A π-calculus Specification of Prolog

Benjamin Z. Li

Department of Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104-6389 USA
zhenli@saul.cis.upenn.edu

Abstract. A clear and modular specification of Prolog using the π-calculus is presented in this paper. Prolog goals are represented as π-calculus processes, and Prolog predicate definitions are translated into π-calculus agent definitions. Prolog's depth-first left-right control strategy as well as the cut control operator are modeled by the synchronized communication among processes, which is similar in spirit to continuation-passing style implementation of Prolog. Prolog terms are represented by persistent processes, while logical variables are modeled by complex processes with channels that, at various times, can be written, read, and reset. Both unifications with and without backtracking are specified by π-calculus agent definitions. A smooth merging of the specification for control and the specification for unification gives a full specification for much of Prolog. Some related and further works are also discussed.

1 Introduction

Prolog is a simple, powerful and efficient programming language, but its depth-first left-right control as well as the control operator cut (!) and its lack of the occurs check destroys the declarative reading of Prolog programs. For example, a left recursive clause will cause an infinite computation while a right recursive clause with same logic reading will terminate the computation. Hence, logic does not provide a simple and formal semantics for Prolog. In [Ros92a] Ross provided an interesting specification of Prolog control by mapping it into processes in the concurrent specification language CCS [Mil89]. In this paper, we develop and extend this approach significantly by using the π-calculus, a richer concurrent specification language. We are not only able to specify Prolog's control primitives but also its correct interaction with Prolog's unification, including the lack of the occur-check and the construction of circular terms.

The π-calculus [MPW92a, MPW92b, Mil91] is a calculus for modeling concurrent systems with evolving communication structure. It has been proven very powerful in modeling functional programming languages[Mil90b] and object-oriented programming languages[Wal90]. In this paper, we will introduce a clear and modular specification for Prolog using the π-calculus. The specification is modular in the sense that the Prolog control part and unification are specified separately, but can be merged together smoothly to form a full specification for Prolog. In fact, part of the motivation behind this paper was to understand how successfully the π-calculus could be used to specify the operational semantics of a non-trivial programming language, in this case Prolog. As we hope it will be clear from this paper, the π-calculus, along with a sorting discipline proposed for it, does indeed provide an attractive specification language.

The rest of this paper is organized as follows. The π-calculus will be briefly introduced in Section 2. A π-calculus specification for Prolog's depth-first left-right control as well as the cut control will be defined in Section 3, and a π-calculus specification for unification will be discussed in Section 4. Section 5 will merge these two specifications together to achieve a specification for full Prolog.

We will compare with some related works and discuss future works in Section 6. Section 7 is the conclusion.

2 The π-calculus

The π-calculus is a model of concurrent computation based upon the notation of naming, which provides an identity to an entity that allows it to concurrently coexist in an environment with other entities. The primitive elements of π-calculus are structureless entities called *Names*, infinitely many and denoted by $\{x, y, z \ldots\} \in \mathcal{N}$. A name refers to a communication channel. If the name x represents the input end of a channel, then the co-name $\overline{x}$ represents its output end. In the following syntax of the π-calculus, P, P_1, P_2 range over process expressions, A ranges over agent identifiers $\mathcal{K}$, and $\tilde{y}$ is an abbreviation for a sequence of names $y_1 \ldots y_n$ $(n \geq 0)$.

$$P ::= 0 \mid \overline{x}\tilde{y}.P \mid x(\tilde{y}).P \mid P_1 + P_2 \mid P_1 | P_2 \mid (\nu x)P \mid [x=y]P \mid [x \neq y]P \mid A(\tilde{y}) \mid !P$$

where

- 0 is the *inaction* process which can do nothing.
- $\overline{x}\,\tilde{y}.P$ can output the name(s) $\tilde{y}$ along the channel x and then becomes P.
- $x(\tilde{y}).P$ can input some arbitrary name(s) $\tilde{z}$ along the the channel x and then becomes $P\{\tilde{z}/\tilde{y}\}$. Of cause, $\tilde{y}$ and $\tilde{z}$ have to be of the same length.
- A summation $P_1 + P_2$ can behave as either P_1 or P_2 non-deterministically.
- A composition $P_1 \mid P_2$ means that P_1 and P_2 are concurrently active, so they can act independently but can also communicate. For example, if $P = \overline{x}z.P'$ and $Q = x(y).Q'$ then $P|Q$ means that either P can output z along channel x; or Q can input an name along channel x; or P and Q can communicate internally by performing a *silent* action τ and then becomes $P'|Q'\{z/y\}$.
- A restriction $(\nu x)P$ declares a new name (private name) x in P that is different from all external names. For example, $(\nu x)(\overline{x}z.P|x(y).Q')$ can only perform internal communication, but $\overline{x}z.P|(\nu x)x(y).Q'$ can not communicate. Actually, $(\nu x)x(y).Q'$ is a dead process, which is equivalent to 0.
- A match $[x=y]P$ behaves like P if x and y are identical, and otherwise like 0.
- A mis-match $[x \neq y]P$ behaves like P if x and y are not identical, and otherwise like 0.[1]
- A *defined* agent $A(\tilde{y})$ must have a corresponding defining equation of the form: $A(\tilde{x}) \stackrel{def}{=} P$. Then $A(\tilde{y})$ is the same as $P\{\tilde{y}/\tilde{x}\}$.
- A replication $!P$, which can be defined as: $!P \stackrel{def}{=} P|!P$, provides infinite copies of P in composition, i.e. $!P = P|P \ldots$.

A *transition* in the π-calculus is of the form: $P \xrightarrow{\alpha} Q$, which means that P can evolve into Q by performing the action α. The action α can be one of the $\tau, \overline{x}\tilde{y}, x(\tilde{y})$ and a fourth action called *bound output action* which allows a process to output a private name and hence widen the scope of the private name. The formal definition of translation relation $\xrightarrow{\alpha}$ is given in [MPW92b]. Here is a simple

[1] the mis-match is not presented in the original π-calculus, but is included here to simplify the specifications presented in this paper.

example:

$$(a.P_1 + b.\overline{c}.P_2) \mid x(y).\overline{y}.Q \mid \overline{x}b.R \xrightarrow{\tau} (a.P_1 + b.\overline{c}.P_2) \mid \overline{b}.Q\{b/y\} \mid R$$
$$\xrightarrow{\tau} \overline{c}.P_2 \mid Q\{b/y\} \mid R$$
$$\xrightarrow{\overline{c}} P_2 \mid Q\{b/y\} \mid R$$

Some convenient abbreviations are used in this paper, such as $x(y)$ (resp. $\overline{x}y$) as an abbreviation for $x(y).0$ (resp. $\overline{x}y.0$), $(\nu x_1 \ldots x_n)P$ for $(\nu x_1)\ldots(\nu x_n)P$, and $\xrightarrow{\alpha_1 \ldots \alpha_n}$ for n sequential transitions $\xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_n}$. We also use the anonymous channel name _ when the name itself does not matter.

Names in $\tilde{y}$ are said to be bound in $x(\tilde{y}).P$, and so is the name y in $(\nu y)P$. Some processes are considered to be equivalent according to the *structural congruence* $\equiv$ relation, defined in [Mil91]. Two processes are *observation-equivalent* or *bisimilar* ($\approx$) if their input/output relationships (ignoring the internal communication τ) are the same. Bisimilarity and process equivalence are discussed in [MPW92b, Mil89].

