

Data Structures Considered Harmful

John G. Cleary, Mark Utting, and Roger Clayton
University of Waikato,
Hamilton, New Zealand.
{jcleary,marku,rjc4}@cs.waikato.ac.nz

Abstract

We describe an approach to logic programming where the execution of a pure logic program is ordered on “temporal” values in the program. The resulting programs are relational and avoid a premature commitment to data structures. This is a strong advantage in a programming language. We present the programming language Starlog as an example of this approach and give a summary of its denotational and operational semantics. Despite being a pure logic language with a monotonic semantics the language is still able to directly mutate data, including assignment of values to variables. Using sophisticated implementation techniques, including bottom-up execution, constraints and the compilation of relations to data structures, we claim that the approach can be executed efficiently.

Introduction

When students are first introduced to logic programming they often get a shock. Rather than seeing a set of assertions about abstract logical entities they instead find themselves having to design data structures and manipulate them. For logicians and for students, this is quite a shock as data structures are not part of the problem nor are they seen as central to logical reasoning. Of course, data structures are traditional and pervasive throughout modern programming. All paradigms including imperative, functional and logical (in the guise of Prolog) programming languages require that procedures be specified in terms of explicit data structures.

Relational databases are a notable exception. Relations are a very abstract way of specifying information that removes the need to consider mappings to memory structures or the details of manipulation. When relational databases were originally proposed by Codd in the 1960's it was seen as a significant advance that programmers no longer had to worry about the details of “navigating” the pointers and hierarchies of the existing data base systems. Part of the surprise of a logician then might be that relations are bona fide constructs within Prolog but that they are used only for expressing static information never for holding the dynamic information within an executing program.

The converse of this is that relational databases are not typically used for programming. The true strength of relational databases is in the queries that can be run against them [TCG93]. Programming in the form of updates breaks the relational paradigm and is done by various *ad hoc* mechanisms. In Prolog, mechanisms such as `assert` and `retract` allow the static relations to be updated but these are much deprecated because they break the semantics of pure logic and are found to be problematic in practice. One consequence of being forced to specify data structures is that programs are typically over specified. The programmer is forced to decide upon and specify things that are irrelevant to the original problem. This in turn makes the program difficult to understand, to modify and to optimise.

This paper describes a programming language that is based upon the simple observation that if the literals in a logic program are temporally ordered then a logic programming language results which has the following properties:

- it is a pure clausal logic programming language with negation;
- it shares its denotational semantics with pure Prolog;
- it is able to express programs that include mutation of data, including assignment;
- it leads naturally to a relational data-structure-free style of programming;

We believe it is possible to efficiently implement the language. Work in progress indicates that a compiler can take advantage of the data-structure-free programming style to select appropriate underlying data representations leading to very efficient compiled code. As well, the use of relations leads to a powerful and open style of programming.

To give an intuition of the approach being advocated here consider a very simple simulation program written in Starlog; the language that we will use to illustrate our approach. The program simulates an engine that can be in one of two states running or stopped. It can be given two commands: start and stop. The following clause states that the engine is running from the time just after it is started:

```
running@T :- start@T0, T>T0.
```

Note the Edinburgh style syntax and the use of the syntactic convention of @ followed by a time variable which provides the temporal ordering. This rule is insufficient on its own as it does not allow the engine to be stopped on a “stop” command. This is achieved by noting that the engine should not be running after a “stop” command. Noting further that the stop must follow the corresponding start the correct clause is:

```
running@T :- start@T0, T>T0, not(stop@T1, T0<T1, T1<T).
```

The complementary “stopped” state can be obtained by negating “running” :

```
stopped@T :- not(running@T), T≥0.
```

If the following facts are added to the program above:

```
start@5. stop@10. start@42
```

then the following results are computed:

```
running@T :- 6≤T, T≤10.
```

```
running@T :- 43≤T.
```

```
stopped@T :- 0≤T, T≤5.
```

```
stopped@T :- 11≤T, T≤42.
```

Starlog is an executable temporal logic. Orgun and Ma [OM94] give an excellent survey of executable temporal logics, including an early version of Starlog (which used real numbers for time, did not use the P@T syntactic convention, and used different implementation techniques). Although Starlog does not directly provide the usual temporal operators (first, next, always, sometime), it is generally possible to obtain an equivalent effect using explicit timestamps and (in some cases) negation.

