \\ & \mathbf{rec}\ y.(\lambda x.t) \end{aligned}$$

The syntax describes how

- to write arithmetical and boolean expressions in a manner of the language **REC**. Notice that the variable x and the branches of the conditional need not evaluate to ground types in **PCF**.
- to construct pairs (t_1, t_2), and project to the first and second components.
- to define functions using λ-abstraction and apply them.
- to force the prior evaluation of a term t_1 before its value is used in the evaluation of t_2.
- to define a function y recursively to be $\lambda x.t$ using $\mathbf{rec}\ y.(\lambda x.t)$. The identifier y refers to the recursive function itself and may be used within t.

Example 1 *As an example of the definition of recursive functions in* **PCF***, let's rewrite the* **REC** *decreasing program DEC in* **PCF***.*

$$\begin{array}{llcl} DEC: & dec(5)\ \mathbf{where}\ dec(x) & = & f(x,0) \\ & f(x,y) & = & \mathbf{if}\ x = y+1\ \mathbf{then}\ y\ \mathbf{else}\ f(x,y+1)\ \mathbf{fi} \end{array}$$

In **PCF** *it can be defined as:*

$$\begin{array}{lll} PCF\ DEC: & \mathbf{let} & f \Leftarrow \mathbf{rec}\ g\ . \\ & & (\lambda z\ .\ \mathbf{if}\ \ \mathbf{fst}(z) = \mathbf{snd}(z)+1\ \mathbf{then}\ \ \mathbf{snd}(z)\ \mathbf{else}\ g(\mathbf{fst}(z),\ \mathbf{snd}(z)+1)\ \mathbf{fi}\ ) \\ & \mathbf{in} & \mathbf{let}\ dec \Leftarrow (\lambda x.f(x,0)) \\ & & \mathbf{in}(dec\ 5) \end{array}$$

where the variable z is a pair of the original variables x and y: (x,y).

Example 2 *As an example of mutual recursion, let's consider the following* **REC** *program:*

$$\begin{array}{llcl} T: & t\ \mathbf{where}\ f(x) & = & a(g(b(x)), f(c(x))) \\ & g(x) & = & a'(g(b'(x)), f(c'(x))) \end{array}$$

Equivalently, we can write it in **PCF** *as in either of the followings:*

$$\begin{array}{lll} PCF\ T: & \mathbf{let} & f \Leftarrow\ \mathbf{rec}\ f\ . \\ & & \mathbf{let}\ g \Leftarrow \mathbf{rec}\ g\ .\ (\lambda x.a'(g(b'(x)), f(c'(x))))\ \mathbf{in}\ a(g(b(x)), f(c(x))) \\ & \mathbf{in} & \mathbf{let}\ g \Leftarrow \mathbf{rec}\ g\ .\ (\lambda x.a'(g(b'(x)), f(c'(x)))) \\ & & \mathbf{in}\ t \end{array}$$

$$\begin{array}{lll} PCF\ T': & \mathbf{let} & h \Leftarrow \mathbf{rec}\ h\ . \\ & & (\lambda x\ .\ (a\ (\mathbf{snd}(h)\ (b(x)), \mathbf{fst}(h)\ (c(x)))), (a'\ (\mathbf{snd}(h)\ (b'(x)), \mathbf{fst}(h)\ (c'(x))))\ ) \\ & \mathbf{in} & t \end{array}$$

where f and g are represented in t by $\mathbf{fst}(h)$ and $\mathbf{snd}(h)$, respectively.

1.2 Typing Rules

By the syntax of **PCF**, we can write down arithmetical terms as we do in **REC**. However, it is also possible to write down nonsense: to try to add two functions, or give a function too many, or too few, arguments. The well-formed terms t are those which receive a type τ, written $t : \tau$ or $type(t) = \tau$. We say a term t is *typable* when $t : \tau$ for some type τ, according to following typing rules.

