Static Analysis of AND-parallelism in Logic Programs based on Abstract Interpretation
추상해석법을 이용한 논리언어의 AND-병렬 태스크 추출 기법

Hiecheol Kim · Yong-Doo Lee
Division of Computer & Communication Engineering
Taegu University
김 희 철 · 이 용 두
(대구대학교 정보통신공학부)

Abstract

Logic programming has many advantages as a paradigm for parallel programming because it offers ease of programming while retaining high expressive power due to its declarative semantics. In parallel logic programming, one of the important issues is the compile-time parallelism detection. Static data-dependency analysis has been widely used to gather some information needed for the detection of AND-parallelism. However, the static data-dependency analysis cannot fully detect AND-parallelism because it does not provide some necessary functions such as the propagation of groundness. As an alternative approach, abstract interpretation provides a promising way to deal with AND-parallelism detection, while a full-blown abstract interpretation is not efficient in terms of computation since it inherently employs some complex operations not necessary for gathering the information on AND-parallelism. In this paper, we propose an abstract domain which can provide a precise and efficient way to use the abstract interpretation for the detection of AND-parallelism of logic programs.

I. Introduction

Due to ease of programming and high expressive power derived from the declarative semantics, logic programming has been increasingly used for a wide range of application areas such as symbolic computing. Parallel logic programming has been recognized as a promising way to improve the performance of logic programming since logic programs frequently require a large amount of computation as they become more realistic. Indeed, many researchers have pursued the goal of designing efficient techniques for parallel logic programming systems.

One major trend to adopt logic programming for parallel execution is to try to maintain the standard semantics of Prolog and let the compiler and the runtime system extract the parallelism hidden in the program. Parallelism in Prolog can be exploited in different levels. In the clause/procedure level, we can exploit medium grained parallelism such as AND- and OR-parallelism. As the runtime test is usually very expensive, compile-time analysis becomes crucial for the detection of the parallelism of the program.

There are two important kinds of information that enables to exploit AND-parallelism in Prolog programs, variable sharing and groundness propagation. Two different approaches are frequently used to obtain the information. One approach is based on the static analysis of some relations, to be used in the exposition of the AND-parallelism, such as data-dependency.(Chang,1985, Debray,1986) For example, the SDDA(Static Data-Dependency Analyzer) detects the data-dependency at compile-time to gather the variable coupling information.(Chang,1985) It is later enhanced to deal with propagation of groundness by using the strong coupling concept.(Chang,1985) The other approach is based on the abstract interpretation that is frequently used as an elegant and sound framework for code optimization of declarative languages. (Abramsdy,1987, Bruynooghe,1987, Cousot,1977, Getzinger,1993, Jacobs,1991, Jacobs,1989, Mellish,1985, Mellish,1986, Xia,1988)

Even though static data-dependency analysis and abstract interpretation is useful for gathering some necessary information for the detection of AND-parallelism, they usually have some problems. The static data-dependency analysis is not sufficient to fully detect the AND-parallelism due to its limited capability with such functions as the propagation of groundness. On the other hand, the full-blown abstract interpretation is not efficient in terms of computation since it inherently employs some complex operations not necessary for gathering the information of AND-parallelism. In order to gather the information precisely and efficiently, a proper abstract domain is needed. The goal of this paper is to propose an abstract domain designed subject to the functionality and efficiency.

<Table 1> Conditions for Abstract Interpretation

Condition	Intuition
1) α and γ are monotonic	The order is preserved in conversion (but might be approximate).
2) $\forall d \in D: d=\alpha(\gamma(d))$	Concretization followed by abstraction is exact.
3) $\forall e \in E: e \subseteq \gamma(\alpha(e))$	Abstraction followed by concretization is accurate, but approximate.
4) $\forall d \in D$: $E_P(\gamma(d)) \subseteq \gamma(D_P(d))$	Abstract interpretation safely mimics concrete interpretation (but might be approximate).

This paper is organized as follows. Section 2 briefly introduces the parallelism in logic programs, the static information required for the detection of AND-parallelism, and finally the concept of abstract interpretation and its framework. Section 3 provides an in-depth discussion on the abstract domain that we developed for AND-parallelism. In section 4, the performance of the proposed domain and its implementation issues are briefly discussed. Finally, concluding remarks and the suggestion of future research are offered in section 5.

II. Background

This section provides an introduction to the parallelism in logic programs, the information required to detect independent AND-parallelism follows, and the outline of abstract interpretation of its framework. Due to space limitation, the detailed description of domain theory and the abstract interpretation are not exposed here. They are found in some other tutorials or references.(Abramsdy,1987, Bruynooghe,1987, Jacobs,1991)

1. Parallelism in Logic Programs

Among many opportunities of parallelism in Prolog program, AND- and OR-parallelism are two important types of parallelism. A Prolog program is normally represented by an AND/OR tree due to its simple syntax and regular structure.

