# Linearity Analysis of Concurrent Logic Programs ⋆

Kazunori UEDA

Department of Information and Computer Science
Waseda University
3–4–1, Okubo, Shinjuku-ku, Tokyo 169-8555, Japan
E-mail: ueda@ueda.info.waseda.ac.jp

**Abstract.** Automatic memory management and the hiding of the notion of pointers are the prominent features of symbolic processing languages. They make programming easy and guarantee the safety of memory references. For the memory management of linked data structures, copying garbage collection is most widely used because of its simplicity and desirable properties. However, if certain properties about runtime storage allocation and the behavior of pointers can be obtaind by static analysis, a compiler may be able to generate object code closer to that of procedural programs. In the fields of parallel, distributed and real-time computation, it is highly desirable to be able to identify data structures in a program that can be managed without using garbage collection. To this end, this paper proposes a framework of linearity analysis for a concurrent logic language Moded Flat GHC, and proves its basic property. The purpose of linearity analysis is to distinguish between fragments of data structures that may be referenced by two or more pointers and those that cannot be referenced by two or more pointers. Data structures with only one reader are amenable to compile-time garbage collection or local reuse. The proposed framework of linearity analysis is constraint-based and involves both equality and implicational constraints. It has been implemented as part of klint v2, a static analyzer for KL1 programs.

## 1 Introduction

In whatever programming language, variables can be viewed as a means of communication as well as a means of storage. When viewed as a means of communication,

– assigning a value to a variable at some point in a time space amounts to sending, and
– reading the value of a variable at another point in the time space amounts to receiving.

⋆ To appear In *Proc. International Workshop on Parallel and Distributed Computing for Symbolic and Irregular Applications*, Ito, T. and Yuasa, T. (eds.), World Scientific, 2000, pp. 253–270.

A value once assigned is usually read at least once before it is altered by subsequent assignments.[1] The communication is one-to-one when the value is read exactly once, while it is one-to-many when the value is read more than once.

When the value to be communicated is non-atomic, it is usually created on a heap and a variable holds a pointer to it rather than the value itself. The 'final' reader of a non-atomic value can free the storage occupied by the value or reuse it for other purposes. In order to achieve recycling, however, the implementation must be able to judge whether each read operation is the final one on the current value of the variable. Since this is difficult in general, a heap is usually managed using runtime garbage collection.

However, suppose a compiler guarantees, by static analysis, that some variable is used only for one-to-one communication. Then the storage occupied by the value can be freed or recycled immediately after it is read. In a concurrent setting, it is usually difficult to identify the final read operation on a variable used for one-to-many communication, but good news about one-to-one communication is that a read operation is always final.

This paper is concerned with concurrent logic programming in which logical (or single-assignment) variables are used as communication channels, and proposes a theoretical framework, called *linearity analysis*, that distinguishes between one-to-one and one-to-many communcation. We are particularly interested in Moded Flat GHC, [8] a concurrent logic language with strong moding, because its mode system infers information flow of logical variables and simplifies linearity analysis.

We have found, from concurrent logic programs written so far, that most logical variables are used for one-to-one communication. [10] In particular, virtually all of the variables with complex protocols such as incomplete messages and streams of streams are one-to-one. This suggests that linearity analysis can provide fundamental information for optimizing memory management.

## 2 Concurrent Logic Languages and Linearity Analysis

GHC (Guarded Horn Clauses) is a concurrent logic language whose syntax is shown in Figure 1. For simplicity, we assume that program clauses contain no guard goals (i.e., conditions of rewriting specified between '`:-`' and '`|`'), but this restriction is not essential for the theoretical framework developed in this paper.

The operational semantics of GHC models the concurrent reduction of goals starting with an initial goal clause. Reduction of a current goal clause involves either of the following:

- rewriting of a non-unification goal to (zero or more) goals, possibly after observing a required substitution (*ask*), or
- execution of a unification goal, which may publish a substitution (*tell*).

[1] In the case of single-assignment variables, the value once assigned will not be altered forever.

$$
\begin{array}{rrll}
\text{(Program)} & P & ::= \text{set of } C & (1)\\
\text{(Program Clause)} & C & ::= A \texttt{ :- | } B & (2)\\
\text{(Body)} & B & ::= \text{multiset of } G & (3)\\
\text{(Goal)} & G & ::= T_1 \texttt{ = } T_2 \quad | \quad A & (4)\\
\text{(Non-unification Goal)} & A & ::= p\texttt{(}T_1\texttt{, \ldots ,}T_n\texttt{)}, & \\
 & & \quad p \text{ is a predicate other than `}\texttt{=}\text{'} & (5)\\
\text{(Term)} & T & ::= \text{(as in first-order logic)} & (6)\\
\text{(Goal Clause)} & Q & ::= \texttt{:- } B & (7)
\end{array}
$$

**Fig. 1.** Syntax of a subset of GHC

We review one-step reduction from a goal clause $Q$ to $Q'$. For notational simplicity, we identify a goal clause with the multiset of body goals in the goal clause.

- *Reduction of a non-unification goal* $g \in Q$ *using a clause (renamed using fresh variables) "*$h$ `:- |` $B$*"*: The synchronization rule of GHC tells that there must be a substitution $\theta$ such that $g = h\theta$, and $Q' = Q \setminus \{g\} \cup B\theta$, where '$\setminus$' and '$\cup$' are multiset difference and union, respectively.
- *Reduction of a unification goal* $(t_1 \texttt{ = } t_2) \in Q$: $Q' = (Q \setminus (t_1 \texttt{ = } t_2))\theta$, where $\theta$ is the most general unifier of $t_1$ and $t_2$. We assume that the program is well-moded, [8] in which case the unification does not fail except due to occur check.

