Continuation Semantics or Expressing Implication by Negation *

Y. Lafont B. Reus T. Streicher

Laboratoire d'Informatique
CNRS - Ecole Normale supérieure
Paris, FRANCE
lafont@dmi.ens.fr

Fakultät für Mathematik und Informatik
Ludwig Maximilians Universität
Munich, GERMANY
[reus|streiche]@informatik.uni-muenchen.de

Abstract

Starting from the ideas of J.Y. Girard on making classical logic constructive, we isolate a fragment of intuitionistic logic with negation instead of implication, and we investigate the corresponding programming paradigm. In particular, we get a nice *continuation passing style* translation of lambda calculus, from which we derive a semantics of non-functional features in functional programming, such as inputs/outputs and Felleisen's $\mathcal{C}$ operator.

1 Introduction

We present a functional language for representing proofs in the $\wedge,\neg$-fragment of intuitionistic logic. This language appears as the kernel of a basic language for programming with continuations which we call BCL. Based on Girard's seminal paper [Girard91] we describe a translation of classical logic into the $\wedge,\neg$-fragment of intuitionistic logic, i.e. we express *implication in terms of negation.* This amounts to a new continuation passing translation of simply typed λ-calculus (for literature about continuations see e.g. [FFKD86, Fisher72, Griffin90, MeWa85]). An important point here is that the proof object for the proposition $\neg\neg\varphi \to \varphi$ of classical logic corresponds to Felleisen's control operator $\mathcal{C}$ [FFKD86, Murthy91]. We extend the calculus for proof objects of the $\wedge,\neg$-fragment of intuitionistic logic by the concepts of general recursion and input/output operations to the full language BCL for which we give a denotational semantics. We further describe

*This work was partially sponsored by the DAAD-project PROCOPE

an extension PCF^{ext} of Plotkin's PCF by control and input/output operators. The continuation passing style translation of simply typed λ-calculus is extended to full PCF^{ext}. From this and the denotational semantics of BCL we derive a new continuation semantics of PCF^{ext}.

2 Sequent calculus

This section is a reformulation of Girard's calculus **LC** [Girard91].

2.1 The calculus LJ$^-$

Consider the propositional fragment **LJ**$^-$ of Gentzen's calculus **LJ** for intuitionistic logic without implication and without rule of right weakening:

- A formula is of the form P (atom), $\neg A$ (negation of a formula), $A \wedge B$ (conjunction) or $A \vee B$ (disjunction).
- A sequent is of the form $\Gamma \vdash \Theta$ where Γ is a sequence of formulae and Θ consists of one or zero formulae.

The rules of **LJ**$^-$ are the following:

- left exchange, left contraction and left weakening:

$$\frac{\Gamma, A, B, \Delta \vdash \Theta}{\Gamma, B, A, \Delta \vdash \Theta} \qquad \frac{\Gamma, A, A \vdash \Theta}{\Gamma, A \vdash \Theta} \qquad \frac{\Gamma \vdash \Theta}{\Gamma, A \vdash \Theta}$$

- identity and cut:

$$\frac{}{A \vdash A} \qquad \frac{\Gamma \vdash A \quad \Delta, A \vdash \Theta}{\Gamma, \Delta \vdash \Theta}$$

- rules of negation:

$$\frac{\Gamma, A \vdash}{\Gamma \vdash \neg A} \qquad \frac{\Gamma \vdash A}{\Gamma, \neg A \vdash}$$

- rules of conjunction:

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B} \qquad \frac{\Gamma, A \vdash \Theta}{\Gamma, A \wedge B \vdash \Theta} \qquad \frac{\Gamma, B \vdash \Theta}{\Gamma, A \wedge B \vdash \Theta}$$

- rules of disjunction:

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \vee B} \qquad \frac{\Gamma \vdash B}{\Gamma \vdash A \vee B} \qquad \frac{\Gamma, A \vdash \Theta \quad \Gamma, B \vdash \Theta}{\Gamma, A \vee B \vdash \Theta}$$

