LMNtal: a language model with links and membranes

Kazunori UEDA[†‡] **Norio KATO**[†]

[†]Dept. of Computer Science,
Waseda University
3-4-1, Okubo, Shinjuku-ku, Tokyo 169-8555, Japan
E–mail:{ueda,n-kato}@ueda.info.waseda.ac.jp
[‡]CREST, Japan Science and Technology Corporation

Abstract

LMNtal (pronounced "*elemental*") is a simple language model based on graph rewriting that uses logical variables to represent links and membranes to represent hierarchies. The two major goals of LMNtal are (i) to unify various computational models based on multiset rewriting and (ii) to serve as the basis of a truly general-purpose language covering various platforms ranging from wide-area to embedded computation. Another important contribution of the model is it greatly facilitates programming with dynamic data structures.

1 Introduction

This work is motivated by two "grand challenges" in computational formalisms and programming languages. One is to have a computational model that unifies various paradigms of computation, especially those of concurrent computation and computation based on multiset rewriting. The other is to design and implement a programming language that covers a variety of computational platforms which are now developing towards both wide-area computation and nanoscale computation. As the first step towards these ends, this paper proposes a language model LMNtal (pronounced "*elemental*") whose design goals are as follows:

1. *simple* — to serve as a computational model as well as the basis of a practical programming language (hence a language *model*).
2. *unifying and scalable* — to unify and reconcile various programming concepts. For instance, LMNtal treats

   (a) processes, messages and data uniformly,

   (b) dynamic process structures and dynamic data structures uniformly, and

   (c) synchronous and asynchronous communication uniformly.

   Also, through such uniformity and resource-consciousness implied by (a) and (b) above, LMNtal is intended to be *scalable*, that is, be applicable to computational platforms of various physical scales.

3. *easy to understand* — since we often use figures to explain and understand concurrent computation and programming with dynamic data structures, the language is designed so that computation can be viewed as diagram transformation.
4. *fast* — optimizing compilation techniques are an important subject of the project, though this paper will focus on basic concepts.

We briefly describe the design background of LMNtal. The first author designed Guarded Horn Clauses (GHC) [14] in mid 1980's, a concurrent language that made use of the power of logical variables to feature channel mobility. Various type systems such as mode and linearity systems were later designed for GHC [15]. A lot of implementation efforts and techniques have been accumulated over the past two decades. Concurrent logic programming was generalized to concurrent constraint programming that allowed data domains other than finite trees, and a concurrent constraint language Janus [13] chose multisets (a.k.a. bags) as an important data domain. Another important generalization was Constraint Handling Rules (CHR) [8] that allowed multisets of atomic formulae in clause heads. CHR was designed as a language for defining constraint solvers, but at the same time it is one of the most powerful multiset rewriting languages.

Given these two extensions, a natural question arises as to whether (the multiset aspect of) the two extensions can be unified or embedded into each other. LMNtal was designed partly as a solution to this question. The language design was first published in [16]. It was then reviewed and revised through intensive discussions, receiving feedback from the implementation effort that ran in parallel. This paper reflects the latest design published in [17].

2 Overview of LMNtal and Related Work

The "four elements" of LMNtal are:
logical links, *multisets*, *nested nodes*, and *transformation*
hence the name LMNtal. This section elaborates these four elements, touching on related work.

1. **Logical links** — Structures of communicating processes can be represented as graphs in which nodes represent processes and links represent communication channels. Likewise, dynamic data structures can be represented using nodes and links. LMNtal treats them uniformly, that is, links represent both one-to-one communication channels between logically neighboring processes and logical neighborhood relations between data cells.

   Two major mechanisms in concurrency formalisms are name-based communication (as in the π-calculus) and constraint-based communication using logical, single-assignment variables (as in concurrent constraint programming [15]). Of these, links of LMNtal are closer to communication using logical variables in that (i) a message sent through a link changes the identity of the link and (ii) links are always private (i.e., third processes cannot access them). The first point is the key difference from the π-calculus. However, they are different from links of concurrent logic/constraint programming and CHR in that LMNtal has no notion of *instantiating* a link variable to a value.

LMNtal links are basically non-directional. However, if links are always followed in a fixed direction to reach partners, the direction could be represented and “reconstructed” using appropriate type systems.

2. **Multisets of nested nodes** — There have been many diverse proposals of computational models equipped with the notion of multisets, early examples of which include Petri Nets and Production Systems. Concurrent processes naturally form multisets; Gamma [2] and Chemical Abstract Machines [3] are two typical computational models based on multiset rewriting; languages based on Linear Logic [10] take advantage of the fact that the both sides of a sequent are multisets; Linda’s tuple spaces are multisets of tuples.