3 Specifying the Prolog Control in the π-calculus

In this section, we will specify the Prolog's depth-first left-right control strategy and the cut control operator in the π-calculus. For simplicity, Prolog terms and unification will be ignored until next section. A Prolog goal is represented as a π-calculus process, and a Prolog predicate definition is translated into a π-calculus agent definition.

3.1 Prolog Goals as Processes

The evaluation of a Prolog goal **G** can result in either *success* or *fail*. A successfully evaluated goal **G** might be *backtracked* to find alternative solutions. So the corresponding π-calculus process $[\![G]\!](s, f, b)$ is associated with three channel names: s (for *success* channel), f (for *fail* channel), b (for *backtracking* channel). Its behavior can be described as follows:

$$[\![G]\!](s, f, b) \approx \begin{cases} \overline{s}.b.G_{alt\_sol}(s, f, b) & \text{if evaluation of the Goal } G \text{ succeeds.} \\ \overline{f} & \text{if evaluation of the Goal } G \text{ fails.} \end{cases}$$

After having found one solution, $[\![G]\!](s, f, b)$ sends an output action $\overline{s}$ and then waits on the *backtracking* channel b before computing alternative solutions (denoted by $G_{alt\_sol}(s, f, b)$). Suppose a goal G can produce n ($n \geq 0$) solutions, then $[\![G]\!](s, f, b) \approx (\overline{s}.b.)^n \overline{f}$.

A left-associative sequential-and control operator $\triangleright$ is introduced in order to simplify the notation of the corresponding process for a Prolog conjunctive goal (P, Q)[2]:

$$(A \triangleright B)(s, f, b) \triangleq (\nu s' f')(A(s', f, f') \mid !s'.B(s, f', b)) \quad (1)$$

$$[\![(P, Q)]\!](s, f, b) \triangleq ([\![P]\!] \triangleright [\![Q]\!])(s, f, b) \quad (2)$$

$$\equiv (\nu s' f')([\![P]\!](s', f, f') \mid !s'.[\![Q]\!](s, f', b)) \quad (3)$$

The behaviors of definition (3) can be understood as follows:

[2] $\triangleq$ is used for translations from Prolog to π-calculus while $\stackrel{def}{=}$ is used in π-calculus agent definitions.

- If $[\![P]\!](s', f, f')$ reports *fail* via $\overline{f}$, so does $[\![P, Q]\!](s, f, b)$ since they use the same channel f.
- If $[\![P]\!](s', f, f')$ reports *success* via $\overline{s'}$, then one copy of $[\![Q]\!](s, f', b)$ will be activated after the synchronized communication of $\overline{s'}$ and s'. And then,
  - If $[\![Q]\!](s, f', b)$ reports *success* via $\overline{s}$, so does $[\![P, Q]\!](s, f, b)$ since both use the same channel s.
  - If $[\![Q]\!](s, f', b)$ reports *fail* via $\overline{f'}$, then $[\![P]\!](s', f, f')$ will be backtracked.

The bang ! before $s'.[\![Q]\!](s, f', b)$ is necessary because backtracked $[\![P]\!](s', f, f')$ may find another solution and then need to invoke $[\![Q]\!](s, f', b)$ again.

Similarly, a left-associative sequential-or control operator $\oplus$ is used for disjunctive goal $(P; Q)$:

$$(A \oplus B)(s, f, b) \triangleq (\nu\, f')(A(s, f', b) \mid f'.B(s, f, b)) \tag{4}$$

$$\begin{aligned}[\![(P; Q)]\!](s, f, b) &\triangleq ([\![P]\!] \oplus [\![Q]\!])(s, f, b) \\ &\equiv (\nu\, f')([\![P]\!](s, f', b) \mid f'.[\![Q]\!](s, f, b))\end{aligned} \tag{5}$$

where $[\![P]\!]$ and $[\![Q]\!]$ use the same s and b channels, corresponding to the **or** relationship of P and Q. However, $[\![Q]\!]$ can be activated only after $[\![P]\!]$ reports *fail* via $\overline{f'}$. Thus, we exactly models Prolog's sequential nature of **or** and Prolog's sequential clause searching as we shall see in Section 3.2.

Using the *sort* notation in [Mil91], we introduce two *sorts*, **Succ** and **Fail** for the *success* channel names and *fail* channel names respectively. Based on the above discussion, it is clear that *backtracking* channel names have the same sort as the *fail* channel names. The following sorting:

$$\{\mathbf{Succ} \mapsto (), \mathbf{Fail} \mapsto ()\}$$

means that both *success* channels and *fail* channels carry no names. A general sorting of the form $\mathbf{S} \mapsto (\mathbf{S_1}, \ldots, \mathbf{S_n})$ means that along a channel s of sort $\mathbf{S}$, we can receive or send n names of sorts $\mathbf{S_1}, \ldots, \mathbf{S_n}$ respectively. We also use $s\hat{}t$ for the concatenation of sorts, e.g. $(S_1)\hat{}(S_2, S_3) = (S_1, S_2, S_3)$. The following notations are used to specify sorts for names and agents:

- s, s_i, s' : **Succ** means that names s, s_i, s' are of sort **Succ**. And similarly, f, b, f', f_i : **Fail** means that f, b, f', f_i are of sort **Fail**.
- $[\![G]\!], [\![P, Q]\!], [\![P; Q]\!]$: (**Succ**, **Fail**, **Fail**) means that agents $[\![G]\!], [\![P, Q]\!], [\![P; Q]\!]$ shall take three name arguments of sorts **Succ**, **Fail**, **Fail** respectively.

3.2 Prolog Predicate Definitions as π-calculus Agent Definitions

A Prolog predicate is defined by a sequence of clauses. Suppose we have the following definition for a predicate P, consisting of m clauses:

$$P = \begin{cases} P :- Body_1. \\ P :- Body_2. \\ \quad \ldots \\ P :- Body_m. \end{cases}$$

Then the corresponding π-calculus agent definition for P : (**Succ**, **Fail**, **Fail**) is:

$$\begin{aligned}P(s, f, b) &\stackrel{def}{=} (P_1 \oplus P_2 \oplus P_3 \oplus \ldots \oplus P_m)(s, f, b) \\ &\equiv (\nu\, f_1 f_2 \ldots f_{m-1})\,(P_1(s, f_1, b) \mid f_1.P_2(s, f_2, b) \mid f_2.P_3(s, f_3, b) \mid \\ &\qquad \ldots \mid f_{m-1}.P_m(s, f, b))\end{aligned} \tag{6}$$

where the definition for each P_i $(1 \leq i \leq m)$ is determined by the *ith* clause of the predicate P.