Typing Rules

Variables: $x : \tau \quad \text{if } \ type(x) = \tau$

Constants: $n : \mathbf{N} \quad \mathbf{true} : \mathbf{T} \quad \mathbf{false} : \mathbf{T}$

Operations: $\dfrac{t_1 : \mathbf{N} \quad t_2 : \mathbf{N}}{t_1 + t_2 : \mathbf{N}} \quad \dfrac{t_1 : \mathbf{N} \quad t_2 : \mathbf{N}}{t_1 = t_2 : \mathbf{T}}$

Conditional: $\dfrac{t : \mathbf{T} \quad t_1 : \tau \quad t_2 : \tau}{\mathbf{if}\ t\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2\ \mathbf{fi} : \tau}$

Products: $\dfrac{t_1 : \tau_1 \quad t_2 : \tau_2}{(t_1, t_2) : \tau_1 \times \tau_2} \quad \dfrac{t : \tau_1 \times \tau_2}{\mathbf{fst}(t) : \tau_1} \quad \dfrac{t : \tau_1 \times \tau_2}{\mathbf{snd}(t) : \tau_2}$

Functions: $\dfrac{x : \tau_1 \quad t : \tau_2}{\lambda x.t : \tau_1 \to \tau_2} \quad \dfrac{t_1 : \tau_1 \to \tau_2 \quad t_2 : \tau_1}{(t_1\ t_2) : \tau_2}$

Let: $\dfrac{x : \tau_1 \quad t_1 : \tau_1 \quad t_2 : \tau_2}{\mathbf{let}\ x \Leftarrow t_1\ \mathbf{in}\ t_2 : \tau_2}$

Rec: $\dfrac{y : \tau \quad \lambda x.t : \tau}{\mathbf{rec}\ y.(\lambda x.t) : \tau}$

Example 3 *By these typing rules, the types of variables in* **PCF** *program PCF DEC can be derived as follows.*

First we can easily get

$$\dfrac{g : \alpha \to \mathbf{N} \qquad \dfrac{z : \alpha \qquad \dfrac{\dfrac{\mathbf{fst}(z) : \mathbf{N} \qquad \dfrac{\mathbf{snd}(z) : \mathbf{N} \quad 1 : \mathbf{N}}{\mathbf{snd}(z) + 1 : \mathbf{N}}}{\mathbf{fst}(z) = \mathbf{snd}(z) + 1 : \mathbf{T}} \qquad \mathbf{snd}(z) : \mathbf{N} \qquad g(\mathbf{fst}(z), \mathbf{snd}(z) + 1) : \mathbf{N}}{\mathbf{if} \cdots \mathbf{then} \cdots \mathbf{else} : \mathbf{N}}}{\lambda z\ .\ \mathbf{if} \cdots \mathbf{then} \cdots \mathbf{else} : \alpha \to \mathbf{N}}}{\mathbf{rec}\ g\ .\ (\lambda z\ .\ \mathbf{if} \cdots \mathbf{then} \cdots \mathbf{else}) : \alpha \to \mathbf{N}}$$

and

$$\dfrac{g : \mathbf{N} \times \mathbf{N} \to \mathbf{N} \qquad \dfrac{\dfrac{z : \mathbf{N} \times \mathbf{N}}{\mathbf{fst}(z) : \mathbf{N}} \qquad \dfrac{\dfrac{z : \mathbf{N} \times \mathbf{N}}{\mathbf{snd}(z) : \mathbf{N}} \quad 1 : \mathbf{N}}{\mathbf{snd}(z) + 1 : \mathbf{N}}}{(\mathbf{fst}(z), \mathbf{snd}(z) + 1) : \mathbf{N} \times \mathbf{N}}}{g(\mathbf{fst}(z), \mathbf{snd}(z) + 1) : \mathbf{N}}$$

which tells us that $g : \mathbf{N} \times \mathbf{N} \to \mathbf{N}$ *and* $z : \mathbf{N} \times \mathbf{N}$*. Then from the Let rule we have* $f : \mathbf{N} \times \mathbf{N} \to \mathbf{N}$*. Hence* $x : \mathbf{N}$ *and* $\lambda x.f(x, 0) : \mathbf{N} \to \mathbf{N}$ *by*

$$\dfrac{x : \mathbf{N} \qquad \dfrac{f : \mathbf{N} \times \mathbf{N} \to \mathbf{N} \qquad \dfrac{x : \mathbf{N} \quad 0 : \mathbf{N}}{(x, 0) : \mathbf{N} \times \mathbf{N}}}{f(x, 0) : \mathbf{N}}}{\lambda x.f(x, 0) : \mathbf{N} \to \mathbf{N}}$$

Finally the Let rule and the second function rule bring us dec $: \mathbf{N} \to \mathbf{N}$ *and PCF DEC*: $\mathbf{N}$.

