Modeling of Call – By – Need and Stream Primitives using CCS

UUCS – 82 – 015
August 1982

by

S.Purushothaman

P.A.Subrahmanyam

Department of Computer Science
University of Utah
Salt Lake City, Utah 84112

Abstract

The semantics of an applicative language are presented using the algebraic primitives introduced in CCS. In particular, the language constructs modeled allow for nondeterminism, stream processing and demand driven (call by need) evaluation.

Sponsored by Defense Advanced Research Projects Agency, US Department of Defense, Contract No. MDA903 – 81 – C – 0414.

Table of Contents

List of Figures

1. Introduction

The study of semantics of programming languages offers an opportunity to abstract away from the details of their implementation and work in a mathematical framework. The basic problems involved in the description of an imperative language have been fairly well identified and solutions have been proposed with a reasonable degree of success. The problems involved in concurrent processing and non-determinacy have been identified but modeling such flavors of programming constructs have had less success in the traditional settings of denotational (continuation) semantics. Details like several possible continuations, and the possibility of future actions of concurrent processes depending on value communication between them, give rise to some of the tougher problems in giving semantics for concurrent and non-deterministic programs. Most of the problems are explicated/referred to in [1]; however, even with the powerful semantic domain of multidomains, the problems of low level synchronization and value passing cannot seemingly be modeled elegantly. In contrast we believe that the domain of processes/behavior expressions proposed in [6] offers a better semantic domain for expressing such low level details.

One of the major advantages of an applicative language is the use of referential transparency in the development or verification of programs. In introducing non-determinism in an applicative language, we lose referential transparency. In addition to modeling low-level details of call-by-need, our motivation is to develop a formalism in which non-deterministic programs are referentially transparent (at least to a certain degree, [2] refers to them as referentially translucent). In contrast to [2], where the authors are guided by considerations of source-to-source transformations, our aim is to express low-level details of implementation and address the question of referential transparency in the same framework.

In this paper we attempt to give semantics for the core of an applicative language using CCS -- a calculus for communicating systems introduced by Milner [6]; we also show the ease with which call-by-need can be explained in this context. The material is organized as follows: in section 2 we explain the idea of behavior expressions and in section 3 we give semantics for an applicative language. The development in section 3 first considers a language that allows only nondeterministic primitives; this is subsequently extended to include stream oriented primitives.

As an example of an important practical application, we remark that the basic scheme presented is of relevance in an algebraic framework for supporting VLSI design. In essence, the abstract high level specification of a chip can be viewed as a black box that accepts (appropriately structured) streams of data at its input ports and provides the desired output streams at its output ports. The functionality provided by the labeled ports of the chip may be suitably expressed by using extended algebraic data types that use parameterized behavior expressions. The evaluation strategies presented here can then be used to model an internal (lower level) implementation of the chip that is intrinsically asynchronous in nature. The ability to do this then allows a rich set of implementation possibilities to be considered.

2. What is CCS?

In this section we outline an intuitive view of CCS (for the algebraic details of CCS, the reader is referred to [6]). CCS has two facets that essentially reflect the static and the dynamic nature of processes. The static part of CCS reflects the fact that the mathematical object "process" can be looked upon as an agent associated with a set of labels/ports via which it can communicate with the external world. An agent can be viewed as a black box, with the labels (ports) providing the sole means of performing experiments on the agent. In characterizing an agent, the the sort set of the

agent is the set of labels/ports. The dynamic behavior is given by behavior expression, which essentially embodies (1) the possible communications of values between agents or between an agent and an observer, and (2) the function of the agent. The agents are amenable to composition, the composed agent now offers experiments on the labels of either of its constituent agents which can additionally communicate among themselves. As a convention, internal communication is only possible between labels which are opposite in nature/co−names of each other. The names and co−names are in bijection with each other and this is reflected by defining co−name($\bar{\alpha}$) = α and co−name(a) = $\bar{\alpha}$. For example, if the labels/ sort set of agent P_1 is $\{\alpha,\bar{\beta},\gamma\}$ and that of P_2 is $\{o,\beta,\omega\}$, the sort set of the composed agent would be $\{\alpha,\bar{\beta},\beta,\gamma,o,\omega\}$ and internal communication can take place only between β and $\bar{\beta}$.

