Updates in a Temporal Logic Programming Language

John G. Cleary* and Vinit N. Kaushik**

University of Calgary,**
and Jade Simulations International,*,**

Calgary, Alberta, Canada

Abstract

Starlog, a pure logic programming language including negation is introduced. It is shown that this language is capable of directly expressing the mutation and change of clauses within a database. This is the problem traditionally solved in Prolog with the semantically-unsatisfactory "assert" and "retract." The standard, minimal-model semantics for normal logic programs is preserved in Starlog, and no extensions are used beyond standard clausal logic. It is noted that by extending startification to a "temporal stratification" the completion of such programs is guaranteed to be consistent. Some short, example programs are described and used to illustrate an effective and efficient bottom-up technique for executing the language based on connection graphs. The execution technique solves the problems of matching generated results against goals and of efficiently and correctly dealing with negations.

1. Introduction

One of the major problems that has resisted effective solution within the logic-programming paradigm has been the representation of change and mutation of data. For example, within a relational-database framework, it is acceptable to directly modify the relations in the database by inserting and deleting tuples. The most natural representation of a relational database within logic programming is as a predicate, with individual tuples being unit clauses of the predicate. To modify this predicate, however, requires that the program modify itself. A pragmatic solution has been to include the `assert` and `retract` primitives within Prolog. Unfortunately, they do not have a simple, declarative semantics.

Change and mutation can be represented explicitly within pure Prolog by using a data structure, such as a list, to represent a sequence of successive states. However, the early part of the lists cannot be readily garbage collected, and so their size grows without bound. This problem is solved in the Concurrent Prolog (CP) family of languages (Shapiro, 1987). Such languages are intended to be applicable to non-terminating programs, such as operating systems. Typically, sequences of states are represented by lists—as suggested above—but in the CP languages, the lists can be garbage collected because the early stack frames are deleted. Thus, the lists and the stack need not grow without bound, and non-terminating programs that execute in finite space can be implemented. Unfortunately, the semantics of the CP languages are complicated by the need to use commitment and don't-care non-determinism. Thus, the standard, least-fixpoint semantics gives only a set of possible answers for the program. It can be difficult to tell which will in fact be generated or indeed if a program is capable of producing any of the answers. For example, some form of delay is used in all the CP languages. Used incorrectly, such delays can lead to deadlock and failure of the program. Such behavior is not accessible to the least-fixpoint semantics.

Various techniques have been proposed to allow updates within deductive databases (Liu and Cleary, 1990), (Naish *et al* , 1987), (Naqvi and Krishnamurthy, 1988). Again, these extend the underlying logic with new, extra-logical primitives, thus complicating or destroying the declarative nature of the semantics.

In this paper, a very different approach to this problem of modelling change is proposed. Time is made explicit within the language. Change and mutation then become ideas that are directly expressible within (the logic of) the language. For example, consider a data base with a predicate `data(Time,Key,Value)` whose interpretation is that at a particular Time, the Key has a Value. The value of the key may change with time. Suppose the key "akey" had the value "a" at time 0, and later at time 9, it had changed to b. This can be expressed by the following pair of time-stamped tuples:

```
    data(0,akey,a)      data(9,akey,b)
```

(The term tuples is used here for—not necessarily ground—unit clauses, by analogy with relational databases, Linda, and other dataspace languages.)

Space precludes us from dealing in detail with the semantics of Starlog. However, it is possible to extend the notion of stratification for normal programs to Starlog (see (Lloyd, 1987, §14) for a description of these semantics). Instead of stratifying the program using a static ordering based on predicate names the execution is stratified on the time values in the predicates. A Starlog program then needs to satisfy a number of conditions to be sure that it has a minimal model. The main condition is that the program should be causal, that is, the time in the head is always greater than or equal to the times in the body. Another condition is that any negation should be accompanied by a finite time advance (the condition is actually weaker and more complex than this). Given these conditions a model can be constructed in almost the same way as for the standard stratification.

A number of other temporal programming languages have been proposed (Galton, 1987). Starlog differes from them in two ways. First it has a standard declarative semantics and second it uses full real valued times (as opposed to integer values). The first is important because the tools and background of logic programming can be brought to bear on Starlog. The second is important because it greatly extends the expressiveness and power of the language. However, this expressiveness is only possible because of the novel execution techniques available in Starlog and described in Section 3.