Of course, this calculus satisfies the *Hauptsatz*, *i.e.* the cut elimination property. The negation here is nonclassical in the sense that $\neg\neg P \vdash P$ is not provable, which is obvious by cut elimination. In fact, it is even minimal in the sense that $P, \neg P \vdash Q$ is not provable, again by cut elimination, since there is no right weakening. On the other hand, $A, \neg A \vdash \neg B$ is always provable as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{A \vdash A}}{A, \neg A \vdash}}{A, \neg A, B \vdash}}{A, \neg A \vdash \neg B}$$

2.2 Standard interpretation of $\mathbf{LJ}^-$

$\mathbf{LJ}^-$ can be seen as a subsystem of **LJ** without negation nor sequents of the form $\Gamma \vdash$. Fix indeed an arbitrary formula Φ and consider $\neg A$ as a notation for $A \Rightarrow \Phi$. Similarly, consider $\Gamma \vdash$ as a notation for $\Gamma \vdash \Phi$. The rules of 2.1 are obviously sound for this interpretation. In particular, the rules of negation are interpreted as follows:

$$\dfrac{\Gamma, A \vdash \Phi}{\Gamma \vdash A \Rightarrow \Phi} \qquad \dfrac{\Gamma \vdash A \quad \dfrac{}{\Phi \vdash \Phi}}{\Gamma, A \Rightarrow \Phi \vdash \Phi}$$

Conversely it is possible to show that, if Φ is an atom not occurring in a sequent $\Gamma \vdash \Theta$, and the interpretation of this sequent is provable in **LJ**, then $\Gamma \vdash \Theta$ is provable in $\mathbf{LJ}^-$.

2.3 Translation of classical logic

Now we shall see that $\mathbf{LJ}^-$ is quite powerful, since classical logic can be embedded into it. First we describe the translation of classical formulae into $\mathbf{LJ}^-$. For this purpose, we choose a *polarity* $\varepsilon(P) \in \{+, -\}$ for each atom P, and we define $|P|$ as

$$P \quad \text{if } \varepsilon(P) \text{ is } +, \qquad \neg P \quad \text{if } \varepsilon(P) \text{ is } -.$$

These notions are extended to all classical formulae, according to the following tables:

$\varepsilon(A)$	$\varepsilon(\neg A)$	$\|\neg A\|$
$+$	$-$	$\|A\|$
$-$	$+$	$\|A\|$

$\varepsilon(A)$	$\varepsilon(B)$	$\varepsilon(A \wedge B)$	$\lvert A \wedge B \rvert$
$+$	$+$	$+$	$\lvert A\rvert \wedge \lvert B\rvert$
$+$	$-$	$+$	$\lvert A\rvert \wedge \neg\lvert B\rvert$
$-$	$+$	$+$	$\neg\lvert A\rvert \wedge \lvert B\rvert$
$-$	$-$	$-$	$\lvert A\rvert \vee \lvert B\rvert$

$\varepsilon(A)$	$\varepsilon(B)$	$\varepsilon(A \vee B)$	$\lvert A \vee B \rvert$
$+$	$+$	$+$	$\lvert A\rvert \vee \lvert B\rvert$
$+$	$-$	$-$	$\neg\lvert A\rvert \wedge \lvert B\rvert$
$-$	$+$	$-$	$\lvert A\rvert \wedge \neg\lvert B\rvert$
$-$	$-$	$-$	$\lvert A\rvert \wedge \lvert B\rvert$

$\varepsilon(A)$	$\varepsilon(B)$	$\varepsilon(A \Rightarrow B)$	$\lvert A \Rightarrow B \rvert$
$+$	$+$	$-$	$\lvert A\rvert \wedge \neg\lvert B\rvert$
$+$	$-$	$-$	$\lvert A\rvert \wedge \lvert B\rvert$
$-$	$+$	$+$	$\lvert A\rvert \vee \lvert B\rvert$
$-$	$-$	$-$	$\neg\lvert A\rvert \wedge \lvert B\rvert$

Finally, we say that A is *positive* if $\varepsilon(A)$ is $+$, *negative* if $\varepsilon(A)$ is $-$, and we define the translation $\langle A\rangle$ of a classical formula A as

$$|A| \quad \text{if } A \text{ is positive,} \qquad \neg|A| \quad \text{if } A \text{ is negative.}$$

Proposition 1 *If $A_1, \ldots, A_n, B_1, \ldots, B_m$ are classical formulae then*

$$A_1, \ldots, A_n \vdash B_1, \ldots, B_m$$

is provable classically if and only if

$$\langle A_1\rangle, \ldots, \langle A_n\rangle, \langle \neg B_1\rangle, \ldots, \langle \neg B_m\rangle \vdash$$

is provable in $\mathbf{LJ}^-$.