Denoting the composition operator by the symbol " |", the arity[1] of " |" is

$$| : \text{agent of sort } S_1 \text{ X agent of sort } S_2 \text{ -> agent of sort } (S_1 \cup S_2).$$

It can be easily shown that the operator | is associative.

In order to internalize communication, the labels of an agent can be hidden/restricted. Once a label is restricted, that port is no longer available for external communication/experimentation. The restriction operator is denoted by the symbol "\". In the above example (P1 | P2)\β would remove both β and $\bar{\beta}$ from the sort set of the combined agent.

The arity of the operator \ is

$$\backslash : \text{agent of sort } S_1 \text{ X agent of sort } S_2 \text{ -> agent of sort } (S_1 - S_2)$$
where S_2 is the set of labels which are hidden away.

Another operation on the agents is relabeling (of ports) and is intuitive. Relabeling is denoted "/". For example, $S[l_1 / l_2]$ refers to relabeling l_2 by l_1 in the sort set for the agent S.

2.1. Dynamic Behavior

The dynamic behavior of an agent is captured by a behavior expression (defined below), while the static behavior of an agent consists in viewing the agent as a black box. The definitions of these behavior expressions can be recursive and parameterized. The sort set of an agent now becomes the primitives for value communication in its behavior expression. It is through the elements of this sort set that a behavior expression communicates with its environment.

The behavior expressions can be intuitively defined as follows:

- The process represented by the behavior expression αu.B, expects a value at the port α which is bound to the variable u. The scope of the variable u is the behavior expression B.
- The process represented by the behavior expression $\bar{\alpha}$ e.B outputs a value e at the port $\bar{\alpha}$ and the ensuing behavior is B.
- The process represented by the behavior expression α.B expects a signal at α. The ensuing behavior is B.
- The process represented by the behavior expression $\bar{\alpha}$.B outputs a signal at the port $\bar{\alpha}$ and the ensuing behavior is B.
- The behavior expression NIL represents a null action; NIL therefore may be thought of

[1]The arity of a function denotes the names of the sets comprising its domains and ranges.

as a terminated process.

- The process represented by the behavior expression (if x then B1 else B2) selects either B1 or B2 depending on the value of x.

Akin to CSP [4], value communication takes place between complementary ports on encountering a matching input/output action pair and synchronization has to take place before the processes can proceed. When there are a multitude of processes waiting to receive a value from the same output port, the decision to communicate the value to one of the waiting processes is made arbitrarily.

The operations on the behavior expressions are as follows:

- *The null action*: Denoted by the operator NIL.
- *Choice Operation*: Denoted by the operator +. Introduces non-determinacy. In B1 + B2 where B1 and B2 are behavior expressions, the experiments offered by both B1 and B2 are the possible experiments, now offered. On acceptance of an experiment, (say) offered by B1, the ensuing behavior expression is B1', where B1' is the mutated behavior of B1. Obviously the experiments offered by B2 now disappear.
- *Composition of agents*: Denoted by the operator |. Composes two agents to form another agent, as characterized earlier.
- *Restriction*: Denoted by $\backslash\alpha$, where α and $\bar{\alpha}$ are the labels to be restricted.
- *Relabeling*: α/β denotes relabeling of a label β by α.

In order to motivate the use of CCS we will describe a register using CCS.

The behavior expression for a register [6] can be given as

$$\begin{aligned} LOC &<= \alpha u.REG(u) \\ REG(u) &<= \alpha w.REG(w) + \bar{\beta}\, u.REG(u) \end{aligned}$$

The intuitive meaning is that the register can be initialized to a value u by an α experiment. Once the register has been initialized there are two possible experiments on the register, α and $\bar{\beta}$. α is the input port at which a value w is accepted and the register contents transformed to w, whereas $\bar{\beta}$ is the output port which is ready to output the value u, following which the register continues with unchanged behavior. The + operator explicates the non-determinacy possible between storing a value and accessing the value from the register.

