A π–calculus Semantics of Logical Variables and Unification*

Brian J. Ross
Department of Computer Science
Brock University
St. Catharines, Ontario, Canada L2S 3A1
bross@cosc.brocku.ca

Abstract

A π–calculus semantics of terms and logical variables, environment creation *visavis* term copying and variable refreshing, and sequential unification is presented. The π–calculus's object–oriented approach to modelling evolving communication structures is used to model the evolving communication environment found in concurrent logic program computations. The novelty of this semantics is that it explicitly models logic variables as active channels. These channels are referenced by π–calculus channel labels, and when used in concert with the ν restriction operator, model variable scopes and environments. Sequential unification without occurs check is modelled by traversing term expressions, and binding variables to terms as appropriate. The π–calculus is well-suited for this, as its object–oriented view of concurrency permits the modelling of the object passing and variable redirection that occurs during unification. This semantics is a central component of a more comprehensive operational semantics of concurrent logic programming languages currently being developed.

1 Introduction

The π–calculus is a process algebra suitable for modelling concurrent networks with evolving communicative structures [MPW89a, MPW89b, Mil91]. The π–calculus is similar to Milner's earlier CCS [Mil89], except that it is embellished with channel-label passing, which gives an object–oriented view of concurrency. The ability to treat channels as objects greatly enriches the descriptive power of the formalism, which can be seen by its modelling of the λ–calculus in [MPW89a].

*In *First North American Process Algebra Workshop, Springer-Verlag, 1993, S. Purushothaman and A. Zwarico (eds.), pp. 216-230.*

This paper introduces a π–calculus semantics of logical variables and unification. The π–calculus is ideal for modelling concurrent logic programming languages. Concurrent logic programs are characterised as networks of concurrent processes which communicate with one another using shared logical variables. As computations proceed, the unification procedure binds logical variables to various term structures. These bindings alter the communication environment on which the interacting processes depend. The π–calculus concept of dynamic communication topologies is used to model the evolving communication environment found in concurrent logic program computations. This model serves as a basis on which more complex concurrent logic program control semantics can be built upon.

Section 2 reviews the logic programming domain being modelled in the paper, as well as some basic ideas behind the π–calculus. A π–calculus semantics of terms and logical variables, environments, and sequential unification is given in section 3. Some examples are in section 4. A discussion concludes the paper in section 5.

2 Review

2.1 Some logic programming concepts

The semantics in this paper models the data domain used in logic programming and term rewriting. This section will briefly review the essentials needed for developing an intuition of the domain being modelled. The following is found in greater depth in [Llo84].

Terms are defined recursively as follows: (i) a variable is a term; (ii) a constant is a term; (iii) if f is an n-ary function, and $t_1, ..., t_n$ are terms, then $f(t_1, ..., t_n)$ is a term. Constants are often considered to be functions of arity 0. Variables are written in upper-case, and constants and functions in lower-case. A shorthand notation for a term of arity > 0 is $f(\tilde{t})$. The variables in a term $\tilde{t}$ are denoted $Vars(\tilde{t})$.

A substitution θ is a finite set of the form $\{v_1 \leftarrow t_1, ..., v_n \leftarrow t_n\}$, where the v_i's are distinct variables and the t_i's are terms. Each $v_i \leftarrow t_i$ is called a binding for v_i. If E is a term, then $E\theta$ is the term obtained from E by simultaneously replacing each occurence of the variable v_i in E by the term t_i as found in θ. A variable-pure substitution is one where all the t_i's are variables.

Let $\theta = \{u_1 \leftarrow s_i, ..., u_m \leftarrow s_m\}$ and $\sigma = \{v_1 \leftarrow t_i, ..., v_m \leftarrow t_m\}$.

The composition $\theta\sigma$ is the substitution obtained from the set

$$\{u_1 \leftarrow s_1\sigma, ..., u_m \leftarrow s_m\sigma, v_1 \leftarrow t_1, ..., v_n \leftarrow t_n\}$$

by deleting any binding $u_i \leftarrow s_i\sigma$ for which $u_i = s_i\sigma$ and deleting any binding $v_j \leftarrow t_j$ for which $v_j \in \{u_1, ..., u_m\}$.

Let E be a term and V be the set of variables occurring in E. A renaming substitution for E is a variable-pure substitution $\{x_1 \leftarrow y_1, ..., x_n \leftarrow y_n\}$ such that $\{x_1, ..., x_n\} \subseteq V$, the variables y_i are distinct, and $(V - \{x_1, ..., x_n\}) \cap \{y_1, ..., y_n\} = \emptyset$.

Let S be a finite set of terms. A substitution θ is called a unifier for S if $S\theta$ is a singleton. A unifier θ for S is called a most general unifier (mgu) for S if, for each unifier σ of S, there exists a substitution γ such that $\sigma = \theta\gamma$. For example, letting X, Y be variables, $\{p(f(X), Z),\ p(Y, a)\}$ has the mgu $\theta = \{Y \leftarrow f(X), Z \leftarrow a\}$, and their common instance is $p(f(X), a)$. On the other hand, $\{p(f(X), a),\ p(Y, f(c))\}$ is not unifiable, as the second arguments cannot unify.

When computing a unifier for a set S, the disagreement set of S is computed in the following way. Locate the leftmost symbol position at which not all expressions in S have the same symbol, and extract from each expression in S the subexpression beginning at that symbol. The set of all such subexpressions is the disagreement set.