The next section introduces the language Starlog, which is an incarnation of these ideas. The introduction is done by way of a database-update program. Section 3 demonstrates, for Starlog, an execution model based on connection graphs. This is done by using two, different programs: one to solve the 2,3,5 problem and another that expands on the update program of section 2. Section 4 concludes with a discussion of the strengths of Starlog and future research directions.

2. The Starlog Language

Starlog is a pure, logic-programming language that allows negation in the body of clauses. Each user-defined predicate has its first parameter reserved for a time-stamp value, e.g., see the data predicate above. Time stamps are real numbers greater than or equal to 0.

As shown above, it is easy to express that a fact is true at a particular time, e.g., `data(0,akey,a)`. However, this is not sufficient to express a notion such as "from time 0 to time 9, `akey` had the value `a`, and from time 9 onward, it had the value `b`." As we will see, it is important for the practicality and expressiveness of the language that notions such as this can be expressed. Such notions are expressed by "constrained tuples":

```
data(T,akey,a)  ← 0≤T, T<9;
data(T,akey,b)  ← 9≤T;
```

The constraints that are allowed include all the ordering operators, `<`, `≤`, `>`, and `≥`. Such constraints can be manipulated and used efficiently (see for example (Cleary, 1987)). For example, it is possible to automatically transform a redundant pair such as "`2≤T, 9≤T`" into "`2≤T`."

A critical constraint on programs in Starlog is that they should be "causal." That is, the time of the head of a clause must be later than (or equal to) any goal in the body. This constraint enables programs to be efficiently executed. A related, although slightly stricter, condition is sufficient to ensure that programs have a well-defined minimal model.

An example of a causal rule is the following, which is a first attempt at a database-update program:

```
data(Tout,Key,Value) ← Tout ≥ Tin, add_data(Tin,Key,Value);        (1)
```

This says that from the time `Tin`, at which the tuple "`add_data(Tin,Key,Value)`" occurs, the `Key` will have that value. The condition "`Tout ≥ Tin`" ensures that the `Key` takes on the value only after the "`add_data`" tuple occurs. This also ensures that the clause is causal. In Starlog, such an ordering condition is added implicitly to every goal in the body of a clause, together with the condition "`Tout≥ 0`," where `Tout` is the time in the clause's head.

The database-update program above is incomplete. Consider what happens when the two tuples "`add_data(0,akey,a)`" and "`add_data(9,akey,b)`" are introduced. Using bottom-up execution, they can be matched against the "`add_data`" goal. This generates two constrained tuples:

```
data(Tout,akey,a) ← Tout ≥ 0;
data(Tout,akey,b) ← Tout ≥ 9;
```

This is not the required result as at any time from 9 onward, `akey` has both the values `a` and `b` associated with it If the value of `akey` is to be changed—rather than added to—then there should be only one associated value, `b`. This problem can be resolved by noting that the real meaning when a value is changed is "`Key` takes a value from `add_data` until the next value is assigned by `add_data`." Paraphrased, this is "`Key` takes a value from `add_data` so long as no other value is assigned by `add_data`." In logic terms, this gives us the more precise rule:

```
data(Tout,Key,Value) ←
      Tout ≥ Tin,
      add_data(Tin,Key,Value),
      not(exists T add_data(T,Key,_), Tout≥T, T>Tin);                (2)
```

The use of negation in this way to limit the temporal scope of a tuple is a key technique in Starlog and enables it to directly represent update and change. To see this in more detail, consider how the rule (2) might be executed with the tuples "`add_data(0,akey,a)`" and "`add_data(9,akey,b)`" as input.

