LMNtal as a Unifying Declarative Language

Kazunori Ueda[†], Norio Kato[‡], Koji Hara[†]and Ken Mizuno[‡]

[†]Dept. of Computer Science, Waseda University
[‡]Center for Verification and Semantics, National Institute of Advanced Industrial Science and Technology (AIST)
{ueda,n-kato,hara,mizuno}@ueda.info.waseda.ac.jp

Abstract. *LMNtal* (pronounced "*elemental*") is a simple language model based on hierarchical graph rewriting that uses logical variables to represent connectivity and membranes to represent hierarchy. LMNtal is an outcome of the attempt to unify constraint-based concurrency and Constraint Handling Rules (CHR), the two notable extensions to concurrent logic programming. LMNtal is intended to be a substrate language of various computational models, especially those addressing concurrency, mobility and multiset rewriting. Another important goal of LMNtal has been to put hierarchical graph rewriting into practice and demonstrate its versatility by designing and implementing a full-fledged, monolithic programming language. In this paper, we demonstrate the practical aspects of LMNtal using a number of examples taken from diverse areas of computer science. Also, we discuss the relationship between LMNtal and CHR, which exhibit both commonalities and differences in various respects.

1 Introduction

The development of the LMNtal language model has been 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 practical programming language that covers a variety of computational platforms which are now developing towards both wide-area computation and nanoscale computation.

LMNtal is an outcome of the attempt to unify constraint-based concurrency (also known as concurrent constraint programming) [10] and (some ideas from) Constraint Handling Rules (CHR) [3], the two notable extensions to concurrent logic programming [9]. LMNtal can be viewed also as a multiset rewriting language equipped with links, where multisets are supported by the membrane construct that allows both nesting and mobility and links are represented by logical variables that essentially work as linear local names (i.e., local names occurring twice).

Despite its versatility, LMNtal is a surprisingly simple language and one can start using it with almost no background about programming or logic or advanced mathematics. This is thanks to its close connection to diagrammatic representation of computational entities and the choice of the class of diagrams and reduction mechanisms to work with.

Since LMNtal was first designed in 2002 [11], it underwent the design review process from both theoretical and practical viewpoints. From a theoretical point of view, the major challenge has been to design the operational semantics in such a way that the interplay between graph structures formed by links and hierarchical structures formed by membranes (that may be crossed by links) is properly handled, which turned out to be quite subtle. From a practical point of view, the major challenge has been to build a full-fledged implementation of the newly designed language to provide designers with a constructive understanding of the language, to point out oversights in language design, to distinguish kernel constructs that require hard-wired support from those that can be implemented on top of the kernel, to accumulate programming experiences, and to identify language features not essential in theory but important in practice.

The LMNtal system, running on a Java platform and now available on the web[1], is the third of our attempts to implement the language. The purpose of this paper is to describe the features of LMNtal as a simple and versatile declarative language by means of various examples. The readers are referred to [12] on LMNtal as a computational model. All the examples in this paper have been tested on our LMNtal implementation.

2 Introductory Examples

2.1 Hierarchical Multiset Rewriting

The first series of examples is to demonstrate hierarchical multiset rewriting in LMNtal. Here we use LMNtal in an interactive mode.

```
$ lmntal
      LMNtal version 0.80.20060319
Type :h to see help.
Type :q to quit.

# 1,1,1, {1,1,1,1,1, {1,1,1}, (1,1:-2)}

1, 1, 1, {2, 2, 1, {1, 1, 1}, @601}

# {out,a,b,c}, d, {e,f}, ({out,$p[]} :- $p[]).

d, a, c, b, {f, e}, @603
```

Symbols starting with lowercase letters and numbers represent *atoms*, those starting with uppercase letters (not appearing yet) represent *links*, and braces

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

represent *membranes*. Symbols starting with the dollar sign represent *process contexts*, and those starting with the at sign represent *rulesets*, which are (possibly compiled) sets of *rewrite rules*.

The first example is simple multiset rewriting (from two `1`'s to `2`), except that membranes are used to form hierarchical multisets. Rules local to membranes act only on those atoms in the same place in the membrane hierarchy. The symbol `@601` indicates a compiled ruleset obtained from the rule `(1,1:-2)`.

The second example shows that a rule can handle cells (i.e., atoms and cells enclosed by membranes) and change the membrane structure. The process context `$p[]` represents a local context of the membrane it belongs to and works as a wildcard. Thus the rule `({out,$p[]} :- $p[])` can be read as "remove the membrane containing the atom `out` and export its content." Notice that the order of atoms and cells are irrelevant because they form multisets.

2.2 List Processing

Next, we describe the use of links using list processing as an example.