An algorithm for determining the most general unifier is the following. S denotes a finite set of terms.

1. Put $k = 0$ and $\sigma_0 = \epsilon$ (the empty subsitution).
2. If $S\sigma_k$ is a singleton, then stop with σ_k as the mgu of S. Otherwise find the disagreement set D_k of $S\sigma_k$.
3. If there exist v and t in D_k such that v is a variable that does not occur in t, then put $\sigma_{k+1} = \sigma_k\{v \leftarrow t\}$, increment k, and go to step 2. Otherwise S is not unifiable, so stop.

The test in step 3 for occurence of a variable v in the term t is called an *occur check*. This step is necessary for preserving the soundness of unification. However, it is computationally expensive, and is normally ignored in Prolog [CM81].

2.2 The π–calculus

The monadic π–calculus of [MPW89a, MPW89b] will be used. We will embellish it with some devices from a later incarnation in [Mil91], and with

$$\textbf{Comm}: \quad (\cdots + x(y).P) \mid (\cdots + \overline{x}z.Q) \rightarrow P\{z/y\} \mid Q$$

$$\textbf{Par}: \quad \frac{P \rightarrow P'}{P|Q \ \rightarrow \ P'|Q} \qquad \textbf{Res}: \quad \frac{P \rightarrow P'}{(\nu x)P \rightarrow (\nu x)P'}$$

$$\textbf{Struct}: \quad \frac{Q \equiv P \quad P \rightarrow P' \quad P' \equiv Q'}{Q \rightarrow Q'}$$

Figure 1: Reduction and inference rules for π–calculus

some notational conveniences from CCS [Mil89]. The reader is referred to these sources for detailed treatments.

Like CCS, the π–calculus is a process algebra which models concurrency via interleaving. The π–calculus differs from CCS in its ability to pass labels as objects along specified channels. These labels can be either constants or other channel labels; the π–calculus does not distinguish label usage. For example, the expression

$$(\nu s)\ s(x).P \ \mid \ \overline{s}y.Q$$

passes label y (on the right) through channel s to the prefix of P. After applying a reduction rule (described below), the expression reduces to:

$$(\nu s)\ P\{y/x\} \ \mid \ Q$$

The label y replaces x in the expression P. The ν term hides s in the expression. The power of the π–calculus comes from the ability to pass channels themselves. For example, the expression

$$\begin{array}{lll} & (\nu s\ y)\ s(x).\overline{x}z.P \ \mid \ \overline{s}y.Q \ \mid \ y(w).R & \textbf{(1)} \\ \text{reduces to} & (\nu s\ y)\ \overline{y}z.P' \ \mid \ Q \ \mid \ y(w).R & \textbf{(2)} \\ \text{and then to} & (\nu s\ y)\ P' \ \mid \ Q \ \mid \ R' & \end{array}$$

where $R' \equiv R\{z/w\}$. This represents the passing of channel label y as an object in (1), and then using it to transfer label z in (2).

The basic syntax and semantics of the π–calculus is as follows. Let the names $x, y... \in \mathcal{X}$, the agents $P, Q, ... \in \mathcal{P}$, and K range over agent identifiers. Then an agent P is inductively defined by

$$P ::= \sum_{i \in I} \pi_i.P_i \ \mid \ P \mid Q \ \mid \ !P \ \mid \ (\nu x)P \ \mid \ [x = y]P \ \mid \ K(\tilde{y})$$

[Mil91] defines a transition relation $\rightarrow$ using one reduction rule and three inference rules (figure 1). This lean semantic definition is made possible by using a structural congruence relation $\equiv$ over expressions. Some examples of $\equiv$ are

$$\begin{array}{rcll} !P & \equiv & P \mid !P & \\ (\nu x)\ \mathbf{0} & \equiv & \mathbf{0} & \\ (\nu x)\ (\nu y)\ P & \equiv & (\nu y)\ (\nu x)\ P & \\ (\nu x)\ (P \mid Q) & \equiv & P \mid (\nu x)\ Q & : if\ \ x\ \ not\ \ free\ \ in\ \ P \end{array}$$

A detailed semantic account of these π–calculus operators is beyond the scope of this review. An informal description follows.

A summation expression $\sum_{i \in I} \pi_i.P_i$ denotes a choice of possible behaviours in an expression. For example, in $P+Q$, both P and Q are alternate behaviors. I is a finite indexing set. If $I = \emptyset$, then the sum is $\mathbf{0}$, which is the empty process denoting inactivity. The term π is an atomic action, and has one of two forms:

$x(y)$	binds the input from link x to y
$\overline{x}y$	output y on link x

Channel communications which contain empty data result in CCS–style actions. For example, in

$$x.P \mid \overline{x}.Q$$

the action x is treated as a channel communication having no object. The $\mid$ operator denotes concurrent composition, and it models interleaved streams with message passing as generated by its constituent agent arguments. As with CCS, a hidden τ action denotes a handshake between two processes whose matching π prefixes are of opposite polarities. The semantics of $\mid$ accounts for the passing of labels on channels within π atoms. Infinite replication is denoted by $!P$. This expression denotes the infinite generation of P:

$$!P \ = \ P \mid P \mid \cdots \mid P \mid !P$$

Label restriction is denoted by (νx), where x is one or more label names. Matching is denoted by $[x = y]P$. The agent P proceeds only as long as the labels contained in x and y are identical. A useful abbreviation for a set of matches is:

$$\begin{array}{rcl} x : [y_1 \Rightarrow A, y_2 \Rightarrow B, ..., else \Rightarrow Z] & \equiv & [x = y_1]A \ \mid \ [x = y_2]B \ \mid \\ & & \cdots \ \mid \ \forall_i [x \neq y_i] Z \end{array}$$

where all y_i are unique. An agent definition is invoked via the expression $K(\tilde{y})$, where K ranges over agent labels. Associated with each agent identifier K is an expression

$$K(x_1, ..., x_k) \stackrel{\text{def}}{=} E$$

where the x_i labels are distinct.