Execution is bottom up so the two tuples can first be resolved against the positive goal to yield the following two derived clauses. (The goals that have been resolved away are shown with ~~a line through them.~~)

data(Tout,akey,a) ←
      Tout ≥ 0,
      ~~add_data(0,akey,a),~~
      not(exists T add_data(T,akey,_), Tout≥T, T>0);   (2a)

data(Tout,akey,b) ←
      Tout ≥ 9,
      ~~add_data(9,akey,b),~~
      not(exists T add_data(T,akey,_), Tout≥T, T>9);   (2b)

Resolving the tuples against the goal inside the negation transforms the first of these (2a) into:

data(Tout,akey,a) ← Tout ≥ 0,
      not(~~add_data(0,akey,a)~~, Tout≥0, 0>0),
      not(~~add_data(9,akey,b)~~, Tout≥9, 9>0);

Using the fact that `0>0` fails and `9>0` succeeds, this further transforms to:

```
data(Tout,akey,a) ← Tout ≥ 0, not(Tout≥9);
```

"`not(Tout≥9)`" is equivalent to "`Tout<9`", so this can finally be transformed to the following constrained tuple:

```
data(Tout,akey,a) ← Tout ≥ 0, Tout<9;
```

This says that `akey` has the value `a` from time 0 up to, but not including, time 9. A similar transformation on (2b) yields:

data(Tout,akey,b) ← Tout ≥ 9,
      not(~~add_data(0,akey,a)~~, Tout≥0, 0>9),
      not(~~add_data(9,akey,b)~~, Tout≥9, 9>9);

0>9 and 9>9 both fail so this reduces to:

```
data(Tout,akey,b) ← Tout ≥ 9;
```

which says that `akey` has the value `b` only, from time 9 onward, as we wanted.

This example execution has ignored two critical issues. The first is that the transformation of the negations depended on the implicit information that there were no other "`add_data`" tuples in the execution. This will not be true in more elaborate programs which may have arbitrarily complex rules generating the `add_data` tuples and where there may as well be an infinite stream of them. The second issue is that a naive implementation of the execution would match every tuple as it is generated against all goals, which might match in the original rules and in the clauses derived from them. This would be very inefficient, especially in large programs. Section 3 deals with both these issues.

The properties of Starlog, exemplified above, can be summarized as:

- it is a pure, clausal, logic-programming language that allows negative goals;
- the result of a program is a set of "constrained tuples";
- execution is bottom up and also uses transformations to simplify negations and constraints;
- every tuple has its first parameter reserved for a time value, which is a positive real number;
- rules must be causal, that is the time in the head must be greater than or equal to the times in the body.

3. Execution of Starlog

This section will informally describe how to execute Starlog by examining in detail two different Starlog programs. The first example is the famous 2,3,5 problem whose main virtue is that it is simple and straightforward. The second example is an example of how information can be updated, in this case how a counter can be periodically incremented.

These examples should not be taken as representative of the areas where we see Starlog being applicable. We have explored Starlog programs in a wide range of areas—meta-programming including a meta-interpreter for Starlog; graphics including a graphical display of the execution of a Starlog program; and search programs including interpreters for pure Prolog. An example of a combined continuous/discrete simulation program in Starlog is described in (Cleary, 1990).

Starlog uses a variant of connection-graph theorem proving for its execution. (Kowalski,1979) and (Bibel,1985) contain descriptions of this technique as used in general-purpose theorem proving. The basic idea in connection-graph theorem proving is that the literals in a (general) graph can be joined by links. Only those literals that have opposite signs and can be unified are initially linked. The main rule used in connection-graph theorem proving is a variant of the resolution rule. Any link can be deleted by resolving the two terms that are linked resulting in a new clause. The old link is deleted, and the new clause inherits links from its parents.

A connection graph in Starlog is a variant of this basic idea. Links are now directional, from a goal in a body to the head of a clause that can unify with it. Rather than a single resolution rule, a number of more specialized rules are used to manipulate such a Starlog connection graph. The full resolution is done only when the target of a link is a constrained tuple. The five Starlog rules are:

1. if a goal has a number of links (n) from it then the entire clause (and the links to its head and from the other goals) can be duplicated n times;
2. if the goal and head joined by a link do not unify then the link can be deleted;
3. if a goal has no links then the entire clause can be deleted;
4. if the goal and head joined by a link do unify and the goal has only one link then the resulting binding can be applied to the goal (and consequently to the whole clause), for example, if the goal `p(X)` points to `p(f(Y,3))` then `X` can be bound to `f(Y',3)` where `Y'` is a newly introduced

variable; it is also possible to apply constraints on values, for example, if the time in the head is constrained to be `T>6` then this same constraint can be propagated back to the goal;

5. if the only head pointed to by a goal is the head of a constrained tuple then a full resolution between the goal and the clause is done.

