Declarative Priority In A Concurrent Logic Language $\mathsf{O_N}$

Keiji Hirata
NTT Communication Science Laboratories
3-1, Morinosato Wakamiya, Atsugi-shi, Kanagawa, 243-0198 Japan
hirata@brl.ntt.co.jp

Kenichi Yamazaki
NTT Network Innovation Laboratories
3-9-11 Midoricho, Musashino-shi, Tokyo, 180-8585 Japan
yamazaki@t.onlab.ntt.co.jp

Abstract

It is well known that priority control is essential for real-world problems, and indeed, many real-world-oriented concurrent logic/constraint languages, such as KL1 and Oz, can deal with priority in explicit or implicit ways. However, the design policies of these languages in terms of priority were ad hoc, since priority has been considered non-logical. It turns out that its procedural meaning has been given in an informal way at most. Our aim is to construct a formal declarative semantics of a prioritized program in order to increase the applicability of concurrent logic languages to real-world problems. In this paper, we first define a model of priority and prove some properties of the model. Then, we design a concurrent logic language called $\mathsf{O_N}$ based on our theoretical framework and discuss some prominent characteristics that are embodied in sample programs written in $\mathsf{O_N}$. The results presented in the paper provide a new insight into priority from the declarative point of view.

1 Introduction

The goals of logic programming include giving a clear-cut boundary between "what to do" and "how to do it". According to Kowalski's famous article [7], what-to-do corresponds to the logic component, and how-to-do-it to the control component. Declarative programming is the concept that a programmer concentrates on what-to-do and leaves how-to-do-it to a language processing system as much as possible. Concurrent logic programming languages (CLLs) have been developed according to this concept. Because of the what-to-do concept, CLLs have excellent programmability, and hence many of them have been successfully used for various reactive, parallel and distributed applications.

On the other hand, certain application fields still require a programmer to describe how-to-do-it. There are some cases where the how-to-do-it description contributes to speedup, and others where it is absolutely necessary in order to ensure execution control proceeds exactly as a programmer intends. In particular, for real-world concurrent processing, specifying how-to-do-it is inevitable and essential; typical and familiar examples are interrupt processing, exception handling, speculative computation, and real-time processing. The standard way to control processes according to the programmer's intention is to give a priority to each process. The concept of priority is by nature incompatible with CLLs because of their fundamental design policy, i.e. specifying what-to-do as much as possible. Therefore, it is in general considered difficult to incorporate priority into the framework of CLLs.

The authors think that in order to extend the applicability of CLLs to real-world problems, priority should be incorporated into their framework. There are three approaches to doing this:
(1) Incorporate a meta-level programming feature. In this approach, we can prioritize all object-level activities but not meta-level ones, because priority can be handled as a first-class data only from a meta-level interpreter.
(2) Provide built-in predicates and/or system libraries dedicated to priority handling. Huntback proposed a predicate for interrupt processing that checks message arrival [6]. Gregory proposed a predicate for speculative computation that checks whether a processor is idle [3]. Since such predicates are ad hoc and inconsistent from the execution-model point of view, it is difficult to understand priority in a declarative way.
(3) Reform a language so that it has a priority annotation but retains its core semantics intact. Suppose that we have two versions of a program, a prioritized one P and a non-prioritized plain one O. Let A_O be the set of all possible answers of program O and A_P that of program P. Then, we have $A_P \subseteq A_O$. Since priority just shrinks the set of answers, a priority annotation is in general regarded as logically transparent. The attempts with this approach are KL1 [11] and the work of Huntback [5]. However, in these languages, the operational semantics in terms of priority is not defined formally; priority is treated just as an implementation-dependent feature to the extent that scheduling is carried out on the best-effort basis.

These classification suggests us a new approach which we adopt in this paper; our standpoint is that (a) priority should be first-class data, (b) priority should be declaratively understood, and (c) priority should have an operational and model-theoretic semantics; a similar statement in CSP is found in [2]. From these points of view, we design a new language that amalgamates priority and the framework of CLLs so that we can give a formal semantics to priority.

This paper is organized as follows. Section 2 defines a model of priority and proves some crucial properties of the model. Section 3 designs a concurrent logic language called O_N based on our framework; priority is introduced as a first-class data to control process invocation and message passing. Then, the operational semantics of O_N is stated. Section 4 demonstrates sample programs written in O_N, including prioritized merge and prioritized event loop, and discusses some prominent characteristics that are embodied in these programs. Section 5 concludes the paper and addresses future work.

2 Formalizing Priority

This section first informally describes our design principle of a new data type and its meaning for priority, formally defines it, and proves some useful properties.

2.1 What Priority Should Be

Priority controls concurrent activities, such as process invocation and message passing, to make nondeterministic execution more or fully deterministic. The relations among priorities directly and indirectly reflect the execution order of concurrent activities. Conventional priority systems use an integer to represent priority in general. We think that this integer-based priority involves serious problems.

First, we believe a priority relation should be *binary* and *relative.* Let us consider the situation where a programmer decides the priority of each process and cannot statically predict what and how many concurrent activities are dynamically spawned. The programmer can decide that a concurrent activity should be executed prior to the other, but he/she can not make a total ordering among several or more concurrent activities. Hence, we think that the binary and relative priority relation is intuitively understandable; it allows us to read and write prioritized programs with relative ease. On the other hand, the integer-based priority relations are total and absolute.

Next, we maintain that the priority relations needed for execution control are as follows: (1) *equal*, (2) *higher* or *lower*, and (3) *unrelated.* There may be no need to explain (2). *Equal* means that the same-priority process should be scheduled fairly. *Unrelated* is further split into two cases: (3a) a programmer does not care about the order of two concurrent activities and (3b) he/she cannot make the order even if he/she is required to. Either way, it is impossible to express this intention using integer-based priority. In a sequential environment, both the "equal" relation and the "unrelated" relation may lead to (fairly) nondeterministic execution. However, in a parallel and/or distributed environment, these two relations may yield different behaviors and effects. As for (3b), let us consider a distributed environment. When a programmer wants to execute a activity A at a remote processor, he/she must know the priorities of all the activities that already exist at the remote processor in order to prioritize A. We think this is unrealistic, and therefore relations (1) and (3) should be distinguished.

Note that for instance, the equal-or-higher relation is meaningless, because there is no distinction between the behavior of this relation and that of the equal relation.

Finally, we believe a priority relation should be *stable.* To treat priority declaratively, once a priority relation is fixed, it should never change. This implies that an unrelated relation should remain unrelated. Suppose that there are two unrelated priorities A and B, and a process with A is executed earlier than one with B by chance. If a new priority relation, say, B > A is defined after the execution of the second process (one with B), the actual execution order (A then B) contradicts the defined priority relation B > A. We think that such a situation should not happen.

