In T. Ito and A. Yonezawa (Eds.),
Theory and Practice of Parallel Programming,
LNCS 907, Springer, 1995, pp. 356–368.

I/O Mode Analysis in Concurrent Logic Programming

Kazunori Ueda

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

Abstract. This paper briefly reviews concurrent logic programming and the I/O mode system designed for the concurrent logic language Flat GHC. The mode system plays fundamental rôles both in programming and implementation in almost the same way as type systems do but in different respects. It provides us with the information on how data are generated and consumed and thus the view of "data as resources". It statically detects bugs resulting from ill-formed dataflow and advocates the "programming as wiring" paradigm. Well-modedness guarantees the safety of unification, the basic operation in concurrent logic programming. Information on the numbers of access paths to data can be obtained by slightly extending the framework, which can be used for compile-time garbage collection and the destructive update of structures.

1 Concurrent Logic Programming

Concurrent logic programming was born around 1980 from the study of the parallel execution of logic programs, and became an important paradigm for concurrent programming in its own right. Relational Language [3] was the first to appear in the form of a complete programming language. Guarded Horn Clauses (GHC) was designed in the Fifth Generation Computer Project in 1984 [11], after thorough examination of its predecessors Concurrent Prolog [9] and an early version of PARLOG [4]. Because of its simplicity, GHC was soon accepted in the Project as the base of KL1, the full-fledged kernel language for the Parallel Inference Machine. GHC was soon subsetted to Flat GHC with a simpler guard construct, and the design of KL1 started based on Flat GHC. While GHC was designed as a concurrent language that did not address *how* programs should be executed, KL1 was designed as a parallel language in which programmers could specify what processor should execute what processes and at what priorities [13]. For the detailed history of the kernel language design in the Fifth Generation Computer Project, the readers are referred to [14].[1]

Several more concurrent logic programming languages were proposed in the past decade, but the difference between all those languages is rather small as

[1] An efficient KL1-to-C compiler system for Unix-based general-purpose computers, named KLIC, can be obtained via anonymous ftp from `ftp.icot.or.jp`.

far as the constructs for reactive concurrent programming are concerned—it's definitely smaller than the difference between CSP, CCS and ACP. All those languages feature asynchronous communication using single-assignment variables which people call *logical* variables. All the languages after (and including) Concurrent Prolog supported *incomplete messages*, a bidirectional communication technique using a single stream, and dynamic process creation and reconfiguration. Although they do not support processes as first-class objects, they support dynamic creation of processes and dynamic creation of streams connected to the processes. Since streams are first-class in concurrent logic programming and can freely passed from one process to another, those languages have effectively supported "mobile" processes since early 1980's.

2 How Processes Communicate

In concurrent logic programming, processes communicate by observing and generating substitutions, namely bindings between variables and their values. Consider an I/O process, say `io(S)`, which models a CRT display terminal and receives a stream `S` of I/O commands such as `get(C)` and `put(63)`. The stream `S` can be represented as a list of `get` and `put` commands, so `S` may be instantiated to a list `[put(63),get(C),get(C'), ...]`, which means that the I/O process should display "?" (63 in ASCII) and then read two characters. The same list may be constructed incrementally as $\texttt{S} = \texttt{[put(63)|S}'\texttt{]}$, $\texttt{S}' = \texttt{[get(C)|S}''\texttt{]}$, $\texttt{S}'' = \texttt{[get(C}'\texttt{)|S}'''\texttt{]}$, and so on. The I/O process will interpret these commands in the order they appear in the stream. If you key in "`A`" and then "`B`", it will generate the bindings $\texttt{C} = \texttt{65}$ and $\texttt{C}' = \texttt{66}$.

The observation of bindings is done by matching (one-way unification) and the generation is done by (two-way) unification. Interestingly, this algebraic account can be restated logically: Bindings can be viewed as equality constraints on the values of variables, matching can be viewed as the checking (*asking*) of whether one constraint implies another, and unification can be viewed as the publication (*telling*) of a constraint. This view was first given by Maher [6] and studied extensively by Saraswat [7] in a generalized setting of *concurrent constraint programming* (CCP).