The control strategy used by Starlog to apply these rules starts with the selection of (one of) the clauses with the lowest time stamp. (Tuples having no proper goals need not be selected. The time is determined by earliest time the clause could succeed given the constraints on its time variable. For example, if T≥6, T≤10 then the time 6 is used as the time.) Rule 1 or 5 is then applied to all the goals in the selected clause, (depending on whether they point to a constrained tuple or to a proper clause). This leads to a number of new clauses. Most goals in these new clauses point to just one head, the exception is when a goal was linked to the head of its own clause. So rule 4 is then used to propagate bindings back to all these goals. This will cause some links to fail (by rule 2) and allows the deletion of some of the new clauses. It is more efficient to perform these operations in a single unified operation, however, it makes explanation simpler to break them down in this way. The Starlog cycle continues with the selection of another low timestamp clause until there are no more clauses. The following subsections show some examples of the operation of this cycle.

The 2,3,5 Problem

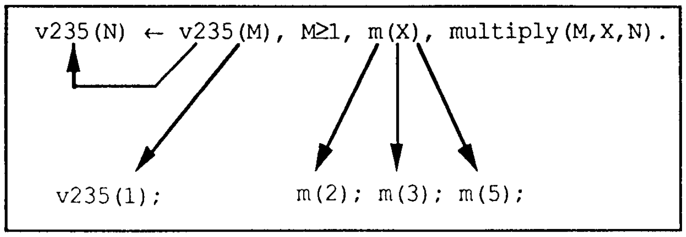
The 2,3,5 problem is to generate all numbers that are products of powers of 2, 3 and 5. The list should be in order and contain no duplicates. The list is, in order, 1,2,3,4,5,6,8,9,10,12,15,16,18,20, . . .

The Starlog program for this generates the numbers in "time" order in the tuple `v235()`. The program is as follows:

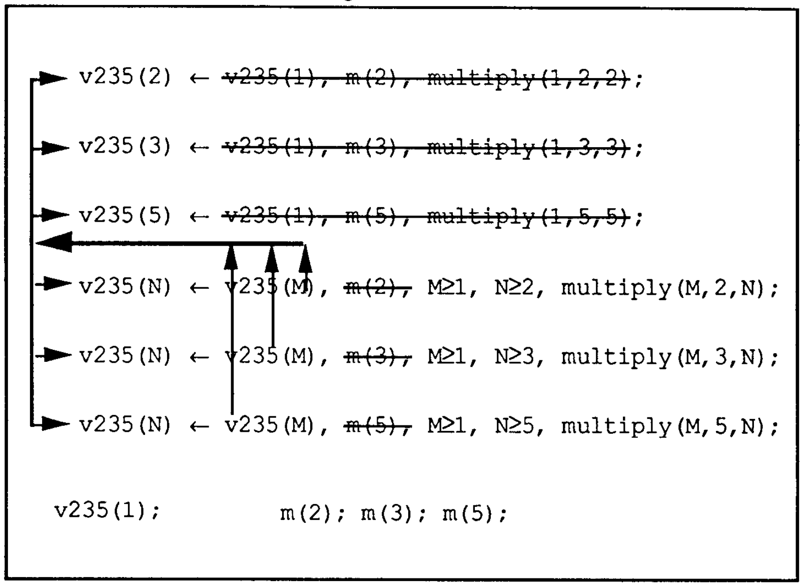
```
v235(1);
v235(N) ← v235(M), M≥1, m(X), multiply(M,X,N);

m(2); m(3); m(5);
```

The first line starts the sequence by saying that 1 is part of the sequence. The next rule says that if M is part of the sequence and X is one of 2, 3, or 5 then `M*X` is also in the sequence. The predicate `m()` is used to hold the allowed multipliers. The diagram below shows the initial links for the program. The goal `m(X)` has three links (to each of the `m()` unit clauses) and `v235(M)` has two (to the unit clause and to the head of its own clause).