From our first claim, we base our priority system on a binary, relative relation, denoted as '$\succ$'. From the second, to create an equal priority, we explicitly use '$=$'. As long as two priorities are not explicitly related to each other using $\succ$ and/or $=$, they are unrelated. From the third, we slightly extend the $\succ$ relation and introduce a language constructor for priority definition of the form $(H_1, H_2, \cdots) \succ P \succ (L_1, L_2, \cdots)$ to create a priority P. The form used is as follows. Here we introduce special symbols $\top$ and $\bot$, which mean the top and bottom priorities respectively. Then, if for a given priority Q we create a new priority P higher or lower than Q (satisfying $P \succ Q$ or $Q \succ P$), we write $\top \succ P \succ Q$ or $Q \succ P \succ \bot$. If for two given priorities P and Q (suppose $P \succ Q$) we create a new in-between priority R satisfying $P \succ R \wedge R \succ Q$, we just write $P \succ R \succ Q$. If we create two unrelated priorities P, Q, we write $\top \succ P \succ \bot \wedge \top \succ Q \succ \bot$. It is advantageous that the condition for stability can be simply stated with the form $(H_1, \cdots, H_n) \succ P \succ (L_1, \cdots, L_m)$. That is, this form requires that the condition $\forall i \forall j H_i \succ L_j$ holds before a new priority P is put between H_i and L_j. At the same time, this form guarantees that the condition holds after the new priority is created.

2.2 Term Model for Priority

Let us elaborate on what information a priority has to keep as a term. The form of a priority definition implies a priority P can be regarded as the function of H_i and L_j. We also have to distinguish A and B created by $\top \succ A \succ \bot \wedge \top \succ B \succ \bot$. Therefore, a priority should have the information of an identification, higher priorities H_i, and lower priorities L_i.

`C` is the set of all constant symbols. `Var` is the set of all variable names. $Var(t)$ is the set of variables occurring in a term t and is defined as usual.

An *atomic formula* is of the form $A \star B$, where A and B are variables, $\top$ (the top symbol), or $\bot$ (the bottom symbol), and $\star$ is either $\succ$ or $=$. A *formula* is constructed from atomic formulas and connectives, $\wedge, \vee$, and $\neg$ as usual. A *structure* consists of a domain and an interpretation. A *valuation* into a structure is a total function from variables to the domain of the structure.

Definition A *priority definition* is of the form $(H_1, \cdots, H_n) \succ P \succ (L_1, \cdots, L_m)$, where P is a variable, and H_i $(i = 1..n)$ and L_j $(j = 1..m)$ are variables, $\top$, or $\bot$.

Definition Let `C` be the set of all constants. Then $\mathcal{T}$ is a *priority type* $\mathtt{C} \times 2^{\mathtt{C}} \times 2^{\mathtt{C}}$, and an instance $t \in \mathcal{T}$ is represented as a triplet $\langle a, \{b_1, \cdots, b_n\}, \{c_1, \cdots, c_m\}\rangle$, where $a, b_1, \cdots, b_n, c_1, \cdots, c_m$ are all constants. Here a works as an identifier of a priority.

If $t \in \mathcal{T}$ is $\langle e_1, e_2, e_3 \rangle$, we introduce a dotted notation $t.i$ to access a component e_i $(i = 1, 2$ or $3)$. We use $\mathcal{T}$ as the set of all instances of type $\mathcal{T}$.

Definition Π is $\{\top, \bot\} \cup \mathcal{T}$. There are two kinds of binary relations on Π, denoted as $\succ$ and $=$, and these are subsets of $\Pi \times \Pi$, respectively. The rules prescribing these relations and special constants are as follows: for

$\alpha, \beta, \gamma, \xi \in \Pi$, (R1) $\alpha \succ \xi$ if $\alpha.1 \in \xi.2$, (R2) $\xi \succ \beta$ if $\beta.1 \in \xi.3$, (R3) $\alpha \succ \beta \wedge \beta \succ \gamma \rightarrow \alpha \succ \gamma$, and (R4) $\alpha = \beta$ if $\alpha.i = \beta.i$ ($i = 1, 2$ and 3); $\top = \top$ and $\bot = \bot$, (R5) $\forall \alpha \in \Pi. \top \succ \alpha \vee \top = \alpha$, (R6) $\forall \alpha \in \Pi. \alpha \succ \bot \vee \bot = \alpha$. Since we intend to build a term model for priority, we adopt Π as a semantic domain and call it a *priority domain*. The relations and the symbols defined by rules (R1)~(R6) are also straightforwardly mapped to the correspondings on the priority domain Π. Although strictly speaking, we should write $\top_\Pi$, $\bot_\Pi$ $\succ_\Pi$ and $=_\Pi$ on Π, we will use $\top$, $\bot$, $\succ$ and $=$ instead as long as there is no confusion.

It follows from the above definition that $\top \succ \bot$ holds.

Definition $\mathcal{P}$ is a *priority structure* consisting of the priority domain Π and the canonical interpretation.

Definition The realizability of a set of formulas is defined in the standard way [9]. A valuation Γ *realizes* a formula ϕ in the priority structure $\mathcal{P}$ and we write $\mathcal{P}, \Gamma \models \phi$, if for $\{V_1, \cdots, V_n\} = \mathit{Var}(\phi), \mathcal{P} \models \phi[\Gamma(V_1), \cdots, \Gamma(V_n)]$, where $\phi[\alpha_1, \cdots, \alpha_n]$ means that $\alpha_1, \cdots, \alpha_n (\in \Pi)$ are respectively put as the values of free variables $V_1, \cdots, V_n$ in ϕ.

A valuation is also represented in a set of pairs of a variable and a value: $\Gamma = \{\langle X_1, \xi_1 \rangle, \langle X_2, \xi_2 \rangle, \cdots\}$.

Definition Let $\mathcal{D}$ be a finite set of priority definitions. We now present an algorithm to compute a valuation of $\mathcal{D}$ into the priority structure $\mathcal{P}$, denoted as Γ.
(S1) $\mathcal{D}_1 = \mathcal{D}$ and $\Gamma_1 = \{\}$.
(S2) Suppose that we have $\mathcal{D}_k$ and Γ_k. Choose $d \in \mathcal{D}_k$ such that $d = (H_1, \cdots, H_n) \succ P \succ (L_1, \cdots, L_m)$ and $\forall i \forall j. \mathcal{P} \models \Gamma_k(H_i) \succ \Gamma_k(L_j)$. Then, we create a new element $\xi (\in \Pi)$, which is $\langle$"P",$\{\Gamma_k(H_1).1, \cdots, \Gamma_k(H_n).1\}$, $\{\Gamma_k(L_1).1, \cdots, \Gamma_k(L_m).1\}\rangle$, where "$P$" is a fresh constant generated from the variable name of P. $\Gamma_{k+1} = \{\langle P, \xi \rangle\} \cup \Gamma_k$ and $\mathcal{D}_{k+1} = \mathcal{D}_k \setminus \{d\}$. Iterate Step (S2), while $\mathcal{D}_k \neq \emptyset$.
(S3) If $\mathcal{D}_k = \emptyset$, $\Gamma = \Gamma_k \cup \{\langle X, \zeta \rangle, \cdots\}$, where X's are $\texttt{Var} \setminus \mathit{Var}(\mathcal{D})$ and ζ's are arbitrary elements in Π.