1.3 Free Variables

The set of free variables $FV(t)$ of a term t can be defined straightforwardly by structure induction on t:

$$
\begin{array}{rcl}
FV(n) & = & \emptyset \\
FV(x) & = & \{x\} \\
FV(t_1 + t_2) & = & FV(t_1) \cup FV(t_2) \\
FV(\mathbf{true}) & = & \emptyset \\
FV(\mathbf{false}) & = & \emptyset \\
FV(t_1 = t_2) & = & FV(t_1) \cup FV(t_2) \\
FV(\mathbf{if}\, t \,\mathbf{then}\, t_1 \,\mathbf{else}\, t_2 \,\mathbf{fi}) & = & FV(t) \cup FV(t_1) \cup FV(t_2) \\
FV((t_1, t_2)) & = & FV(t_1) \cup FV(t_2) \\
FV(\mathbf{fst}(t)) & = & FV(t) \\
FV(\mathbf{snd}(t)) & = & FV(t) \\
FV(\lambda x.t) & = & FV(t) \setminus \{x\} \\
FV((t_1\ t_2)) & = & FV(t_1) \cup FV(t_2) \\
FV(\mathbf{let}\, x \Leftarrow t_1 \,\mathbf{in}\, t_2) & = & FV(t_1) \cup (FV(t_2) \setminus \{x\}) \\
FV(\mathbf{rec}\, y.(\lambda x.t)) & = & FV(\lambda x.t) \setminus \{y\}
\end{array}
$$

The substitution rules we are going to follow in the semantical descriptions of **PCF** will be those by Curry.

A term t is *closed* iff $FV(t) = \emptyset$, i.e. a term t is closed when it has no free variables.

2 Eager Operational Semantics of PCF

Different from the programming languages **REC** where terms are evaluated to natural numbers, the natural semantics for **PCF** evaluates terms to *canonical forms*. In **PCF**, functions and products become "first-class" values and hence should be evaluated to, instead of natural numbers, corresponding canonical forms.

The judgement $t \in C_\tau$ that a term t is a canonical form of type τ in eager OS is defined by structural induction on τ:

Ground type:	numerals are canonical forms, i.e. $n \in C_{\mathbf{N}}$ booleans are canonical forms, i.e. $\mathbf{true} \in C_{\mathbf{T}}$ and $\mathbf{false} \in C_{\mathbf{T}}$
Product type:	pairs of canonical forms are canonical, i.e. $(c_1, c_2) \in C_{\tau_1 \times \tau_2}$ if $c_1 \in C_{\tau_1}$ and $c_2 \in C_{\tau_2}$
Function type:	closed abstractions are canonical forms, i.e. $\lambda x.t \in C_{\tau_1 \to \tau_2}$ if $\lambda x.t : \tau_1 \to \tau_2$ and $\lambda x.t$ is closed

Note that canonical forms are special kinds of *closed* terms.

In **PCF**, the evaluation relation $\to$ is of the form

$$t \to c$$

where t is a typable closed term and c is a canonical form, meaning t evaluates to c. It is defined inductively next.

Evaluation Rules

Canonical forms: $c \rightarrow c$ where $c \in C_\tau$

Operations:
$$\frac{t_1 \rightarrow n_1 \quad t_2 \rightarrow n_2}{t_1 + t_2 \rightarrow n_1 \oplus n_2} \qquad \frac{t_1 \rightarrow n_1 \quad t_2 \rightarrow n_2}{t_1 = t_2 \rightarrow n_1 \equiv n_2}$$

Conditional:
$$\frac{t \rightarrow \mathbf{true} \quad t_1 \rightarrow c_1}{\mathbf{if}\ t\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2\ \mathbf{fi} \rightarrow c_1} \qquad \frac{t \rightarrow \mathbf{false} \quad t_2 \rightarrow c_2}{\mathbf{if}\ t\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2\ \mathbf{fi} \rightarrow c_2}$$

Products:
$$\frac{t_1 \rightarrow c_1 \quad t_2 \rightarrow c_2}{(t_1, t_2) \rightarrow (c_1, c_2)} \qquad \frac{t \rightarrow (c_1, c_2)}{\mathbf{fst}(t) \rightarrow c_1} \qquad \frac{t \rightarrow (c_1, c_2)}{\mathbf{snd}(t) \rightarrow c_2}$$

