5. Jonas Barklund and Haakan Millroth. Providing iteration and concurrency in logic programs through bounded quantifications. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 817–824, ICOT, Japan, 1992. Association for Computing Machinery.
6. C. Beeri, S. Naqvi, R. Ramakirishnan, O. Shmueli, and S. Tsur. Sets and negation in a logic database langauge (LDL1). In *PODS*. ACM Press, 1987.
7. P. Brisset and O. Ridoux. Continuations in λprolog. In *ICLP93*. MIT Press, 1993.
8. D. Chan. Constructive negation based on the completed database. In *ICLP88*. MIT Press, 1988.
9. D. Chan. An extension of constructive negation and its application in coroutining. In *NACLP89*. MIT Press, 1989.
10. M. Dincbas, P. Van Hentenryck, H. Simonis, and A. Aggoun. The constraint logic programming language CHIP. In *2nd FGCS*, 1988.
11. E. Lusk et. al. The Aurora or-parallel prolog system. *New Generation Computing*, 7:243–271, 1990.
12. M. Feeley. *An Efficient and General Implementation of Futures on Large Scale Shared-Memory Multiprocessors*. PhD thesis, Brandeis University, 1993.
13. M. Feeley and J. S. Miller. A parallel virtual machine for efficient scheme compilation. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming and Computer Architecture*, 1990.
14. C. Haynes. Logic continuations. In *ICLP*, pages 671–685, 1986.
15. P. Van Hentenryck and Y. Deville. The cardinality operator: A new logical connective for constraint logic programming. In *ICLP91*. MIT Press, 1991.
16. T. Hickey. unpublished manuscript. 1991.
17. Péter Kacsuk. *Execution Models of Prolog for Parallel Computers*. MIT, 1990.
18. G. Kuper. Logic programming with sets. In *PODS*. ACM Press, 1987.
19. H. Milroth. *Reforming Compilation of Logic Programs*. PhD thesis, Uppsala University, 1990.
20. Vijay A. Saraswat. *Concurrent Constraint Programming*. MIT, 1990.
21. T. Sato and F. Motoyoshi. A complete top-down interpreter for first order programs. In *ILPS91*. MIT Press, 1991.
22. D. A. Smith. Multilog: Data or-parallel logic programming. In *JICSLP '92 Workshop on Parallel Implementations of Logic Programming Systems*, 1992.
23. D. A. Smith. Multilog: Data or-parallel logic programming. In *ICLP93*. MIT Press, 1993.
24. D. A. Smith. Analysis of environment representation schemes for Multilog. Technical report, Waikato University, (Submitted), 1994.
25. D. A. Smith. Why multi-SLD beats SLD even on a uniprocessor. Technical report, University of Waikato, (submitted for publication) 1994.
26. D. A. Smith and Timothy Hickey. Multi-SLD resolution. In *Logic Programming and Automated Reasoning*. Springer-Verlag, 1994.
27. D.A. Smith. *MultiLog: Data Or-Parallel Logic Programming*. PhD thesis, Brandeis University, 1993.
28. P. Stuckey. Constructive negation for constraint logic programming. In *LICS*, 1991.
29. Bo-Ming Tong and Ho-Fung Leung. Concurrent constraint logic programming on massively parallel simd computers. In *International Logic Programming Symposium*, 1993.
30. P. VanHentenryck. *Constraint Satisfaction in Logic Programming*. MIT Press, 1989.

This article was processed using the LaTeX macro package with LLNCS style

goals from multiple recursion levels appear as conjuncts in a single clause body. In this way, the successive computations corresponding to different levels of recursion can be performed using and-parallelism.

[3] describes an extension to Prolog in which there is an explicit syntax for expressing various quantifiers or quantifier-like constructs, including `and`, `sum`, and `product`. The authors extend the operational semantics to allow computation of the quantified formula on a data parallel machine. The result is thus (in our terminology) a form of data and-parallelism. [5] considers bounded quantification where the quantifier is existential. In this case, the result is a form of data or-parallelism.