There is only one clause which can be selected and after applying rules 1 and 5 to the two proper goals, 6 new clauses replace the original. Rule 4 is then used to propagate all the bindings. No clauses need be deleted and the new connection graph is shown below. (Constraints that have been satisfied and goals deleted by rule 5 are shown with a ~~strike through~~).



v235(2) ← ~~v235(1), m(2), multiply(1,2,2)~~;

v235(3) ← ~~v235(1), m(3), multiply(1,3,3)~~;

v235(5) ← ~~v235(1), m(5), multiply(1,5,5)~~;

v235(N) ← v235(M), ~~m(2),~~ M≥1, N≥2, multiply(M,2,N);

v235(N) ← v235(M), ~~m(3),~~ M≥1, N≥3, multiply(M,3,N);

v235(N) ← v235(M), ~~m(5),~~ M≥1, N≥5, multiply(M,5,N);

v235(1); m(2); m(3); m(5);

The 6 new clauses include three tuples `v235(1), v235(2), v235(5)`. The tuple `v235(1)` and the three tuples for the `m()` predicate do not take part in any further computation because they are not linked to anything. Note that this is like compiling out the references to `m()`: that is these clauses do not need to be looked at again and specialized clauses have been generated for each case. In the new state the 6 new clauses are each pointed at by the 3 `v235(M)` goals giving a total of 18 links.

On the next step the clause "`v235(N) ← v235(M), M≥1, N≥2, multiply(M,2,N)`" is selected because its earliest time stamp is 2. In fact it generates the tuples `v235(4)`, `v235(6)` and `v235(10)` immediately and others later. As execution continues in this way some answers are generated more than once. For example, `v235(6)` is generated two different ways because 6 = 2*3 and 3*2. Starlog ensures that only one copy of `v235(6)` is actually used to generate further answers (if this deletion of duplicates was not done the program could grow exponentially faster than it needs to). This deletion of duplicates can be done efficiently by keeping pseudo links between the heads of clauses that can possibly be duplicates of each other. Space precludes us from giving full details of how to efficiently delete duplicates.

Counting

The 2,3,5 program above does not address the problem of negation nor does it use Starlog's ability to represent the updating of information. One simple problem that does this is to count the number of solutions for an infinite sequence such as the 2,3,5 sequence. In the interests of a simple example the sequence we will count is the multiples of 2. This will be done using a tuple `count(Time,N)` which says that at `Time` there will have been `N` solutions before that time. The correct set of constrained tuples is:

```
count(0,0);
count(T,1) ← T>0, T≤2;  count(T,2) ← T>2, T≤4;
count(T,3) ← T>4, T≤6; ....
```

The multiples of 2 will be generated in the tuples `i2(N)` using the following two program clauses:

```
i2(0) ← ;
i2(N) ← i2(M), add(2,M,N);
```

The program to generate `count` uses a negation construct almost identical to the update example given earlier. In this instance the arrival of a new `i2` value acts analogously to the arrival of the `add_data` tuple. The old value of `count` is then retrieved and incremented to form its next value.

```
count(0,0); %Initialize the counting
count(Tout,N) ← Tout > Tin,
      count(Tin,M), add(1,M,N), %get the old value and add 1
      i2(Tin), %and do it when the value arrives
      %and it remains in effect until the next i2 value arrives
      not(exists T i2(T), Tout>T, T>Tin);
```

Starlog uses a variant of the negation by failure rule to deal with negated goals such as these. The rules can correctly deal with a wider range of negation calls than SLDNF which can only deal correctly with ground negated goals. A subset of the negation rules used are:

N1. if any goal inside a negation has no links then the entire negation can be deleted from the clause.

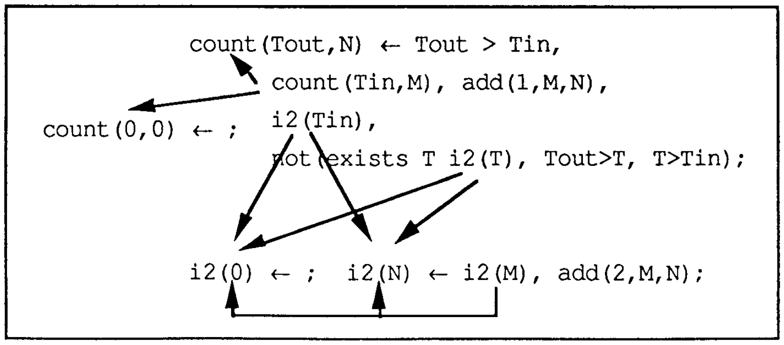
N2. if a goal inside a negation has more than one link then multiple copies (of the negation) can be generated where each of the copies has a goal with a single link;

N3. if there is an empty goal inside a negation then the whole clause can be deleted;

N4. if a negated goal consists of just constraints then it is possible in some cases to rewrite the negation, for example "`not(T>9)`" can be rewritten as "`T≤9`";

This set of rules is not exhaustive, a complete and more detailed list can be found in (Kaushik,1991).

