Introduction to KL1

Takashi Chikayama

ICOT

Foreword

A programming language called KL1 was designed to make writing and following improvements of parallel processing programs easier. Its design is based on a concurrent logic programming language Guarded Horn Clauses. The language is used as the common kernel language for parallel inference machines, which are under development in the Japanese Fifth Generation Computer Systems project. The operating system of the parallel inference machines and various application programs are described in this language.

This document is intended to give a brief description the language specification and some of the commonly used programming techniques of the language KL1 to non-experts of the area. It is an almost literal translation of a tutorial text (in Japanese) given in the Japanese domestic Logic Programming Conference '91 by the author himself. It is his great pleasure if this document could help the readers understand the language better.

August 1991
Takashi Chikayama

Contents

1 Language Overview 4

2 Execution Mechanism of KL1 6
- 2.1 Program Syntax and Basic Execution Mechanism . . . 6
  - 2.1.1 A Simple Example . . . 6
  - 2.1.2 Program Execution . . . 7
  - 2.1.3 Unification . . . 8
  - 2.1.4 Syntax of Clauses and Basic Execution Mechanism . . . 8
  - 2.1.5 Concurrency, Communication, Synchronization . . . 9
  - 2.1.6 More on Execution Mechanism . . . 10
- 2.2 Characteristics of KL1 . . . 11
  - 2.2.1 When Compared with Procedural Languages . . . 11
  - 2.2.2 When Compared with Functional Languages . . . 11
  - 2.2.3 When Compared with Other Logic Programming Languages . . . 12
  - 2.2.4 When Compared with Object-Oriented Languages . . . 12

3 Data Types and Primitive Operations 13
- 3.1 Variables without Specified Types . . . 13
- 3.2 Atomic Data . . . 13
  - 3.2.1 Symbolic Atoms . . . 14
  - 3.2.2 Numbers . . . 14
- 3.3 Structured Value . . . 15
  - 3.3.1 Vectors . . . 16
  - 3.3.2 Strings . . . 16
  - 3.3.3 Conses . . . 16
  - 3.3.4 Incomplete Data Structures . . . 17
  - 3.3.5 Unification of Structured Data . . . 17
  - 3.3.6 Identity of Structures and Variables . . . 18

4 Processes and Stream Communication 20
- 4.1 Notational Conventions . . . 20
  - 4.1.1 Denoting Guard Unification within Head . . . 20
  - 4.1.2 Omitting the Goal "`true`" . . . 21
- 4.2 Processes . . . 21
  - 4.2.1 Processes of KL1 . . . 21
  - 4.2.2 Summing Up Elements of a List . . . 22

4.2.3 List of Natural Numbers . . . 23
4.3 Stream Communication . . . 23
4.3.1 Combining Processes . . . 24
4.3.2 Stream Communication . . . 25
4.3.3 Process State . . . 26
4.3.4 Structured Messages . . . 26

5 Process Networks **28**
5.1 Filters . . . 28
5.2 Concatenating Streams . . . 29
5.3 Merging Streams . . . 31
5.4 Built-in Mergers . . . 32
5.5 Dispatchers . . . 34
5.6 Servers . . . 34

6 Specifying Physical Execution **36**
6.1 Principles of Specification of Parallel Execution . . . 36
6.1.1 Programs Specify Load Distribution . . . 36
6.1.2 Separating Concurrency and Parallelism . . . 37
6.2 Goal Distribution . . . 37
6.3 Goal Priority . . . 38
6.4 Clause Priority . . . 38

Chapter 1

Language Overview

KL1 is a programming language designed to make writing and following improvements of parallel processing programs easier [8]. Its design is based on a concurrent logic programming language Guarded Horn Clauses (GHC) [7]. The language is used as the common kernel language for parallel inference machines (PIMs), which are under development in the Japanese Fifth Generation Computer Systems (FGCS) project. The operating system of the PIMs [2] and various application programs are described in this language.

The characteristics of language KL1 might be described by a single sentence:

> **KL1 is a concurrent logic programming language based on Guarded Horn Clauses**

which is probably sufficient for experts in this area. This, however, might look only like a sequence of jargons to those who do not have expert knowledge on this not-so-popular area of computer science. A sentence like:

> **KL1 is a language for parallel symbolic processing**

might be a better description in more common words.

In symbolic processing, complicated data structures are often required. How such data are represented and management of memory area (or disk storage area) they occupy are vital issues. If too much effort goes in such issues, however, less effort can be put into more essential program design. Programming languages for symbolic manipulation are intended to provide the automatic mechanisms to help such management. To be more concrete, *symbolic atoms* gives a standard way to uniquely represent entities with unique names; *automatic memory management* mechanism provides automatic allocation and deallocation of standard data structures. In this respect, KL1 has similar features as other languages for symbolic processing, such as Lisp.

For parallel processing, the whole computation should be divided into smaller subcomputations and they should proceed with synchronization whenever required. For this purpose, there are many language systems that augments parallel execution and synchronization mechanisms (in cooperation with the operating system) to languages designed for sequential processing.

This approach, however, has two large problems. One is that, as the basic execution principle of the programming language remains sequential, when to apply parallel execution must be explicitly specified in the source program. This makes difficult to run the same

program efficiently on different hardware systems with varying parallelism. The situation is worse when developing a program to solve a problem for which good strategy for parallel processing is yet to be known. Frequently, the strategy can only be searched on repeated trials and errors. In such cases, rewriting the program in each such step might introduce new bugs to be debugged, making the development process quite effort-demanding.

Another problem is the difficulty of synchronization. It is not so easy to have the needs of synchronization always in mind during programming. Moreover, once a synchronization bug is introduced, it might appear in different forms for each execution, because of a slight difference in execution order of parallel processes. This makes debugging very difficult.

KL1 took a different approach. It is not a sequential programming language extended with parallel execution features; it is a born concurrent languages, in which every bit of computation can be done concurrently. This requires, in principle, rather frequent synchronization, but making synchronization fully automatic by introducing the *data-flow synchronization mechanism*, programmers *cannot* introduce simple synchronization bugs into their programs. In KL1, physical parallelism can be specified separately by the feature called *pragma.* Thus, change in physical parallelism will not change the correctness[1] of the programs. No debugging of synchronization will be needed when the physical parallelism of the program execution is altered.

Thanks to these characteristics of the KL1, the amount of effort in research and development of parallel processing software is greatly reduced. By changing only the pragmas and not changing the logic of the programs, various physical parallel execution strategy can be tested without being bothered with newly introduced bugs. This enables the programmers concentrate on more essential problem of load distribution method.

[1] Partial correctness, to be precise.

Chapter 2

Execution Mechanism of KL1

This chapter describes the outline of the execution mechanism of KL1.

2.1 Program Syntax and Basic Execution Mechanism

This section describes the syntax of KL1 programs and its basic execution mechanism using simple examples.

2.1.1 A Simple Example

As one of the most simple programs of KL1, let us consider a program of "inverter", which inverts the input, 0 or 1, to the output 1 or 0.

```
:- module inverter.
:- public not/2.

not(In, Out):- In = 0 | Out = 1.
not(In, Out):- In = 1 | Out = 0.
```

The first two lines are for modularization of programs, we will not go further into it. The latter two lines are the essential part of the program. These two lines mean the following.

- If the first argument of "not", that is "In", is 0, make the second argument "Out" be 1.
- If the first argument of "not", that is "In", is 1, make the second argument "Out" be 0.

Each of these lines is called a *clause*, which is the basic unit of program definition.

The heading part of each each clause "`not(In, Out)`" means that the clause is a part of the definition of the predicate "not", which has two arguments, which are called "In" and "Out" within this clause. This part is called the *head* of the clause. The word *predicate* is used here as KL1 is a logic programming language, but this can be interpreted as "subroutine" here.

Following the head comes a delimiter "`:-`", and the part after this delimiter and before the vertical bar "`|`" is called the *guard*. The guard gives the condition whether this clause can be applied or not.

Between the vertical bar and the full stop "." is called the *body* of the clause. This part specifies what to do when the clause is applied. Various things can be written in the body, but, in this example, operations to determine the value of the output argument such as "`Out = 1`" are described.

2.1.2 Program Execution

When the simple program given above is called interactively, the interaction might be as follows.

```
?- inverter:not(1, X).
X = 0
yes.
```

To invoke the program interactively, type in the predicate name, its arguments enclosed in parentheses and a full stop after the prompter "`?-`". The word "`inverter`" and the colon before the predicate name "`not`" is the name of the module where the predicate is defined. A predicate name with its arguments written this way is called a *goal*. Goals are a unit of execution of KL1 programs.

Here, "`1`" and "`X`" are given as arguments. The number 1 denotes integer value 1, as can be guessed easily. In KL1, names beginning with upper case letters denote variables. Here, a new variable "`X`" which appears here the first time is passed as the second argument.

The program execution will proceed as follows.

1. The formal argument given in the definition `In` corresponds to the actual argument `1`. This satisfies the condition given in the guard of the second clause. The guard condition of the first clause is not satisfied. Thus, the second clause is selected.

2. When a clause is selected, its body is executed. In the body, the second argument `Out`, which corresponds to the actual argument "`X`" in this case, is given the value `0`.

