ANNOTATIONS FOR DISTRIBUTED PROGRAMMING IN LOGIC

Raghu Ramakrishnan & Avi Silberschatz

Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712

TR-85-15 September 1985

Annotations for Distributed Programming in Logic †

Raghu Ramakrishnan
Avi Silberschatz

Department of Computer Science
University of Texas
Austin, TX 78712

ABSTRACT

It has been recognised that languages like Concurrent Prolog and Parlog which use committed choice non-determinism have departed from the original concept of logic programming, but no new paradigm has been suggested. In this paper we propose that programs in such languages be viewed as rewrite rules that hierarchically decompose a process into a network of distributed processes. Logical variables and unification provide a powerful means of communication, and annotations can provide a synchronization mechanism that complements them.

1. Introduction

Horn Clause logic programming [1, 2] has aroused a great deal of interest in the past decade because of its declarative semantics and potential for parallel execution. Languages like Prolog [3] and IC-Prolog [4] have tried to realize the Horn Clause paradigm, as advocated by Kowalski [1], by retaining the declarative semantics and giving the programmer some control over the execution. These attempts have not been entirely successful since none of these languages preserves logical completeness, thus corrupting the declarative semantics. This has been partly due to their depth-first execution strategy (which raises the possibility of an infinite computation) and also due to annotations such as the *cut*, which prune the search space with nothing to ensure that no valid solutions are pruned. All this notwithstanding, it is possible for a careful programmer to write code that can be understood declaratively, thus making them reasonable approximations to the logic programming paradigm.

Unfortunately, extracting the parallelism inherent in Horn Clauses has proved to be a difficult task, and this has prompted the design of languages such as Concurrent Prolog (CP) [5] and Parlog [6] which use *committed choice* non-determinism. Essentially, these languages force the programmer to non-deterministically 'choose' one of the alternatives at each OR-node in the search tree. Such a drastic reduction of the search space simplifies concurrent execution, but clearly logical completeness and the Horn Clause paradigm are hopelessly compromised. While it has been recognized that these languages have departed from the Horn Clause model [7], no new paradigm has been developed, and one of the contributions of this paper is to present such a paradigm.

† This research was partially supported by NSF Grants MCS81-22039, MCS82-14613 and MCS81-04017 and ONR Contracts N00014-80-K-0987 and N00014-83-K-0490.

Nonetheless, these languages continue to be interesting for two reasons. First, they greatly simplify the problem of a parallel implementation of Horn Clauses while retaining sufficient issues that pose a challenge. The experience gained in the implementation of such languages could be invaluable in the implementation of full-fledged Horn Clauses. Second, research in the use of these languages has shown their potential in several applications. Future versions of these languages with more sophisticated programming support could well be important programming tools in their own right.

The execution of programs in such languages leads to a set of concurrent processes, and logical variables are used as communication channels between these processes. Different variable annotations have been proposed as a means for synchronization, but they all seem to have been designed with the mechanics of unification and the search tree model of logic programming in mind. This paper looks at such annotated variable designs and argues that we need to change the way we think about them and about the programs in which they are used. We do not look at annotations such as the cut, which is used to restrict the search space, or the annotations in IC-Prolog which are used to increase efficiency (by suggesting indexing strategies or guiding execution) in languages that retain OR-parallelism.

In Section 2 we look at variable annotations in Concurrent Prolog, Parlog, a proposal by Saraswat [8] and Guarded Horn Clauses [9]. We discuss their shortcomings and argue in Section 3 that programs in such languages are better thought of as a set of rewrite rules for hierarchically decomposing a process into a system of distributed processes, rather than as logic programs, and that annotations should be designed from this perspective. We then describe the design of such an annotation and list certain syntactic restrictions required to enforce its semantics. Section 6 shows how these syntactic restrictions can be enforced by a compile-time check. Section 7 demonstrates that the new annotation supports the programming techniques available in Concurrent Prolog (which has the most powerful annotation of all the languages considered) and we present our conclusions in Section 8.