A notational convenience is the use of a CCS–style action renaming function. This can be implemented in the π–calculus as:

$$A[x/y] \quad \equiv \quad (\nu y)\ (A \mid !(y(w).\overline{x}w))$$

In addition, some useful abbreviations are:

$$\begin{array}{lcl} \pi & \equiv & \pi.\mathbf{0} \\ x(y_1 \cdots y_n) & \equiv & x(w).w(y_1).\cdots.w(y_n) \\ \overline{x}y_1 \cdots y_n & \equiv & (\nu w)\ \overline{x}w.\overline{w}y_1.\cdots.\overline{w}y_n \end{array}$$

Finally, two sequencing operators will be used in the paper:

$$\begin{array}{rcl} P\ \ Before\ \ Q & \stackrel{\text{def}}{=} & (\nu\ d)\ (P[d/done] \mid d.Q) \\ P \bullet Q & \stackrel{\text{def}}{=} & (\nu\ t')\ (P[t'/t] \mid t'.Q) \end{array}$$

In $Before$ (used in the appendix), P must produce the action $\overline{done}$ as the last action before it terminates, which triggers Q's execution. The $\bullet$ operator is like $Before$, but sequences on t.

3 Semantics

3.1 Terms

Conceptually, the labels used by the semantics fall into one of three categories:

(i) User–defined constants $\mathcal{C}$
(ii) Reserved constants $\mathcal{R} = \{\phi, \xi, t, f, get, free\}$
(iii) Channel labels $\mathcal{X}$

The set of user-defined constants $\mathcal{C}$ is a finite predefined set built from the constants in a logic program. The reserved labels $\mathcal{R}$ are distinguished labels, and can be considered to be uniquely coloured to separate them from the

elements of $\mathcal{C}$. The ϕ label is an argument tuple delimiter, and ξ is used in the definition of logical variables. The labels t and f denote logical true and false respectively, and in the context of this paper can be taken to mean success and failure. *get* and $free$ are semaphore signals. Channel names are generic labels. Labels are often indexed (eg. u_2, t').

The denotation of terms and data structures is similar to that used in [MPW89a], except that, instead of using a Lisp–style list structure, a flat static–length graph data structure is adopted. This static structure suffices because pure logic program terms have static arities. Letting $\mathtt{k} \in \mathcal{C}$, t_i be terms, x be a logical variable term, and $[\![\ \ ]\!]_t$ be a term translation function, then terms are recursively modelled in figure 2. The u argument is used as a channel on which to transmit the term structure. The communication of ϕ delimits the end of the term. The replication "!" is used so that terms can be repeatedly read by other expressions; otherwise, it is ephemerally read once. The figure shows a partial translation of logical variables that is completed in section 3.2.

Let E be an agent identifier, and let the term t be an intended argument. The following abbreviation is convenient.

$$E(..., \underline{t}, ...) \ \equiv \ (\nu u)\, E(..., u, ...) \ \mid \ [\![\, t \,]\!]_t(u) \qquad for\ some\ new\ u$$

3.2 Logical Variables and Environments

An environment is a collection of computational mechanisms which affect the course of a computation. In this paper, an environment incorporates two aspects: (i) general computational mechanisms (processes, programs), and (ii) the context within which logical variables are defined. En-

$$[\![\, \mathtt{k}(t_1, ..., t_n) \,]\!]_t(u) \ = \ (\nu v x_1 ... x_n)\, !\,\overline{u}v \mid !\overline{v}\,\mathtt{k}x_1 ... x_n\phi \ \mid \ [\![\, t_1 \,]\!]_t(x_1) \ \mid \cdots \mid \ [\![\, t_n \,]\!]_t(x_n) \quad (n > 0)$$

$$where\ x_i \ \equiv \ \begin{cases} new\ label\ v_i & : t_i\ is\ non-variable\ term \\ t_i & : t_i\ is\ a\ variable \end{cases}$$

$$[\![\, \mathtt{k} \,]\!]_t(u) \ = \ (\nu v)\, !\,\overline{u}v \mid !\overline{v}\,\mathtt{k}\phi$$

$$[\![\, x \,]\!]_t(u) \ = \ \begin{cases} NewVar(x) & : variable\ x\ not\ translated\ yet \\ (nothing) & : otherwise \end{cases}$$

Figure 2: Term translation

$$
\begin{array}{lcl}
NewVar(x) & \stackrel{\text{def}}{=} & (\nu w)\ Var(x,w)\ \mid\ !\overline{w}\xi \\
Var(x,y) & \stackrel{\text{def}}{=} & \overline{x}y.Var(x,y) + x(w).w(z).z : [\xi \Rightarrow Var(x,w), \\
 & & \qquad\qquad\qquad\qquad\qquad\qquad else \Rightarrow Set(x,w)] \\
Set(x,y) & \stackrel{\text{def}}{=} & \overline{x}y.Set(x,y)
\end{array}
$$

Figure 3: Logic variable channels

vironments therefore include a representation of process memory, and determine the scope or communication limits of logical variables. Semantically, an environment is an expression which contains logical variable definitions and their scopes, as well as any other mechanisms which can access these variables.