On the implementation side, our use of triggers in the implementation of Starlog is similar to the techniques used by Wunderwald [Wun96] for converting bottom-up rules into Prolog rules. Triggers can also be regarded as a refinement of the delta-set evaluation technique that is sometimes used in the implementation of bottom-up deductive database systems [TCG93,RSS92]. Our emphasis on avoiding explicit data structures, and allowing the compiler to choose different representations depending on usage, appears to be new in the logic programming context. However, it is similar in concept to using abstract classes in object-oriented programming which can be implemented by different concrete classes without changing the client program. This selection of a concrete class is normally done by the programmer, but we plan to let the compiler choose, based on the results of program analysis as well as user-supplied annotations. The rest of the paper describes these ideas more fully. The next section describes Starlog and gives several more small example programs. The following sections outline its semantics and describe implementation issues. The paper concludes with a short summary.

Starlog

Starlog is a particular programming language that embodies many of the ideas discussed above. All of the examples used in this paper are written in Starlog and can be executed on existing Starlog systems. In this section we will outline the salient points in the design of

Starlog, pointing out inadequacies along the way. Then a number of example programs are explained.

Temporal ordering in Starlog is done on a single variable which is distinguished by following an “@”. Thus a clause in Starlog is of the form:

```
head( $X_1, \dots, X_n$ ) @T :-  $B_1(Y_1, \dots)$ @ $T_1, \dots, \text{not}(C_1(Z_1, \dots)$ @ $T_2), \dots D_1, \dots$ .
```

Implicit constraints are added to all temporal variables: that they are integer and that they are greater than or equal to zero. A causality constraint is also added that the time of the head is greater or equal than the times of all the literals in the body, that is: $T \geq T_1, T \geq T_2$.

Three types of literals can occur in the body: positive temporal goals ($B_1(Y_1, \dots)$ @ T_1) which refer to user defined code, negative literals ($\text{not}(C_1(Z_1, \dots)$ @ T_2) and non-temporal goals (e.g. $X \geq Y + 2$) which use only built in routines.

Hamming Numbers

The program below computes the Hamming numbers (all multiples of 2, 3 and 5, that is numbers of the form $2^a 3^b 5^c$).

```
hamming@1.  
hamming@N :- N is I*2, hamming@I.  
hamming@N :- N is I*3, hamming@I.  
hamming@N :- N is I*5, hamming@I.  
print(hamming)@T :- hamming@T.
```

The representation of the Hamming numbers is that the predicate `hamming` is true at each time that is a Hamming number. The first clause says that 1 is a Hamming number and the next three say that given a Hamming number then multiplying it by 2, 3 or 5 will give another Hamming number. Notice also that the `print()` statement which outputs the result of the computation is the opposite way round to Prolog as it appears in the head not the body of a clause. When executed bottom up and in time order, this program will generate a successive listing of all the Hamming numbers. This idiom for output allows external actions to be effected within a pure logical paradigm.

Assignment

A critical claim of our approach is that it is capable of directly dealing with the mutation of data. In imperative languages this is dealt with by assignment. In functional languages some other mechanism such as monads is necessary [TCG93] Similarly, Mercury [SHC96] uses unique modes to ensure that mutable data structures are used in a single-threaded fashion. The next programming example shows how assignment can be programmed within Starlog. It is similar to the engine simulation, but allows tuples to be inserted and deleted from a relational database.

Consider a database that has two types of objects: keys and values. The keys are the letters x and y and the values are the letters a , b , and c . The database is functional. That is, at a particular time, each key has associated with it exactly one value. The database is implemented in Starlog using a relation `value(Key, Value)@T`. For example `value(x, a)@10` says that “the key x has the value a at time 10”.

To assign a new value to a key the relation `assign(Key, Value)@T` will be used. For example `assign(x, b)@10` says that “from time 10 onward the key x will have the value b independent of any value it had before”. Also if x had no previous value then a new one will be created. There is an arbitrary choice that needs to be made at this point. The assignment `assign(y, b)@3` could reasonably give x its value at time 3 and onward or at time 4 and onward. The convention used here is that the assignment will not take effect till the next time. That is, y has the value b from time 4 onward.

<code>assign(x,a)@1</code>		<code>value(x,a)@T ← T≥2, T≤5.</code>
<code>assign(y,b)@3</code>		<code>value(y,b)@T ← T≥4.</code>
<code>assign(x,c)@5</code>		<code>value(x,c)@T ← T≥6.</code>

Figure 1. Example of Assignment in a Relational Database.

Figure 1 shows the resulting `value()` relation for a given `assign()` relation. Assignments are made to `x` and `y` at the times 1, 3 and 5. This leads to the key `x` having the value `a` from time 2 to 5 inclusive and then a value `c` from time 6 onward. The key `y` has no value until time 4 and from there onward has the value `b`.

The problem now is to construct a program that uses the `assign` relation to generate the `value` relation. The program is constructed in two steps. The first step defines `value` as being true from just after an `assign` statement until that key is deleted (via an auxiliary `delete()` relation). In this `assign()` acts like the `start` relation in the initial example and `delete` like the `stop` relation. The rule for `value` is:

`value(K,V)@T :- assign(K,V)@T0, T>T0, not(delete(K,T0)@T).`

That is, “a key `K` has a value `V` from the time it is assigned until `delete` is true for the same key”.