   However, not all of them feature multisets as first-class citizens; many of the programming languages featuring multisets (e.g., Gamma, Linda, CHR) include them in a way different from other data structures. The advantage of having multisets as first-class citizens is that it gives us greater expressive power such as the nesting and the mobility of multisets.

   LMNtal features multiset hierarchies and encapsulation by allowing a multiset of nodes enclosed by a membrane to be a node. Hierarchical multisets can be found in the ambient calculus [4] and the bigraphical model [11], as well as in the fields of formal languages such as the P-system [12] and knowledge representation [6].

   Hierarchization of multisets plays many important rôles, for instance in (i) logical management of computation (e.g., user processes running under administrative processes), (ii) physical management of computation (e.g., region-based memory management), and (iii) localization of computation (i.e., reaction rules placed at a certain “place” of a tree of hierarchy can act only on processes at that place).

3. **Transformation** — LMNtal has a rewrite-rule-based syntax. There has been a lot of work on graph grammars transformation [1], including hierarchical graph transformation [5], but LMNtal’s emphasis is on its design from the programming language point of view. The key design issue has been the treatment of free links.

   Rewrite rules specify reaction between elements of a multiset, but reaction between interlinked elements can be much more efficient (in finding partners) than reaction between unlinked elements.

LMNtal features both channel mobility and process mobility. It allows dynamic reconfiguration of process structures as well as the migration of nested computation.

$$
\begin{array}{rlll}
P ::= & \mathbf{0} & & \text{(null)} \\
\mid & p(X_1,\ldots,X_m) & (m \geq 0) & \text{(atom)} \\
\mid & P, P & & \text{(molecule)} \\
\mid & \{P\} & & \text{(cell)} \dagger \\
\mid & T \mathtt{:-} T & & \text{(rule)} \\
T ::= & \mathbf{0} & & \text{(null)} \\
\mid & p(X_1,\ldots,X_m) & (m \geq 0) & \text{(atom)} \\
\mid & T, T & & \text{(molecule)} \\
\mid & \{T\} & & \text{(cell)} \dagger \\
\mid & T \mathtt{:-} T & & \text{(rule)} \\
\mid & \mathtt{@}p & & \text{(rule context)} \dagger \\
\mid & \mathtt{\$}p\mathtt{[}X_1,\ldots,X_m \mathtt{|} A\mathtt{]} & & \text{(process context)} \dagger \\
\mid & p(*X_1,\ldots,*X_m) & (m > 0) & \text{(aggregate)} \dagger \\
A ::= & \mathtt{[]} & & \text{(empty)} \dagger \\
\mid & *X & & \text{(bundle)} \dagger
\end{array}
$$

Figure 1: Syntax of LMNtal (Lines with daggers (†) are not in Flat LMNtal)

3 Syntax of LMNtal

3.1 Links and Names

First of all, we presuppose two syntactic categories:

- *Links* (or *link variables*), denoted by X. In the concrete syntax, links are denoted by identifiers starting with capital letters.
- *Names* (including numbers), denoted by p. In the concrete syntax, names are denoted by identifiers different from links. The name "=" is the only reserved name in LMNtal.

3.2 Syntax

The two major syntactic categories of LMNtal are processes and process templates. The former is the subject of the language that evolves with program execution. The latter is used in reaction rules and can express local contexts of processes, namely contexts within particular cells.

The syntax of LMNtal is given in Figure 1. As usual, parentheses ( ) are used to resolve syntactic ambiguities. Commas for molecules connect tighter than the ":-" for rules. P and T have several syntactic conditions, as will be detailed in this section. The part of a process not included in any rule is called the *non-rule part* of the process. Cells can be arbitrarily nested. The part of a cell $\{P\}$ or $\{T\}$ not contained in nested cells is called the *toplevel* of $\{P\}$ or $\{T\}$, respectively.

We can think of a subset of LMNtal, *Flat LMNtal*, that does not allow process hierarchies. The syntax of Flat LMNtal does not feature the lines with daggers (†).

The rest of this section explains processes, rules and process templates in more detail.

3.2.1 Processes

A process P must observe the following *link condition*:

Link Condition: Each link in the non-rule part of P can occur *at most twice.*

A link occurring only once in P is called a *free link* of P. A link occurring twice in P is called a *local link* of P. A *closed process* is a process containing no free links.