If this computation meets the following two cases at Step (2), it terminates halfway; Γ is not generated. One is that there is no d found in $\mathcal{D}_k$ satisfying $\forall i \forall j. \mathcal{P} \models \Gamma_k(H_i) \succ \Gamma_k(L_j)$. The other is that for a variable P, $\langle P, \xi \rangle$ is added more than once. At Step (S3), adding set $\{\langle X, \zeta \rangle, \cdots\}$ makes Γ_k a total function. Since $\mathcal{D}$ is finite, $\mathcal{D}_k = \emptyset$ is reached eventually, and the algorithm always terminates.

The algorithm for computing the valuation presented above is nondeterministic. This is because there are several choices for a priority definition $d \in \mathcal{D}_k$ to create a new element ξ. Thus, the more than one valuation of $\mathcal{D}$ may be computed with different sequences of d. However, the following proposition states that the valuations are unique.

Definition A finite set of priority definitions $\mathcal{D}$ is *well-formed* if there exists a sequence of d chosen at Step (S2) so that we successfully reach $\mathcal{D}_k = \emptyset$ and obtain an answer valuation Γ.

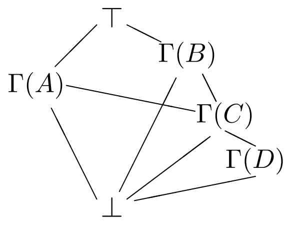
Proposition 2.1 *(Declarative Priority)*
Let $\mathcal{D}$ be a finite well-formed set of priority definitions. Then, the computation by the above algorithm always reaches the identical answer valuation independently from the selection sequences of $d \in \mathcal{D}$.

Outline of proof. It is obvious that all valuations, if they exist, are identical, because ξ is always uniquely determined at Step (S2). It hence suffices to show that the algorithm can always choose a priority definition at every Step (S2) and finally moves to Step (S3). Since $\mathcal{D}$ is well formed, there exists a selection sequence of priority definitions, denoted by S_1. We use proof by contradiction. Suppose that at some step (S2), the algorithm can not choose proper d, and let χ be the set of the priority definitions already selected until the step. Here we consider the sequence of priority definitions by deleting χ from the sequence S_1, denoted by S_1'. Note that the left most priority definition of S_1' can be chosen at Step (S2). Thus, if the algorithm selects the left most one of S_1' as d, it can generate a new ξ. Hence contradiction. Since $\mathcal{D}$ is finite, the algorithm always reaches $\mathcal{D}_k = \emptyset$. □

Definition Let $\mathcal{D}$ be a finite well-formed set of priority definitions and Γ the valuation of $\mathcal{D}$ into the priority structure $\mathcal{P}$ computed by the above algorithm. Then, the *term model* of $\mathcal{D}$ and rules (R1)$\sim$(R6) consists of priority domain Π, the canonical interpretation, and Γ (namely $\mathcal{P}$ and Γ).

Example 2.2 *Suppose that the set of priority definitions $\mathcal{D}$ is $\{\top \succ A \succ \bot, \top \succ B \succ \bot, (A,B) \succ C \succ \bot, C \succ D \succ \bot\}$. Then, we obtain the valuation* $\Gamma = \{\, \langle A, \langle \text{“}A\text{”}, \{\top\}, \{\bot\}\rangle\rangle, \langle B, \langle \text{“}B\text{”}, \{\top\}, \{\bot\}\rangle\rangle, \langle C, \langle \text{“}C\text{”}, \{\text{“}A\text{”}, \text{“}B\text{”}\}, \{\bot\}\rangle\rangle, \langle D, \langle \text{“}D\text{”}, \{\text{“}C\text{”}\}, \{\bot\}\rangle\rangle \,\} \cup \{\langle X, \zeta\rangle, \cdots\}$, *where “A”, “B”, “C” and “D” are distinct constants, X's are* $\texttt{Var} \setminus Var(\mathcal{D})$ *and ζ's are arbitrary elements in* Π.

The following graph depicts the priority relations of $\mathcal{D}$; the righthand side shows all the priority relations deduced from $\mathcal{D}$.



$$\begin{array}{lll}
\top \succ \Gamma(A), & \Gamma(A) \succ \bot, & \Gamma(A) \succ \Gamma(C),\\
\top \succ \Gamma(B), & \Gamma(B) \succ \bot, & \Gamma(B) \succ \Gamma(C),\\
\top \succ \Gamma(C), & \Gamma(C) \succ \bot, & \Gamma(C) \succ \Gamma(D),\\
\top \succ \Gamma(D), & \Gamma(D) \succ \bot, & \Gamma(A) \succ \Gamma(D),\\
 & & \Gamma(B) \succ \Gamma(D)
\end{array}$$

2.3 Properties

Definition Let $\mathcal{P}$ be the priority structure. For a formula ϕ, let $\phi^{\mathcal{P}}$ denote the set of all valuations $\{\Gamma \mid \mathcal{P}, \Gamma \models \phi\}$. Then, for formulas ψ and ϕ, we say that ψ *entails* ϕ and write $\psi \models \phi$ if $\psi^{\mathcal{P}} \subseteq \phi^{\mathcal{P}}$.

The new element ξ computed at Step (S2) of the above algorithm, if it exists, always satisfies the relation $\Gamma_k(H_1) \succ \xi \wedge \cdots \wedge \Gamma_k(H_n) \succ \xi \wedge \xi \succ \Gamma_k(L_1) \wedge \cdots \wedge \xi \succ \Gamma_k(L_m)$ because of rules (R1) and (R2) in Section 2.2, and this relation does not violate the existing priority relations. This implies that a priority definition $(H_1, \cdots, H_n) \succ P \succ (L_1, \cdots, L_m)$ can be logically regarded as $H_1 \succ P \wedge \cdots \wedge H_n \succ P \wedge P \succ L_1 \wedge \cdots \wedge P \succ L_m$. Therefore, we can write $\mathcal{P}, \Gamma \models \mathcal{D}$, $\mathcal{D} \models A \succ B$ if $\mathcal{D}$ entails $A \succ B$, and $\mathcal{D} \models \neg(A \succ B)$ if it does not. Similarly, $\mathcal{D} \models A = B$ if $\mathcal{P}, \Gamma \models A = B$. Also, $\mathcal{D} \models A \not\sim B$ is a notational convenience for $\mathcal{D} \models \neg(A \succ B \vee B \succ A \vee A = B)$.