To capture this stopping condition we need a rule for `delete(K,T0)@T` which means “at time `T` there has been an earlier assignment to the key `K` which occurred after time `T0`”. In the form of a rule this is:

`delete(K,T0)@T :- assign(K,_)@T1, T>T1, T1>T0.`

Debugging

In the introduction we alluded to the advantages of a relational style of programming which avoids the need to navigate complex data structures to access data. This observation leads to an open programming style. An excellent illustration of this openness occurs in debugging. Normally, to display the internal execution of a program it is necessary either to use an elaborate run time debugging system or to sprinkle the code with strategically placed print statements. In our system, because accessing a relation is trivial, it is easy to write a program that prints all the internal results of execution:

`print(X)@T :- X@T, X≠print(_).`

As each fact `X@T` is generated it is first checked to make sure it is not itself a `print` (otherwise there would be an infinite regress of `print()` facts generated) and then is immediately printed. This phenomenon can be extended to more elaborate forms of this debug/printing program. For example, by user input that decides which relations are to be displayed. Even more powerful is a form that allows the values of relations in the past to be queried as a result of input by the user:

`print(X@S)@T :- input(query(X@S))@T, X@S.`

For example if at time 10 the user were to input `query(value(a,Z)@3)` the result would be to print the value of the key “a” at the earlier time 3. (We comment on the implementation implications of this program below).

The open programming approach can be extended further to include integrity checks on execution. For example the following code checks that the `value` relation is in fact functional on the time and key:

```

print(["value not functional for key:",K])@T :-
    value(K,V1)@T, value(K,V2)@T, V1≠V2.

```

The importance of these examples lies in the fact that they are succinct and that the original code did not have to be modified to include them. That is the clauses act like daemons monitoring execution and taking action only when particular conditions are observed. It is particularly gratifying that this is possible within a pure logic language without any need for special language constructs.

Semantics

The first important point to note about our approach is that all the programs are pure logic programs; no extra-logical constructs are added to the language at all. The convention of using @ followed by a variable which provides a temporal ordering is a purely syntactic convention which accords no special semantic status to the temporal variable.

Negation

As illustrated in the examples above negation is essential if interesting programs are to be written. As a result it is important that it be accounted for in any semantics for the language. The approach we have taken is two fold. First we use a three-valued Kripke–Kleene semantics that gives a three-valued least model for any program. We then treat any program that generates undefined values as being in error. The practical side of this is the need to check for “undefined” errors either statically at compile time or dynamically at run time.

Static checking is most easily done by ensuring that a program is stratified. That is, there is a provable ordering on the Herbrand Universe. In Starlog this is partially ensured by the requirement that the time in the head of every clause be no less than the times in the body literals. However, this still allows programs such as:

```
p@0 :- not(p@0).
```

which leave $p@0$ undefined. To deal with this the ordering requirement on clauses is strengthened to require every head to be strictly greater than the body literals. To make this practical, a static stratification on the names of the relations within one time value is allowed. The hamming program is thus stratified in this strong sense because firstly it can be proven that the multiples $2 * I$, $3 * I$, and $5 * I$ are all strictly greater than I (given the lemma that $I \geq 1$) and secondly because `print()` can be statically stratified later than `hamming`.

In the `assign` and `delete` relations a proof of the ordering is easier because it need only take account of the timestamps. In practice, we have written a simple tool that given a Starlog program will check for static orderings of the relations and report if it cannot find any. In general this procedure is undecidable, but some simple knowledge of arithmetic and inequalities (for example that $T+1 > T$, or that $I \geq 1$ implies $2 * I > I$) suffices for most programs we have encountered. In difficult cases the programmer can easily add redundant relationships to make the ordering explicit.

A different approach could be taken to the semantics by, say, using the well-founded semantics. However, what we are attempting here is to produce a precise and simple programming language not a powerful deductive system. Thus we have eschewed the power and complexity of the well-founded semantics for the weaker and simpler approach above.

Operational Semantics

The semantic approach we have used above could be equally applied to Prolog as well as Starlog. Indeed Starlog programs, given a suitable goal, could be executed by a Prolog system. However, this would be very inefficient. The intent is that bottom-up execution with concomitant tabling be used [RSS92, Wun96]. To capture this intent it is necessary to supply some form of operational semantics. Details of such a semantics are given in [LC99].