The skeleton of a list can be represented, using `'.'` atoms (cons) and a `'[]'` atom (nil), as $\texttt{'.'}(A_1, X_1, X_0), \ldots, \texttt{'.'}(A_n, X_n, X_{n-1}), \texttt{'[]'}(X_n)$. Here, A_i is the link to the ith element and X_0 is the link to the whole list (from somebody else owning the list). This corresponds to a list formed by the constraints $X_0 = \texttt{[}A_1 \,|\, X_1\texttt{]}$, $\ldots$, $X_{n-1} = \texttt{[}A_n \,|\, X_n\texttt{]}$, $X_n = \texttt{[]}$ in (constraint) logic programming languages, with the notable exception that an LMNtal list is a *resource* (i.e., entity with ownership) rather than a value. All links are point-to-point; link names occurring twice in an expression represent *local links* and those occurring once represent *free links* which are supposed to be connected to an atom outside the expression.

LMNtal links are non-directional like chemical bonds. The directionality of a list is determined by which arguments of atoms are connected together.

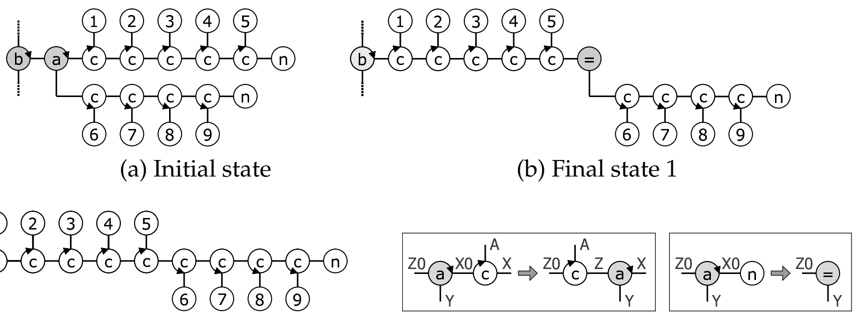
Two lists can be concatenated using the following two rules:

```
append(X0,Y,Z0), '[]'(X0) :- Y=Z0.
append(X0,Y,Z0), '.'(A,X,X0) :- '.'(A,Z,Z0), append(X,Y,Z).
```

(As above, each rule can be written in the period-terminated form as well as in the comma-separated form.) Figure 1 shows a graphical representation of the `append` program and its execution (`c`(ons) for `'.'` and `n`(il) for `'[]'`), where `b` is the consumer of the result of `append` and the atom '=' autonomously disappears after connecting its arguments together.

LMNtal doesn't distinguish between predicate symbols (procedure names) and function symbols (constructors); they are equal in status, though a type system could be employed to distinguish between them.

LMNtal provides a simple, systematic and powerful *term abbreviation scheme*. We can exploit the fact that a local link name has exactly two occurrences and abbreviate $p(s_1, \ldots, s_m), q(\ldots, s_m, \ldots)$ to $q(\ldots, p(s_1, \ldots, s_{m-1}), \ldots)$. For instance, the list shown in the beginning of this section, with $n = 3$, can be abbreviated to $\texttt{'.'(}A_1\texttt{,'.'(}A_2\texttt{,'.'(}A_3\texttt{,[]))},X_0\texttt{)}$. This is structurally equivalent to $X_0 = Y$,



(a) Initial state (b) Final state 1

(c) Final state 2 (d) Rewrite rules

Fig. 1. List concatenation

`'.'`$(A_1$`,'.'`$(A_2$`,'.'`$(A_3$`, []))`$,Y)$ (see Section 3.2), and by eliminating Y using the scheme again we obtain $X_0 =$ `'.'`$(A_1$`,'.'`$(A_2$`,'.'`$(A_3$`, [])))` or $X_0 =$ `[`A_1, A_2, A_3`]`, following the Prolog convention.

Likewise, we write $p(\ldots, \{P\}, \ldots)$ to mean $p(\ldots, A, \ldots), \{+A, P\}$, where P is a process (Section 3.1) and A is a link. The unary atom '+' is used as the standard atom to terminate incoming links to a cell.

A notable consequence of the above scheme is that `f(5)`, `5(f)`, `f=5`, `5=f`, and `(5(X),f(X))` represent exactly the same thing, namely the unordered pair (or the diatomic molecule) of a unary `f` and a unary `5`.

Using abbreviation, the list concatenation program can be written in a (concurrent) logic programming form

```
append([],Y,Z) :- Y=Z.
append([A|X],Y,Z0) :- Z0=[A|Z], append(X,Y,Z)
```

and in the term rewriting form

```
Z= append([],Y) :- Z= Y.
Z= append([A|X],Y) :- Z= [A|append(X,Y)].
```

The above program resembles `append` in Interaction Nets [4]. Indeed, LMNtal generalizes Interaction Nets by removing the restriction to binary interaction and allowing hierarchical processes.