In either case, reduction in general involves the rewriting of a variable (say $v$) to a term $t$ ($\neq v$). In the reduction of a non-unification goal, $v$ must be a variable in the (renamed) program clause, while in the reduction of a unification goal, $v$ must be a variable in the goal clause. When $v$ has more than one input occurrence (occurrence that is rewritten to $t$ as the result of reduction), or equivalently, when $v$ is used for one-to-many communication, the number of pointers from $Q'$ to $t$ is increased. (Throughout the paper, we assume that assignment of a structured value is done by sharing rather than by copying.) The purpose of linearity analysis is to statically analyze exactly *where* shared data structures may occur—in which predicates, in which arguments, and in which part of the data structures taken by those arguments.

A data structure that has not been referenced by a variable for one-to-many communication is never shared by two or more readers. A compiler can know exactly when it is read finally and becomes garbage, and generate object code that returns the structure to a free list or recycles it locally.

## 3 Terminology

**Definition.** We say that an occurrence of a variable is a *channel occurrence* if it is the leftmost occurrence in a clause head or an occurrence in a clause body.

A variable can be thought of as a communication channel for one-shot or repetitive communication (the most typical of repetitive communication is stream communication), and a channel occurrence can be thought of as an endpoint of a channel. The condition 'leftmost' is rather arbitrary; the motivation is that only one of the (possibly many) occurrences of a variable can be called a channel occurrence. The condition does not imply that the arguments in a clause head are processed from left to right.

**Definition.** A variable that has at most two channel occurrences in a program clause or a goal clause is called a *linear* variable, and a variable that possibly has three or more occurrences is called a *nonlinear* variable.

Thus it is always safe to say some variable is nonlinear, but the purpose of linearity analysis is to detect as many linear variables as possible.

**Example.** In the quicksort program shown in Figure 2, all the variables except `X` in the second clause of ternary `qsort` are *linear*.

```
:- module main.
qsort(Xs,Ys) :- true | qsort(Xs,Ys,[]).

qsort([],    Ys0,Ys ) :- true | Ys=Ys0.
qsort([X|Xs],Ys0,Ys3) :- true |
   part(X,Xs,S,L), qsort(S,Ys0,Ys1), Ys1=[X|Ys2],
   qsort(L,Ys2,Ys3).
part(_,[],    S, L ) :- true | S=[], L=[].
part(A,[X|Xs],S0,L ) :- A>=X | S0=[X|S], part(A,Xs,S,L).
part(A,[X|Xs],S, L0) :- A< X | L0=[X|L], part(A,Xs,S,L).
```

**Fig. 2.** A quicksort program

Strong moding guarantees that each variable generated during program execution has exactly one *output* occurrence, namely an occurrence that can determine its top-level value. This means that a variable with exactly two channel occurrences is used for one-to-one communication, and a variable with only one channel occurrence is used for one-to-zero communication.

**Definition.** A *path* is a sequence of pairs, of the form $\langle symbol, arg\rangle$, of function/predicate symbols and argument positions. In this paper, we regard constant symbols as nullary function symbols. Paths are used to specify occurrences of variables or function symbols in a goal or a term. Let $P_{Atom}$ be the set of all paths for specifying occurrences in goals, and $P_{Term}$ the set of all paths for specifying occurrences in terms.

For example, a function symbol `b` occurs in a goal `p(f(a,b),C)` at $\langle \mathtt{p}, 1\rangle\langle \mathtt{f}, 2\rangle \in P_{Atom}$. An empty sequence in $P_{Term}$ specifies the principal function symbol of a term in question.

## 4 Linearity Annotation

To distinguish between non-shared and shared data structures in a computational model without the notion of pointers, we consider giving a linearity annotation 1 or $\omega$ to every occurrence of a function symbol $f$ appearing in (initial or reduced) goal clauses and body goals in program clauses.[2] The annotations appear as $f^1$ or $f^\omega$ in the theoretical framework, though the purpose of linearity analysis is to reason about the annotations and compile them away so that the program can be executed without having to maintain linearity annotations at run time.

Intuitively, the principal function symbol of a structure possibly referenced by more than one pointer must have the annotation $\omega$, while a structure always pointed to by only one pointer in its lifetime can have the annotation 1. Another view of the annotation is that it models a one-bit reference counter that is not decremented once it reaches $\omega$.

The annotations must observe the following closure condition: If the principal function symbol of a term has the annotation $\omega$, all function symbols occuring in the term must have the annotation $\omega$. In contrast, a term with the principal function symbol annotated as 1 can contain a function symbol with either annotation, which means that a subterm of a non-shared term may possibly be shared.

Given linearity annotations, the operational semantics is extended to handle them so that they may remain consistent with the above intuitive meaning.