Intuitively, $\mathbf{0}$ is an empty process; $p(X_1, \ldots, X_m)$ is an atom with m links; P, P is parallel composition (or multiset union); $\{P\}$ is a process grouped by the *membrane* $\{\ \}$; and T `:-` T is a rewrite rule for processes.

An atom $X = Y$, called a *connector*, connects one side of the link X and one side of the link Y.

Note that the link condition never prevents us from composing two processes P_1 and P_2. When each of P_1 and P_2 satisfies the link condition but the composition P_1, P_2 does not, there must be a link occurring twice in one and at least once in the other. Since the former is a local link, we can always α-convert it to a fresh link (Section 4.1). The links used in rules are not considered in the link condition because they are understood to be local to the rules.

3.2.2 Rules and Process Templates

Rules have the form T `:-` T, where the T's are called *process templates.* The first and the second T are called the left-hand side (LHS) and the right-hand side (RHS), respectively.

Process templates have three additional constructs, namely *rule contexts*, *process contexts*, and *aggregates*. *Contexts* in LMNtal refer to the rest of the entities in the innermost surrounding membrane. Rule contexts are to represent multisets of rules, while process contexts are to represent multisets of cells and atoms.

A process context consists of a name $\$p$ and an argument `[`$X_1, \ldots, X_m$ `|` A`]`. The argument of a LHS process context specifies the set of free links that the context must have. X_i denotes a specific link if it occurs elsewhere in the LHS and an arbitrary free link if it does not occur in the LHS. The final component A is called a *residual.* A residual of the form $*V$ receives the bundle of zero or more free links other than $X_1, \ldots, X_m$, and a residual `[]` means that there should be no free links other than $X_1, \ldots, X_m$.

An aggregate represents a multiset of atoms with the same name, whose multiplicity coincides with the number of links represented by the argument bundles.

The precise semantics of these additional contexts will be given in Section 4.

Rules have several syntactic conditions. Firstly, process contexts and rule contexts in a rule must observe the following:

Syntactic Conditions:

1. A rule cannot occur in the LHS of a rule.
2. Aggregates cannot occur in the LHS of a rule.
3. Rule contexts and process contexts occurring in the LHS of a rule must occur within a cell.

Note that the first condition disallows the decomposition of rules. The third condition means that rule contexts and process contexts deal only with local contexts delimited by membranes.

Secondly, rules must satisfy the following *occurrence conditions* on links and other syntactic constructs:

Occurrence Conditions:

1. A link and a bundle occurring in a rule must occur exactly twice in the rule.
2. Links occurring in the argument of a process context must be pairwise distinct.
3. Bundles occurring in the LHS of a rule must be pairwise distinct.
4. A rule context and a process context occurring in a rule must occur exactly once in the LHS and must not occur in another rule occurring inside the rule.
5. The toplevel of each cell occurring in the LHS of a rule may have at most one process context and at most one rule context.

Condition 1 implies that a rule cannot have free links. Condition 2 is imposed because the links specify the *set* of free links to be owned by a process matching the process context. Condition 3 is because a bundle in the LHS of a rule is to receive, rather than compare, a set of free links of the matching process. The "must occur once" condition in Condition 4 means that a rule context or a process context must receive a multiset of rules or a process upon application of the rule, and the "exactly once" condition means that they cannot be used to compare two contexts. Note that rule contexts and process contexts may occur more than once in the RHS of a rule. Condition 5 is to ensure that the values received by rule contexts and process contexts are uniquely determined.

Of the links occurring in a rule L `:-` R, those occurring only in L are consumed links; those occurring only in R are links generated by the rule, and those occurring once in L and once in R are inherited links.

Finally, we introduce several *consistency conditions*:

Consistency Conditions:

1. The residuals of the process contexts with the same name in a rule must be either all empty (`[]`) or all bundles.

2. The arity m of the process contexts with the same name in a rule must coincide.
3. The process contexts having the same bundle must have the same name.
4. For each aggregate $p(*X_1, \ldots, *X_m)$ $(m > 0)$ in a rule, there must be a process context name $\$q$ and each $*X_i$ must occur as the residual of a process context with the name $\$q$ in the rule.

For example, the rule

```
{exch,$a[X,Y|[]]} :- {$a[Y,X|[]]}
```

satisfies Consistency Conditions 1 and 2 (Condition 3 holds vacuously) and says that when a cell contains an atom `exch` and exactly two free links at its toplevel, the two free links are crossed and the atom `exch` is erased.