Suppose the *ith* clause of P is defined as: $'P :- B_1, B_2, \ldots, B_n.'$, then the agent P_i : (**Succ, Fail, Fail**) is defined as follows:

$$P_i(s,f,b) \stackrel{def}{=} ([\![B_1]\!] \rhd [\![B_2]\!] \rhd [\![B_3]\!] \rhd \ldots \rhd [\![B_n]\!])(s,f,b) \quad (7)$$

$$\equiv (\nu\, s_1 \ldots s_{n-1}\, f_1 \ldots f_{n-1})([\![B_1]\!](s_1,f,f_1) \mid !s_1.[\![B_2]\!](s_2,f_1,f_2) \mid !s_2.[\![B_3]\!](s_3,f_2,f_3) \mid \ldots \mid !s_{n-1}.[\![B_n]\!](s,f_{n-1},b)\ ) \quad (8)$$

The behavior of above definition can be described in a similar way as the behavior of definition (3). We can also imagine that there exist two chains in definition (8): a *success* chain connected by s_i's, and a *backtracking* chain connected by f_i's, as illustrated in Fig. 1.

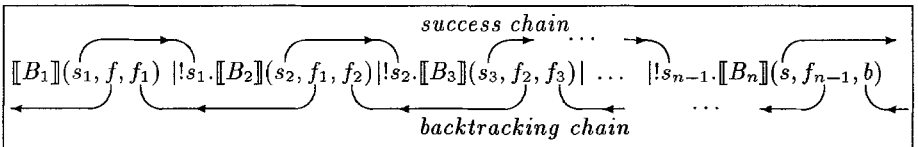


Fig. 1. success and backtracking chain

If the *ith* clause of P is a fact (i.e. a clause with empty body): $'P.'$, then the agent P_i is defined as: $P_i(s,f,b) \stackrel{def}{=} \overline{s}.b.\overline{f}$, which says that P_i reports *success* via $\overline{s}$ only once and then reports *fail* via $\overline{f}$ upon receiving *backtrack* via b.

Now, let us look at a propositional Prolog example and its π-calculus translation:

$a.$	$A(s,f,b) \stackrel{def}{=} \overline{s}.b.\overline{f}$
$b :- a.$	$B_1(s,f,b) \stackrel{def}{=} A(s,f,b) \quad (i.e. \equiv \ \overline{s}.b.\overline{f})$
$b :- b, a.$	$B_2(s,f,b) \stackrel{def}{=} (B \rhd A)(s,f,b)$
	$\equiv (\nu s_1 f_1)(B(s_1,f,f_1) \mid !s_1.A(s,f_1,b))$
	$B(s,f,b) \stackrel{def}{=} (B_1 \oplus B_2)(s,f,b)$
	$\equiv (\nu f')(B_1(s,f',b) \mid f'.B_2(s,f,b))$

Using the structural congruence relation and transition relation discussed in Section 2, the process $B(s,f,b)$ behaves as follows:

$$B(s,f,b) \equiv (\nu f')(B_1(s,f',b) \mid f'.B_2(s,f,b)) \equiv (\nu f')(A(s,f',b) \mid f'.B_2(s,f,b))$$

$$\equiv (\nu f')(\overline{s}.b.\overline{f'} \mid f'.B_2(s,f,b)) \stackrel{\overline{s}.b}{\rightarrow} (\nu f')(\overline{f'} \mid f'.B_2(s,f,b))$$

$$\stackrel{\tau}{\rightarrow} (0 \mid B_2(s,f,b)) \equiv B_2(s,f,b) \equiv (\nu s_1 f_1)(B(s_1,f,f_1) \mid !s_1.A(s,f_1,b))$$

$$\equiv (\nu s_1 f_1)((\nu f')(B_1(s_1,f',f_1) \mid f'.B_2(s_1,f,f_1)) \mid !s_1.A(s,f_1,b))$$

$$\equiv (\nu s_1 f_1)((\nu f')(\overline{s_1}.f_1.\overline{f'} \mid f'.B_2(s_1,f,f_1)) \mid !s_1.A(s,f_1,b))$$

$$\stackrel{\tau}{\rightarrow} (\nu s_1 f_1)((\nu f')(f_1.\overline{f'} \mid f'.B_2(s_1,f,f_1)) \mid A(s,f_1,b) \mid !s_1.A(s,f_1,b))$$

$$\equiv (\nu s_1 f_1)((\nu f')(f_1.\overline{f'} \mid f'.B_2(s_1,f,f_1)) \mid \overline{s}.b.\overline{f_1} \mid !s_1.A(s,f_1,b))$$

$$\stackrel{\overline{s}.b}{\rightarrow} (\nu s_1 f_1)((\nu f')(f_1.\overline{f'} \mid f'.B_2(s_1,f,f_1)) \mid \overline{f_1} \mid !s_1.A(s,f_1,b))$$

$$\stackrel{\tau.\tau}{\rightarrow} (\nu s_1 f_1)(B_2(s_1,f,f_1)) \mid !s_1.A(s,f_1,b)) \rightarrow \ldots\ldots$$

which shows that $B(s,f,b)$ can perform infinitely many $\overline{s}.b.$ actions, reflecting the infinitely many solutions of predicate b.

3.3 Specifying the Cut

In Prolog, the cut ! is a non-declarative control operator used to cut search space. It can also be treated as a special goal, which is translated into a π-calculus agent Cut : **(Fail)**^**(Succ, Fail, Fail)**:

$$Cut(f_0)(s,f,b) \stackrel{def}{=} \overline{s}.b.\overline{f_0} \tag{9}$$

The process $Cut(f_0)(s,f,b)$, when activated, will report *success* on $\overline{s}$ as usually, but will reports *fail* on $\overline{f_0}$ instead of the regular *fail* channel ($\overline{f}$). As we shall see in definition (11), the f_0 must be the *fail* channel of the calling process. Suppose that the *ith* clause of the predicate P contains a cut as the *jth* goal in the body: $P :- B_1 \ldots B_{j-1}, !, B_{j+1}, \ldots, B_n.$

Then the agent P_i : **(Fail)**^**(Succ, Fail, Fail)** and P : **(Succ, Fail, Fail)** are defined as follows:

$$\begin{aligned} P(s,f,b) &\stackrel{def}{=} (P_1 \oplus P_2 \oplus \ldots \oplus P_i(f) \oplus \ldots \oplus P_m)(s,f,b) && (10)\\ &\equiv (\nu f_1 \ldots f_{m-1})(P_1(s,f_1,b) | f_1.P_2(s,f_2,b) | \ldots | \\ &\qquad f_{i-1}.P_i(f)(s,f_i,b) | \ldots | f_{m-1}.P_m(s,f,b)) \\ P_i(f_0)(s,f,b) &\stackrel{def}{=} ([\![B_1]\!] \rhd \ldots [\![B_{j-1}]\!] \rhd Cut(f_0) \rhd [\![B_{j+1}]\!] \rhd \ldots [\![B_n]\!])(s,f,b) && (11)\\ &\equiv (\nu\, s_1 \ldots s_{n-1}\, f_1 \ldots f_{n-1})([\![B_1]\!](s_1,f,f_1) \mid !s_1.[\![B_2]\!](s_2,f_1,f_2) \mid \\ &\quad \ldots | !s_{i-2}.[\![B_{i-1}]\!](s_{i-1},f_{i-2},f_{i-1}) | !s_{i-1}.Cut(f_0)(s_i,f_{i-1},f_i) | \\ &\qquad !s_i.B_{i+1}(s_{i+1},f_i,f_{i+1}) | \ldots | !s_{n-1}.[\![B_n]\!](s,f_{n-1},b)\ ) && (12) \end{aligned}$$

In (10), the *fail* channel f of a calling process is passed to P_i, and then is passed to the Cut process in (11). By identifying the failure of Cut (upon backtracking) with the failure of the calling process P, we achieve the effect of the cut.

