Technical Communications of the International Conference on Logic Programming, 2010 (Edinburgh), pp. 222–225
http://www.floc-conference.org/ICLP-home.html

TOWARDS A PARALLEL VIRTUAL MACHINE FOR FUNCTIONAL LOGIC PROGRAMMING

ABDULLA ALQADDOUMI

New Mexico State University, Computer Science Department,
P.O. Box 30001, MSC CS, Las Cruces, NM 88003, USA
E-mail address: aalqaddo@cs.nmsu.edu

Abstract. Functional logic programming is a multi-paradigm programming that combines the best features of functional programming and logic programming. Functional programming provides mechanisms for demand-driven evaluation, higher order functions and polymorphic typing. Logic programming deals with non-determinism, partial information and constraints. Both programming paradigms fall under the umbrella of declarative programming. For the most part, the current implementations of functional logic languages belong to one of two categories: (1) Implementations that include the logic programming features in a functional language. (2) Implementations that extend logic languages with functional programming features. In this paper we describe the undergoing research efforts to build a parallel virtual machine that performs functional logic computations. The virtual machine will tackle several issues that other implementations do not tackle: (1) Sharing of sub-terms among different terms especially when such sub-terms are evaluated to more than one value (non-determinism). (2) Exploitation of all forms of parallelism present in computations. The evaluation strategy used to evaluate functional logic terms is needed narrowing, which is a complete and sound strategy.

1. Introduction

Functional logic programming is a multi-paradigm programming that combines the best features of functional programming and logic programming. Functional programming provides mechanisms for demand-driven evaluation, higher order functions and polymorphic typing. Logic programming deals with non-determinism, partial information and constraints. Both programming paradigms fall under the umbrella of declarative programming.

For the most part, the current implementations of functional logic languages belong to one of two categories: (1) Implementations that include the logic programming features in a functional language. (2) Implementations that extend logic languages with functional programming features. Interested readers are referred to [Han07] for a survey on such languages and implementations.

In this paper we describe the undergoing research efforts to build a parallel virtual machine that performs functional logic computations. The virtual machine will tackle several issues that other implementations do not tackle: (1) Sharing of sub-terms among different terms especially when such sub-terms are evaluated to more than one value (non--determinism). (2) Exploitation of all forms of parallelism present in computations. The

Key words and phrases: functional logic programming; term rewriting system; non-determinism; needed narrowing; and-parallelism; or-parallelism.

Technical Communications of the 26th International Conference on Logic Programming, Edinburgh, July, 2010
Editors: Manuel Hermenegildo, Torsten Schaub
LIPIcs - Leibniz International Proceedings in Informatics. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany
Digital Object Identifier: 10.4230/LIPIcs.ICLP.2010.222

evaluation strategy used to evaluate functional logic terms is needed narrowing [Ant00], which is a complete and sound strategy.

2. Progress and Research Objectives

2.1. Types of Parallelism

Functional logic computations can exploit and-parallelism, or-parallelism or both. And-parallelism arises when two or more sub-terms need computation in order to compute their ancestor term. Computing such sub-terms simultaneously will decrease the waiting time of their parent. And-parallelism is usually independent, but when the computation of such sub-terms simultaneously involves a shared variable, it becomes dependent. In case of dependencies, synchronization of terms is required. Or-parallelism arises when a choice operator "?" or a variable need computation. The choice operator, "?", and logic or extra variables represent non-determinisim in functional logic computations. Equation (1) below is an example of non-deterministic operation using the choice operator. In this case, "coin" can be replaced by either 0 or 1. The computation of a choice operator in parallel can be done by invoking as many processes as there are arguments and locally for each process, its argument will rewrite the choice with its selected argument. The same thing is true for logic or extra variables.

```
coin = 0 ? 1                (1)
```

2.2. Research Objectives

The goal of my research is to implement a parallel virtual machine that will execute functional logic programs efficiently, specifically for the functional logic language, Curry [Han06]. Curry is a functional logic language that seamlessly integrates the logic and functional features. The evaluation strategy used in the virtual machine will be based on needed narrowing. Needed narrowing is based on the concept that only needed arguments of a term must be selected and evaluated to rewrite the term. Needed arguments can be decided by using definitional trees. For more details about needed narrowing, please refer to [Ant00]. For more details about definitional trees, please refer to [Ant92]. Computing normal forms of terms may involve computation of several sub-terms at the same time (and-parallelism). This is called don't-know non-determinism, because the strategy does not know which sub-term to execute first. A sequential evaluation strategy will compute the sub-terms one after the other. When choices or variables are encountered during the computation of some term, the system can try all such alternatives that will replace the choice or logic variable in order to compute the term in hand (or-parallelism). This is called don't-care non-determinism, because the system will not care which alternative of the choice or variable was chosen to compute the term. Our virtual machine will exploit both (and-parallelism and or-parallelism).

2.3. Progress

The system is implemented for the time being in Ruby. The current implementation includes basic libraries and parsing of expressions with sharing. Ruby supports the use of threads. The evaluation strategy will select needed sub-terms after each rewriting step and different threads will compute those needed sub-terms simultaneously. The first stage of the implementation that is done in Ruby will be used as proof of correctness and the final version of the virtual machine will be developed in C or Java to improve its efficiency.

