Modeling Backtracking, Disjunctive Constraints, and Control/Data Or-Parallelism Extended Abstract

Donald A. Smith
Department of Computer Science
University of Waikato
Hamilton, New Zealand
dsmith@cs.waikato.ac.nz
Phone: (64)(7) 838-4503, Fax: (64)(7) 838-4155

September 13, 1994

Abstract

We present a series of concise Scheme interpreters of logic programming languages. Our main aim is to illuminate the roles of backtracking, disjunctive constraints, or-parallelism, and lazy evaluation as control structures for managing disjunction in logic programming. A further aim is to demonstrate, by means of an elegant and concise formalism, the simplicity of multi-SLD resolution as a generalization of SLD resolution.

Multi-SLD resolution is a variant of SLD resolution based on a simple idea: *Let the allowed constraints be closed under disjunction, and provide a mechanism for collecting the solutions to a goal and turning these solutions into a disjunctive constraint.* This idea leads to a novel execution model for logic programming, called *data or-parallelism*, in which multiple constraint environments partially replace backtracking as the operational embodiment of disjunction. The model has a natural implementation on data-parallel computers since each disjunct of a disjunctive constraint can be handled by a single (virtual) processor.

Starting from a basic SLD interpreter, small, localized changes express significant variations. The most important of these is the use of *disjunctive*

constraints as a replacement for standard (control) backtracking. Another variation is *constraint backtracking*, whereby lazy evaluation is used during the computation of the constraint component of the abstract machine state. *Two-dimensional backtracking* is the combination of control and constraint backtracking. We describe a classification of logic programming interpreters according to whether and how they implement backtracking and disjunctive constraints. Control or-parallelism results from eager evaluation of non-deterministic choice. Standard control backtracking results from lazy evaluation of non-deterministic choice, using a failure continuation. Data or-parallelism results from eager evaluation of disjunctive constraints. Constraint backtracking arises from lazy evaluation of disjunctive constraints.

Keywords

(Multi-)SLD resolution, (disjunctive) constraints, parallel logic programming, control or-parallelism, data or-parallelism, Scheme, substitutions, continuations, lazy evaluation, backtracking.

1 Introduction

In this paper we present a series of logic programming interpreters that illustrate alternative implementations of disjunction. The implementation of disjunction in top-down logic programming languages like Prolog typically relies on backtracking and depth-first search, or on the provision of multiple threads of control (control or-parallelism[1]). However, both backtracking and control or-parallelism have disadvantages: backtracking returns answers one at a time, often causing similar work to be repeated when a choice turns out to be the wrong one; and control or-parallelism (an approximation to breadth-first search) is expensive in implementation complexity.

An alternative implementation strategy called multi-SLD resolution has been described in several publications [5],[8]. The essential idea is to extend the SLD inference rule to permit multiple substitution environments and to provide a mechanism for collecting the solutions to a goal and turning these solutions into a set of substitutions (a disjunctive constraint). The canonical example that illustrates multi-SLD resolution and the resulting *data or-parallelism* is the query[2]

[1] The threads of control can be managed by multiple processors or even by one processor.

[2] In general, multiple variables can get bound by a `disj` goal.

```
| ?- generate(X),test(X).
```

To solve this query, standard Prolog enumerates the solutions to `generate/1` one by one via backtracking and tests each solution separately with `test/1`. A control or-parallel implementation (such as Aurora [3] or Muse [2]) starts up multiple Prolog search engines to explore branches of the SLD tree in parallel. In contrast, if we prefix the goal `generate(X)` with the operator `disj`, then an implementation based on multi-SLD resolution enumerates the solutions to `generate/1` subset by subset and creates from each subset a set of binding environments which are tested en mass (in parallel) by `test/1`. As a result, `test/1` is executed once per subset rather than once per solution and fewer instructions are executed overall. In addition, for many programs, the same or similar computation is performed for each invocation of `test` (e.g., the creation of a list), and using the engine/multi distinction (Section **??**) this shared computation can be 'factored out' and performed only once.