Although we omit a more detailed discussion of the AND/OR tree, it should be noted that the AND/OR tree describes the parallelism which exists in Prolog programs. The AND/OR tree of a logic program reveals that some branches of the tree can be executed in parallel. When the partitioning is done at a clause node, to make calls to subgoals in parallel is referred to as AND-parallelism. OR-parallelism comes from the observation that there are usually multiple clauses with the same predicate symbol in their head literals. When the program execution is partitioned for each branches of a predicate node, i.e., for each alternative clauses, the parallelism exploited is referred to as OR-parallelism.

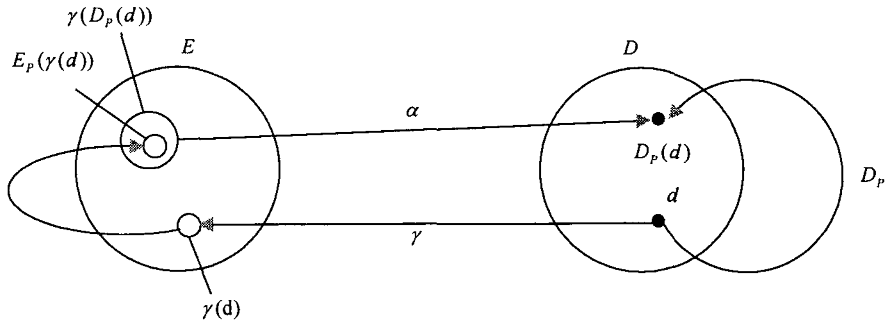
The task which executes one of the OR-branches can continue with the next subgoal in its parent's clause. In the AND-parallelism, when the parallel execution is restricted only to the subgoals which do not interact with each other without any share variables, the AND-parallelism is, in particular, referred to as independent AND-parallelism. Independent AND-parallelism is the main concern of this paper. Hereafter, we will call it just as AND-parallelism.

2. Abstract Interpretation Frameworks for Logic Programs

Abstract interpretation provides a good theoretical model which helps a wide range of program analyses.(Cousot,1977) It allows to generate automatically correct statements about the substitutions, which can occur during program execution, by a simulation based on some approximation. Abstract interpretation thus replaces the standard semantics of the program to be analyzed with "collecting" semantics. by replacing concrete data values with abstract descriptions. the semantics are used to "collect" information about program states reachable during the execution of the program.

Formally, the abstract interpretation is explained as follows.

If E is the powerset of a set of data object(the concrete domain) in a program P, and D is a partiality ordered set of descriptions(the abstract domain). An abstract interpretation is formally defined by four functions: Ep: $E \rightarrow E$, Dp: $D \rightarrow D$, α: $E \rightarrow D$, γ: $D \rightarrow E$ (<Fig 1>). These functions must satisfy the conditions listed in <Table 1>. In order to establish an abstract interpretation framework, three basic components are thus required:



<Fig 1> Abstract Interpretation Functions

- An abstract domain.
- A specific collection of operations defined over the values of the abstract domain.
- An abstract interpreter which implements the analysis algorithm of a program and operates on values in an abstract domain.

There have been some attempts to formalize the abstract interpretation of logic program.(Bruynooghe,1987, Jacobs,1991, Jacobs,1989, Mellish,1986) They adopt different kinds of frameworks for the abstract interpretation of logic programs. They mostly use denotational semantics, while some use the schemes based on the AND/OR tree. For these different frameworks, the major differences are (i) the kinds of primitive operations they introduce and (ii) the method to deal with the recursion. For the kinds of primitive operations, Jacobs uses two(Jacobs,1991) and Bruynooghe uses six primitive operations.(Bruynooghe,1987) Generally, the finer operations an abstract framework employs, the more easily the operations can be implemented. On the other hand, if an abstract framework uses fewer primitive operations, the implementation and proof of the primitive operations become more complicated and difficult. The problem of handling recursion is how to compute the least fixpoint and several ways are adopted such as bottom-up and top-down methods.(Abramsdy,1987)

III. Proposed Abstract Domain

In order to identify the information on variable sharing and the propagation of groundness with great efficiency and accuracy, we develop an abstract domain to be used for the abstrract interpretation. This section presents the abstract domain.

1. AND-parallelism and its detection

For the detection of AND-parallelism, three kinds of information are essential as explained below:

- *Static sharing information*

  Static sharing indicates that some goals in a clause have textually the same variables in their arguments. The goals which have at least one statically shared variables cannot be executed in parallel within the context of independent AND-parallelism. For example, consider two goals a(X,Y,Z) and b(X) which appear in a clause. Because these goals share variable X, they cannot be executed in parallel. The static sharing information can be identified in a very straightforward way, e.g., by scanning the variables and comparing textually their names.

