Programming with Logical Links: Design of the LMNtal language* (Extended Abstract)

Kazunori Ueda[†‡] and Norio Kato[†]

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

Abstract. We propose *LMNtal*, a simple language model based on the rewriting of hierarchical graphs that use logical variables to represent links. The two major goals of the model 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.

* This is a revised version of the paper presented at the Third Asian Workshop on Programming Languages and Systems (APLAS 2002), Shanghai, November 2002, pp. 115–126.

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

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) [12] 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 [13]. A lot of implementation efforts and techniques have been accumulated. Concurrent logic programming was generalized to concurrent constraint programming that allowed data domains other than finite trees, and a concurrent constraint language Janus [11] chose multisets (also called bags) as an important data domain. Another important generalization was Constraint Handling Rules (CHR) [7] 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.

2 Overview of LMNtal and Related Work

The "four elements" of LMNtal are *logical links, multisets, nested nodes, and transformation* — hence the name LMNtal.

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 [13]). 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 find 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 make use 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 to be a node. Hierarchical multisets can be found in the ambient calculus and the bigraphical model [9], as well as in the fields of formal languages [10] and knowledge representation [5]. The major difference from those models is in how the structuring and the nesting of nodes are combined.
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 hierarchy can act only on processes at that hierarchy).
3. **Transformation**. LMNtal has a rewrite-rule-based syntax. There has been a lot of work on graph grammars and graph rewriting/transformation [1], including hierarchical graph transformation [4], but LMNtal's emphasis is on its design from the programming language point of view. The key design issue has been how to deal with free links properly.
Rewrite rules specify reaction between elements of a multiset, but reaction between interlinked elements can in general 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 migration of nested computation.

3 Syntax

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 Processes

Now we define processes, P, as follows:

$$
\begin{array}{rll}
P ::= & \mathbf{0} & \text{(null)} \\
\mid & p(\overrightarrow{X}) & \text{(atom, where } \overrightarrow{X} \text{ is a sequence of links)} \\
\mid & P, P & \text{(molecule)} \\
\mid & \{P\} & \text{(cell) } \dagger \\
\mid & \{P\}/ & \text{(quiet cell) } \dagger \\
\mid & (T \,\texttt{:-}\, T) & \text{(reaction rule)}
\end{array}
$$

We use the two terms *processes* and *nodes* interchangeably. A process P must observe the following *link condition*:

Link Condition: Each link in P (excluding those links occurring in reaction rules) can occur *at most twice*.

Conditions applied to links occurring in reaction rules will be given in Section 3.3.

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; $p(\overrightarrow{X})$ is an atomic process with zero or more links; P, P is parallel composition (aka *molecules*); $\{P\}$ and $\{P\}/$ are processes grouped by the membranes $\{\ \}$ (hence called *cells*); and $(T \,\texttt{:-}\, T)$ is a rewrite rule for processes. A molecule turns out to be a synonym of a process because, as will be defined later, any process P is equivalent to the molecule $\mathbf{0}, P$.

An atom $X \texttt{=} Y$ connects one side of the link X and one side of the link Y. The difference between $\{P\}$ and $\{P\}/$ is that the latter indicates that P is irreducible.

Free links of cells are unordered, while those of atoms are ordered. Free links of a cell actually emanate from their components rather than the enclosing membrane of the cell.

Note that the link condition never prevents us from composing two processes, say 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 variable 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 name before the composition. The links in reaction rules need not be fresh because reaction rules are understood to be closed.

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

3.3 Process Templates

Next we define *process templates*, T:

$$
\begin{array}{rll}
T ::= & \mathbf{0} & \text{(null)} \\
| & p(\overrightarrow{X}) & \text{(atom)} \\
| & \$p(\overrightarrow{X^*}) & \text{(process variable)} \dagger \\
| & @p & \text{(rule variable)} \dagger \\
| & T, T & \text{(molecule)} \\
| & \{T\} & \text{(cell)} \dagger \\
| & \{T\}/ & \text{(quiet cell)} \dagger
\end{array}
$$

The rules with the †'s are omitted from Flat LMNtal.

A *process variable*, $\$p(\overrightarrow{X^*})$, matches processes other than reaction rules. $\overrightarrow{X^*}$ is a possibly empty sequence of mutually different links, optionally followed by an `*`, and specifies what links may or must occur free. All links explicitly mentioned in the $\overrightarrow{X^*}$ must occur as free links, and the optional `*` means that links not mentioned explicitly may occur free as well.

Examples: `$p(X,Y)`: a process having exactly `X` and `Y` as its free links
`$p(X,*)`: a process having at least `X` as a free link
`$p`: a process with no free links (i.e., matches a closed process)

A *rule variable*, $@p$, matches a (possibly empty) multiset of rules.