7 Conclusion

We presented a series of concise logic programming interpreters written in the programming language Scheme. The novel features of the interpreters are, first, the fact that the binding environment contains, in the simplest case, a disjunction (list) of substitutions, rather than a single substitution as in standard Prolog; second, the presence of code for collecting solutions to a goal and turning the solutions into a disjunctive constraint; third, the generalization to (n-) streams rather than lists for representing disjunctive constraints; and fourth, the implementation of the engine/multi distinction and top-down dereferencing in Section 4, based on the form of the environment tree described in Sections 2.1 and 2.2.

The interpreters demonstrate how disjunctive constraints are an alternative to standard (control) backtracking as a means of implementing disjunction in logic programming. Furthermore, the distinction between eager and lazy evaluation of disjunctive constraints leads to the notions of data or-parallelism and data backtracking, respectively, in much the same way that the distinction between eager and lazy evaluation of non-deterministic choice leads to the notions of control or-parallelism and control backtracking.

Acknowledgements

I thank Tim Hickey for helpful discussions, ideas, and encouragement. In particular, he helped devise the engine/multi distinction. I also thank Jacques Cohen for useful comments on my writing.

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. H. Arro, J. Barklund, and J. Bevemyr. Parallel bounded quantifiers: Preliminary results. In *Proceedings of the JICSLP'92 Post-Conference Joint Workshop on Distributed and Parallel Logic Programming Systems*, 1992.
4. J. Barklund. *Parallel Unification.* PhD thesis, Uppsala University, 1990.

solutions to some prior goals. Constraint backtracking results from sequential, lazy evaluation of the constraints, so that there is effectively only a single substitution.

Lazy evaluation limits concurrency. The concurrency can be either actual concurrency or concurrency simulated by a uniprocessor (using iteration over a list or array). Indeed, the interpreters make clear the possible sorts of parallelism in MultiLog. Control or-parallelism corresponds to the concurrent solving of `G1` and `G2` in the body of the code for `';'`. Data or-parallelism corresponds to concurrent unification in multiple substitutions of the `Substs` parameter, subsequent to the collection of solutions of a `disj` goal. Control and-parallelism corresponds to the concurrent solving of `G1` and `G2` in the body of the code for `','`.

Scheme is essentially a sequential programming language, and it cannot directly express concurrency (be it actual concurrency or just virtual concurrency on a uniprocessor). We distinguished between data backtracking and data or-parallelism indirectly, by using streams and lists, respectively, to represent disjunctive constraints. Scheme cannot express the distinction between control backtracking and control or-parallelism, since there is no way to represent multiple threads of control (say, by a data structure). However, it would be possible to express control or-parallelism if we wrote our interpreters in one of those dialects of Scheme with the `future` construct [12], [13]. (For example, in Figure 4 replacing the code for disjunctions with `(combine-streams (solve G1 Substs) (future (solve G2 Substs)))` would result in control or-parallelism.) The table above would then be twice the size, with entries `cop` for control or-parallelism. Note that an interpreter could have both control or-parallelism and data or-parallelism.

The reader may be wondering what *data and-parallelism* refers to. This label can be applied to the systems Reform [4] [19] and Parallel Bounded Quantifiers [3] [5], which we describe in the next section, along with other related work.

6 Related Work

There is little work directly related to that described here. See [23] for a contrast with previous work in parallel and constraint logic programming: [30], [10], [15], [18], [6], [17], [8], [9], [28], [21], [16]. Since the publication of [23] a more closely related work has emerged. Firebird [29] is a concurrent, committed-choice constraint logic programming language [20] whose execution model involves two components: an and-parallel inference engine (the front-end) and a massively parallel constraint solver (the back-end). "In a non-deterministic derivation step, if there is any unbound domain variable X in the system with domain $\{a_1, \ldots, a_n\}$, Firebird will create n or-parallel branches, each of which executes with an additional constraint $X = a_i$, $1 \leq i \leq n$." In principle these n partitions are independent computations, but in practice data parallelism is achieved by restricting computation so that "the same goal is evaluated in all partitions, but with different sets of arguments", represented by a vector. Since Firebird is a committed-choice language, backtracking is not available, and completeness would, it seems, be lost. However, the author of [29] reports (in private correspondence) that in more recent work he has extended the model.