3. As nothing remains to be done, its the end of the execution.

As the result of this execution process, the value of the variable `X` given as the second argument is determined to be 0, which is indicated into the next line.[1] The message "`yes`" following shows that the execution terminated without any problems.[2]

If the first argument to be passed is altered to 0, of course, the execution will be as follows.

[1] Variables in KL1 are quite different from variables in procedural languages such as C. In procedural languages, "variables" are value holders where different values might be stored as the computation process goes on. Variables in KL1 is more like variables in mathematics; their value is either defined or not defined and, once defined, it will never change.

[2] As stated below, program execution might not terminate normally.

```
?- inverter:not(0, Y).
Y = 1
yes.
```

2.1.3 Unification

Goals with two expressions in both sides of an equation symbol "`=`", such as "`In = 0`" or "`Out = 1`" in the above inverter example program, are called *unification*. As the equation symbol is used, these goals roughly mean that both sides have the same value, but the details depend on where they appear.

Guard Unification Unifications appearing in the guard, which specifies the clause application condition, mean that the equality of the both sides is a part of the condition. As they only test whether they are equal or not, they are sometimes called *passive unification*. In the above example, "`In = 0`" tests whether "`In`" is 0 or not.

Body Unification Unifications appearing in the body, which specifies action to be taken when the clause is selected, mean that both sides should be made equal somehow. They are sometimes called *active unification*. In the above example, "`Out = 1`" determines the value of the undefined variable "`Out`", thus actively making both sides equal.

2.1.4 Syntax of Clauses and Basic Execution Mechanism

Clauses that define a program has the following syntax.

PredName(*ArgList*, ...) `:-` *Guard* `|` *Body*.

Each part has the following meanings.

PredName gives the name of the predicate (or subroutine, if you like) for which this clause gives (a part of) the definition.

ArgList determines the correspondence of actual arguments given to the predicate and the variables written in the clause definition. In KL1, the scope of the variables is confined to one clause definition (or one line of interactive input). Variables with the same name but in different clause mean distinct variables.

Guard specifies the condition needed to be satisfied to apply the clause. Any number of goals can be written separated by commas and the condition is considered to be satisfied when all of them are satisfied. In the guard, only unifications and invocations of certain predicates defined in the language can be written.

Body specifies the action to be taken when the clause is selected. Like the guard, any number of goals can be given here and all the goals will be executed when the clause is selected. Unlike in guard, user-defined predicates can be invoked from the body, in addition to unifications and language-defined predicates.

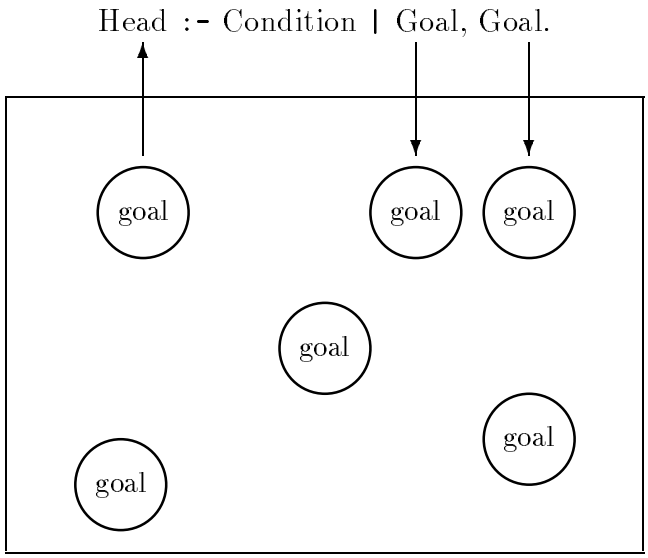

Figure 2.1: Execution Mechanism of KL1

Predicates in KL1 can be classified into two categories: *built-in predicates* as defined by the language and *user-defined predicates* as defined in the programs.

Most of the built-in predicates provides primitive operations for various data. Their behavior is defined as a part of the specification of the language.

A special built-in predicate is a predicate without any argument "`true`". When this appears in the guard, it means a condition which is always true. In the body, it means to do nothing (no operation).

On execution of a goal of user-defined predicates, the guard of the clauses defining the predicate will be tested first. Then one of the clauses whose guard is satisfied is selected, and goals in its body (including unifications) will be executed (sometime). Repetition of this process is the execution process of KL1 programs.

2.1.5 Concurrency, Communication, Synchronization

A clause can be regarded as a rewriting rule that converts a goal satisfying the guard condition to the goals in the body. As shown in Figure 2.1, the basic step of execution consists of taking some goal out of the set of goals to be executed[3], converting it to the body goals, and then returning them to the set. This step is a *reduction* step, as normally goals are converted into simpler goals. When all the goals are finally converted to the most simple goal "`true`" by repetition of such reduction steps, the execution terminates.[4]

These rewriting steps do not necessarily made sequentially. As rewriting steps for multiple goals can be simultaneous, the execution can be actually parallel.

In KL1, multiple goals can have *shared variables*. When one goal determines the value of a variable, other goals sharing the same variable can use the value to determine which

[3] As there can be multiple goals which are completely the same, it is a multi-set, to be more precise.

[4] There are programs of KL1 whose execution will never terminate and still meaningful.

clause to apply for their reduction. This is the communication mechanism of KL1.

Consider the following goals for the inverter program given above.

```
?- inverter:not(1, X), inverter:not(X, Y).
```

In this example, the two goals share the variable "`X`".

In KL1, when multiple goals are to be executed, the order of the execution is not strictly determined.[5]

Consider that the left-hand side goal is executed first. As the first argument is 1, the value of the second argument "`X`" is determined to be 0. As the right-hand side goal has this "`X`" as its first argument, clause to be applied is decided using this value 0, and the right goal determines the value of its second argument "`Y`" to 1. Thus, the execution yields the value 0 for "`X`" and 1 for "`Y`".

What if the order is exchanged? If the right-hand side goal is tried first, as the value of its first argument "`X`" has not been determined yet, the guard conditions of both of the clauses defining the predicate "`not`", namely `In = 0` and `In = 1`, cannot be tested. In such cases, execution of the goal will be postponed until some future time. This behavior is called *suspension*.

As the right-hand side goal cannot proceed further, the left-hand side goal is the only goal to be executed. Executing the left goal, the value of the variable "`X`" is determined to 0. This makes the reason of the suspension of the right-hand side goal vanish. Thus, the right goal can resume its execution. This time, as there is no obstacles in testing the guard conditions, execution proceeds and the value of the variable "`Y`" is determined to 1. The final results are the same irrespective of the execution order.

This is the mechanism of KL1 for communication and synchronization between goals. To summarize:

- Communication is through shared variables.
- Synchronization is automatically done when values are tested in guards.

The only mechanism KL1 provides for selective execution (such as if-then-else) is the clause selection by guard conditions. If the values required for checking the condition are not available yet, the execution will be automatically suspended. Once a variable is determined its value, it will retains the same value never changing it afterwards. Thus, in KL1 programs, there is no possibility of making erroneous decision because of unexpected order of execution.

2.1.6 More on Execution Mechanism

The basic execution mechanism of KL1 is described above. There are some more details not explained yet.

[5] By using the priority mechanism described below, the order can be roughly specified, but it gives only non-strict preference.

What if more than one clause were applicable?

Which clause will be selected when guards of the two or more clauses are satisfied? The language specification of KL1 does not define which. It is completely up to the implementation.[6] Thus, in correct KL1 programs, either the guard conditions should be exclusive or, if they are not exclusive, selecting any of the applicable clauses should yield the desired result.

What if none of the clauses were applicable?

What will happen if guards of all the clauses for a predicate are found to be false? For example, when if the predicate "`not`" in the above example is called as "`not(3, X)`". In such cases, the execution leads to a *failure*. Failures, like other unexpected anomalies such as arithmetical overflow, are handled as an exception by the exception handling mechanism of KL1.

What if a active unification tried to give value to a variable, which already had some value?

If the value the variable already had is the same as one the unification tried to give, nothing will happen. If they are different, it will be a failure of unification, which will be treated as an exception. Although the behavior is well defined, the programming style of unifying two variables both of which already have values is not recommended. Such operation is not only inefficient but also makes the program harder to read.

2.2 Characteristics of KL1

Let us review the characteristics of the execution mechanism of KL1 by comparing it with other programming languages.

2.2.1 When Compared with Procedural Languages

Unlike in procedural languages, KL1 does not have variables which *changes* its value as computation proceeds. A variable has either its value determined or not determined yet, and once determined, it will not change ever. By this, goal reductions are liberate from caring for *global* state of storage, providing more freedom in the order of computation, including concurrent execution. This is quite advantageous for describing parallel processing.

2.2.2 When Compared with Functional Languages

KL1 shares the characteristics of being free from global state with functional languages,[7] and they also share its benefits for parallel processing.

[6] A mechanism to specify priority between clauses is provided, but it only gives preference and the implementation might not obey it. Actually, in non-shared memory systems like Multi-PSI, if some clause can check the guard condition with data available locally, the implementation prefers to use that clause.

[7] Here, functional languages mean *pure* functional languages. Commonly used Lisp-family languages, in this respect, are procedural languages.