Proof: It is clear that $\langle A\rangle$ is classically equivalent to A, and since $\mathbf{LJ}^-$ is a subsystem of **LK**, the condition is sufficient. The converse is proved by induction on classical proofs, considering all possible combinations of polarities. *Q.e.d.*
Unlike usual double-negation translations, this one is compatible with substitution (as long as polarities are respected) and it preserves the usual canonical isomorphisms (see [Girard91]). It is also "economical" in the sense that negations are only introduced as a last resort. However, it cannot be compatible with cut elimination, since it encodes a nonconstructive logic into a constructive one (see [GLT89]).

3 Lambda calculus

3.1 Kernel language

Of course, it is possible to express $\mathbf{LJ}^-$ in a *natural deduction style*. As usual, the rules of disjunction are rather unpleasant (see [GLT89]) and therefore, we shall consider the fragment where negation and conjunction are the only connectives.

A proof of a sequent $A_1, \ldots, A_n \vdash \Theta$ will be represented by

$$x_1 : A_1, \ldots, x_n : A_n \vdash t : \Theta$$

where t is a term with (possibly not occurring) free variables $x_1, \ldots, x_n$. If Θ is empty, we say that t is a *command*, and we simply write

$$x_1 : A_1, \ldots, x_n : A_n \vdash t.$$

The rules of deduction can be seen as typing rules for lambda calculus with explicit pairs:

- variable:

$$\frac{}{\Gamma,\, x : A \vdash x : A} \qquad \frac{\Gamma \vdash x : A}{\Gamma,\, y : B \vdash x : A}$$

- abstraction and application:

$$\frac{\Gamma,\, x : A \vdash u}{\Gamma \vdash \lambda x.u : \neg A} \qquad \frac{\Gamma \vdash u : \neg A \quad \Gamma \vdash v : A}{\Gamma \vdash u\, v}$$

- pairing and projections:

$$\frac{\Gamma \vdash u : A \quad \Gamma \vdash v : B}{\Gamma \vdash (u, v) : A \wedge B}$$

$$\frac{\Gamma \vdash u : A \wedge B}{\Gamma \vdash \pi^1 u : A} \qquad \frac{\Gamma \vdash u : A \wedge B}{\Gamma \vdash \pi^2 u : B}$$

In all those rules, Γ stands for an arbitrary sequence $x_1 : A_1, \ldots, x_n : A_n$. In the second one, x and y are distinct variables.
Cut elimination is expressed by the usual conversions:

$$(\lambda x.u)\, v \to u[v/x], \qquad \pi^1(u, v) \to u, \qquad \pi^2(u, v) \to v.$$

As usual, we shall write $t \to^* t'$ if t reduces to t' in several steps of reduction.

3.2 Translation of lambda calculus

Now, we shall see what happens to the translation of 2.3 when it is restricted to intuitionistic proofs, expressed in a natural deduction style. In fact, we shall even restrict ourselves to the implicational fragment of intuitionistic logic.

First we notice that $A \Rightarrow B$ is negative whenever both A and B are negative. Consequently, if we choose a negative polarity for each atom, all the formulae of the implicational fragment are negative. This means that, if A is such a formula, then $\langle A \rangle$ is $\neg|A|$ and $\langle \neg A \rangle$ is $|A|$, where $|A|$ is inductively defined by:

$$|P| = \neg P, \qquad |A \Rightarrow B| = \neg|A| \wedge |B|$$

Therefore, a proof of the sequent $A_1, \ldots, A_n \vdash B$ in this fragment must be translated into a proof of

$$\neg|A_1|, \ldots, \neg|A_n|, |B| \vdash$$

in $\mathbf{LJ}^{\neg}$. Now, assume that the original proof is represented by a lambda term t with (possibly not occurring) free variables $x_1, \ldots, x_n$. Then its translation will be represented by a command $\overline{t}$ with one extra free variable y, which has to be thought of as the current continuation, possibly using pairing and projections. In $\overline{t}$, the x_i is of type $\neg|A_i|$, and y is of type $|B|$.
In fact, for any pure lambda term t without free occurrence of the variable y, the translation $\overline{t}$ can be inductively defined as follows:

- $\overline{x} \;=\; x\,y$
- $\overline{\lambda x.u} \;=\; \overline{u}[\pi^1 y/x;\, \pi^2 y/y]$
- $\overline{u\,v} \;=\; \overline{u}[(\lambda y.\overline{v}, y)/y]$

In the second case, x and y must be substituted in parallel.
This translation is compatible with β-conversion:

Proposition 2 *Let t be a pure lambda term:*

i) If t is in normal form, so is $\overline{t}$.

ii) If t reduces to t' then $\overline{t}$ reduces to $\overline{t'}$.

Proof: By definition

$$\overline{(\lambda x.u)\,v} \;=\; \overline{u}[\pi^1(\lambda y.\overline{v}, y)/x;\, \pi^2(\lambda y.\overline{v}, y)/y]$$

in which $\pi^1(\lambda y.\overline{v}, y)$ and $\pi^2(\lambda y.\overline{v}, y)$ are convertible. Clearly, there is no other way to create a *redex* in the translation of a lambda term, and so we have (i).
Since reduction is compatible with substitution, (ii) amounts to showing

$$\overline{(\lambda x.u)\,v} \rightarrow^* \overline{u[v/x]}$$

for any two pure lambda terms u and v. But if we apply the appropriate conversions to $\pi^1(\lambda y.\overline{v}, y)$ and $\pi^2(\lambda y.\overline{v}, y)$, we already get

$$\overline{(\lambda x.u)\,v} \rightarrow^* \overline{u}[\lambda y.\overline{v}/x].$$

It remains to see that

$$\overline{u}[\lambda y.\overline{v}/x] \rightarrow^* \overline{u[v/x]}$$

which is easily proved by induction on u. *Q.e.d.*

3.3 Extended language

Now we shall enrich the kernel language of 3.1 with recursion and primitives for arithmetics, to get an analogue of the language PCF of Gordon Plotkin. For recursion, we could take the usual rule:

$$\frac{\Gamma,\, x : A \vdash u : A}{\Gamma \vdash \mu x.u : A}$$

But in fact, this is more than needed, and we shall content ourselves with the following one:

$$\frac{\Gamma,\, x : \neg A,\, y : A \vdash u}{\Gamma \vdash \rho\, x\, y.u : \neg A}$$

which is an instance of the first rule if we define

$$\rho\, x\, y.u \;=\; \mu x.\lambda y.u.$$

The conversion for our restricted recursion is:

$$(\rho\, x\, y.u)\, v \to u[\rho\, x\, y.u/x;\; v/y]$$

For zero, successor, predecessor and branching, we introduce an atomic type N of natural numbers, and we adopt the following rules:

$$\frac{}{\Gamma \vdash 0 : N} \qquad \frac{\Gamma \vdash u : N}{\Gamma \vdash s\, u : N} \qquad \frac{\Gamma \vdash u : N}{\Gamma \vdash p\, u : N}$$

$$\frac{\Gamma \vdash u : N \qquad \Gamma \vdash v \qquad \Gamma \vdash w}{\Gamma \vdash \beta\, u\, v\, w}$$

Note that our branching is restricted to commands. The corresponding conversions are:

$$p\, 0 \to 0, \qquad p\,(s\, u) \to u,$$
$$\beta\, 0\, v\, w \to v, \qquad \beta\,(s\, u)\, v\, w \to w.$$

Finally, we introduce a one element type **U** with just one typing rule:

$$\frac{}{\Gamma \vdash * : U}$$

and no conversion rules.