The rule

```
{kill,$a[|*X]} :- killed(*X)
```

satisfies Consistency Condition 3 (the other conditions hold vacuously) and says that when a cell contains an atom `kill` at its toplevel, the cell is erased and each link crossing the membrane is terminated by a unary atom `killed`.

The above conditions do not allow dynamic composition of rules, but do allow statically determined rules to be spawned dynamically and the set of rules inside a cell to be copied and migrated to another cell. Thus LMNtal enables the cell-wise compilation of the set of rules while providing certain higher-order features.

4 Operational Semantics

We first define structural congruence ($\equiv$) and then the reduction relation ($\longrightarrow$).

4.1 Structural Congruence

We define the relation $\equiv$ on processes as the minimal equivalence relation satisfying the rules shown in Figure 2. Two processes related by $\equiv$ are essentially the same and are convertible to each other in zero steps. Here, $[Y/X]$ is a *link substitution* that replaces X with Y.

(E1)–(E3) are the characterization of molecules as multisets. (E4) allows the renaming (α-conversion) of local names. Note that the link Y cannot occur free in P for the link condition on $P[Y/X]$ to hold. (E5)–(E6) are structural rules that make $\equiv$ a congruence. (E7)–(E9) are concerned with `=`. (E7) says that a self-absorbed loop is equivalent to **0**, while (E8) expresses the symmetry of `=`. (E9) is an absorption law of `=`, which says that a connector can be absorbed by another atom (which can again be a connector). Because of the symmetry of $\equiv$, (E9) says that an atom can emit a connector as well.

$$
\begin{array}{llcll}
\text{(E1)} & \mathbf{0}, P & \equiv & P & \\
\text{(E2)} & P, Q & \equiv & Q, P & \\
\text{(E3)} & P, (Q, R) & \equiv & (P, Q), R & \\
\text{(E4)} & P & \equiv & P[Y/X] & \text{if } X \text{ is a local link of } P \\
\text{(E5)} & P \equiv P' & \Rightarrow & P, Q \equiv P', Q & \\
\text{(E6)} & P \equiv P' & \Rightarrow & \{P\} \equiv \{P'\} & \\
\text{(E7)} & X = X & \equiv & \mathbf{0} & \\
\text{(E8)} & X = Y & \equiv & Y = X & \\
\text{(E9)} & X = Y,\ P & \equiv & P[Y/X] & \text{if } P \text{ is an atom and } X \text{ is a local link of } P
\end{array}
$$

Figure 2: Structural congruence on LMNtal processes

$$
\text{(R1)}\ \frac{P \longrightarrow P'}{P, Q \longrightarrow P', Q} \qquad \text{(R2)}\ \frac{P \longrightarrow P'}{\{P\} \longrightarrow \{P'\}} \qquad \text{(R3)}\ \frac{Q \equiv P \quad P \longrightarrow P' \quad P' \equiv Q'}{Q \longrightarrow Q'}
$$

$$
\text{(R4)}\quad \{X = Y, P\} \longrightarrow X = Y,\ \{P\} \quad (X \text{ and } Y \text{ are distinct and don't occur in } P)
$$

$$
\text{(R5)}\quad X = Y,\ \{P\} \longrightarrow \{X = Y, P\} \quad (X \text{ occurs in the non-rule part of } P)
$$

$$
\text{(R6)}\quad T\theta, (T \mathrel{:-} U) \longrightarrow U\theta, (T \mathrel{:-} U) \quad (T\theta \text{ doesn't contain = in its non-rule part})
$$

Figure 3: Reduction relation on LMNtal processes

4.2 Reduction Relation

Computation proceeds by rewriting processes using rules collocated in the same "place" of the nested membrane structure.

We define the reduction relation $\longrightarrow$ on processes as the minimal relation satisfying the rules in Figure 3. Note that the right-hand side of $\longrightarrow$ must observe the link condition of processes.

Of the six rules, (R1)–(R3) are structural rules. (R1) says that reductions can proceed concurrently based on local rewrite conditions. Fine-grained concurrency of LMNtal originates here. (R2) says that computation within a cell can proceed independently of the exterior of the cell. For a cell to evolve autonomously, it must contain its own set of rules. Computation of a cell without rules will be controlled by rules outside the cell. (R3) incorporates structural congruence into the reduction relation.

(R4) and (R5) deal with the interaction between connectors and membranes. (R4) says that, when a connector in a cell connects two links coming from outside, the cell can expel the connector. (R5) says that, when one argument of a connector is a link entering a cell, the connector itself can go into the cell.