S.-Å Tärnlund and his students have introduced the Reform model of logic programming [4] [19]. The basic idea is to unfold recursive program clauses so that

```
solve:: Goal -> Envs -> SuccessContinuation ->
    FailureContinuation -> Answer.
```

The **Envs** argument contains both an engine environment and an environment tree; bindings of engine variables are stored in the former, while bindings of multi-variables are stored in the latter. Unification of t_1 and t_2 is performed by traversing the terms and dereferencing engine variables; if the terms are not identical, if unification hasn't already failed, and if the terms contain multi-variables, then the environment tree is traversed top-down. If t_1 and t_2 become identical at some node, then unification succeeds in all descendant nodes. If t_1 and t_2 become non-unifiable at some node, then unification fails in all descendant nodes. Bindings are stored at the leaves.

5 Categorizing Prolog and MultiLog Interpreters

The interpreters in Section 3 illuminate the roles of control backtracking, disjunctive constraints (data or-parallelism), and lazy evaluation as control structures that can express disjunction. Looking back, we can now categorize the interpreters according to which subset of these three methods they implement. Using **cb** for control backtracking, **db** for data backtracking (lazy evaluation of disjunctive constraints), and **dop** for data or-parallelism, the table below identifies for each nonempty subset of {**cb** **db**, **dop**}, the corresponding interpreter.

Subset	Name	Section(s)	How Substitutions are Represented
cb	Prolog	3.1	(single substitution)
cb,dop	Eager MultiLog	3.2, 3.3	∞-streams (lists)
cb,db	Lazy MultiLog	3.4	1-streams
cb,db,dop	Mixed MultiLog	3.4	n-streams
dop	Eager Direct Style	3.5	∞-streams (lists)
db	Lazy Direct Style	3.5	$1-$streams
db,dop	Mixed Direct Style	3.5	n-streams

The pure Prolog interpreter in Section 3.2 uses only control backtracking to express disjunction. The basic MultiLog interpreter in Section 3.2 uses control backtracking and eager disjunctive constraints, as does the interpreter in Section 3.3. (The two differ on whether all solutions to `disj` goals are obtained at once.) The interpreters in Section 3.4 potentially use all three of control backtracking, constraint backtracking, and eager disjunctive constraints. The direct style interpreter in Section 3.5 uses (lazy or eager) constraint backtracking, but no control backtracking.

Control and data backtracking are both forms of lazy evaluation. Control or-parallelism, a restriction of breadth-first search, arises from concurrent exploration of multiple backtrack points. Conversely, control backtracking arises from lazy evaluation of alternative choice points (failure continuations). Similarly, data or-parallelism results from concurrent unification in multiple substitutions (disjunctive constraints); this occurs by virtue of a data structure representing multiple

```
(define (solve Goal Substs)
  (if (eq? Substs '()) '()
    (match Goal
      (true Substs)
      ((G1 ',' G2) ; conjunction
          (solve G2 (solve G1 Substs)))
      ((G1 ';' G2) ; disjunction
          (combine-streams (solve G1 Substs) (solve G2 Substs)))
      ((X = Y) (multi-unify X Y Substs))
    (Pred(Args)   ; Procedure call.
        (let* ( (Definition (rename (definition-of-predicate Pred)))
                (Formals (formals-of-definition Definition)))
            (solve (body-of-definition Definition)
                (multi-unify Formals Args Substs)))))))
```

Fig. 4. MultiLog Interpreter in Direct Style

environment disjunction. In the code, we have replaced the call to `append-streams` by a call to `combine-streams`: we do not wish to specify how many solutions to a `disj` goal are to be computed at once; nor do we wish to specify in what order the solutions should be combined.