The largest difference with functional languages is not in the basis of the language, i.e., whether functions or predicates. It is that KL1 has *nondeterminacy*, that is, the same program with the same input data run twice might yield different results. This may happen when the guard conditions of more than one clause are found to be true. This can become a headache in debugging, but is quite advantageous for efficiently solving problems on different hardware architecture. Programs in pure functional languages, even executed in parallel, do essentially the same computation on any hardware; on the other hand, programs in KL1 can be written in a way that it will adapt itself to the available parallelism of the hardware.

2.2.3 When Compared with Other Logic Programming Languages

Prolog The syntax of KL1 resembles that of Prolog, which is also based on Horn logic.[8] They share the same characteristics that programs can be regarded as logical formulae and the way of making their correspondence is also the same. From the programming language viewpoint, however, KL1 and Prolog are best considered to be two completely different languages. These two, although based on the same Horn logic, departed to quite different directions.

Concurrent Prolog, Parlog and GHC Concurrent Prolog[4], Parlog[3], Guarded Horn Clauses (GHC) are languages all aiming at the same direction.[6] KL1 is based on Flat GHC (FGHC), which is a simplified version of GHC, which already is the simplest of this language family, sometimes called *concurrent* logic programming languages.

2.2.4 When Compared with Object-Oriented Languages

KL1 itself is not an object-oriented language. The programming style frequently used in KL1 or any other concurrent logic programming languages, however, is object-oriented (or agent-oriented) style, representing objects as concurrent processes. Most of the object-oriented programming paradigm can be described in KL1 in a straightforward manner. This programming style will be described in the following chapters.

Summary

The most essential parts of the language specification of the KL1 were described, including:

- Basic execution mechanism as concurrent reductions,
- Communication between goals by shared variables, and
- Synchronization by suspending guard condition tests.

[8] A subset of first order logic, for which mechanical theorem proving is relatively easy. Prolog (pure version without *cut* nor *assert*) and KL1 can be considered as sound but incomplete automatic theorem prover for this logic.

Chapter 3

Data Types and Primitive Operations

Up to now, only concrete values we used in the examples are integer values. Of course, KL1 programs can handle data of various other types. This chapter describes basic data types provided by KL1 and primitive operations on them.

3.1 Variables without Specified Types

In KL1 programs, the types of values variables can have are not explicitly declared. The same variable in the same clause in the source program (but not the same variable in execution time) can hold values of different types in one invocation and another. This is similar to Prolog or Lisp.

This has both merits and demerits. One demerit is that an efficient implementation becomes more difficult, as what kind of value a variable can have cannot be determined until run time, in principle.[1] For human programmers, the type of the values of variables might have been a great help in understanding the behavior of programs. In this respect, so-called strongly typed languages in which all variables have declared types might be better.

On the other hand, the merits are: no special notation such as *union* in C is needed to use variables which have different kind of data depending on other data; no special mechanism such as *generic package* in Ada is needed to write programs that can be used in common with different types of data.

In one sentence, variables without types makes reading programs more difficult, but writing programs easier. This makes the language suitable for experimental programs, which will be rewritten many times in trial and error style program development process. This might be one of the reasons why Lisp has been used widely in AI research.

3.2 Atomic Data

Atomic data are data without internal structure and the value itself (rather than its contents) has the meaning. KL1 has the following atomic data types.

[1] This problem can be solved at least partially by static analysis called *type inference* during compilation.

Symbolic Atoms denote no specific meaning. They are identifiers only their identity is significant. They are useful in uniquely representing various notions in programs.

Integers represent usual integer values. Normal decimal notation with optional sign is used (such as ("`3`", "`-15`", etc).

Floating-Point Numbers approximate real numbers. Decimal notation with a decimal point (such as "`3.14`") is used. Exponent part can be added optionally (as in "`0.314e+1`").

3.2.1 Symbolic Atoms

Symbolic atoms in KL1, unlike those in Lisp, do not have any attributes. Even the correspondence to their name strings is only maintained by the compiler and the programming system and not defined by the language itself. Thus, the only possible operation on symbolic atoms is equality check in the guard.

The notation of symbolic string is similar to Edinburgh Prolog [1], which is one of the following.

- A lower case letter followed by a sequence of any number (including zero) of letters, digits or underlines ("`_`"). For example, "`kl1`", "`multiPSI`" or "`psi_3`".
- A sequence of special characters (some of "`~`", "`+`", "`-`", "`*`", "`/`", "`\`", "`^`", "`<`", "`>`", "`=`", "`‘`", "`’`", "`:`", "`.`", "`?`", "`@`", "`#`", "`$`", "`&`"). For example, "`+`" or "`:-`".
- A sequence of any characters quoted by single quotes ("`’`"). If single quote characters are to be included, they should be doubled. For example, "`’Hello world’`" or "`’’’quoted’’’`".
- Special one character atoms. There are three of them, which are "`!`", "`;`" and "`[]`".

3.2.2 Numbers

Arithmetics and comparison on integer and floating-point numbers can be effected by *built-in predicates* provided as language primitives.

Magnitude comparison on integers can be made by writing built-in predicates in guard conditions. It is much more convenient to use macro notations than to directly use the built-in predicates.

For comparison of integers in the guard, binary operators "`<`", "`>`", "`=<`" and "`>=`" can be used. For example, "`X =< Y`" tests whether "X is less than or equal to Y". For equality and inequality, "`=:=`" and "`=\=`" can be used. Equality can also be tested by unification but, as is also true with all the macros for integer comparison listed here, "`=:=`" allows arithmetical expressions as its operands.

For example, a predicate that returns the larger of the first and the second arguments to the third argument can be defined as follows.

```
max(X, Y, Z) :- X >= Y | Z = X.
max(X, Y, Z) :- X =< Y | Z = Y.
```

In this program, if X and Y are the same, the guard conditions of the both clauses hold. As mentioned above, the language does not specify which of the two clauses will be selected in such cases. In this program, however, there will be no problem because both yields the same result.

Integer arithmetics are also provided as built-in predicates that can be done in both the guard and the body, but macro notations are also convenient for them. The variable to be given the value should be written in the left-hand side of a binary operator "`:=`", and usual arithmetical expression using "`+`", "`-`", "`*`", "`/`" and parentheses in the right-hand side.[2]

For example, a predicate that returns the sum of the first and the second argument to the third argument can be defined as follows.

```
sum(X, Y, Z) :- true | Z := X + Y.
```

Here, the guard is "`true`" which means always true, i.e., that this clause can be applied unconditionally.

Almost the same set of operations are available for floating-point numbers, but they are operations different from those for integers. They are distinguished by putting a dollar sign "`$`" in front of the name of the operator. For example, floating-point versions of the above two example predicates `max` and `sum` can be defined as follows.

```
max(X, Y, Z) :- X $>= Y | Z = X.
max(X, Y, Z) :- X $=< Y | Z = Y.

sum(X, Y, Z) :- true | Z $:= X + Y.
```

3.3 Structured Value

Structured value consists of element data gathered in one way or another, depending on their types. KL1 has the following structured data types.[3]

Vector is a structure consisting of values of arbitrary data types as its elements. Each element can be accessed by integer index. Elements of one vector structure may have values of different types.

String is a structure consisting of integer values within some range. Depending on the range of the element values, 1-, 8-, 16- and 32-bit strings are available. Strings are basically the same as vectors but, as the type of the elements and their values are restricted, more compact representation and more efficient operations are possible.

Cons is a structure with two elements. One "cons" is nothing more than a vector with two elements, but by making a combination multiple conses, more complicated but flexible structures such as lists can be represented.

[2] In addition, bit-wise and, or, xor, complement and shift operations are provided, which will not be described here in detail.

[3] Other data structures such as one representing executable object code are also available, which will not be discussed in detail here.

3.3.1 Vectors

Vectors are a data structure which can have any number of elements (possibly zero elements) of arbitrary types.

The canonical notation of vectors is writing elements between braces, "`{`" and "`}`". For example, a vector with three integers 0, 1 and 2 as its elements can be denoted as "`{1, 2, 3}`". As types of elements of vectors can be of any type, they can be vectors again. For example, "`{a, b, {0, 1, 2}, d}`" has four elements, one of them being a vector with three elements.

The elements of vectors are indexed from zero. For example, the element with index 1 of the vector "`{a, b, {0, 1, 2}, d}`" is "`b`".

An alternative notation is provided for vectors with two or more elements whose first (index 0) element being a symbolic atom. They can be denoted by the name of that atom followed by an open parenthesis "`(`", list of elements other than the first delimited by commas, and a close parenthesis "`)`". For example, the vector "`{a, b, {0, 1, 2}, d}`" mentioned above can also be denoted as "`a(b, {0, 1, 2}, d)`". This notation is convenient for discriminating various data structures by giving them some name as their first element. Such structures are sometimes called a *functor*. When a vector structure is regarded as a functor structure, the first argument is called the *functor name* and the rest are called the *arguments*. Thus, the functor "`a(b, {0, 1, 2}, d)`" has the name "`a`" and its first argument is "`b`".