Since a backtracking never passes across a cut, the ! before s_{i-1} and s_i in definition (12) is not necessary. So we can optimize the definitions using an optimized sequential-and operator $\rhd|$:

$$\begin{aligned} (A \rhd| B)(s,f,b) &\stackrel{\triangle}{=} (\nu s' f')(A(s',f,f') \mid s'.B(s,f',b)) && (13)\\ P_i(f_0)(s,f,b) &\stackrel{def}{=} ([\![B_1]\!] \rhd \ldots [\![B_{j-1}]\!] \rhd| Cut(f_0) \rhd| [\![B_{j+1}]\!] \rhd \ldots [\![B_n]\!])(s,f,b) && (14)\\ &\equiv (\nu\, s_1 \ldots s_{n-1}\, f_1 \ldots f_{n-1})([\![B_1]\!](s_1,f,f_1) \mid !s_1.[\![B_2]\!](s_2,f_1,f_2) \mid \\ &\quad \ldots | !s_{i-2}.[\![B_{i-1}]\!](s_{i-1},f_{i-2},f_{i-1}) | s_{i-1}.Cut(f_0)(s_i,f_{i-1},f_i) | \\ &\qquad s_i.B_{i+1}(s_{i+1},f_i,f_{i+1}) | \ldots | !s_{n-1}.[\![B_n]\!](s,f_{n-1},b)\ ) && (15) \end{aligned}$$

If the second clause for the predicate b in the example of Section 3.2 is replaced with $'b :- b, !, a.'$, then we have:

$$\begin{aligned} A(s,f,b) &\stackrel{def}{=} \overline{s}.b.\overline{f} \\ B_1(s,f,b) &\stackrel{def}{=} A(s,f,b) \quad (\equiv\ \overline{s}.b.\overline{f}) \\ B_2(f_0)(s,f,b) &\stackrel{def}{=} (B \rhd| Cut(f_0) \rhd| A)(s,f,b) \\ &\equiv (\nu s_1 s_2 f_1 f_2)(B(s_1,f,f_1) \mid s_1.Cut(f_0)(s_2,f_1,f_2) \mid s_2.A(s,f_2,b)) \\ B(s,f,b) &\stackrel{def}{=} (B_1 \oplus B_2)(s,f,b) \ \equiv\ (\nu f')(B_1(s,f',b) | f'.B_2(f)(s,f,b)) \end{aligned}$$

Now, the behavior of process $B(s,f,b)$ is different from the one in Section 3.2:

$$\begin{aligned}
B(s,f,b) &\equiv (\nu f')(B_1(s,f',b) \mid f'.B_2(f)(s,f,b)) \equiv (\nu f')(\overline{s}.\overline{f'} \mid f'.B_2(f)(s,f,b)) \\
&\xrightarrow{\overline{s}.b} (\nu f')(\overline{f'} \mid f'.B_2(f)(s,f,b)) \xrightarrow{\tau} B_2(f)(s,f,b) \\
&\equiv (\nu s_1 s_2 f_1 f_2)(B(s_1,f,f_1) | s_1.Cut(f)(s_2,f_1,f_2) | s_2.A(s,f_2,b)) \\
&\equiv (\nu s_1 s_2 f_1 f_2)((\nu f')(\overline{s_1}.f_1.\overline{f'} \mid f'.B_2(f)(s_1,f,f_1)) | \\
&\qquad\qquad s_1.Cut(f)(s_2,f_1,f_2) | s_2.A(s,f_2,b)) \\
&\xrightarrow{\tau} (\nu s_1 s_2 f_1 f_2)((\nu f')(f_1.\overline{f'} \mid f'.B_2(f)(s_1,f,f_1)) | \\
&\qquad\qquad Cut(f)(s_2,f_1,f_2) | s_2.A(s,f_2,b)) \\
&\equiv (\nu s_1 s_2 f_1 f_2)((\nu f')(f_1.\overline{f'} \mid f'.B_2(f)(s_1,f,f_1)) | \overline{s_2}.f_2.\overline{f} | s_2.A(s,f_2,b)) \\
&\xrightarrow{\tau} (\nu s_1 s_2 f_1 f_2)((\nu f')(f_1.\overline{f'} \mid f'.B_2(f)(s_1,f,f_1)) | f_2.\overline{f} | A(s,f_2,b)) \\
&\equiv (\nu s_1 s_2 f_1 f_2)((\nu f')(f_1.\overline{f'} \mid f'.B_2(f)(s_1,f,f_1)) | f_2.\overline{f} | \overline{s}.b.\overline{f_2}) \\
&\xrightarrow{\overline{s}.b} (\nu s_1 s_2 f_1 f_2)((\nu f')(f_1.\overline{f'} \mid f'.B_2(f)(s_1,f,f_1)) | f_2.\overline{f} | \overline{f_2}) \\
&\xrightarrow{\tau.\overline{f}} (\nu s_1 s_2 f_1)((\nu f')(f_1.\overline{f'} \mid f'.B_2(f)(s_1,f,f_1))) \approx 0
\end{aligned}$$

which shows that after reporting twice *success* via $\overline{s}$ (corresponding to the two solutions of the predicate b), $B(s,f,b)$ will terminate.

4 Specifying Unification in the π-calculus

In this section, we will show how to represent Prolog terms as π-calculus processes, and logical variables as complex processes with channels that, at various times, can be written, read, and reset. We then show how to specify, in π-calculus, unification with and without backtracking.

4.1 Terms as Processes

The following sorting will be used for specifying terms:

$$\{\mathbf{Tag} \mapsto (), \mathbf{Cell} \mapsto (\mathbf{Tag}, \mathbf{Ptr} \cup \mathbf{Cell}), \mathbf{Ptr} \mapsto (\mathbf{Cell}, \mathbf{Ptr})\}$$

A channel x:**Cell** is used for representing terms, and different kinds of terms are distinguished by different names of sort **Tag** that x carries:

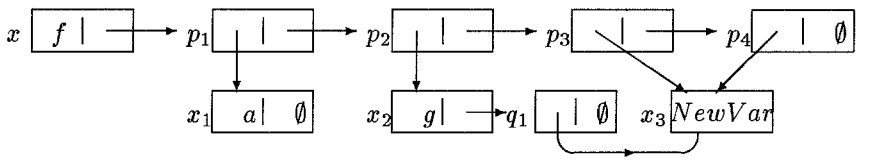
- **Var** : **Tag** denotes an unbound logical variable. **Ref** : **Tag** denotes a bound logical variable. When the first name carried on a channel x : **Cell** is **Ref**, then the second name must be of sort **Cell**. In all other cases, the second name must be of sort **Ptr**.
- f, g, k : **Tag** denote function or constant terms where f, g, k are functor names or constant names. As stated above, the second name that carried on x must be of sort **Ptr**.