Note that if lists instead of streams are used for the `Substs` argument, then the interpreter utilizes data or-parallelism alone to express disjunction. However, since the interpreter then lacks all backtracking, excessive non-termination would occur.

4 Implementing Top-Down Dereferencing and the Engine-Multi Distinction

We now outline a Scheme implementation of the Environment-Tree Model with top-down dereferencing of Section 2.1, and of the distinction between engine and multi-variables of Section 2.2. The main datatype declarations for environment trees are as follows. Again, for readability we use Prolog-like pattern matching syntax.

- EngineVariable = engineVariable(String)
- MultiVariable = multiVariable(String)
- Term= EngineVariable + MultiVariable + Constant + Constant '(' Terms ')'
- EngineBinding = (EngineVariable,Term) — an association
- MultiBinding = (MultiVariable,Term) — an association
- EngineEnv = List of EngineBinding — for bindings of engine variables
- MultiBindings = List of MultiBinding — for bindings of Multi variables
- EnvTree = internal(MultiBindings,List of EnvTree) + leaf(MultiBindings)
- Envs = (EngineEnv,EnvTree) – a pair

The `solve` interpreter is of type

that for a disjunction of equations $E_1 \vee E_2 \vee \ldots$ to be satisfiable, it is sufficient that just one E_i is satisfiable.

In fact, the repercussions of this change are far-reaching, since it leads to a form of *two-dimensional backtracking* in which backtracking occurs both in the control component (Prolog's standard form of backtracking) and in the constraint component (the computation of unifications accumulated by previous goals). In the above interpreter, backtracking occurs first in the constraint component, in the sense that the unprocessed tail of the stream is expanded further. Only if the tail of the stream expands to the empty list does control backtracking occur.

We use the terms *constraint backtracking* and *data backtracking* to refer to the lazy evaluation that occurs in the `Substs` argument to the interpreter. Two-dimensional backtracking, then, results from the combination of (standard) control backtracking and constraint backtracking.

Even Lazier Generation of Constraints The notion of two-dimensional backtracking can be generalized further. In the definition of `solutions` in Section 3.2, replace `(append Substs1 (FC1))` with `(append-streams Substs1 FC1)`, where `append-streams` is defined as follows:

```
(define (append-streams Str Closure)
  (if (null? Str)
    (Closure)
    (cons-stream (head Str) (append-streams (tail Str) Closure))))
```

Then not all solutions to a `disj` goal are obtained at once. Instead, solutions are obtained on demand, just as substitutions satisfying multi-unifications are obtained on demand by the change of Section 3.3. In this way, the interpreter is even 'lazier' than the one of the previous section, where `solutions` forced the eager collection of solutions to a `disj` goal.

And by using an n-stream, one gets something similar to the effect of dynamic reversion to backtracking of Section 3.3 — even without redefining `solutions` to return subsets of solutions at once. As many substitutions can be computed concurrently as there are resources to compute them.

In fact, using streams, one can eliminate backtracking in the control part altogether, as we show in the next section.

3.5 Back to Direct Style Using Streams

The role of the failure continuation argument to the `solve` interpreter of Figure 3 is to process answers to ";" goals one at a time, postponing subsequent choices until they are demanded by the user or by backtracking. In effect, `solve` returns a stream of sets of answer substitutions, and we can rewrite `solve` into direct style using streams so that `solve` has type Goal $\times$ (Stream of Substs) $\rightarrow$ (Stream of Substs). The resulting interpreter, shown in Figure 3.5, resembles the stream-based logic programming interpreter in Chapter 4 of [1].

For this interpreter (Figure 4), the `SC` and `FC` parameters to `solve` have been eliminated. Furthermore, there is only one form of disjunction: MultiLog's multiple a `disj` goal to the success continuation, along with a revised failure continuation that, when called, re-invokes the goal to get further solutions.