1. The annotations of function symbols in program clauses and *initial* goal clauses are given according to how the structures they represent are implemented. For instance, consider the following goal clause:

   ```
   :- p([1,2,3,4,5],X), q([1,2,3,4,5],Y).
   ```

   If the implementation chooses to create a single instance of the list `[1,2,3,4,5]` and let the two goals share them, the function symbols (there are 11 of them including `[]`) must be given $\omega$. If two instances of the list are created and given to `p` and `q`, either annotation is compatible with the implementation.
2. Suppose a substitution $\theta = \{v_1 \leftarrow t_1, \ldots, v_n \leftarrow t_n\}$ is applied upon one-step reduction from $Q$ to $Q'$.
   (a) When $v_i$ is nonlinear, the substitution instantiates more than one occurrence of $v_i$ to $t_i$ and makes $t_i$ shared. Accordingly, all data structures inside $t_i$ (i.e., the subterms of $t_i$) become shared as well. So, prior to rewriting the occurrences of $v_i$ by $t_i$, we change all the annotations of the function symbols constituting $t_i$ to $\omega$.
   (b) When $v_i$ is linear, $\theta$ does not increase the number of references to $t_i$. So we rewrite $v_i$ by $t_i$ without changing the annotations in $t_i$.

[2] The notation is after related work [4, 7] on different computational models.

## 5 Linearity Constraints

The linearity of a well-moded program can be characterized using a linearity function.

**Definition.** A *linearity function* is a function from $P_{Atom}$ to the binary codomain $\{nonshared, shared\}$.

In this paper, we write $\lambda$ to stand for a linearity function.

The motivation of a linearity function is to distinguish between those paths where function symbols with $\omega$ *can* appear and those where function symbols with $\omega$ *cannot* appear. Suppose we can prove that a function symbol with $\omega$ cannot appear at $p$ such that $\lambda(p) = nonshared$. Then the (sole) reader of the data structure at a *nonshared* path can safely discard the top-level structure after accessing its elements. (There is one subtle point in this optimization, which will be discussed in Section 7.)

The above property can be established by enforcing *linearity constraints* on the function $\lambda$. Linearity constraints imposed by each program clause $h$ `:-|` $B$ or a goal clause `:-` $B$ are shown in Figure 3. The linearity constraints refer to the mode of a program represented by a function $m$. The mode constraints [8] on a well-moding $m$ are given in Figure 4. Here, a *submode* $m/p$ is defined as a function satisfying $(m/p)(q) = m(pq)$. The function $m/p$ represents the part of $m$ viewed at the path $p$. The functions *IN* and *OUT* are constant functions that always return *in* and *out*, respectively. An overline '$\overline{\ \ }$' inverts the polarity of a mode, a submode, or a mode value. We omit the motivations of each mode constraints and properties they enjoy. [8]

(BF$_\lambda$) If a function symbol $f^\omega$ occurs at the path $p$ in $B$, then $\lambda(p) = shared$.

(LV$_\lambda$) If a linear variable occurs both at $p_1$ and $p_2$, then
$\forall q \in P_{Term}(m(p_1q) = in \wedge \lambda(p_1q) = shared \Rightarrow \lambda(p_2q) = shared)$ (if $p_1$ is a head path);
$\forall q \in P_{Term}(m(p_1q) = out \wedge \lambda(p_1q) = shared \Rightarrow \lambda(p_2q) = shared)$ (if $p_1$ is a body path).

(NV$_\lambda$) If a nonlinear variable occurs at $p$, then
$\forall q \in P_{Term}(m(pq) = out \Rightarrow \lambda(pq) = shared)$ (if $p$ is a head path);
$\forall q \in P_{Term}(m(pq) = in \Rightarrow \lambda(pq) = shared)$ (if $p$ is a body path).

(BU$_\lambda$) For a unification body goal $=_k$,
$\forall q \in P_{Term}(\lambda(\langle =_k, 1\rangle q) = \lambda(\langle =_k, 2\rangle q))$.

**Fig. 3.** Linearity constraints imposed by a clause $h$ `:-|` $B$

To allow different unification goals to have different modes and/or linearities, which is a limited form of polymorphism, each unification goal in program clauses and an initial goal clause is given a unique serial number. In this paper, goals

(HF) If a function symbol occurs at $p$ in $h$, then $m(p) = in$.
(HV) If a variable occurring $p$ in $h$ occurs elsewhere in $h$, then $m/p = IN$.
(BU) For a unification body goal $=_k$, $m/\langle =_k, 1\rangle = \overline{m/\langle =_k, 2\rangle}$.
(BF) If a function symbol occurs at $p$ in $B$, then $m(p) = in$.
(BV) Let a variable $v$ occur $n\,(\geq 1)$ times in $h$ and $B$ at $p_1, \ldots, p_n$, of which the occurrences in $h$ are at $p_1, \ldots, p_k$ $(k \geq 0)$. Then

$$\begin{cases} \mathcal{R}\big(\{m/p_1, \ldots, m/p_n\}\big), & k = 0; \\ \mathcal{R}\big(\{\overline{m/p_1}, m/p_{k+1}, \ldots, m/p_n\}\big), & k > 0; \end{cases}$$

where $\mathcal{R}(S)$ is a 'cooperativeness' relation which states that, for all paths $q$, $\exists s \in S\big(s(q) = out \wedge \forall s' \in S\backslash\{s\}\ \big(s'(q) = in\big)\big)$ holds.

**Fig. 4.** Mode constraints imposed by a clause $h$ :- | $B$

other than unification are assumed to be monomorphic; that is, different goals with the same predicate symbol have the same modes and linearities. This is for the sake of simplicity and it is possible to incorporate mode polymorphism [2] and in the same way linearity polymorphism.