A channel p : **Ptr** is used for representing, in a form similar to a link list structure (see the coming example), the arguments of a function. $\emptyset$:**Ptr** denotes the end of a list. The following is the translation from Prolog terms to π-calculus processes:

$$\begin{aligned}
[\![f(t_1,\ldots,t_n)]\!](x) &\triangleq (\nu\ p_1 \ldots p_n\ x_1 \ldots x_n)\ (!\overline{x}\ f\ p_1 \mid !\overline{p_1}\ x_1\ p_2 \mid !\overline{p_2}\ x_2\ p_3 \mid \ldots \\
&\qquad\qquad \mid !\overline{p_n}\ x_n\ \emptyset \mid [\![t_1]\!](x_1) \mid \ldots \mid [\![t_n]\!](x_n)) \\
[\![k]\!](x) &\triangleq !\overline{x}\ k\ \emptyset \\
[\![X]\!](x) &\triangleq \mathit{NewVar}(x)\ \mathit{Defined\ in\ Fig.\ 2}
\end{aligned}$$

with the restriction that several occurrences of a same variable X must use the same channel x and such $[\![X]\!](x)$ must be translated only once, e.g. the three occurrences of X in the following example. The picture shows the 'link list' representation of the arguments of $f(a, g(X), X, X)$.

$$[\![f(a, g(X), X, X)]\!](x) \triangleq (\nu p_1\ p_2\ p_3\ p_4\ x_1\ x_2\ x_3)(!\overline{x}\ f\ p_1$$
$$\mid !\overline{p_1}\ x_1\ p_2 \mid !\overline{p_2}\ x_2\ p_3 \mid !\overline{p_3}\ x_3\ p_4 \mid !\overline{p_4}\ x_3\ \emptyset$$
$$\mid !\overline{x_1}\ a\ \emptyset \mid (\nu q_1)(!x_2(g\ q_1) \mid !\overline{q_1}\ x_3\ \emptyset) \mid NewVar(x_3))$$



4.2 Logical Variable and Pure Unification

A definition of the agent $NewVar$ and a specification of pure unification without backtracking is illustrated in Fig. 2, where a logical variable is represented by

$$NewVar(x) \stackrel{def}{=} \overline{x}\ \mathbf{Var}\ \emptyset.NewVar(x) + x(\_\ y).BndVar(x, y)$$
$$BndVar(x, y) \stackrel{def}{=} !\overline{x}\ \mathbf{Ref}\ y$$
$$Dref(x, r) \stackrel{def}{=} x(tag\ y).([tag = \mathbf{Ref}]Dref(y, r) + [tag \neq \mathbf{Ref}]\overline{r}\ x)$$
$$(x =_u y)(s, f, b) \stackrel{def}{=} (\nu\ r_x\ r_y)(Dref(x, r_x) | Dref(y, r_y) | r_x(x_1).r_y(y_1).$$
$$([x_1 = y_1]\overline{s}.b.\overline{f} +$$
$$[x_1 \neq y_1]x_1(tag_x\ p_x).y_1(tag_y\ p_y).$$
$$([tag_x = \mathbf{Var}]\ \overline{x_1}\ \_\ y_1.\overline{s}.b.\overline{f} +$$
$$[tag_y = \mathbf{Var}]\ \overline{y_1}\ \_\ x_1.\overline{s}.b.\overline{f} +$$
$$[tag_x \neq \mathbf{Var}][tag_y \neq \mathbf{Var}]([tag_x = tag_y](p_x =_p p_y)(s, f, b) +$$
$$[tag_x \neq tag_y]\ \overline{f}\ )\ )\ )\ )$$
$$(p =_p q)(s, f, b) \stackrel{def}{=} [p = q]\ \overline{s}.b.\overline{f} +$$
$$[p \neq q]\ p(x\ p_1).q(y\ q_1).((x =_u y) \bowtie (p_1 =_p q_1))(s, f, b)$$

Fig. 2. Pure Unification Without Backtracking

a process with two states:

- the unbound state $NewVar(x)$ which can either send (**Var** $\emptyset$) along channel x to indicate that x is not bound yet, or receive $(\_\ y)$ on channel x to indicate that x is going to be bound to the cell y, and therefore enter the bound state $BndVar(x, y)$.
- the bound state $BndVar(x, y)$ which always send (**Ref** y) along channel x to indicate that x is bound to y.

When x is bound to y in the state $BndVar(x, y)$, y itself may denote an unbound logical variable and may be bound to another cell later. Hence, possible reference chains can exist, such as in Fig. 3. In order to unify two cells

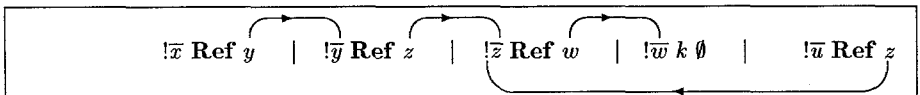


Fig. 3. Reference Chain

(representing two terms), *dereferencing* is performed by a process $Dref(x, r)$ which, when applied to a cell x, will send along channel r the last cell of the chain that contains the cell x. In the example of Fig. 3, $Dref(x, r)$ will cause $\overline{r}\ w$.

The process $(x =_u y)(s, f, b)$ performs unification, which first invokes $Dref$ to dereference possible chains in order to get the last cells x_1 and y_1:

- If x_1 and y_1 are the same cell, then done.
- If x_1 (resp. y_1) is an unbound variable, then x_1 (resp. y_1) is bound to the cell y_1 (resp. x_1).
- If x_1 and y_1 are both bound, then check whether they have same tag (i.e. same functor/constant name), if yes, then call agent $=_p$ to unify their arguments.

Basically, the agent $=_p$ sequentially calls the unification agent $=_u$ to unify all pairs of the corresponding arguments. Now, let us look at a simple example, illustrated in Fig. 4, where four unbound variables x, y, u, v are to be unified.

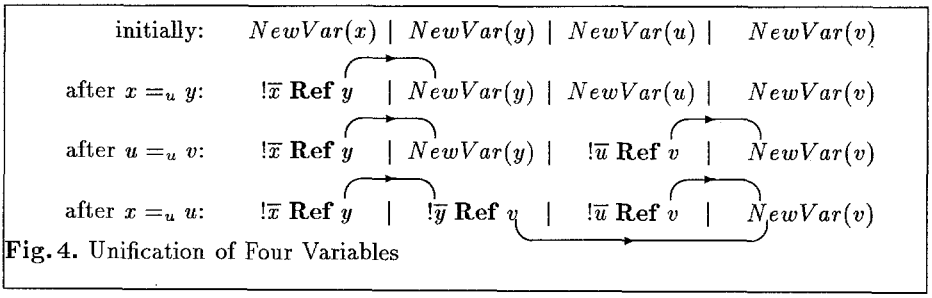


Fig. 4. Unification of Four Variables

In the last step, unification $x =_u u$ is actually performed at cell y and v due to the dereferencing.

