ParTy — A Parallel-Types Analyser

Luís Fernando Pias de Castro
lfcastro@inf.ufrgs.br
Fone: +55 (053) 981-1831

Cláudio Fernando Resin Geyer
geyer@inf.ufrgs.br
Fone: +55 (051) 316-6802

Instituto de Informática - UFRGS
Caixa Postal 15064
Porto Alegre – RS

Keywords: Type inference, Abstract interpretation, Prolog, Parallelism

This work presents a type inference system aimed at the parallel exploitation of logic programs. It emphasizes the determination of costs related to the exportation of goals on distributed memory parallel machines, so types are based on the actual storage model of terms in Warren's Abstract Machine. The proposed type domain introduces a novel notion of type interval, which provides a greater accuracy on type analysis.

1. Introduction

Prolog [1] [2] is a declarative programming language based on first-order logic. It provides the programmer ways to express problems in a high level of abstraction. One important characteristic of this language is its clear distinction between the logic description of a problem and the control needed to solve it.

Parallelism has been widely used as a means of improving the performance of programs. Under imperative programming paradigms a programmer has to be aware of the issues related to parallelism, since he must code the control of the execution of the program. Since Prolog is a declarative language, a programmer doesn't have to deal with the control of the execution. This implies that Prolog is a well suited language for automatic exploitation of parallelism.

There are several parallel implementations of Prolog. However, most of them are based on shared memory machines. Opera [3] [4] is a project aimed at building a parallel Prolog system based on distributed memory machines.

The inclusion of parallelism in the control of a languages such as Prolog adds some overhead due to the issues related to managing the distribution of work among several processors. This overhead is even greater in the case of distributed memory machines where processors don't share memory and, thus, any communication between processors must be done via messages.

There are cases in which the parallel execution of a task may be even slower than the usual sequential execution. That's because the cost of transferring the arguments for the task may be greater than the execution cost of the task.

This work describes a type inference system based on abstract interpretation aimed at providing information to the runtime system to determine the cost related to sending a task to some other processor. This may be used by the runtime system when deciding about the parallel or sequential execution of a task.

The type system described reflects the actual storage of information in Prolog systems based on Warren's Abstract Machine. It introduces a novel notion of type intervals. A type interval denotes the situation in which a variable may be linked to different terms at runtime. It mantains the lesser and the greater types the variable may be linked to, according to a given order relation.

The rest of this paper is divided into four sections. Next section describes the basic concepts used througout the remaining text.

Section 3 defines the static analysis model, along with the parallel type domain proposed. Section 4 provides some description about possible uses of the parallel-type system. The final section describes the conclusions of the work.

2. Basic Concepts

In this section we are going to describe the basic concepts needed to understand the proposed type inference system, that will be used throughout the remaining of the text.

2.1. Abstract Intepretation

Abstract interpretation is a static analysis technique developed by Cousot and Cousot [5]. It's based on the idea of obtaining an approximate semantics of a program, by abstracting the domain over which this program works. In order to do this, first we have to define an abstract domain which reflects the kind of information we want to infer. Also, it's necessary to create equivalent abstract functions for every primitive function of the language in which the program has been written.

An analysis, according to abstract interpretation, involves a fixpoint calculus in order to obtain the abstract semantics of a given program.

Creating a static analyser, based on abstract interpretation, involves three steps:

(i) defining an abstract domain based on the information we want to infer;

(ii) creating the corresponding abstract functions for the primitive operations of the language and

(iii) defining an execution model for the fixpoint calculus.

2.2. Logic Programming

The logic programming paradigm provides the user a way to express problems at a high level of abstraction. Prolog is the most well-known logic programming language used nowadays, and it's based on two independent works of Colmerauer [1] and Kowalski [2].

Prolog's main characteristic is the clear distinction between logic and control. It means that, when a programmer wants to describe a problem, he only has to be concerned about its logic. The underlying system will be responsible for providing a control system to solve the problem.

On one hand, this characteristic makes program development easier, since the programmer doesn't have to specify the control of the program. On the other hand, it allows the underlying runtime system to implement different control systems.

Most of Prolog implementations are based on the Warren Abstract Machine (WAM) [6]. WAM is a stack-oriented abstract machine specially developed for the execution of Prolog programs.