3 GHC as a Model of Concurrency

Although the family of concurrent logic languages [10] was initially proposed as a tool for programming rather than a model of concurrency, its essence is simple enough to be viewed as a model of concurrency. The somewhat simplified syntax of GHC, which ignores guard goals, is given in Figure 1. The syntax is rule-based rather than expression-based simply for historical and practical reasons.

Rule (1) in concerned with indeterminate choice. Rule (2) is concerned with information receiving (at A, called the *head*) and subsequent reduction to B (the *body*). It implicitly expresses the hiding of variables occurring only in B

as well. Information receiving is done by matching A with a goal (say G) to be reduced, where only the variables in A can be instantiated. If A and G are not unifiable, G cannot be reduced using the program clause A `:-` B. If they are unifiable but only by instantiating G, the matching is suspended until G is sufficiently instantiated by the execution of other unification goals. Thus the matching between A and G may involve equality checking and synchronization as well as information receiving.

Rule (3) is concerned with parallel composition. Rule (4) says that each goal is either a unification goal to publish information or a non-unification goal to be reduced using program clauses. The initial multiset of goals to be reduced is given by a goal clause (Rule (7)).

The CCP framework considered a lot more combinators, but we have made sure that the above (small) set of constructs, which corresponds to indeterminate CCP with *ask* and (eventual) *tell*, is usually enough.

$$
\begin{array}{rll}
\text{(program) } P & ::= \text{set of } C\text{'s} & (1)\\
\text{(program clause) } C & ::= A \;\texttt{:-}\; B & (2)\\
\text{(body) } B & ::= \text{multiset of } G\text{'s} & (3)\\
\text{(goal) } G & ::= T_1 \;\texttt{=}\; T_2 \quad | \quad A & (4)\\
\text{(non-unification atom) } A & ::= p\texttt{(}T_1\texttt{,}\ldots\texttt{,}T_n\texttt{)}, \quad p \neq \text{`}\texttt{=}\text{'} & (5)\\
\text{(term) } T & ::= \text{(as in first-order logic)} & (6)\\
\text{(goal clause) } Q & ::= \;\texttt{:-}\; B & (7)
\end{array}
$$

Fig. 1. The (simplified) syntax of GHC

4 Channels as First-Class Objects

What makes concurrent logic/constraint programming unique is that communication channels (streams) are first-class data structures (lists). Ordinary list operations are used for sending and receiving messages, while in other models of concurrency, channels and operations on channels are usually provided as *built-in* language constructs. This uniformity in the language constructs was achieved by the ability of (concurrent) logic languages to deal with partial information such as lists with uninstantiated tails. The uniformity contributes much to the simplicity of the semantical framework and program analysis.

Channels as data structures turned out to be quite flexible. Various communication protocols have been used in actual programming, including streams of streams (of streams ...), demand-driven communication, and incomplete messages with reply boxes. In addition, difference lists invented in logic programming turned out to be extremely useful also in concurrent logic programming. Difference lists allow us to form fragments of a message sequence independently—in an arbitrary order or in parallel—, connect them later on, and feed the result to the receiver process.

5 Experiences with GHC/KL1 Programming

Hundreds of thousands of lines of GHC/KL1 programs have been written inside and outside the Fifth Generation Computer Project. The applications include an operating system for the Parallel Inference Machine (PIMOS), a parallel theorem prover (MGTP) that discovered a new fact in finite algebra, genetic information processing, and so on.

People found the communication and synchronization mechanisms of GHC/KL1 quite natural. They were seldom bothered by synchronization bugs[2] and gradually learned to model their problems in an object-oriented manner using concurrent processes and streams. Although writing correct *concurrent* programs was not so hard, writing efficient *parallel* programs turned to be a quite separate issue. A visual performance debugger played an important rôle.

We have found that logical variables are normally used for cooperative rather than competitive communication. Most variables in a run-time configuration have exactly two occurrences each, in which case they are used as *cables* for one-to-one communication. This is closely related to the fact that (surprisingly) many program clauses in Prolog and concurrent logic programs are *linear* in the sense that each variable in a clause occurs exactly twice. Consider `append` in Prolog:

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

The two clauses are linear in our sense, though they are not *left-linear* in that some variables occur more than once in the heads. On the other hand, the GHC counterpart of `append` is both linear and left-linear:

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

Some GHC variables occur three or more times, most of which are used to distribute ground terms such as numbers or shared global data. Clauses containing such variables will generate variables with three or more occurrences in a run-time configuration, which will be used as *hubs* for one-to-many communication.

