Resource-Passing Concurrent Programming*

Kazunori UEDA

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

Abstract. The use of types to deal with access capabilities of program entities is becoming increasingly popular.
In concurrent logic programming, the first attempt was made in Moded Flat GHC in 1990, which gave polarity structures (modes) to every variable occurrence and every predicate argument. Strong moding turned out to play fundamental rôles in programming, implementation and the in-depth understanding of constraint-based concurrent computation.
The moding principle guarantees that each variable is written only once and encourages capability-conscious programming. Furthermore, it gives less generic modes to programs that discard or duplicate data, thus providing the view of "data as resources." A simple linearity system built upon the mode system distinguishes variables read only once from those read possibly many times, enabling compile-time garbage collection. Compared to linear types studied in other programming paradigms, the primary issue in constraint-based concurrency has been to deal with logical variables and highly non-strict data structures they induce.
In this paper, we put our resource-consciousness one step forward and consider a class of 'ecological' programs which recycle or return all the resources given to them while allowing concurrent reading of data structures via controlled aliasing. This completely recyclic subset enforces us to think more about resources, but the resulting programs enjoy high symmetry which we believe has more than aesthetic implications to our programming practice in general.
The type system supporting recyclic concurrent programming gives a $[-1, +1]$ capability to each occurrence of variable and function symbols (constructors), where positive/negative values mean read/write capabilities, respectively, and fractions mean non-exclusive read/write paths. The capabilities are intended to be statically checked or reconstructed so that one can tell the polarity and exclusiveness of each piece of information handled by concurrent processes. The capability type system refines and integrates the mode system and the linearity system for Moded Flat GHC. Its arithmetic formulation contributes to the simplicity.
The execution of a recyclic program proceeds so that every variable has zero-sum capability and the resources (i.e., constructors weighted by their capabilities) a process absorbs match the resources it emits. Constructors accessed by a process with an exclusive read capability can be reused for other purposes.
The first half of this paper is devoted to a tutorial introduction to constraint-based concurrency in the hope that it will encourage cross-fertilization of different concurrency formalisms.

1 Introduction – Constraint-Based Concurrency

The *raison d'être* and the challenge of symbolic languages are to construct highly sophisticated software which would be too complicated or unmanageable if written in other languages.

Concurrent logic programming was born in early 1980's from the process interpretation of logic programs [47] and forms one of many interesting subfields addressed by the logic programming paradigm [45].

The prominent feature of concurrent logic programming is that it exploits the power of logical, single-assignment variables and data structures – exactly those of first-order logic – to achieve various forms of communication.

Essentially, a logical variable is a communication channel that can be used for output at most once (hence single-assignment) and for non-destructive input zero or more times. The two well-established operations, unification and matching (also called one-way unification), are used for output and

* The original version of this paper appeared in *Proc. Fourth Int. Symp. on Theoretical Aspects of Computer Software*, Kobayashi, N. and Pierce, B. (Eds.), Lecture Notes in Computer Science 2215, Springer, 2001, pp.95–126.

input. Thanks to the single-assignment property, the set of all unification operations that have been performed determines the current binding environment of the universe, which is called a (*monotonic*) *store* (of equality constraints) in concurrent constraint programming (CCP) [28] that generalizes concurrent logic programming. The store records what messages have been sent to what channels and what channels has been fused together.

In CCP, variable bindings are generalized to constraints, unification is generalized to the *tell* of a constraint to the store, and matching is generalized to the *ask* of a constraint from the store. The *ask* operation checks if the current store logically entails certain information on a variable.

Constraint-based communication embodied by concurrent logic programming languages has the following characteristics:

1. *Asynchronous.* In most concurrent logic languages, *tell* is an independent process that does not occur as a prefix of another process as in *ask*. This form of *tell* is sometimes called *eventual tell* and is a standard mechanism of information sending. (We do not discuss the other, prefixed form, *atomic tell*, in this paper.) Since *eventual tell* simply adds a new constraint to the store, the store can become inconsistent when an attempt is made to equate a variable to two different values. This can be avoided by using a non-standard type system, called a *mode system* [41], that controls the number of write capabilities of each variable in the system. The advocation of *eventual tell* apparently motivated Honda and Tokoro's asynchronous π-calculus [15].
2. *Polyadic.* Concurrent logic programming incorporated (rather than devised) well-understood built-in data structuring mechanisms and operations. Messages can be polyadic at no extra cost on the formalism; it does not bother us to encode numbers, tuples, lists, and so on, from scratch. The single-assignment property of logical variables does not allow one to use it for repetitive communication, but streams – which are just another name of lists in our setting – can readily be used for representing a sequence of messages incrementally sent from a process to another.
3. *Mobile.* A process[1] (say P) can dynamically create another process (say P') and a fresh logical variable (say v) with which to communicate with P'. Although process themselves are not first-class, logical variables are first-class and its occurrences (other than the one 'plugged' to P') can be freely passed from P to other processes using another channel. The logical variable connected to a process acts as an object identity (or more precisely, channel identity because a process can respond to more than one channel) and the language construct does not allow a third process to forge the identity.

   When P creates two occurrences of the variable v and sends one of them to another process (say Q), v becomes a private channel between P' and Q that cannot be monitored by any other process unless P' or Q passes it to somebody else. This corresponds to scope extrusion in the π-calculus.

   Another form of reconfiguration happens when a process fuses two logical variables connected to the outside. The fusion makes sense when one of the variables can be read (input capability) and the other can be written (output capability), in which case the effect of fusing is implicit delegation of messages. Again, the process that fused two logical variables loses access to them unless it retains a copy of them. As will be discussed later, our type systems have dealt with read/write capabilities of logical variables and the number of access paths (occurrences) of each variable.

   Although not as widely recognized as it used to be, Concurrent Prolog [30] designed in early 1980s was the first simple high-level language that featured channel mobility in the sense of the π-calculus. When the author proposed Guarded Horn Clauses (GHC) [36] [37] as a simplification of Concurrent Prolog and PARLOG [8], the principal design constraint was to retain channel mobility and evolving process structures [32], because GHC was supposed to be the basis of KL1 [39], a language in which to describe operating systems of the Parallel Inference Machines as well as various knowledge-based systems.
4. *Non-strict.* Logical variables provide us with the paradigm of *computing with partial information.* Interesting programming idioms including short-circuits, difference lists and messages with reply boxes, as well as channel mobility, all exploit the power of partially instantiated data structures.

[1] We regard a process as an entity that is implemented as a multiset S of goals and communicates with other processes by generating and observing constraints on variables not local to S.

Some historical remarks would be appropriate here.

Concurrent logic programming was a field of active research throughout the 1980's, when a number of concurrent logic languages were proposed and the language constructs were tested through a number of implementations and applications [31]. The synchronization primitive, now known as *ask* based on logical entailment, was inspired independently around 1984 by at least three research groups, which suggests the stability of the idea [32].

Although concurrent logic languages achieved their flexibility with an extremely small number of language constructs, the fact that they were targeted to programming rather than reasoning about concurrent systems lead to little cross-fertilization with later research on mobile processes.

CCP was proposed in late 1980s as a unified theory underlying concurrent logic languages. It helped high-level understanding of constraint-based concurrency, but the study of constraint-based communication at a concrete level and the design of type systems and static analyses call for a fixed constraint system – most typically that of (concurrent) logic programming known as the Herbrand system – to work with.

2 The Essence of Constraint-Based Communication

2.1 The Language

To further investigate constraint-based communication, let us consider a concrete language, a subset of Flat GHC [38] whose syntax is given in Fig. 1.

(program)	P ::= set of R's	(1)
(program clause)	R ::= A `:-` | B	(2)
(body)	B ::= multiset of G's	(3)
(goal)	G ::= T_1 `=` T_2 | A	(4)
(non-unification atom)	A ::= p`(`T_1`,` $\ldots$ `,`T_n`)`, $p \neq$ '`=`'	(5)
(term)	T ::= (as in first-order logic)	(6)
(goal clause)	Q ::= `:-` B	(7)
(program clause, alternative)	R ::= $!\forall(A$ `.` $B)$	(2′)
(goal clause, alternative)	Q ::= B`,`P	(7′)

Fig. 1. The simplified syntax of Flat GHC

For simplicity, the syntax given in Fig. 1 omits guard goals from (2), which correspond to conditions in conditional rewrite rules. We use the traditional rule-based syntax rather than the expression-based one because it facilitates our analysis. The alternative syntax (2′) (7′) indicates that a program clause, namely a rewrite rule of goals, could be regarded as a replicated process that accepts a message A and spawns B, where the universal closure $\forall$ means that a variable either occurs in A and will be connected to the outside or occurs only in B as local channels. In this formulation, the program is made to reside in a goal clause.

2.2 Operational Semantics

The reduction semantics of GHC deals with the rewriting of goal clauses.

A *configuration* is a triple, $\langle B, C, P\rangle$, where B is a multiset of goals, C a multiset of equations (denoting equality constraints) that represents the store, and P a set of program clauses. A computation under a program P starts with the initial configuration $\langle B_0, \emptyset, P\rangle$, where B_0 is the body of the given goal clause.