The view taken here is that logical variables are channels on which reading and writing can take place. These communications are restrained according to the state of the variable, and variables are considered to be "write–once". If one reads from an unbound logical variable, then some sort of communication will be given by the variable to indicate its unbound state. On the other hand, once a variable is bound to a term, then reading from it results in a communication of that term. Similarly, writing a term to an unbound variable results in its being bound to it. Such a term can be a data structure or another variable, and if the latter, a notion of memory or structure sharing is required.

A logical variable has two states, unbound and bound, which correspond respectively to the two agents Var and Set in figure 3. Variable channels define an active environment, in the sense that they actively communicate with other mechanisms. Variables are initialized in the Var state, which occurs with the use of $NewVar$ during the environment definition. They stay in Var until they are bound to a non–variable term, at which time they are managed by Set. In both states, the label x is a bidirectional channel. Reading from the channel will result in either the "unbound" signal ξ being emitted in Var, or the term structure being communicated in Set. When in the Var state, a variable can be reset to another unbound variable an indefinite number of times. It can only be set to a non–variable term structure once. This can be seen in the definition of Set, which disallows any further writing. An important feature of the denotation of logical variables is that the variable channels indirectly point to their bound structures. This

will be the basis for unification in section 3.3.

When translating a term, each logical variable will have a unique bidirectional channel defined for it, which is denoted a private channel label. Each of these channels has an associated agent expression which suitably defines the communication characteristics of logical variables in the language being modelled. Consider a term $\mathtt{k}(\tilde{t})$ that contains within it a set of logical variables $\{v_1, ..., v_n\}$. The semantic translation of $\mathtt{k}(\tilde{t})$ is:

$$[\![\, \mathtt{k}(\tilde{t}) \,]\!](u) \;=\; (\nu v_1 ... v_n)\, [\![\, \mathtt{k}(\tilde{t}) \,]\!]_t(u) \;\mid\; NewVar(v_1) \;\mid\; \cdots \;\mid\; NewVar(v_n)$$

$NewVar$ (see figure 3) initializes a new logical variable. In $NewVar(v_i)$, the label v_i denotes the variable channel used to access the logical variable, and will initially communicate ξ, which denotes its being uninitialized. The restriction operator ν directly denotes variable scope, thereby defining a local environment. If this restriction is removed, then these variables can communicate outside of this expression.

Multiple environments are easily represented. Two terms $\mathtt{t}(u)$ resident in separate environments are denoted:

$$((\nu u)\, [\![\, \underline{\mathtt{t}(u)} \,]\!]_t \;\mid\; \mathcal{E}_1) \;\mid\; ((\nu u)\, [\![\, \underline{\mathtt{t}(u)} \,]\!]_t \;\mid\; \mathcal{E}_2)$$

Restriction delimits the scope as expected, and recursive definition of such terms within agent expressions are handled by label renaming in the π–calculus.

Environments are normally defined by the structure of a program. When defining the semantics of utilities such as unification, however, it is useful to be able to control the creation of environments to some extent. In particular, the ability to rename variables is useful, as it permits tentative operations to be performed on terms without destroying the originals. This is akin to creating temporary local memory, and copying desired terms into it for manipulation and testing. A term copying operator $\rightsquigarrow$ is defined for this purpose. Letting $\tilde{t}$ range over term expressions, then

$$\textbf{Copy} \quad \overline{u \rightsquigarrow v \mid [\![\, \tilde{t} \,]\!]_t(u) \mid \mathcal{E} \;\xrightarrow{\tau}\; [\![\, \tilde{t} \,]\!]_t(u) \mid [\![\, \tilde{t} \,]\!]_t(v) \mid \mathcal{E}}$$

Here, u is a channel which communicates some term to be copied, and v is to be a channel on which the copied term is to be communicated. After the transition, the environment is supplemented by the definition of a copy of this term on channel v, which uses fresh variables as part of the term retranslation. The net result is that a new term is output on channel v,

which is identical to the original term on channel u, except that it refers to a structure with new variables.

The intuitive definition of $\rightsquigarrow$ in **Copy** above is an abstract account of its behavior. The $\rightsquigarrow$ operator is in fact modellable in the π–calculus, and one possible definition for it is in appendix A. (This definition of $\rightsquigarrow$ is not particularly pleasing, however, and a more aesthetic definition is desirable.) The idea behind any implemenation of term copying is to recursively traverse the term as translated in section 3.1, and while doing so, communicate a copy which reflects the original term's structure, while generating refreshed variables.

3.3 Unsafe sequential unification

The theoretical elegance of unification belies the complex procedural considerations encountered when deriving an operational semantics for it. An $=$ agent performs sequential term unification without occurs check. It takes two terms as arguments, and attempts to unify them in three steps:

1. The argument structures are copied into "safe" private memory.
2. Unification is attempted on the private copies.
3. If step 2 is successful, the original structures are unified. Otherwise, the whole unification step fails.