2.3. Parallel Processing

Parallel processing is, nowadays, a widely accepted technique to provide high performance for several kinds of programs. But the introduction of control for parallel processing in existing programs written in imperatives languages is considerably hard. This has led to research on applying parallel processing to other programming paradigms.

Prolog is very well suited for parallel processing, since the user has (at least in theory) no control on how a problem is going to be solved. There are two main kinds of parallelism in Prolog:

- **Or-parallelism**, when two or more clauses of a given predicate are executed in parallel and
- **And-parallelism**, when several goals of a given clause are executed at the same time.

And-parallelism can, still, be split in two categories:

- **Independent And-parallelism**, which exploits the parallel execution of goals which don't share variables and
- **Dependent And-parallelism**, where goals are executed in parallel if they share one or more variables.

There are several implementations of parallel Prolog systems. For example, **Muse** [7] or-parallel system, the independent and-parallelism **&-Prolog system** [8], and **Andorra-I** [9], which exploits both or-parallelism and dependent and-parallelism. However, most of the existing systems are based on shared-memory machines.

2.4. Opera Independent And-Parallelism

Opera is a parallel Prolog system, which exploits or-parallelism and independent and-parallelism. Opera is aimed at exploiting parallelism on distributed-memory machines.

Opera's or-parallelism is based on a multi-sequential model [3]. When a processor asks for work it receives a clause, which must be completely executed by this processor. Each processor executes a WAM-based abstract machine called TWAM.

The independent and-parallelism model used in Opera is called restricted and-parallelism. According to this model, conditional graph expressions are used to denote situations when parallel execution of goals may be exploited. The WAM-based abstract machine used for and-parallelism is the RAP-WAM [10], which has been adapted to work on distributed-memory machines [11].

3. Type Inference for Distributed-Memory Parallel Machines

3.1. Static Analysis Model

The static analysis model presented here includes the execution model for the fixpoint calculus and the abstraction of the primitive operations of Prolog. Also, it's parameterized by domain-dependent functions in such a way that makes it a domain-independent static analysis model.

This model is based on a three levels algorithm. In the first level, a function called compute_fixpoint coordinates the several iterations that have to be done over the program, in order to execute the fixpoint calculus. On a second level, functions exec_rec_pred and exec_nrec_pred are responsible for executing recursive and non-recursive predicates, respectively. These functions also manipulate a memory structure called *Memo Table*, which is used to hold the partial solutions obtained on each iteration. In a third level, function exec_clauses assures that every clause of a predicate will be analysed when analysing that predicate.

In order to instantiate this model for a given abstract domain, there must be defined four domain-dependent functions: calc_in_subst, calc_succ_subst, calc_call_subst and calc_ret_subst.

calc_in_subst is called at the start of the analysis of a clause. Its in-arguments are the variables of the call, a substitution over these variables, and the arguments of the clause. It performs the unification between the variables of the call and the variables of the clause, and returns the clause entrance substitution.

Before returning from the analysis of a clause, function calc_succ_subst is called. It takes as its arguments the arguments for the clause, the arguments for the call, and the exit substitution. It performs the unification between the clause- and the call-arguments, and returns the success substitution over the arguments of the call.

calc_call_subst is called before each goal of a clause is analysed. It takes as arguments the current substitution and the arguments for the next goal. It returns the current substitution restricted to the variables that appear in the arguments for the next goal.

The function calc_ret_subst is called after a goal has been analysed and it takes the substitution prior to the analysis of the goal and the success substitution for the goal as arguments. It, then, applies the changes described in the success substitution for the goal to the previous substitution, returning a new current substitution.

3.2. The Parallel Type Domain

The abstract domain shown here has been specially designed to infer types for Prolog. Its goal is to provide information for the runtime system to be able to decide when the parallel execution of a goal will increase the overall execution time of a program.

In distributed memory machines, the parallel execution of a goal involves transferring the arguments for the goal from one processor to another, usually via messages. This transfer time has to be considered when deciding if a goal should be executed in another processor, since the cost of transferring the arguments from one processor to the other, and the results back to the first one, may be even greater than the cost of executing the goal sequentially. Also, the cost of calculating the size of the messages may decrease the performance of the system.

In order to try to decrease the decisions that have to be taken at runtime and to decrease the cost of calculating the size of the messages, the type system has been designed to reflect the actual storage of Prolog terms in the stacks of WAM-based abstract machines.