We have three rules given in Fig. 2. In the rules, $F \models G$ means that G is a logical consequence of F. $\mathcal{V}_F$ denotes the set of all variables occurring in a syntactic entity F. $\forall\mathcal{V}_F(F)$ and $\exists\mathcal{V}_F(F)$ are abbreviated to $\forall(F)$ and $\exists(F)$, respectively. $\mathcal{E}$ denotes the standard syntactic equality theory over

$$\frac{\langle B_1, C, P\rangle \longrightarrow \langle B_1', C', P\rangle}{\langle B_1 \cup B_2, C, P\rangle \longrightarrow \langle B_1' \cup B_2, C', P\rangle} \quad \text{(i)}$$

$$\frac{}{\langle \{t_1 = t_2\}, C, P\rangle \longrightarrow \langle \emptyset, C \cup \{t_1 = t_2\}, P\rangle} \quad \text{(ii)}$$

$$\frac{}{\langle \{b\}, C, \{h \texttt{ :- | } B\} \cup P\rangle \longrightarrow \langle B, C \cup \{b = h\}, \{h \texttt{ :- | } B\} \cup P\rangle} \begin{pmatrix} \mathcal{E} \models \forall\big(C \Rightarrow \exists \mathcal{V}_h (b = h)\big) \\ \text{and } \mathcal{V}_{h,B} \cap \mathcal{V}_{b,C} = \emptyset \end{pmatrix} \quad \text{(iii)}$$

Fig. 2. The reduction semantics of GHC

1. $\forall\big(\neg(f(\mathtt{X}_1, \ldots, \mathtt{X}_m) = g(\mathtt{Y}_1, \ldots, \mathtt{Y}_n))\big)$, for all pairs f, g of distinct constructors (including constants)
2. $\forall\big(\neg(t = \mathtt{X})\big)$, for each term t other than and containing $\mathtt{X}$
3. $\forall\big(\mathtt{X} = \mathtt{X}\big)$
4. $\forall\big(f(\mathtt{X}_1, \ldots, \mathtt{X}_m) = f(\mathtt{Y}_1, \ldots, \mathtt{Y}_m) \Rightarrow \bigwedge_{i=1}^{m}(\mathtt{X}_i = \mathtt{Y}_i)\big)$, for each m-ary constructor f
5. $\forall\big(\bigwedge_{i=1}^{m}(\mathtt{X}_i = \mathtt{Y}_i) \Rightarrow f(\mathtt{X}_1, \ldots, \mathtt{X}_m) = f(\mathtt{Y}_1, \ldots, \mathtt{Y}_m)\big)$, for each m-ary constructor f
6. $\forall\big(\mathtt{X} = \mathtt{Y} \Rightarrow \mathtt{Y} = \mathtt{X}\big)$
7. $\forall\big(\mathtt{X} = \mathtt{Y} \wedge \mathtt{Y} = \mathtt{Z} \Rightarrow \mathtt{X} = \mathtt{Z}\big)$

Fig. 3. Clark's equality theory $\mathcal{E}$, in clausal form

finite terms and atomic formulas defined in Fig. 3. The second condition of Fig. 3, characterizing the finiteness of terms, is known as the *occur check*.

In Fig. 2, Rule (i) expresses concurrent reduction of a multiset of goals. Rule (ii) says that a unification goal simply publishes (or posts) a constraint to the current store. Rule (iii) deals with the reduction of a non-unification goal b to B using a clause h `:- |` B, which is enabled when the publication of $b = h$ will not constrain the variables in b. This means that the head unification is effectively restricted to matching. The second side condition guarantees that the guarded clause has been renamed using fresh variables. An immediate consequence of Rules (i)–(iii) is that the store grows monotonically and the reduction of b using a clause h `:- |` B remains enabled once it becomes enabled.

Sometimes it's more convenient to treat reduction in a traditional way as rewriting of goal clauses. The goal clause corresponding to a configuration $\langle B, C, P\rangle$ is `:-` $B\theta$, where θ is the most general unifier (mgu) of the set C of constraints. This substitution-based formulation is closer to actual implementation, but an advantage of the constraint-based formulation is that it can represent inconsistent stores, while mgu's can represent consistent stores only.

Yet another formulation may omit the second component, C, of a configuration together with Rule (ii) that simply moves an unguarded unification goal to the separate store. In this case, the current store is understood to comprise all the unguarded unification goals in B. However, we think it makes sense to distinguish between the three entities, namely definitions (code), processes, and the store.

2.3 Relation to Name-Based Concurrency

How can the constraint-based concurrency defined above relates to name-based concurrency?

First of all, predicate names can be thought of as global channel names if we regard the reduction of a non-unification goal (predicate name followed by arguments) as message sending to predicate definition. However, we don't regard this as a crucially important observation. We would rather forget this correspondence and focus on other symbols, namely variables and constructors.

Variables are local names that can be used as communication channels. Instead of *sending* a message *along* a channel, the the message is written to the channel itself and the receiver can asynchronously read the channel's value. For instance, let `S` be shared by processes P and Q (but nobody else) and suppose P sends a message `S=[read(X)|S']`. The message sends two subchannels, one a reply box `X` for the request `read`, and the other a *continuation* for subsequent communication. Then the goal in Q that owns `S`, say `q(S)`, can read the message using a clause head `q([read(A)|B])`,

identifying `A` with `X` and `B` with `S'` at the same time.[2] Alternatively, the identification of variables can be dispensed with by appropriately choosing an α-converted variant of the clause.

There is rather small difference between message passing of the asynchronous π-calculus and message passing by unification, as long as only one process holds a write capability *and* use it once. These conditions can be statically checked in well-moded concurrent logic programs [41] and in the π-calculus with a linear type system [19]. When two processes communicate repeatedly, constraint-based concurrency uses streams because one fresh logical variable must be prepared for each message passing, while in the linear π-calculus the same channel could be recycled as suggested in [19]. When two client processes communicate with a single server in constraint-based concurrency, an arbitration process should be explicitly created. A stream merger is a typical arbiter for repetitive multi-client communication:

```
merge([],Ys,Zs) :- | Zs=Ys.
merge(Xs,[],Zs) :- | Zs=Xs.
merge([A|Xs],Ys,Zs0) :- | Zs0=[A|Zs], merge(Xs,Ys,Zs).
merge(Xs,[A|Ys],Zs0) :- | Zs0=[A|Zs], merge(Xs,Ys,Zs).
```

In contrast, in name-based concurrency (without linearity), arbitration is built in the communication rule

$$a(y).Q \mid \overline{a}b \longrightarrow Q\{b/y\}$$

which chooses one of available outputs (forming a multiset of messages) on the channel a.

The difference in the semantics of input is much larger between the two formalisms. While *ask* is an non-destructive input, input in name-based concurrency destructively consumes a message, which is one of the sources of nondeterminism in the absence of choice operators. In constraint-based concurrency, non-destructiveness of *ask* is used to model one-way multicasting or data sharing naturally. At the same time, by using a linearity system, we can guarantee that only one process holds a read capability of a logical variable [46], in which case *ask* can destroy a message it has received, as will be discussed in detail in this paper.

One feature of constraint-based concurrency included into name-based concurrency only recently by the Fusion calculus [48] is that two channels can be fused into a single channel.

2.4 Locality in Global Store

The notion of shared, global store provided by CCP must be understood with care. Unlike conventional shared-memory multiprocessing, constraint store of CCP is highly structured and localized. All channels in constraint-based concurrency are created as local variables most of which are shared by two or a small community of processes, and a process can access them only when they are explicitly passed as (part of) messages or by fusing.

The only names understood globally are

1. predicate symbols used as the names of recursive programs, and
2. function symbols (constructors) for composing messages, streams, and data structures, and so on.

Although predicate symbols could be considered as channels, they are channels to classes rather than to objects. Constructors are best considered as non-channel names. They have various rôles as above, but cannot be used for sending messages through them. They can be examined by matching (*ask*) but cannot be equated with other constructors under strong moding.

3 I/O Mode Analysis

3.1 Motivation

By early 1990's, hundreds of thousands of lines of GHC/KL1 code were written inside and outside the Fifth Generation Computer Project [32]. The applications include an operating system for the

[2] In the syntax advocated by CCP, one should first ask $\exists$`A,B(q(S)=q([read(A)|B]))` (or equivalently, $\exists$`A,B(S=[read(A)|B])`) first and then tell `q(S)=q([read(A)|B])`.

Parallel Inference Machine (PIMOS) [6], a parallel theorem prover (MGTP) that discovered a new fact in finite algebra [10]. genetic information processing, and so on.

People found the communication and synchronization mechanisms of GHC/KL1 very natural. Bugs due to concurrency were rather infrequent[3] and people learned to model their problems in an object-based manner using concurrent processes and streams. At the same time, writing efficient *parallel* programs turned out to be a separate and much harder issue than writing correct *concurrent* programs.

By late 1980's, we had found that logical variables in concurrent logic languages were normally used for cooperative rather than competitive communication. Because the language and the model based on eventual *tell* provided no mechanism to cope with the inconsistency of a store (except for exception handers of KL1) and an inconsistent store allows any constraint to be read out, it was the responsibility of the programmers to keep the store consistent. Although shared logical variables were sometimes used for n-to-n signalling, in which two or more processes could write the same value to the same variable, for most applications it seemed desirable to provide syntactic control of interference so that the consistency of the store could be guaranteed statically. Obviously, a store remains consistent if only one process is allowed to have a write capability of each variable, as long as we ignore the *occur check* condition (Sect. 2.2).

The mode system[4] of Moded Flat GHC [41][43] was designed to establish this property while retaining the flexibility of constraint-based communication as much as possible. Furthermore, we can benefit very much from strong moding, as we do from strong typing in many other languages:

1. It helps programmers understand their programs better.
2. It detects a certain kind of program errors at compile-time. In fact, the Kima system we have developed [2][3] goes two steps forward: it *locates*, and then automatically *corrects*, simple program errors using constraint-based mode and type analyses. The technique used in Kima is very general and could be deployed in other typed languages as well.
3. It establishes some fundamental properties statically (Sect. 3.5):
   (a) well-moded programs do not collapse the store.
   (b) all variables are guaranteed to become *ground* terms upon termination.
4. It provides basic information for program optimization such as
   (a) elimination of various runtime checks,
   (b) (much) simpler distributed unification, and
   (c) message-oriented implementation [41][40].

3.2 The Mode System

The purpose of our mode system is to assign *polarity structures* (modes) to every predicate argument and (accordingly) every variable occurrence in a configuration, so that each part of data structures will be determined cooperatively, namely by *exactly one* process that owned a write capability. If more than one process owned a write capability to determine some part a structure, the communication would be competitive rather than cooperative. If no process owned a write capability, the communication would be neither cooperative or competitive, because the readers would never get a value.

Since variables may be bound to complex data structures in the course of computation whose exact shapes are not known beforehand, a polarity structure reconstructed by the mode system should tell the polarity structures of all possible data structures the program may create and read. To this end, a mode is defined as a function from the set of paths specifying positions in data structures occurring in goals, denoted P_{Atom}, to the set $\{in, out\}$. Paths here are strings of $\langle$*symbol*, *argument-position*$\rangle$ pairs in order to be able to specify positions in data structures that are yet to be formed.

[3] Most bugs were due to higher-level design problems that often arose in, for example, programs dealing with circular process structures concurrently.