Whether communication is one-to-one or one-to-many, unification goals will always succeed as long as they are used for cooperative communication. Nevertheless, programmers have often got an error message "unification failure", which indicates that variables are inadvertently used for non-cooperative communication. For instance, goals `X = []` and `X = [H|T]` will noncooperatively attempt to

[2] Bugs regarding the causality of information flow are a higher-level problem that often arises in, for example, programs dealing with circular process structures.

bind `X` to different values. A static means to ensure the security of unification was strongly desired.

Note that unification failure is very similar to division-by-zero; for instance, a Pascal statement `X := 5 div 0` can be viewed as insisting that different values, `X * 0` (= `0`) and `5`, should become identical.

6 Mode System for Flat GHC

To guarantee the security of unification statically, a mode system for Flat GHC was designed in 1990 [12]. This and subsequent sections will introduce it rather informally but from various aspects. We refer the readers to [15] for the precise and detailed description.

As a metaphor, consider two electric devices connected by a cable for conveying signals. The cable may have a structure such as an array of wires. As the nature of a cable, its both ends, viewed from outside the cable, should have opposite polarity structures. That is, a signal given to one end of a wire will go out from the other end, though different wires may have different directionalities. The plug at an end of a cable will be inserted into the socket on a device. For the plug and the socket to be compatible, they should again have opposite polarity structures when viewed from outside. These constraints together imply that the two sockets on the two devices should have opposite polarity structures.

Our mode system infers such polarity structures of goal arguments in essentially the same manner. (To be exact, it infers the polarity structures of the arguments of *predicates* defining the behavior of goals, rather than dealing with individual goals themselves.) A mode represents the polarity structure of goal arguments. While an electric cable has an array structure, a logical variable may be instantiated to a nested structure with a complicated communication protocol. To deal with such protocols, a mode is 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 not strings of argument positions; instead they are strings of $\langle symbol, argument\text{-}position\rangle$ pairs in order to be able to specify positions in data structures that are yet to be formed. For instance, when the argument `S` of an I/O server `io(S)` is instantiated to `[put(63),get(C),get(C′), ...]`, the symbol at

$$\langle \mathtt{io}, 1\rangle\langle \mathtt{.}, 2\rangle\langle \mathtt{.}, 2\rangle\langle \mathtt{.}, 1\rangle\langle \mathtt{get}, 1\rangle \quad (\text{`}\mathtt{.}\text{' is a list constructor}),$$

which did not exist initially, turns out to be `C′`. Formally, the sets of paths for specifying positions in terms and atoms 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 *Atom* are the sets of function and predicate symbols, respectively, and N_p and N_f are the sets of positive integers up to and including the arities of p and f, respectively.

Mode analysis tries to find a mode (i.e., polarity structure) 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. Function symbols 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.

For example, a variable occurring exactly twice, both in the body of a clause, constrains the two occurrences to have opposite polarity structures—as an electric cable does. A variable occurring exactly once in the head and exactly once in the body constrains the two occurrences to have the same polarity structure, because the head can be compared to a device viewed from *inside*, not from outside. The two arguments of a unification goal are constrained to have opposite polarity structures, because a unification goal can be viewed as an adaptor that simply connects its two arguments and transmits information from one side to the other. A function symbol occurring in a body goal represents a signal to be input to the goal, while a function symbol in a head represent a signal to be received. In either case, the path where a function symbol occurs is constrained to have the mode *in*. All these constraints are merged to find well-modings for user-defined goals.

As an example, consider the program for indeterminate stream merging:

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

The first clause has two occurrences of Y, one in the head and the other in the body, so they should have the same mode. The two occurrences of Z should have the same mode as well. On the other hand, the occurrences of Z and Y in the body should have opposite modes because they occur in the opposite sides of a unification goal. So we obtain the constraint

$$\forall q \in P_{Term}\big(m(\langle \mathtt{merge}, 2\rangle q) \neq m(\langle \mathtt{merge}, 3\rangle q)\big) \ . \tag{1}$$

Also, because the function symbol [] occurs in the first argument, we have

$$m(\langle \mathtt{merge}, 1\rangle) = \mathit{in} \ . \tag{2}$$

Similarly, from the second clause we have

$$\forall q \in P_{Term}\big(m(\langle \mathtt{merge}, 1\rangle q) \neq m(\langle \mathtt{merge}, 3\rangle q)\big) \ . \tag{3}$$

We have $m(\langle \mathtt{merge}, 2\rangle) = \mathit{in}$ as well, but this is implied by (1), (2), and (3).