The complete program above has the following initial connection graph.



```
                    count(Tout,N) ← Tout > Tin,
                                count(Tin,M), add(1,M,N),
count(0,0) ← ;                  i2(Tin),
                                not(exists T i2(T), Tout>T, T>Tin);

                    i2(0) ← ;   i2(N) ← i2(M), add(2,M,N);
```

The count clause will be chosen for expansion first. The i2() and count() goals have two links each and so a total of 4 clauses will be generated. However, two of these fail because of incompatible bindings leaving the following two new clauses:

```
count(Tout,1) ← Tout > 0,
      count(0,0), add(1,0,1),
      ~~i2(0),~~
      not(exists T i2(T), Tout>T, T>0);                         (3a)
count(Tout,N) ← Tout > Tin,
      count(Tin,M), add(1,M,N),
      i2(Tin), Tin≥2,
      not(exists T i2(T), Tout>T, T>Tin);                       (3b)
```

At this point the negations have not been expanded but using rule N2 the following version is obtained from (3a) by expanding the two links from the negated `i2()` goal:

```
count(Tout,1) ← Tout > 0,
      not(~~i2(0),~~ Tout>0, 0>0),
      not(exists T i2(T), T≥2, Tout>T, T>0);
```

The goal `0>0` fails allowing this to be simplified to:

```
count(Tout,1) ← Tout > 0,
      not(exists T i2(T), T≥2, Tout>T, T>0);                    (4a)
```

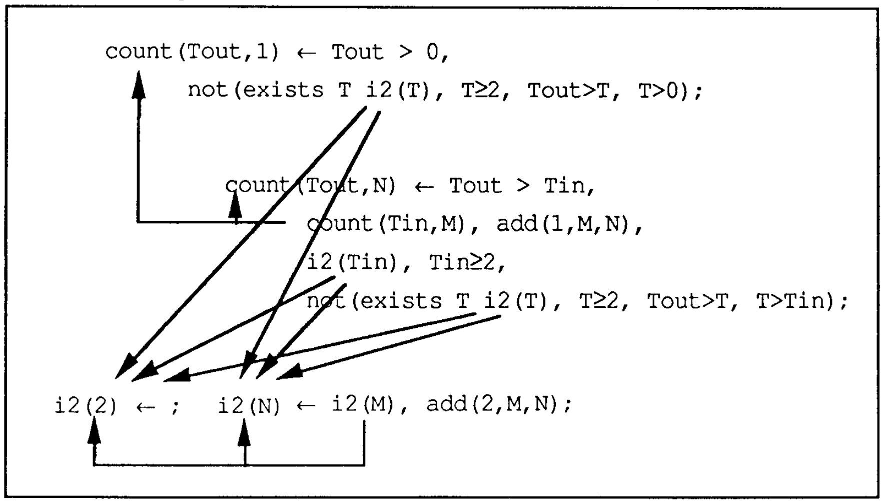
Note the constraint `T≥2` introduced into the negation, this was obtained by using rule 4 to propagate it from the `i2` clause. This clause is the one that will shortly generate the constrained tuple that says the count is one from time 0 to 2. At this point however it only "knows" that the count is 1 after time 0 and has not yet "found out" when the next increments to be done.

Similarly (3b) generates the following clause after expanding both the positive and negated goals:

```
count(Tout,N) ← Tout > Tin,
      count(Tin,M), add(1,M,N),
      i2(Tin), Tin≥2,
      not(~~i2(0),~~ Tout>0, 0>Tin),
      not(exists T i2(T), T≥2, Tout>T, T>Tin);                  (4b)
```

The two constraints `Tin≥2` and `0>Tin` are contradictory so the first negation can be eliminated.