The function $\lambda$ satisfying the linearity constraints is computed statically using program clauses and an initial goal clause. Linearity constraints in Figure 3 are trivially satisfied by letting $\lambda(p) = shared$ for all $p$. However, the purpose of linearity analysis is to compute the 'smallest' $\lambda$ satisfying linearity constraints, where the partial ordering is defined as

$$\lambda_1 \leq \lambda_2 \Leftrightarrow \forall p \in P_{Atom}(\lambda_1(p) = shared \Rightarrow \lambda_2(p) = shared).$$

How to solve linearity constraints to compute the smallest $\lambda$ will be discussed in Section 8.

## 6 Subject Reduction Theorem

This section gives a fundamental property that a linearity function enjoys.

**Definition.** Let $v$ be a variable and $t$ a term. We say that the *extended occur check* for unification between $v$ and $t$ *fails* if $t$ is $v$ or $t$ contains $v$.

**Theorem 1 (subject reduction).** Suppose $\lambda$ satisfies the linearity constraints of a program $P$ and a goal clause $Q$, and $Q$ is reduced in one step to $Q'$, where the reduced goal $g \in Q$ is not a unification goal for which extended occur check fails. Then $\lambda$ satisfies the linearity constraints of $Q'$ as well.

**Proof.** Based on extensive case analysis. The cases can be divided into two based on whether the goal reduced is a non-unification goal or unification.

[Case 1] The reduction has rewritten a non-unification goal $g$ using a (renamed) clause $C \in P$ of the form "$h$ :-| $B$".

What we must consider are the constraints ($\mathrm{LV}_\lambda$) and ($\mathrm{NV}_\lambda$) imposed by the variables occurring in $g \in Q$ and the constraints ($\mathrm{BF}_\lambda$) imposed by the occurrences of function symbols brought into $B\theta$ ($\subseteq Q'$) by $\theta$ (these occurrences originate from the occurrences of the function symbols in $g$). Constraints imposed by the other variables in $Q'$ and those imposed by other occurrences of functions, which were already in either $Q \setminus \{g\}$ or $B$, need not be considered because they are exactly the same as those in $Q$. This means it suffices to consider all the symbols in $g$.

1. A function symbol $f^\kappa$ ($\kappa \in \{1, \omega\}$) occurs at the path $p$ in $g$. Then, either
   - the function $f$ occurs at $p$ in $h$, or
   - there exist $p' \in P_{Atom}$ and $q \in P_{Term}$ such that $p = p'q$ and a variable (say $v$) occurs at $p'$ in $h$.

   In the former case, ($\mathrm{BF}_\lambda$) is not applicable because the occurrence disappears upon reduction. So it suffices to consider the latter case, which may introduce new occurrences of $f$ into $B\theta$ ($\subseteq Q'$). Suppose $v$ occurs $n$ ($\geq 0$) times in $B$ at $r_1, \ldots, r_n$, and let $g_j$ be the goal to which the $j$th occurrence belongs. Then $f^{\kappa'}$ occurs in the goal $g_i\theta$ at $r_i q$, for $i = 1 \ldots, n$, where $\kappa' = \kappa$ if $n \leq 1$ and $\kappa' = \omega$ if $n > 1$ by Rule 2a in Section 4. We must show that $Q'$ enjoys ($\mathrm{BF}_\lambda$).
   (a) When $n \leq 1$ ($v$ is linear): It suffices to consider the case $n = 1$. If $\kappa = \omega$, $\lambda(r_i q) = \mathit{shared}$ must hold, but this can be derived as follows:
      i. $\lambda(p) = \mathit{shared}$, by ($\mathrm{BF}_\lambda$) applied to $f^\omega$ in $Q$,
      ii. $m(p) = \mathit{in}$, by (BF) applied to $f^\omega$ in $Q$,
      iii. $\lambda(p) = \mathit{shared} \Rightarrow \lambda(r_i q) = \mathit{shared}$, by 1(a)ii and ($\mathrm{LV}_\lambda$) applied to $v$ in $C$,
      iv. $\lambda(r_i q) = \mathit{shared}$, by 1(a)i and 1(a)iii.
   (b) When $n > 1$ ($v$ is nonlinear): Since $\kappa' = \omega$, we must show that $\lambda(r_i q) = \mathit{shared}$ holds.
      i. $m(p) = \mathit{in}$, by (BF) applied to $f^\kappa$ in $Q$,
      ii. $m(r_i q) = \mathit{in}$, by 1(b)i and (BV) applied to $v$ in $C$,
      iii. $\lambda(r_i q) = \mathit{shared}$, by 1(b)ii and ($\mathrm{NV}_\lambda$) applied to $v$ in $C$.
2. A variable $w$ occurs at $p$ in $g$. Suppose $w$ occurs $l$ ($\geq 1$) times in $g$ at $p_1 (= p)$, $p_2, \ldots, p_l$ and $m$ ($\geq 0$) times in $Q \setminus \{g\}$ at $p_{l+1}, \ldots, p_{l+m}$.