The third clause contains the goal Z0=[A|Z]. Let this be the kth unification goal in the program. (We suffix unification goals here because we allow different unification goals to have different modes.) Then we have $m(\langle \mathtt{=}_k, 2\rangle) = \mathit{in}$ because of the list constructor, and consequently $m(\langle \mathtt{=}_k, 1\rangle) = \mathit{out}$. The latter constraint is conveyed by `Z0` and we have $m(\langle \mathtt{merge}, 3\rangle) = out$, but this is again implied by the constraints we already have. The constraint imposed by the list constructor in the head is exactly the same as (2). The variable `A` occurs twice, once in the head and once in the body, so we have

$$\begin{aligned}\forall q \in P_{Term}\big(\, & m(\langle \mathtt{merge}, 1\rangle\langle\,.\,,1\rangle q) = m(\langle \mathtt{=}_k, 2\rangle\langle\,.\,,1\rangle q) \\ & \neq m(\langle \mathtt{=}_k, 1\rangle\langle\,.\,,1\rangle q) = m(\langle \mathtt{merge}, 3\rangle\langle\,.\,,1\rangle q)\big) \ ,\end{aligned}$$

namely

$$\forall q \in P_{Term}\big(m(\langle \mathtt{merge}, 1\rangle\langle\,.\,,1\rangle q) \neq m(\langle \mathtt{merge}, 3\rangle\langle\,.\,,1\rangle q)\big) \ .$$

This happens to be implied by (3). The variable `X` occurs at $\langle \mathtt{merge}, 1\rangle\langle\,.\,,2\rangle$ in the head and at $\langle \mathtt{merge}, 1\rangle$ in the body, so we have

$$\forall q \in P_{Term}\big(m(\langle \mathtt{merge}, 1\rangle\langle\,.\,,2\rangle q) = m(\langle \mathtt{merge}, 1\rangle q)\big) \ . \tag{4}$$

The constraint imposed by the variable `Y` is a tautology

$$\forall q \in P_{Term}\big(m(\langle \mathtt{merge}, 2\rangle q) = m(\langle \mathtt{merge}, 2\rangle q)\big) \ .$$

Lastly, the variable `Z` occurs twice as well, and we have

$$\begin{aligned}\forall q \in P_{Term}\big(\, & m(\langle \mathtt{merge}, 3\rangle q) \neq m(\langle \mathtt{=}_k, 2\rangle\langle\,.\,,2\rangle q) \\ & \neq m(\langle \mathtt{=}_k, 1\rangle\langle\,.\,,2\rangle q) = m(\langle \mathtt{merge}, 3\rangle\langle\,.\,,2\rangle q)\big) \ ,\end{aligned}$$

namely

$$\forall q \in P_{Term}\big(m(\langle \mathtt{merge}, 3\rangle q) = m(\langle \mathtt{merge}, 3\rangle\langle\,.\,,2\rangle q)\big) \ ,$$

which is, again, implied by the constraints we already have.

We can obtain constraints from the fourth clause a similar way, but all those constraints turn out to be implied by the constraints obtained from other clauses. To summarize, the conjunction of the four constraints (1), (2), (3), and (4) represents all what are imposed by the `merge` program.

There are many good reasons to use a constraint-based framework. First of all, because it is inherently incremental, program modules making up a large program can be analyzed separately so that the results may be merged later. Secondly, the system provides a unified framework for mode inference, mode declaration and mode checking, where mode declaration is simply a mode constraint given by a programmer. Thirdly, it can deal with generic modes quite naturally. A predicate of a generic nature may have many possible well-modings, in which case the analysis will return the principal (i.e., most-general) mode. In these respects, our mode system has much in common with the type system of ML, though their purposes are quite different. A generic mode is represented as a finite set of (satisfiable) mode constraints that all well-modings must satisfy, as we saw in the `merge` example.