Let us alter the type of Answer from Answer: SetofSubsts to Answer: SetofSubsts → SetofSubsts, and replace the code for `solutions` with the new code below. The extra SetofSubsts argument to `Answer` is used for holding the collected solutions to a `disj` goal. No other changes to the interpreter are needed, because only the definition of `solutions` constrains the type of `Answer`. The top level call to `solve` should pass in a `SetofSubsts` argument, but thanks to 'eta-reduction', this argument to `Answer` need not appear in the code for `solve`. In this sense, `Answer` is a type parameter, and `solve` is polymorphic in the type of `Answer`, which determines the type of `Solve`, `SC`, and `FC`.

```
(define (solutions Goal Substs SC FC)
  (solve Goal Substs
    (lambda (NewSolns FC1) (lambda (OldSolns)
      (let ((CombinedSolns (append NewSolns OldSolns)))
         (choose CombinedSolns
           (lambda () ((FC1) CombinedSolns)) ; collect more solutions
           (lambda () ; that's enough for now
              ((SC CombinedSolns FC1) '()))))))
    (lambda () (lambda (solns)
       ((SC solns FC) '())))))
```

Each time `Goal` succeeds, the third argument to the call to `solve` is invoked. If `Goal` fails, the last argument is invoked. The function `choose` decides, based on the size of `CombinedSolns`, whether to collect more solutions or to pass the already collected solutions to the success continuation. In the latter case, the failure continuation `FC1` passed to `SC` will, if called, continue the search for solutions to `Goal`.

3.4 Lazy MultiLog: Streams and Constraint Backtracking

A significant modification of the `solve` interpreter of Section 3.2 is to remove the requirement that the call `(multi-unify T1 T2 Substs)` perform unification in *all* the substitutions in `Substs`. Instead, it is sufficient for unification to succeed for at least one substitution. The needed modifications are simple: alter the type of `Substs` from `List of Substs` to `Stream of Substs` and replace in the code for `multi-unify` `cons` with `cons-stream` and `cdr` with `tail`.

With a bit more work the code can be rewritten to process an *n-stream* of substitutions, for some finite $n \geq 1$. In an n-stream, up to n elements are computed eagerly; the nth `cdr` being a closure. Standard streams are 1-streams and standard lists are ∞-streams, in an obvious sense. This change would preserve the possibility of a data-parallel implementation but would allow the system to better constrain the available concurrency.

From an operational point of view, the change from lists to streams has the consequence that the substitutions will be created lazily, on demand; not all substitutions in the constraint component will be reduced to solved form at each resolution step. From a logical point of view, the change from sets to streams reflects the fact

```
(define (solve Goal Substs SC FC)
 (if (null? Substs) (FC)
  (match Goal
    (true    (SC Substs FC)) ; The empty goal
    (G1 ',' G2   ; Conjunction
        (solve G1 Substs
            (lambda (Substs2 FC2) (solve G2 Substs2 SC FC2))
            FC))
    (G1 ';' G2   ; Regular, backtracking disjunction
        (solve G1 Substs SC (lambda () (solve G2 Substs SC FC))))
    ((disj G1)   ; MultiLog, multi environment disjunction
        (solutions G1 Substs SC FC))
    (X = Y       ; Multi-unification
        (SC (multi-unify X Y Substs) FC))
    (Pred(Args)   ; Procedure call.
        (let* ( (Definition (rename (definition-of-predicate Pred)))
                (Formals (formals-of-definition Definition)))
            (solve (body-of-definition Definition)
                (multi-unify Formals Args Substs)
                SC FC))))))
```

Fig. 3. Scheme Code for MultiLog Interpreter

```
          (multi-unify T1 T2 (cdr Substs))))))
```

We omit the code for `unify-single`, which returns either `'#f`, indicating failure of unification, or a representation of the substitution resulting from unification.

The code for `solutions`, used to implement disjunctive goals `disj` G, is particularly concise:

```
(define (solutions Goal Substs SC FC)
  (SC (solve Goal Substs
         (lambda (Substs1 FC1) ; success continuation
            (append Substs1 (FC1)))
         (lambda () '())) ; failure continuation
       FC))
```

The procedure collects all solutions to `Goal` that extend some substitution in `Substs`; it does this by calling `Solve` with a success continuation that appends the returned list with the list returned by invoking the failure continuation. The final failure continuation returns the empty list.