2. Synchronization in Committed-Choice Languages

Committed-choice languages generally allow clauses of the form:

$$\langle\text{head term}\rangle \ \texttt{<--}\ \langle\text{guard terms}\rangle \mid \langle\text{body term}_1\rangle, \ldots ,\langle\text{body term}_k\rangle$$

The important language decisions include control constructs for ordering the evaluation of clauses (as, for example, in Parlog), restrictions on the guards including the variables accessible to them and the variables they can bind, and the annotations for synchronization and communication. We confine ourselves to the last of these issues. Many of the issues involved in the design of such languages stem from the interaction of these design choices, but we will not examine them in detail since they are not central to the point we wish to make.

In this section we look at some of the existing committed-choice languages and the annotations they provide for synchronization and communication. We begin by examining Concurrent Prolog and then discuss Saraswat's annotation, Parlog and Guarded Horn Clauses.

Concurrent Prolog has been the subject of extensive discussions in the Prolog Digest (a network mailing group). It has also been examined at length in [8]. The read-only annotation in Concurrent Prolog was originally defined as follows. Let X? be an annotated term, T an unannotated term and Y an unannotated variable. Unify(X?,T) succeeds with X bound to T if X is a variable, else X is recursively unified to T. The behaviour of Unify(X?,Y?) is not clearly defined. It could suspend until X and Y are instantiated or X and Y could be recursively unified. The unification of X? with Y suspends until X is instantiated if Y is a non-variable. If Y is a variable it succeeds with Y made read-only and bound to X.

This definition leads to the following undesirable characteristics:

1. Unification becomes order-dependent.
2. ?-annotations in clause heads can cause deadlocks depending on the order of unification.
3. Static analysis is not enough since ?-annotations can be propagated at run-time, leading to many unexpected behaviours.

Saraswat discusses these issues at length and suggests another annotation. This annotation (|) can only be used in clause heads and it is not passed on like the ?-annotation is. Unify(X|,Y) suspends until Y is instantiated and then unifies X and Y recursively. If it annotates term T1 which is a subterm of term T, all sub-terms of T (including T) that contain T1 must also be |-annotated. The semantics are defined with respect to a parallel unification algorithm and are thus trivially independent of the order of unification.

This alleviates many of the problems with the read-only variable, and as Saraswat points out, all CP programs presented so far can be easily translated using the |-annotation. However, the notion of synchronization can only be understood in terms of unification - we can only make statements of the form "Process P will wait until the argument in the invoking goal which matches term T in the head of P has been instantiated". This does not preclude the possibility that the unification of head and goal may itself instantiate the required argument.

This makes it difficult to attach a semantics that describes the synchronization of a process with respect to other processes in the network since a process may 'feed itself' in the course of unification! A rough analogy is a process waiting on a semaphore that is allowed to signal itself. If we are to use our experience with more conventional distributed programs to understand programs that use logical variables for communication, we must have a synchronization mechanism that matches our intuition, and allows us to make statements of the form "Process P will wait until some (other) process Q instantiates a variable (sends a message)".

The OR-relations subset of Parlog (whose origins can be traced back to Clark and Gregory's Relational Language [10]) comes closer to this objective. The proviso that goal variables cannot be instantiated during unification with a clause head (in fact, until execution of the guard is complete) effectively resolves the problem we pointed out in Saraswat's annotation. However, this appears to be far too strong a restriction. While it is desirable to prevent a process from instantiating a variable that it is waiting upon, preventing it from instantiating any variable in the goal (until completion of the guards) unduly sacrifices a lot of the expressiveness of logical variables. We must note, however, that this decision in Parlog was prompted by several other factors, mostly related to the nature of the goal computations.

Further, Parlog insists upon each argument of a clause head being given a mode (input or output), and allows only matching (from or to the goal). Again, these restrictions were motivated partly by efficiency considerations, but they are overly restrictive. We feel that the synchronization mechanism should be designed so as to retain as much of the expressive power of the logical variable as possible. It should also allow the extraction of as much parallelism as the program permits.