Mode analysis as constraint solving is decidable and efficient. As mentioned above, a variable occurring in a linear and left-linear clause imposes a binary constraint; a unification goal imposes a binary constraint; and each occurrence of a function symbol imposes a unary constraint. A set of binary and unary mode constraints can be represented as a feature graph (feature structures with cycles) 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.

The union (i.e., conjunction) of two sets of constraints can be computed efficiently as unification over feature graphs. The cost of the unification is almost proportional to the size of the program and the complexity of the data structures used (in terms of the size of the grammar to generate them). The *size* of the data structures that can be generated does not matter. A variable with more than two occurrences imposes a non-binary constraint, but in most cases they can be reduced to unary or binary constraints using other constraints. There are non-binary constraints that may require generate-and-test search. However, they are rather exceptional and our solution is *not* to implement generate-and-test search but let programmers declare the modes of predicate arguments involving such variables.

Concurrent languages Doc [5], $\mathcal{A}'\mathcal{UM}$ [16] and Janus [8] attempt to ensure well-modedness by allowing each variable to occur only twice and letting programmers distinguish between input and output occurrences using annotations. These annotations can be regarded as mode declarations and contribute to readability and/or ease of compilation, but they are optional because they can be inferred in principle. In addition, our mode system can naturally deal with variables with three or more occurrences.

Another good news with our mode system is that, with slight extension, it can statically distinguish between paths that are used only for one-to-one communication and paths that may be used for one-to-many communication. In other words, we can analyze whether a datum occurring at some path has exactly one reader or possibly many. This information is fundamental for memory management, as we will see later.

7 Fundamental Properties of the Mode System

The mode system guarantees the following basic properties. Let P be a program, Q a goal clause containing the multiset of goals to be reduced, and m be a mode. Let $P : m$ and $Q : m$ mean that m is a well-moding of P and Q, respectively.

Lemma 1. *If $Q : m$ and Q contains a unification goal $t_1 = t_2$, then at least one of t_1 and t_2 is a variable.*

The lemma says that a unification goal in a well-moded goal clause will not fail unless the occur check does not fail.

Theorem 2 (Subject reduction theorem). *Suppose $P : m$, $Q : m$, and Q is reduced in one step to Q' using P. Then $Q' : m$ holds unless the extended occur check fails in that reduction.*

The theorem is not obvious because of its *unless* condition. The extended occur check is the occur check that additionally excludes the unification between identical variables. In an electric device metaphor, unifying identical variables corresponds to connecting the two ends of a cable together and forming a self-loop not connected to anywhere else. This is certainly an operation which is not likely to be performed in meaningful programs.

From Lemma 1 and Theorem 2, the following important corollary follows:

Corollary 3 (Safety of unification). *Suppose $P : m$ and $Q : m$. Then the execution of Q under P does not cause unification failure (failure of unification goals) unless the extended occur check fails.*

Another basic property is:

Theorem 4 (Groundness theorem). *Suppose $P : m$, $Q : m$, and Q has been reduced (in one or more steps) to an empty clause, using P, without causing the failure of the extended occur check. Then all the variables in Q are instantiated to ground (i.e., variable-free) terms.*

This theorem says that if the termination of Q can be proved, the groundness property comes for free. The proofs of all the above results can be found in [15].

8 Implication of the Mode System

The mode system of Flat GHC has been exhibiting a vast influence on programming, implementation, and language design.

8.1 Programming

Introducing a mode system means to subset a language, but we have felt almost no loss of expressive power. The mode system can deal with (i) complex data structures such as streams of incomplete messages and streams of streams, (ii) difference lists, and (iii) mutual recursion, all with no difficulty. A difference list is considered a fragment of a longer list whose tail will eventually be connected to another difference list or a (complete) list. Hence its head and tail will be constrained to have exactly opposite modes.

The mode system provides us with useful debugging information and programming guidelines. Firstly, it is useful for the detection of careless mistakes like the misspelling of variable names. A variable occurring only once can be compared to a cable not connected to anywhere else and very often indicates a program error. Such an occurrence (say at p) imposes a strong mode constraint in our mode system, namely $\forall q \in P_{Term}\big(m(pq) = in\big)$ or $\forall q \in P_{Term}\big(m(pq) = out\big)$, depending on whether the occurrence is in the head or the body. This strong constraint is highly likely to be incompatible with the other mode constraints and cause a mode error. We have found that even without a mode system, many careless mistakes can be detected by simply counting the number of occurrences of each variable; it is as useful as checking whether there are unplugged sockets or dangling cables before using electric devices.