3.3 Returning Subsets of Solutions to `disj` Goals

The MultiLog interpreter of the previous section collects all solutions to each `disj` goal. In this section we modify the interpreter to pass subsets of the solutions to

```
(define (solve Goal Subst SC FC)
 (if (failed-substitution? Subst) (FC).
  (match Goal
    (true    (SC Subst FC)) ; The empty goal
    (G1 ',' G2   ; Conjunction
        (solve G1 Substs
            (lambda (Subst2 FC2) (solve G2 Subst2 SC FC2))
            FC))
    (G1 ';' G2   ; Regular, backtracking disjunction
        (solve G1 Subst SC (lambda () (solve G2 Subst SC FC))))
    (X = Y       ; unification
        (SC (unify X Y Subst) FC))
    (Pred(Args)   ; Procedure call.
        (let* ( (Definition (rename (definition-of-predicate Pred)))
                (Formals (formals-of-definition Definition)))
            (solve (body-of-definition Definition)
                (unify Formals Args Subst)
                SC FC))))))
```

Fig. 2. Scheme Code for Prolog Interpreter

tive constraints. The code is almost identical to the code of the previous section, the only differences being the presence of the line for `disj` goals, and the presence of multiple substitutions in second argument to `solve`.

For this interpreter assume types

- SetofSubsts= List of Subst
- SuccessContinuation= SetofSubsts $\times$ FailureContinuation $\rightarrow$ Answer

and let the type variable Answer be SetofSubsts. The function `solve`, shown in Figure 3, is of type (Goal $\times$ SetofSubsts $\times$ SuccessContinuation $\times$ FailureContinuation) $\rightarrow$ Answer. A call `(solve Goal Substs SC FC)` executes `Goal` in each environment in `Substs`. If `Goal` fails in each environment, then `solve` invokes `FC`. Otherwise, it invokes `SC` with the set of consistent extended environments and with an updated failure continuation.

The function `multi-unify` is of type Term $\times$ Term $\times$ SetofSubsts $\rightarrow$ SetofSubsts. For each substitution θ_i in `Substs`, `(multi-unify T1 T2 Substs)` performs the unification `T1`θ_i =`T2`θ_i; the unification either fails or results in a substitution σ_i. `multi-unify` returns the set of all substitutions $\theta_i\sigma_i$ such that the unification succeeds.

```
(define (multi-unify T1 T2 Substs)
  (if (null?  Substs)
    '()
    (let ((first-res (unify-single T1 T2 (car Substs))))
      (if first-res
          (cons first-res (multi-unify T1 T2 (cdr Substs)))
```

Assume that there are primitive types Variable, Constant, and `PredicateSymbol`, and type constructors $\rightarrow$ (function) $\times$ (product), $+$ (union), and List of. Also assume that ',' (comma), ';' (semi-colon), and $=$ (equals) are infix constructors and that disj is a prefix constructor. Figure 1 defines various types used to indicate the types of the interpreters' functions. The variable Answer is a type variable whose value (a type) varies among the various interpreters.

- Term= Variable + Constant + Constant '(' Terms ')'
- Subst= Variable → Term
- Terms= Term + (Term ',' Terms)
- Goal= true + (Goal ',' Goal) + (Goal ';' Goal) + (disj Goal) + (Term = Term) + PredicateSymbol(Terms)
- SuccessContinuation= Subst × FailureContinuation → Answer
- FailureContinuation= → Answer

Fig. 1. Types Used in the Interpreters

3.1 A Standard Prolog Interpreter: Control Backtracking

Figure 2 displays a standard Prolog interpreter utilizing control backtracking (failure continuations) to express disjunction. The use of continuations to model the operations of the control and choice stacks of Prolog is a well-known technique (e.g., [14], [7]). The Prolog interpreter in Figure 2 is displayed for comparison purposes only.