Guarded Horn Clauses (the proposed core of the Kernel Language for ICOT's Parallel Inference Machine) take an interesting approach. Their philosopy is close to ours in that they explicitly think in terms of processes and communication. They eschew any form of annotation and rely entirely upon the definition of the guards for synchronization. They place the restriction that the unification of the goal and the head and the computation of the guards may not instantiate any variables in the goal. If such an instantiation is required for progress, the process must suspend until some other process binds the goal variable. This is essentially the same mechanism we saw in Parlog. The Parlog requirement that all arguments be moded is absent (In fact, there are no modes). The result is a very simple language.

We again observe that this sacrifices some of the expressiveness of logical variables and imposes more sequencing than is strictly necessary. The reason is that, in effect, all variables in the goal are treated as variables to be synchronized upon. This also imposes a high execution time overhead by unnecessarily suspending several processes. Our approach considers the alternative of retaining the underlying view of distributed processes and using annotations to make the language more expressive and efficient, at the cost of some syntactic elegance.

3. The Proposed Model of Computation

The Horn Clause paradigm is inappropriate for languages that use committed choice non-determinism because the completeness of the declarative semantics has been lost. Given a solution (ie a set of bindings for the input arguments) we can assert that the logical implications of the clauses are satisfied. However, given that a solution does exist for a given goal, we cannot guarantee that it will be found (and of course, we cannot find all such solutions). So if a program fails to find a solution, in what terms do we understand its failure? Is there no solution, or is it just that one particular (non-deterministic!) execution sequence failed to find it?

We suggest that it is better to think of these programs as rewrite rules. The clauses in a program are viewed as templates which can be used to reduce (rewrite) processes. We interpret them as follows. In any network, a process which matches the template in the head of the clause can be replaced by the network of processes represented by the terms on the right hand side of the clause. Since the terms in the guard must be satisfied before we choose the rewrite rule to apply, in effect we replace the original process by the network of processes in the body. If there are several clauses with heads that match a given goal, these can be matched with the goal and their guards invoked in parallel (using local environments if necessary). Each of these clauses represents one potential rewrite rule that can be used to reduce the goal. The process of matching the heads and satisfying the guards is used to determine which of these is to be applied. Given a set of clauses such that their heads match with the given goal and their guard computations terminate successfully, one of them is chosen non-deterministically. Each of these clauses represents an acceptable rewrite rule.

Once we choose a rewrite rule, the process representing the goal is replaced by a set of processes corresponding to the goals on the right hand side of the selected rewrite rule. This is a network with the same external interface as the original process in that the global variables (channels) that are accessed by the processes in it are the same variables accessed by the original process. Thus we see an hierarchical decomposition of a process into a network of processes with the same external interface. This process structure is very similar to the networks discussed in [11] although the nature of the channels is completely different. The order in which each of the processes in this network is further reduced is again non-deterministic.

Kahn described a simple model of computation using communicating processes [12,13] wherein the communication consisted essentially of streams. van Emden and de Lucena Filho demonstrated how computations in this model could be realized in Logic [14]. A close examination of their work reveals that the subset of Logic used in this realization is such that there is precisely one acceptable rewrite rule at each OR node. Further, as they point out, AND siblings are chosen for reduction in a certain (implicit) order. The other possible orders represent interesting variants of Kahn's model wherein certain processes 'run ahead' of their input streams.

This work can be viewed as the kernel of our proposed model. However, the languages we are considering introduce some features that go beyond the simple subset of Logic which serves to emulate Kahn's model. First, there are two sources of non-determinism that need to be controlled; the choice of a rewrite rule and the choice of a goal to expand. The latter can be resolved by expanding all goals in parallel. This presents no difficulties in itself, but it makes the former choice more difficult. The second feature that goes beyond the subset of Logic discussed by van Emden and de Lucena adds to this difficulty. It is the fact that communication can no longer be thought of as just streams. A single variable can serve as the vehicle for to and fro communication (each process binds it to a list consisting of the message and a variable which is used to continue the dialogue).

So, to program effectively using this paradigm, we need some means of synchronizing the various processes. This can be done entirely within the guard computations by including sufficient goals in the guard and using a variable binding strategy similar to Parlog or GHC. However, there could be situations wherein the synchronization conditions are difficult to express in this manner and annotations could provide a more natural and efficient mechanism. Further, annotations allow us to gain more parallelism, as was pointed out earlier.