Previous papers [5] [4] informally introduced MultiLog and multi-SLD resolution; described a machine architecture (the Multi-WAM) for executing MultiLog programs; and presented benchmark results for sequential and parallel implementations of the language. Even on a uniprocessor computer, multi-SLD was shown to be as fast as or faster than SLD for many combinatorial search problems. [7] presents a model that explains the observed speedups. [8] formalizes multi-SLD resolution, examines some of its properties, and proves its soundness and completeness. [6] presents an analysis of environment representation schemes for MultiLog. The author's dissertation [9] discusses all of these issues in more detail.

Here our aim is to clarify the operational semantics of multi-SLD resolution and to expose the relations amongst control backtracking, control or-parallelism, disjunctive constraints, and streams as alternative and complimentary methods for expressing disjunction in top-down logic programming languages.

The plan of the paper is as follows. Section **??** informally describes multi-SLD resolution and summarizes some of its properties. In Section **??** we present a series of working MultiLog interpreters written in typed, almost[3] pure, Scheme [1]. The basic multi-SLD interpreter of Section **??**

[3]The one non-functional exception is the use of a global counter to implement clause renaming. Functionality can be restored by passing around a counter, as we did in Haskell interpreters of MultiLog that have been type checked and run using Mark Johnson's Gofer system. The use of a global counter leads, we think, to clearer code. Also, the use of

collects all solutions to a `disj` goal; its code differs only slightly from the code for an SLD interpreter in Section ??. Section ?? exploits parametric polymorphism in the interpreter of Section ?? to implement dynamic reversion to backtracking: the collection of subsets of solutions to `disj` goals; the code for this interpreter differs from the code for the previous interpreter in only one subroutine. Section ?? uses lazy evaluation (streams) in the representation of environments, so that the disjunction of substitutions comprising the constraint component of the abstract machine state is processed incrementally, on demand; again, the change needed to implement this variation is small and localized. Section ?? uses lazy evaluation to eliminate control backtracking altogether. Section ?? sketches an implementation of top-down dereferencing, an optimization whose logical justification is sketched in Section ??. Section ?? categorizes the interpreters of Section ?? according to their use of backtracking, disjunctive constraints, and lazy evaluation. It then interprets control backtracking as arising from lazy evaluation of breadth-first search, just as constraint backtracking arises from lazy evaluation of disjunctive constraints. Section ?? covers related work. Section ?? concludes.

References

[1] H. Abelson and G. Sussman. *Structure and Interpretation of Computer Programs*. MIT, 1985.

[2] K. Ali and R. Karlsson. The Muse Or-Parallel Prolog model and its performance. In *NACLP*, pages 757–776, 1990.

[3] E. Lusk et. al. The Aurora or-parallel prolog system. *New Generation Computing*, 7:243–271, 1990.

[4] D. A. Smith. Multilog: Data Or-Parallel Logic Programming. In *JICSLP '92 Workshop on Parallel Implementations of Logic Programming Systems*, 1992.

[5] D. A. Smith. MultiLog: Data Or-Parallel Logic Programming. In David S. Warren, editor, *Proceedings of the Tenth International Conference on Logic Programming*, Budapest, Hungary, 1993. MIT Press.

Scheme instead of Haskell allows us to more perspicuously model lazy evaluation, which is the default mode of evaluation in Haskell.

[6] D. A. Smith. Analysis of Environment Representation Schemes for MultiLog. Technical report, University of Waikato, 1994.

[7] D. A. Smith. Why Multi-SLD beats SLD (even on a uniprocessor). In *Programming Language Implementation and Logic Programming.* Springer-Verlag, 1994.

[8] D. A. Smith and Timothy Hickey. Multi-SLD Resolution. In *Logic Programming and Automated Reasoning.* Springer-Verlag, 1994.

[9] D.A. Smith. *MultiLog: Data Or-Parallel Logic Programming.* PhD thesis, Brandeis University, 1993.