From now on, we shall call this extended language BCL (for *Basic Continuation Language*). In BCL, computing values is quite trivial:

Proposition 3 *If* $\vdash t : A$ *is derivable in BCL, then* t *is reducible to a head normal form*

$$\lambda x.u, \quad (u,v), \quad \rho\, x\, y.u, \quad 0, \quad s\, u \quad \textit{or} \quad *$$

by means of the following size decreasing conversions:

$$\pi^1(u,v) \to u, \qquad \pi^2(u,v) \to v,$$

$$p\,0 \to 0, \qquad p\,(s\,u) \to u,$$

Proof: By induction on the size of t. If t is not itself in head normal form, it must of the form $\pi^1 u$, $\pi^2 u$ or $p\,u$, and we can apply the induction hypothesis to u. In the first case, for instance, u reduces to a head normal form (v,w), so that t reduces to v, which itself reduces to a head normal form by induction hypothesis. The other cases are similar. *Q.e.d.*
Note that this property would not hold if $\mu x.u$ were adopted for recursion.

3.4 Inputs and outputs

BCL itself can be smoothly extended with features of questionable status in functional programming languages, such as inputs and outputs. To simplify, we assume that all input and output values are of type N, and we introduce the following rules:

$$\frac{\Gamma, x : N \vdash u}{\Gamma \vdash \iota\, x.u} \qquad \frac{\Gamma \vdash u : N \quad \Gamma \vdash v}{\Gamma \vdash o\, u\, v}$$

The intended meaning is:

- for $\iota\, x.u$, read the first value from the input channel, and then execute u with x instantiated by this value,
- for $o\, u\, v$, write the value of u on the output channel, and then execute v.

Of course, the execution order is crucial in both cases: In $o\, u\, v$ for instance, v should not be executed before u is printed.
There is no conversion for $\iota\, x.u$ or $o\, u\, v$, because all three express a behaviour rather than a computation. Nevertheless, as we shall see, this behaviour can be made completely explicit by means of an appropriate semantics.

4 Semantics for BCL

In this section we shall give a direct semantics to BCL.

The interpretation of **N** are the natural numbers N which come with the standard functions 0, *succ*, and *pred*, where $pred(0) = 0$. The interpretation of the unit type **U** is the type U with one element, namely $*$.
$Stream(N)$ is the domain of all finite and infinite sequences of natural numbers with the prefix ordering. The operation <> denotes the empty sequence, & the append function, $first$ and $rest$ the projection on the first element and on the rest of the stream, respectively.
The denotations of commands live in the domain $Stream(N) \to Stream(N)$. The idea is that the behaviour of a command is determined by its input/output behaviour (i.e. a continuous function from $Stream(N)$ to $Stream(N)$).
Let $[\![\_]\!]$ denote the intepretation function for BCL which is inductively defined by the following equations:

- $[\![0]\!]\,\gamma = 0,\ [\![p]\!]\,\gamma = pred,\ [\![s]\!]\,\gamma = succ$
- $[\![(a,b)]\!]\,\gamma = \langle [\![a]\!]\gamma, [\![b]\!]\gamma \rangle$
- $[\![\pi^i(u)]\!]\,\gamma = pr_i([\![u]\!]\,\gamma)$
- $[\![u\,v]\!]\,\gamma = ([\![u]\!]\,\gamma)([\![v]\!]\,\gamma)$
- $[\![\lambda x.u]\!]\,\gamma\ = \lambda a.\,[\![u]\!]\,\gamma[a/x]$
- $[\![\rho\,x\,y.u]\!]\,\gamma =$ the least g such that $g\ = \lambda a.[\![u]\!]\,\gamma[g/x;\ a/y]$,
- $[\![\beta\,u\,v\,w]\!]\,\gamma = \begin{cases} v & \text{if } [\![u]\!]\,\gamma = 0 \\ w & \text{if } \exists k{:}N.\ [\![u]\!]\,\gamma = succ(k) \end{cases}$
- $[\![*]\!]\,\gamma = *$

For the input/output features we formalize the intended meanings of Section 2.4.:

- $[\![\iota\,x.u]\!]\,\gamma = \lambda s{:}Stream(N).\ [\![u]\!]\,(\gamma[first(s)/x])(rest\ s)$
- $[\![o\,u\,v]\!]\,\gamma = \lambda s{:}Stream(N).([\![u]\!]\,\gamma)\ \&\ ([\![v]\!]\,\gamma)(s)$

Since ρ only works on domains of type $\neg A$, i.e. $A \Rightarrow \Phi$, it is sufficient for the basic types to be predomains as if Φ is a domain, $A \to \Phi$ is a domain, too, even if A is a predomain. Moreover, we use the λ-sign as part of BCL, i.e. on the syntactical level and on the right hand side of the equations, i.e. on the semantical level.