4. The New Annotation '%'

The annotation '%' has been designed using a process oriented approach. It has the same semantics whether it annotates a variable in a call or in a clause head. It is a guarantee that the process in which it appears (it may appear either in the call which specifies the required process or a clause head being matched in an attempt to spawn such a process) will not instantiate it to a non-variable. In other words, such an instantiation must be due to some other process (a sibling of the call or a sibling of an ancestor), and this fact can be used to make the process wait for input (which may in fact decide the nature of this process since attempts to match the call against a clause head may be delayed until the input is available). These semantics are guaranteed by the definition regardless of the order of unification.

The unification of %-annotated terms and variables is defined below for the case where they appear in a call and for the case where they appear in a clause head. Further, some syntactic requirements must be met in order to obtain the desired semantics and these are specified along with the rationale behind them. In the rest of this paper we use the following convention unless otherwise stated. A capital letter from the end of the alphabet (e.g., X, Y) denotes a variable, terms are specified by 'Term', and all variables and terms are assumed to be unannotated unless explicitly shown with a postfixed '%'.

We first consider the use of '%' in clause heads. If it annotates a variable, say X, then this is a guarantee that X will be instantiated to a non-variable by some external process. This is achieved by the rule that X% in a clause head will unify with Term% (in a call) with X instantiated to Term, will unify with Y% with X bound to Y, and will fail otherwise. We could have defined unification with Y% to suspend, but we allowed it to proceed, binding X to Y, in order to extract as much concurrency as possible. The value of X is not really needed until it has to be matched with a non-variable, and our decision essentially ensures that X will be instantiated to the same value as Y (by some other process of course!). Note that Saraswat's |-annotation is defined to suspend in this case, thus sacrificing some parallelism. Unification with an unannotated term or variable in the call is defined to fail in order to avoid instantiating X% during the process of matching the call against the head. To illustrate this consider Example 1 below.

Example 1

To illustrate why unification with an unannotated term or variable should be defined to fail consider the following:

```
Call:   Equivalence(Y, Y)
Clause: Equivalence(5, X%) <--- ...
```

Suppose that it is not. Clearly, the two Y's equivalence X to 5, and a left-to-right unification would succeed with X being instantiated to 5. However, a right-to-left unification would instantiate Y to X, with or without preserving the '%' annotation depending on our definition of '%'. The second definition would cause unification to succeed with X instantiated to 5. The first would suspend until some other process instantiated Y to a non-variable, which is what we want. There are two unpleasant aspects to this example; first, the semantics could be violated in that this process instantiates one of its %-annotated variables, and second, a different order of unification yields a different result. The fact that this other result is what we want in this paticular example is secondary since we desire a semantics that can be understood without following the details of the unification, which is really an implementation issue. Note that defining unification of a %-annotated variable in the clause head with an unannotated variable in the call to fail, while allowing unification with an unannotated term in the call to succeed (or vice-versa) will not work either.

Using Saraswat's annotation, this example would always succeed in instantiating X to 5. This behaviour is order-independent, but is clearly an example of a process 'feeding itself' as we phrased it earlier, and thus violates the semantics we desire.

If both the occurrences of Y were annotated with '%' unification would suspend until some other process instantiates Y regardless of the order of unification, consistently yielding the semantics we desire. □

When Y% appears in a call, unification with Term% in a clause head suspends until Y has been instantiated by some other process, unification with X% in a clause head succeeds with X bound to Y and unification with Term or X (not annotated) in the clause head is defined to fail. This prevents the annotation % from being 'passed on' due to unification, and the annotation of all clauses and calls becomes static, making it much easier to understand programs. Since the '%' annotation (and in fact all such annotations, by their very nature) determines how the process described by a clause behaves, allowing this annotation to be inherited at run-time through unification implies that this behaviour can only be understood by following the dynamic propagation of '%'. Disallowing this ensures that we can completely understand the process described by a clause by a static reading of that clause (and of course, other clauses that it calls upon), without regard to the actual calls which match this clause. As we shall see later, it also allows us to detect any incorrect uses of '%' at compile time.