A new aspect of type systems for Prolog introduced in this work is the notion of intervals of types. A type interval is used to describe variables that assume different types in runtime. It holds the smaller and the bigger types a variable can assume, according to an appropriate ordering function. The use of intervals provide for a greater accuracy of the analysis.

The Parallel-Types domain is composed by the following types:

- **$bottom$** — represents the absence of information about the type of a variable;
- **int** — represents integer numbers;
- **float** — describes floating pointing numbers;
- **atom(Size)** — represents an atom, which is composed by *Size* characters;
- **struct(Size,Arity,[Type_List])** — describes a structure, which has a functor (name) composed by *Size* characters, has arity *Arity*, and each argument has a corresponding type, described in *[Type_List]*.

- **list(Size,Type)** — describes a list, which has *Size* elements of type *Type*;
- **\$top\$** — denotes variables for which the analyser hasn't been able to determine a type; and
- [**minor_type, major_type**] — denotes a type-interval, whose smaller type is *minor_type*, and bigger is *major_type*.

This domain has two characteristics that have to be noted. First, *struct* and *list* are two types that may be recursively defined. It means that one may say, for example, that a list has elements of the type *list*. In order to guarantee that the domain is finite, a depth restriction has to be defined. In fact, the depth restriction is a parameter of the analyser.

Second, the notion of type-interval presumes the existance of an order relation over the simple (punctual) types. This relation is given by the sizeof function which will be described below.

3.2.1. The Order Relation

The order relation defined for this domain over the simple (non-interval) types is based on the storage model of term on WAM-based Prolog abstract machines.

Definition 1 (Total Order Operator) *A given type T_1 precedes another type T_2 if and only if the storage size (given by the* sizeof *function) of T_1 is lesser or equal to the size of T_2.*

$$T_1 \sqsubseteq^{\mathcal{T}} T_2 \Leftrightarrow sizeof(T_1) \leq sizeof(T_2) \tag{1}$$

The size of a type is calculated according to the way terms represented by the given type are stored in the stacks of the WAM. The sizeof function returns the size of a type. It's defined as follows.

Definition 2 *Sizeof*

$$sizeof(bottom) = \mathsf{MAX\_SIZE} \tag{2}$$

$$sizeof(int) = 4 \tag{3}$$

$$sizeof(float) = 8 \tag{4}$$

$$sizeof(atom(size)) = size \tag{5}$$

$$sizeof(struct(...)) = 1 + size + arity + \sum_{i=1}^{arity} sizeof(type[i]) \tag{6}$$

$$sizeof(list(...)) = 1 + 3 \times size + size \times sizeof(type) \tag{7}$$

$$sizeof(top) = \mathsf{MAX\_SIZE} \tag{8}$$

Note that these formulas have been optimized for the Opera system. This way, some constants have been used, denoting that an integer is stored using 4 bytes and a floating point number requires 8 bytes to be stored. These values must be adapted according to the Prolog system desired.

The two most complex cases are those for the size of a list and for the size of a structure. The formulas are defined according to the storage of these types, as shown below.

Figure 1 shows the storage of two different structures. The storage of the structure $f(a, b)$ starts at cell 1 and the structure $f(c, d, e)$ is stored from cell 8. From that stack representation, we note that each argument of a structure needs one REF cell, besides the cells needed to store the argument

1	STR	2
2	f/2	-
3	REF	5
4	REF	6
5	CON	a
6	CON	b
7	...	.
8	STR	9
9	f/3	-
10	REF	13
11	REF	14
12	REF	15
13	CON	c
14	CON	d
15	CON	e
..	...	.

Figure 1: WAM storage of $f(a,b)$ and $f(c,d,e)$

itself. Also, it's necessary one STR cell to describe the type of term stored (a structure), plus the number of cells needed to store the name of the structure. Therefore, the storage requirement of a structure is given by equation 6.

Even though a list may be represented by means of binary structures, this storage model requires too much space. As lists are very common in Prolog, the WAM uses a more optimized way to represent them. Figure 2 shows the storage of two lists. The first one is a list with two elements, which starts at cell 1. The other list has three elements, and starts at cell 11. As we can see, each new element of a list requires one LIS cell and two REF cells, plus the cells required to represent the element itself. Besides, there must be always a *nil* cell, to terminate the list. Therefore, it's straightforward to come up with equation 7 to determine the size of a list.