Functions:
$$\frac{t_1 \rightarrow \lambda x.t \quad t_2 \rightarrow c_2 \quad t[c_2/x] \rightarrow c}{(t_1\ t_2) \rightarrow c}$$

Let:
$$\frac{t_1 \rightarrow c_1 \quad t_2[c_1/x] \rightarrow c_2}{\mathbf{let}\ x \Leftarrow t_1\ \mathbf{in}\ t_2 \rightarrow c_2}$$

Rec: $\mathbf{rec}\ y.(\lambda x.t) \rightarrow \lambda x\ .\ (t\ [\mathbf{rec}\ y.(\lambda x.t)/y])$

Remarks

- In the Function Rule, in order to evaluate $(t_1\ t_2)$, the rule requires that t_2 to be evaluated first hence an eager semantics.
- The rule for recursive definitions "unfolds" the recursion $\mathbf{rec}\ y.(\lambda x.t)$ once, leading immediately to an abstraction $\lambda x.(t\,[\mathbf{rec}\ y.(\lambda x.t)/y])$, and so a canonical form. The rule will be used to unfold the recursive function once for each recursive call.
- The evaluation relation $\rightarrow$ respects types. If t is of type τ, and $t \rightarrow c$, then c is of type τ. This is called *subject reduction*.
- The evaluation relation $\rightarrow$ is deterministic.
- The evaluation relation $\rightarrow$ is not necessarily terminating. For example considering $dec(0)$.

3 Eager Denotational semantics of PCF

As mentioned above, values in **PCF** are not restricted to just define natural numbers. Since we want to define a denotational semantics, i.e. a partial function from terms to *values*, it seems reasonable to take a closed term t of type τ to denote an element of $(V_\tau)_\perp$ where V_τ is a cpo of *values* of type τ. The cpo's of values in **PCF** can thus be defined by structural induction on type expressions:

$$\begin{aligned} V_{\mathbf{N}} &= \mathbf{N} \\ V_{\tau_1 \times \tau_2} &= V_{\tau_1} \times V_{\tau_2} \\ V_{\tau_1 \rightarrow \tau_2} &= [V_{\tau_1} \rightarrow (V_{\tau_2})_\perp] \end{aligned}$$

The final clause captures the idea that a function value takes a value as input and delivers a value as output or diverges.

In order to handle free variables, the denotational semantics requires a notion of environment to supply values to the free variables. An environment ρ is a function

$$\rho \in \mathbf{Env} = \mathbf{Var} \rightarrow \bigcup\{V_\tau | \tau \text{ a type}\}$$

which respects types in that

$$x : \tau \implies \rho(x) \in V_\tau$$

for any $x \in \mathbf{Var}$ and type τ.

Now a closed term t of type τ is denoted by a function from environments to the cpo $(V_\tau)_\perp$:

$$[\![t]\!] = \mathbf{Env} \to (V_\tau)_\perp \qquad \text{where } t : \tau.$$

The denotation of typable terms t of **PCF** is given by following structural induction (in simplified Meta-Lambda language):

Denotational Semantics Clauses:

$$
\begin{array}{rcl}
[\![x]\!] & = & \lambda\rho\ .\ \rho(x) \\
[\![n]\!] & = & \lambda\rho\ .\ n \\
[\![t_1 + t_2]\!] & = & \lambda\rho\ .\ let\ n_1 \Leftarrow [\![t_1]\!]\rho,\ n_2 \Leftarrow [\![t_2]\!]\rho\ .\ n_1 \oplus n_2 \\
[\![\mathbf{true}]\!] & = & \lambda\rho\ .\ \mathbf{true} \\
[\![\mathbf{false}]\!] & = & \lambda\rho\ .\ \mathbf{false} \\
[\![t_1 = t_2]\!] & = & \lambda\rho\ .\ let\ n_1 \Leftarrow [\![t_1]\!]\rho,\ n_2 \Leftarrow [\![t_2]\!]\rho\ .\ n_1 \equiv n_2 \\
[\![\mathbf{if}\, t_0\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2\ \mathbf{fi}]\!] & = & \lambda\rho\ .\ [\![t_0]\!]\rho \to [\![t_1]\!]\rho,\ [\![t_2]\!]\rho \\
[\![(t_1, t_2)]\!] & = & \lambda\rho\ .\ let\ v_1 \Leftarrow [\![t_1]\!]\rho,\ v_2 \Leftarrow [\![t_2]\!]\rho\ .\ (v_1, v_2) \\
[\![\mathbf{fst}(t)]\!] & = & \lambda\rho\ .\ let\ v \Leftarrow [\![t]\!]\rho\ .\ \pi_1(v) \\
[\![\mathbf{snd}(t)]\!] & = & \lambda\rho\ .\ let\ v \Leftarrow [\![t]\!]\rho\ .\ \pi_2(v) \\
[\![\lambda x.t]\!] & = & \lambda\rho\ .\ (\lambda v\ .\ [\![t]\!]\rho[v/x]) \\
[\![(t_1\ t_2)]\!] & = & \lambda\rho\ .\ let\ \phi \Leftarrow [\![t_1]\!]\rho,\ v \Leftarrow [\![t_2]\!]\rho\ .\ \phi(v) \\
[\![\mathbf{let}\ x \Leftarrow t_1\ \mathbf{in}\ t_2]\!] & = & \lambda\rho\ .\ let\ v \Leftarrow [\![t_1]\!]\rho\ .\ [\![t_2]\!]\rho[v/x] \\
[\![\mathbf{rec}\ y.(\lambda x.t)]\!] & = & \lambda\rho\ .\ (\mu\phi\ .\ (\lambda v\ .\ [\![t]\!]\rho[v/x, \phi/y]))
\end{array}
$$