The previous paragraphs dealt with the use of Y% in calls and clause heads. They also implicitly indicate our treatment of Term% in calls and heads. Term% in either a call or a head can only match a %-annotated variable or term. When it appears in a head, it causes suspension till the corresponding (%-annotated) argument is instantiated to a term. In a call, it is used when (and only when) the corresponding argument in the target clause head is %-annotated. We may think of %-annotations as part of the specification of the desired process. We thus use %-annotated terms in the call to provide a guarantee that the process defined by a clause head is being correctly used, an approach that is along the same principles as strong typing elsewhere. Note that nothing has been said about the instantiation of the variables in the %-annotated terms. To illustrate this consider Example 2 below.

Example 2

This example emphasizes the fact that no assurance is given as to which process instantiates the variables *inside* a %-annotated term. Consider the following:

```
Call:   No_guarantee([X|Y]%, X, Y)
Clause: No_guarantee(Z%, 5, 6) <--- ...
```

Unification always succeeds, instantiating X to 5 and Y to 6. The only fact guaranteed by the %-annotation, that the argument corresponding to Z will be instantiated to a term by some other process, has been satisfied trivially by virtue of the corresponding argument being a term [X|Y]. This term, moreover, is %-annotated, thus indicating that this call is to be matched with a clause that expects to find its first argument instantiated to a term; and since this is precisely what the given clause expects, unification succeeds. []

We summarise these rules in Table 1. The notation we use is as follows. GL refers to the argument in the goal, HD refers to the argument in the clause head, and the table specifies the unification algorithm for all combinations of these arguments. The entry 'Unify(GL,HD)' indicates GL and HD are terms and are unified recursively as usual. The entry 'HD = Ref(GL)' indicates that HD is bound to GL (or instantiated to GL, if GL is a term). 'Suspend(GL)' indicates that unification suspends till GL is instantiated.

There is another important case of unification to consider. We mentioned earlier that we could use the guards to equivalence two variables. This becomes important due to the restriction that variables in a clause head be unique. We use the operator '=' to do this. This operator unifies its two arguments recursively, and if one of them is a %-annotated variable it suspends until it has been instantiated. The %-annotation is ignored when it decorates a term (we think of this as a %-annotated variable that has been instantiated as required by the %-annotation). Suspension occurs even when both arguments are %-annotated variables. This is because binding the two variables and allowing unification to proceed will result in both of the variables being instantiated when one of them is, thus violating our semantics that this process will never instantiate one of its %-annotated variables.

The intended semantics for '%' can be achieved with these rules only if some syntactic restrictions are met. These are described and motivated in the next section.

HEAD / GOAL	TERM	VAR	TERM%	VAR%
TERM	Unify(GL,HD)	HD = Ref(GL)	Fail	Fail
VAR	GL = Ref(HD)	HD = Ref(GL)	Fail	Fail
TERM%	Fail	Fail	Unify(GL,HD)	HD = Ref(GL)
VAR%	Fail	Fail	Suspend(GL)	HD = Ref(GL)

Table 1. Unification Rules for %-annotated Variables and Terms

5. Syntactic Restrictions

The rules defined in Section 4 for unification do not, by themselves, give us the semantics we want. There are several ways in which the requirement that a process should not instantiate one of its %-annotated variables can be breached. The following syntactic restrictions are designed to stem these breaches, and we present a set of examples illustrating (and justifying) them after first presenting them below.

We require that variables in a clause head occur just once in that head, and that the use of % in a call be consistent in that either all or no occurrences of a variable Y in a call are annotated. If % annotates a term or a variable in a clause head, all superior terms (all terms which contain the annotated term or variable) must also be %-annotated. Further, if X% appears in a clause head, all occurrences of X on the right hand side of the clause must be %-annotated. This is because X might be bound to some Y (%-annotated) in the call, and this process must not instantiate X (and thus Y). These restrictions are necessary to ensure the desired semantics, but they are sensible requirements. (The requirement that all variables in a clause head occur just once cannot be justified on its own merits, but it is not overly restrictive since we can always use the guards to state that two given variables must be instantiated to unifiable terms.) An efficient compile-time check suffices to enforce these rules.