   Because there exists $\theta$ such that $g = h\theta$, for each $p_i$ ($1 \leq i \leq l$), there exists a prefix $p'_i \in P_{Atom}$ of $p_i$ such that a variable (say $v_i$) occurs at $p'_i$ in $h$, and a path $q_i \in P_{Term}$ such that $p_i = p'_i q_i$. Suppose $v_i$ occurs $n_i$ ($\geq 0$) times in $B$ at $r_{i1}, \ldots, r_{in_i}$. Then $w$ is made to occur at $r_{ij} q_i$ ($1 \leq i \leq l$, $1 \leq j \leq n_i$) in $B\theta$.[3]

   When some $v_i$ is nonlinear, $w$ in $Q'$ becomes nonlinear as well. We consider those paths where $w$ occurs, namely

[3] The variable $w$ may occur less than $n_1 + \cdots + n_l$ times because the two occurrences of $w$ in $g$ may be received by different occurrences of the same variable in $h$ and brought to $B\theta$ not independently.

– $r_{ij}q_i$ ($1 \le i \le l$, $1 \le j \le n_i$) (brought by $\theta$) and
– $p_{l+1}, \ldots, p_{l+m}$ (inherited from $Q\backslash\{g\}$).

(a) For the occurrences brought by $\theta$,
   i. $\forall q \in P_{Term}(m(r_{ij}q) = in \Rightarrow \lambda(r_{ij}q) = shared)$, by (NV$_\lambda$) applied to $v_i$ in $C$,
   ii. $\forall q \in P_{Term}(m(r_{ij}q_iq) = in \Rightarrow \lambda(r_{ij}q_iq) = shared)$, by 2(a)i,

   so (NV$_\lambda$) is satisfied for the paths of $w$ brought by $\theta$.

(b) For the occurrences inherited from $Q\backslash\{g\}$, if $w$ is nonlinear in $Q$, (NV$_\lambda$) applied to $Q'$ is immediate from (NV$_\lambda$) applied to $Q$. If $w$ is linear in $Q$, we have $m \le 1$ and now it suffices to consider the case where $m = 1$, namely the case where $w$ occurs at $p(= p_1)$ in $g$ and $p_2$ elsewhere. The goal is to show $\forall q \in P_{Term}(m(p_2q) = in \Rightarrow \lambda(p_2q) = shared)$, so we first assume $m(p_2q) = in$ for some $q$. Then
   i. $m(pq) = out$, by assumption and (BV) applied to $w$ in $Q$,
   ii. $\lambda(pq) = shared$, by 2(b)i and (NV$_\lambda$) applied to $v_i$ in $C$,
   iii. $\lambda(p_2q) = shared$, by 2(b)i, 2(b)ii and (LV$_\lambda$) applied to $w$ in $Q$.

   So (NV$_\lambda$) is satisfied for the occurrences of $w$ inherited from $Q\backslash\{g\}$.

When all the $v_i$'s are linear, $w$ is linear in $Q'$ if it is linear in $Q$, and nonlinear in $Q'$ otherwise. For each case, the linearity constraints to be satisfied by $Q'$ can be shown to hold with similar arguments.

[Case 2] The reduction has executed a unification goal $t_1 =_k t_2$. By the assumption of well-modedness, there exists an $i$ such that $m(\langle =_k, i\rangle) = out$. Without loss of generality, we can assume $i = 1$, in which case unification degenerates to assignment to the *left-hand side* variable. By the assumption of extended occur check, $t_2$ is not identical to the variable $t_1$ or a term containing $t_1$. So $Q'$ is equal to $\big(Q \setminus \{t_1 =_k t_2\}\big)\{t_1 \leftarrow t_2\}$. Let the variable $t_1$ occur $n$ ($\ge 0$) times in $Q \setminus \{t_1 =_k t_2\}$ at $r_1, \ldots, r_n$. Then each symbol in $t_2$ is duplicated $n$ times and occurs in $Q'$. It suffices to show that these occurrences enjoy the linearity constraints (BF$_\lambda$), (LV$_\lambda$), and (NV$_\lambda$).

1. A function symbol $f^\kappa$ occurs at $\langle =_k, 2\rangle q$. The constraint (BF$_\lambda$) tells that it suffices to consider the case $\kappa = \omega$.
   (a) $\lambda(\langle =_k, 2\rangle q) = shared$, by (BF$_\lambda$) applied to $f^\omega$ in $Q$,
   (b) $\lambda(\langle =_k, 1\rangle q) = shared$, by 1a and (BU$_\lambda$) applied to $Q$,
   (c) $m(\langle =_k, 2\rangle q) = in$, by (BF) applied to $f^\omega$ in $Q$,
   (d) $m(\langle =_k, 1\rangle q) = out$, by 1c and (BU) applied to $Q$,
   (e) $m(r_iq) = in$ ($1 \le i \le n$), by 1d and (BV) applied to $t_1$ in $Q$,
   (f) when $t_1$ is linear, $\lambda(r_1q) = shared$, by 1b, 1d and (LV$_\lambda$) applied to $t_1$ in $Q$,
   (g) when $t_1$ is nonlinear, $\lambda(r_iq) = shared$ ($1 \le i \le n$), by 1e and (NV$_\lambda$) applied to $t_1$ in $Q$.

   By executing unification $t_1 =_k t_2$, $f^\omega$ is made to occur newly at $r_1q, \ldots, r_nq$ in $Q'$. However, as shown above, $\lambda$ satisfies (BF$_\lambda$) imposed by those new occurrences.