The function `solve` is of type (Goal × Subst × SuccessContinuation × FailureContinuation) → Answer, where the type Answer is arbitrary and depends on the instantiation of `SC` and `FC` in the top-level call to `solve`. A call `(solve Goal Subst SC FC)` executes `Goal` in the context of the single substitution `Subst`. If `Goal` fails in this environment, then `solve` invokes `FC`. Otherwise, it invokes `SC` with an extended environment `Subst` and with an updated failure continuation. For readability, the code uses Prolog-style infix operators for data of type Goal. It assumes the existence of a global database of clauses for user predicates, accessible by the call `definition-of-predicate`.

The code for `(G1 ';' G2)` tries solving `G1` first, with a failure continuation that tries `G2`. So the interpreter implements a depth-first search strategy. Similarly, in a conjunction `(G1 ',' G2)`, `G1` is done first, with a *success* continuation that does `G2`. The function `unify` is of type Term × Term × Subst → Subst. We omit the listings of `unify` and other support code whose functionality should be obvious.

3.2 A Scheme Interpreter of MultiLog: Backtracking and Disjunctive Constraints

In this section we exhibit Scheme code for an interpreter of MultiLog's multiple environment model of logic programming using control backtracking and disjunccomponent share equations resulting from normal multi-resolution steps appearing in the multi-derivation up to that point.

The significance of this fact is that disjuncts appearing together in a constraint component share many of the same bindings; only bindings dependent on `disj` goals can differ between substitutions appearing together in the constraint component. This fact is the basis for the distinction between 'engine' (sequential) and 'multi' (parallel) variables.

Our implementation represents the abstract constraint component C in the concrete form $\alpha \wedge \beta$, where α is a conjunction (a substitution) representing the shared, common bindings, and β is a disjunction representing the bindings that differ among substitutions. Variables bound in α are called *engine* (sequential) variables; variables bound in β are called *multi* (parallel) variables [23], [27]. Unifications involving engine variables are faster than unifications involving multi variables, since the disjuncts in β need not play a role. Engine unifications are done globally, once per *subset* of solutions to the generator goals, rather than once per solution.

Consider, for example, the following program and query, which binds `L` to lists of binary digits.

```
bit(0). bit(1).
bits([]).  bits([H|T]):- disj bit(H),bit(T).
| ?- bits(L).
Yes  L = [].  More? y
Yes  L = [A], (A=0 or A=1).  More? y
Yes  L = [A,B], (A=0,B=0 or A=0,B=1 or A=1,B=0 or A=1,B=1).  More? y
....
```

The `disj`-independent variable `L` and each `cdr` of `L` (the variable `T` in the body of the second clause for `bit/1`) get bound either to `[]` or to a `cons` cell. ($A_i = H$ corresponds to `bits(L)=bits([])` or to `bits(L)=bits([H|T])`.) It is reasonable to store the bindings of `L` and `T` once, globally. This representation is reflected in the format of the output in the example query above.

3 The Interpreters

In this section we present a series of logic programming interpreters written in the programming language Scheme [1], with which we assume the reader is familiar. By writing the interpreters in Scheme we can make explicit things that would likely be hidden (or awkward) in a Prolog implementation: binding environments (substitutions), success continuations (the control stack), and failure continuations (the choice stack).[5] We note, by the way, that as is often the case with logic programming meta-interpreters, a MultiLog meta-interpreter has just one new clause: `solve(disj G):- disj G.`

[5] The continuation arguments SC and FC are represented by higher order functions (closures), *not* by Scheme continuations generated by `call-with-current-continuation`.

tiple binding environments, with unification performed 'in parallel' on the multiple substitutions.

In a depth-first implementation of multi-SLD resolution, when control backtracks into the atom a, another, non-empty, finite subset of solutions is collected, and so on. In this way, the solutions to a generator in a generate-and-test program are enumerated subset by subset, instead of one at a time.

The various substitutions in the constraint component of the abstract machine state share much structure, and the unifications in the various substitutions are not, after all, independent. The redundancy among substitutions is the basis for powerful optimizations that are described in the next two subsections.