4.3 Unification with Backtracking

In Prolog, a backtracked unification must undo all the variable bindings performed during the unification, i.e. reset the corresponding variables to the unbound state. We can modify the above unification agent definitions to incorporate the backtracking, as shown in the Fig. 5. Only the definitions for $BndVar$ and $=_u$ need to be changed. In the bound state $BndVar(x, y)$, a special cell (**Bck** _) can be received along channel x to indicate that x must be reset to the unbound state $NewVar(x)$. So, having reported *success* ($\overline{s}$) after binding y_1 to x_1 (resp. x_1 to y_1), the process $(x =_u y)(s, f, b)$ waits on the *backtracking*

$$BndVar(x,y) \stackrel{def}{=} \overline{x}\,\mathbf{Ref}\ y.BndVar(x,y) + x(tag\ \_).[tag = \mathbf{Bck}]NewVar(x)$$

$$\begin{aligned}(x =_u y)(s,f,b) \stackrel{def}{=}\ & (\nu\ r_x\ r_y)(Dref(x,r_x)|Dref(y,r_y)|r_x(x_1).r_y(y_1).\\ & ([x_1 = y_1]\overline{s}.b.\overline{f} +\\ & [x_1 \neq y_1]x_1(tag_x\ p_x).y_1(tag_y\ p_y).\\ **\rightarrow\ & ([tag_x = \mathbf{Var}]\ \overline{x_1}\ \_\ y_1.\overline{s}.b.\overline{x_1}\ \mathbf{Bck}\ \emptyset.\overline{f} +\\ **\rightarrow\ & [tag_y = \mathbf{Var}]\ \overline{y_1}\ \_\ x_1.\overline{s}.b.\overline{y_1}\ \mathbf{Bck}\ \emptyset.\overline{f} +\\ & [tag_x \neq \mathbf{Var}][tag_y \neq \mathbf{Var}]([tag_x = tag_y]\ (p_x =_p p_y)(s,f,b) +\\ & [tag_x \neq tag_y]\ \overline{f})\ )\ )\ )\end{aligned}$$

Fig. 5. Unification with Backtracking

channel b. If it receives a *backtracking* request b, then it will undo the variable binding by sending out a special cell (**Bck** _) along channel $\overline{x_1}$ (resp. y_1) before reporting failure via $\overline{f}$.

Occur-check is not performed at the above unification, hence circular terms are allowed to be constructed. However, an occur-check can be specified in π-calculus and be easily incorporated into the above unification.

5 Prolog

Now we are going to merge the specifications described in Section 3 and 4 to give a full specification of Prolog. We assume that all Prolog clauses have been simplified in such a way that all head unifications are explicitly called in the body, i.e. all clause heads contain only variables as arguments, such as in the following *append* example:

$$\begin{array}{l} append([\,],X,X). \\ append([W|X],Y,[W|Z]) :- \\ \quad append(X,Y,Z). \end{array} \Longrightarrow \begin{array}{l} append(X,Y,Z) :- X=[\,],Y=Z. \\ append(X,Y,Z) :- X=[W|X1],Z=[W|Z1], \\ \quad append(X1,Y,Z1). \end{array}$$

5.1 Undo Chains

The agent *Cut* defined in Section 3.3 is not powerful enough when combined with unification. Given a clause in the form: '$P :- B_1,\ldots,B_{j-1},!,B_{j+1},\ldots,B_n.$', a backtracking to the cut ! not only causes the failure of the calling process, but also has to undo all the variable bindings performed in $B_1,\ldots,B_{j-1}$. In order to achieve this effect, an *undo* chain is used by associating each goal process with two additional channels of sort **Undo**, i.e. $[\![G]\!](u,v)(s,f,b)$ with sorting $[\![G]\!] : (\mathbf{Undo},\mathbf{Undo})\hat{}(\mathbf{Succ},\mathbf{Fail},\mathbf{Fail})$. We call v the input *undo* channel, and u the output *oudo* channel. Upon receiving an *undo* signal on v, $[\![G]\!](u,v)(s,f,b)$ will undo all the variable bindings it has performed, and then send an *undo* signal on $\overline{u}$, which will cause another process to undo the variable bindings. An *undo* chain runs from right to left in a conjunction, similarly to the *backtracking* chain as shown in Fig. 1. The modified definitions after incorporating the *undo* channels are shown in Fig. 6.

$$(A \rhd B)(u,v)(s,f,b) \triangleq (\nu u's'f')(A(u,u')(s',f,f') \mid !s'.B(u',v)(s,f',b))$$

$$(A \rhd\!| B)(u,v)(s,f,b) \triangleq (\nu u's'f')(A(u,u')(s',f,f') \mid s'.B(u',v)(s,f',b))$$

$$(A \oplus B)(u,v)(s,f,b) \triangleq (\nu f')(A(u,v)(s,f',b) \mid f'.B(u,v)(s,f,b))$$

$$Cut(f_0)(u_0)(u,v)(s,f,b) \stackrel{def}{=} \overline{s}.(v.\overline{u} + b.([u \neq u_0]\overline{u}.u_0.\overline{f_0} + [u = u_0].\overline{f_0}))$$

$$(x = y)(u,v)(s,f,b) \stackrel{def}{=} (\nu s'f'b')((x =_u y)(s',f',b') \mid (f'.\overline{f} + s'.\overline{s}.(b.\overline{b'}.f'.\overline{f} + v.\overline{b'}.f'.\overline{u})))$$

Fig. 6. Incorporating the Undo Channels

- Since $\rhd$ and $\rhd\!|$ are used for conjunctions, so their definitions are modified only to expand the *undo* chain, in a similar way as expanding the *backtracking* chain.
- The Cut : **(Fail)^(Undo)^(Undo, Undo)^(Succ, Fail, Fail)** reports *success* as usual, then
  - if it receives backtracking request b, then it sends $\overline{u}$ to undo all the variable bindings performed by the goals before the cut. As we shall see in the definition (17) in Fig. 7, the u_0 must be the output *undo* channel of the leftmost goal. So signaling on u_0 means that all the undo's have been finished, and only then the Cut reports $fail$ on $\overline{f_0}$. The match $[u = u_0]$ means that the cut itself is the leftmost goal, so no undo is necessary.
  - if it receives *undo* request v, then it passes out the undo request on $\overline{u}$. This case can happen when the *undo* is caused by other cut or by a Not, defined in Section 5.3.
- The unification agent $(x = y)$: **(Undo, Undo)^(Succ, Fail, Fail)** calls the backtracking unification agent $=_u$, defined in Section 4.3. If $=_u$ fails (f'), then it reports $fail$ $(\overline{f})$. If $=_u$ succeeds (s'), then it reports *success* $(\overline{s})$, but then
  - If a *backtracking* request b is received, then $=_u$ is backtracked $(\overline{b'})$ (which will automatically undo variable bindings as described in Fig. 5). Since a backtracked unification must fails, so after signaling on f', the agent $=$ reports fails$(\overline{f})$.
  - If an *undo* request v is received, then by backtracking the $=_u$ via $\overline{b'}$, all the variable bindings performed by $=_u$ will be undone. After signaling on f', which also means the undo is finished, the agent $=$ passes the undo request to other goals via $\overline{u}$.