5 PCF with Control Features and Input/Output

We present an extension of PCF [Plotkin 77] by control and input/output operations. For continuation style programming we will need the type *Beh*, which stands for *behaviour*. From the logical point of view *Beh* corresponds to $\perp$ and the $\mathcal{C}$ operator corresponds to the proof term for the proposition $\neg\neg\varphi \to \varphi$ valid in classical logic.

5.1 PCF^{ext}-Syntax

Let PCF^{ext}denote the extension of PCF by continuation features and input/output. The context-free language reads as follows:

5.1.1 Types

- Basic type expressions: *Nat*, *Beh*.
- Type forming operation: If σ and τ are types then $\sigma \to \tau$ is a type, too.

5.1.2 Terms

- Basic expressions:

$$0 : Nat,\ \ succ,\, pred : Nat \to Nat,\ \ read : Nat$$

- Compound expressions: Let e,e_1,e_2 be terms, then the following expressions are terms:

$$(e_1\, e_2),\ \ \lambda x{:}\sigma.\, e,\ \ Y,\ \ cond(e, e_1, e_2),\ \ \mathcal{C}e,\ \ write\, e$$

5.2 Typing Rules

Well-formed terms are defined together with their types by the following rules:

$$\frac{\Gamma \vdash e_1 : \sigma \to \tau \quad \Gamma \vdash e_2 : \sigma}{\Gamma \vdash (e_1 e_2) : \tau} \qquad \frac{\Gamma, x : \tau \vdash e : \sigma}{\lambda x{:}\tau.e : \tau \to \sigma}$$

$$\frac{}{\Gamma \vdash Y : (\sigma \to \sigma) \to \sigma} \qquad \frac{\Gamma \vdash e : Nat \quad \Gamma \vdash e_1, e_2 : \tau}{\Gamma \vdash cond(e, e_1, e_2) : \tau}$$

$$\frac{\Gamma \vdash e : (\sigma \to Beh) \to Beh}{\Gamma \vdash \mathcal{C}e : \sigma} \qquad \frac{\Gamma \vdash e : Nat}{\Gamma \vdash write\ e : Beh \to Beh}$$

6 Translating PCF^{ext} into BCL

We give a first interpretation of PCF^{ext} by translating it into BCL. This translation extends the translation of simply typed λ-calculus to BCL (see Section 2.2) and therefore we only have to give the defining clauses for the constructs of PCF^{ext} not yet contained in the λ-calculus.

- $\overline{0} = y\,0$

- $\overline{succ} = (\pi^1 y)(\lambda n.(\pi^2 y)(s\, n))$
- $\overline{pred} = (\pi^1 y)(\lambda n.(\pi^2 y)(p\, n))$
- $\overline{Y} = (\rho\, u\, v.(\pi^1 y)(u, v))(\pi^2 y)$
- $\overline{cond(e, e_1, e_2)} = \overline{e}[\lambda n.\, \beta\, n\, (\overline{e_1}\, y)\, (\overline{e_2}\, y)/y]$
- $\overline{\mathcal{C}t} = \overline{t}[(\lambda p.(\pi^1 p)\, y, *)/y]$

For input/output we define:

- $\overline{read} = \iota\, x.y\, x$
- $\overline{write\ e} = \overline{e}[\lambda n.o\, n\, ((\pi^1 y)\, *)/y]$

7 Continuation Semantics of PCF^{ext}

For modelling the behaviour of programs we have to choose an interpretation of *Beh*. As we want to express input/output behaviour the natural choice for the interpretation of *Beh* will be $B := \mathit{Stream}(N) \to \mathit{Stream}(N)$, i.e. the same domain that we have used for interpreting commands in BCL.