Lemma 2.3 *Let $\mathcal{D}$ be a well-formed set of priority definitions, and let $\mathcal{V}$ be $\{\top, \bot\} \cup Var(\mathcal{D})$. Then, for $A, B \in \mathcal{V}, \mathcal{D} \models \neg(A \succ B \wedge B \succ A)$ and $\mathcal{D} \models \neg(A \succ A)$.*

Outline of proof. Induction on the construction of a valuation and proof by contradiction are used. $\square$

The lemma shows that our priority is transitive but neither antisymmetric nor reflexive. This property corresponds to that of the '<' predicate of the Tempo language [4].

Proposition 2.4 *(Satisfaction Completeness of Priority Relations)*
Let $\mathcal{D}$ be a well-formed set of priority definitions, and let $\mathcal{V}$ be $\{\top, \bot\} \cup Var(\mathcal{D})$. Then, for $A, B \in \mathcal{V}, \mathcal{D} \models (A \succ B \vee B \succ A \vee A = B \vee A \not\sim B)$, and these cases are mutually execution.

Outline of proof. In the case of $\Gamma(A) = \Gamma(B)$, the proof is straightforward. Otherwise, it follows from Lemma 2.3. $\square$

Thus, a well-formed set of priority definitions is consistent.

Proposition 2.5 *(Stability of Priority Relations)*
Let $\mathcal{D}$ and $\mathcal{D}'$ be finite well-formed sets of priority definitions satisfying $\mathcal{D} \subseteq \mathcal{D}'$, and let $\mathcal{V}$ be $\{\top, \bot\} \cup Var(\mathcal{D})$. Then, for $A, B \in \mathcal{V}, \mathcal{D} \models A \star B \Rightarrow \mathcal{D}' \models A \star B$, where $\star$ is either $\succ$, $=$ or $\not\sim$.

Outline of proof. Since $\mathcal{D} \subseteq \mathcal{D}'$, new priority definitions are just added to $\mathcal{D}$ and we get $\mathcal{D}'$. Consider the following cases: (i) $\mathcal{D} \models A \succ B$ is transformed to $\mathcal{D}' \models (A \succ B \wedge A \diamond B)$, (ii) $\mathcal{D} \models A = B$ to $\mathcal{D}' \models (A = B \wedge A \succ B)$, and (iii) $\mathcal{D} \models A \not\sim B$ to $\mathcal{D}' \models A \diamond B$, where $\diamond$ is either $\succ$ or $=$. It is proved by Proposition 2.4 that cases (i) and (ii) do not occur. It follows from the construction of a valuation Γ that case (iii) does not occur. $\square$

2.4 Aggregate Priority

Definition Let $\mathcal{D}$ be a well-formed set of priority definitions, and $\{\vec{A}\}$ and $\{\vec{B}\}$ nonempty sets of priority variables. First, we define a partial ordering between these sets: $\mathcal{D} \models \{\vec{A}\} \succeq \{\vec{B}\}$ if $(\forall X \in \{\vec{A}\} \exists Y \in \{\vec{B}\}. \mathcal{D} \models X \succ Y \vee X = Y) \wedge (\forall Y \in \{\vec{B}\} \exists X \in \{\vec{A}\}. \mathcal{D} \models X \succ Y \vee X = Y)$. Then, $\mathcal{D} \models \{\vec{A}\} = \{\vec{B}\}$ if $\mathcal{D} \models \{\vec{A}\} \succeq \{\vec{B}\} \wedge \{\vec{B}\} \succeq \{\vec{A}\}$, and $\mathcal{D} \models \{\vec{A}\} \succ \{\vec{B}\}$ if $\mathcal{D} \models \{\vec{A}\} \succeq \{\vec{B}\} \wedge \{\vec{A}\} \neq \{\vec{B}\}$. Also, $\mathcal{D} \models \{\vec{A}\} \not\sim \{\vec{B}\}$ is a notational convenience for $\mathcal{D} \models \neg(\{\vec{A}\} \succ \{\vec{B}\} \vee \{\vec{B}\} \succ \{\vec{A}\} \vee \{\vec{A}\} = \{\vec{B}\})$.

The relations $\succ$, $=$ and $\not\sim$ between two aggregate priorities have the same properties as those between two priorities; that is, $\succ$ for $\{\vec{A}\} \succ \{\vec{B}\}$ is transitive but neither antisymmetric nor reflexive. Lemma 2.3 and Propositions 2.4 and 2.5 can be easily extended for aggregate priority.

For example, let $\mathcal{D}$ be a finite well-formed set of priority definitions, and suppose $\mathcal{D} \models A \succ B$. Then $\mathcal{D} \models \{A\} \succ \{A, B\} \wedge \{A, B\} \succ \{B\}$ holds.

The aggregate priority is used for defining an operational semantics of a new language in Section 3.3.

3 New Language

The $\mathsf{O_N}$ language is a concurrent logic language that is integrated with priority, the semantics of which is based on the framework given in Section 2.

3.1 Syntax

The syntax of $\mathsf{O_N}$ is based on FGHC [11] (Fig. 1).

Program	::=	:- Calls. Defs.	
Calls	::=	Calls, Calls $\mid$ true $\mid$ Unif $\mid$ Pred $\mid$ PDef	
Defs	::=	Defs. Defs $\mid$ Pred :- Guard $\mid$ Calls.	
Guard	::=	Guard, Guard $\mid$ true $\mid$ Unif	
Unif	::=	Var $\overset{\mathsf{PVar}}{=}$ Term	$\cdots$ priority-annotated unification
Pred	::=	p(Arg,$\cdots$,Arg)$^{\mathsf{PVar}}$	$\cdots$ priority-annotated predicate
PDef	::=	(Prios) $\succ$ PVar $\succ$ (Prios)	$\cdots$ priority definition
Prios	::=	Prios, Prios $\mid$ $\top$ $\mid$ $\bot$ $\mid$ PVar	
Arg	::=	Atom $\mid$ Var $\mid$ PVar	
Term	::=	Atom $\mid$ Var $\mid$ f(Arg,$\cdots$,Arg)	

Figure 1: Syntax

Here, p in Pred represents a predicate name, and f in Term a function name. The structures of a predicate and a function are flat. Atom, Var and PVar respectively represent an atom, a logic variable, a priority variable. The domain of a logic variable is the Herbrand universe, and that of a priority variable is Π introduced in Section 2.2. As a convention in this paper, we will use P, Q, R, H, L to range over PVar. Here, we impose the following syntactical constraint: for every definition clause $p(V_1, \cdots, V_n)^P$:- $V_{i_1} \overset{Q_1}{=} t_1, \cdots, V_{i_l} \overset{Q_l}{=} t_l$ | Calls, variables $V_1, \cdots, V_n, \mathit{Var}(t_1), \cdots, \mathit{Var}(t_l), P, Q_1, \cdots, Q_l$ are distinct. This syntactical constraint enables us to explicitly add a priority annotation to every predicate call and every active/passive unification.