Process variables and rule variables in a reaction rule must observe the following occurrence rules:

1. (*left linearity*) Each process variable and each rule variable must have *exactly one* occurrence on the left-hand side, and that occurrence must be in a cell. (An obvious consequence is that they cannot appear in Flat LMNtal.)
2. Each cell on the left-hand side may have at most one process variable and at most one rule variable at its top level.
3. (*linearity*) A process variable with a non-empty $\overrightarrow{X^*}$ must occur exactly once on the right-hand side.

As will be apparent from the semantics given in Section 4, values received by process variables and rule variables are uniquely determined by virtue of the occurrence rules above. The third rule says that a process that may contain free links cannot be copied or discarded freely. In contrast, a closed process can be copied, and this may give us an efficient means of copying a network of atoms enclosed in a membrane.

A link in each reaction rule (L `:-` R) can either

1. occur exactly twice, or
2. occur twice in L and twice in R.

$$\mathbf{0}, P \equiv P \tag{1}$$
$$P, Q \equiv Q, P \tag{2}$$
$$P, (Q, R) \equiv (P, Q), R \tag{3}$$
$$P \equiv P', \text{ if } P \text{ and } P' \text{ are } \alpha\text{-convertible} \tag{4}$$
$$P \equiv P' \Rightarrow P, Q \equiv P', Q \tag{5}$$
$$P \equiv P' \Rightarrow \{P\} \equiv \{P'\} \tag{6}$$
$$\{P\} \equiv \{P\}/, \text{ if } P \text{ is irreducible} \tag{7}$$
$$X = X \equiv \mathbf{0} \tag{8}$$
$$X = Y \equiv Y = X \tag{9}$$
$$X = Y,\ P_{X,*} \equiv P_{X,*}[Y/X] \tag{10}$$
$$\{X = Y, P\} \equiv \{P\}, X = Y \tag{11}$$

Fig. 1. Structural Congruence of LMNtal processes

Links occurring only in L are consumed links; those occurring only in R are links generated by rules; those occurring once on each side are inherited links; and those occurring twice on each side are recycled links.

Note that the syntax of T does not have $(T \text{ :- } T)$. This means that LMNtal as defined in this paper does not allow program manipulation involving inspection and/or dynamic creation of processes. However, it is possible to capture the whole rules in a membrane using rule variables and move them around. This is regarded as a higher-order construct in a limited form, which is compatible with compilation of rule sets.

4 Operational Semantics

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

4.1 Structural Congruence

We regard the left-hand and the right-hand sides of each equation in Fig. 1 as equivalent (i.e., convertible to the other in zero steps). Here, $[Y/X]$ is a *link substitution* that replaces a link X with a link Y, and $P_{\overrightarrow{X^*}}$ is a process satisfying $\overrightarrow{X^*}$.

Note that the link condition must hold in Rule (5). Rule (7) says that a terminated process can post a flag "/" as necessary. Rule (8) stands for the removal (and less interestingly, the creation) of vacuous loops and Rule (9) stands for the symmetry of =. Rule (10) says that an = process can be absorbed (and emitted) by an adjacent process. This can happen even when the adjacent process is a cell and X occurs deep in the membrane structure. Rule (11) says that an = process can freely move across membranes.

$$\frac{P \longrightarrow P'}{P, Q \longrightarrow P', Q} \quad \text{(i)} \qquad\qquad \frac{P \longrightarrow P'}{\{P\} \longrightarrow \{P'\}} \quad \text{(ii)}$$

$$\frac{P \longrightarrow P'}{Q \longrightarrow Q'} \quad (\text{if } P \equiv Q \text{ and } P' \equiv Q') \quad \text{(iii)}$$

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

Fig. 2. The reduction semantics of LMNtal

4.2 Reduction Relation

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

The left-hand side of a reaction rule may have process variables and rule variables whose rôles are to match and receive processes and rules placed in inner membranes. So we start with defining *process substitution* as follows:

$$[W_1/\$p_1(\overrightarrow{X^*}_1), \ldots, W_n/\$p_n(\overrightarrow{X^*}_n), R_1/@q_1, \ldots R_m/@q_m]$$

Here, W_i is a process that enjoys the free link specification $\overrightarrow{X^*}_i$ and does not contain reaction rules, and R_i is a multiset of reaction rules, where the p_i's and the q_i's are pairwise distinct, respectively. The above process substitution replaces a process variable $\$p_i(\overrightarrow{X^*}_i)$ with W_i and a rule variable $@q_i$ with R_i, respectively and simultaneously. Henceforth a process substitution is written as θ.