3.2.2. Example

This section aims to provide an example of how the domain-specific functions work during an analysis.

Suppose the analysis of the second clause of the *append/3* predicate shown in figure 3. Also, suppose the query shown at the same figure is given with the following abstract substitution:

$$\phi_{call} = \{A \rightarrow list(5, int), B \rightarrow list(8, atom(10)), C \rightarrow \$bottom\$\}. \tag{9}$$

The analysis starts by calling calc_in_subst, in order to unify the variables of the call to the arguments of the head of the clause. The result of this unification is the substitution

$$\Phi_1 = \{H \rightarrow int, L \rightarrow list(4, int), L2 \rightarrow list(8, atom(10)), L3 \rightarrow \$bottom\$\}. \tag{10}$$

The second step of the analysis is to prepare a substitution to be used when calling the first (and only) goal of the clause. This is the job of the function calc_call_subst. It creates a new substitution with only the variables which appear in the next goal.

$$\Phi_{in1} = \{L \rightarrow list(4, int), L2 \rightarrow list(8, atom(10)), L3 \rightarrow \$bottom\$\}. \tag{11}$$

1	LIS	2
2	REF	7
3	REF	4
4	LIS	5
5	REF	8
6	REF	9
7	CON	a
8	CON	b
9	CON	nil
10	...	.
11	LIS	12
12	REF	20
13	REF	14
14	LIS	15
15	REF	21
16	REF	17
17	LIS	18
18	REF	22
19	REF	23
20	CON	c
21	CON	d
22	CON	e
23	CON	nil

Figure 2: WAM representation of $[a, b]$ and $[c, d, e]$

```
append([ ], L, L).
append([H|L], L1, L2) :- append(L, L1, L2).

?- append(A, B, C).
```

Figure 3: Predicate *append/3* and the query

Then, the goal is called, with the last given substitution. The result of executing a goal is another substitution. In this example, let's suppose we have already reached a fixpoint. So, the result of recursively calling *append/3*, in this case, should be the substitution

$$\Phi_{out1} = \{L \rightarrow list(4, int), L2 \rightarrow list(8, atom(10)), L3 \rightarrow list(12, interval(int, atom(10)))\}. \quad (12)$$

This substitution has to be joined to the substitution given in equation 10. This is done by the function calc_ret_subst. Again, the result of this action is given by a new substitution.

$$\Phi_2 = \{H \rightarrow int, L \rightarrow list(4, int), L2 \rightarrow list(8, atom(10)), L3 \rightarrow list(12, interval(int, atom(10)))\}. \quad (13)$$

As there are no more goals to be called, the last function to be executed is calc_succ_subst, which applies the last substitution to the variables of the call. It then gives the success substitution for this clause (according to the given query).

$$\phi_{succ} = \{A \rightarrow list(5, int), B \rightarrow list(8, atom(10)), C \rightarrow list(13, interval(int, atom(10)))\}. \quad (14)$$

4. Applications of Parallel Types

There are several applications for the type system described in this work. Here we describe two of the most important applications, that will be used in Opera.

4.1. The Unification Process

The most important primitive operation of Prolog is unification. At the same time, it's the most complex operation of Prolog. This is partially due to the fact that, as Prolog lacks type annotation, the unification algorithm must test for several cases, when unifying two terms.

Using the type inference system describe in this paper, it's possible for the compiler to generate specialized code for the unification process, according to the types of the terms that are being unified.

4.2. Determination of Costs

One of the main characteristics of and-parallelism is its small granularity [12]. It means that the cost of (sequential) execution of a goal is usually small.

On distributed memory machines, this may cause a great problem, since the cost for exporting a goal to other machine involves transferring WAM stacks via messages. So, the scheduler in such a system has to be prepared to decide whether or not the parallel execution of a goal will bring benefits to the overall execution time of the program.

The parallel-type inference system described may be used to provide information to the scheduler in order to make this decision easier.

Two naive implementations of schedulers may be provided by an optimistic and a pessimistic schedulers. The first one would aim to execute in parallel any goal which **may** provide speedup to the system. So, it would use the minor element of any interval provided by the analyser, when comparing the cost of exportation to the cost of execution of the goal.

On the other hand, a pessimistic scheduler would execute in parallel only those goals which will **surely** provide speedup to the system. This way, it would compare the cost of execution of the goal to the major element of any intervalar type provided by the analyser.