There are two kinds of concurrent activities in CLLs, process invocation and message passing. Prioritizing goal reduction and active unification in $\mathsf{O_N}$ corresponds to controlling process invocation and message passing, respectively. The intention of a priority-annotated active unification $X \overset{P}{=} t$ is to send a message with a priority P added, while that of a priority-annotated passive unification $X \overset{P}{=} t$ is to receive a message t and its associated priority P. The intention of a priority-annotated predicate call $p(V_1, \cdots, V_n)^P$ is to call a process with a priority P. If a callee is a definition clause $p(V_1, \cdots, V_n)^P$:- $V_{i_1} \overset{Q_1}{=} t_1, \cdots, V_{i_l} \overset{Q_l}{=} t_l$ | Calls, the priority that is associated with the call is represented as the aggregate priority $\{P, Q_1, \cdots, Q_l\}$. Note that since $P, Q_1, \cdots, Q_l$ are all elements of the set, the priority of a process invocation and that of message passing are treated equally.

3.2 Priority of Passive Unification

Definition $\Phi(G)$ is the set of priority variables occurring in Guard G; $\Phi(G) = \{Q_1, \cdots, Q_l\}$ if G is the Guard part of a definition clause, $V_1 \overset{Q_1}{=} t_1, \cdots, V_l \overset{Q_l}{=} t_l$.

We consider the priorities to which $\Phi(G)$ are instantiated. Now we have the following two types of passive unifications in $\mathsf{O_N}$: $X \overset{P}{=} f(Y_1, \cdots)$ and $X \overset{P}{=} Y$. As for the former, it is usually expected that a corresponding active unification $X \overset{Q}{=} f(Z_1, \cdots)$ has been or will be executed elsewhere in a program. Hence, we also have to care about the implicit unifications $Y_1 = Z_1, \cdots$ that are subsequently spawned.

For simplicity, in this paper, we suppose that a program is well-moded [12]. This guarantees that there is at most one writer for every variable. Then, it suffices that the following three patterns are taken into account: (i) P for a passive unification $X \overset{P}{=} t$, (ii) P for a passive unification $X \overset{P}{=} Y$ and (iii) P_i for an implicit active unification $Y_i \overset{P_i}{=} Z_i$ when we have an active unification $X \overset{Q}{=} f(\cdots, Z_i, \cdots)$ and a passive unification $X \overset{P}{=} f(\cdots, Y_i, \cdots)$. Here, let t be a Term and $\mathcal{D}$ a set of priority definitions.

For (i), suppose that the chain of active unifications which instantiates a variable X is of the form $X \overset{P_1}{=} X_1, X_1 \overset{P_2}{=} X_2, \cdots, X_{n-1} \overset{P_n}{=} t$. Then, P for a passive unification $X \overset{P}{=} t$ is bound to the minimum value among the values of $P_1, P_2, \cdots, P_n$, denoted as $P_1 \downarrow \cdots \downarrow P_n$. Operator $\downarrow$ is defined as follows:

$$P_i \downarrow P_j = \begin{cases} P_j & \cdots\ \mathcal{D} \models P_i = P_j \text{ or } \mathcal{D} \models P_i \succ P_j \\ Q \text{ defined by } (P_i, P_j) \succ Q \succ \bot & \cdots\ \text{otherwise.} \end{cases}$$

Operator $\downarrow$ is commutative and associative. This rule means that the lowest priority on a chain determines the whole priority.

For (ii), suppose that the priority variables of all active unifications which contribute to the bindings of X and/or Y are $P_1, \cdots, P_n$. Then, P for a passive unification $X \overset{P}{=} Y$ is $P_1 \downarrow \cdots \downarrow P_n$. For example, when (a) there are active unifications $X \overset{P_1}{=} Z, Z \overset{P_2}{=} Y$, (b) $X \overset{P_1}{=} t, Y \overset{P_2}{=} t$, and (c) $X \overset{P_1}{=} Y, Y \overset{P_2}{=} t$, P for a passive unification $X \overset{P}{=} Y$ is $P_1 \downarrow P_2$ in each case.

For (iii), suppose that an implicit active unification subsequently spawned is $Y_i \overset{P_i}{=} Z_i$ for every i. Then, P_i is instantiated to $\top$. This rule means that whatever priority is set to the active unification at the top level, it does not affect the priorities of the argument-level implicit unifications.

3.3 Operational Semantics

Section 2.3 defines the entailment of formulas, denoted as $\models$. On the other hand, for conventional logical formulas F and G, we also write $F \models G$ if G is a logical consequence of F [8]. So, we will properly use $\models$ in these two meanings.

Definition A *configuration* is a triplet $\langle \mathcal{C}, \mathcal{S} \rangle{:}\mathcal{V}$ in which $\mathcal{C}$ stands for Callsand $\mathcal{S}$ stands for a *constraint store*, which is a set of active unifications Unif and priority definitions PDef. $\mathcal{V}$ is the set of variables contained in $\mathcal{S}$. An initial configuration is $\langle \mathcal{C}_0, \emptyset \rangle{:}\emptyset$, where $\mathcal{C}_0$ stands for an initial goal. Note that priority is carried into a constraint store with an active unification. This suggests that our framework for prioritizing concurrent activities is an extension of the conventional concurrent constraint (cc) framework [9].

Definition Let $\mathcal{C}$ be a set of goals and $\mathcal{S}$ a constraint store. For a goal $b \in \mathcal{C}$, a function $\pi_{\mathcal{S}} : b \to 2^{\{\top,\bot\}\cup\mathsf{PVar}}$ calculates the aggregate priority associated with the goal reduction of b, where b is either Unif, Pred or PDef. Then, b is *executable* in $\mathcal{S}$ if $\pi_{\mathcal{S}}(b) \neq \emptyset$. $\pi_{\mathcal{S}}(b)$ is computed as follows:

(Unif) b is an active unification $X \overset{P}{=} t$: $\pi_{\mathcal{S}}(b) = \{P\}$ if $\mathcal{S} \models \top \succ P$. Otherwise, $\pi_{\mathcal{S}}(b) = \emptyset$.