Notice that, according to the denotational semantics, terms $\mathbf{let}\ x \Leftarrow t_1\ \mathbf{in}\ t_2$ are definable purely by $(\lambda x.t_2)\ t_1$.

Proof.

$$
\begin{array}{rcl}
[\![(\lambda x.t)\ t_1]\!]\rho & = & let\ \phi \Leftarrow [\![\lambda x.t_2]\!]\rho,\ v \Leftarrow [\![t_1]\!]\rho\ .\ \phi(v) \\
& = & let\ \phi \Leftarrow (\lambda v'.\ [\![t_2]\!]\rho[v'/x]),\ v \Leftarrow [\![t_1]\!]\rho\ .\ \phi(v) \\
& = & let\ v \Leftarrow [\![t_1]\!]\rho\ .\ [\![t_2]\!]\rho[v/x] \\
& = & [\![let\ x \Leftarrow t_1\ .\ t_2]\!]\rho
\end{array}
$$

□

Since **PCF** is so similar to the Meta-Lambda language, most of the above rules serve only to move things from inside $[\![]\!]$ to the outside. Note that the simplified version of Meta-Lambda language does not handle many undefined value as the fully packaged Meta-Lambda. We use the simplified version here to capture the most important concepts and constructs of programming languages and leave out the tedious job of handling undefined values. However, for theoretical proofs, the simplified version is not good enough and may even be misleading.

Like the operational semantics, the denotational semantics of **PCF** respects types. For example in the semantic clause for $[\![\lambda x.t]\!]$, $v : \tau_1$ and $\lambda x.t : \tau_1 \to \tau_2$.

4 Correctness of Operational and Denotational Semantics

The question now is if the natural semantics we defined for **PCF** correct with respect to the denotational semantics:

$$(\forall t \in closed(\textbf{Term}) \ . \ \forall c \in \bigcup\{C_\tau \mid \tau \text{ a type}\} \ . \ t \rightarrow c \Leftrightarrow [\![t]\!] = [\![c]\!]) \quad ?$$

Unfortunately, this is not true. Consider the following example,

$$c_1 = \lambda x.x \qquad c_2 = \lambda x.(x+0).$$

Clearly, c_1 and c_2 denote the same function, yet they are two different canonical forms. The added feature of **PCF**, functions as values, prevents the above relationship to be established.

In general, $[\![t]\!] = [\![c]\!] \Rightarrow t \rightarrow c$ does not hold at any type involving functions spaces. However, its converse does hold no matter what is the type of t: $t \rightarrow c \Rightarrow [\![t]\!] = [\![c]\!]$. This leads us to the definition of *adequacy*.