(R6) is the key rule of LMNtal. The substitution θ is to represent what has been received by each process/rule context and what multiset of atoms each aggregate represents. In Flat LMNtal, θ becomes unnecessary and (R6) is simplified to

$$\text{(R6}'\text{)}\quad T, (T \,\texttt{:-}\, U) \longrightarrow U, (T \,\texttt{:-}\, U).$$

(R6′) describes the reaction between a process and a rule not separated by membranes.

Matching between a process and the LHS of a rule under (R6′) should generally be done by α-converting the rule using (E4) and (R3). The whole resulting process, namely $U, (T \,\texttt{:-}\, U)$ and its surrounding context, should observe the link condition, but this can always be achieved by α-converting $T \,\texttt{:-}\, U$ before use so that the new local links don't cause name crash with the context.

The substitution θ in (R6) is represented as a finite set of *substitution elements* of the form β_i/α_i (meaning that α_i is replaced by β_i), and should satisfy the following three conditions. In the third condition, we assume that the occurrences of the process context name $\texttt{\$}p$ in the RHS U are uniquely numbered, and that the function v is a one-to-one mapping from link names and natural numbers to link names.

1. The domain of θ is the set of all rule contexts, process contexts and aggregates occurring in the LHS T or in the non-rule part of the RHS U.

2. For each rule context $\texttt{@}p$ in T, θ should contain $P/\texttt{@}p$, where P is a sequence of rules.

3. For each process context $\texttt{\$}p\texttt{[}X_1, \ldots, X_m \,\texttt{|}\, A\texttt{]}$ in T, the following (i)–(iii) hold, where P is a process whose free links are $\{X_1, \ldots, X_{m+n}\}$ (if $A = \texttt{[]}$ then $n = 0$; otherwise $n \geq 0$), whose local links are $\{Z_1, \ldots, Z_\ell\}$, and which has no rules outside cells.

   (i) a free link of T occurring in an atom doesn't occur in P.

   (ii) If $A = \texttt{[]}$ then

   (a) $P/\texttt{\$}p\texttt{[}X_1, \ldots, X_m\texttt{]} \in \theta$

   (b) For $\texttt{\$}p\texttt{[}Y_1, \ldots, Y_m\texttt{]}$ with the number h in the RHS U,

   $$P[v(Z_1, h)/Z_1, \ldots, v(Z_\ell, h)/Z_\ell, Y_1/X_1, \ldots, Y_m/X_m] \\ /\, \texttt{\$}p\texttt{[}Y_1, \ldots, Y_m\texttt{]} \in \theta$$

   (iii) If $A = *V$ then

   (a) $P/\texttt{\$}p\texttt{[}X_1, \ldots, X_m \,\texttt{|}\, *V\texttt{]} \in \theta$

   (b) $v(V, i) = X_{m+i}$ for $1 \leq i \leq n$

   (c) For $\texttt{\$}p\texttt{[}Y_1, \ldots, Y_m \,\texttt{|}\, *W\texttt{]}$ with the number h in the RHS U,

   $$P[v(Z_1, h)/Z_1, \ldots, v(Z_\ell, h)/Z_\ell, Y_1/X_1, \ldots, Y_m/X_m, \\ v(W, 1)/X_{m+1}, \ldots, v(W, n)/X_{m+n}] \\ /\, \texttt{\$}p\texttt{[}Y_1, \ldots, Y_m \,\texttt{|}\, *W\texttt{]} \in \theta$$

   (d) For each $q(*V_1, \ldots, *V_k)$ in the non-rule part of U such that some V_i is V,

   $$(\ q(v(V_1, 1), \ldots, v(V_k, 1)),\ \ldots, q(v(V_1, n), \ldots, v(V_k, n))\ ) \\ /\, q(*V_1, \ldots, *V_k) \in \theta$$

When the RHS contains a process context of the same name, say $\$p$`[`$X'_1, \ldots, X'_m$`|`$*V'$`]`, a process isomorphic to the process matched by the corresponding context in the LHS is created. Its free links corresponding to $X_1, \ldots, X_m$ are connected to $X'_1, \ldots, X'_m$, respectively, and the free links corresponding to $*V$ are connected to the links represented by $*V'$.

An aggregate $p(*X_1, \ldots, *X_m)$ denotes as many copies of the m-ary atom p as the number of links denoted by the bundle $*X_i$. Each $*X_i$ must have the same origin with respect to the process context name (Consistency Condition 4); in other words, the other occurrences of the $*X_i$'s must all appear in process contexts with the same name. Occurrence Condition 4 implies that exactly one of $*X_1, \ldots, *X_m$ occurs in the LHS of a rule.