Let us stretch this example a bit further and consider the composition of the register and an agent performing some experiments on it. Let the program (closed behavior expressions, explained later) be

$$Expr <= \bar{\alpha}\, 6.\bar{\alpha} 5.\beta w.\bar{o}\, w.NIL$$

The expression Expr does a series of experiments on the register. It outputs values 6 and 5 at its $\bar{\alpha}$ port and then demands a value at port β. The value received is output to the environment at the $\bar{o}$ port.

Consider now the system composition:

$$Ex = (Expr \mid LOC)\backslash\alpha\backslash\beta$$

The sort set of Ex is $\{\bar{o}\}$. For an external observer the compound agent offers a value at $\bar{o}$ and then dies.

The behavior of the compound agent can be expanded to

$(Expr \mid LOC)\backslash\alpha\backslash\beta$

$= \{ (\bar{\alpha}6.\ \bar{\alpha}5.\ \beta w.\bar{o}\ w.NIL) \mid LOC \}\backslash\alpha\backslash\beta$

$= \{ (\bar{\alpha}6.\ \bar{\alpha}5.\ \beta w.\bar{o}\ w.NIL) \mid (\alpha u.REG(u)) \}\backslash\alpha\backslash\beta$

$= \tau.\{ (\bar{\alpha}5.\ \beta w.\bar{o}\ w.NIL) \mid REG(6) \}\backslash\alpha\backslash\beta$

what has happened is that the value 6 has been communicated from Expr to the register. The communication is internal and is represented by the τ action. *Thus, the derived behavior expression denotes that an internal communication has taken place, but conveys no more information.*

Further expansion of the behavior expression is as follows:

$= \tau.\tau \{ (\beta w.\bar{o}\ w.NIL) \mid REG(5) \}\backslash\alpha\backslash\beta$

$= \tau.\tau \{ (\beta w.\bar{o}\ w.NIL) \mid (\alpha w.REG(w) + \bar{\beta}5.REG(5)) \}\backslash\alpha\backslash\beta$

The mutated form of Expr now requests for input and the register is ready to output the value on request.

$= \tau.\tau.\tau.\{ \bar{o}\ 5.NIL) \mid (\alpha w.REG(w) + \bar{\beta}5.REG(5)) \}\backslash\alpha\backslash\beta$

At this stage the compound system cannot move by itself and the only possible experiment is o by the observer.

In developing an algebra, congruent terms in the algebra have to be identified. In our algebra the notion of observation congruence is used. Two terms/programs in our algebra are observationally equivalent if for all experiments on the two programs, the ensuing programs remain equivalent. This notion leads to referential translucency referred to earlier. We do not pursue this here. For further details of observational equivalence the reader is referred to [6].

Prior to introducing the Expansion Theorem which we used in the example above, we list some useful definitions.

1. The CCS programs/behavior expressions can be looked upon as a guarded command language, where the input and output actions correspond to guards and the expressions following them correspond to the commands.
2. Behavior expressions which do not contain free variables in them are said to be closed form expressions. We will be dealing mainly with closed form expressions.
3. Taking into consideration the fact that a behavior expression is a set of non-deterministic expressions with guards, a behavior expression can be represented as

$$\Sigma g_i.E_i$$

4. The function "names" takes an expression as argument and gives the set of labels / port identifiers in the expression.
5. The values that can be passed are restricted to the primitive data types. This stems from the type equation proposed for processes in [5] called resumptions, which are analogous to continuations in the standard semantics. Later we will see that this restriction hampers attempts to give semantics for demand-driven execution.
6. In any behavior expression/program, the variable used to receive values in an input

action binds all of the occurrences of the variable in the expression following it. For instance in g.E, the scope of the variables occurring in g is the expression E.

The Expansion theorem provides a framework for deriving the actions of a composite system from the actions of its primitive elements.

Expansion Theorem

Let $B = (B_1 \mid B_2 \mid \ldots \mid B_n) \backslash A$. Then

$B = \Sigma.\{g.\ (B_1 \mid B_2 \mid .. \mid B_i' \ldots \mid B_n) \backslash A\}$

– – where $g.B_j'$ is a summand of B_i
– – and name (g) does not belong to A

+

$\Sigma.\{\tau.\ (B_1 \mid \ldots \mid B_i\{e/x\} \mid \ldots \mid B_j' \mid \ldots \mid B_n) \backslash A\}$

– – where $\alpha x.B_i'$ is a summand of B_i
– – and $\bar{\alpha} e.B_j'$ is a summand of B_j
– – when "i" is not equal to "j".

provided that in the first term no free variable in B_k is bound by the guard g.