Given that logic variables are modelled as non-invertable channels, bindings to them are permanent. We do not want to bind logic variables unless it is certain that doing so is indeed desired. Therefore, in step 2, copies of the terms are first unified, which is a conditional pre–unification performed before the actual one.

The intended behaviour of $=$ is defined by these operator–level transitions:

$$\textbf{Unify}: \quad \frac{}{\underline{\tilde{t}_1} = \underline{\tilde{t}_2} \mid \mathcal{E} \xrightarrow{\tau.\bar{t}} \mathcal{E}\theta} \qquad\qquad \textbf{Fail}: \quad \frac{}{\underline{\tilde{t}_1} = \underline{\tilde{t}_2} \mid \mathcal{E} \xrightarrow{\tau.\bar{f}} \mathcal{E}}$$

$\mathcal{E}$ is the environment containing all the other logical variables executing concurrently with the unification operation. $Unify$ models the case when the terms unify, in which case $\bar{t}$ denoting success is first transmitted, followed by the incorporation of the unifying substitution θ into the environment. Failure does not change $\mathcal{E}$, and is denoted by $\overline{f}$.

$$
\begin{aligned}
u = v \;&\stackrel{\text{def}}{=}\; (\nu u' v' t')\; u \rightsquigarrow u' \mid v \rightsquigarrow v' \mid u' \stackrel{2}{=} v'\ [t'/t] \mid t'.\bar{t}.(u \stackrel{2}{=} v) \\
u \stackrel{2}{=} v \;&\stackrel{\text{def}}{=}\; u(x).v(y).x : [\ y \Rightarrow \bar{t}, \\
&\qquad\qquad else \Rightarrow x(a).y(b).a : [\ \xi \Rightarrow \overline{u} y.\bar{t}, \\
&\qquad\qquad\qquad\qquad \alpha \Rightarrow b : [\ \xi \Rightarrow \overline{v} x.\bar{t}, \\
&\qquad\qquad\qquad\qquad\qquad\qquad \alpha \Rightarrow (x \stackrel{r}{=} y), \\
&\qquad\qquad\qquad\qquad\qquad\qquad else \Rightarrow \overline{f}\ ], \\
&\qquad\qquad\qquad\qquad \phi \Rightarrow b : [\ \phi \Rightarrow \bar{t},\ else \Rightarrow \overline{f}\ ]\ ]\ ] \\
a \stackrel{r}{=} b \;&\stackrel{\text{def}}{=}\; a(x).b(y).x : [\ \phi \Rightarrow y : [\phi \Rightarrow \bar{t},\ else \Rightarrow \overline{f}], \\
&\qquad\qquad else \Rightarrow y : [\ \phi \Rightarrow \overline{f},\ else \Rightarrow (x \stackrel{2}{=} y) \bullet (a \stackrel{r}{=} b)\ ]\ ]
\end{aligned}
$$

Figure 4: Unification

The $=$ definition is in figure 4. The terms referenced by u and v are first copied with $\rightsquigarrow$, and these copies are unified using $\stackrel{2}{=}$, which does the actual term traversing and binding of logical variables. If this conditional unification succeeds, then the original terms are unified. Otherwise, the terms are not unifiable, and the unification step ends in failure. In $\stackrel{2}{=}$, α ranges over $\mathcal{C}$. The $\stackrel{2}{=}$ operator performs a unification check on one level of the term structures of the arguments in u and v. Firstly, if x and y (the memory references) are the same, then unification holds trivially. Otherwise, the terms are read, and unification proceeds on the structures. For example, the first $\xi \Rightarrow \overline{u} y.\bar{t}$ expression means that the term at u is unbound, and therefore unifies with the other term; the other term is written to the channel u, thus unifying them. The rest of $\stackrel{2}{=}$ unifies the terms in a case-by-case fashion, and is fairly self-explanatory. The $\stackrel{r}{=}$ operator recursively applies $\stackrel{2}{=}$ to the non–empty tuples of arguments for each term. This is only done if both u and v have the same function name α, which ranges over program constants $\mathcal{C}$. Arity discrepancies result in failure. The correctness of the definitions in figure 4 can be verified by structural induction over terms to be unified.

Note that the recursive expression $(x \stackrel{2}{=} y) \bullet (a \stackrel{r}{=} b)$ in $\stackrel{r}{=}$ is sequential. This could be made parallel if more sophisticated environment contention schemes are modelled.

3.4 Atomic (safe) unification

$$u \stackrel{s}{=} v \stackrel{\text{def}}{=} (\nu t' f')\, \overline{get}.(\, u = v\, [t'/t, f'/f] \mid f'.\overline{f}.\overline{free} \;+\; t'.\bar{t}.\overline{free})$$

Figure 5: Atomic Unification

The definition of $=$ in section 3.3 is not safe. Problems arise when more than one mechanism tries to access and alter the environment simultaneously. This is because the accessing of terms requires traversal of the term structures and accessing of logical variable channels, which are not instrinsically atomic operations.

A safe atomic unification operator $\stackrel{s}{=}$ is defined in figure 5. This operator uses two semaphore signals, *get* and $free$, which permit the locking of the environment during the unification of terms by $=$. The following semaphore can be used in concert with agents which are performing concurrent unifications on the same environment:

$$Semaphore \stackrel{\text{def}}{=} get.free.Semaphore$$

An example of semaphore usage is in section 4.2.