3. Future Work

Computation of choice operations or logic variables as mentioned earlier leads to non-determinism and having more than one term to rewrite the current choice or variable. Such terms could possibly be shared between more than one parent-term. Therefore, the information about the option that rewrites the term must be global in the whole graph. This will ensure all solutions obtained are correct. We will present several solutions that can ensure the correctness of the solutions obtained from terms where choices or variables are shared within them. All the solutions are based on needed narrowing as a strategy for instantiating variables.

```
double x = x + x          (2)
double coin               (3)
```

In equation (2) the operation "double" will be rewritten as a plus operation. The argument of "double" must be represented as one shared argument between the first and second arguments of plus. In (1) we defined the non-deterministic operation coin that can be rewritten as 0 or 1. In the evaluation of (3), the correct computation must yield the values 0 or 2. If there was no implementation of sharing, equation (3) would be rewritten as "coin + coin". Such term would evaluate to any of the four values, 0, 1, 1, or 2.

3.1. Stack Copying

This evaluation strategy will perform computations of deterministic terms normally. When a non-deterministic term is encountered (choice or variable), separate contexts of the graph will be managed by different processes. There will be as many environments as the number of the arguments of the choice or the variable. This solution is inspired from strategies of implementing or-trees in logic programming [Gup01]. This approach will be very expensive in terms of memory but optimizations can be done. The other main drawback is the re-computation of terms. In the evaluation of (3), two separate environments will be created to accommodate each argument of the operation "coin": "double 0" and "double 1". Note that in case non-deterministic terms appear on both sides of the tree, re-computations of such non-deterministic terms may not be avoidable.

3.2. Fingerprinting

This evaluation strategy is built on replacing choices or variables with sets. These sets will contain all the arguments of the choice or values of the variable's domain. Each element of the set will be tagged with a unique fingerprint that will identify the origin of the element. The fingerprint consists of two parts, parent and position tags. Whenever a set contributes in other computations, compatibility of fingerprints will be checked to avoid

inconsistency. Two terms are said to be incompatible if they have the same parent tag but different position tag. The drawback of such approach is the overhead of bookkeeping of fingerprints and the compatibility tests performed.

In the evaluation of (3), the following set will be created to replace the non–deterministic operation "coin": $\{0^{A1}, 1^{A2}\}$. The first element of the set, "0", will be tagged with the fingerprint A1, and the second element of the set, "1", will be tagged with the fingerprint A2. The "plus" operation will perform the compatibility check between the two sets: $\{0^{A1}, 1^{A2}\}$ and $\{0^{A1}, 1^{A2}\}$. The resulting set will become $\{0^{A1} + 0^{A1}$, $\cancel{0^{A1} + 1^{A2}}$, $\cancel{1^{A2} + 0^{A1}}$, $1^{A2} + 1^{A2}\}$ ==> $\{0^{A1} + 0^{A1}, 1^{A2} + 1^{A2}\}$ ==> $\{0^{A1}, 2^{A2}\}$. The two stroked options refer to incompatibility between the elements and hence removed. Note that the compatibility test is performed before the operation is applied to avoid any unnecessary computations.

3.3. Promotion of Choice or Variable (Bubbling)

This evaluation strategy will promote non-deterministic operations. When a choice or a variable is encountered, the choice or variable will be promoted to take the place of its parent(s) and make as many copies of the parent(s) as the number of the arguments of the choice or variable. In case a choice has more than one parent, a nearest common ancestor between such parents will be computed and the spine that connects both parents with the common ancestor will be copied. All copies of the spine will be identical in their arguments except for the argument that had the choice which will be replaced by one of the choice arguments. Eventually the choice will reach the root of the graph. A rewriting of (3) in this case will be "double coin ==> (double 0) ? (double 1) ==> (0 + 0) ? (1 + 1) ==> 0 ? 2". Use of Promotion is inspired from [Ant07, Jan94, Lop97].

References

[Ant92] Sergio Antoy. Definitional trees. In *In Proc. of the 3rd International Conference on Algebraic and Logic Programming*, pp. 143–157. Springer LNCS, 1992.

[Ant00] Sergio Antoy, Rachid Echahed, and Michael Hanus. A needed narrowing strategy. *J. ACM*, 47(4):776–822, 2000.

[Ant07] Sergio Antoy, Daniel W. Brown, and Su-Hui Chiang. Lazy context cloning for non-deterministic graph rewriting. *Electron. Notes Theor. Comput. Sci.*, 176(1):3–23, 2007.

[Gup01] Gopal Gupta, Enrico Pontelli, Khayri A.M. Ali, Mats Carlsson, and Manuel V. Hermenegildo. Parallel execution of prolog programs: a survey. *ACM Trans. Program. Lang. Syst.*, 23(4):472–602, 2001.

[Han06] M. Hanus (ed.). *Curry: An Integrated Functional Logic Language (Vers. 0.8.2)*. 2006. Available at `http://www.informatik.uni-kiel.de/~curry`.

[Han07] M. Hanus. Multi-paradigm declarative languages. In *Proceedings of the International Conference on Logic Programming (ICLP 2007)*, pp. 45–75. Springer LNCS 4670, 2007.

[Jan94] Sverker Janson. *AKL - A Multiparadigm Programming Language*. Ph.D. thesis, Uppsala University, SICS, 1994.

[Lop97] Ricardo Lopes and Vitor Santos Costa. The BEAM: Towards a first EAM implementation, 1997.