(Pred) b is a predicate call $p(V_1, \cdots, V_n)^P$: For every $p(W_1, \cdots, W_n)^Q$:- $G \mid B \in$ Defs, $\pi_{\mathcal{S}}(b) = \{P\} \cup \Phi(G)$ if $\mathcal{S} \models \top \succ P$ and $\mathcal{S} \models \exists \Delta G\{V_1/W_1, \cdots, V_n/W_n\}$, where $\Delta = \mathit{Var}(G\{V_1/W_1, \cdots, V_n/W_n\})$. Otherwise, $\pi_{\mathcal{S}}(b) = \emptyset$.

(PDef) b is a priority definition $(H_1, \cdots, H_n) \succ P \succ (L_1, \cdots, L_m)$: $\pi_{\mathcal{S}}(b) = \{\top\}$ if $\forall i \forall j. \mathcal{S} \models H_i \succ L_j$. Otherwise, $\pi_{\mathcal{S}}(b) = \emptyset$.

In case (Pred), it suffices that we just check the entailment of the logical formula G and ignore all priority variables $\Phi(G)$. This is guaranteed by the Active Unification rule presented below; that is, for a passive unification $X \overset{Q}{=} t$, that Q is instantiated is equivalent to that X is instantiated.

Definition Let $\mathcal{C}$ be a set of goals and $\mathcal{S}$ a constraint store. Then, a goal $m \in \mathcal{C}$ has the *maximum* priority in $\mathcal{S}$ if $m \in E = \{e \mid e \in \mathcal{C}$ is executable in $\mathcal{S}\}$ and $\forall b \in E \setminus \{m\}. \mathcal{S} \models \pi_{\mathcal{S}}(m) \succ \pi_{\mathcal{S}}(b) \vee \pi_{\mathcal{S}}(m) \not\sim \pi_{\mathcal{S}}(b)$.

In general, there are more than one goal that have the maximum priority in $\mathcal{S}$.

Program execution is represented by successive configurations $C_0, C_1, C_2, \cdots$. A transition is a binary relation on configurations, $\rightarrow \subseteq C \times C$; the transition $C_j \to C_{j+1}$ is defined by the transition rules presented in the Plotkin style in Fig. 2. A transition is made in one step and is thus atomic. In the figure, $\vec{X}$ and $\vec{Y}$ mean variable sequences of the same length.

Predicate Call	
$\langle \{p(\vec{X})^P\}, \mathcal{S} \rangle{:}\mathcal{V} \to \langle B\theta, G\theta \cup \mathcal{S} \rangle{:}\mathcal{V} \cup \mathcal{V}_{(G,B)}$	*if* $p(\vec{X})^P$ is executable in $\mathcal{S}$, the corresponding clause is $p(\vec{Y})^Q$:- $G \mid B$, $\theta = \{P/Q, \vec{X}/\vec{Y}\}$, and $\mathcal{V} \cap \mathcal{V}_{(G,B)} = \emptyset$.
Active Unification	
$\langle \{X \overset{P}{=} t\}, \mathcal{S} \rangle{:}\mathcal{V} \to \langle \emptyset, \{X \overset{P}{=} t\} \cup \mathcal{S} \rangle{:}\mathcal{V}$	*if* $X \overset{P}{=} t$ is executable in $\mathcal{S}$.
Priority Definition	
$\langle \{(H_1, \cdots, H_n) \succ P \succ (L_1, \cdots, L_m)\}, \mathcal{S} \rangle{:}\mathcal{V} \to$ $\langle \emptyset, \{(H_1, \cdots, H_n) \succ P \succ (L_1, \cdots, L_m)\} \cup \mathcal{S} \rangle{:}\mathcal{V}$	*if* $(H_1, \cdots, H_n) \succ P \succ (L_1, \cdots, L_m)$ is executable in $\mathcal{S}$.
Goal Selection	
$\dfrac{\langle \{b\}, \mathcal{S} \rangle{:}\mathcal{V} \to \langle \mathcal{C}', \mathcal{S}' \rangle{:}\mathcal{V}'}{\langle \{b\} \cup \mathcal{C}, \mathcal{S} \rangle{:}\mathcal{V} \to \langle \mathcal{C}' \cup \mathcal{C}, \mathcal{S}' \rangle{:}\mathcal{V}'}$	*if* b has the maximum priority in $\mathcal{S}$.

Figure 2: Transition Rules

It follows from the Priority Definition rule that the priority definition is executed immediately after commitment, since its aggregate priority is always $\{\top\}$. There are illegal programs in that $\mathcal{C} \neq \emptyset$ but no *if* condition in every rule in Fig. 2 is satisfied; thus, the execution is suspended perpetually.

4 Programming with Priority

This section demonstrates programming with priority in $\mathsf{O_N}$. Basically, there are two methods in CLLs to react to an asynchronous message from an external process: event loop and merge. Thus, we take them as sample programs in the following subsections. After that, it is shown that $\mathsf{O_N}$ has an independency property in terms of the priority control of process invocation and message passing. We think that this property plays an important role in the process-message paradigm.

4.1 Prioritized Event Loop

Fig. 3 is a sample program that demonstrates prioritizing clause selection in $\mathsf{O_N}$. The *loop*/2 predicate usually makes a recursive call while waiting for an express message from *interrupt*/1 which is executed at a higher priority. When a message from *interrupt*/1 arrives at *loop*/2, both the definition

$$\begin{array}{ll}
\text{:- } \top \succ P \succ \bot, \top \succ Q \succ P, & loop(Sig, D)^P \text{ :- } Sig \overset{Q}{=} [Ex|T] \mid \\
\quad interrupt(Sig)^Q, & \quad handler(Ex)^Q, \\
\quad loop(Sig, D)^P, D \overset{P}{=} init. & \quad loop(T, D)^P. \\
 & loop(Sig, D)^P \text{ :- } true \qquad\qquad\quad \mid \\
interrupt(Sig)^Q \text{ :- } exception(Ex) \mid & \quad body(D, E)^P, \\
\quad Sig \overset{Q}{=} [Ex|T], & \quad loop(Sig, E)^P. \\
\quad interrupt(T)^Q. &
\end{array}$$

Figure 3: Prioritized Event Loop

clauses of *loop*/2 become the candidates for commitment. Then, the aggregate priority of the first clause is $\{Q, P\}$, while that of the second is $\{P\}$. Since $Q \succ P$, we have $\{Q, P\} \succ \{P\}$. Thus, the first clause always precedes the second; as soon as a message from *interrupt*/1 arrives, it is processed.