2. A variable $w$ $(\neq t_1)$ occurs at $\langle =_k, 2\rangle q$. Suppose $w$ occurs $l$ $(\geq 1)$ times in the goal $t_1 =_k t_2$ at $\langle =_k, 2\rangle q_1$, $\langle =_k, 2\rangle q_2$, ..., $\langle =_k, 2\rangle q_l$ and $m$ $(\geq 0)$ times in $Q \setminus \{t_1 =_k t_2\}$ at $p_{l+1}$, ..., $p_{l+m}$. By executing $t_1 =_k t_2$, $w$ is made to occur newly at $r_1 q_i$, ..., $r_n q_i$ $(1 \leq i \leq l)$. So it suffices to examine the linearity constraints of these paths.
When $t_1$ is nonlinear in $Q$, $w$ in $Q'$ becomes nonlinear as well. However, by (NV$_\lambda$) applied to $t_1$ in $Q$, $\forall s \in P_{Term}(m(r_j s) = \mathit{in} \Rightarrow \lambda(r_j s) = \mathit{shared})$ holds for $1 \leq j \leq n$, which implies (NV$_\lambda$) applied to the new occurrences of $w$ in $Q'$.
When $t_1$ and $w$ are both linear in $Q$, $w$ remains linear in $Q'$. We consider the less obvious case of $l = 2$ and $m = 0$, namely the case where the other occurrence of $w$ in $Q$ is also in $t_2$. (The other case where $l = 1$ and $m = 1$ is easier and thus omitted.) The goal is to show $\forall q \in P_{Term}(m(r_i q_i q) = \mathit{out} \wedge \lambda(r_i q_i q) = \mathit{shared} \Rightarrow \lambda(r_{3-i} q_{3-i} q) = \mathit{shared})$, for $i = 1, 2$. Without loss of generality we can focus on the case $i = 1$, so we first assume $m(r_1 q_1 q) = \mathit{out}$ and $\lambda(r_1 q_1 q) = \mathit{shared}$ for some $q$. Then
   (a) $\lambda(\langle =_k, 1\rangle q_1 q) = \mathit{shared}$, by (LV$_\lambda$) applied to $t_1$ in $Q$,
   (b) $\lambda(\langle =_k, 2\rangle q_1 q) = \mathit{shared}$, by 2a and (BU$_\lambda$),
   (c) $m(\langle =_k, 1\rangle q_i q) \neq m(r_i q_i q)$, by (BV) applied to $t_1$,
   (d) $m(\langle =_k, 1\rangle q_1 q) = \mathit{in}$, by the assumption and 2c,
   (e) $m(\langle =_k, 2\rangle q_1 q) = \mathit{out}$, by 2d and (BU),
   (f) $\lambda(\langle =_k, 2\rangle q_2 q) = \mathit{shared}$, by 2b, 2e and (LV$_\lambda$) applied to $w$ in $Q$,
   (g) $\lambda(\langle =_k, 1\rangle q_2 q) = \mathit{shared}$, by 2f and (BU$_\lambda$),
   (h) $m(\langle =_k, 2\rangle q_2 q) = \mathit{in}$, by 2e and (BV) applied to $w$ in $Q$,
   (i) $m(\langle =_k, 1\rangle q_2 q) = \mathit{out}$, by 2h and (BU),
   (j) $\lambda(r_2 q_2 q) = \mathit{shared}$, by 2g, 2i, and (LV$_\lambda$) applied to $t_1$ in $Q$.

   When $t_1$ is linear in $Q$ and $w$ is nonlinear in $Q$, $w$ is in general nonlinear in $Q'$. In this case also, (NV$_\lambda$) imposed by $w$ in $Q'$ can be derived in a similar manner from the linearity constraints of $Q$. Q.E.D.

Thus we have established that data structures occurring at a *nonshared* path in a goal in the course of computation are never shared.

## 7 Applications of Linearity Analysis

Linearity analysis provides fundamental information for the optimization of memory management that can potentially lead to novel applications of concurrent logic languages.

1. *Local reuse of data structures.* The sole reader of a data structure can recycle the structure it has read—for instance to create a new data structure. This enables update-in-place of data structures in a language without the notion of destructive assignments. Features like Lisp's `nconc` and `rplacd` need not be exposed to programmers any more.

Local reuse may not necessarily have an impact on the performance of list processing on a single-processor machine, but it is essential in array processing in single-assignment languages. Despite their importance, arrays tend to be ignored in declarative languages. Since copying an array in each 'update' operation would be prohibitive, multi-version structures were often adopted as reasonable implementation of mutable arrays. However, if array variables are guaranteed to be linear, the implementation need not bother to create multi-version structures. Thus static linearity analysis seems essential to make declarative languages competitive with procedural languages in terms of performance. Linearity analysis enables not only update-in-place but also in-place splitting and merging of arrays. This opens up the possibility of *parallel* updating of a single array allocated on shared memory. [9]

However, for concurrent logic programs, linearity analysis alone is not always sufficient for the local reuse of data structures due to the flexibility of logical variables. It *is* sufficient for the optimization of numeric or character arrays in which only instantiated data can be stored. When a data structure is allowed to contain uninstantiated logical variables and the writers of uninstantiated variables point directly to the (empty) slots of the data structure, the structure cannot be recycled until all the empty slots are filled and read. To enable local reuse in the presence of partially instantiated data structures, analysis of instantiation states should be used together with linearity analysis.