2.3 Numbers and Arithmetics

LMNtal supports numbers and arithmetics. However, for uniformity LMNtal treats every atom as resource and numbers are not exceptions. Accordingly, all numbers (integers and floats) are unary atoms and are connected to the "owner" of the number (which can be an arithmetic operator). LMNtal still allows 0-ary `8` and `3.14` as atoms but they are not supported by the arithmetic features of the language. In our LMNtal system, a ground expression autonomously evaluates to a number using a *system ruleset* implicitly built into every cell. For instance, the cell

$$\begin{array}{rll}
\text{(Process)} & P ::= \mathbf{0} \;\mid\; p(X_1,\ldots,X_m) \;\mid\; P,P \;\mid\; \{P\} \;\mid\; T\,\texttt{:-}\,T \\
\text{(Process template)} & T ::= \mathbf{0} \;\mid\; p(X_1,\ldots,X_m) \;\mid\; T,T \;\mid\; \{T\} \;\mid\; T\,\texttt{:-}\,T \\
 & \quad\;\mid\; \texttt{@}p \;\mid\; \texttt{\$}p\texttt{[}X_1,\ldots,X_m\,\texttt{|}\,A\texttt{]} \;\mid\; p(\texttt{*}X_1,\ldots,\texttt{*}X_n) \\
\text{(Residual)} & A ::= \texttt{[]} \;\mid\; \texttt{*}X
\end{array}$$

Fig. 2. Syntax of LMNtal

```
{n(1), n(2), n(3), n(4), n(5), (n(A),n(B):-n(A*B))}
```

evaluates nondeterministically to `{n(120), (n(A),n(B):-n(A*B))}`. LMNtal and our LMNtal implementation do not specify the strategy of multiset matching, but our implementation has a *shuffle* mode that randomizes the selection of atoms to see if the program may yield different results. For rules belonging to the same membrane, the textual order determines their natural priorities that can also be randomized by a runtime option.

3 The Core Language

We briefly describe the syntax and the semantics of LMNtal. For details omitted from here, the readers are referred to [12].

3.1 Syntax

The syntax of LMNtal is given in Figure 2, where two syntactic categories, *link names* (denoted by X) and *atom names* (denoted by p), are presupposed. The atom name = is reserved for atomic processes for link connection. The subject entities of LMNtal are called *processes*.

Intuitively, **0** is an inert process; $p(X_1,\ldots,X_m)$ $(m \geq 0)$ is an *atom* with m links; P,P is parallel composition called a *molecule*; $\{P\}$, a *cell*, is a process grouped by the membrane $\{\ \}$; and $T\,\texttt{:-}\,T$ is a rewrite rule for processes.

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

A process P must observe the following *link condition*: Each link name in P may occur *at most twice*. Furthermore, each link name of a rule must occur exactly twice. As usual, α-conversion can be used to avoid clashes of link names.

A *rule context*, $\texttt{@}p$, matches a (possibly empty) multiset of rules inside a membrane, while a *process context*, $\texttt{\$}p\texttt{[}X_1,\ldots,X_m\,\texttt{|}\,A\texttt{]}$ $(m \geq 0)$, matches processes other than rules inside a membrane. The argument of a process context specifies what links may or must occur free. When the residual A is `[]`, the argument is abbreviated to $\texttt{[}X_1,\ldots,X_m\texttt{]}$ and means that the set of free links of $\texttt{\$}p$ must be exactly $\{X_1,\ldots,X_m\}$. When A is of the form $*X$ (called a *bundle*), it represents zero or more free links of the context that may occur in addition to the "must-occur" links $X_1,\ldots,X_m$.

$$\text{(E1)}\quad \mathbf{0}, P \equiv P \qquad \text{(E2)}\quad P, Q \equiv Q, P \qquad \text{(E3)}\quad P, (Q, R) \equiv (P, Q), R$$

$$\text{(E4)}\quad P \equiv P[Y/X] \qquad \text{if } X \text{ is a local link of } P$$

$$\text{(E5)}\quad P \equiv P' \;\Rightarrow\; P, Q \equiv P', Q \qquad \text{(E6)}\quad P \equiv P' \;\Rightarrow\; \{P\} \equiv \{P'\}$$

$$\text{(E7)}\quad X = X \equiv \mathbf{0} \qquad \text{(E8)}\quad X = Y \equiv Y = X$$

$$\text{(E9)}\quad X = Y,\ P \equiv P[Y/X] \qquad \text{if } P \text{ is an atom and } X \text{ occurs free in } P$$

$$\text{(E10)}\quad \{X = Y,\ P\} \equiv X = Y,\ \{P\} \qquad \text{if exactly one of } X \text{ and } Y \text{ occurs free in } P$$