Secondly, the mode system advocates the "programming as wiring" paradigm, or equivalently, programming with linear clauses. (A variable in a linear clause can be regarded as a cable that wires two atoms.) This programming style leads to more generic well-modings and also encourages "structured programming" in terms of dataflow. We found that this style is less error-prone than allowing unrestrictive use of variables with more occurrences. For instance, we have programmed operations on self-adjusting binary trees and unification over feature graphs, both using processes and streams (rather than records and pointers). The reconfiguration of processes they involve is rather complicated, but debugging was not so hard.

Thirdly, the mode system encourages the graceful termination of processes; that is, a process cannot discard its arguments upon termination if it contains variables to be instantiated by that process. Thanks to graceful termination, we can prove that a well-moded terminating program instantiates all the variables to ground terms (Theorem 4). In stream programming, this means that all streams will be closed upon termination.

8.2 Implementation

While unification is bidirectional by nature, mode analysis determines the direction of dataflow at compile time. This information is particularly useful in the distributed implementation of unification goals. Another important aspect is that mode analysis will make the use of native code more realistic. Without it, the compiled code must prepare for exceptional situations that will not happen in normal programs.

Compile-time distinction between one-to-one and possibly one-to-many communication provides fundamental information for memory management. Since a compiler can now pinpoint when each datum used in one-to-one communication becomes unnecessary, it can be reclaimed (compile-time garbage collection) or destructively updated to create a new datum. Update-in-place is extremely important for the efficient support of arrays. Previous implementations of KL1 featured a one-bit reference counting scheme [2]. The reference counting worked quite efficiently on Parallel Inference Machines, but a static scheme is clearly more desirable on stock microprocessors.

Analysis of the mode and the number of receivers enables a sophisticated implementation technique that we call *message-oriented implementation* [12] [15]. Message-oriented implementation compiles unification for stream communication into low-latency message passing, which is particularly beneficial to programs in which processes suspend frequently and incur much process switching overhead.

8.3 Language Design

We are currently exploring the possibility of using concurrent logic languages for (massively) parallel array processing. KL1 supports data structures called vectors (one-dimensional arrays) and provides several vector operations. However, the mode analysis tells us some of them have more generic modes than others. An operation with a more generic mode is considered more fundamental. For element access, for instance, the most basic operation turns out to be

```
set_vector_element(V, I, E, NewE, NewV) .
```

This operation receives an array `V` and the index value `I`, and returns through `E` the `I`th value of `V`. In addition, it returns through `NewV` an array which is identical to `V` except that the `I`th element is replaced by `NewE`.

Interestingly, this seemingly "combined" operation is more generic than just either reading or setting an array element. The reason is that it keeps unchanged the number of references to the whole original array and its elements. For instance, the above `set_vector_element` consumes one reference `V` to the whole array and one reference `NewE` to the new element, and generates a reference `E` to the old `I`th element and a reference `NewE` to the updated array. Thus the operation preserves the number of access paths to any of the elements in the original or updated array. In contrast, an access operation that simply returns a reference to the `I`th element, such as Prolog's `arg`, will impose stronger mode constraints because it loses access paths to the other elements.

The moral of the above result is that in a moded framework, data have an aspect of *resources* whose access paths should not be copied or discarded freely. An array element should, by default, be *removed* from the array once accessed, and the resulting blank should be filled with another value. When performing some operation on an array element and storing the result back to the array, the blank can be tentatively filled with a variable which will be instantiated to the result value. Another typical example of data as resources is a bidirectional communication stream, which could be passed to another process but should not be copied.

An exception to the above principle is that *read-only* arrays, namely arrays whose elements are (or are to be instantiated to) ground terms, can be accessed without removing their elements. Because ground terms are read-only, the principle of cooperative communication allows them to be freely copied or pointed to by a new read-only reference.

Other generic array operations include splitting an array, concatenating two arrays, changing the dimensionality or the shape of an array, and exchanging two elements in an array. Note that they all preserve the number of access paths to elements. All those operations except concatenation are constant-time operations as long as the original arrays are single-reference arrays. Concatenation can be done in constant time as well, provided the two arrays to be concatenated happens to lie next to each other.