- *Aliasing information*

  Some goals which do not have statically shared variables may not be executed in AND-parallel. At runtime, variables with different textual names can be with each other as the result of reference bindings at unification. The set of variables associated with reference bindings forms an alias. The information on the variable aliasing is very crucial for the detection of proper AND-parallelism. If a set of variables are aliased, the goals which have at least one variables among them in their arguments cannot be executed in parallel. Generally, the static data-dependency analysis or the abstract interpretation techniques can be used for finding the aliasing information.

- *Propagation of Groundness*

  Although the shared variables and the variable aliasing limit the amount of the AND-parallelism to be exploited, it is noted that once the shared variables or aliased variables are grounded, they do not limit AND-parallelism anymore. Consider the following example.

  ```
  ?- a(X,Y,Z), b(X), c(Z), d(Y,Z).
  a(U,V,U).
  b(2).
  c(2).
  ```

d(1,W).

Due to the statically shared variable Z, it seems that c(Z) and d(Y,Z) cannot be executed in parallel. But, in real execution of the program, we know that variable Z is grounded with constant 2 before the execution of goal c(Z) and d(Y,Z) since variable Z is aliased with X which is bounded with 2. It means that because the two goals do not have any shared variables, they can be executed in AND-parallel.

2. The proposed abstract domain

Two variables are *strongly coupled* at a given point in the program execution if and only if they always couple at the point regardless of the path of execution taken to reach the point and gounding one of the variables always results in the gounding of the other. The *coupled variables* which are among the coupled variables, the variables that not strongly coupled are referred to as *weakly coupled.*

The proposed abstract domain is based on the concept of sharing group.(Jacobs,1989) Let *svar(x)* be the set of strongly coupled variables for a variable *x*. For a given substitution θ, variables *u* and *v* ($u,v \in dom(\theta)$) are denoted to share variable *w* if $w \in svar(u) \cap svar(v)$.

- Sharing group of variable w for substitution θ, denoted as sg(θ,w), is the set of variables that share w. Let Subst and Var be respectively the set of substitutions and variables and P(x) be the power set of component x. Then, sg is a mapping as below:

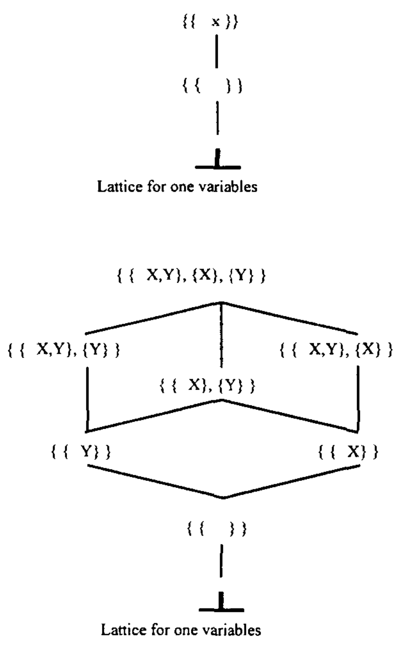
$$sg : Subst \times Var \rightarrow P(Var)$$

- Sharing S is defined to be a set of sharing groups, i.e., S = P(P(Var)). Intuitively, substitution θ is approximated by S if S contains the sharing group of each variable x ($x \in rgv(\theta)$). That is,

$$\gamma(S)=\{[s] \mid \forall v \in rgv(s,v): sg(s,v) \in S\}$$

From sharing *S*, we can derive independency relation and ground propagation information for the detection of AND-parallelism. as follows

- Groundness : if variable x does not belong to any set of S, x is a ground term. That is, variable x is grounded if there is no variable w shared by x.
- *Independence* : if *x* and *y* are not appear together in any set of *S*, variable *x* and *y* are independent.
- *Coupling* : Variable *x* and *y* are coupled if they belong to at least one of any sets in *S*.
- Ground Propagation : Grounding an element in a sharing group will ground the other elements in the ground if and only if the elements are not appear in any other sharing group.



<Fig 2> Lattice for the proposed domain

<Fig 2> shows the lattices for one and two variables in which a variable can appear in different sharing groups. The lattice does not directly show the information about groundness and independence because the variables are not exposed in the lattice. However, we can use the rules derived in the previous sections to gather the information.