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

$$\text{(R4)}\quad \{X = Y, P\} \longrightarrow X = Y,\ \{P\} \qquad \text{if } X \text{ and } Y \text{ occur free in } \{X = Y, P\}$$

$$\text{(R5)}\quad X = Y,\ \{P\} \longrightarrow \{X = Y, P\} \qquad \text{if } X \text{ and } Y \text{ occur free in } P$$

$$\text{(R6)}\quad T\theta, (T \text{ :- } U) \longrightarrow U\theta, (T \text{ :- } U)$$

Fig. 3. Structural Congruence and Reduction Relation of LMNtal

The following examples will illustrate the role of process contexts. The LHS `{p(X)}` matches a cell exactly consisting of `p(X)` (`X` free), while the LHS `{p(X), $q[|*Y]}` matches a cell *containing* `p(X)` (`X` free) and the LHS `{p(X), $q[X|*Y]}` matches a cell *containing* `p(X)` (`X` local). The context receives the rest of the processes inside the membrane, and the bundle `*Y` is bound to the sequence of free links (order determined by the system) of the whole cell.

The final form, $p(*X_1, \ldots, *X_n)$ $(n > 0)$, represents an *aggregate* of n-ary atoms, which are to be connected to may-occur free links of process contexts.

Rewrite rules must observe syntactic conditions on possible occurrences of rules (inside the rules) and contexts, as well as on link names [12], all of which are for the use of contexts and aggregates to make sense.

3.2 Operational Semantics

The operational semantics of LMNtal (Figure 3) consists of two parts, namely structural congruence (E1)–(E10) and the reduction relation (R1)–(R6). Here, $[Y/X]$ is a *link substitution* that replaces X with Y.

(E1)–(E3) are the characterization of molecules as multisets. (E4) represents α-conversion of local link names. (E5)–(E6) are structural rules that make $\equiv$ a congruence. (E7) says that a self-absorbed loop is equivalent to $\mathbf{0}$, while (E8) expresses the symmetry of connectors. (E9)–(E10) are absorption/emission rules of connectors for atoms and cells, respectively.

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

(R1)–(R3) are standard structural rules. (R1) says that reductions can proceed concurrently based on local reducibility conditions. Fine-grained concurrency of LMNtal originates from this rule. (R2) says that computation inside a membrane can proceed independently of the exterior of the membrane. For a cell to evolve autonomously, it must contain its own set of rules. Computation of a cell containing no rules are to be controlled by rules outside the cell. (R3) incorporates structural congruence into the reduction relation.

(R4)–(R5) are the mobility rules of connectors across membranes. The central rule of LMNtal is (R6). The substitution θ is a mapping from process templates to processes and represents what process (or multiset of rules) has been received by each process context (or rule context), respectively, and what multiset of atoms each aggregate represents [12]. The simplest way of viewing rules with process/rule contexts and aggregates is to view them as *rule schemes* that represent sets of rules with no contexts or aggregates.

We can think of a subset of LMNtal, *Flat LMNtal*, that does not allow cell hierarchies (and accordingly, process contexts, rule contexts and aggregates). In Flat LMNtal, θ becomes unnecessary and (R6) is simplified to

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

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 $U,(T \mathrel{:\!-} U)$ and its surrounding context, should observe the link condition, but this can always be achieved by α-converting $T \mathrel{:\!-} U$ before use so that the local link names in U won't cause name crashes with the context.

3.3 Extension: Guard and Typed Process Contexts

Our LMNtal system extends the above core language by featuring the notion of *guards* and *typed process contexts*. Guards are to express conditional rewrite rules, but the question is what should constitute conditions in our setting of graph rewriting. Suppose we have a multiset of integers expressed as

```
n(3), n(-5), n(101), n(72), n(18), n(47), n(11).
```

and want to find the maximum value of them by deleting others. One can achieve this using a one-liner

```
n($i), n($j) :- $i =< $j | n($j).
```

which is an abbreviation of

```
n(I), $i[I], n(J), $j[J] :- $i =< $j | n(K), $j[K].
```

Here, the process contexts don't appear in membranes that would delimit the contexts, but instead appear as operands of the guard test `$i=<$j` which constrains its arguments to unary atoms. The guard test checks if the two unary atoms are both integers and the second one is not smaller.

Other guard tests available include `int($i)` to ensure that `$i` is an integer, `unary($u)` to ensure that `$u` is a unary atom, and `ground($g)` to capture a minimal (and hence connected) graph structure with exactly one free link. Note that '`=<`', `unary`, and `ground` can all be regarded as *type constraints*; `int` is a subtype of `unary` which in turn is a subtype of `ground`, and '`=<`' ia s subtype of the product `int` × `int`.

Graph structures received by typed process contexts and checked by guard tests can be copied or discarded freely. Thanks to this mechanism, many of concurrent logic programs ever written run as LMNtal programs with very minor modifications, where the *ask* operation (the synchronization primitive) is naturally replaced by graph matching.