5.2 Full Specification of Prolog

As in Section 3, suppose that a predicate P of arity k is defined by m clauses, and the ith clause contains a cut ! as the jth body goal: $'P(X_1, \ldots, X_k) :- B_1, \ldots, B_{j-1}, !, B_{j+1}, \ldots, B_n.'$. Let $\tilde{x}$ denotes $x_1, \ldots, x_k$, then a full specification of Prolog is shown in Fig. 7. Compared to the translation in Section 3, there are only the following changes:

- the agents P and P_i now take additional $k+2$ arguments, i.e.
  P : **(Cell, ..., Cell)^(Undo, Undo)^ (Succ, Fail, Fail)**,
  P_i : **(Cell, ..., Cell)^(Fail)^(Undo, Undo)^(Succ, Fail, Fail)**.
- Cut takes additional argument u', which is also the output *undo* channel of leftmost process $[\![B_1]\!]$, satisfying the requirement stated in Section 5.1.

$$P(\tilde{x})(u,v)(s,f,b) \stackrel{def}{=} (P_1(\tilde{x}) \oplus \ldots \oplus P_i(\tilde{x})(f) \oplus \ldots \oplus P_m(\tilde{x}))(u,v)(s,f,b) \quad (16)$$

$$P_i(\tilde{x})(f_0)(u,v)(s,f,b) \stackrel{def}{=} (\nu\ v_1 \ldots v_s)(\nu\ u'v')(NewVar(v_1) \mid \ldots \mid NewVar(v_s) \mid (\ [\![B_1]\!] \rhd \ldots \rhd [\![B_{j-1}]\!] \bowtie Cut(f_0)(u') \bowtie [\![B_{j+1}]\!] \rhd \ldots \rhd [\![B_n]\!]\ )(u',v')(s,f,b) \mid v.\overline{v'}.u'.\overline{u}\ ) \quad (17)$$

$$[\![B(t_1 \ldots t_k)]\!](u,v)(s,f,b) \triangleq (\nu\ y_1 \ldots y_k)(B(y_1,\ldots,y_k)(u,v)(s,b,f) \mid [\![t_1]\!](y_1) \mid \ldots \mid [\![t_k]\!](y_k)\ ) \quad (18)$$

with the following restrictions:

- Suppose $V_1,\ldots,V_s$ are the all shared local variables among $B_1,\ldots,B_n$, then $V_1,\ldots,V_s$ will be translated into $NewVar(v_1) \mid \ldots \mid NewVar(v_s)$ as in (17).
- in (18), any occurrence of shared local variables $(V_1,\ldots,V_s)$ or head variables $(X_1 \ldots X_k)$ in the body goals will be represented directly by the corresponding $v_1 \ldots v_s$ or $x_1 \ldots x_k$ without generating new channels.

Fig. 7. Full Specification of Prolog

- $v.\overline{v'}.u'.\overline{u}$ in definition (17) is used to handle the *undo* request (v) from outside of P_i. Hence we distinguish between the *uodo* request set by the inside Cut which should be stopped at $[\![B_1]\!]$, and the *undo* request (v) from outside of P_i which, after finishing all the *undo*'s of this clause (via $\overline{v'}.u'$), should be passed out to other process via $\overline{u}$.

The restrictions in Fig. 7 require that each shared local variable in the body is translated only once and every occurrence of same variable in different subgoals will use the same channel name (of sort **Cell**). If the *ith* clause contains no cut, then use $[\![B_j]\!]$ in place of $Cut(f_0)(u)$ in definition (17) and eliminate the argument (f_0) from P_i. If the *ith* clause is a fact of the form $'P(X_1,\ldots,X_k).'$, then the agent P_i is defined as follows:

$$P_i(\tilde{x})(u,v)(s,f,b) \stackrel{def}{=} \overline{s}.(b.\overline{f} + v.\overline{u})$$

which is also the definition for *true*. As an example, the translation of append is shown in Fig. 8. The [] is the functor name for the list structure.

$$Append(x,y,z)(u,v)(s,f,b) \stackrel{def}{=} (Append_1(x,y,z) \oplus Append_2(x,y,z))(u,v)(s,f,b)$$

$$Append_1(x,y,z)(u,v)(s,f,b) \stackrel{def}{=} (\nu\ e)(!\overline{e}\ []\ \emptyset \mid ((x=e) \rhd (y=z))(u,v)(s,f,b))$$

$$Append_2(x,y,z)(u,v)(s,f,b) \stackrel{def}{=} (\nu\ x_1 z_1 u)(NewVar(x_1)|NewVar(z_1)|NewVar(w) |(\nu l_1 l_2)(((x=l_1) \rhd (z=l_2) \rhd Append(x_1,y,z_1))(u,v)(s,f,b) | (\nu\ p_1 p_2)(!\overline{l_1}\ []\ p_1 \mid !\overline{p_1}\ w\ p_2 \mid !\overline{p_2}\ x_1\ \emptyset) | (\nu\ q_1 q_2)(!\overline{l_2}\ []\ q_1 \mid !\overline{q_1}\ w\ q_2 \mid !\overline{q_2}\ z_1\ \emptyset)))$$

Fig. 8. Encoding the *Append* predicate

5.3 Negation as Failure and Other Prolog's Primitives

Based on the principle of *negation as failure*, we can define a control operator Not for Prolog negation *not* (\+).

$$[\![not(P)]\!] \triangleq Not([\![P]\!])$$

$$(Not(A(\tilde{x})))(u,v)(s,f,b) \triangleq (\nu u'v's'f'b')(\ A(\tilde{x})(u',v')(s',f',b') \mid (s'.\overline{v'}.u'.\overline{f}+f'.\overline{s}.(b.\overline{f}+v.\overline{u}))\ )$$

Agent $Not(A(\tilde{x}))$ first calls $A(\tilde{x})$: if $A(\tilde{x})$ fails (f'), then $Not(A(\tilde{x}))$ reports *success* ($\overline{s}$) and then handles possible backtracking request ($b.\overline{f}$) or passes undo request ($v.\overline{u}$); if $A(\tilde{x})$ succeeds (s'), then $Not(A(\tilde{x}))$ reports *fail* ($\overline{f}$) after undoing possible variable bindings (because of $A(\tilde{x})$) via $\overline{v'}.u'$.

Prolog's condition operator (P–> Q; R), as well as *true* and *fail*, are specified in π-calculus as follows:

$$[\![P\text{–>}Q;R]\!](u,v)(s,f,b) \triangleq (\nu u'v')((([\![P]\!] \triangleright\!| Cut(f)(u') \triangleright\!| [\![Q]\!]) \oplus [\![R]\!])(u',v')(s,f,b) \mid v.\overline{v'}.u'.\overline{u}\ )$$

$$True(u,v)(s,f,b) \stackrel{def}{=} \overline{s}.(b.\overline{f}+v.\overline{u})$$

$$Fail(u,v)(s,f,b) \stackrel{def}{=} \overline{f}$$

For the similar reason as in the definition (17) of Fig. 7, $v.\overline{v'}.u'.\overline{u}$ is used in defining the condition agent.