[4] Modes have sometimes been called directional types. In any case modes are (non-standard) types that deal with read/write capabilities.

Formally, the sets of paths for specifying positions in terms and atomic formulas are defined, respectively, using disjoint union as:

$$P_{Term} = (\sum_{f \in Fun} N_f)^* \ , \quad P_{Atom} = (\sum_{p \in Pred} N_p) \times P_{Term} \ ,$$

where *Fun* and *Pred* are the sets of constructors and predicate symbols, respectively, and N_f and N_p are the sets of positive integers up to and including the arities of f and p, respectively.

3.3 Mode Analysis

Mode analysis tries to find a mode $m : P_{Atom} \to \{in, out\}$ under which every piece of communication will be performed cooperatively. Such a mode is called a *well-moding*. A well-moding is computed by constraint solving. Constructors in a program/goal clause will impose constraints on the possible polarities of the paths at which they occur. Variable symbols may constrain the polarities not only of the paths at which they occur but of any positions below those paths. The set of all these mode constraints syntactically imposed by the symbols or the symbol occurrences in a program does not necessarily define a unique mode because the constraints are usually not strong enough to define one. Instead it defines a 'principal' mode that can best be expressed as a mode graph, as we will see in Section 3.6.

Mode constraints imposed by a clause h :- | B, where B are multisets of atomic formulae, are summarized in Fig. 4. Here, *Var* denotes the set of variable symbols, and $\widetilde{a}(p)$ denotes a symbol occurring at p in an atomic formula a. When p does not lead to a symbol in a, $\widetilde{a}(p)$ returns $\bot$. A *submode* of m at p, denoted m/p, is a function (from P_{Term} to $\{in, out\}$) such that $(m/p)(q) = m(pq)$. *IN* and *OUT* are constant submodes that always return *in* or *out*, respectively. An overline, "$-$", inverts the polarity of a mode, a submode, or a mode value.

(HF) $\forall p \in P_{Atom}\big(\widetilde{h}(p) \in Fun \Rightarrow m(p) = in\big)$
(if the symbol at p in h is a constructor, $m(p) = in$)

(HV) $\forall p \in P_{Atom}\big(\widetilde{h}(p) \in Var \wedge \exists p' \neq p\big(\widetilde{h}(p) = \widetilde{h}(p')\big) \Rightarrow m/p = IN\big)$
(if the symbol at p in h is a variable occurring elsewhere in h, then $m/p = IN$)

(BU) $\forall k > 0\, \forall t_1, t_2 \in Term\big((t_1 =_k t_2) \in B \Rightarrow m/\langle =_k, 1\rangle = \overline{m/\langle =_k, 2\rangle}\big)$
(the two arguments of a unification body goal have complementary submodes)

(BF) $\forall p \in P_{Atom} \forall a \in B\big(\widetilde{a}(p) \in Fun \Rightarrow m(p) = in\big)$
(if the symbol at p in a body goal is a constructor, $m(p) = in$)

(BV) Let $v \in Var$ 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 $\mathcal{R}$ is a 'cooperativeness' relation:

$$\mathcal{R}(S) \stackrel{\text{def}}{=} \forall q \in P_{Term}\, \exists s \in S\big(s(q) = out \wedge \forall s' \in S \backslash \{s\}\, \big(s'(q) = in\big)\big)$$

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

For goal clauses, Rules (BU), (BF) and (BV) are applicable.

Note that Rule (BV) ignores the second and the subsequent occurrences of v in h. The occurrences of v that are not ignored are called *channel occurrences*. Note also that s can depend on q in the definition of $\mathcal{R}$. Intuitively, Rule (BV) means that each constructor occurring in a possible instance of v will be determined by exactly one of the channel occurrences of v.

Unification body goals, dealt with by Rule (BU), are polymorphic in the sense that different goals are allowed to have different modes. To deal with polymorphism, we give each unification

body goal a unique number. Polymorphism can be incorporated to other predicates as well [43], but we do not discuss it here.

3.4 Moding Principles

What are the principles behind these moding rules?

In concurrent logic programming, a process implemented by a multiset of goals can be considered an information processing device with inlets and outlets of constraints that we call *terminals*. A variable is a one-to-n ($n \geq 0$) communication channel connecting its occurrences, and each occurrence of a variable is considered to be plugged into one of the terminals of a goal.

We say that a variable is *linear* when it has exactly two occurrences in a goal clause. Similarly, a variable in a program clause is said to be *linear* when it has exactly two channel occurrences in the clause.

A variable occurring both in the head and in the body of a program clause is considered a channel that connects a goal (which the head matches) and its subgoals. A constructor is considered an unconnected plug that acts as the source or the absorber of atomic information, depending on whether it occurs in the body or the head. While channels and terminals of electric devices usually have array structures, those in our setting have nested structures. That is, a variable that connects the terminals at $p_1, \ldots, p_n$ also connects the terminals at $p_1 q, \ldots, p_n q$, for all $q \in P_{Term}$. Linear variables are used as *cables* for one-to-one communication, while nonlinear variables are used as *hubs* for one-to-many communication.

A terminal of a goal always has its counterpart. The counterpart of a terminal at p on the caller side of a non-unification goal is the one at the same path on the callee side, and the counterpart of a terminal at $\langle =_k, 1 \rangle q$ in the first argument of a unification goal is the one at $\langle =_k, 2 \rangle q$ in the second argument. Reduction of a goal is considered the removal of the pairs of corresponding terminals whose connection has been established.

The mode constraints are concerned with the direction of information flow (1) in channels and (2) at terminals. The two underlying principles are:

1. When a channel connects n terminals of which at most one is in the head, exactly one of the terminals is the outlet of information and the others are inlets.
2. Of the two corresponding terminals of a goal, exactly one is the outlet of information and the other is an inlet.

Rule (BV) comes from Principle 1. An input (output) occurrence of a variable in the head of a clause is considered an outlet (inlet) of information from inside the clause, respectively, and this is why we invert the mode of the clause head in Rule (BV). Rule (BV) takes into account only one of the occurrences of v in the head. Multiple occurrences of the same variable in the head are for equality checking before reduction, and the only thing that matters after reduction is whether the variable occurs also in the body and conveys information to the body goals.

Rules (HF) and (HV) come from Principle 2. When some clause may examine the value of the path p in a non-unification goal, $m(p)$ should be constrained to *in* because the examination is done at the *outlet* of information on the *callee* side of a goal. The strong constraint imposed by Rule (HV) is due to the semantics of Flat GHC: when a variable occurs twice or more in a clause head, these occurrences must receive identical terms from the caller.

Rule (BU) is exactly the application of Principle 2 to unification body goals. Any value fed through some path $\langle =_k, i \rangle q$ in one of its arguments will come out through the corresponding path $\langle =_k, 3 - i \rangle q$ in the other argument.

Rule (BF) also comes from Principle 2. A non-variable symbol on the caller side of a goal must appear only at the inlet of information, because the information will go out from the corresponding outlet.

The relation $\mathcal{R}$ enjoys the following properties:

$$\mathcal{R}\big(\{s\}\big) \Leftrightarrow s = OUT \tag{1}$$

$$\mathcal{R}\big(\{s_1, s_2\}\big) \Leftrightarrow s_1 = \overline{s_2} \tag{2}$$

$$\mathcal{R}\big(\{IN\} \cup S\big) \Leftrightarrow \mathcal{R}\big(S\big) \tag{3}$$

$$\mathcal{R}\big(\{OUT\} \cup S\big) \Leftrightarrow \forall s' \in S\big(s' = IN\big) \tag{4}$$
$$\mathcal{R}\big(\{s, s\} \cup S\big) \Leftrightarrow s = IN \wedge \mathcal{R}\big(S\big) \tag{5}$$
$$\mathcal{R}\big(\{\overline{s}, s\} \cup S\big) \Leftrightarrow \forall s' \in S\big(s' = IN\big) \tag{6}$$
$$\mathcal{R}\big(\{\overline{s}\} \cup S_1\big) \wedge \mathcal{R}\big(\{s\} \cup S_2\big) \Rightarrow \mathcal{R}\big(S_1 \cup S_2\big) \tag{7}$$
$$\mathcal{R}\big(\bigcup_{1 \le i \le n}\{s_i\}\big) \Rightarrow \mathcal{R}\big(\bigcup_{1 \le i \le n}\{s_i/q\}\big), \quad q \in P_{Term} \tag{8}$$

Proofs are all straightforward. Property (7) is reminiscent of Robinson's resolution principle.

Properties (1) and (2) say that Rule (BV) becomes much simpler when the variable v has at most two channel occurrences. When it has exactly two channel occurrences at p_1 and p_2. Rule (BV) is equivalent to $m/p_1 = m/p_2$ or $m/p_1 = \overline{m/p_2}$, depending on whether one of the occurrences is in the head or the both occur in the body. When v has only one channel occurrence at p, Rule (BV) is equivalent to $m/p = IN$ or $m/p = OUT$, depending on whether the occurrence is in the head or the body.

3.5 Properties of Well-Moded Programs

The three important properties of well-moding are as follows:

1. Let m be a well-moding of a clause R, and let $t_1 =_k t_2$ be a unification (body) goal in R. Then there exists an i such that (i) $m(\langle =_k, i\rangle) = out$ and (ii) t_i is a variable.
   This means a unification body goal is effectively assignment to an variable with a write capability.
2. (*Subject Reduction*) Let m be a well-moding of a program P and a goal clause Q. Suppose Q is reduced by one step into a goal clause Q' (in the substitution-based formulation (Sect. 2.2)), where the reduced goal $g \in Q$ is *not* a unification goal that unifies a variable with itself or a term containing the variable. Then m is a well-moding of P and Q' as well.
   As a corollary, well-moded programs keep store consistent as long as the reductions obey the above condition on the reduced goal, which is called the *extended occur-check condition*.
3. (*Groundness*) Let m be a well-moding of a program P and a goal clause Q. Assume Q has been reduced to an empty multiset of goals under the extended occur-check condition. Then, in that execution, a unification goal of the form $v =_k t$ such that $m(\langle =_k, 1\rangle) = out$, or a unification goal of the form $t =_k v$ such that $m(\langle =_k, 2\rangle) = out$, must have been executed, for any variable v occurring in Q.
   As a corollary, the product of all substitutions generated by unification body goals maps all the variables in Q to ground (variable-free) terms.