The two parts of the expansion theorem take care of communication of values with the environment(experimentation) and the internal communication of values between two processes in the composed system. The guard of the first summand explains the fact that the whole system expects an input from its environment, whereas the guard τ in the second summand takes care of the internal value communication.

3. An applicative language

The applicative language that we consider has the usual features of functional abstraction, function application, if – then – else, primitive functions and stream processing functions. The additional feature that we have added is a choice operator. The basic idea is to reflect call – by – need and lazy evaluation in the semantics. We first consider a language without stream processing functions. In section 3.2 we add streams to our language but restrict them to have elements from the flat domain of integers and boolean.

3.1. Syntax

The syntax of our example language in BNF is

<Expr> ::= <PrimitiveValues>

| *if* <Expr> *then* <Expr> *else* <Expr>

| <Expr> [] <Expr>

| g(<Expr>$_1$, ... , <Expr>$_n$)
– – where g is an n – ary strict function
– – on the primitive data types.

| F(<Expr>$_1$, ... , <Expr>$_n$)
– – where F is a functional abstraction
– – defined in the program

<Prog> ::= {<Decl> ,}* <Expr>

<Decl> ::= <FunctionId> = $\lambda x_1 \ldots x_n$. <Expr>

<FunctionId> ::= Identifiers

<PrimitiveValues> ::= Integers | Boolean

We will make the following assumptions about the language.

1. We will, for the moment, consider only functional abstractions without any free variables in them as we do not wish to concern ourselves with details of binding.
2. The primitive functions are strict.
3. To achieve demand-driven evaluation, if-then-else is to evaluate only the conditional and return the arm of if-then-else chosen unevaluated.
4. Function applications are evaluated by call-by-need.

The following conventions will be made use of in providing the semantics.

1. Expression results are passed to their context via the label $\bar{\rho}$ and received by its complementary label.
2. The map from syntax to semantics is given by the function [...], i.e.,

   [syntactic object] = semantics of the syntactic object.

3. We will use structural induction in giving the semantics.
4. An auxiliary function *result* is used to pass the result of evaluation of an expression to its context and is defined as

   B_1 *result* $B_2 = (B_1 \mid B_2) \backslash \rho$.

   B_1 communicates the result at port ρ and the agent receives the same at the port $\bar{\rho}$. The connection between these two ports is hidden from external observation.
5. Primitive values themselves are expressed as behavior expressions. For example

   $[1] \doteq \bar{\rho}\,1.NIL$

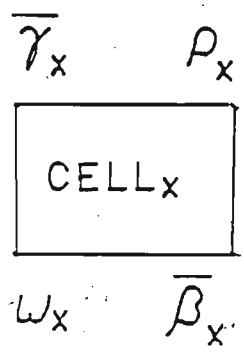
6. For every variable x used in the functional abstraction, a data structure called "CELL" is allocated. The behavior expression for the CELL is defined as follows (refer figure 3-1).

   $CELL_x <= \omega_x.\, \bar{\gamma}_x.\, \rho_x y.\bar{\beta}_x y.NEW_x(y)$

   $NEW_x(y) <= \omega_x.\, \bar{\beta}_x y.\, NEW_x(y).$

   Intuitively, the first reference to the parameter x is serviced by the agent $CELL_x$, while all subsequent references to the parameter x are serviced by the agent $NEW_x(y)$, which retains its value y across references. The conventions used in the evaluation of an actual parameter (and thus in the behavior expression CELL) are as follows:

   - The demand for the value of an actual parameter is received at the port ω_x. The request for the value is serviced by communicating the value through the label $\bar{\beta}_x$.
   - The first reference of the variable x, forces the agent $CELL_x$ to demand the value from the agent for the actual parameter at the port γ_x. The agent for the actual parameter returns the value at the port $\bar{\rho}_x$, which is received by the agent $CELL_x$ at the corresponding port ρ_x.