4 Examples

4.1 Single unification call

Given two terms to be unified, if either term contains a variable reference, then the variable channel is defined in the environment $\mathcal{E}$. During unification, when a variable is bound to a new term, its variable channel is set to communicate this bound object. $\mathcal{E}$ will contain channel agents for all the logic variables which, after a successful unification step, will be adjusted to communicate their newly bound objects. To see the results of unification, one inspects the states of these channels by reading from them.

The following is an example of how unification affects the environment. Consider the unification of $t(X, b(Y), a(c))$ and $t(Z, W, a(Z))$. This is denoted:

$$\underline{\mathtt{t}(x, \mathtt{b}(y), \mathtt{a}(\mathtt{c}))} = \underline{\mathtt{t}(z, w, \mathtt{a}(z))} \mid NewVar(x) \mid NewVar(y) \mid NewVar(w) \mid NewVar(z) \mid \mathcal{E} \qquad (\dagger)$$

where $\mathcal{E}$ are other various agents in the environment. Before unification, querying any of the channels $\{x, y, w, z\}$ results in the broadcast of the variable's current state. For example, querying x with the expression

$$x(a).a(b).\overline{out}b \;\mid\; NewVar(x)$$

results in ξ being broadcast on *out*. During unification, $=$ binds the variable agents to their unifying terms. Given that the binding substitution for the above is $\theta = \{X \leftarrow Z,\ W \leftarrow b(Y),\ Z \leftarrow c\}$, then the new environment after unification is:

$$(\dagger) \;\stackrel{\tau.\bar{t}}{\rightarrow}\; Set(x, z) \mid NewVar(y) \mid Set(w, \underline{\mathtt{b}(y)}) \mid Set(z, \underline{\mathtt{c}}) \mid \mathcal{E}$$

Conceptually, this is equivalent to a new environment $\mathcal{E}'$, that has incorporated within it $\mathcal{E}\theta$.

4.2 Concurrently competing unifications

The $\stackrel{s}{=}$ agent incorporates a simple semaphore mechanism for synchronizing unifications which might compete to unify shared variables. Two semaphore signals, $\overline{get}$ and $\overline{free}$, are defined within $\stackrel{s}{=}$, and the following semaphore can be used in conjunction with a call to it:

$$Semaphore \stackrel{\text{def}}{=} get.free.Semaphore$$

Now $\stackrel{s}{=}$ will only proceed to unify two terms when access is permitted using *get*. Consider the expression:

$$(\nu\ get\ \ free)\ \ \underline{t(X)} \stackrel{s}{=} \underline{t(c)} \;\mid\; \underline{s(X)} \stackrel{s}{=} \underline{s(d)} \mid\ Semaphore \;\mid\; \mathcal{E} \qquad (\ddagger)$$

Both these unifications refer to the same X in the environment $\mathcal{E}$. If a contention scheme is not implemented, then it is possible for X to be simultaneously bound to conflicting terms. With a semaphore, only one of the two unifications will be allowed to access $\mathcal{E}$ at one instant in time. Assuming that X is uninitialized, then expanding **(‡)**:

$$(\ddagger) \;\;=\;\; (\nu\ get\ free)\ \ (\mathcal{E}\theta_1 \mid Semaphore) \;\;+\;\; (\mathcal{E}\theta_2 \mid Semaphore)$$

where $\theta_1 = X \leftarrow c$ and $\theta_2 = X \leftarrow d$.

When using multiple environments, separate unifications can be denoted in a single expression:

$$\begin{array}{l}(\nu\ get\ free\ V_1)(\tilde{s}_1 = \tilde{s}_2 \mid \tilde{s}_3 = \tilde{s}_4 \mid Semaphore \mid \mathcal{E}_1) \;\mid \\ (\nu\ get\ free\ V_2)(\tilde{t}_1 = \tilde{t}_2 \mid \tilde{t}_3 = \tilde{t}_4 \mid Semaphore \mid \mathcal{E}_2)\end{array}$$

where $V_i = Vars(\mathcal{E}_i)$. Having separate restrictions on the V_i means that they are mutually exclusive sets of variables ($V_1 \cap V_2 = \emptyset$).

4.3 Merging streams

Parallel logic programs often use logical variables to implement streams (eg. [Sha87], volume 1, part III). This example shows how the π–calculus model of logical variables represent streams, and how two streams can be merged.

Consider the following logic program clause:

$$p([1|W]) :- p(W).$$

Without going into details of the operational semantics of logic programs, when clause p is queried with an unbound logic variable, an infinite stream of 1's is generated on W. For example, the query "? − $p(X)$." results in the infinite list $[1, 1, 1, 1, ...]$ being bound onto X. This occurs because the expression $p(W)$ is a recursive call to p which binds W to the term $[1|W']$ in the recursive call, and this carries on *ad infinitum*[1].