3.6 Mode Graphs and Principal Modes

It turns out that most of the mode constraints are either of the six forms: (i) $m(p) = in$, (ii) $m(p) = out$, (iii) $m/p = IN$, (iv) $m/p = OUT$, (v) $m/p_1 = m/p_2$, or (vi) $m/p_1 = \overline{m/p_2}$. We call (i)–(iv) *unary* constraints and (v)–(vi) *binary* constraints.

A set of binary and unary mode constraints can be represented as a feature graph (feature structures with cycles), called a *mode graph*, in which

1. paths represent paths in P_{Atom},
2. nodes may have mode values determined by unary constraints,
3. arcs may have "negative signs" that invert the interpretation of the mode values beyond those arcs, and
4. binary constraints are represented by the sharing of nodes.

Figure 5 is the mode graph of the `merge` program. An arc of a mode graph represents the pair of a predicate/constructor (abbreviated to its initial in the figures) and an argument position. A dot "." stands for the list constructor. The pair exactly corresponds to a feature of a feature graph. A sequence of features forms a path both in the sense of our mode system and in the graph-theoretic sense.

A node is possibly labeled with a mode value (*in* shown as "↓", or *out* shown as "↑") to which any paths p_1, p_2, ... terminating with that node are constrained, or with a constant submode (*IN*

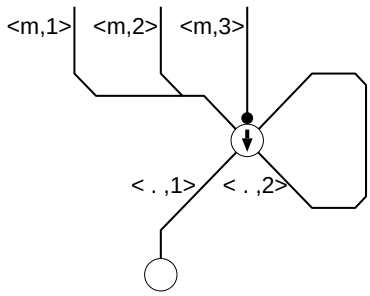

Fig. 5. Mode graph of the `merge` program

shown as "↓" with a grounding sign (as in Fig. 7), or *OUT*) to which the submodes m/p_1, m/p_2, ... are constrained.

An arc is either a negative arc (bulleted in the figures) or a positive arc. When a path passes an odd number of negative arcs, that path is said to be *inverted*, and the mode value of the path is understood to be inverted. Thus the number of bulleted arcs on a path determines the *polarity* of the path.

A binary constraint of the form $m/p_1 = m/p_2$ or $m/p_1 = \overline{m/p_2}$ is represented by a shared node with two (or more) incoming paths with possibly different polarities. When the polarities of the two incoming paths are different, the shared node stands for complementary submodes; otherwise the node stands for identical submodes.

Figure 5 has a node, under the arc labeled $\langle ., 1\rangle$, that expresses no constraints at all. It was created to express binary constraints, but all its parent nodes were later merged into a single node by other constraints.

All these ideas have been implemented in the mode analyzer for KL1 program, *klint v2* [44], which can output a text version of the mode graph as in Fig. 5.

As another example, consider a program that simply unifies its arguments:

```
p(X,Y) :- | X = Y.
```

The program forms a mode graph shown in Fig. 6. This graph can be viewed as the *principal mode* of the predicate `p`, which represents many possible particular modes satisfying the constraint $m/\langle \mathtt{p}, 1\rangle = \overline{m/\langle \mathtt{p}, 2\rangle}$. In general, the principal mode of a well-moded program, represented as a mode graph, is uniquely determined, as long as all the mode constraints imposed by the program are unary or binary.

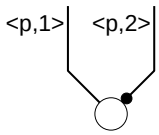

Fig. 6. Mode graph of the unify program

Constraints imposed by the rule (BV) may be non-binary. Non-binary constraints are imposed by nonlinear variables, and cannot be represented as mode graphs by themselves. However, by *delaying* them, most of them are reduced to unary/binary ones by other constraints. In this case they can be represented in mode graphs, and the programs that imposed them have unique principal modes (as long as they are well-moded). Theoretically, some non-binary constraints may remain unreduced, whose satisfiability must be checked eventually.

When some constraints remain non-binary after solving all unary or binary constraints, *klint v2* assumes that nonlinear variables involved are used for one-way multicasting rather than bidirectional communication. Thus, if a nonlinear variable occurs at p and $m(p)$ is known to be *in* or *out*, *klint*

v2 imposes a stronger constraint $m/p = IN$ or $m/p = OUT$, respectively. This means that a mode graph computed by *klint v2* is not always 'principal', 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 multicasting and the strengthening causes no problem in practice.

The union (i.e., conjunction) of two sets of constraints can be computed efficiently as unification over feature graphs. For instance, adding a new constraint $m/p_1 = m/p_2$ causes the subgraph rooted at p_1 and the subgraph rooted at p_2 to be unified. A good news is that an efficient unification algorithm for feature graphs has been established [1].

Figure 7 shows the mode graph of a quicksort program using difference lists. The second and the third clause of `part` checks the principal constructor of `A` and `X` using guard goals, so the moding rule of variables occurring in guard goals (not stated in this paper) constrains $m(\langle \texttt{part}, 1\rangle)$ and $m(\langle \texttt{part}, 2\rangle\langle \texttt{.}, 1\rangle)$ to *in*. The head and the tail of a difference list, namely the second and the third arguments of `qsort`, are constrained to have complementary submodes.

```
qsort([],    Ys0,Ys ) :- | Ys=Ys0.
qsort([X|Xs],Ys0,Ys3) :- |
    part(X,Xs,S,L), qsort(S,Ys0,[X|Ys2]), qsort(L,Ys2,Ys3).
part(_,[],    S, L ) :- | 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).
```

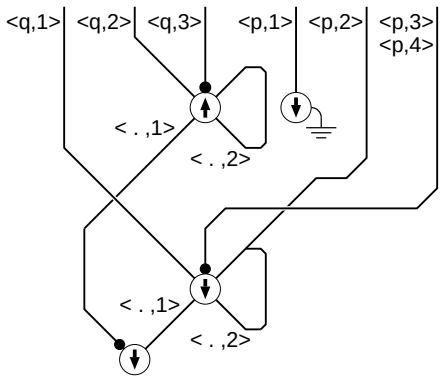

Fig. 7. A quicksort program and its mode graph

4 Linearity Analysis

4.1 Motivation and Observation

The mode system guarantees the uniqueness of *write* capability of each variable in a runtime configuration. Furthermore, although it does not impose any constraint on the number of read capabilities (occurrences), it imposes less generic, stronger mode constraints on programs that may discard or duplicate data. The modes of the paths where singleton variables (one-to-zero communication channels) may occur are constrained to *IN* or *OUT*, and paths of nonlinear variables (one-to-many communication channels) may very well be constrained to *IN* or *OUT*. Thus the mode system effectively prefers programs that do not discard or duplicate data by giving them weaker mode constraints, providing the view of "data as resources" to some extent.

Our experiences with Prolog and concurrent logic programming show that surprisingly many variables in Prolog and concurrent logic programs are linear. For instance, all the variables in the `merge` program (Fig. 5) are linear, and all but one of the variables in `qsort` (Fig. 7) are linear. This indicates that the majority of communication is one-to-one and deserves special attention.

As a non-toy example, we examined the mode constraint solver of *klint v2*, which comprised 190 KL1 clauses [43]. Those clauses imposed 1392 constraints by Rule (BV), one for each variable in the program, of which more than 90% were of the form $m/p_1 = m/p_2$ (1074) or $m/p_1 = \overline{m/p_2}$ (183). Thus we can say that the clauses are highly linear. Furthermore, all of the 42 non-binary constraints were reduced to unary or binary constraints using other unary or binary constraints. Actually they were reduced to 6 constraints of the form $m/p_1 = \overline{m/p_2}$ and 72 constraints of the form $m/p = IN$. This means that nonlinear variables were all used under simple, one-way communication protocols.

4.2 The Linearity System

The purpose of linearity analysis [46] is to statically analyze exactly *where* nonlinear variables and shared data structures may occur – in which predicates, in which arguments, and in which part of the data structures carried by those arguments. This complements mode analysis in the sense that it is concerned with the number of *read* capabilities.

To distinguish between non-shared and shared data structures in a reduction semantics without the notion of pointers, we consider giving a linearity annotation 1 or ω to every occurrence of a constructor f appearing in (initial or reduced) goal clauses and body goals in program clauses.[5] The annotations appear as f^1 or f^ω 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 constructor of a structure possibly referenced by more than one pointer must have the annotation ω, 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 ω.