LMNtal features another important guard test, `uniq`$(X_1, \ldots, X_n)$. This test succeeds if (i) each X_i is connected to a (connected) graph with no free links other than X_i, and (ii) the rule has not been applied to the same tuple of graphs before. As a special case of $n = 0$, `uniq` succeeds if the rule in question has not been used before. The `uniq()` test, which was inspired by CHR's propagation rules, is a general tool for avoiding infinite application of rules whose RHS is a super(multi)set of the LHS.

4 Ideas Behind the Language Design

4.1 Basic Ingredients

The "four elements" of LMNtal are *logical links, multisets (or membranes), nested nodes, and transformation*.

Links, which are represented by linear local names, are used to represent both one-to-one communication channels between logically neighboring processes and logical neighborhood relations between data cells. LMNtal viewed as a process calculus is different from many other process calculi (or more precisely, it is a special case of them) in that a message sent through a link changes the identity of the link and that links are always private in the sense that the third party cannot access them. The conception of logical links originate from logical variables in logic programming, but again they are the special case of logical variables in that LMNtal has no notion of *instantiating* a link variable to a value; it just retains the notion of *fusing* two variables.

The notion of multisets can be found in diverse computational models including classical Petri Nets and Production Systems. However, not many of them feature multisets as first-class citizens, and the essence of membranes is to give hierarchical structures to the systems of multisets. Models and languages featuring membranes include the Chemical Abstract Machine [1], mobile ambients [2], P-systems [6], bigraphical reactive systems [5], LMNtal, and the Kell Calculus [7].

4.2 Uses of Membranes: The π-calculus Example

Membranes play many roles in LMNtal programming. First, they are used to represent records or feature structures. Second, they can encapsulate rules and delimit their scope of effect. Third, they can protect processes from the rewrite rules that would otherwise act on the processes.

The following example that encodes the communication mechanism of the asynchronous π-calculus illustrates the first and the third uses of membranes. The rules are prefixed by rule names here.

```
snd@@ snd({$y[|*V]},X) :- {$y[|*V], m(X)}.
get@@ get({m(X),$y[|*V]},Z), {$body[Z|*V]} :- {$y[|*V]}, $body[X|*V].
cp@@ {name(N),$p[N|*Y],+Z}, Z=cp(Z0,Z1) :- {name(N),$p[N|*Y],+Z0,+Z1}.
rm@@ {name(N),$p[N|*Y],+Z}, Z=rm :- {name(N),$p[N|*Y]}.
```

Here, a π-calculus name is represented by a cell containing the `name()` attribute and referenced by incident links (each marked by the '+' atom). The cell also works as a message buffer where the links marked by the `m` atom are connected to outstanding messages. The first rule gives the semantics of sending x to y, while the second rule gives the semantics of receiving a message x from y and substitute it for the formal name z. The last two rules, `cp` for copy and `rm` for remove, are used when a formal name is used more than once or not used at all. For instance, a π process $(a(z).b(y).\overline{z}\langle y\rangle) \mid \overline{a}\langle c\rangle \mid \overline{b}\langle d\rangle$ is encoded as

```
(get(A0,Z), {get(B0,Y), {snd(Z,Y)}}).
snd(A1,C).
snd(B1,D).
{name(a),+A0,+A1}, {name(b),+B0,+B1}, {name(c),+C}, {name(d),+D}.
```

which reduces to

```
{name(a)}, {name(b)}, {name(c),m(_78)}, {name(d),'+'(_78)}, @601
```

meaning that the channel `c` contains an outstanding message `d`, while `a` and `b` do not hold any messages nor are referenced any more. Note the use of membranes in the encoding of $(a(z).b(y).\overline{z}\langle y\rangle)$; prefixed processes are protected by membranes until the `get` rule removes them.

4.3 Logical Interpretation

LMNtal processes are designed to allow diagrammatic representation, but at the same time processes and rules allow logical interpretation. We focus on Flat LMNtal (LMNtal without membranes) due to space limitation.

A (Flat) LMNtal process is a conjunction of atoms where local links are interpreted as existentially quantified variables. First-order logic with equality (to handle connectors) works as the underlying logic in many cases. For programs dealing with multisets (i.e., conjunction of two or more identical formulae), however, we should drop weakening and contraction from the logic, which results in a (tiny) fragment of linear logic. Programs with don't-care nondeterminism are another example in which linear logic interpretation would be more appropriate. In the following, we focus on the classical case.

A rule T :- U is interpreted as $\forall(\exists T \Leftrightarrow \exists U)$, where T and U are conjunctions of atoms, the two $\exists$'s quantify the *local* links of T and U, respectively, and the leftmost $\forall$ quantifies the *free* links of T and U (each occurring once in T and once in U). Thus a LMNtal rule is not a clause in the classical sense, but it is easy to see that the rewrite rule (R6′), $\exists T \wedge \forall(\exists T \Leftrightarrow \exists U) \longrightarrow \exists U \wedge \forall(\exists T \Leftrightarrow \exists U)$, is a sound reasoning. (Note that the quantifier of the redex $\exists T$ works on local links of T only; free links in T are left free.) For instance, under the append program, the molecule append([a,b],[c],X),answer(X) reduces to answer([a,b,c]), which is a sound deduction.