III. Analysis of the Performance

In order to completely evaluate the performance, it is necessary to implement the proposed domain in a compiler which supports some abstract interpretation framework. Without a full-blown compiler, we demonstrate the feasibility and the performance of the proposed domain just by manual application of the proposed domain to an example sample program. The example program is shown below.

```
?-a(X,Y,Z,U),b(U),c(Y),d(Z),
  e(X),f(X).
a(A,A,B,g(A,B)).
b(g(1,2)).
c(1).
d(2).
e(1).
f(1).
```

According to the definitions, it is straightforward that the sharing group for variable *A* and *B* in the program is defined as follows.

sg(θ ,X) = {X,Y,U}

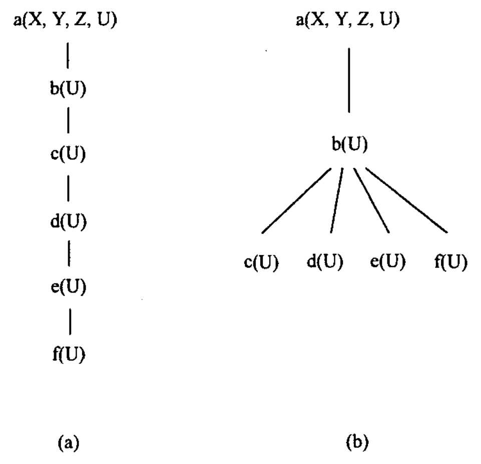
sg(θ ,Y) = {Z,U}

From these sharing groups, sharing S is defined.

S = {{X,Y,U},{Z,U}}.

According to the information derivation rules discussed earlier, we can figure out *Y* and *Z* are independent because they do not appear together in any set of sharing *S*.

The groundness propagation rule asserts that grounding *X* or *Y* will ground *Y* or *X*, grounding *U* will ground *X*, *Y* and *Z*, and grounding *Z* will strongly couple *X*, *Y*, and *U*. As we know ground *X* and *U* will ground *X*, *Y* and *Z*. after the match of *b(U)* with *b(G(1,2))*, subgoals *c(Y)*, *d(Z)*, *e(X)* and *f(X)* can be executed in parallel.



<Fig 3> Data-dependency graph

<Fig 3> shows two data dependence graphs. One (<Fig 3(a)>) is the graph resulting from the analysis based on the SDDA (Chang,1985), the other (<Fig 3(b)>) is the graph resulting from the above analysis based on our proposed domain. As shown from the <Fig 3(a)>, the SDDA based analysis fails to detect any possible AND-parallelism. On the other hand, our proposed domain provides a data-dependency graph which fully detects all the possible AND-parallelism in the example program.

Ⅳ. Conclusion

In this paper, we proposed an abstract domain which can provide a precise and efficient way to use abstract interpretation for the detection of AND-parallelism of logic programs. Applied to an example program for which the static data-dependency analysis (using SDDA) fails to extract any AND-parallelism, the proposed domain successfully extracts all the AND-parallelism available in the program. Even though we need more complete evaluation using a wide range of benchmarks, the analysis result obtained for a sample program reflects the potential of our proposed domain for the detection of AND-parallelism. As a future research, it is needed define some primitive operations which will incorporate the proposed domain into an abstract interpretation framework. Furthermore, it is required to implement them into a compiler to build an efficient AND-parallel logic system.

References

[1] S. Abramsdy and C. Hankin, editors. Abstract Interpretations of Declarative Languages. *Horwood*, 1987

[2] M. Bruynooghe. A Framework for the Abstract Interpretation of Logic Programs. *Technical Report CM 62,* CS Dept. K.U. Leuven, 1987.

[3] J. Chang. High Performance Execution of Prolog Programs Based on a Static Data Dependency Analysis. *PhD thesis*, University of California, 1985.

[4] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction of Approximation of Fixpoint. In *Proceedings of the 4th symposium on Principles of Programming Languages,* 1977.

[5] S. Debray and D. Warren. Automatic mode inferencing for Prolog Programs. In *Proceedings of the 3th Symposium on Logic programming*, 1986.

[6] T. Getzinger. Abstract Interpretation for the Compile-Time Analysis of Logic Programs. *PhD thesis*, University of Southern California, Sep. 1993.

[7] D. Jacobs. A Framework for the Abstract Interpretation of Logic Programs. *Technical report,* CS dept. University of Southern California, Sep. 1991.

[8] D. Jacobs and A. Langen. Accurate and Efficient Approximation of Variable Aliasing in Logic Programs. In *Proceedings of North American Conference on Logic Programming*, 1989.

[9] C. Mellish Some global optimization for a prolog compiler. *Journal of Logic Programming*, 2, 1985

[10] C. Mellish. Abstract interpretation of Prolog programs. In *Proceedings of the 3th International Conference on Logic Programming*, 1986.

[11] H. Xia and W. Giloi. A new application of abstract interpretation in Prolog programs: Data-dependency analysis. In *Proceedings of IFIP WG 10.0 Workshop on Concepts and Characteristics of Declarative Systems*, 1988.