3.3.2 Strings

A string is a data structure which has integer values within certain range as its elements. It is often used to represent character strings which has codes character as its elements. Such character strings can be denoted as, for example, "`"abc"`", writing the element characters within double-quote symbols. In most of the implementations of KL1 currently available, the default range of the elements is 0 through 65,535 (unsigned 16 bit integers) and JIS[4] 16-bit character code set is used.

Elements of strings are also indexed by integers beginning from zero. For example, the element with index 1 of the string "`"abc"`" is a the character code for "b" in JIS 16-bit code (which actually is 2662 in hexadecimal).

3.3.3 Conses

A "cons" is a data structure with two elements. Following Lisp, these elements are called *car* and *cdr*, respectively. For their denotation, two elements are written within brackets "[" and "]", separated by a vertical bar "`|`". For example, "`[a|b]`" denotes a cons whose car is a symbolic atom "`a`" and cdr is "`b`".

Conses themselves are a simple data structure with two elements, but various data structures can be represented but combining many of them. The most frequently used such data structures are list structure. In list structures, the car part represents one element of the list and the cdr part represents the rest. A list without any elements is represented as the symbolic atom "`[]`". Represented this way, a list with two elements

[4] Japan Industry Standard.

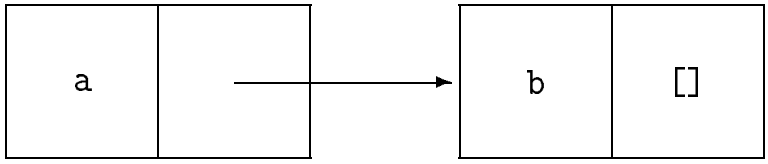

Figure 3.1: Representing a List with Conses

will be "`[a | [b | []]]`" using nested conses (Figure 3.1). As this notation is not quite convenient for longer lists, this can also be denoted as "`[a, b]`".

As lists are data structures very frequently used in KL1, and understanding their syntax is crucial in understanding what follows, let us explain some more examples.

The basic notation for lists is "`[CAR|CDR]`", which consists of the first element "`CAR`" and the tail of the list "`CDR`". An empty list is represented by an atom "`[]`". If "`CDR`" happens to be empty, that is, when the list consists of only one element "`CAR`", such a list can be denoted as "`[CAR|[]]`" in the basic notation, or, alternatively, as "`[CAR]`".

A list consisting of four elements, "`one`", "`two`", "`three`" and "`four`" can be denoted as "`[first, second, third, fourth]`". On the other hand, a list consisting of four or more elements, but with the first four elements being "`first`", "`second`", "`third`" and "`fourth`", can be denoted as "`[first, second, third, fourth |Rest]`" where the variable "`Rest`" corresponds to the list beginning with the fifth argument, or an empty list, if the whole list had only four elements.

3.3.4 Incomplete Data Structures

Data structures with elements of arbitrary types, such as vectors and conses, can have values of their elements left undetermined. Such data structures are called an *incomplete data structure*.

Elements of incomplete data structures which do not have their value determined yet are denoted as variables. For example, "`[a|X]`" means an incomplete "cons" whose car is a symbolic atom "`a`" and the cdr is a variable "`X`".

Variables in incomplete data structures can also be shared among goals. Their values can be determined by goals other than the goal which has the whole data structure. Thus, incomplete structures can be made more complete gradually by some other goals. Incomplete data structures play important roles in various programming technique of KL1, some of which will be described later in this document, and are the source of flexible programming styles of KL1.

3.3.5 Unification of Structured Data

Structured data also can be operands of unification.

Guard Unification As stated above, guard unifications check out whether the both operands have the same value or not. For structured data, the unification rule becomes recursive.

1. First, check if both operands have the same type and the same number of elements. Otherwise, they are not equal.

2. Then, apply unification to all the corresponding elements (with the same index values) of two structures. When and only when all the elements are equal, the two structures are equal as a whole.

On unification of elements, if one of the operands is a variable not corresponding to an argument,[5] the other operand, which is a element of the other structure, is given to the variable as its value. For example, the following program defines a predicate which, when a cons is given as its first argument, returns its car and cdr to the second and the third arguments.

```
carcdr(Cons, Car, Cdr) :- Cons = [X|Y] | Car = X, Cdr = Y.
```

Note that, when the first argument is not a cons, the unification in the guard will not succeed and thus this clause cannot be applied, according to the first rule of the guard unification above,

Body Unification When one of the operands of a body unification is a variable whose value is not determined yet, its value is determined to be the other operand, whether it is structured or atomic.

If both operands are structures, the unification rule is recursive as follows. [6]

1. First, check if both operands have the same type and the same number of elements. Otherwise, the unification fails.

2. Then, apply unification to all the corresponding elements (with the same index values) of two structures. When and only when all the elements are successfully unified, the two structures are successfully unified.

3.3.6 Identity of Structures and Variables

It may be worth noting that structures in KL1 are structured values, unlike structured storage in procedural languages that has ever changing values. If two structures are the same one time, they will be for ever in KL1. There is no distinction between *equality* and *identity* as far as values are concerned. After two structures are unified, that is, their equality is proved in a guard unification or they are made equal in a body unification, they both can be considered to be identical.

On the other hand, variables should not be copied. One variable may be accessed through two or more paths. If the value of the variable is determined through one of the paths, then that value should be made visible from other paths, too. Thus, variables have their identity, while values (including structures) don't.

If some elements of a structure are not determined yet, that is, if it is an incomplete data structure, the undetermined elements, that are variables, should not be copied. In

[5] Nor given value applying this rule.

[6] Although these are the rules, applying body unification to two operands both having their values determined already is not a recommended coding style.

the implementation level, they should be shared (through pointers) so that all accesses will go to the same place.

Although incomplete data structures introduce some subtlety, lack of identity of structures makes distributed memory implementation of the language much easier. Each processor can keep its own copy of shared structures, rather than accessing their origin, allowing much more efficient processing.

Summary

Data structures and operations on them available in KL1 are described. The most characteristic feature among them is that of incomplete data structures, i.e., data structures whose undetermined element can be safely handled by KL1 programs.

Chapter 4

Processes and Stream Communication

The basics of the KL1 language were described in the previous sections. In this section, a programming technique very frequently used in KL1 is described, in which processes and streams connecting them are used.

Since this programming style was first proposed by Shapiro and Takeuchi[5], it has been essentially used in the development of the operating system of parallel inference machines, PIMOS[2], and various application programs in KL1.

4.1 Notational Conventions

Before going into programming technique issues, let us introduce several notational conventions.

4.1.1 Denoting Guard Unification within Head

Guard unification with one of the arguments passed to the predicate can be more tersely written by writing the pattern to match against it directly at the position of the argument in the head. For example, the above example program:

```
not(In, Out) :- In = 1 | Out = 0.
```

can also be written in a shorter form as:

```
not(1, Out) :- true | Out = 0.
```

A similar convention applies to when the guard unification is not against a concrete value. For example:

```
same(X, Y, Ans) :- X = Y | Ans = same.
```

checks the equality of two arguments to the predicates. This can also be expressed by writing the same variable twice in the head as follows.

```
same(X, X, Ans) :- true | Ans = same.
```

4.1.2 Omitting the Goal "true"

Unconditional guards, i.e., guards which have only the invocations of "true", the whole guard can be omitted along with the vertical bar separating the guard with the body. For example, the inverter program above can be written as follows.

```
not(1, Out) :- Out = 0.
```

Not only the guard but also the body had only the invocations of "true", the body also can be omitted along with ":-" symbol. For example, a clause such as:

```
one(1) :- true.
```

can be written as follows.

```
one(1).
```

4.2 Processes

As explained above, the basic execution mechanism of KL1 is concurrent (that is, possibly parallel without specific order) repetition of reductions of goals. This bare mechanism, however, is not in sufficiently high level for describing complicated computation process in a way easily understood. The notion of *process* helps grouping the repeated reductions making higher level computation structures.

4.2.1 Processes of KL1

Predicates with an invocation of the same predicate (probably with slightly different arguments) in their body can be regarded as expressing an execution loop. One reduction of a goal of such a predicate will result in another goal for the same predicate, which is to be reduced again. For example, a predicate that counts down to zero starting with the given argument can be defined by the following two clauses.

```
count_down(0).
count_down(N) :- N > 0 | M := N-1, count_down(M).
```

N.B.: In this example, counting down is *not* done by "N := N-1". As stated above, variables in KL1 are not places to store changing values; rather they are names given to some values themselves. Thus, a new name "M" is needed for "N-1", which is a value different from "N".

The body of the second clause in the above example contains two goals: "`M := N-1`" that performs subtraction and "`count_down(M)`", recursive invocation. As the order of goals in the body does not make any significance in KL1, their execution order is not defined. The recursion may precede the subtraction and they may also be executed in parallel. As both of the clauses for the predicate `count_down` have a guard condition or its abbreviation in the head that tests the argument, the reduction of the recursive goal will anyway be suspended until the value of the argument, i.e., the result of subtraction, is obtained.

Although each reduction is only a fragment of computation, the repetitive reductions of recursive invocations as a whole can be regarded as a computation process of considerable size. This repetitive reductions is called a *process*.[1] In the rest of this section, the characteristics of processes in KL1 are explained with concrete examples.