Note the symmetry of our logical reading of rules. This means that the initial state append([a,b],[c],X),answer(X) can be deduced from answer([a,b,c]) as well. Computationally, the base case of append, if reversed, would cause divergence because it would match any link in the current configuration. However, there is a class of reversible programs whose initial states can be restored from the final states by the backward application of LMNtal rules, an interesting topic of future study.

Our logical interpretation is to be contrasted with that of CHR where its simplification rule T<=>U is interpreted as $\forall(T \Leftrightarrow \exists U)$ [3]. The difference comes from the purpose of the languages (in general) and the roles of the variables (technically) that will be discussed below.

4.4 Relation to CHR

LMNtal rules without membranes resemble simplification rules of CHR. Indeed, Flat LMNtal could be thought of as a linear fragment of CHR. However, we find LMNtal and CHR quite different and rather complementary. First, CHR was developed for glass-box constraint programming, while LMNtal was developed as a declarative concurrent language with powerful data structures (hierarchical graphs that subsume first-class multisets). Second, CHR is to be used with some platform language (such as Prolog and Java) while LMNtal was developed as a monolithic, stand-alone language. Third, LMNtal comes with membranes that can be used not only as data structures but also as control structures, which are essential for a stand-alone language. Fourth, CHR resembles Prolog in the use of constructors and logical variables, while LMNtal is constructor-free and restricts variables to linear ones representing connectivity. Fifth, computation in LMNtal is not necessarily intended to be confluent because its intended applications include concurrent programming.

Nevertheless, a logic variable (possibly with attributes as in some Prolog implementations) can be encoded using membranes. Furthermore, the uniq guard test (Section 3.3) has been used successfully to encode propagation rules of CHR.

5 Overview of the Implementation

The current version of our LMNtal system consists of 43,000 lines of Java code including programming environments. Both the lmntal and lmnc commands invoke the compiler to generate dedicated intermediate code. The lmntal command executes it interpretively, while lmnc translates it further into Java code packed into a Java archive file, which can be executed by the lmnr command. Several subtleties, mostly resulting from the hierarchical nature of the language, have been identified and resolved in the course of design reviews.

LMNtal is a fine-grained concurrent language, but how to implement it correctly and efficiently is far from obvious for the following two reasons. First, it features both connectivity (links) and hierarchy (membranes) in such a way that links may connect remote atoms across one or more membranes. Second, different rulesets belonging to different membranes may attempt to rewrite the same process competitively; for instance, a process located at some place in the membrane hierarchy may be manipulated by both local and non-local rules. In addition, since LMNtal features interface to Java and Java allows users to create threads—explicitly or implicitly by using GUI's—, being able to control asynchronous execution is an important requisite. The major challenge at this stage of language development has been to establish a correct implementation scheme amenable to asynchronous rewriting by multiple tasks (though we have already implemented several, mostly intra-rule, optimization techniques including dependency-directed backtracking of multiset matching, reuse of atoms and links upon reduction, etc.).

One of our decisions was to let a link cross a membrane via two "immigration" proxies, one inside the membrane and the other outside. For instance, the standard internal representation of the configuration `{a(X)},b(X)` is

```
{a(X0), $in(X1,X0)}, $out(X1,X2), b(X2)
```

as one can see by raising the verbosity level of the system. Semantically, these proxies are just connectors (=) sticking to the membrane.

These proxies enable concurrent rewriting of subgraphs belonging to different places of the membrane structure. Having proxies both inside and outside the membrane also eases the implementation of (R4) and (R5) by letting them react autonomously when two `$in`'s or two `$out`'s are tied together.

6 More Examples

The programs we have successfully expressed using LMNtal are quite diverse, including the pure and call-by-name lambda calculi, the synchronous and the asynchronous π-calculus (see Section 4.1 for the asynchronous case), the ambient calculus, bigraphs and their composition [5], bottom-up parsers and unparsers, calculators with GUI, fullerenes (as examples of highly-connected graph structures), to name a few.

6.1 The Lambda Calculus

The lambda calculus based on graph reduction can be elegantly encoded in LMNtal. Here we give a nondeterministic version (Fig. 4).

This is a simplified version of the encoding into Interaction Nets by Sinot [8]. The first rule captures the essence of the lambda calculus, β-reduction, while all the other rules are to handle nonlinear (i.e., $\neq 1$) use of variables by copying or removing graph structures. The key idea of [8] was to use two different atoms, `cp` and `dp`, to control graph copying.