2.1 Sharing from `disj` Multi-resolution Steps: Environment Trees

After a multi-resolution step, each of the substitutions in the constraint component extends some unique input substitution from before the step. Moreover, for `disj` multi-resolution steps, any given input substitution can have multiple output substitutions extending it; if the argument goal succeeds m times then each input substitution can have up to m child substitutions.

As multiple `disj` goals are encountered, execution results in an implicit tree of substitutions, organized according to the parent-child relationship. Each surviving disjunct in the constraint component extends some ancestor disjunct in each previous constraint component of the multi-derivation. These ideas are formalized in the notion of *environment tree*, which refers to the tree of surviving substitutions organized according to the parent-child relationship [27].

The structure of the environment tree motivates the representation called the Environment-Tree Model in which environments are stored in the form of a tree with shared ancestor bindings.The alternative representation in which each environment is a vector requires the copying of input environments during the collection of solutions to `disj` goals.

The use of an environment tree enables an important optimization called *top-down dereferencing* [24], whereby dereferencing is performed by a downward inorder traversal of the environment tree, instead of by searching upward in each association list. Top-down dereferencing leads to a savings of $O(\log n)$ time (n is the number of environments), compared to the naive model in which dereferencing occurs independently from each leaf [24]. Top-down dereferencing also allows early detection of success or failure of entire branches of the tree. Section 4 outlines a Scheme implementation of top-down dereferencing.

2.2 The Distinction between Engine and Multi Variables

Consider that a multi-SLD derivation consists of a sequence of normal multi-resolution steps interspersed with (occasional) `disj` multi-resolution steps. Each surviving disjunct after a normal step is consistent with the head unification associated with that step. And each disjunct contains, where appropriate, equations (bindings) resulting from the head unification associated with the step. Consequently, at any step of a multi-derivation, the various disjuncts of the constraint

The plan of the paper is as follows. Section 2 informally describes multi-SLD resolution and summarizes some of its properties. In Section 3 we present a series of working MultiLog interpreters written in typed, almost[3] pure, Scheme [1]. The basic multi-SLD interpreter of Section 3.2 collects all solutions to a `disj` goal; its code differs only slightly from the code for an SLD interpreter in Section 3.1. Section 3.3 exploits parametric polymorphism in the interpreter of Section 3.2 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 3.4 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 3.5 uses lazy evaluation to eliminate control backtracking altogether. Section 4 sketches an implementation of top-down dereferencing, an optimization whose logical justification is sketched in Section 2.1. Section 5 categorizes the interpreters of Section 3 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 6 covers related work. Section 7 concludes.

2 Overview of Multi-SLD

The abstract machine state of a multi-SLD interpreter consists of two components: a list of goals, and a disjunction (set) of substitutions.[4] There are two sorts of multi-SLD resolution steps. In a normal multi-SLD resolution step, some atom is selected from the goal list and resolved against some clause in the program; since there are multiple substitutions in the constraint component, unification of the atom with the head of the clause occurs independently in the various substitutions. If any substitutions survive the resolution step, then the surviving substitutions, extended with the bindings resulting from head unification, become the constraint component of the next abstract machine state, whose goal list is found by replacing the selected atom with the body of the clause.

In a `disj` multi-SLD resolution step, a subcomputation is begun on the selected atom a (which in practice is annotated by the unary control operator `disj`) and some *finite*, nonempty subset of the solutions to a is collected and installed as the new constraint component. The new goal component consists of the previous goal list minus the selected atom. Subsequent goals execute in the context of these mul-

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

[4] More generally and from the viewpoint of CLP, the second component consists of a disjunction of allowed constraints.

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 [23],[26]. 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]

```
| ?- 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 [11] 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 2.2) this shared computation can be 'factored out' and performed only once.

Previous papers [23] [22] 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. [25] presents a model that explains the observed speedups. [26] formalizes multi-SLD resolution, examines some of its properties, and proves its soundness and completeness. [24] presents an analysis of environment representation schemes for MultiLog. The author's dissertation [27] 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.

[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.

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

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

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.