The "data as resources" paradigm clearly has some connection with the "propositions-as-resources" view of Linear Logic and also with Linear Lisp [1], though the precise connection is yet to be studied.

9 Conclusions

The static mode system for concurrent logic programming has been described informally but from various aspects. The mode system plays fundamental rôles both in programming and implementation in almost the same way as type systems do but in different respects. It also brings two paradigms, data as resources and programming as wiring, into the language. Mode analysis is quite efficient and can be done incrementally.

In previous implementations, I/O modes and the number of access paths were checked either at runtime or analyzed by abstract interpretation. The mode system we propose is much simpler and more systematic. We do not claim that the mode system can completely replace abstract interpretation. For instance, the time-of-call/exit properties of goals such as the instantiation states of their arguments, which in general depend on the scheduling of goals, can be obtained only by abstract interpretation. However, sophisticated program analysis should benefit from the basic information provided by the mode system.

Programmers may find the rigid handling of "data as resources" enforced by the mode system rather awkward, but we believe that the benefit of the mode system in programming and implementation will more than make up the rigidness in most applications of concurrent logic programming.

References

1. Baker, H. G., Lively Linear Lisp—'Look Ma, No Garbage!' *Sigplan Notices*, Vol. 27, No. 8 (1992), pp. 89–98.
2. Chikayama, T. and Kimura, Y., Multiple Reference Management in Flat GHC. In *Proc. 4th Int. Conf. on Logic Programming*, MIT Press, Cambridge, MA, 1987, pp. 276–293.
3. Clark, K. L. and Gregory, S., A Relational Language for Parallel Programming. In *Proc. ACM Conf. on Functional Programming Languages and Computer Architecture*, ACM, 1981, pp. 171–178.
4. Clark, K. L. and Gregory, S., PARLOG: Parallel Programming in Logic. *ACM. Trans. Prog. Lang. Syst.*, Vol. 8, No. 1 (1986), pp. 1–49.
5. Hirata, M., Programming Language Doc and Its Self-Description or, $X = X$ is Considered Harmful. In *Proc. 3rd Conf. of Japan Society of Software Science and Technology*, 1986, pp. 69–72.
6. Maher, M. J., Logic Semantics for a Class of Committed-Choice Programs. In *Proc. Fourth Int. Conf. on Logic Programming*, MIT Press, Cambridge, MA, 1987, pp. 858–876.
7. Saraswat, V. A. and Rinard M., Concurrent Constraint Programming (Extended Abstract). In *Conf. Record of the Seventeenth Annual ACM Symp. on Principles of Programming Languages*, ACM, 1990, pp. 232–245.

8. 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.), MIT Press, 1990, pp. 431–446.
9. Shapiro, E. Y., A Subset of Concurrent Prolog and Its Interpreter. ICOT Tech. Report TR-003, Institute for New Generation Computer Technology, Tokyo, 1983.
10. Shapiro, E., The Family of Concurrent Logic Programming Languages. *Computing Surveys*, Vol. 21, No. 3 (1989), pp. 413–510.
11. Ueda, K., Guarded Horn Clauses. ICOT Tech. Report TR-103, ICOT, Tokyo, 1985. Also in *Logic Programming '85*, Wada, E. (ed.), Lecture Notes in Computer Science 221, Springer-Verlag, Berlin Heidelberg, 1986, pp. 168–179.
12. Ueda, K. and Morita, M., A New Implementation Technique for Flat GHC. In *Proc. Seventh Int. Conf. on Logic Programming*, The MIT Press, Cambridge, MA, 1990, pp. 3–17.
13. 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.
14. Ueda, K., The Fifth Generation Project: Personal Perspectives, *Commun. ACM*, Vol. 36, No. 3 (1993), pp. 65–76.
15. 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.
16. Yoshida, K. and Chikayama, T., $\mathcal{A}'\mathcal{UM}$ — A Stream-Based Concurrent Object-Oriented Language, in *Proc. Int. Conf. on Fifth Generation Computer Systems 1988*, ICOT, Tokyo, 1988, pp. 638–649. Also in *New Generation Computing*, Vol. 7, No. 2–3 (1990), pp. 127–157.

This article was processed using the LaTeX macro package with LLNCS style