A more sophisticated scheduler should be based on experimentations in order to statistically find an optimal point between the two limits of intervalar types.

5. Conclusions

This work presents a type inference system designed to be used at the cost evaluation of the parallel execution of goals under distributed-memory machines.

The notion of type inference isn't new to logic programming systems. Azzoune [13] defines an inference system based on generating and solving inequations on type variables. Corsini and Musumbu [14] define a type analysis system based on abstract interpretation, allowing for the inference of partially instantiated terms. Codish and Lagoon [15] present an interesting directional-types inference system based on the use of program abstraction and ACI-unification.

The parallel type system presented in this work is unique in introducing a notion of interval of types. This notion provides for a more accurate analysis.

This work contributes to the Opera system in two ways. First, it allows the introduction of specialized unification algorithms, improving the parallel and sequential execution of programs.

Second, it provides information about the storage size of the arguments of a goal. This information can be used to calculate the cost of the parallel execution of a given goal, thus providing the scheduler a way to determine whether or not the execution of the goal in another processor will improve the performance of the program.

References

[1] COLMERAUER, A. et al. **Un Système de Communication Homme-Machine en Français**. GIA: Université d'Aix-Marseille II, 1972.

[2] KOWALSKI, R. **Logic for Problem Solving**. New York: Elseviér, 1979.

[3] BRIAT, J.; FAVRE, M.; GEYER, C. et al. **Opera**: a Parallel Prolog system and its Implementation on Supernode. Grenoble: Laboratoire de Genie Informatique de Grenoble/CAP-Gemini-Inovation, 1990. (Technical Report).

[4] GEYER, C.; YAMIN, A.; WERNER, O. Projeto Opera: Um Modelo E/OU para Prolog. In: CONGRESSO DA SOCIEDADE BRASILEIRA DE COMPUTAÇÃO, 12., 1992, Rio de Janeiro. **Anais...** Rio de Janeiro: SBC, 1992.

[5] COUSOT, P.; COUSOT, R. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In: ACM SYMP. ON PRINCIPLES OF PROGRAMMING LANGUAGES, 4., 1977. **Proceedings...** ACM Press, 1977.

[6] WARREN, D.H.D. **An Abstract Prolog Instruction Set**. Manchester:SRI International: AI Center: Computer Science and Technology Division, 1983. (Technical Note 309).

[7] ALI, K.; KARLSSON, R. **The Muse Or-Parallel Prolog Model and Its Performance**. Kista:Swedish Institute of Computer Science, 1990.

[8] HERMENEGILDO, M. &-Prolog and its Performance: Exploiting Independent And-Parallelism. In: INTERNATIONAL CONFERENCE ON LOGIC PROGRAMMING, 1990. **Proceedings...**, MIT Press, 1990.

[9] COSTA, V.S.; WARREN, D.H.D.; YANG, R. The Andorra-I Engine: A Parallel Implementation of the Basic Andorra Model. In: INTERNATIONAL CONFERENCE ON LOGIC PROGRAMMING, 8., 1991. **Proceedings...**, MIT Press, 1991.

[10] HERMENEGILDO, M. **An Abstract machine Based Execution Model for Computer Architecture Design and Efficient Implementation of Logic Programs in Parallel**. Austin: University of Texas at Austin, 1986. Ph.D. Thesis.

[11] WERNER, O. **Uma Máquina Abstrata Estendida para o Paralelismo E na Programação em Lógica**. Porto Alegre: CPGCC da UFRGS, 1994. Master Thesis.

[12] BARBOSA, J. **GRANLOG: Um Modelo para Análise Automática de Granulosidade na Programação em Lógica**. Porto Alegre: CPGCC da UFRGS, 1996. Master Thesis.

[13] AZZOUNE, H. Type Inference in Prolog. **9th. International Conference on Automated Deduction**, Argonne, USA, May 1988.

[14] CORSINI, M.-M. e MUSUMBU, K. Type Inference in Prolog: a New Approach. **Theoretical Computer Science**, Elseviér, v. 119, 1993.

[15] CODISH, M. e LAGOON, V. Type Dependencies for Logic Programs Using ACI-Unification **From: ftp://132.72.40.8/pub/people/codish/aci-type.ps**, Ben-Gurion University of the Negev, 1996.