ω_x, $\bar{\gamma}_x$, ρ_x and $\bar{\beta}_x$ are the labels for CELL$_x$

Figure 3 – 1: Definition of the data structure CELL

3.1.1. Semantics

We now detail the semantics for our language. We will as a convention give the formal semantics for each of the syntactic classes and follow it with an explanatory note.

Primitive Values

$$[n] = \bar{\rho}\, n.\ \text{NIL}$$

- The agent passes the value n to the context and degrades to NIL.
- Primitive values are created every time they are used.

Variable

$$[id] = \bar{\omega}_{id}.\ \beta_{id} y.\ \bar{\rho}\, y.\ \text{NIL}$$

- As we are working with an applicative language, variables appear only within a functional abstraction.
- Variable "id" is associated with a unique agent CELL$_{id}$ as defined earlier.
- The signal $\bar{\omega}_{id}$ is used to demand the value from the agent CELL$_{id}$ and the value is subsequently received at the label β_{id}. Once the value from CELL$_{id}$ is available, it is passed on to the surrounding context (through the label $\bar{\rho}$ in accordance with our previously stated convention).

if – then – else

[*if* Expr *then* Expr1 *else* Expr2] = [Expr] *result* ρx. if x then [Expr1] else [Expr2]

- The condition is evaluated and depending on the value of the conditional the arm of if – then – else is chosen.
- This is an approximation to demand – driven evaluation, as what we would like to do is return the result unevaluated. Presently we are precluded from doing so, as the values that can be communicated have been restricted to be only primitive values. To achieve demand – driven evaluation a behavior expression has to be returned.
- Note the difference between *if – then – else* and the clause if – then – else. The former is a construct in our example language whereas the latter is a construct in CCS.

Primitive Functions

$$[g(expr_1, \ldots, expr_n)] =$$
$$\{ [expr_1]\{\bar{\rho}_1/\bar{\rho}\} \mid \ldots \mid [expr_n]\{\bar{\rho}_n/\bar{\rho}\} \mid$$
$$\rho_1 x_1 \ldots \rho_1 x_1.\ \bar{\rho}\,(\, g(x_1, .. x_n)\,)\ \}\ \backslash \rho_1 \ldots \rho_n$$

- The arguments are evaluated and passed on to the agent computing the function g. Note that g denotes a primitive function in the language, and therefore can be immediately evaluated.

Choice Operator

$[\ \text{Expr1}\ []\ \text{Expr2}\ ] = \tau.[\text{Expr1}] + \tau.[\text{Expr2}]$

- [] is the choice operator in the language.
- We are explicitly making use of the + operator (in CCS) to reflect non-determinacy. Appending a τ guard to the two summands, implies that the choice is made internally and that the choice is not observable. This is called "erratic non-determinism" by Broy in [1].

Functional abstraction

$[F = \lambda x_1 \ldots x_n . \text{Expr}\ ] = W$
where
$W = (\ CELL_{x_1} \mid \ldots \mid CELL_{x_n} \mid$
$([\text{Expr}]\ \textit{result}\ \rho y.\ \bar{\rho}_F y.\ \text{NIL})) \backslash \omega_1 \ldots \omega_n\ \beta_1 \ldots \beta_n$

- The variables in the body of the function(i.e. Expr) access values from the data-structure CELL. A copy of the agent CELL is bound to each variable in a invocation of the function.
- The binding is achieved by restricting the labels ω (used for demanding the value of the actual argument) and β (used to communicate the actual value).
- The labels used in CELL to demand the value of the actual argument (from the agent denoting the actual parameter) viz., γ_x and ρ_x (for the variable x), which are restricted after binding a function call to a copy of the function.
- The function body returns the result of a function application through the label $\bar{\rho}_F$. This label is bound with its co-name and restricted when binding a function call to a copy of the function.

The mechanism for functional abstraction/function definition is detailed in figure 3-2.