The reduction relation is given in Fig. 2. Of the four rules, the first three are familiar structural rules. Rule (i) asserts that computation can be concurrent and proceed locally. Rule (ii) asserts that computation enclosed by a membrane can evolve autonomously. Note, however, that for a cell to evolve autonomously, it must have its own set of reaction rules. A cell without reaction rules should be controlled by rules outside the membrane. Rule (iv) is the central rule of LMNtal. It says that reaction rules themselves are preserved (as catalysts are). The Flat LMNtal version of Rule (iv) degenerates to:

$$\frac{}{P, (P \text{ :- } Q) \longrightarrow Q, (P \text{ :- } Q)} \quad \text{(iv}')$$

The matching of corresponding local links in a redex process and those in the left-hand side of a rule is done implicitly using α-conversion. One may wonder if the process `p(A,A)` can be reduced using a rule `p(X,Y) :- q(X,Y)`, because the rule cannot be α-converted to the form `p(A,A) :- ...` . However, since `p(A,A)` is equivalent to `p(A,B),A=B` (`B` being fresh), one can reduce it as:

$$\texttt{p(A,A)} \equiv \texttt{p(A,B),A=B} \longrightarrow \texttt{q(A,B),A=B} \equiv \texttt{q(A,A)}.$$

Note that the right-hand side of Rule (iv) must enjoy the link condition. When a new link is created upon reduction, name conflict should be avoided by α-converting the rule (T :- U) appropriately before use.

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 $\mathtt{c}(A_1, X_1, X_0)$, ..., $\mathtt{c}(A_n, X_n, X_{n-1})$, $\mathtt{n}(X_n)$. 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 $X_0 = \mathtt{c}(A_1, X_1)$, ..., $X_{n-1} = \mathtt{c}(A_n, X_n)$, $X_n = \mathtt{n}$ in 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
```

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.
```

In this way, LMNtal has eliminated syntactic distinction between processes and data, though an optimizing compiler might implement them differently.

The above program resembles `append` in Interaction Nets [8] proposed more than a decade ago. 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" [8]. LMNtal generalizes Interaction Nets by removing the constraint of 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),$p(*)}, c(A,X,X0) :-
        c(A,Y,Y0), {i(X),o(Y),$p(*)}
```

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 variable `$p(*)` is to match the rest of the input streams and pass them to the right-hand side. Fig. 3 shows a redex to which the above rewrite rule is applicable.

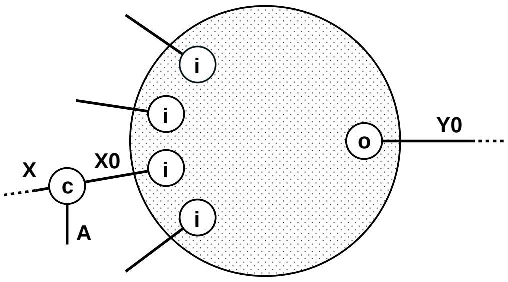

Fig. 3. Multiway Stream Merging

5.3 Process Migration

Consider two cells sharing a communication link. Most of the time, they run independently using individual sets of reaction rules, but sometimes one migrates processes to the other through the link. The rule for migration is given at an outside layer.

It is the rôle of the outside 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 go(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.

```
{$p(S,*), {@q,$q(*),go(S,D)}}, {$r(D,*)} :-
        {$p(S,*)}, {{@q,$q(*),arrived(S,D)}, $r(D,*)}
```

The innermost membrane enclosing `@q,$q(*),go(S,D)` is to specify what resources are to be migrated. When `@q` 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 from `D` to `S` after migration, that is, `D` becomes a local link in the destination site. This is an important characteristic of logical links. The membrane delimiting migrated resources can be removed in the destination site.

5.4 Reaction Control

Suppose we want to simulate a chemical experiment which uses three test tubes (say `X`, `Y` and `Z`, which are links for identifying the test tubes from outside), of which the tube `Z` is initially empty. The experiment starts with two independent reactions within `X` and `Y`, and when the reactions terminate, the final products are transferred to `Z`. The rule for the transfer can be written as follows:

```
{id(X),$p,@p}/, {id(Y),$q,@q}/, {id(Z)} :-
        {id(X),@p}, {id(Y),@q}, {id(Z),$p,$q}
```

Note that termination is a global property of a membrane and can be checked only from outside the membrane. This is compatible with the local nature of LMNtal reaction rules that can be fired by examining and locking a small number of relevant processes rather than the whole universe.

5.5 Arithmetics

Let us consider the treatment of natural numbers based on their first principle.

In LMNtal, $\mathtt{a}(X), \mathtt{5}(X)$ can be interpreted as a configuration in which the process `a` has the value `5`. Now suppose `a` increments its argument and evolves into `b`. This can be achieved by giving rules that realize the following reductions:

$$\begin{aligned} \mathtt{a(X),\ 5(X)} &\longrightarrow \mathtt{b(X1),\ s(X1,X),\ 5(X)} \\ &\longrightarrow \mathtt{b(X1),\ 6(X1)} \end{aligned}$$

The process `a` sends a message `s` in the first reduction, then `s` and `5` react in the second reduction.

In practice, numbers (and operations on numbers) will be provided as built-in data types. In this case, it is convenient to think of $\mathtt{5}(X)$ as equivalent to

$$\mathtt{s}(X, X_4), \mathtt{s}(X_4, X_3), \mathtt{s}(X_3, X_2), \mathtt{s}(X_2, X_1), \mathtt{s}(X_1, X_0), \mathtt{0}(X_0).$$

The second reduction above will then become a congruence relation. This is convenient because one can match $\mathtt{s}(X, Y)$ against $\mathtt{5}(X)$ to check if the latter is positive and simultaneously to obtain $\mathtt{4}(Y)$, which is likely to be passed to a tail-recursive process.

5.6 Cyclic Data Structures

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

$$\mathtt{b}(S, L_n, L_0), \mathtt{n}(A_1, L_0, L_1), \ldots, \mathtt{n}(A_n, L_{n-1}, L_n),$$

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(S1,S), n(A,L0,L1), b(S,L1,L2) :-
        b(S1,L0,L1), n(A,L1,L2)
right(S1,S), b(S,L0,L1), n(A,L1,L2) :-
        n(A,L0,L1), b(S1,L1,L2)
put(A,S1,S), b(S,L0,L2) :-
        n(A,L0,L1), b(S1,L1,L2)
...
```

Shape Types [6] are another attempt to facilitate manipulation of dynamic data structures. Interestingly, Shape Types took a dual approach, namely they used variables to represent nodes and names to represent links.

6 Conclusions

We have proposed 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. Design of several interesting extensions is underway, which include:

1. negative conditions (i.e., rule firing based on the non-existence of certain process structures in a membrane),
2. the *void* construct [13] for resource-conscious programming,
3. various syntactic sugars, including guards, vectors, and functional notations,
4. encoding of names in terms of links and membranes, and
5. more relaxed occurrence rules of process variables.

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.

Many things remain to be done. The most important issue in the language design is to equip it with various 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 uni-directional pointers), and properties about free links of cells, can be guaranteed statically using type systems. Other important design issues include exception handling and meta-level architectures.

Experimental implementation of LMNtal is underway. Representation of processes and links, optimizing compilation of reaction rules, and parallel and distributed implementation are examples of challenging topics in our implementation project. At the present writing, we have built a compiler and a runtime of a reasonably large subset of the language (with membranes) in Java.

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 ((C)(2) 11680370, Priority Areas (C)(2)13324050 and (B)(2)14085205), Ministry of Education. This work has been conducted as part of the *Universal Computing* project of the 21st Century COE Program, *Productive ICT Academia*, Waseda University.

References

1. Andries, M. *et al.*, Graph Transformation for Specification and Programming. *Sci. Comput. Program.*, Vol. 34, No. 1 (1999), pp. 1–54.
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, 1990, pp. 81–94.
4. Drewes, F., Hoffmann, B. and Plump, D., Hierarchical Graph Transformation. *J. Comput. Syst. Sci.*, Vol. 64, No. 2 (2002), pp. 249–283.
5. 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.
6. Fradet, P. and Le Métayer, D., Shape Types. In *Proc. POPL'97*, ACM, 1997, pp. 27–39.
7. Frühwirth, T., Theory and Practice of Constraint Handling Rules. *J. Logic Programming*, Vol. 37, No. 1–3 (1998), pp. 95–138.
8. Lafont, Y., Interaction Nets. In *Proc. POPL'90*, ACM, 1990, pp. 95–108.
9. Milner, R., Bigraphical Reactive Systems. In *Proc. CONCUR 2001*, LNCS 2154, Springer, 2001, pp. 16–35.
10. Păun, Gh., Computing with Membranes. *J. Comput. Syst. Sci.*, Vol. 61, No. 1 (2000), pp. 108–143.
11. 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.
12. 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.
13. Ueda, K., Resource-Passing Concurrent Programming. In *Proc. TACS 2001*, LNCS 2215, Springer, 2001, pp. 95–126.