7.1 Continuations

The interpretation function on terms is given by a (family of) semantic function

$$[\![\_]\!]_\sigma : PCF^{ext}_\sigma \to Env \to C_\sigma \to B$$

where *Env* is the environment that assigns to every variable a semantic object of correct type. We have to define what the type C_σ, called the *continuations of type* σ, is:

- $C_{Beh} = U$
- $C_{Nat} = N \to B$
- $C_{\sigma\to\tau} = (C_\sigma \to B)\times C_\tau$

Note that we have the following isomorphisms:

$$[\![\sigma \to \tau]\!] = C_{\sigma\to\tau} \to B = (C_\sigma \to B) \times C_\tau \to B \simeq$$

$$\simeq (C_\sigma \to B) \to (C_\tau \to B) = [\![\sigma]\!] \to [\![\tau]\!]$$

7.2 Interpreting terms

The following semantic equations derive from translating PCF^{ext} to BCL and then interpreting BCL.
For the standard part we have

- $[\![x]\!]_\sigma\ \gamma\ k = \gamma(x)$
- $[\![0]\!]_{Nat}\ \gamma\ k = k\ 0$
- $[\![succ]\!]_{Nat \to Nat}\ \gamma\ \langle k, f \rangle = k(\lambda n{:}N. f(succ\ n))$
- $[\![pred]\!]_{Nat \to Nat}\ \gamma\ \langle k, f \rangle = k(\lambda n{:}N. f(pred\ n))$
- $[\![(t\, s)]\!]_\sigma\ \gamma\ k = [\![t]\!]_{\tau \to \sigma}\ \gamma\ \langle [\![s]\!]_\tau\, \gamma\, ,\ k \rangle$
- $[\![\lambda x{:}\tau.t]\!]_{\tau \to \sigma}\ \gamma\ \langle m, k \rangle = [\![t]\!]_\sigma\ \gamma[m/x]\ k$
- $[\![Y]\!]_{(\sigma \to \sigma) \to \sigma}\ \gamma$ is the least F such that

$$F\langle f, k \rangle = f\langle \lambda k' : C_\sigma.\, F\langle f, k' \rangle, k \rangle \quad ^{1}$$

- $[\![cond(e, e_1, e_2)]\!]_\sigma\, \gamma\, k = [\![e]\!]_{Nat}\, \gamma\, \lambda n{:}N.\ \mathit{if}\, n = 0\ \mathit{then}\ ([\![e_1]\!]_\sigma\, \gamma\, k)\ \mathit{else}\ ([\![e_2]\!]_\sigma\, \gamma\, k)$

The semantics of $\mathcal{C}$ is given by the following equation:

- $[\![\mathcal{C}t]\!]_\sigma\ \gamma\ k = [\![t]\!]_{(\sigma \to Beh) \to Beh}\ \gamma\ \langle \lambda p : (C_\sigma \to B) \times U.\, (\pi_1\, p)k\, ,\ * \rangle$

For input/output we define:

- $[\![read]\!]_{Nat}\ \gamma\ k = \lambda s{:}Stream(N).\, k\, (first\ s)\, (rest\ s)$
- $[\![write\ e]\!]_{Beh \to Beh}\ \gamma\ \langle f, * \rangle = [\![e]\!]_{Nat}\, \gamma\, (\lambda n{:}N. \lambda s{:}Stream(N).\ n\, \&\, (f\ *\ s))$

The meaning of a term of type σ is a function with domain C_σ and codomain B. To assign to such a term an input/output behaviour one simply has to apply it to a "canonical" continuation of the appropriate type. This will be done by introducing *standard continuations* and defining

$$[\![t]\!]_\sigma^{Prog} = [\![t]\!]_\sigma\ (st(\sigma))$$

where $st(\sigma) \in C_\sigma$, the standard continuation, is defined as follows:

- $st(Beh) = *$
- $st(Nat) = \lambda n{:}N.\ \lambda s{:}Stream(N).\ <>$
- $st(\sigma \to \tau) = \langle \lambda y{:}C_\sigma.\, \lambda s{:}Stream(N).\ <>\, , st(\tau) \rangle$

---

[1] This could be also written as
$\mu(\lambda F.\, \lambda p{:}(C_{\sigma \to \sigma} \to B) \times C_\sigma.\, (\pi_1\ p)\langle \lambda k'.F\langle \pi_1(p), k' \rangle, \pi_2(p) \rangle)$

8 Examples

The main motivation for the control operator $\mathcal{C}$ is to express "jumps" within functional programs. The typical use of "jumps" is *raising* of errors and *handling* of errors.