Let us give two examples. The LHS of the rule

```
kill(S), {i(S),$p[|*P]} :- killed(*P)
```

can reduce the process

```
kill(S), {i(S),a(X),b(Y,Z),c(Z,U)},
```

by letting `$p[|*P]` receive `a(X),b(Y,Z),c(Z)` , and the process becomes

```
killed(X), killed(Y).
```

In this example, the membrane is used to delimit the process structure to be controlled, and the tag `i( )` is attached to the message channel from outside. The above rule says that, when a `kill` message is sent through the channel, the target cell is deleted and each free link owned by the cell is terminated by an atom `killed`.

Next, consider the process

```
cp(S,S1,S2), {i(S),a(X),b(Y,Z),c(Z)}
```

and the rule

```
cp(S,S1,S2), {i(S),$p[|*P]} :-
    {i(S1),$p[|*P1]}, {i(S2),$p[|*P2]}, cp(*P,*P1,*P2) .
```

Then the process becomes

```
{i(S1),a(X1),b(Y1,Z1),c(Z1)}, {i(S2),a(X2),b(Y2,Z2),c(Z2)},
cp(X,X1,X2), cp(Y,Y1,Y2) .
```

In short, the `cp` message makes two copies of the target cell and connects the free links of the copied cells and the original free links using ternary `cp` atoms (Figure 4)

(R6) has the side condition that the non-rule part of the redex cannot contain connectors. Since the connectors can be migrated and absorbed using (E6), (R4) and (R5), we don't have to be able to specify connectors in the LHSs of rules. A desirable result of this side condition is that the semantics of connectors cannot be redefined.

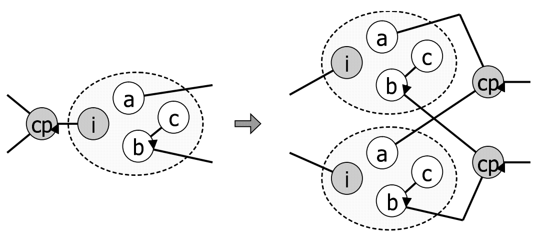


Figure 4: Cell copying using process contexts and aggregates

5 Program Examples

5.1 Concatenating Lists

The skeleton of a linear list can be represented, using element processes `c`(ons) and a terminal process `n`(il), as $\mathtt{c}(A_1, X_1, X_0)$, ..., $\mathtt{c}(A_n, X_n, X_{n-1})$, $\mathtt{n}(X_n)$. Here, A_i is the link to the ith element and X_0 is the link to the whole list (from somebody else). This corresponds to a list formed by the constraints $X_0 = \mathtt{c}(A_1, X_1)$, ..., $X_{n-1} = \mathtt{c}(A_n, X_n)$, $X_n = \mathtt{n}$ in (constraint) logic programming languages, except that the LMNtal list is a resource rather than a value. Two lists can be concatenated using the following two rules:

```
append(X0,Y,Z0), c(A,X,X0) :- c(A,Z,Z0), append(X,Y,Z)
append(X0,Y,Z0), n(X0) :- Y=Z0
```

Figure 5 shows a graphical representation of the append program and its execution.

The above program has clear correspondence with `append` in GHC:

```
append(X0,Y,Z0) :- X0=c(A,X) | Z0=c(A,Z), append(X,Y,Z).
append(X0,Y,Z0) :- X0=n | Y=Z0.
```

but LMNtal has eliminated syntactic distinction between processes and data.

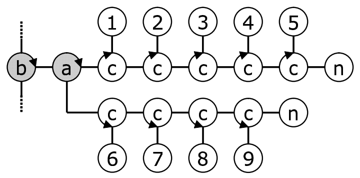
The above program resembles `append` in Interaction Nets [9]. Indeed, Lafont writes "our rules are clearly reminiscent of clauses in *logic programming*, especially in the use of variables (see the example of difference-lists), and our proposal could be related to PARLOG or GHC" [9]. LMNtal generalizes Interaction Nets by removing the restriction to binary interaction and allowing hierarchical processes.