The annotations must observe the following closure condition: If the principal constructor of a term has the annotation ω, all constructors occurring in the term must have the annotation ω. In contrast, a term with the principal constructor annotated as 1 can contain a constructor 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 constructors 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],X).
   ```

   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 constructors (there are 11 of them including `[]`) must be given ω. 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 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 constructors constituting t_i to ω.
   (b) When v_i is linear, θ 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.

As in mode analysis, the linearity of a (well-moded) program can be characterized using a linearity function, a mapping from P_{Atom} to the binary codomain $\{\mathit{nonshared}, \mathit{shared}\}$, which satisfies the closure condition

$$\forall p \in P_{Atom} \forall q \in P_{Term} \big(\lambda(p) = \mathit{shared} \Rightarrow \lambda(pq) = \mathit{shared}\big).$$

The purpose of linearity analysis is to reconstruct a linearity function (say λ) that satisfies all *linearity constraints* imposed by each program clause and a goal clause `:-` B, which are shown in Fig. 8. The *klint v2* system reconstructs linearity as well as mode information.

[5] The notation is after related work [19, 35] on different computational models.

(BF_λ)	If a function symbol f^ω occurs at the path p in B, then $\lambda(p) = \mathit{shared}$.	
(LV_λ)	If a linear variable occurs both at p_1 and p_2, then	
	$\forall q \in P_{\mathit{Term}}\big(m(p_1 q) = \mathit{in} \wedge \lambda(p_1 q) = \mathit{shared} \Rightarrow \lambda(p_2 q) = \mathit{shared}\big)$	(if p_1 is a head path);
	$\forall q \in P_{\mathit{Term}}\big(m(p_1 q) = \mathit{out} \wedge \lambda(p_1 q) = \mathit{shared} \Rightarrow \lambda(p_2 q) = \mathit{shared}\big)$	(if p_1 is a body path).
(NV_λ)	If a nonlinear variable occurs at p, then	
	$\forall q \in P_{\mathit{Term}}\big(m(pq) = \mathit{out} \Rightarrow \lambda(pq) = \mathit{shared}\big)$	(if p is a head path);
	$\forall q \in P_{\mathit{Term}}\big(m(pq) = \mathit{in} \Rightarrow \lambda(pq) = \mathit{shared}\big)$	(if p is a body path).
(BU_λ)	For a unification body goal $=_k$, $\forall q \in P_{\mathit{Term}}\big(\lambda(\langle =_k, 1\rangle q) = \lambda(\langle =_k, 2\rangle q)\big)$.	

Fig. 8. Linearity constraints imposed by a clause h :- | B

As expected, the linearity system enjoys the subject reduction theorem:

- Suppose λ satisfies the linearity constraints of a program P and a goal clause Q, and Q is reduced in one step to Q' under the extended occur-check condition. Then λ satisfies the linearity constraints of Q' as well.

An immediate consequence of the subject reduction theorem is that a constructor with ω cannot appear at p such that $\lambda(p) = \mathit{nonshared}$. The (sole) reader of the data structure at a *nonshared* path can safely discard the top-level structure after accessing its elements. One feature of our linearity system is that it can appropriately handle data structures whose sharing properties change (from *nonshared* to *shared*) in their lifetime, allowing update-in-place of not yet shared data structures.

There is one subtle point in this optimization. The optimization is completely safe for built-in data types such as numeric or character arrays that allow only instantiated data to be stored. However, when a structure is implemented so that its field may itself represent an uninstantiated logical variable, the structure cannot be recycled until the variable is instantiated (through an *internal pointer* to the variable) and read. Most implementations of Prolog and concurrent logic languages (including KLIC [7]) represent structures this way for efficiency reasons, in which case local reuse requires strictness analysis, namely the analysis of instantiation states of variables, in addition to linearity analysis. The implementation of KL1 on the Parallel Inference Machine disallowed internal pointers to feature local reuse based on the 1-bit reference counting technique [5].

5 From Linearity to Strict Linearity

5.1 Polarizing Constructors

We already saw that most if not all variables in concurrent logic languages are linear variables. To start another observation, consider the following insertion sort program:

```
sort([],    S) :- | S=[].
sort([X|L0],S) :- | sort(L0,S0), insert(X,S0,S).
insert(X,[],    R) :-      | R=[X].
insert(X,[Y|L], R) :- X=<Y | R=[X,Y|L].
insert(X,[Y|L0],R) :- X>Y  | R=[Y|L], insert(X,L0,L).
```

Here again, all the variables are linear (we do not count the occurrences in guard goals when considering linearity). However, an even more striking fact is that, by slight modification, all constructors (including constants) can be made to occur exactly twice as well:

```
sort([],    S) :- | S=[].
sort([X|L0],S) :- | sort(L0,S0), insert([X|S0],S).
insert([X],     R) :-      | R=[X].
insert([X,Y|L], R) :- X=<Y | R=[X,Y|L].
insert([X,Y|L0],R) :- X>Y  | R=[Y|L], insert([X|L0],L).
```

This suggests that the notion of linearity could be extended to cover constructors as well. We call it *strict linearity*. A linear variable is a *dipole* with two occurrences with opposite polarities. Likewise, a linear constructor is a dipole with two occurrences with opposite polarities, one in the head and the other in the body of a clause. The two occurrences of a linear constructor can be regarded as two *polarized* instances of the same constructor.

If all the constructors are linear in program clauses as in the second version of `sort`, all deallocated cells can be reused locally to allocate new cells without accessing a non-local free list. That is, as long as the input list is not shared, the program can construct the final result by reorganizing input cells and without generating any garbage cells or allocating new cells from non-local storage.

5.2 Strict Linearity

We say that a program clause is *strictly linear* if all the variables have exactly two channel occurrences in the clause and all the constructors have exactly two occurrences, one in the head and the other in the body.

The above definition does not require that predicate symbols occur exactly twice. If this is enforced, all body goals can inherit its goal record (that records the argument of goals) from the parent and the program can run with a fixed space, but we must have a means to terminate tail recursion, as will be discussed in Sect. 5.3. Although strictly linear programs still require allocation and deallocation of goal records, goal records are inherently non-shared and much more manageable than heap-allocated data. So strict linearity is a significant step toward resource-conscious programming.

Let me give another example.

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

The base case receives an empty list but does not use it. A short value such as `[]` could be regarded as a zero-resource value, but we prefer to consider n-ary constructors to convey $n+1$ units in general, in which case the program recovers strict linearity using an extra argument:

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

Note that the first version of `append` can be thought of as a *slice* of the second version.

5.3 Void: The Zero-Capability Symbol

All the examples above are transformational processes. It is much less clear how one can program reactive, server-type processes that respond to streams of requests in a strictly linear setting. Consider the following stack server:

```
stack([],         D    ) :- | true.
stack([push(X)|S],D    ) :- | stack(S,[X|D]).
stack([pop(X)|S], [Y|D]) :- | X=Y, stack(S,D).
```

One way to recover the strict linearity of this program is:

```
stack([](Z),            D) :- | Z=[](D).
stack([push([X|*],Y)|S],D) :- | Y=[push(*,*)|*], stack(S,[X|D]).
stack([pop(X,Z)|S], [Y|D]) :- | X=[Y|*], Z=[pop(*,*)|*], stack(S,D).
```

Note that an empty list, which can be regarded as an "end-of-transaction" message to a process, has been changed to a unary constructor conveying a reply box, through which the current "deposit" will be returned. Upon receiving a message, the server immediately returns the resource used by the client to send the message. In a physical mail metaphor, cons cells can be compared to envelopes and `push` and `pop` messages can be compared to cover letters, which real-world servers often fail to find a nice way of recycling.

Observe that we need to extend the language with a special symbol, `*`, to indicate void positions of structures. A void position will be given *zero capability* so that no read or write to the position will take place.

What is the resource aspect of variable occurrences and the void symbol? We assume that each variable occurrence uses one unit and one void symbol uses one unit. Furthermore, we assume that a non-variable term is always pointed to from a variable occurrence and an argument of a non-variable term or a goal always points to a variable occurrence. This canonical representation is not always space-optimal, but makes resource counting simple and uniform.

It is not difficult to see that the strictly linear versions of `append` and `stack` do not allocate or deallocate resources for variable occurrences and voids during tail-recursive iteration.

An interesting aspect of the *void* construct is that it could be used to recover the linearity of *predicate* symbols by allowing multi-head clauses as in Constraint Handling Rules [12]. For instance, `sort` in Sect. 5.1 could be rewritten as

```
sort([],    S) :-               | S=[], sort(*.*).
sort([X|L0],S), insert(*,*) :- | sort(L0,S0), insert([X|S0],S).
```

where the goals with void arguments could be considered as free, inactive goals waiting for habitants. The first clause makes the current goal inactive, while the second clause explicitly says that it requires one free `insert` goal for the reduction of `sort([X|L0],S)`. However, in this paper we assume that these free goals are implicit.

Some readers will enjoy the symmetry of strictly linear programs, while others may find it cumbersome and want compilers to reconstruct strict linear versions of their programs automatically. In any case, strictly linear programs are completely recyclic and are a step towards small-footprint symbolic computation with highly predictable behavior.

One natural question is how to write programs whose output size is essentially larger than the input size. An obvious solution is to require the initial process to pass a necessary number of cells obtained from the runtime library. This will work well as long as the output size is predictable. Some programs may have the output size that is essentially the same as the input size but may require more resource to represent intermediate results. We have two solutions to this. One is to let the initial process provide all necessary resource. The other is to require that the input size and the output size be balanced for each process spawned during computation, but allow a subprocess to use (and then return) more resource than that received from the parent process. The notion of strict linearity has to be relaxed somewhat to enable the latter alternative.

5.4 Constant-Time Property of Strictly Linear Programs

The cost of the primitive operations of strictly linear concurrent logic programs are highly predictable despite their non-strict data structures.

The primitive operations of concurrent logic programs are:

1. spawning of new non-unification goals (including tail-recursive ones),
2. termination of a non-unification goal (upon reduction with a base-case clause),
3. *ask* or term matching, which may involve
   (a) synchronization, namely suspension and resumption of the process, and
   (b) pointer dereferencing, and
4. *tell*, namely execution of a unification body goal, which may involve pointer dereferencing.

On a single-processor environment, spawning and termination of a goal involves (de)allocation of a fixed-size goal record and manipulation of a goal queue, which can both be regarded as constant-time operations.

Synchronization involves the hooking and unhooking of goals on an uninstantiated variable, but in a linear setting, the number of goals hooked on each variable is at most one.

The cost of dereferencing reflects the length of the chain of pointers formed by unification. Due to the flexibility logical variables bring in the way of data structure formation, even in sequential Prolog it is possible to create an arbitrarily long chain of pointers between variables.

In a linear setting, however, every uninstantiated variable has exactly two occurrences. We can represent it using two cells that form a size-two cycle by pointing to each other. Then the unification

of two linear variables, say v_1 and v_2, which consumes one v_1 and one v_2 by the unification goal itself, can be implemented by letting the other occurrence of v_1 and the other occurrence of v_2 point to each other. This keeps the size-two property of uninstantiated linear variables unchanged. The writing to a linear variable v, which consumes one occurrence of v, dereferences the pointer to reach the other occurrence of v and instantiate it. The reader of v dereferences it exactly once to access its value.

6 Allowing Concurrent Access within Strict Linearity

Strict linearity can be checked by slightly extending the mode and linearity systems described earlier. However, rather than doing so, we consider extending the framework to allow concurrent access to shared resource. There are two reasonable ways of manipulating resource such as large arrays concurrently.

One is to give different processes exclusive (i.e., non-shared) read/write capabilities to *different* parts of a data structure. This is easily achieved by (i) splitting a non-shared structure, (ii) letting each process work on its own fragment and return results by update-in-place, and (iii) joining the results into one. For instance, parallel quicksort of an array has been implemented this way using optimized versions of KLIC's vectors [25]. Concurrent manipulation of this type fits nicely within the present framework of mode and linearity systems because no part of an array becomes shared.

On the other hand, some applications require concurrent accesses with non-exclusive, read capability to the whole of a data structure to allow concurrent table lookup and so on. When the accesses can be sequentialized, the structure with an exclusive capability can be returned finally either

1. by letting each process receive and return an exclusive capability one after another or
2. by guaranteeing the sequentiality of accesses by other language constructs (`let`, guard, *etc.*) as in [49] and [18].

However, these solutions cannot be used when we need to run the readers concurrently or in parallel. Our goal is to allow some process (say P) to collect all released non-exclusive capabilities so that P may restore an *exclusive* capability and update it in place.

For this purpose, we refine the $\{in, out\}$ capability domain of the mode system to a continuous domain $[-1, +1]$. As in the mode system, the capability is attached to all paths. Let κ be the capability of the principal constructor of some occurrence of a variable in a configuration. Then

1. $\kappa = -1$ means 'exclusive write',
2. $-1 < \kappa < 0$ means 'non-exclusive write',
3. $\kappa = 0$ means no capability,
4. $0 < \kappa < 1$ means 'non-exclusive read', and
5. $\kappa = +1$ means 'exclusive read'.

This is a refinement of the mode system in the sense that *out* corresponds to -1 and *in* corresponds to $(0, +1]$. This is also a refinement of the linearity system in the sense that *nonshared* corresponds to ± 1 and *shared* corresponds to $(0, +1]$.

Then, what meaning should we give to the $(-1, 0]$ cases? Suppose a `read` process receives an exclusive read capability to access `X0` and split the capability to two non-exclusive capabilities using the following clause:

```
read(X0,X) :- | read1(X0,X1), read2(X0,X2), join(X1,X2,X).
```

The capability `X0` conveys can be written as a function $\mathbf{1}$, a constant function such that $\mathbf{1}(p) = +1$ for all $p \in P_{Atom}$. We don't care how much of the $\mathbf{1}$ capability goes to `read1` but just require that the capabilities of the two `X0`'s sum up to $\mathbf{1}$. Different paths in one of the split occurrences of `X0` may convey different capabilities. Also, we assume that the capabilities given to `read1` and `read2` are returned with the opposite polarity through `X1` and `X2`. Logically, `X1` and `X2` will become the same as `X0`. Then, `join` process defined as

```
join(A,A,B) :- | B = A.
```

checks if the first two arguments are indeed aliases and then returns it through the third argument. Note the interesting use of a nonlinear head. The capability constraint for the join program is that the capabilities of the three arguments must sum up to a constant function $\mathbf{0}$.

Now the `X` returned by `join` is guaranteed to convey the $\overline{\mathbf{1}}$ (exclusive write) capability that complements the capability received through `X0`.

7 Operational Semantics with Capability Counting

In the linearity analysis described earlier, the operational semantics was augmented with a linearity annotation 1 or ω given to every occurrence of a constructor f appearing in (initial or reduced) goal clauses. Here, we replace the annotation with a capability annotation κ ($0 < \kappa \leq 1$). $\kappa = 1$ (exclusive) corresponds to the 1 annotation meaning 'non-shared', while $\kappa < 1$ (non-exclusive) refines (the reciprocal of) ω. Again, the annotations are to reason about capabilities statically and are to be compiled away.

The annotations must observe the following closure condition: If the principal constructor of a term has a non-exclusive capability, all constructors occurring in the term must have non-exclusive capabilities as well. In contrast, a term with an exclusive principal constructor can contain a constructor with any capability.

The operational semantics is extended to handle annotations so that they may remain consistent with the above intuitive meaning. However, before doing so, let us consider how we can start computation within a strictly linear framework. The goal clause

```
:- sort([3,1,4,1,5,9],X).
```

is not an ideal form to work with because variables and constructors are monopoles. Instead, we consider a strictly linear version of the goal clause

```
main([3,1,4,1,5,9],X) :- | sort([3,1,4,1,5,9],X).
```

in which the head complements the resources in the body. The head declares the resources necessary to initiate computation and the resources to be returned to the environment. The reduction semantics works on the body goal as before, except that the unification goal to instantiate `X` remains intact. Then the above clause will be reduced to

```
main([3,1,4,1,5,9],X) :- | X = [1,1,3,4,5,9].
```

which can be thought of as a normal form in our polarized setting.

Hereafter, we assume that an initial goal clause is complemented by a head to make it strictly linear.

1. All the constructors in the body of an *initial* goal clause are given the annotation 1. This could be relaxed as we did in giving the linearity annotations (Sect. 4) to represent initially shared data, but without loss of generality we can assume that initial data are non-shared.
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,[6] the substitution instantiates more than one occurrence of v_i to t_i and makes t_i shared. Accordingly, all 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 constructors constituting t_i as follows: Let f^κ be a constructor in t_i and t_i is about to be copied to m places (this happens when v_i occurs $m+1$ times in the goal clause). Then κ is split into $\kappa_1, \ldots, \kappa_m$, where $\kappa_1 + \ldots + \kappa_m = \kappa$, $\kappa_j > 0$ ($1 \leq j \leq m$), and the κ_j's are mutually different and not used previously as capabilities.
   (b) When v_i is linear, θ 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.

Furthermore, to deal with capability polymorphism described later, we index the predicate symbols of the goals in an initial goal clause with 1, 2, and so on. The indices are in fact singleton sequences of natural numbers which will be extended in each reduction. That is, when reducing a non-unification goal b_s (s being the index) to spawn $b^1, \ldots, b^n$ using Rule (iii) of Fig. 2, the new goals are indexed as $b^1_{s.1}, \ldots, b^n_{s.n}$.

[6] The term '*sublinear*' might be more appropriate than *nonlinear* here.

(BU_c) $\forall s\,\forall t_1, t_2 \in \mathit{Term}\big((t_1 =_s t_2) \in B \Rightarrow c/\langle =_s, 1\rangle + c/\langle =_s, 2\rangle = \mathbf{0}\big)$
(the arguments of a unification body goal have complementary capabilities)

(BV_c) Let $v \in \mathit{Var}$ 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
1. $-c/p_1 - \ldots - c/p_k + c/p_{k+1} + \ldots + c/p_n = \mathbf{0}$ (*Kirchhoff's Current Law*)
2. if $k = 0$ and $n > 2$ then $\mathcal{R}\big(\{c/p_1, \ldots, c/p_n\}\big)$
3. if $k \geq 1$ and $n - k \geq 2$ then $\mathcal{R}\big(\{\overline{c/p_1}, c/p_{k+1}, \ldots, c/p_n\}\big)$

where $\mathcal{R}$ is a 'cooperativeness' relation:

$$\mathcal{R}(S) \stackrel{\text{def}}{=} \exists s \in S\big(s < \mathbf{0} \wedge \forall s' \in S \backslash \{s\}\ \big(s' > \mathbf{0}\big)\big)$$

(HV_c) $\forall p \in P_{\mathit{Atom}}\big(\widetilde{h}(p) \in \mathit{Var} \wedge \exists p' \neq p\big(\widetilde{h}(p) = \widetilde{h}(p')\big) \Rightarrow c/p > \mathbf{0}\big)$
(if the symbol at p in h is a variable occurring elsewhere in h, then $c/p > \mathbf{0}$)

(HF_c) $\forall p \in P_{\mathit{Atom}}\big(\widetilde{h}(p) \in \mathit{Fun} \Rightarrow (c(p) > 0$
$\wedge\, \exists! q \in P_{\mathit{Atom}} \exists a \in B(\widetilde{a}(q) = \widetilde{h}(p) \wedge c(p) = c(q) \wedge (c(p) < 1 \Rightarrow c/p = c/q))\big)\big)$
(if the symbol at p in h is a constructor, $c(p) > 0$ and there's exactly one partner in B at q such that $c(p) = c(q)$ (and $c/p = c/q$ if non-exclusive))

(BF_c) $\forall p \in P_{\mathit{Atom}} \forall a \in B\big(\widetilde{a}(p) \in \mathit{Fun} \Rightarrow (c(p) > 0$
$\wedge\, \exists! q \in P_{\mathit{Atom}}(\widetilde{h}(q) = \widetilde{a}(p) \wedge c(p) = c(q) \wedge (c(p) < 1 \Rightarrow c/p = c/q))\big)\big)$
(if the symbol at p in B is a constructor, $c(p) > 0$ and there's exactly one partner at q in h such that $c(p) = c(q)$ (and $c/p = c/q$ if non-exclusive))

(Z_c) $\forall p \in P_{\mathit{Atom}} \forall a \in B\big((\widetilde{h}(p) = * \vee \widetilde{a}(p) = *) \Rightarrow c/p = \mathbf{0}\big)$
(a void path has a zero capability)

(NZ_c) $\forall p \in P_{\mathit{Atom}} \forall a \in B\big((\widetilde{h}(p) \in \mathit{Fun} \cup \mathit{Var} \vee \widetilde{a}(p) \in \mathit{Fun} \cup \mathit{Var}) \Rightarrow c(p) \neq 0\big)$
(a non-void path has a non-zero capability)

Fig. 9. Capability constraints imposed by a clause h `:- |` B

8 The Capability System

Our capability system generalizes the mode system. As suggested earlier, the capability type (say c) of a program or its fragment is a function

$$c : P_{\mathit{Atom}} \rightarrow [-1, +1].$$

The framework is necessarily polymorphic with respect to non-exclusive capabilities because a non-exclusive capability may be split into two or more capabilities. This is why different goals created at runtime should be distinguished using indices.

The following closure conditions of a capability function represent the uniformity of non-exclusive capabilities:

1. $0 < c(p) < 1 \Rightarrow \forall q\big(0 < c(pq) < 1\big)$
2. $-1 < c(p) < 0 \Rightarrow \forall q\big(-1 < c(pq) < 0\big)$

Our capability constraints, shown in Fig. 9, generalizes mode constraints (Fig. 4) without complicating it. Here we have inherited all the notational conventions from the mode system (see Sect. 3.3) and modified them appropriately.

As an example, consider the following program.