2. *Distributed implementation.* In distributed applications in which pointers across sites can be limited to pointers to non-shared data, global garbage collection becomes unnecessary and the management of global pointers such as exporting and importing [5] can be greatly simplified. This opens up the possibility of using declarative languages in network programming applications in which program analysis and verification is still extremely difficult.
3. *Real-time and embedded applications.* In applications such as robot control, in which (soft) real-time processing is essential, an alternative to stop-and-copy garbage collection must be employed. A number of incremental and concurrent garbage collection algorithms have been proposed, [3] but compile-time garbage collection, where applicable, seems to be the most desirable solution to the problem. Linearity analysis is expected to play an important role in resource analysis as well—particularly the analysis of the amount of storage needed to execute a program. We believe that declarative programming with resource analysis will be a realistic tool for embedded and hard real-time applications.

## 8 Implementation—klint v2

A static analyzer for KL1 programs called *klint v2* [11] features both mode and linearity analyses. This section outlines the implementation of *klint v2*.

Basically, mode and linearity analyses are constraint satisfaction problems that can be solved using very similar techniques. In *klint v2*, a set of mode constraints is represented using a feature graph called a mode graph, [8] and solving a set of mode constraints means to merge (small feature graphs representing) new constraints into the 'current' mode graph, which is done mostly as unification over feature graphs. Non-binary constraints, which cannot be solved by unification, are imposed only by non-linear variables, and all the other constraints can be merged into the current mode graph within almost linear time with respect to the size of the mode graph. [8] For non-binary constraints, *klint v2* first postpones them in the hope that they become unary or binary by the information from other constraints. It turns out that many non-binary constraints are simplified finally.

When some constraints remain non-binary after solving all unary or binary constraints, *klint v2* assumes that nonlinear variables involved have simple, one-way dataflow rather than bidirectional dataflow such as in message streams with reply boxes. Thus, if a nonlinear variable occurs at $p$ and $m(p)$ is known to be *in* or *out*, *klint v2* imposes a stronger constraint $m/p = \mathit{IN}$ or $m/p = \mathit{OUT}$, respectively. This means that a mode graph computed by *klint v2* is not always most general, but the strengthening of constraints reduces most non-binary constraints to unary ones. Our observation is that virtually all nonlinear variables have been used for one-way communication and the strengthening causes no problem in practice.

Following mode analysis, *klint v2* creates another feature graph called a *linearity graph*. Given the result of mode analysis, (BF$_\lambda$) and (NV$_\lambda$) are unary and (BU$_\lambda$) is binary. However, (LV$_\lambda$) is still a implicational constraint of the form $\lambda(p_1 q) = \mathit{shared} \Rightarrow \lambda(p_2 q) = \mathit{shared}$. Since most variables in a program are linear, it is unrealistic to implement an implicational constraint using delaying.

If the implication can be strengthened to a bidirectional one as in

– (LV$'_\lambda$) $\forall q \in P_{\mathit{Term}}(\lambda(p_1 q) = \mathit{shared} \Leftrightarrow \lambda(p_2 q) = \mathit{shared})$,

the constraint can be solved using unification. Obviously (LV$'_\lambda$) $\Rightarrow$ (LV$_\lambda$) holds, and this approximation works well in detecting linear paths for most programs. However, consider a numeric array used as a shared look-up table. Such an array may well remain non-shared during initialization and then becomes shared. The change of the sharing property in the lifetime of a data structure is appropriately handled by (LV$_\lambda$) using one-way constraint propagation, but with the approximated version (LV$'_\lambda$), the structure is regarded as shared since its creation. This is undesirable because the initialization phase may very well want to exploit the efficiency of update-in-place.

*klint v2* circumvents this problem as follows. Since the data structures whose sharing property changes in their lifetime have simple dataflow (i.e., no bidirectional communication), we employ the full version (LV$_\lambda$) only when $m/p_1$ and $m/p_2$ are known to be $\mathit{IN}$ or $\mathit{OUT}$, and the approximate version (LV$'_\lambda$) otherwise. Suppose $p_1$ is a head path and $m/p_1$ is known to be $\mathit{IN}$. Then the first constraint of (LV$_\lambda$) is simplified to

$$\forall q, r \in P_{\mathit{Term}}(\lambda(p_1 q) = \mathit{shared} \Rightarrow \lambda(p_2 q r) = \mathit{shared}).$$

It turns out that this constraint is easy to implement using the notion of a propagator; that is, when $\lambda(p_1 q)$ is constrained to *shared*, it is propagated to the graph node representing $p_2$. A propagator is simply a graph edge (from $p_1$ to $p_2$) representing a 'null' feature, in contrast with other edges that represent $\langle symbol, arg\rangle$ features. A propagator is an essential tool for the eager evaluation of the constraint.

A graph node marked *shared* must express the closure condition $\forall p{\in}P_{Atom} \forall q{\in} P_{Term}(\lambda(p) = shared \Rightarrow \lambda(pq) = shared)$, but this can be represented in much the same way as the representation of constant submode functions $IN$ and $OUT$.