Function application

$[F(\text{Expr}_1, \ldots\ \text{Expr}_n)\ ] =$

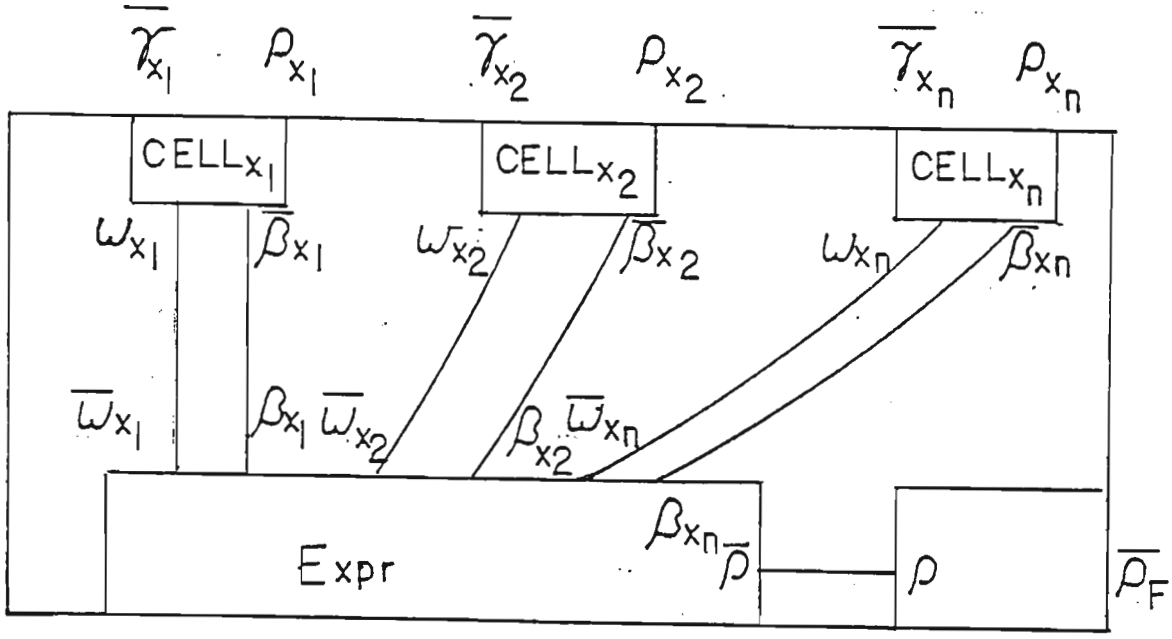
$(\gamma_{x_1}.[\text{Expr}_1]\ \{\rho_{x_1}/\rho\} \mid \ldots \mid \gamma_{x_n}.[\text{Expr}_n]\ \{\rho_{x_n}/\rho\} \mid$

$W\ \textit{result}\ \rho y.\ \bar{\rho}_F y.\ \text{NIL})\ \backslash \rho_{x_1} \ldots \rho_{x_n}\ \gamma_{x_1} \ldots \gamma_{x_n}\ \rho_F$

where W is the behavior expression for the functional abstraction of F defined earlier.

The reader is urged to read the following explanation with reference to the figure 3-3

- This behavior expression contains the following agents:
  - An agent for each of the actual parameters to the function call; these are clothed by the guard γ_x. This guard has to be peeled off before the denotation for the actual parameter can be evaluated.
  - An agent W modeling the function.
- As mentioned earlier the variables in the function body access the value of the actual parameter through the agent CELL.
- The CELLs are encapsulated in the agent W.
- As a CELL exists for each of the formal variables, the onus of demanding the value of the actual parameter is taken care off by the corresponding CELL.
- The labels ρ_x and γ_x used by a CELL and its corresponding agent for the actual arguments are bound in this behavior expression and restricted.