```
p_s(X,Y,...) :- | r_s.1(X,Y1,...), p_s.2(X,Y2,...), join_s.3(Y1,Y2,Y).
p_s(X,Y,...) :- | X =_s.1 Y.
join_s(A,A,B) :- | B =_s.1 A.
```

Then the capability constraints they impose include:

1. From the first clause of `p`:
   (a) $-c/\langle \mathtt{p}_s, 1\rangle + c/\langle \mathtt{r}_{s.1}, 1\rangle + c/\langle \mathtt{p}_{s.2}, 1\rangle = \mathbf{0}$ (by (BV_c) applied to `X`)
   (b) $c/\langle \mathtt{r}_{s.1}, 2\rangle + c/\langle \mathtt{join}_{s.3}, 1\rangle = \mathbf{0}$ (by (BV_c) applied to `Y1`)
   (c) $c/\langle \mathtt{p}_{s.2}, 2\rangle + c/\langle \mathtt{join}_{s.3}, 2\rangle = \mathbf{0}$ (by (BV_c) applied to `Y2`)
   (d) $-c/\langle \mathtt{p}_s, 2\rangle + c/\langle \mathtt{join}_{s.3}, 3\rangle = \mathbf{0}$ (by (BV_c) applied to `Y`)
2. From the second clause of `p`:
   (a) $-c/\langle \mathtt{p}_s, 1\rangle + c/\langle \mathtt{=}_s, 1\rangle = \mathbf{0}$ (by (BV_c) applied to `X`)
   (b) $-c/\langle \mathtt{p}_s, 2\rangle + c/\langle \mathtt{=}_s, 2\rangle = \mathbf{0}$ (by (BV_c) applied to `Y`)
   (c) $c/\langle \mathtt{=}_s, 1\rangle + c/\langle \mathtt{=}_s, 2\rangle = \mathbf{0}$ (by (BU_c))
3. From `join`:
   (a) $c/\langle \mathtt{join}_s, 1\rangle > \mathbf{0}$ (by (HV_c) applied to `A`)
   (b) $c/\langle \mathtt{join}_s, 2\rangle > \mathbf{0}$ (by (HV_c) applied to `A`)
   (c) $-c/\langle \mathtt{join}_s, 1\rangle - c/\langle \mathtt{join}_s, 2\rangle + c/\langle \mathtt{=}_s, 2\rangle = \mathbf{0}$ (by (BV_c) applied to `A`)
   (d) $-c/\langle \mathtt{join}_s, 3\rangle + c/\langle \mathtt{=}_s, 1\rangle = \mathbf{0}$ (by (BV_c) applied to `B`)
   (e) $c/\langle \mathtt{=}_s, 1\rangle + c/\langle \mathtt{=}_s, 2\rangle = \mathbf{0}$ (by (BU_c))

In each constraint, the index s is universally quantified. These constraints are satisfiable if (and only if) $c/\langle \mathtt{r}_{s.1}, 1\rangle + c/\langle \mathtt{r}_{s.1}, 2\rangle = \mathbf{0}$. Suppose this can be derived from other constraints. Suppose also that $c/\langle p_{s_0}, 1\rangle = \mathbf{1}$ holds, that is, `p` is initially called with a non-shared, read-only first argument. Then the above set of constraints guarantees $c/\langle p_{s_0}, 2\rangle = \overline{\mathbf{1}}$, which means that the references to `X` distributed to the `r`'s will be fully collected as long as all the `r`'s eventually return the references they have received.

Note that the above constraints (1(a) and several others) and Rule (NZ_c) entail $\mathbf{0} < c/\langle \mathtt{r}_{s.1}, 1\rangle < \mathbf{1}$ and $\mathbf{0} < c/\langle \mathtt{p}_{s.2}, 1\rangle < \mathbf{1}$. That is, these paths are constrained to be non-exclusive paths. It is easy to see that a set of constraints cannot entail a constraint of the form $\mathbf{0} < c/p < \mathbf{1}$ unless some variable is nonlinear.

We have not yet worked out on theoretical results, but conjecture that the following properties hold (possibly with minor modification):

1. The three properties shown in Sect. 3.5, namely (i) degeneration of unification to assignment, (ii) subject reduction, and (iii) groundness.
2. (*Conservation of Constructors*) A reduction does not gain or lose any constructor in the goal clause, with its capability taken into account as its weight.

The Rules (HF_c) and (BF_c) can be relaxed so that the name of the constructor examined in the head can be changed when it is recycled in the body, as long as the constructor comes with an exclusive capability and its arity does not change. When this modification is done, the Conservation of Constructors property should be modified accordingly to allow the changes of names.

This modification is important when computation involves a lot of constants such as numbers. Indeed, some relaxation will be necessary to accommodate arithmetics in our framework in a reasonable way. For instance, to perform local computation such as `Y:=X+2`, it would be unrealistic to obtain constructors `+` and `2` from the parent process and let them escape through `Y`. Rather, we want to allocate and garbage-collect them locally and let `Y` emit an integer constant.[7]

9 Related Work

Relating the family of π-calculi and the CCP formalism has been done as proposals of calculi such as the γ-calculus [33], the ρ-calculus [24] and the Fusion calculus [48], all of which incorporate constraints (or name equation) in some form. The γ-calculus is unique in that it uses procedures with encapsulated states to model concurrency and communication rather than the other way around. The ρ-calculus introduces constraints into name-based concurrency, while constraint-based concurrency aims to demonstrate that constraints alone are adequate for modeling and programming concurrency. The Fusion calculus simplifies the binding operators of the π-calculus using the unification of names. A lesson learned from Constraint Logic Programming [17] is that, even when general constraint

[7] In actual implementations, `+` and `2` will be embedded in compiled code and can be considered zero-resource values.

satisfaction is not intended, formulation in terms of constraints can be more elegant and less error-prone. The simplicity of constraint-based concurrency and the existence of working implementations suggest that encoding all these calculi in constraint-based concurrency would be worthwhile.

In addition to ρ and Fusion, various calculi based on the π-calculus have been proposed, which include Lπ (Local π) [22], the Join calculus [11] and πI (Internal π) [26]. They are aimed at nicer semantical properties and/or better correspondence to programming constructs. Some of the motivations of these calculi are in common – at least at a conceptual level – with the design of constraint-based concurrency with strong moding. For instance, πI restricts communicated data to local names in order to control name scope, and Lπ restricts communicated data to those with output capabilities in order to allow names to act as object identities. Both objectives have been achieved in constraint-based concurrency. Lπ abolished name matching based on the observation that it would be too strong a capability. The counterpart of name matching in constraint-based concurrency is matching with a nonlinear head, which imposes a strong mode constraint that bans the comparison of channels used for bidirectional communication.

In concurrent, logic, and/or functional languages and calculi, a number of type systems to deal with polarities and linearities have been proposed.

In π-calculi and functional languages, Kobayashi proposes a linear type system for the π-calculus [19], which seems to make the calculus close to constraint-based concurrency with linear, moded variables because both linear channels and linear logic variables disallow more than one write access and more than one read access. Turner *et al.* introduce linearity annotation to a type system for call-by-need lambda calculus [35]. All these pieces of work could be considered the application of ideas with similar motivations to different computational models. In concurrent logic programming, the difficulty lies in the treatment of arbitrarily complex information flow expressed using logical variables. Walker discusses types supporting more explicit memory management [50]. Session types [13] shares the same objective with our mode system.

Languages that feature linearity can be found in various programming paradigms. Linear Lisp [4] and Lilac [20] are two examples outside logic programming, while a survey of linear logic programming languages can be found in [23].

There is a lot of work on compile-time garbage collection other than that based on typing. In logic programming, most of the previous work is based on abstract interpretation [14]. Mercury [34] is a logic programming language known for its high-performance and enables compile-time garbage collection using mode and uniqueness declarations [21]. However, the key difference between Mercury and GHC is that the former does not allow non-strict data structures while the latter is highly non-strict.

Message-oriented implementation of Moded Flat GHC, which compiles stream communication into tight control flow between communicating processes, can be thought of as a form of compile-time garbage collection [41][40]. Another technique related to compile-time garbage collection is process fusion by unfold/fold transformation [38], which should have some relationship with deforestation of functional programs.

Janus [27] establishes the linearity property by allowing each variable to occur only twice. In Janus, a reserved unary constructor is used to give a variable occurrence an output capability. Our technique allows both linear and nonlinear variables and distinguishes between them by static analysis, and allows output capabilities to be inferred rather than specified.

Concurrent read accesses under linear typing was motivated by the study on parallel array processing in Moded Flat GHC [42] [25], which again has an independent counterpart in functional programming [29].

10 Conclusions and Future Work

This is the first report on the ongoing project on garbage-free symbolic computation based on constraint-based concurrency.

The sublanguage we propose, namely a strictly linear subset of Guarded Horn Clauses, retains most of the power of the cooperative use of logical variables, and also allows resource sharing without giving up the linguistic-level control over the resource handled by the program.

The capability type system integrates and generalizes the mode system and the linearity system developed and used for Flat GHC. Thanks to its arithmetic and constraint-based formulation, the