4.1 Adequacy of DS for PCF

DEFINITION 1 *The denotational semantics are adequate with respect to the operational semantics iff for all closed terms t,*

1. $t \rightarrow c \Rightarrow [\![t]\!] = [\![c]\!]$.
2. $[\![t]\!] \Downarrow \ \Rightarrow t \downarrow$ *where*
   - $[\![t]\!] \Downarrow \ \Leftrightarrow \exists v \in V_\tau \ . \ [\![t]\!] = \lfloor v \rfloor$
   - $t \downarrow \ \Leftrightarrow \exists c.t \rightarrow c$

THEOREM 1 *For all closed terms of* **PCF**, *our DS is adequate with respect to our OS.*

Remarks

- Adequacy for **PCF** implies that

$$(\forall t \in closed(\textbf{Term}) \ . \ \forall n \in \mathbf{N} \ . \ t \rightarrow n \Leftrightarrow [\![t]\!] = n).$$

- The first part of adequacy, $t \rightarrow c \Rightarrow [\![t]\!] = [\![c]\!]$, can be proved easily by rule induction.
- The second part, $[\![t]\!] \Downarrow \Rightarrow t \downarrow$, is more difficult to prove. Please refer to Section 11.4 of Winskel's book.

4.2 Denotational and Operational Equivalence

DEFINITION 2 *The denotational equivalence for all closed terms t_1 and t_2 is defined as:*

$$t \sim_D c \Rightarrow [\![t_1]\!] = [\![t_2]\!].$$

DEFINITION 3 *The operational equivalence for all closed terms t_1 and t_2 is defined as:*

$$t \sim_O c \Rightarrow \forall C[\cdot] \ . \ C[t_1] \downarrow = C[t_2] \downarrow$$

where $C[\cdot]$ is called a **PCF** context. *A* **PCF** *context is a term with a distinguishable free variable which can be substituted by a typable term t to obtain a typable term $C[t]$.*

DEFINITION 4 *A denotational semantics is fully abstract with respect to the operational semantics if*

$$t_1 \sim_O t_2 \Leftrightarrow t_1 \sim_D t_2$$

Remark

PCF is not fully abstract because $t_1 \sim_O t_2 \Rightarrow t_1 \sim_D t_2$ does not hold. Please refer to Winskel.

5 Eager vs. Lazy PCF

So far, we have been considering a call-by-value, or eager, version of **PCF**. We would now like to look at a call-by-name, or lazy, version of **PCF**. The major differences between eager and lazy **PCF** are listed bellow. For more details, please refer to Chapter 11 of Winskel.

- Similar to **REC**, the most notable difference between eager and lazy **PCF** is the function application rule. For OS, it has been changed from

  $$\frac{t_1 \to \lambda x.t \quad t_2 \to c_2 \quad t[c_2/x] \to c}{(t_1\ t_2) \to c}$$

  to

  $$\frac{t_1 \to^l \lambda x.t \quad t[t_2/x] \to^l c}{(t_1\ t_2) \to^l c}$$

  because in lazy evaluation terms will be passed to a function as is without being first evaluated. Similar modifications has been done for DS also.

- Accordingly, because the dependency of the Let clause to function applications, the Let clause has been changed from

  $$\frac{t_1 \to c_1 \quad t_2[c_1/x] \to c_2}{\mathbf{let}\ x \Leftarrow t_1\ \mathbf{in}\ t_2 \to c_2}$$

  to

  $$\frac{t_2[t_1/x] \to^l c}{\mathbf{let}\ x \Leftarrow t_1\ \mathbf{in}\ t_2 \to^l c}$$

- The lazy canonical forms for product type has been changed from

  $$(c_1,\ c_2) \in C_{\tau_1 \times \tau_2} \quad \text{if } c_1 \in C_{\tau_1}\ \&\ c_2 \in C_{\tau_2}$$

  to

  $$(t_1,\ t_2) \in C^l_{\tau_1 \times \tau_2} \quad \text{if } c_1 : \tau_1\ \&\ t_2 : \tau_2 \text{ with } t_1 \text{ and } t_2 \text{ closed}$$

  Hence the rules related have been modified accordingly.

- As in **REC**, the definitions of domains of values have been extended with undefined bottom ($\bot$):

  $$\begin{aligned} V^l_{int} &= \mathbf{N} \\ V^l_{\tau_1 \times \tau_2} &= (V^l_{\tau_1})_\bot \times (V^l_{\tau_2})_\bot \\ V^l_{\tau_1 \to \tau_2} &= [(V^l_{\tau_1})_\bot \to (V^l_{\tau_2})_\bot] \end{aligned}$$

  Accordingly, the definition of environments has been changed to

  $$\rho \in \mathbf{Env} = \mathbf{Var} \to \bigcup \{(V^l_\tau)_\bot | \tau \text{ a type}\}$$