The idea behind these restrictions is again to prevent this process from instantiating a %-annotated variable during the process of matching the call against the head. Examples 3 through 5 illustrate why they are needed.

Example 3

This example shows why variables in the clause head must be unique. Consider the following:

```
Call:   Sneaky_Equivalence([X|Y%], [X|5%])
Clause: Sneaky_Equivalence(Z, Z) <---  ...
```

Left-to-right unification will succeed with Y being instantiated to 5. Right-to-left unification will suspend until some other process instantiates Y, and will then try to unify it with 5. Note that %-annotating the two Z's and the corresponding terms would still yield the same undesirable behaviour. The problem lies in the fact that the Z's equivalence Y with 5. It might be argued that this process will in any case fail if Y is not eventually instantiated to 5, and there is no harm in letting this failure be discovered later. The flaw lies in the fact that there might be several alternative clauses (in other words, potential process descriptions). The call, as we observed earlier, is the specification of a needed process, and this entire exercise of matching with the clause heads and trying to satisfy the guards is intended to find a process template (one of the clauses) which meets the specifications. If, for instance, we assume an empty guard, the choice is made once the unification of the call with a clause head succeeds. The above order-dependent behaviour could thus lead to the call reducing to a process that fails or one that succeeds. Such a possibility is inherent in any committed-choice logic programming language, but it should at least be independent of things like the order of unification! The fact that this objective is compromised by some extra-logical features should not be interpreted as licence to design other extra-logical features with the same deficiency. □

Example 4

This illustrates why variables in a call must be annotated consistently, consider the following.

```
Call:    Fickle(X%, X)
Clause:  Fickle(Y%, 5) <--- ...
```

Left-to-right unification will suspend until some other process instantiates X but right-to-left unification will succeed with X and Y instantiated to 5. If both occurrences of X in the call are %-annotated, this problem does not arise since unification always suspends until X is instantiated by another process. []

Example 5

If a term T in a clause head contains a %-annotated subterm T1, then every sub-term of T including T must be %-annotated. This is because the call call could equivalence two terms in the head as in this example.

```
Call:    Machiavelli(Z, Z)
Clause:  Machiavelli([5|Y%], [5|6%]) <--- ...
```

Left-to-right unification will suspend whereas right-to-left unification will succeed with Y instantiated to 6. However, if both the terms in the clause head as well as the two Z occurrences are %-annotated, unification will always suspend until Z is instantiated by another process, hopefully to some term of the form [_|U%], where the underscore stands for an arbitrary term or variable, and U is either '6' or a variable. At this point, unification will again suspend if U is a variable. []

6. Compile Time Checks

Compile time checking can be used to enforce all of the syntactic restrictions described above. We summarise these compile-time checks:

1. Each variable occurrence in a clause head is unique.
2. A %-annotated variable in a clause head is only used as a %-annotated variable throughout that clause.
3. All superior terms of a %-annotated term or variable in a clause head are %-annotated.
4. Each %-annotated variable on the right hand side of a clause that does not appear in the clause head is:

   a. Used consistently in each call (ie. it is always %-annotated or always unannotated).
   b. Appears in at least one call without being annotated.

If a given program passes these checks, the unification algorithm described above ensures the intended semantics. If a process suspends, it is waiting for some other process to instantiate a %-variable. Incidentally, 4.b above detects one obvious instance where such a wait could be futile. The problems involved in detecting such failures are the familiar starvation and deadlock problems of traditional process models since we have eliminated the side effects of unification which enable a process to unexpectedly 'feed' itself!

The only other way in which the annotations intrude (in the sense that they sometimes lead to unexpected situations) is that a call may fail because no clause head matching the annotations is found. We view this as being in the same class of errors as there not being a clause head with the same functor as the call or the same number of arguments since the annotations are as much a part of the static description of a clause as the functor name or number of arguments.