labels ω_{x_i}, β_{x_i} and ρ are restricted

Figure 3 – 2: A function definition

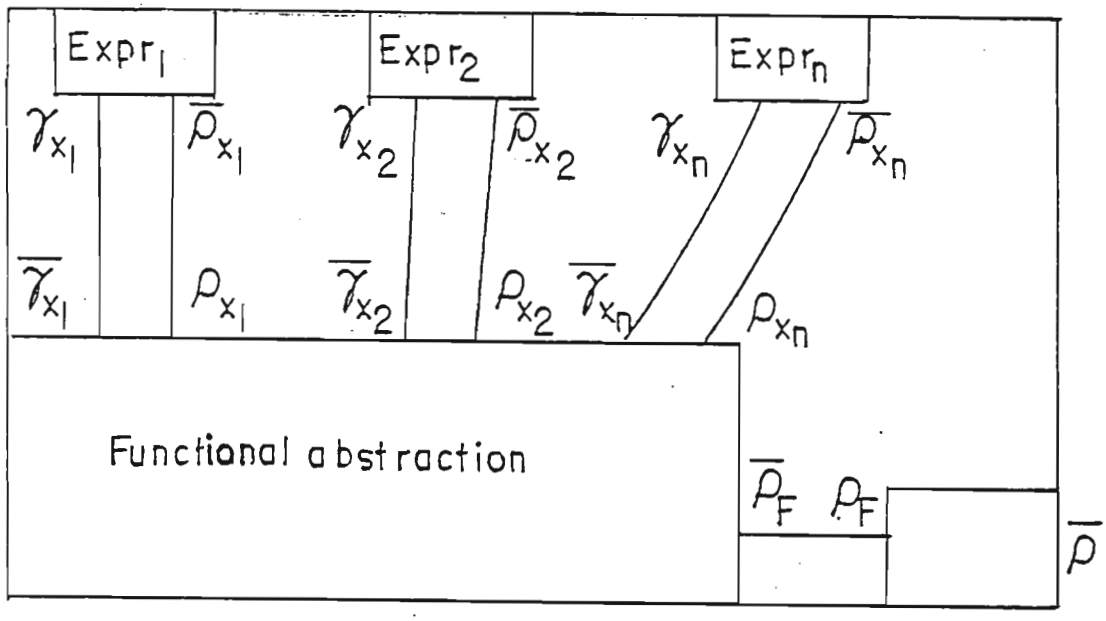
3.2. Stream processing functions

The ability to handle infinite objects are a necessity in any applicative language. In this paper we will add stream processing functions like append, first, rest etc. to our basic constructs but restrict the streams to have only primitive objects. Thus we add the following functions with their associated arities.

1. app : element X stream – > stream
2. first : stream – > element
3. rest : stream – > stream
4. nil : () – > stream
5. isnil : stream – > Bool

The function "nil" returns the distinguished element *nil* in the domain of streams. Associated with null stream we have a test for null streams as well. Typically the append function is implemented as a lazy function.

The modeling of append demands that the values returned by append (or any lazy function) be "delayed" [3] and that the strict functions "force" their arguments. By the same argument, the function "rest" has to return unevaluated streams. Modeling of lazy functions demand that the streams be modeled as behavior expressions in the semantic domain, thus there is a need to add behavior expressions to the values that can be communicated. Hence for the present, we add behavior expressions to the the values that can be passed around, without questioning the basis of CCS.



labels γ_{x_i}, ρ_{x_i} and ρ_F are hidden

Figure 3 – 3: Mechanism for function call

The upgrading of the allowed syntax is as follows.

<Expr> ::= ... | stream

| app (<primitive element> , stream) |

| first (stream)

| rest (stream)

| nil

| isnil (stream)

| ...

In packaging up a stream for delayed evaluation, the ports used are η,κ,ν to ask for first, rest and the question isnil. In the treatment of streams that follow, we have consciously avoided answering the question of "first(nil)" as we still not aware of what role the element $\perp$ plays in CCS.

The semantics are as following:

append

[app (el,st)] = W

Where W <= δ . (B_1 + B_2 + B_3)

where

$B_1 <= \eta$. [el] *result* ρe. $\bar{\rho}_A$e. MUT (e)

$B_2 <= \kappa$. $\bar{\rho}_A$([st]) .W

$B_3 <= \nu$. $\bar{\rho}_A$ ff . W

MUT (e) <= δ . (η . $\bar{\rho}_A$e . MUT (e) +

κ . $\bar{\rho}_A$([st]) . MUT (e) +

ν . $\bar{\rho}_A$ff . MUT (e))

where ff is the false value and [st] is the rest of the stream, which is passed as a behavior expression.