5.2 Stream Merging

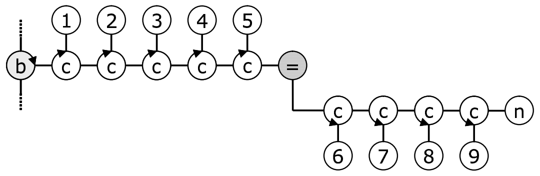
As in logic programming, streams can be represented as lists of messages, and n-to-1 communication by stream merging can be programmed as follows:

```
{i(X0),o(Y0),$p[|*Z]}, c(A,X,X0) :-
        c(A,Y,Y0), {i(X),o(Y),$p[|*Z]}
```

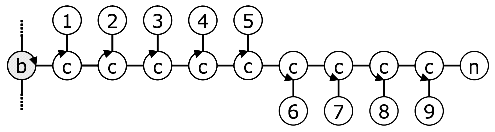
Here, the membrane `{ }` of the left-hand side records n (≥ 1) input streams with the name `i` and one output stream with the name `o`. The process context `$p[|*Z]` is to match the rest of the input streams and pass them to the RHS. Figure 6 shows a redex to which the above rewrite rule is applicable and the result of reduction.



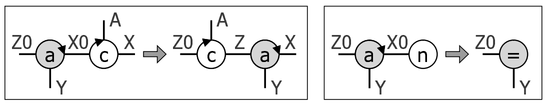
(a) Initial state



(b) Final state 1



(c) Final state 2



(d) Rewrite rules

Figure 5: List concatenation

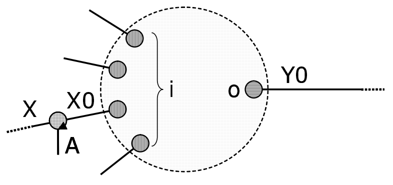
5.3 Process Migration

Consider two cells that share a communication link. Suppose they run independently using individual sets of reaction rules most of the time but sometimes migrate processes to each other through the link. The rule for migration is given in an upper layer.

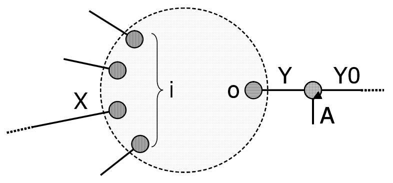
It is the rôle of the upper layer to determine the protocol of process migration, while the cells "hook" processes to be migrated on the communication link according to the protocol. Here we assume that the innermost cell containing $\mathtt{g}(S, D)$ is to be migrated by the upper layer, where S and D are the source and the destination sides of the communication link, respectively.

```
{$s[S0|*S], @s, {g(S0,D0),$m[|*M],@m}},
{$d[D0|*D], @d} :-
    {$s[S|*S], @s},
    {{s(S,D),$m[|*M],@m}, $d[D|*D], @d}
```

When `@m` is non-empty, the rule acts as active process migration; otherwise it acts as data migration. Note that the communication link between the source and the destination processes changes after migration. This is an important characteristic of



(a)



(b)

Figure 6: Multiway stream merging

logical links. The membrane delimiting migrated resources can be removed at the destination site.

5.4 Cyclic Data Structures

Most declarative languages are good at handling lists and trees but are awkward in handling cyclic data structures. This is not the case with LMNtal. In LMNtal, a bidirectional circular buffer with n elements can be represented as

$$\mathtt{b}(S, X_n, X_0), \mathtt{n}(A_1, X_0, X_1), \ldots, \mathtt{n}(A_n, X_{n-1}, X_n),$$

where `b` is a header process, the A_i's are links to the elements, and S is the link from the client process. Operations on the buffer are sent through S as messages such as `left`, `right` and `put`. The reaction rules between messages and the buffer can be defined as follows:

```
left(S,S0), n(A,L,C0), b(S0,C0,R) :- b(S,L,C), n(A,C,R)
right(S,S0), b(S0,L,C0), n(A,C0,R) :- n(A,L,C), b(S,C,R)
put(A,S,S0), b(S0,L,R) :- n(A,L,C), b(S,C,R)
...
```

Shape Types [7] are another attempt to facilitate manipulation of dynamic data structures. Interestingly, Shape Types took a dual approach, namely they used variables to represent graph nodes and names to represent links.

6 Conclusions

We have presented a concise language model LMNtal, which has logical links, multisets, nested nodes and transformation as its “big four” elements. LMNtal was inspired by communication using logical variables, and its principal goal as a concurrent language has been to unify processes, messages, and data. There are many languages and computation models that support multisets and/or graph rewriting, but LMNtal is unique in the design of link handling and its combination with hierarchical multisets.

CHR is another multiset rewriting language that features logical variables. While Flat LMNtal can be thought of as a linear fragment of CHR, LMNtal and CHR have many differences in the use of logical variables, control of reactions, intended applications, and so on. It is a challenging research topic to embed CHR into LMNtal.