8.1 Raising an error

For raising of errors we use the PCF^{ext} term $\in Beh \to Beh$:

$$\mathit{raise} \equiv_{def} \ \lambda b : Beh.\ \mathcal{C}(\lambda f : \sigma \to Beh.\ b)$$

This corresponds to Felleisen's operator $\mathcal{A}$.
Notice that the variables of type *Beh* in PCF^{ext} play a role corresponding to error elements in, say SML (see [MTH90]).

8.2 Error Handling

If e and e' are of type σ and err is a variable of type *Beh*, then the intuitive meaning of the term $\in Beh$

$$\mathit{handle}\ err\ \mathit{in}\ e\ \mathit{by}\ e'$$

is that whenever in the evaluation of e the error err is raised then evaluate e' instead.
In PCF^{ext} this can be expressed as

$$\mathit{handle}\ err\ \mathit{in}\ e\ \mathit{by}\ e' \equiv_{def} \mathcal{C}_\sigma(\lambda f{:}\sigma \to Beh. f((\lambda err{:}Beh.\ e)(fe')))$$

One could equivalently extend the syntax of PCF^{ext} by introducing a construct "*handle err in e by e*$'$" and giving the semantic equation:

$$[\![\mathit{handle}\ err\ \mathit{in}\ e\ \mathit{by}\ e']\!]_\sigma\ \gamma\ k = [\![e]\!]_\sigma\ \gamma[([\![e']\!]_\sigma\ \gamma\ k)/err]\ k$$

The reader may convince himself that this yields the same result as the implementation via $\mathcal{C}$ above.

8.3 Further Examples

An operation "_; _" of type $(Beh \to Beh) \times \sigma \to \sigma$ can be defined in the following way:

$$e_1\,;\,e_2 =_{def} \mathcal{C}(\lambda f{:}\sigma \to Beh.\ e_1\,(f\ e_2))$$

It can be used for programming side-effects, e.g. the program "*write 0;* $f(x)$" prints 0 and yields $f(x)$ as a result.

The following program prints all natural numbers:

$$Y(\lambda b : Nat \to Beh.\, \lambda n : Nat.\ (write\, n)\,(b\,(succ\, n)))\, 0$$

Using "_; _" one could alternatively formulate this as

$$Y(\lambda b : Nat \to Beh.\, \lambda n : Nat.\ write\, n\, ;\, b\,(succ\, n))\, 0.$$

References

[FFKD86] M. Felleisen, D. Friedman, E. Kohlbecker & B. Duba, Reasoning with continuations, in *Proceedings of the First IEEE Symposium on Logic in Computer Science*, pp. 131–141 (1986).

[Fisher72] M. J. Fischer, Lambda calculus schemata, in *Proceedings of ACM Conference on Proving Assertions About Programs*, pp. 104–109 (1972).

[Gallier91] J. Gallier, *Constructive Logics. Part I: A tutorial on Proof Systems and Typed λ-Calculi*, Research Report Digital PRL 8 (1991).

[Gentzen69] *The collected Papers of Gerhard Gentzen*, E. Szabo, ed. (North-Holland, 1969).

[GLT89] J.Y. Girard, Y. Lafont & P. Taylor, *Proofs and Types*, Cambridge Tracts in Theoretical Computer Science 7 (Cambridge University Press, 1989).

[Girard91] J.Y. Girard, A new constructive logic: classical logic, *Mathematical Structures in Computer Science* 1, pp. 255–296 (Cambridge University Press, 1991).

[Griffin90] T.G. Griffin, A Formulae-as-Types Notion of Control, in *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Programming Languages* pp. 47–58 (1990).

[MeWa85] A. R. Meyer & M. Wand, Continuation semantics in typed lambda-calculi (summary) in R. Parikh, editor, *Logics of Programs*, LNCS 193, pp. 219–224 (Springer-Verlag, 1985).

[MTH90] R. Milner, M. Tofte, R. Harper: *The Definition of Standard ML.* The MIT Press (1990).

[Murthy91] C.R. Murthy, An Evaluation Semantics for Classical Proofs, in *Proceedings of the Sixth Annual IEEE Symposium on Logic in Computer Science*, pp. 96–107 (1991).

[Plotkin 77] G. Plotkin: LCF considered as a programming language. *Theoretical Computer Science* 5(3), pp. 223-256 (North Holland, 1977).