type system is kept quite simple. We plan to build a constraint-based type reconstructor in the near future. A challenging issue from the theoretical point of view is the static analysis of the extended occur-check condition. However, we have already been successful in detecting the (useless) unification of identical nonlinear variable as erroneous; if `X` is unified with itself when it has the third occurrence elsewhere, the third occurrence is constrained to have zero capability, which contradicts Rule (NZ_c). Another important direction related to resource-consciousness is to deal with time as well as space bounds. We need to see how type systems developed in different settings to deal with resource bounds [16][9] can relate to our concurrent setting.

Undoubtedly, the primary concern is the ease of programming. Does resource-conscious programming help programmers write correct programs enjoying better properties, or is it simply burdensome? We believe the answer to the former is at least partly affirmative, but to a varying degree depending on the applications. One of the grand challenges of concurrent languages and their underlying theories is to provide a common platform for various forms of non-conventional computing including parallel computing, distributed/network computing, real-time computing, and mobile computing [45]. All these areas are strongly concerned with physical aspects and we hope that a flexible framework with the notion of resources will be a promising starting point towards a common platform.

Acknowledgments Discussions with the members of the programming language research group at Waseda helped the development of the ideas described here. This work is partially supported by Grant-In-Aid for Scientific Research ((C)(2) 11680370, Priority Areas (C)(2)13324050), Ministry of Education.

References

1. Aït-Kaci, H. and Nasr, R., LOGIN: A Logic Programming Language with Built-In Inheritance. *J. Logic Programming*, Vol. 3, No. 3 (1986), pp. 185–215.
2. Ajiro, Y., Ueda, K. and Cho, K., Error-Correcting Source Code. In *Proc. Fourth Int. Conf. on Principles and Practice of Constraint Programming (CP'98)*, LNCS 1520, Springer-Verlag, 1998, pp. 40–54.
3. Ajiro, Y. and Ueda, K., Kima: an Automated Error Correction System for Concurrent Logic Programs. To appear in *Automated Software Engineering*, 2001.
4. Baker, H. G., Lively Linear Lisp—'Look Ma, No Garbage!' *Sigplan Notices*, Vol. 27, No. 8 (1992), pp. 89–98.
5. 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.
6. Chikayama, T., Operating System PIMOS and Kernel Language KL1. In *Proc. Int. Conf. on Fifth Generation Computer Systems 1992 (FGCS'92)*, Ohmsha and IOS Press, Tokyo, 1992, pp. 73–88.
7. Chikayama, T., Fujise, T. and Sekita, D., A Portable and Efficient Implementation of KL1. In *Proc. 6th Int. Symp. on Programming Language Implementation and Logic Programming (PLILP'94)*, LNCS 844, Springer-Verlag, 1994, pp. 25–39.
8. Clark, K. L. and Gregory, S., PARLOG: Parallel Programming in Logic. *ACM. Trans. Prog. Lang. Syst.*, Vol. 8, No. 1 (1986), pp. 1–49.
9. Crary, K. and Weirich, S., Resource Bound Certification. In *Proc. 27th ACM Symp. on Principles of Programming Languages (POPL'00)*, 2000, pp. 184–198.
10. Fujita, H. and Hasegawa, R., A Model Generation Theorem Prover in KL1 Using a Ramified-Stack Algorithm. In *Proc. Eighth Int. Conf. on Logic Programming (ICLP'91)*, The MIT Press, Cambridge, MA, 1991, pp. 535–548.
11. Fournet, C., Gonthier, G. Lévy, J.-J., Maranget, L. and Rémy, D., A Calculus of Mobile Agents. In *Proc. 7th Int. Conf. on Concurrency Theory (CONCUR'96)*, LNCS 1119, Springer-Verlag, 1996, pp. 406–421.
12. Frühwirth, T., Theory and Practice of Constraint Handling Rules. *J. Logic Programming*, Vol. 37, No. 1–3 (1998), pp. 95–138.
13. Gay, S. and Hole, M., Types and Subtypes for Client-Server Interactions. In *Proc. European Symp. on Programming (ESOP'99)*, LNCS 1576, Springer-Verlag, 1999, pp. 74–90.
14. Gudjonsson, G. and Winsborough, W. H., Compile-time Memory Reuse in Logic Programming Languages Through Update in Place. *ACM Trans. Prog. Lang. Syst,* Vol. 21, No. 3 (1999), pp. 430–501.
15. Honda, K. and Tokoro, M., An Object Calculus for Asynchronous Communication. In *Proc. Fifth Conf. on Object-Oriented Programming (ECOOP'91)*, LNCS 512, Springer-Verlag, 1991, pp. 133–147.
16. Hughes, J. and Pareto, L., Recursion and Dynamic Data-structures in Bounded Space: Towards Embedded ML Programming. In *Proc. Fourth ACM SIGPLAN Int. Conf. on Functional Programming (ICFP'99)*, 1999, pp. 70–81.

17. Jaffar, J. and Maher, M. J., Constraint Logic Programming: A Survey. *J. Logic Programming*, Vol. 19–20 (1994), pp. 503–582.
18. Kobayashi, N., Quasi-Linear Types In *Proc. 26th ACM Symp. on Principles of Programming Languages (POPL'99)*, ACM, 1999, pp. 29–42.
19. Kobayashi, N., Pierce, B. and Turner, D., Linearity and the Pi-Calculus. *ACM Trans. Prog. Lang. Syst.*, Vol. 21, No. 5 (1999), pp. 914–947.
20. Mackie, I., Lilac: A Functional Programming Language Based on Linear Logic. *J. Functional Programming*, Vol. 4, No. 4 (1994), pp. 1–39.
21. Mazur, N., Janssens, G. and Bruynooghe, M., A Module Based Analysis for Memory Reuse in Mercury. In *Proc. Int. Conf. on Computational Logic (CL2000)*, LNCS 1861, Springer-Verlag, 2000, pp. 1255–1269.
22. Merro, M., Locality in the π-calculus and Applications to Distributed Objects. PhD Thesis, Ecol des Mines de Paris, 2000.
23. Miller, D., A Survey on Linear Logic Programming. *The Newsletter of the European Network in Computational Logic*, Vol. 2, No. 2 (1995), pp.63–67.
24. Niehren, J. and Müller, M., Constraints for Free in Concurrent Computation. In *Proc. Asian Computing Science Conf. (ACSC'95)*, LNCS 1023, Springer-Verlag, 1995, pp. 171–186.
25. Sakamoto, K., Matsumiya, S. and Ueda, K., Optimizing Array Processing of Parallel KLIC. In *IPSJ Trans. on Programming*, Vol. 42, No. SIG 3(PRO 10) (2001), pp. 1–13 (in Japanese).
26. Sangiorgi, D., π-Calculus, Internal Mobility and Agent-Passing Calculi. *Theoretical Computer Science*, Vol. 167, No. 1–2 (1996), pp. 235–274.
27. Saraswat, V. A., Kahn, K. and Levy, J., Janus: A Step Towards Distributed Constraint Programming. In *Proc. 1990 North American Conf. on Logic Programming (NACLP'90)*, The MIT Press, Cambridge, MA, 1990, pp. 431–446.
28. Saraswat, V. A. and Rinard, M., Concurrent Constraint Programming (Extended Abstract). In *Proc. 17th Annual ACM Symp. on Principles of Programming Languages (POPL'90)*, ACM, 1990, pp. 232–245.
29. Sastry, A. V. S. and Clinger, W., Parallel Destructive Updating in Strict Functional Languages. In *Proc. 1994 ACM Conf. on LISP and Functional Programming*, 1994, pp. 263–272.
30. Shapiro, E. Y., Concurrent Prolog: A Progress Report. *IEEE Computer*, Vol. 19, No. 8 (1986), pp. 44–58.
31. Shapiro, E., The Family of Concurrent Logic Programming Languages. *ACM Computing Surveys*, Vol. 21, No. 3 (1989), pp. 413–510.
32. Shapiro, E. Y., Warren, D. H. D., Fuchi, K., Kowalski, R. A., Furukawa, K., Ueda, K., Kahn, K. M., Chikayama, T. and Tick, E., The Fifth Generation Project: Personal Perspectives. *Comm. ACM*, Vol. 36, No. 3 (1993), pp. 46–103.
33. Smolka, G., A Foundation for Higher-order Concurrent Constraint Programming. In *Proc. First Int. Conf. on Constraints in Computational Logics*, LNCS 845, Springer-Verlag, 1994, pp. 50–72.
34. Somogyi, Z., Henderson, F. and Conway, T., The Execution Algorithm of Mercury, An Efficient Purely Declarative Logic Programming Language. *J. Logic Programming*, Vol. 29, No. 1–3 (1996), pp. 17–64.
35. 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.
36. Ueda, K., Guarded Horn Clauses. ICOT Tech. Report TR-103, ICOT, Tokyo, 1985. Also in *Logic Programming '85*, Wada, E. (ed.), LNCS 221, Springer-Verlag, 1986, pp. 168–179.
37. Ueda, K., Guarded Horn Clauses. D. Eng. Thesis, Univ. of Tokyo, 1986.
38. Ueda, K. and Furukawa, K., Transformation Rules for GHC Programs. In *Proc. Int. Conf. on Fifth Generation Computer Systems 1988 (FGCS'88)*, ICOT, Tokyo, 1988, pp. 582–591.
39. Ueda, K. and Chikayama, T. Design of the Kernel Language for the Parallel Inference Machine. *The Computer Journal*, Vol. 33, No. 6 (1990), pp. 494–500.
40. Ueda, K. and Morita, M., Message-Oriented Parallel Implementation of Moded Flat GHC. *New Generation Computing*, Vol. 11, No. 3–4 (1993), pp. 323–341.
41. 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.
42. Ueda, K., Moded Flat GHC for Data-Parallel Programming. In *Proc. FGCS'94 Workshop on Parallel Logic Programming*, ICOT, Tokyo, 1994, pp. 27–35.
43. Ueda, K., Experiences with Strong Moding in Concurrent Logic/Constraint Programming. In *Proc. Int. Workshop on Parallel Symbolic Languages and Systems (PSLS'95)*, LNCS 1068, Springer-Verlag, 1996, pp. 134–153.
44. Ueda, K., *klint* — Static Analyzer for KL1 Programs. Available from `http://www.icot.or.jp/ARCHIVE/Museum/FUNDING/funding-98-E.html`, 1998.
45. Ueda, K., Concurrent Logic/Constraint Programming: The Next 10 Years. In *The Logic Programming Paradigm: A 25-Year Perspective*, Apt, K. R., Marek, V. W., Truszczynski M., and Warren D. S. (eds.), Springer-Verlag, 1999, pp. 53–71.

46. Ueda, K., Linearity Analysis of Concurrent Logic Programs. In *Proc. Int. Workshop on Parallel and Distributed Computing for Symbolic and Irregular Applications*, Ito, T. and Yuasa, T. (eds.), World Scientific, 2000, pp. 253–270.
47. van Emden, M. H. and de Lucena Filho, G. J., Predicate Logic as a Language for Parallel Programming. In *Logic Programming*, Clark, K. L. and Tärnlund, S. -Å. (eds.), Academic Press, London, 1982, pp. 189–198.
48. Victor, B., The Fusion Calculus: Expressiveness and Symmetry in Mobile Processes, PhD Thesis, Uppsala Univ., 1998.
49. Wadler, P., Linear Types Can Change the World! In *Prof. IFIP TC2 Working Conf. on Programming Concepts and Methods*, Broy, M. and Jones, C. (eds.), North-Holland, 1990, pp. 347–359.
50. Walker, D. P., Typed Memory Management. PhD thesis, Cornell Univ., 2001.