We have released an initial version of prototype implementation in Java[1]. It features

- a construct for detecting inactive cells,
- built-in number types,
- the notion of type constraints for typechecking and comparison of numbers and symbols, and
- foreign language interface,

in addition to most of the constructs described in this paper.

Many things remain to be done. The most important issue in the language design is to equip it with useful type systems. We believe that many useful properties, for instance shapes formed by processes and links, the directionality of links (i.e., whether links can be implemented as unidirectional pointers), and properties about free links of cells, can be guaranteed statically using type systems. Challenging topics in our implementation project include optimization of the representation of processes and links, optimizing compilation of reaction rules, and parallel and distributed implementation. Since LMNtal is intended to unify existing computational models, relating LMNtal to them by embedding them into LMNtal is another important research subject. When the embeddings are simple enough, LMNtal would act as a common implementation language of various models of computation.

Last but not least, we should accumulate applications. Some interesting applications other than ordinary concurrent computation are graph algorithms, multi-agent systems, Web services, and programming by self-organization.

Acknowledgments

Discussions with the members of the programming language research group at Waseda helped the development of the ideas described here. This work is partially supported by Grant-In-Aid for Scientific Research ((B)(2) 16300009, Priority Areas (C)(2)13324050 and (B)(2)14085205), MEXT and JSPS.

References

[1] Andries, M. *et al.*, Graph Transformation for Specification and Programming. *Sci. Comput. Program.*, Vol. 34, No. 1 (1999), pp. 1–54.

[1] `http://www.ueda.info.waseda.ac.jp/lmntal/`

[2] Banâtre, J.-P. and Le Métayer, D., Programming by Multiset Transformation. *Commun. ACM*, Vol. 35, No. 1 (1993), pp. 98–111.

[3] Berry, G. and Boudol, G., The Chemical Abstract Machine. In *Proc. POPL'90*, ACM, pp. 81–94.

[4] Cardelli, L. and Gordon, A. D. : Mobile Ambients, in *Foundations of Software Science and Computational Structures*, Nivat, M. (ed.), LNCS 1378, Springer-Verlag, 1998, pp. 140–155.

[5] Drewes, F., Hoffmann, B. and Plump, D., Hierarchical Graph Transformation. *J. Comput. Syst. Sci.*, Vol. 64, No. 2 (2002), pp. 249–283.

[6] Engels, G. and Schürr, A., Encapsulated Hierarchical Graphs, Graph Types, and Meta Types. *Electronic Notes in Theor. Comput. Sci.*, Vol. 1 (1995), pp. 75–84.

[7] Fradet, P. and Le Métayer, D., Shape Types. In *Proc. POPL'97*, ACM, 1997, pp. 27–39.

[8] Frühwirth, T., Theory and Practice of Constraint Handling Rules. *J. Logic Programming*, Vol. 37, No. 1–3 (1998), pp. 95–138.

[9] Lafont, Y., Interaction Nets. In *Proc. POPL'90*, ACM, pp. 95–108.

[10] Miller, D. : Overview of Linear Logic Programming, to appear in *Linear Logic in Computer Science*, Ehrhard, T., Girard, J.-Y., Ruet, P. and Scott, P. (eds.), Cambridge University Press.

[11] Milner, R., Bigraphical Reactive Systems. In *Proc. CONCUR 2001*, LNCS 2154, Springer, 2001, pp. 16–35.

[12] Păun, Gh., Computing with Membranes. *J. Comput. Syst. Sci.*, Vol. 61, No. 1 (2000), pp. 108–143.

[13] Saraswat, V. A., Kahn, K. and Levy, J., Janus: A Step Towards Distributed Constraint Programming. In *Proc. 1990 North American Conf. on Logic Programming*, MIT Press, 1990, pp. 431–446.

[14] Ueda, K., Concurrent Logic/Constraint Programming: The Next 10 Years. In *The Logic Programming Paradigm: A 25-Year Perspective*, Apt, K. R., Marek, V. W., Truszczynski M., and Warren D. S. (eds.), Springer-Verlag, 1999, pp. 53–71.

[15] Ueda, K., Resource-Passing Concurrent Programming. In *Proc. TACS 2001*, LNCS 2215, Springer, 2001, pp. 95–126.

[16] Kazunori Ueda and Norio Kato, Programming with Logical Links: Design of the LMNtal Language. In *Proc. Third Asian Workshop on Programming Languages and Systems (APLAS 2002)*, 2002, pp. 115–126.

[17] Ueda, K. and Kato, N., The Language Model LMNtal. *Computer Software*, Vol. 21, No. 2 (2004), pp. 44-61 (in Japanese).