7. The Programming Techniques Supported by '%'

The %-annotation supports all the programming techniques that the read-only annotation in Concurrent Prolog does. It does not allow us to create protected channel variables (ie. specify that a process always expects to find a given argument to be a variable), and it is not intended to do so. We note that the current definition of the read-only annotion in CP, as opposed to the original definition, does not permit this either.

We present some more examples to support this claim. These are programs taken from Shapiro's paper 'A Subset of Concurrent Prolog' and expressed using the %-annotation.

Example 6

This is an implementation of Quicksort in CP with the %-annotation. The logic is exactly as in Shapiro's original program.

```
quicksort(Unsorted%, Sorted) <--- qsort(Unsorted%, Sorted-[]).

qsort([X|Unsorted]%, Sorted-[]) <---
   partition(Unsorted%, X%, Smaller, Larger),
   qsort(Smaller%, Sorted-[X|Sorted1]),
   qsort(Larger%, Sorted1-[]).

qsort([]%, []-[]).

partition([X|Xs]%, A%, Smaller, [Y|Larger]) <---
   A < X, X = Y | partition(Xs%, A%, Smaller, Larger).

partition([X|Xs]%, A%, [Y|Smaller], Larger) <---
   A >= X, X = Y | partitioned(Xs%, A%, Smaller, Larger).

partition([]%, _%, [], []).
```

While the logic is the same as in Shapiro's program, the above version has one important difference. Each clause describes precisely what annotations it expects of its arguments. This information is available without looking at any of its calls, and so each clause can be understood by considering it statically, with the assurance that nothing unexpected will happen at run-time due to some annotations that are passed on during execution. This program also illustrates the use of matching in guards in order to keep variable occurrences in the clause head unique.[]

References

1. Kowalski, R.A. "Predicate Logic as a Programming Language"
Proceedings of the IFIP Congress 1974, pp 569-574.

2. van Emden, M.H. and Kowalski, R.A. "The Semantics of Predicate Logic as Programming Language"
JACM 23, 1976.

3. Clocksin, W.F. and Mellish, C.S. "Programming in Prolog"
Springer-Verlag, 1981.

4. Clark, K.L., McCabe, F.G. and Gregory, S. "IC-Prolog Language Features"
Logic Programming, Clark, K.L. and Tarnlund, S-A. (Eds.), Academic Press, 1982, pp 253-266.

5. Shapiro, E.Y. "A Subset of Concurrent Prolog and its Interpreter"
Technical Report TR-003, ICOT, February 1983.

6. Clark, K. and Gregory, S. "Parlog: Parallel Programming in Logic"
Research Report DOC 84/4, April 1984 (Revised June 1985), Department of Computing, Imperial College of Science and Technology.

7. Kowalski, R.A. "Directions for Logic Programming"
Proceedings of the 1985 Symposium on Logic Programming, pp 2-7.

8. Saraswat, V. "Problems with Concurrent Prolog"
Technical Report, Computer Science Department, Carnegie-Mellon University, June 1985.

9. Ueda, K. "Guarded Horn Clauses"
Technical Report TR-103, ICOT, June 1985.

10. Clark, K.L. and Gregory, S. "A Relational Language for Parallel Programming"
Proceedings of the 1981 ACM Conference on Functional Programming Languages and Computer Architecture, Portsmouth, October 1981, pp 171-178.

11. Chandy, K.M. and Misra, J. "Proofs of Networks of Processes"
IEEE Transactions on Software Engineering, Vol. SE-7, No 4, July 1981, pp 417-426.

12. Kahn, G. "The Semantics of a Simple Language for Parallel Programming"
Proceedings of the IFIP Congress, 1974, pp 471-475.

13. Kahn, G. and MacQueen, D.B. "Coroutines and Networks of Parallel Processes"
Proceedings of the IFIP Congress, 1977, pp 993-998.

14. van Emden, M. and de Lucena Filho, G.J. "Predicate Logic as a Language for Parallel Programming"
Logic Programming, Clark, K.L. and Tarnlund, S-A. (Eds.), Academic Press, 1982, pp 189-198.