4.2.2 Summing Up Elements of a List

A program to compute the sum of elements of a list can be defined as follows.

```
sum([], PSum, Sum) :- Sum = PSum.
sum([One|Rest], PSum, Sum) :-
    NewPSum := PSum + One,
    sum(Rest, NewPSum, Sum).

sum(List, Sum) :- sum(List, 0, Sum).
```

This program consists of two predicate definitions. Both have the same name "`sum`" but with different numbers of arguments (sometimes called *arities*). In KL1, predicates with the same name but different arity are considered to be different predicates. Such predicates are often used for auxiliary purpose, as in this program.

The first two clauses with three arguments is the main routine that actually makes the sum. When this predicate is examined individually, its function is to compute the total sum of the elements of the list given as the first argument and the partial sum given as the second argument, and return the result to the third argument. The first clause means: if the first argument is an empty list, as no more elements to be added are there, the second argument should be the result. The other clause means: if the list is not empty, having "`NewPSum`" be the sum of the first list element "`One`" and the second argument "`PSum`", the total of this "NewPSum" and the rest of the elements in "`Rest`" should give the grand total.

The last clause defines two argument predicate, which is the top level. It means: by adding the sum of elements of a list with 0, the sum of the list elements can be obtained.

Note that, the addition in the body of the second clause and the recursive invocation can be executed in parallel. As the guard condition only test whether the list is empty or not, the result of the addition is not needed to select which clause to execute. Thus, recursions can proceed without waiting for the additions to complete, making many addition

[1] Note that, in KL1, the notion of processes if *not* a part of the language, but a notion available in a programming technique, although the technique is so commonly used that it virtually is a part of the language.

goals to be executed concurrently.[2] As one of the arguments of the additions (`PSum`) is the result of the addition one cycle before, the order will be determined among additions in this case.[3]

As is this example, what is called a *process* in KL1 is not necessarily a sequential process, but may have concurrency within itself. Whether to consider a process being a process totally depends on the viewpoints.

4.2.3 List of Natural Numbers

A program that makes a list of all natural numbers less than some given number can be defined as follows.

```
naturals(N, M, List) :- N >= M | List = [].
naturals(N, M, List) :- N < M |
    List = [N|Rest],
    N1 := N + 1,
    naturals(N, M, Rest).

naturals(M, List) :- naturals(0, M, List).
```

This program also consists of two predicates, the top level one with two arguments and an auxiliary one with three arguments.

The predicate defined by the first two clauses makes a list of integers beginning with the first argument up to one less than the second argument and returns it to the third argument. The first clause means: if the first argument is not less than the second, return an empty list. The second clause means: if the first argument *is* less than the second, return a list beginning with the first argument; the rest of the list should begin with one more than the original first argument.

The last clause defines the top level predicate. By calling the three argument auxiliary predicate with its first argument 0, the desired list will be obtained.

In this example also, the goals in the second clause, i.e., a body unification, an addition and a recursive invocation are executed concurrently (in any order or in parallel).

4.3 Stream Communication

In this section, ways to combine many *processes* described in the above section, to make more complicated computation.

[2] Concurrency, that is, possibility of parallel execution need not suggest speed-up by actually executing in parallel. Because of overhead of communication, doing such a simple operations as integer additions in parallel is not beneficial and KL1 implementations will not try that. However, when the simple additions in this example are replaced by more complicated operations, such as *bignum* additions, actual parallelism may become really beneficial.

[3] the situation will change when additions are replaced with more complicated operations that allow pipelined processing described below.

4.3.1 Combining Processes

Two or more processes can be combined so as to the result of one process can be used in computation of other processes. For example, sum of natural numbers less than a given integer can be computed by a program combining the two programs explained above, defined as follows.

```
sum_up_to(N, Sum) :- naturals(N, List), sum(List, Sum).
```

In this program, the variable "List" is shared from two processes, serving as the medium to pass the list made by the process "naturals" to the process "sum".

As the invocations of these two processes are written in a body of the same clause, their execution order is not defined. As the predicate "sum" with two arguments has not guard conditions, it will call three argument "sum". As this depends its clause selection on the value of its first argument, it will be suspended until its value is determined by another process.

Let us review here the definition of the predicate that actually creates a list of natural numbers.

```
naturals(N, M, List) :- N >= M | List = [].
naturals(N, M, List) :- N < M |
    List = [N|Rest],
    N1 := N + 1,
    naturals(N, M, Rest).
```

As mentioned above, the goals in the second clause, i.e., a body unification, an addition and a recursive invocation can be done in any order. As the unification can precede other goals, the result can be returned *before* the computation ends. Of course, the result has not been computed completely yet. The cdr of the list is still left undefined and thus the length of the list is not determined yet. This is a typical incomplete data structure. However, the result is already known to be a "cons" by this unification, and its car is defined to be the first argument passed, that is, an integer 0 on its first invocation.

Let us now look again into the summing up predicate, the three argument "sum".

```
sum([], PSum, Sum) :- Sum = PSum.
sum([One|Rest], PSum, Sum) :-
    NewPSum := PSum + One,
    sum(Rest, NewPSum, Sum).
```

As the result of the predicate "naturals" has been decided to be a "cons", guard condition (actually written in the head) can be checked out and the second clause will be selected. The car part of the first argument (here, the variable "One" matches against) is also determined to be 0. As another 0 is passed in the original invocation to the second argument "PSum", the addition $0 + 0$ is already executable, after only a little step in the "naturals" process.

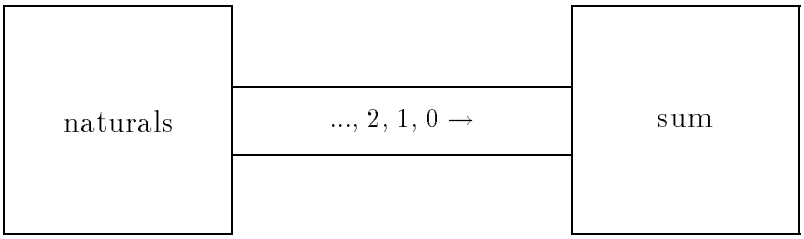

Figure 4.1: Inter-Process Communication Channel

The recursive invocation cannot proceed like this. As the process "`naturals`" only made its first step, the variable "`Rest`" is still undefined. In the recursively called "`sum`", the value of the first argument is not defined, making it impossible yet to select a clause. This "`Rest`" is now shared between two processes, "`naturals`" and "`sum`", the former generating its value and the latter consuming it. The recursive invocation of "`sum`" cannot proceed before "`naturals`" makes further steps forward. As "`naturals`" proceeds each step, "`sum`" also can proceed one more step.

As in this example, returning an incomplete structure as its value, gradually defining its elements afterwards, is a very convenient programming style of KL1 for higher concurrency. If the result should not be returned before the whole structure is completed, the execution of "`sum`" will be suspended until the execution of "`naturals`" completely terminates, lowering the parallelism.[4]

4.3.2 Stream Communication

The two processes explained in the last section are in relation that values generated incrementally by the process "`naturals`" are consumed also incrementally by the process "`sum`". The two processes shared an incomplete data structure, which is actually a list whose elements are incrementally defined.

This list structure can be regarded as a communication channel between two processes. From this viewpoint, the process "`naturals`" sends natural numbers one by one, while the other process "`sum`" receives the numbers one by one to proceed the computation (Figure 4.1). If the data to be received is not available yet, the computation of the process will suspend.

The communication channel is realized by the data structure "cons". Its car conveys the data flowing through the channel and its cdr represents the rest of the channel. At the end of the communication, that is, when there are no more data to convey, the cdr will become "`[]`". As the communication channel is represented as a data structure, the order of data in the channel is defined also by the data structure completely. The order of execution can never change the order of data. Such a communication channel is called a *stream*.[5]

Data conveyed by streams can be regarded as a *message* exchanged between processes.

[4] Note again that concurrency does not always mean actual parallel execution is beneficial. Physical parallelism should be decided considering other issues such as communication overhead.

[5] Actually the order does not matter in the example given here, as what is computed is sum of them, but, in general, the order may be significant.

In this viewpoint, the process "natural" is sending requests to add numbers to the process "sum".

4.3.3 Process State

Processes are doing their job according to the messages they receive. If what should be done is completely determined by the message, there will be no need to write that way. In stead of making a process and sending messages, defining a predicate and calling it will suffice.

The merit of using message-driven processes is that processes can have their *state*. For example, a process which holds an integer value as its state and changes the value according to the received messages can be defined as follows.

```
counter(Stream):- counter(Stream, 0).

counter([], Count).
counter([up|Stream], Count) :-
    NewCount := Count + 1,
    counter(Stream, NewCount).
counter([down|Stream], Count) :-
    NewCount := Count - 1,
    counter(Stream, NewCount).
```

The process defined in this program has the value of the counter as its second argument. It initial value is 0, as invoked from the top level predicate, which will be incremented and decremented each time "up" and "down" message arrives. As in this example, a process in KL1 *retains its state as its arguments.*

To use such a process, the predicate should be called with an argument used for the message stream as follows.

```
?- counter(Stream), some_other_process(Stream).
```

This message stream will be the *only* interface with this process and the others. The value of the counter retained as the process status can never been accessed directly from outside. Using arguments to retain the process state is encapsulating information hiding it from outside.