6 Future and Related Works

Ross's CCS Semantics of Prolog The work described here has been more or less influenced by Ross's works on the CCS semantics of Prolog control [RS91, Ros92a] and on the π-calculus semantics of logical variables [Ros92b]. However, our approach, described in Section 3, *exactly* models Prolog's left-right sequential control while [RS91, Ros92a] does not. Namely, given a conjunction goal (P, Q), its corresponding π-calculus agent $[\![P]\!] \triangleright [\![Q]\!]$ works in such a way that the *success* of $[\![P]\!]$ will invoke $[\![Q]\!]$, and at the same time the agent $[\![P]\!]$ will suspend until a backtracking request is received from $[\![Q]\!]$. However, the corresponding CCS agent $[\![P]\!] \triangleright [\![Q]\!]$ in [RS91, Ros92a] works in a different way that the *success* of $[\![P]\!]$ will invoke $[\![Q]\!]$, but at the same time the agent $[\![P]\!]$ can concurrently works to find alternative solutions to invoke another copy of $[\![Q]\!]$. Hence a certain kind of *or-parallelism* exists, so it is unclear how unification can be incorporated (at the process level) into such a scheme since *or-parallelism* usually requires multi-environments.

The approach of this paper is also simpler since only two[3] control operators are needed, corresponding to Prolog's sequential-and and sequential-or, while [Ros92a] uses five control operators. This paper specifies not only Prolog's control but also unification, while [Ros92a] does not specify unification as process. Although a π-calculus semantics of unification has been attempted in [Ros92b], it fails to work for some examples, such as the example of unification of four variables in Fig. 4.

[3] $\triangleright\!|$ is only an optimization of $\triangleright$ in the sense that $\triangleright$ works well in the place of $\triangleright\!|$, as described in Section 3.

Warren's Abstract Machine The representation of logical variables and the approach for unification described in Section 4 are similar to those in the WAM [War83, AK91], especially in the way of using reference chains for binding variables. However, these two approaches are at different levels, i.e. algebraic process level and abstract instruction level. Since the WAM is much closer to an actual implementation than the π-calculus specifications, it necessarily has more complicate and explicit notation of environment stack.

Evolving Algebra The comparison of this work with other semantics specifications of Prolog, especially the work in evolving algebra by Börger and Rosenzweig [BR90, BR92], will be interesting.

Continuation-Passing Style The specification presented in this paper is similar in spirit to the continuation-passing style used in functional programming implementation of Prolog [Hay87, EF91]. In $P(s, f, b)$, the s can be treated as the address of the *success* continuation process which is invoked when P sends $\overline{s}$; the f can be treated as the address of *failure* continuation process which is invoked when P sends $\overline{f}$; and the b is used to pass the address of *failure* continuation to the next process in the conjunction since b is the same channel as the *failure* channel of next process, as discussed in Section 3. Instead of passing the continuation itself as in most continuation-passing styles, only the address of the continuation is passed in our approach.

Concurrent Logic Programming Languages As the π-calculus is a model of concurrent computation, the approaches of this paper can be extended to specify the semantics of the family of concurrent logic programming languages. As a sequel, some results have been achieved in specifying the flat versions of these languages, such as Flat Parlog[CG86] and Flat GHC[Ued86, Sha89], which do not require multi-environment for *committed or-parallelism* and whose unifications are eventual in the sense that no backtracking is required. The *and-parallelism* is modeled by using the π-calculus concurrent composition (|) processes. The *committed or-parallelism* is modeled in two steps: the *or-parallelism* part is modeled by using concurrent composition (|) processes while the *committed non-determinism* part is modeled by the non-determinism of the π-calculus summation (+) process. We are also successful in specifying a deadlock-free concurrent unification in the π-calculus by associating each logical variable with a lock.

High-Order Features Using the techniques described in [Mil91], we can replace all π-calculus agent definitions with replications. Hence, a predicate definition is now translated into a persistent process which can be called via a unique channel name (served as the address of the process) associated with it. Since a channel name can be passed around, we might be able to extend this technique to specify high-order logic programming languages [Mil90a] in π-calculus. Some preliminary results show this is promising.

7 Conclusion

This paper presents a concise π-calculus specification of Prolog that combines a continuation-passing style specification of control with a WAM style specification of logical variables and unification. Several examples have been tested successfully using a π-calculus interpreter written by the author. Given a π-calculus specification of Prolog, the π-calculus bisimulation theory [Mil89, MPW92b] may be a promising tool in Prolog program transformation and reasoning, following lines developed by Ross in [Ros92a, RS91]. Since the π-calculus is a low level calculus with a simple computation mechanism, it is easy to imagine that a π-calculus specification of Prolog may yield an actual implementation of Prolog.

Acknowledgments: I own many thanks to Dale Miller who directed me to this project and contributed a lot of important ideas presented in this paper. I also thank Srinivas Bangalore for helpful discussions. This project has been funded in part by NSF grants CCR-91-02753 and CCR-92-09224.

References

[AK91] H. Ait-Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction.* MIT Press, 1991.

[BR90] Egon Börger and Dean Rosenzweig. From prolog algebras towards wam – a mathematical study of implementations. In *Computer Science Logic (LNCS 533)*, pages 31–66. Springer-Verlag, 1990.

[BR92] E. Börger and D. Rosenzweig. Wam algebras– a mathematical study of implementation part 2. In *Logic Programming (LNCS 592)*, pages 35–54. Springer-Verlag, 1992.

[CG86] K.L. Clark and S. Gregory. Parlog: Parallel programming in logic. *ACM Trans. Prog. Lang. Syst.*, 8(1):1–49, January 1986.

[EF91] C. Elliott and Pfenning F. A semi-functional implementation of a high-order logic programming language. In P. Lee, editor, *Topics in Advanced Language Implementation*, pages 287–325. The MIT Press, 1991.

[Hay87] C.T. Haynes. Logic continuations. *Journal of Logic Programming*, 4(2):157–176, 1987.

[Mil89] R. Milner. *Communication and Concurrency.* Prentice Hall, 1989.

[Mil90a] D. Miller. Abstractions in logic programming. In P. Odifreddi, editor, *Logic and Computer Science*, pages 329–359. Academic Press, 1990.

[Mil90b] R. Milner. Function as processes. Technical Report 1154, INRIA, Sophia Antipolis, February 1990.

[Mil91] R. Milner. The polyadic π-calculus: A turorial. Technical Report ECS-LFCS-91-180, LFCS, University of Edinburgh, 1991.

[MPW92a] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, i. *Information and Computation*, 100(1):1–40, 1992.

[MPW92b] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, ii. *Information and Computation*, 100(1):41–77, 1992.

[Ros92a] B.J. Ross. *An Algebraic Semantics of Prolog Control.* PhD thesis, University of Edinburgh, Scotland, 1992.

[Ros92b] B.J. Ross. A π-calculus semantics of logical variables and unificatin. In *Proc. of North American Process Algebra Workshop*, Stony Brook, NY, 1992.

[RS91] B.J. Ross and A. Smaill. An algebraic semantics of prolog program termination. In *Eighth International Logic Programming Conference*, pages 316 – 330, Paris, France, June 1991. MIT Press.

[Sha89] E. Shapiro. The family of concurrent logic programming languages. *ACM Computing Surveys*, 21(3):412–510, September 1989.

[Ued86] K. Ueda. Guarded horn clause. In *Logic Programming'85*, pages 168–179. LNCS 221, Springer-Verlag, 1986.

[Wal90] D. Walker. π-calculus semantics of object-oriented programming languages. Technical Report ECS-LFCS-90-122, LFCS, University of Edinburgh, 1990.

[War83] D.H.D Wareen. An abstract prolog instruction set. Technical Report Note 309, SRI International, Menlo Park, CA, 1983.