As an example, we show the result of linearity analysis of the quicksort program shown in Figure 2 (Section 3).

```
*** Linearity Graph ***
node(0): (unconstrained)
 <(main:qsort)/2,1> ---> node(24)
 <(main:qsort)/2,2> ---> node(16)
 <(main:qsort)/3,1> ---> node(24)
 <(main:qsort)/3,2> ---> node(16)
 <(main:qsort)/3,3> ---> node(16)
 <(main:part)/4,1> ---> SHARED
 <(main:part)/4,2> ---> node(24)
 <(main:part)/4,3> ---> node(24)
 <(main:part)/4,4> ---> node(24)
node(24): (unconstrained)
 <cons,2> ---> node(24)
node(16): (unconstrained)
 <cons,1> ---> SHARED
 <cons,2> ---> node(16)
```

This is a textual representation of the linearity graph of quicksort. The paths indicated `SHARED`, namely

1. the first argument of `part`,
2. the elements of the list at the second argument of binary `qsort`, and
3. the elements of the lists at the second and the third arguments of ternary `qsort`

become shared no matter whether the input list from the first argument of binary `qsort` is non-shared or shared. However, all these paths are known to have scalar (integer) values by type analysis subsequently performed by *klint v2*. On the other hand, the list skeletons returned by the quicksort program is guaranteed to be non-shared.

## 9 Related Work

Study of the memory management of concurrent logic languages has a long history. A method that uses a one-bit reference counter called MRB (multiple reference bit) for each pointer was designed for Flat GHC [1] and adopted in a KL1 implementation on a Parallel Inference Machine. [5] Roughly speaking, linearity analysis proposed in this paper tries to compile away MRBs and related operations by analyzing the value of MRBs statically.

Janus [6] establishes the linearity property by allowing each variable to occur only twice. Our technique allows both linear and nonlinear variables and distinguishes between them by static analysis.

Various techniques for the distributed implementation of concurrent logic languages were proposed, [5] including import and export tables of pointers and weighted export counting. We are not claiming that all these techniques become unnecessary, but the management of data structures guaranteed to be non-shared by linearity analysis is greatly simplified.

Kobayashi proposes a type system with linearity information for the $\pi$-calculus. [4] In functional programming, Turner et al. introduce linearity annotation to the type system. [7] All these pieces of work could be considered the application of ideas with similar motivations to different computational models. In functional programming, the difficulty lies in the variety of evaluation rules and higher-order functions, while in concurrent logic programming, the difficulty lies in the treatment of arbitrarily complex information flow expressed by logical variables. Note that the mode and the linearity systems of Moded Flat GHC are essentially type systems in a broad sense.

## 10 Conclusions and Future Work

We have proposed a framework of linearity analysis for the concurrent logic language Moded Flat GHC and studied its fundamental property. Linearity analysis can be used with mode and type analyses to generate object code closer to that of procedural programs. Also, it opens up the possibility of writing distributed, embedded, and real-time software in a very simple concurrent programming language such as Moded Flat GHC and compiling them into safe and efficient code with systematic static analysis. Our future plan is to apply concurrent logic languages to the above areas where compilation into efficient code requires serious physical considerations.

## Acknowledgments

The author is indebted to Masahiro Yasugi for their comments on earlier versions of this paper. This work is partially supported by Grant-In-Aid ((A)(1)09245101 and (C)(2)11680370) for Scientific Research, Ministry of Education, and Waseda University Grant (98A-575) for Special Research Projects.

## References

1. Chikayama, T. and Kimura, Y., Multiple Reference Management in Flat GHC. In *Logic Programming: Proc. of the Fourth Int. Conf (ICLP'87)*, The MIT Press, 1987, pp. 276–293.

2. Cho, K. and Ueda, K., Diagnosing Non-Well-Moded Concurrent Logic Programs. In *Proc. 1996 Joint Int. Conf. and Symp. on Logic Programming (JICSLP'96)*, The MIT Press, 1996, pp. 215–229.
3. Jones, R. and Lins, R., *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley & Sons, Chichester, England, 1996.
4. Kobayashi, N., Pierce, B. C. and Turner, D. N., Linearity and the Pi-calculus. In *Proc. 23rd ACM SIGACT-SIGPLAN Symp. on Principles of Programming Languages (POPL'96)*, ACM, 1996, pp. 358–371.
5. Nakajima, K., Inamura, U., Ichiyoshi, N., Rokusawa, K. and Chikayama, T., Distributed Implementation of KL1 on the Multi-PSI/V2. In *Proc. Sixth Int. Conf. on Logic Programming*, The MIT Press, 1989, pp. 436–451.
6. Saraswat, V. A., Kahn, K. and Levy, J., Janus: A Step Towards Distributed Constraint Programming. In *Proc. 1990 North American Conference on Logic Programming*, Debray, S. and Hermenegildo, M. (eds.), The MIT Press, 1990, pp. 431–446.
7. Turner, D. N., Wadler, P. and Mossin, C., Once Upon a Type. In *Proc. Seventh Int. Conf. on Functional Programming Languages and Computer Architecture (FPCA'95)*, ACM, 1995, pp. 1–11.
8. Ueda, K. and Morita, M., Moded Flat GHC and Its Message-Oriented Implementation Technique. *New Generation Computing*, Vol. 13, No. 1 (1994), pp. 3–43.
9. Ueda, K., Moded Flat GHC for Data-Parallel Programming. In *Proc. FGCS'94 Workshop on Parallel Logic Programming*, ICOT, Tokyo, 1994, pp. 27–35.
10. Ueda, K., Experiences with Strong Moding in Concurrent Logic/Constraint Programming. In *Proc. Int. Workshop on Parallel Symbolic Languages and Systems*, LNCS 1068, Springer, 1996, pp. 134–153.
11. Ueda, K., *klint* — Static Analyzer for KL1 Programs. Available from `http://www.icot.or.jp/ARCHIVE/Museum/FUNDING/funding-98-E.html`, 1998.