How such information, then, can be accessed? In the above program, the counter values can be altered by sending messages, but the value can never be read out. It will be explained in the next section.

4.3.4 Structured Messages

Up to here, all messages sent through streams were atomic data. Structured data, of course, can be used as messages.

Functor structures are frequently used as messages. As explained above, functor structures are actually vectors with their first element (the functor name) being a symbolic atom

representing the kind of structure. Functors are conveniently used as messages, the functor name indicating the kind of message and other elements (arguments of the functor) describing details.

The above example program of counter can revised to allow increment and decrement by arbitrary amount by adding arguments to messages, as follows.

```
counter(Stream):- counter(Stream, 0).

counter([], Count).
counter([up(N)|Stream], Count) :-
    NewCount := Count + N,
    counter(Stream, NewCount).
counter([down(N)|Stream], Count) :-
    NewCount := Count - N,
    counter(Stream, NewCount).
```

To read out the counter value, the following clause, which also defines functor message, can be added to the program.

```
counter([show(Value)|Stream], Count) :-
    Value = Count,
    counter(Stream, NewCount).
```

The message "`show`" for reading out the counter value should be sent with the argument "`Value`" left undefined, to let the counter process return the value there. Such messages with undefined arguments are called an *incomplete message*. The incomplete message mechanism is another example of the use of incomplete data structures. Using this style, single stream can be used for both-way communication.

Summary

Programming style of KL1 using *processes* and *streams* is explained. Incomplete data structures play significant role in this programming style.

Chapter 5

Process Networks

The programming style of KL1 with processes and message streams connecting them is explained in the previous chapter. In this chapter, various typical methods of combining processes with streams to construct more complex programs.

5.1 Filters

A program to compute the sum of natural numbers less than the given value was explained in the previous chapter. The program consisted of two processes, one generating a list of natural numbers and another summing up the elements of the list.

Let us now consider a slightly different problem of computing the sum of *squares* of natural numbers up to the given value. Two methods to make the program revising one in the previous chapter might be as follows.

- Altering the process generating a list of natural numbers so that it will create a list of *squares* of them.
- Altering the process summing up the list elements so that it will sum up the *squares* of them.

These two methods are both trying to alter the existing program. Such methods, of course, will work and probably good enough for this particular problem. If, however, both processes were much more complicated and alteration was much difficult, could there be any better ways to do the job?

A program that makes a list of natural numbers and a program that sums them up is already there. If a program that converts a list of integers to a list of the squares of each element can be made and put in between the two processes, it can be an excellent answer. The following program does this.

```
square([], Out) :- Out = [].
square([One|Rest], Out) :-
    Square := One * One,
    Out = [Square|OutTail],
    square(Rest, OutTail).
```

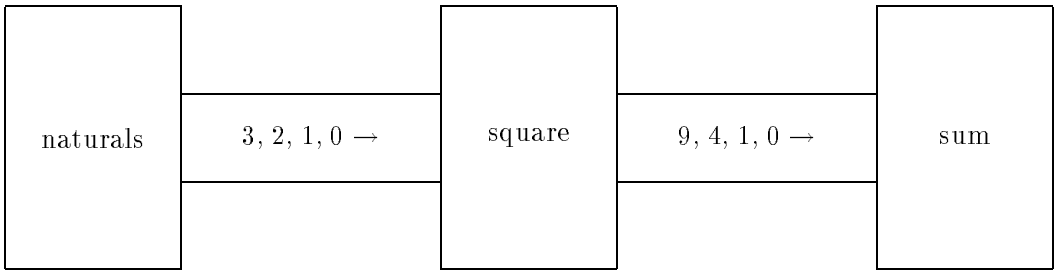

Figure 5.1: Filters

With this predicate defined, the top level program will be as follows.

```
square_sum_up_to(N, Sum) :-
    naturals(N, Naturals),
    square(Naturals, Squares),
    sum(Squares, Sum).
```

The predicate "`square`" takes a lists as its input and creates another list as its output. How this looks like when the this predicate is regarded as implementing a process and two lists as streams? In this view, the program defines a process that receives messages (which are integers) from one stream and send messages (which are square of the received messages) to another stream. A process like this, making some modification to the message received from an input stream and send the modified message to the output stream, is called a *filter*. The overall structure of this program is that processes "`naturals`" and "`sum`" are communicating through the filter "`square`" (Figure 5.1).

In this particular example, the process "`square`" playing the role of a filter does not have any internal state, sending messages depending only on the received messages. In general, outgoing messages can depend not only on incoming messages but also on the state of filter processes. As the state of processes may be altered by incoming messages, filtering function can be history sensitive.

5.2 Concatenating Streams

Let us alter the problem slightly again. Now the problem is to compute the total of both *squares* and *cubes* of natural numbers less than the given integer.

A similar method as the solution to the previous problem can be used, making a filter which generates $n^2 + n^3$ for input n. This, however requires revision of the already working predicate "`square`" and the revision also makes it more complicated.[1] Let us then leave the "`square`" predicate as it is and define another filter predicate that sends cubes of input message.

[1] Again, if this particular example problem is all you need, you might have better rewrite everything from scratch. The method described here is actually useful when required computation is much more complicated and rewriting filters requires considerable effort.

Although you now should be quite certain how to define such a predicate. It can be defined as follows.

```
cube([], Out) :- Out = [].
cube([One|Rest], Out) :-
    Cube := One * One * One,
    Out = [Cube|OutTail],
    cube(Rest, OutTail).
```

The outgoing messages of the "naturals" process can be sent to both "square" and "cube" processes by simply letting them read the same message stream as follows.

```
square_sum_up_to(N, Sum) :-
    naturals(N, Naturals),
    square(Naturals, Squares),
    cube(Naturals, Cubes),
    ...
```

Note that the stream "Naturals" here simply represents a data structure. The messages in it will never be lost just because somebody read it. To obtain next messages in the stream, processes have to examine through the data structure explicitly. Letting both two processes read all the messages from the same message stream, thus, is quite simple as above.

A more difficult problem is how the messages in the output streams, "Squares" and "Cubes" in the above program, can be fed to the process "sum".

One method is to let all the messages from one stream go first, and after all of them are sent, let messages from the other go. A predicate to do this switching can be written as follows.

```
append([], In2, Out) :- Out = In2.
append([Msg|In1], In2, Out) :-
    Out = [Msg|OutTail],
    append(In1, In2, OutTail).
```

This predicate takes two input streams as the first and the second argument, and the third argument will be the output stream. The overall function of this predicate is sending messages from the first input stream to the output stream first, and sending message from the second input after the first stream comes to an end.

When there remain some messages in the first input stream, that is, when the first argument is a not-empty list, the second clause is selected; the first message is sent to output. When the first input stream comes to an end, that is, the first argument becomes an empty list, the first clause is selected; the unification in the body of the first clause connects the output stream directly with the second input stream, which means that all the message coming from that stream should be automatically transferred to the output without receiving and sending them individually.

If this predicate is to be interpreted as a predicate on lists rather than on streams, it makes a concatenation of two lists given as the first and the second arguments and returns it to the third argument.

Using this predicate, the top level of the program will be as follows.

```
queer_sum(N, Sum) :-
    naturals(N, Naturals),
    square(Naturals, Squares),
    cube(Naturals, Cubes),
    append(Squares, Cubes, Both),
    sum(Both, Sum).
```

Note that, in the "append" program, the values of the messages are not examined at all. It only checks whether there are any messages coming or not and forwarding them without looking into them. Thus, the same "append" program can be used for streams with any kind of messages. This benefits from the fact that variables of KL1 does not have specific type declared.

5.3 Merging Streams

The method described in the previous section is not completely satisfactory. In that method, messages from the stream "Cubes" will not arrive the process "sum" until the stream "Squares" is closed. If the process "square" is time consuming, the process "sum" running concurrently with it will become idle. As the process "cube" may already have sent some output to the stream "Cubes", those might be used by "sum", but, as the stream is choked at "append", they cannot reach "sum".

Let us then consider a method in which, before one input stream coming to an end, messages from the other stream can also be forwarded. Messages coming from either stream should be forwarded to the output. The following program defines such a predicate.

```
merge([], In2, Out) :- Out = In2.
merge(In1, [], Out) :- Out = In1.
merge([Msg|In1], In2, Out) :-
    Out = [Msg|OutTail],
    merge(In1, In2, OutTail).
merge(In1, [Msg|In2], Out) :-
    Out = [Msg|OutTail],
    merge(In1, In2, OutTail).
```

The first two clauses mean when either of the input streams comes to an end, the other stream should be directly connected to the output. The remaining two clauses is for forwarding messages from either of the input streams to the output.

A process forwarding multiple input streams to an output stream is called a *merger*. The "append" predicate described above is a kind of merger, but with stronger restriction.