- The first element is not evaluated until the function "first" has been applied / the head of the stream has been demanded at η (in B_1).
- Once the function "first" has been applied, the head of the stream is available at the port $\bar{\rho}_A$, as can be seen in the behavior expression MUT.
- As any stream is formed out of appending the primitive elements to a stream, the value returned for the question of "isnil" is always false.
- The signal δ acts as the "delay" and the signal $\bar{\delta}$ acts as the "force" when some strict function is applied. The force/delay technique has been borrowed from [3]. A lazy function (f) in the force/delay technique passes back the result unevaluated, as say, in returning an function application g(x), f returns λ().(g(x)), which is forced by a strict function by applying () to closure λ().(g(x)).

first

[first (st)] = ([st] | δ . $\bar{\eta}$. ρ_Ae. $\bar{\rho}$ e.NIL) \ ρ_A

- NIL is not the element nil, rather it is the null action among the behavior expressions.
- The function "first" forces the evaluation of the head of the stream in case it has not been evaluated, otherwise it just acts as a signal.

rest

[rest (st)] = ([st] | δ . $\bar{\kappa}$. ρ_As . $\bar{\rho}$ s . NIL) \ ρ_A

- The value returned is a behavior expression and will have to be forced again to get the head of the stream.

nil

[nil] = Q
where Q <= δ .(ρ_A*nil* . Q + $\bar{\nu}$ tt .Q)

- *nil* is also being guarded by the delay, mainly to maintain consistency with the append function.
- Note also that on a query of "isnil" , this element will return the value true.

isnil

[isnil (st)] = ([st] | δ . ρ_Ae.
ρe . NIL) \ ρ_A

- isnil(app(nil,nil)) would yield a false value, as the basic assumption is that once the function "app" has been applied it is no more a null list.
- The test for nullity does not demand the evaluation of the first element.

In adding the stream processing functions to the language, we have demonstrated the flexibility of using CCS. One basic restriction of CCS that we have violated is passing behavior expressions as values. We as of now, do not know what the repercussions of this violation are. The answer to this question is to be answered in the future. While we are at it, it should now be possible to adapt this to the if – then – else statement making it completely demand driven. We do so as follows

[*if* Expr *then* Expr1 *else* Expr2] =

$$([\text{Expr}]\{\rho_1/\rho\} \mid (\rho_1 x\ .\ \text{if } x \text{ then } \rho([\text{Expr1}]) \text{ else } \rho([\text{Expr2}]))) \backslash \rho_1$$

The notion of passing around behavior expression is similar to the extensions proposed in [7], where labels are passed around as values to exploit CCS for evaluation strategies.

4. Conclusion

We have taken an applicative language with an indeterminate operator and have shown that it is possible to give semantics for the low – level details involved in evaluation strategies of applicative languages. In having committed the violation of extending CCS we have posed a new problem of how exactly CCS can be extended and exploited. The generalization of the language to handle error conditions like $\perp$ and structured streams are possible extensions to this work. It remains to be seen how the congruence relations over CCS can be exploited to talk about properties of programs, like detection of deadlocks etc. in the context of applicative languages.

References

[1] Manfred Broy, *A Fixed Point Approach to Applicative Multiprogramming*, Lectures at the International Summer School on Theoretical Foundations of Programming Methodology, (Aug 1981).

[2] Thomas Myers,A.Toni Cohen, *Towards an Algebra of Nondeterministic Programs*, Proceedings of Symposium on LISP and Functional Programming, ACM, (Aug 1982).

[3] Peter Henderson, *Functional Programming – Aplications and Implementations*, (Prentice – Hall, 1980).

[4] C.A.R.Hoare, Communicating Sequential Processes, *CACM 21*, 8 (Aug 1978), 666 – 677.

[5] G.Milne,R.Milner, Concurrent Processes and their Syntax, *J. ACM 26*, 2 (April 1979), 302 – 321.

[6] Robin Milner, *LNCS*, Volume 92: *Calculus of Communicating Systems*, (Springer Verlag, 1980).

[7] Ronan Sleep,J.R.Kennaway, *Applicative objects as Processes*, Proceedings of Symposium on LISP and Functional Programming, ACM, (Aug 1982).