The next clause selected will be the `i2` clause which will generate the new tuple `i2(2)` and a new instance of the generating clause. Combining (4a) and (4b) with the new state of `i2` gives the following new connection graph (the tuple `i2(0)` has no links and has been omitted):



```
count(Tout,1) ← Tout > 0,
      not(exists T i2(T), T≥2, Tout>T, T>0);

      count(Tout,N) ← Tout > Tin,
            count(Tin,M), add(1,M,N),
            i2(Tin), Tin≥2,
            not(exists T i2(T), T≥2, Tout>T, T>Tin);

i2(2) ← ;   i2(N) ← i2(M), add(2,M,N);
```

At this point the negation within the first `count` clause will be expanded yielding the new clause:

```
count(Tout,1) ← Tout > 0,
      not(~~i2(2),~~ 2≥2, Tout>2, 2>0),
      not(i2(T), T≥2, Tout>T, T>0);
```

`2≥2` and `2>0` succeed so the first negation is simplified to `not(Tout>2)` and then to `Tout≤2`. This constraint together with the constraints `T≥2` and `Tout>T` inside the negation are contradictory so that the second negation can be eliminated. The result of all this is the constrained tuple:

```
count(Tout,1) ← Tout>0, Tout≤2;
```

which is the expected first new value for the count. The intelligent propagation of arithmetic constraints and detection of the contradictions between them was crucial to this successful execution. More details of how this can be done are to be found in (Cleary,1987).

5. Conclusions

This paper has shown how Starlog can deal directly and logically with updating information. It has also shown by way of example how such programs can be executed efficiently. This capability of Starlog depends on two critical execution techniques: connection graphs and an intelligent arithmetic constraint system. However, we do not see Starlog as having significance only because it solves this problem. Initial programming has shown it to be able to contribute concise and effective programs in a wide range of areas from simulations, graphics, meta-execution, etc.

The language is basically an OR-parallel logic programming language and so has potential for parallel execution. The connection graph technique lends itself well to parallel execution and can be seen as a refinement and improvement upon the tuple matching algorithms used to implement dataspace languages such as Linda (Carriero and Gelernter,1989). Many of the programming techniques of such dataspace languages are applicable to Starlog. As well, many of the well known features of logic programming, such as "built-in" relational databases, are available within Starlog.

It is also important that Starlog preserves a pure declarative semantics. This means that the large body of techniques such as program transformation, partial execution, algorithmic debugging, type induction etc. which have been developed for logic programs are applicable to Starlog. Combined with the fact that it does not need to step outside logic to be applicable in "real world" problems makes it a powerful potential tool for many difficult programming tasks.

Acknowledgements

This work was assisted by a grant from the Natural Sciences and Engineering Research Council of Canada.

References

Bibel, Wolfgang(1983) "*Matings in Matrices*," Comm. A.C.M., **26**(11), pp. 844-852, November.

Carriero, N., and Gelernter, D.(1989) "*LINDA in Context*," Comm. A.C.M., **32**(4), pp. 444-458, April.

Cleary, J.G.(1987) "*Logical Arithmetic*," Future Computing Systems, **2**(2), pp. 125-149.

Cleary, J.G.(1990) "*Colliding Pucks Solved in a Temporal Logic*," Proc. Distributed Simulation Conference, San Diego, California, January.

Galton, A., Ed. (1987) *Temporal Logics and Their Applications*, Academic Press.

Kaushik, V.N.(1991) "*Starlog: from Semantics to Interpretation*," MSc. Thesis University of Calgary.

Kowalski, R. (1979) *Logic for Problem Solving*, North-Holland, New York.

Lloyd, J.W. (1987) *Foundations of Logic Programming*, Springer-Verlag.

Mengchi Liu, and Cleary, J.G.(1990) "*Deductive Databases: Where to Now?*," to appear in J. Australian Computer Soc. also in Far-East Workshop on Deductive Databases, Melbourne, April.

Mengchi Liu, and Cleary, J.G.(1990) "*Deductive Databases: Where to Now?*," .

Naish, L., Thom, L.A., and Ramamohanarao, K.(1987) "*Concurrent Database Updates in Prolog*," Fouth Int. Conf. on Logic Programming, Melbourne, Australia, pp. 178-195, July.

Naqvi, S., and Krishnamurthy, R.(1988) "*Database Updates in Logic Programming*," ACM Symp. on PODS, pp. 261-262.

Shapiro, E., Ed. (1987) *Concurrent Prolog - Collected Papers*, MIT Press, Cambridge, MA.