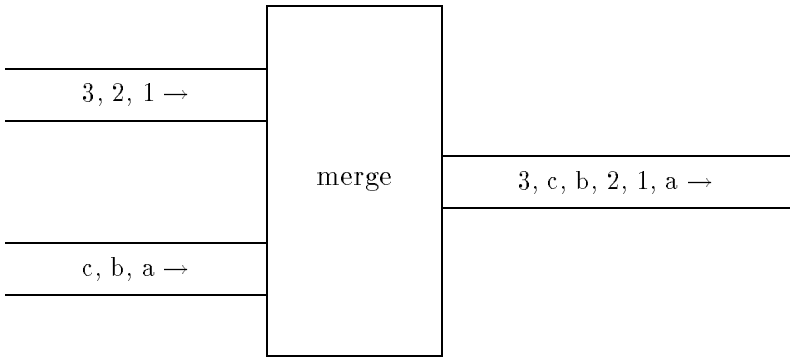

Figure 5.2: Merger

Examine the program carefully what will happen if both streams already have messages to be forwarded. In such a case, both the third and the fourth clauses have their clause application condition satisfied. As stated many times before, the language does not define which clause will be selected. The same merger program with the same input data thus can send messages to the output stream in different order (Figure 5.2). This is a typical example where the *nondeterminacy* of KL1 is made apparent and this nondeterminacy sometimes makes debugging more difficult. It is also possible, of course, to write deterministic programs as "`append`" to avoid the problem, but nondeterminacy can be indispensable for gaining concurrency as in this example. Note that, although the order of messages from different input streams is nondeterministic, the order between messages originally from the same input stream will be preserved in the output.

Using this "`merge`", the top level program will be as follows.

```
queer_sum(N, Sum) :-
    naturals(N, Naturals),
    square(Naturals, Squares),
    cube(Naturals, Cubes),
    merge(Squares, Cubes, Both),
    sum(Both, Sum).
```

With this program, messages from both processes "`square`" and "`cube`" can be forwarded to the process "`sum`" as soon as they become available, giving the program better concurrency than one using "`append`". The overall structure of processes of this program is shown in Figure 5.3.

5.4 Built-in Mergers

The merger defined in KL1 explained in the previous section still has problems. One is the number of input streams. As predicates with varying number of arguments cannot be defined in KL1, only mergers with fixed number of input streams can be defined. Thus, to realize mergers with dynamically changing number of inputs, a tree-like network of mergers has to be made, which allows addition of new input streams dynamically. When

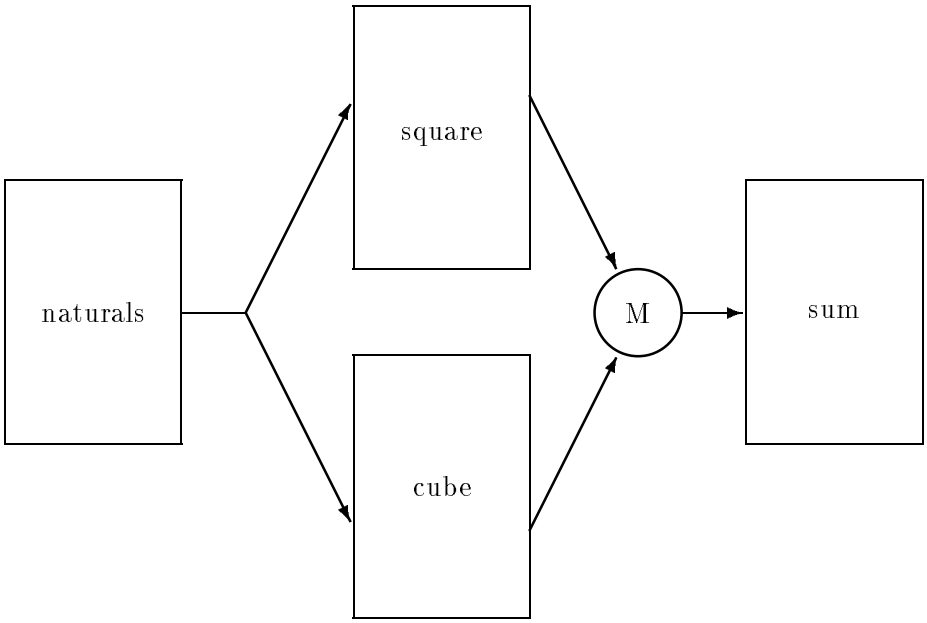

Figure 5.3: Process Network with a Merger

a message arrives at one of the input streams of such a network, it must go through as many mergers as the height of the tree structure. An unbalanced tree structure, thus, may result in very low performance. The tree structures can be balanced on addition and deletion of input streams by any of the well-known tree balancing algorithms, but this makes the program more complicated.

Even if the tree is completely balanced, a merger network with n input streams will have height proportional to $\log n$ and each message has to go through $\log n$ mergers. On the other hand, in procedural concurrent programming languages, using locking and destructive assignment, it is quite easy to realize a merger with constant overhead. As mergers are very frequently used in process and stream programming style of KL1, difference in order of computation is unbearable.

To solve this problem, KL1 provides mergers as one of its built-in features. Built-in predicate for stream merging has two arguments and invoked as "merge(In, Out)". When a message is sent to the input stream (that is, when the first argument is given its value a "cons" structure with the message as its "car"), it is simply forwarded to the output stream (the second argument is unified with another "cons" structure having the message as its "car"). The process starts working as a merger when, instead of a cons, a vector structure such as "{In_1, In_2, ..., In_n}" is given as its first argument. The merger then becomes an n input merger. The number of inputs can be increased by giving any one of the input "In_k" a vector value again. To decrease the number, simply closing input streams (that is, giving them a symbolic atom "[]" as their value) will suffice. When all the inputs are closed, the output is also closed (unified with "[]").

The built-in merger can forward messages with constant overhead irrespective of the number of input streams, and the constant is also much smaller than mergers defined in KL1.

5.5 Dispatchers

In the above program computing the total of both squares and cubes, both of the filters making squares and cubes required same inputs. If that is not the case, that is, if some of the messages should go to different processes depending its contents, how should it be programmed?

Let us, this time, consider a still more meaningless example, in which we compute the sum of squares of even numbers and cubes of odd numbers for natural numbers less than the given value. To do this, we can use a process which dispatches the numbers to one of two filters depending on whether the input is even or odd. Such a program can be defined as follows.

```
dispatch([], Odds, Evens) :- Odds = [], Evens = [].
dispatch([One|Rest], Odds, Evens) :- One/2 =\= 0 |
    Odds = [One|OddsTail],
    dispatch(Rest, OddsTail, Evens).
dispatch([One|Rest], Odds, Evens) :- One/2 =:= 0 |
    Evens = [One|EvensTail],
    dispatch(Rest, Odds, EvensTail).
```

The predicate defined by this program has one input stream and two output streams. The received messages are sent to either of the output streams depending on their contents. Such a process is called a *dispatcher*.

Using this predicate, the top level well be as follows.

```
queer_sum(N, Sum) :-
    naturals(N, Naturals),
    dispatch(Naturals, Odds, Evens),
    square(Evens, Squares),
    cube(Odds, Cubes),
    merge(Squares, Cubes, Both),
    sum(Both, Sum).
```

The overall process structure will be as shown in Figure 5.4.

5.6 Servers

Another important element for constructing process networks is processes called a *server*. Servers have an input stream which usually is connected to many *client* processes through a merger. Client processes send data to be shared among clients in messages to the server and the server keeps them, in some suitable format, as its state (actually, arguments of its recursive invocation). Clients can also read out the data by sending messages for query.

The process of counter explained in the previous chapter is a typical server process.

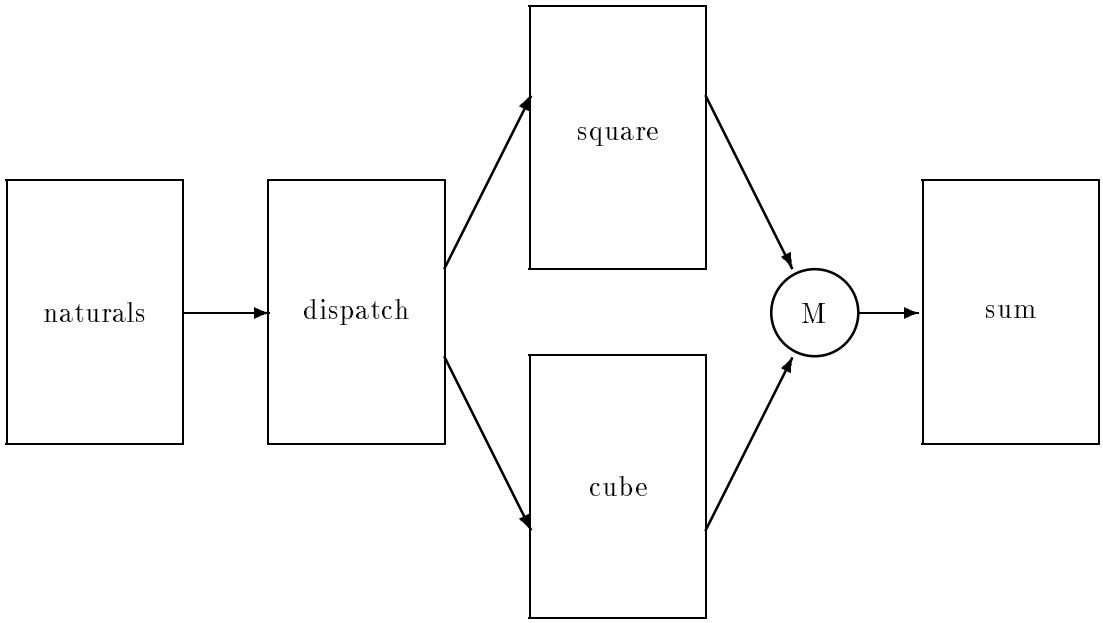

Figure 5.4: Process Network with a Dispatcher

Summary

KL1 programs are often constructed with many processes connected by streams. Typical element processes of such process networks and the ways to connect them are explained in this chapter.