A π–calculus translation of p is

$$P(u) \stackrel{\text{def}}{=} (\nu w)\ s = \underline{[1, w]}\ \mid\ P(w)\ \mid\ NewVar(w)$$

which is invoked by a query expression,

$$P(s)\ \mid\ NewVar(s)$$

Each invocation of agent P results in a new environment containing a fresh variable w. The query expression evolves as follows:

$$\begin{array}{rrl}
 & (\nu tf) & P(s)\ \mid\ NewVar(s) \\
= & (\nu tfw) & s = \underline{[1, w]}\ \mid\ P(w)\ \mid\ NewVar(w)\ \mid\ NewVar(s) \\
\stackrel{\tau}{\rightarrow} & (\nu tfw) & P(w)\ \mid\ NewVar(w)\ \mid\ Set(s, \underline{[1, w]}) \\
\stackrel{\tau}{\rightarrow} & (\nu tfww') & P(w')\ \mid\ NewVar(w')\ \mid\ Set(w, \underline{[1, w']})\ \mid\ Set(s, \underline{[1, w]}) \\
\stackrel{\tau}{\rightarrow} & (\nu tfww'w'') & P(w'')\ \mid\ NewVar(w'')\ \mid\ Set(w', \underline{[1, w'']})\ \mid \\
 & & \quad Set(w, \underline{[1, w']})\ \mid\ Set(s, \underline{[1, w]}) \\
\stackrel{\tau}{\rightarrow} & \ldots &
\end{array}$$

The environment being built reflects the binding $\{S \leftarrow [1, 1, 1, ...]\}$.

[1] The notation $[X|Y]$ is a standard logic program abbreviation for $'.'(X, Y)$, where '.' is a list constructor functor, X is a list element, Y is the tail, and $\emptyset$ is a null list. Thus, $[1, 2, 3] \equiv' .'(1,'.'(2,'.'(3, \emptyset)))$.

Consider two streams as above, where P generates an infinite number of 1's, and Q an infinite number of 2's. The task is to nondeterministically merge these streams into a stream w,

$$(\nu t f u v)\ P(u)\ \mid\ Q(v)\ \mid\ Merge(u,v,w)\ \mid\ \mathcal{E}_0$$

where $\mathcal{E}_0$ defines u, v, and w. One model for $Merge$ is:

$$\begin{aligned} Merge(u,v,w) \stackrel{\text{def}}{=}\ & (\nu t_1 t_2 f x a z)!Nonvar(u)[t_1/t]\ \mid\ !Nonvar(v)[t_2/t]\ \mid \\ & (t_1.(u = \underline{[x|a]}\ \mid\ w = \underline{[x|z]}\ \mid\ Merge(a,v,z)) \\ & +\ t_2.(v = \underline{[x|a]}\ \mid\ w = \underline{[x|z]}\ \mid\ Merge(u,a,z)))\ \mid\ \mathcal{E}_1 \end{aligned}$$

where $\mathcal{E}_1$ defines x, a, and z, and

$$Nonvar(u) \stackrel{\text{def}}{=}\ u(x).x(y).y : [\xi \Rightarrow \overline{f},\ else \Rightarrow \overline{t}]$$

The complication in merging these two streams arises from the fact that the stream variables from P and Q might remain unbound when $Merge$ is invoked. This is because P, Q, and $Merge$ execute in parallel, with no ordering implicit in their communication to one another. $Merge$ is therefore designed so that it only merges bound streams, which is the case when u or v are bound. This is determined by $Nonvar$. The merged stream on w is built iteratively, by nondeterministically choosing bound stream elements from P and Q, and inserting them to the front of w.

5 Discussion

This π–calculus semantics is a central component of a fuller semantics of concurrent logic program languages currently being developed [Ros92a]. The motivation for using the π–calculus is to investigate its appropriateness a kernel language for modelling a wide variety of concurrent logic program phenomena. We plan to use this semantics to refine efficient implementations of concurrent logic languages. So far, the active environment model of this paper is useful for modelling the communication which occurs between processes and logic variables. In addition, the π–calculus can model the control constructs of these languages, a flavour of which can be seen in the merge example of section 4.3. Another avenue being investigated is the modelling of various parallel unification algorithms, rather than the sequential unification done here. To do this, a more detailed contention scheme is necessary. One possibility is to put semaphores around individual logical variables, instead of entire local environments.

The domain modelled here is a simple one, and actual implementations of concurrent logic languages are considerably more complex due to efficiency considerations. For example, many committed logic languages use various variable protection schemes, such as in GHC [Ued86], which does not permit variables to be instantiated in the guard. This could be modelled in the π–calculus by redefining the semantics of logical variable channels, and requires an appropriately redefined unification algorithm. The semantics given here *could* be refined into more sophisticated models which are more amenable to efficient implementation on particular hardware. Given that the π–calculus has a well–founded semantics, encoding abstract machines in it can permit formal analyses of language design. Abstract machines such as that in [Lin84] shares similarities with ours, and could be encoded in the π–calculus if desired.

Two other process algebra models of concurrent logic programs are by Belmesk and Habbas [BH92] and de Boer and Palamidessi [dBP92], and a related approach is by Saraswat and Rinard [SR90]. All these papers take the Herbrand domain and unification to be abstract concepts which are used directly in semantic axioms. This paper differs by modelling terms and unification at the lower level of terms using a kernel of basic π–calculus operators, while still permitting their level of abstraction if desired. Such a level of description is needed when modelling operational characteristics of concurrent languages such as memory contention – concepts which are transparent when too abstract a view is taken.

Beckman uses CCS to model concurrent logic languages [Bec87], and Ross applies CCS to sequential Prolog [Ros92b]. In both these papers, CCS's value passing is not rich enough for directly modelling logical variables. As a result, Beckman redefines the composition "|" operator so that binding substitutions are automatically distributed to the entire environment. This paper shows how such a data domain can be implemented in a concurrent environment. One other related work is by Walker [Wal90], who uses the π–calculus for modelling imperative object–oriented programming languages.