4.2 Prioritized Merge

Suppose that process *merge*/3 receives messages from process *interrupt*/1 with higher priority than ones from process *routine*/1. $\mathsf{O_N}$ can implement two methods to prioritize messages from *interrupt*/1: (1) controlling the priorities set to messages and (2) controlling the priorities set to processes. In method (1), the priority of each message is made equal to that of its sender, and the process priorities are set as $interrupt \succ routine$. Here the priority of *merge*/3 does not matter. In method (2), the priority of every message is made equal, and the process priorities are set as $interrupt \succ merge \succ routine$. Method (1) can be implemented only in $\mathsf{O_N}$, while method (2) can also be implemented in KL1. Fig. 4 shows the prioritized merge program based on method (1).

$$
\begin{array}{ll}
\text{:- } \top \succ P \succ \bot, \top \succ Q \succ P, & interrupt(Sig)^Q \text{ :- the same as Fig. 3} \\
\quad \top \succ R \succ \bot, & routine(D)^P \text{ :- } true \mid \\
\quad merge(Sig, D, Z)^R, & \quad gen\_data(X)^P, \\
\quad interrupt(Sig)^Q, & \quad D \overset{P}{=} [X|Ds], \\
\quad routine(D)^P. & \quad routine(Ds)^P.
\end{array}
$$

$$
\begin{array}{l}
merge(X, Y, Z)^R \text{ :- } X \overset{Q}{=} [H|Xs] \mid Z \overset{Q}{=} [H|Zs], merge(Xs, Y, Zs)^R. \\
merge(X, Y, Z)^R \text{ :- } Y \overset{P}{=} [H|Ys] \mid Z \overset{P}{=} [H|Zs], merge(X, Ys, Zs)^R.
\end{array}
$$

Figure 4: Prioritized Merge

4.3 Contradictory Prioritization in KL1

To explain the concept of the independency of controlling process invocation and message passing, we show that KL1 does not have the independency property.

In KL1, process priority is controlled by `@priority`, and clause selection by `alternatively`. An integer is dynamically given to `@priority` as its argument. Accordingly, the priority of KL1 has the total ordering. If both the definition clauses above and below the `alternatively` pragma are ready for commitment (that is, their variables are sufficiently instantiated), one of the clauses on the upper side is always selected. As such, in KL1, the process control by priority is dynamic, while the clause selection is static. In other words, process priority is controlled by a message sender, while clause selection is controlled by a message receiver. Unfortunately, the KL1 priority system causes contradictory prioritization.

The following KL1 program is problematic.

```
:- (X = a)@priority(10),        p(X,Y) :- X = a | true.
   (Y = b)@priority(20),        alternatively.
   p(X,Y)@priority(Np).         p(X,Y) :- Y = b | true.
```

In this program, suppose that `Np`, which is given as the priority of `p(X,Y)`, is instantiated to an integer somewhere else. Then, the commitment of `p(X,Y)` *depends* on not only the position of `alternatively` but also on the value of `Np`. If `Np` > 10, the second definition clause is selected; if `Np` $= 10$, the clause selection is nondeterministic; if `Np` < 10, the first is selected. This behavior is caused by the contradiction of the dynamic relation between priorities of `X = a` and `Y = b` and the static relation between the first and second clauses of `p/2` posed by `alternatively`. It is also reported in [2] that such contradictory prioritization may appear in Ada programming.

On the other hand, $\mathsf{O_N}$ prevents a programmer from writing such a program with contradictory prioritization, since the aggregate priority is employed for clause selection and all the priorities for process invocation and message passing are dynamically given by message senders.

4.4 Independency of Priority Control

We show that process invocation is independent from the prioritization of message passing in $\mathsf{O_N}$ and vice versa. Let $\mathcal{S}$ be the constraint store after the completion of program execution, through this subsection.

Program Feature 4.1 *There are the following programs A and B. Which process, $p/2$ or $q/2$, precedes the other is determined only by the ordering between P and Q; it is independent from the message priority represented by R.*

Program A:

$:\!-\ \top \succ P \succ \bot, \top \succ R \succ \bot, P \succ Q \succ \bot,$
$\quad p(X,Y)^P, q(X,Z)^Q, X \overset{R}{=} t.$

$p(X,Y)^P :\!-\ X \overset{R}{=} t \mid Y \overset{P}{=} a.$

$q(X,Z)^Q :\!-\ X \overset{R}{=} t \mid Z \overset{Q}{=} b.$

Program A

Program B:

$:\!-\ \top \succ P \succ \bot, \top \succ R \succ \bot,$
$\quad p(X,Y)^P, q(X,Z)^P, X \overset{R}{=} t.$

The definitions of $p/2$ and $q/2$ are the same as Program A.

Program B

Proof. In Program A, $\mathcal{S} \models P \not\succ R$ and $\mathcal{S} \models Q \not\succ R$. We consider the following three cases in terms of the invocation order of $p(X,Y)^P, q(X,Z)^Q$, and $X \overset{R}{=} t$. (A1) The order $p(X,Y)^P, q(X,Z)^Q$ and $X \overset{R}{=} t$: First the invocations of $p(X,Y)^P$ and $q(X,Z)^Q$ are suspended. When $X \overset{R}{=} t$ is executed, both $p(X,Y)^P$ and $q(X,Z)^Q$ become executable. Since their aggregate priorities are $\{P,R\}$ and $\{Q,R\}$ respectively and $\{P,R\} \succ \{Q,R\}$, $p(X,Y)^P$ precedes $q(X,Z)^Q$. (A2) The order $p(X,Y)^P, X \overset{R}{=} t$ and $q(X,Z)^Q$: First the invocation of $p(X,Y)^P$ is suspended. After $X \overset{R}{=} t$ is completed, the execution of $p(X,Y)^P$ resumes. This is followed by $q(X,Z)^Q$. (A3) The order $X \overset{R}{=} t, p(X,Y)^P$ and $q(X,Z)^Q$: Since $X \overset{R}{=} t$ has been executed, $p(X,Y)^P$ immediately becomes executable. Therefore, in every case, the commitment of $p(X,Y)^P$ always precedes that of $q(X,Z)^Q$.