For instance, the Church numeral 2 ($\lambda fx.f(fx)$) is encoded as

```
beta@@ H=apply(lambda(A, B), C) :- H=B, A=C.

l_c@@ lambda(A,B)=cp(C,D) :- C=lambda(E,F), D=lambda(G,H), A=dp(E,G), B=dp(F,H).
a_c@@  apply(A,B)=cp(C,D) :- C= apply(E,F), D= apply(G,H), A=cp(E,G), B=cp(F,H).
l_d@@ lambda(A,B)=dp(C,D) :- C=lambda(E,F), D=lambda(G,H), A=dp(E,G), B=dp(F,H).
a_d@@  apply(A,B)=dp(C,D) :- C= apply(E,F), D= apply(G,H), A=dp(E,G), B=dp(F,H).
l_r@@ lambda(A,B)=rm :- A=rm, B=rm.
a_r@@  apply(A,B)=rm :- A=rm, B=rm.
c_r@@     cp(A,B)=rm :- A=rm, B=rm.
d_r@@     dp(A,B)=rm :- A=rm, B=rm.
r_r@@          rm=rm :- .
d_d@@ dp(A,B)=dp(C,D) :- A=C, B=D.
c_d@@ cp(A,B)=dp(C,D) :- C=cp(E,F), D=cp(G,H), A=dp(E,G), B=dp(F,H).
u_c@@ U=cp(A,B) :- unary(U) | A=U, B=U.
u_d@@ U=dp(A,B) :- unary(U) | A=U, B=U.
u_r@@ U=rm      :- unary(U) | .
```

Fig. 4. The lambda calculus, nondeterministic version

```
  lambda(cp(F0,F1),lambda(X,apply(F0,apply(F1,X))),Result).
```

and

```
  N=n(2) :- N=lambda(cp(F0,F1), lambda(X, apply(F0,apply(F1,X)))).
  N=n(3) :- N=lambda(cp(F0,cp(F1,F2)), lambda(X,
                  apply(F0,apply(F1,apply(F2,X))))).
  res=apply(apply(apply(n(2), n(3)), s), 0).
  H=apply(s, I) :- int(I) | H=I+1.
```

evaluates to a molecule `res=9` (plus the initial rules), where the readers are reminded that exponentiation of Church numerals is encoded as $\lambda mn.nm$. As illustrated above, the encoding in LMNtal allows some free names (`s` in this example) to be "evaluated" by rules given by the user (δ-reduction).

We have also encoded the call-by-name lambda calculus, and ran recursive functions expressed using the fixpoint (**Y**) combinator.

6.2 Highly Connected Graph Structures

In most declarative languages (except logic programming languages featuring unification over rational terms), cyclic or highly connected data structures are harder to manipulate than lists and trees. To demonstrate that this is not the case with LMNtal, we give a simple program to build a fullerene (C_{60}) structure consisting of only two rules and two initial atoms:

```
  dome(L0,L1,L2,L3,L4,L5,L6,L7,L8,L9) :-
      p(T0,T1,T2,T3,T4), p(L0,L1,H0,T0,H4), p(L2,L3,H1,T1,H0),
      p(L4,L5,H2,T2,H1), p(L6,L7,H3,T3,H2), p(L8,L9,H4,T4,H3).
  dome(E0,E1,E2,E3,E4,E5,E6,E7,E8,E9).  /* top half */
```

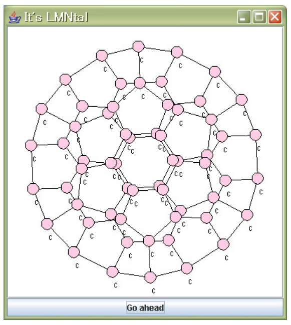


Fig. 5. The C_{60} structure

```
{ module(io).
  ...
  io.input(Message, X) :- [:/*inline*/
    String s = javax.swing.JOptionPane.showInputDialog(null, me.nth(0));
    me.setName(``done'');
    me.nthAtom(0).setName(s);
    :](Message, X).
  ...
}
```

Fig. 6. The `io` module

```
dome(E0,E9,E8,E7,E6,E5,E4,E3,E2,E1).  /* bottom half */

p(L0,L1,L2,L3,L4) :- X=c(L0,c(L1,c(L2,c(L3,c(L4,X))))).
```

The first rule and the next two initial atoms build a icosahedron (polyhedron with 20 triangles) by sewing up two five-triangle domes, and the final rule turns it into a fullerene structure. Figure 5 shows the graph structure rendered using the visualization option (-g) of the LMNtal system.

6.3 Modules and Foreign-Language Interface

The system comes with modules and foreign-language interface, both extremely important for developing applications. For instance, the `io` module, coming as a standard library, starts with the definition of Fig. 6.

A module is just a membrane containing the module name declaration and a set of rules. When a(n extended) name of the form *modulename.atomname* is mentioned in some membrane, the ruleset belonging to the module is implicitly imported to that membrane.