Acknowledgements: Thanks to Robin Milner for hints on how to model logical variables, and to Robert Scott for his helpful advice on concurrent logic programming. This research was done at the University of Victoria (Canada).

References

[Bec87] L. Beckman. *Towards an Operational Semantics for Concurrent Logic Programming Languages*. PhD thesis, Uppsala University, 1987.

[BH92] M. Belmesk and Z. Habbas. A Process Calculus with Shared Variables. *Journal of Computers and Artificial Intelligence*, 1992.

[CM81] W.F. Clocksin and C.S. Mellish. *Programming in Prolog*. Springer-Verlag, 1981.

[dBP92] F. S. de Boer and C. Palamidessi. A Process Algebra of Concurrent Constraint Programming. In *Joint International Conference & Symposium on Logic Programming*, Washington, DC, 1992. MIT Press.

[Lin84] G. Lindstrom. OR-parallelism on applicative architectures. In *2nd International Logic Programming Conference*, Uppsala, 1984.

[Llo84] J.W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 1984.

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

[Mil91] R. Milner. The Polyadic π-Calculus: A Tutorial. Technical Report ECS-LFCS-91-180, LFCS, U. of Edinburgh, 1991.

[MPW89a] R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes, Part I. Technical Report ECS-LFCS-89-85, LFCS, U. of Edinburgh, 1989.

[MPW89b] R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes, Part II. Technical Report ECS-LFCS-89-86, LFCS, U. of Edinburgh, 1989.

[Ros92a] B.J. Ross. A π–calculus Semantics of Committed Logic Program Control, 1992. In preparation.

[Ros92b] B.J. Ross. *An Algebraic Semantics of Prolog Control*. PhD thesis, University of Edinburgh, Scotland, 1992.

[Sha87] E.Y. Shapiro. *Concurrent Prolog vol. 1 and 2*. MIT Press, 1987.

[SR90] V. Saraswat and M. Rinard. Concurrent Constraint Programming. In *Proceedings 17th Principles of Programming Languages*, pages 232–245, San Francisco, 1990.

[Ued86] K. Ueda. *Guarded Horn Clauses*. PhD thesis, University of Tokyo, 1986.

[Wal90] D. Walker. π-calculus Semantics of Object-Oriented Programming Languages. Technical Report ECS-LFCS-90-122, LFCS, U. of Edinburgh, 1990.

A Implementation of $\rightsquigarrow$

Figure 6 contains one possible π–calculus implementation of $\rightsquigarrow$, which uses some list processing utilities in figure 7. In $\rightsquigarrow$, the $\overset{2}{\rightsquigarrow}$ operator traverses the term to be copied, and echoes new instances of term components. Fresh channels and logic variables are generated during the traversal. However, a record must be kept of any new logical variable channels created, since multiple instances of a variable in a term should be denoted by the same fresh copy. Therefore, a list is checked via *mem* whether a variable has already been renamed. If not, then a new variable is created, and the old and new variable labels are saved in the list (via $\overline{add}$). Otherwise, it has been renamed already, and the new variable created previously is used instead. The $\overset{r}{\rightsquigarrow}$ operator recurs on the term arguments.

$$
\begin{aligned}
u \rightsquigarrow v &\stackrel{\text{def}}{=} (\nu\ done\ add\ mem\ m)\ u \overset{2}{\rightsquigarrow} v \mid List \\
u \overset{2}{\rightsquigarrow} v &\stackrel{\text{def}}{=} u(x).x(y).y : [\xi \Rightarrow \overline{mem}x \mid m(a).a : [\phi \Rightarrow (\nu w)\ Var(v,w) \mid !\overline{w}\xi \mid \overline{add}xw, \\
&\qquad\qquad\qquad\qquad\qquad\qquad else \Rightarrow \overline{v}a\ ], \\
&\qquad\qquad\qquad else \Rightarrow (\nu w)\ Set(v,w) \mid !(\overline{w}y.\overline{done}\ Before\ x \overset{r}{\rightsquigarrow} w)\ ] \\
x \overset{r}{\rightsquigarrow} w &\stackrel{\text{def}}{=} x(z).z : [\phi \Rightarrow \overline{w}\phi,\ else \Rightarrow (\nu a)\ z \overset{2}{\rightsquigarrow} a \mid \overline{w}a.\overline{done}\ Before\ x \overset{r}{\rightsquigarrow} w\ ]
\end{aligned}
$$

Figure 6: Term copying

$$
\begin{aligned}
List &\stackrel{\text{def}}{=} (\nu x)\ \overline{x}\phi \mid L(x) \\
L(x) &\stackrel{\text{def}}{=} (\nu z)\ add(y_1,y_2).(L(z) \mid \overline{z}y_1y_2x) \\
&\qquad\qquad + mem(y).(L(x) \mid Member(y,x)) \\
Member(v,x) &\stackrel{\text{def}}{=} x(a,b).a : [\phi \Rightarrow \overline{m}\phi,\ v \Rightarrow \overline{m}b, \\
&\qquad\qquad\qquad\qquad else \Rightarrow x(z).Member(v,z)\ ]
\end{aligned}
$$

Figure 7: List utilities