Next, Program B behaves as the same FGHC program with the priority annotations ignored, since the aggregate priorities are equal. Thus, whether $p/2$ precedes $q/2$ or vice versa is nondeterministic. Consequently, the execution order of $p/2$ and $q/2$ is independent of R. $\square$

Program Feature 4.2 *There are the following programs C and D. The value of variable Z is determined only by the ordering of P and Q; it is independent from the process priority represented by R.*

Program C:

$:\!-\ \top \succ P \succ \bot, \top \succ R \succ \bot, P \succ Q \succ \bot,$
$\quad X \overset{P}{=} a, Y \overset{Q}{=} b, p(X,Y,Z)^R.$

$p(X,Y,Z)^R :\!-\ X \overset{P}{=} a \mid Z \overset{P}{=} a.$

$p(X,Y,Z)^R :\!-\ Y \overset{Q}{=} b \mid Z \overset{Q}{=} b.$

Program C

Program D:

$:\!-\ \top \succ P \succ \bot, \top \succ R \succ \bot,$
$\quad X \overset{P}{=} a, Y \overset{P}{=} b, p(X,Y,Z)^R.$

The definition of $p/3$ is the same as Program C.

Program D

Proof. In Program C, $\mathcal{S} \models P \not\succ R$ and $\mathcal{S} \models Q \not\succ R$. We consider the following three cases in terms of the invocation order of $X \overset{P}{=} a, Y \overset{Q}{=} b$ and $p(X,Y,Z)^R$. (C1) The order $X \overset{P}{=} a, Y \overset{Q}{=} b$ and $p(X,Y,Z)^R$: The aggregate priorities of the two definition clauses of $p/3$ are $\{P,R\}$ and $\{Q,R\}$. Since $\{P,R\} \succ \{Q,R\}$, the first clause is always selected and $Z = a$ is obtained. (C2) The order $X \overset{P}{=} a, p(X,Y,Z)^R$ and $Y \overset{Q}{=} b$: Upon the invocation of

$p(X,Y,Z)^R$, the first clause of $p/3$ is selected and $Z = a$ is obtained. (C3) The order $p(X,Y,Z)^R, X \overset{P}{=} a$ and $Y \overset{Q}{=} b$: The invocation of $p(X,Y,Z)^R$ first suspends. After $X \overset{P}{=} a$ is completed, the execution of $p(X,Y,Z)^R$ resumes, the first clause of $p/3$ is selected and $Z = a$ is obtained. Therefore, in every case, the first clause of $p/3$ is selected and we get $Z = a$.

Next, Program D behaves as the same FGHC program with the priority annotations ignored, since the aggregate priorities are equal. Thus, which clause of $p/3$ is selected, the first or the second, is nondeterministic. Consequently, the clause selection of $p/3$ is independent of R. $\Box$

A counterpart of a phenomenon known as priority inversion may occur in O_N programs. We think it is not appropriate to amalgamate into O_N a mechanism to resolve priority inversion such as the priority-inheritance algorithm, since the design principle of O_N is to give a means of describing priority to a programmer. Actually, we successfully wrote a simple priority-inheritance algorithm in O_N.

5 Concluding Remarks

From the above development and the sample programs, we think that our framework can make priority a means for representing a programmer's intention related to execution control in declarative programming. Moreover, thinking about priority may give us another new insight into execution control and the process-message paradigm.

According to the operational semantics of O_N, every time a goal is reduced, its language processing system must compute the aggregate priorities of many goals and choose a goal with the maximum priority among them. The operational semantics employs fine-grained concurrency, and it seems quite inefficient if naively implemented. Thus, detecting specific patterns in which the overhead of the goal selection and goal scheduling can be alleviated will be inevitable. We expect that the static analysis of priority in a program will be able to detect some profitable characteristics for program execution. One is the condition that makes the computation of the goal selection lightweight. Actually, we have already discovered some prioritization patterns to be profitable. Another is the thread extraction and some other information for goal scheduling. The analysis may suggest suspension-free threads and as static goal-scheduling strategies as possible. This characteristic benefits the coarse-grained concurrency employed by almost all practical implementations of CCLs [1]. In addition, if the analysis detects unrelated threads in terms of priority, each thread may be regarded as an execution unit in a distributed environment.

There is also another interesting approach to share our basic motivation for treating prioritized concurrent activities, in which an agent of the form **if** a **else** A, representing default, is introduced to detect negative information [10]. Since we think this approach is closely related to ours, a comparison between them will be the subject of future investigations.

We are just at the first step, where the declarative semantics for priority is given and a concurrent logic language with the concept of priority is designed.

As stated in Section 1, priority restricts the set of answers as $A_P \subseteq A_O$; our research is motivated by the exploration of the semantics for priority. Therefore, future work will be to identify A_P and clarify how priority affects the language semantics.

Acknowledgements: We would like to thank Prof. Kazunori Ueda for his encouraging suggestions to the material of this paper. We thank the anonymous ICLP'99 referees who carefully read and gave valuable comments, although we unfortunately could not reflect all of them in this paper. We also thank Mr. Yasuyuki Tsukada of NTT for the improvement of the theoretical treatment of priority and the board members of the KLIC Association for the discussion about practical use. Thanks are due to Dr. Shunichi Uchida, the director of Research Institute For Advanced Information Technology (AITEC), for his environmental support.

References

[1] Chikayama, T., KLIC: A Portable Parallel Implementation of a Concurrent Logic Programming Language, *Proc. of Parallel Symbolic Languages and Systems (PSLS'95)*, also in *LNCS 1068*, 1995.

[2] Fidge, C. J., A Formal Definition of Priority in CSP, *ACM Trans. on Prog. Lang. and Sys.*, Vol.15, No.4, pp.681–705, Sep. 1993.

[3] Gregory, S., Experiments with Speculative Parallelism in Parlog, *Proc. of the 10th ISLP*, 1993.

[4] Gregory, S., and Ramirez, R., Tempo: a declarative concurrent programming language, *Proc. of the 12th ICLP*, 1995.

[5] Huntback, M., Speculative Computation and Priorities in Concurrent Logic Languages, *Proc. of the 3rd UK Conference on Logic Programming (ALPUK'91)*, 1991.

[6] Huntback, M. and Ringwood, G., Programming in Concurrent Logic Language, *IEEE Software*, pp.71–82, Nov. 1995.

[7] Kowalski, R., Algorithm = Logic + Control, *Comm. ACM* 22, 7, pp.424–436, 1979.

[8] Lloyd, J. W., *Foundations of Logic Programming*, Second, Extended Edition, Springer-Verlag, 1987.

[9] Saraswat, V. A., *Concurrent Constraint Programming*, The MIT Press, 1993.

[10] Saraswat, V., Jagadeesan, R. and Gupta, V., Timed Default Concurrent Constraint Programming, *J. of Symbolic Computation*, Vol.22, Nos. 5&6, pp.475-520, 1996.

[11] Ueda, K. and Chikayama, T., Design of the Kernel Language for the Parallel Inference Machine. *The Computer Journal*, Vol.33, No.6, pp.494–500, 1990.

[12] Ueda, K. and Morita, M., Moded Flat GHC and Its Message-Oriented Implementation Technique, *New Generation Computing*, Vol.13, No.1, pp.3–43, 1994.