This example also shows the use of Java code in LMNtal. The Java code appears as a special atom, quoted by `[:` and `:]` and starting with `/*inline*/`, which is called an *inline execution atom*. The code is expanded in the translated

Java code and executed in the final phase of rule application, so that the code can access graph structures built by the RHS of the rule. The special variable `me` refers to the atom (which is the Java code being executed) and `mem` (not used in the example) refers to the membrane it belongs to. The *i*th argument given to the inline code can be accessed as `me.nth(`*i*`)`. Using these mechanisms, one can manipulate LMNtal's graph structures from within the inline atom. For instance, executing `result=io.input("Hello")` will show a pop-up window saying `"Hello"`, and typing in `"World!"` into the text field will cause the the whole molecule to evolve into `result=done("World!")`.

One can also define Java classes used by inline code using an *inline declaration atom*, a quoted atom starting with `/*inline_define*/`. The foreign-language interface greatly facilitated the development of the LMNtal system because one could easily provide and test new features (arrays, sockets, window toolkits, etc.) before or without hard-coding them into the runtime system.

7 Concluding Remarks

We have presented a concise declarative language LMNtal with diverse program examples. The main contribution of the work is that we have succeeded in putting hierarchical graph rewriting into practice and demonstrated its versatility. We have also shown the logical interpretation of LMNtal computation. LMNtal inherits some of the ideas from constraint-based concurrency and CHR, but cell hierarchy that allows trans-membrane links and local rulesets required us to develop a new implementation technique almost from scratch.

CHR is another multiset rewriting language that features logical variables. While Flat LMNtal could be thought of as a linear fragment of CHR, they exhibit differences as well as commonalities in various respects including the use of logical variables, the control structure, and principal applications. Still, we believe that their commonalities call for the cross-fertilization of the ideas, results and experiences we have accumulated.

Parallel and distributed implementations of LMNtal are both underway, building upon the asynchronous execution scheme we have developed.

LMNtal opens up many interesting research issues. One of the most important issues in language design and implementation is to equip it with useful type systems. We believe that many useful properties, for instance shapes formed by processes and the directionality of links (i.e., whether links can be implemented as one-way pointers), can be guaranteed statically using type systems and enable aggressive compiler optimization. Another important topic is to design and implement appropriate constructs for (don't-know) nondeterministic computation. Although LMNtal started as a concurrent language, it is now addressing diverse applications including those involving (don't-know) nondeterminism (e.g., verification), and backtracking or exhaustive search for possible reduction paths is becoming much more important than we had expected. We have finished a prototype implementation of backtracking and exhaustive search for Flat LMNtal. Extending it to deal with hierarchical graphs is far from obvious, and is a challenging topic of future research.

Acknowledgments

The authors are indebted to the past and current members of the LMNtal project, particularly Takahiko Nagata, Shingo Yajima, Shintaro Kudo and Ken Sakurai for the development of the ideas described here and the code they contributed. 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. Berry, G. and Boudol, G., The Chemical Abstract Machine. In *Proc. POPL'90*, ACM, pp. 81–94.
2. Cardelli, L. and Gordon, A. D. : Mobile Ambients, in *Foundations of Software Science and Computational Structures*, Nivat, M. (ed.), LNCS 1378, Springer, 1998, pp. 140–155.
3. Frühwirth, T., Theory and Practice of Constraint Handling Rules. *J. Logic Programming*, Vol. 37, No. 1–3 (1998), pp. 95–138.
4. Lafont, Y., Interaction Nets. In *Proc. POPL'90*, ACM, pp. 95–108.
5. Milner, R., Bigraphical Reactive Systems. In *Proc. CONCUR 2001*, LNCS 2154, Springer, 2001, pp. 16–35.
6. Păun, Gh., Computing with Membranes. *J. Comput. Syst. Sci.*, Vol. 61, No. 1 (2000), pp. 108–143.
7. Schmitt, A. and Stefani, J.-B., The Kell Calculus: A Family of Higher-Order Distributed Process Calculi. In *Proc. Int. Workshop on Global Computing.*, LNCS 3267, Springer, 2005, pp. 146–178.
8. Sinot, F.-R., Call-by-Name and Call-by-Value as Token-Passing Interaction Nets. In *Proc. TLCA 2005*, LNCS 3461, Springer, 2005, pp. 386–400,
9. 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.
10. Ueda, K., Resource-Passing Concurrent Programming. In *Proc. TACS 2001*, LNCS 2215, Springer, 2001, pp. 95–126.
11. Ueda, K. and Kato, N., 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.
12. Ueda, K. and Kato, N., LMNtal: a language model with links and membranes. In *Proc. Fifth Int. Workshop on Membrane Computing (WMC 2004)*, LNCS 3365, Springer, 2005, pp. 110–125.