Chapter 6

Specifying Physical Execution

The basics of the KL1 language and programming techniques for writing concurrent programs are explained in the previous chapters. This chapter explains about *pragma* which specifies how concurrent programs that *can* run in parallel should actually run in parallel.

6.1 Principles of Specification of Parallel Execution

Before going into details, the principles made in the design of KL1 on how to specify parallel execution are described in this section.

6.1.1 Programs Specify Load Distribution

To perform computation efficiently on actual parallel processing hardware, the following two aspects must be considered.

Distribution of Load: If computation is concentrated on one or small number of processors, the whole computation will not end until those processors finish the job, while making all other processors idle waiting for them. Thus, the problem to be solved *should* be divided into subproblems small enough and distributed to as many processors as possible, giving equal load to all the processors.

Reduction of Communication: To distribute subproblems, communication is needed for distributing such subproblems and also for gathering the results. Also, in most efficient algorithms, subproblems are not completely independent, requiring communication between processors solving different part of the problem during the computation. If too much communication is required, its cost may dominate the computation cost, sometimes making parallel execution less efficient than sequential execution. Thus, the problem *should not* be divided into too small subproblem to reduce communication between the solvers.

These two target of load distribution and communication reduction are in a contradictory relation: too much distribution may result in too much communication; too much reduction of communication won't allow load distribution. Thus, finding a method of division into subproblems without too much communication and finding a reasonable trade-off point of communication cost and load imbalance cost is a central issue of research and development of parallel processing software.

How a problem can be divided into subproblems depends heavily on the problem itself and algorithms used. When the amount of computation can be predicted accurately, as in computing products of many matrices, automatic load distribution might be able to do decent job. In the area of knowledge information processing, however, the result of solving one subproblem might drastically change the amount of computation needed to solve other subproblems, making automatic load distribution very difficult with the current level of parallel processing software technology.

As this is the status quo, KL1 is designed as a *tool* for the research automatic load distribution. Load distribution is thus not automatic by the language implementation, but specified in the programs. So as to ease experimentation of diverse load distribution methods, the language is designed to make changes in load distribution easy, which will be discussed in more detail in the next section.

6.1.2 Separating Concurrency and Parallelism

In KL1, *concurrency* and *parallelism* are separately specified. Here, we use the two words in the following meanings.

Concurrency specifies which part of the program *can be* executed in parallel. Concurrency affects the correctness of the programs. If program parts that should never be executed in parallel are made concurrent, the program might not work correctly. By specifying clause application condition in the guard, clause selection will be automatically suspended until data required for checking the condition become available. This *data-flow synchronization* implicitly specifies allowed concurrency.

Parallelism specifies which concurrency should be actually exploited for efficient parallel execution. This is specified by *pragma* in programs. As pragma only specifies which of concurrently executable portions of computation should really executed in parallel, it will never affect the correctness of the program. However, it may affect the *efficiency* of the program drastically.

During research of efficient load distribution methods, diverse methods have to be tested. If debugging of parallel programs, which, even done only once, already demands more effort than for sequential programs, has to be done for each time a new method is tested, the required effort might become enormous. By separating concurrency and parallelism, KL1 allows load distribution methods to be altered without affecting the correctness of programs.

6.2 Goal Distribution

For distribute computation, the *node* to execute a goal is specified. A node is a processor or a group of processors within which load distribution is automatic. As communication cost is relatively small among processors physically sharing a bus or memory, automatic load distribution may be tried among such group of processors. For large scale systems, however, such tight coupling may not be appropriate, making communication cost much higher.

Execution node specifications can be given to body goals with the following notation.

Goal`@node(`*Node*`)`

In the current implementation, nodes are specified by their sequential numbers.[1]

An example of a program specifying goal distribution follows.

```
p([One|Rest], N, State) :-
    q(One, State, NewState)@node(N),
    N1 := N + 1,
    p(Rest, N1, NewState).
```

In this example, each message received by "`p`" are handled on different node by the predicate "`q`". Goals without any node specification pragma (an addition and a recursive invocation of "`p`" in this example) are executed in the same node as the original goal.

6.3 Goal Priority

When there are more concurrently executable goals than the number of physical processors, the order of their execution sometimes affects the efficiency drastically. Thus, to suggest which goal should be executed earlier for better efficiency, goals can given a priority by a pragma. Priority specifications can be given to body goals with the following notation.

Goal`@priority(`*Priority*`)`

The *Priority* part gives concrete details of the specification, which is out of the scope of this introductory document.

Priority pragmas are not necessarily obeyed faithfully, as it may prevent efficient implementation. As pragmas do not affect the correctness of programs, the implementation may safely neglect some pragmas. In the current implementation, priority specification is effective only among goals in the same node.

6.4 Clause Priority

When more than one clause have their guard condition satisfied, the language does not define which will be selected. Programs should be written so that any selection will yield the correct result. However, even if the program is written so that selection of clauses will not affect the correctness, it does sometimes affect efficiency of programs considerably. Priority between clauses can be specified by a pragma to suggest which clause should be selected for better efficiency.

To specify priority among clauses, write clauses with higher priority first, a pragma "`alternatively.`" which looks similar to a clause next, and write the rest of clauses last. The "`alternatively.`" pragma can appear more than once. No priority relation is

[1] When the system is shared with multiple users by dividing the nodes into node groups for each user, logical node numbers beginning with zero are given for each such partition.

assumed within clauses before the first"`alternatively.`", after the last one, or between two of them.

Clause priority specification, also, might be neglected sometimes, for more efficient implementation.

Summary

This chapter described *pragma* feature of KL1, which suggests the language implementation the way to realize more efficient processing. It may be worthwhile to note again that pragmas are only suggestions for efficiency, not specifying how the implementation should behave rigidly. Pragmas can be neglected sometimes. Thus, programs cannot depend on pragmas to be correct.

Bibliography

[1] David L. Bowen, Lawrence Byrd, Fernando C. N. Pereira, Luis Moniz Pereira, and David H. D. Warren. *DECsystem-10 Prolog User's Manual*, November 1983.

[2] Takashi Chikayama, Hiroyuki Sato, and Toshihiko Miyazaki. Overview of the parallel inference machine operating system (PIMOS). In *Proceedings of FGCS'88*, Tokyo, Japan, 1988.

[3] Keith L. Clark and Steve Gregory. Parlog: A parallel logic programming language. *ACM Transaction on Programming Languages and Systems*, 8(1), 1986.

[4] Ehud Shapiro. A subset of Concurrent Prolog and its interpreter. ICOT Technical Report TR-003, ICOT, 1983.

[5] Ehud Shapiro and Akikazu Takeuchi. Object oriented programming in Concurrent Prolog. ICOT Technical Report TR-004, ICOT, 1983. Also in New Generation Computing, Springer-Verlag Vol.1 No.1,1983.

[6] Akikazu Takeuchi and Kouichi Furukawa. Parallel logic programming languages. In Ehud Shapiro, editor, *Proceedings of Third International Conference on Logic Programming*, pages 242–254, Berlin, 1986. Springer-Verlag.

[7] Kazunori Ueda. Guarded Horn Clauses. ICOT Technical Report TR-103, ICOT, 1985.

[8] Kazunori Ueda and Takashi Chikayama. Design of the kernel language for the parallel inference machine. *The Computer Journal*, December 1990.

Index

active unification, 8
arithmetics, 15
atom, 14

body, 7
built-in predicate, 9

car, 16
cdr, 16
clause, 6
client, 34
communication, 9
comparison, 14
concurrency, 9, 37
cons, 15, 16

data-flow synchronization, 37
dispatcher, 34

failure, 11
filter, 29
floating-point number, 14
functor, 16

goal, 7
guard, 7

head, 6

identity, 18
incomplete data structure, 17, 18
    as message, 27
incomplete message, 27
integer, 14

list, 15, 16

macro notation, 14
merger, 31
message, 25
    incomplete ..., 27
    structured ..., 26

node, 37
nondeterminacy, 12, 32
number, 14

parallelism, 37
passive unification, 8
pragma, 36, 37
predicate, 6
    built-in ..., 9
    user-defined ..., 9
process, 21, 22

reduction, 9

server, 34
shared variable, 9
state, 26
stream, 25
string, 15, 16
structure, 15
structured message, 26
suspension, 10
symbolic atom, 14
synchronization, 9

unification, 8
    active ..., 8
    of structures, 17, 18
    passive ..., 8
user-defined predicate, 9

variable, 13
    shared ..., 9
vector, 15, 16