The difficulties of such a semantics can be seen from the following `even@T` program. The simplest way to generate all the even numbers would be to say that 0 is an even number and that if a number is even then that number plus 2 is also even. However, there is another

formulation that says that every positive number is even if the immediately preceding number is not even. Written as a Starlog clause this is:

```
even@T :- T is S+1, not(even@S), T≥0.
```

Normally bottom-up execution proceeds by taking known facts, deducing consequences from them and then using these newly deduced facts and so on. However, this even program contains no facts, just a single clause with a negation in the body. The way that this stalemate is broken is to use the temporal ordering. First the earliest time that any relation can succeed is computed (to be more precise, all the literals in the bodies except the calls to builtin functions are deleted and the earliest time that any of the resulting clauses could succeed is computed). Then the negations in the program are rewritten using constructive negation. For example, in the first step of the even program, the earliest possible success time for even is 0. The negation `not(even@S)` is then rewritten as `not(even@S, S≥0)`. This can then be further rewritten using constructive negation [Cle97] as `((S<0) or (S≥0, not(even@S)))`. The result of this is two new clauses:

```
even@T :- T is S+1, S<0, T≥0.
```

```
even@T :- T is S+1, S≥0, not(even@S), T≥0.
```

After simplification of the inequalities this gives one fact:

```
even@0.
```

and a new program clause:

```
even@T :- T is S+1, not(even@S), T≥1.
```

Thus a fact has been deduced from a single program clause without any prior facts. The operational semantics then consists of repeatedly performing two phases. In the first phase, the earliest success times are computed and constructive negation is used to deduce any new facts. Then the second phase propagates all newly generated facts in a standard bottom-up fashion.

It has been shown [*ibid*] that this procedure always guarantees progress given a stratified program and some weak assumptions on the decidability of the builtin functions.

Implementation

With any pure approach to programming the suspicion arises that it may not be easy to implement efficiently. In this section, we will argue strongly that the opposite is true. That in fact, the purity and data-structure-free style of programming allow efficient generated code of a type not achievable in any extant languages. This section first outlines some of the implementation problems that arise and their solutions. Then it describes work in progress that aims to compile the neutral relational style of Starlog into explicit and efficient data structures.

One approach to implementing Starlog is to execute it bottom up and to maintain a table of all the facts generated for each relation. The execution of a clause then becomes a search through the tabled facts followed by insertion of the newly generated facts in the tables. This is inefficient for two main reasons. Firstly, it does not provide accurate information on when clauses should be executed. Secondly efficient searching of the tables will require the construction of indices that give efficient access. However, this can potentially be expensive as the more complex and precise the indices are the more expensive they are to construct.

Triggers

The cost can be reduced by using triggers to fire clauses[SU99]. Consider a rule such as:

```
p(X)@T :- q(X)@T, ....
```

Execution of the clause can wait until a fact for `q(X)@T` is generated and then fire the execution of the clause. One example of this is the `print()` clause in the Hamming program. Whenever a new hamming fact is generated the clause can be fired and the print

statement executed. Given that the hamming results are generated in time order the `print()` execution can be done with essentially zero scheduling overhead. This approach can be taken even further for `hamming`. Assume that the hamming facts are placed in a single sequential list. At each iteration of execution the smallest remaining Hamming number is selected from the list and then all four clauses are fired, one to do the `print` and three to insert new `hamming` facts into the list.

Such triggering cannot deal with all cases. Consider an elaboration of the clause above:

$$p(X)@T \text{ :- } q(X)@T, r(X)@T, \dots$$

If $r(X)@T$ is guaranteed to be **after** $q(X)@T$ in the temporal ordering then a two step trigger can be used. That is, the clause is first triggered on $q(X)@T$ then a suspension is recorded for this case which is triggered when a new $r(X)@T$ fact is deduced. If both the facts for $q(X)@T$ and $r(X)@T$ have bound values for X then it is necessary to provide some form of indexing for the stored suspension of the clause. This is a converse strategy, rather than storing and indexing the facts generated by $r(X)@T$ the suspensions of the clause generated by $q(X)@T$ are stored and indexed. Which of the two strategies will be the more efficient will depend on many other details of the program.

If $r(X)@T$ is guaranteed to be **before** $q(X)@T$ then it is necessary to store the values of $r(X)@T$ in a table and to search this after the clause is triggered on $q(X)@T$. If there is no provable relationship between the ordering of $r(X)@T$ and $q(X)@T$ then it may be necessary to store and index both of them. Because Starlog is a pure logic language both the programmer and compiler are free to do an efficiency trade-off by determining which facts or suspensions will be tabled and indexed and which will be treated as triggers.

One principle that emerges from this discussion is that the way that a relation is used is as important as the way that it is generated for determining how it should be represented internally. Most languages require that the implementer of code predetermine one representation, even though the optimal choice will be determined by its use. Leaving the choice as late as possible is the appropriate way to get good performance. This approach is not without its problems. For example, it makes modular independent compilation of code very difficult and implies that all the source for a program must be present at compilation time. Also the insertion of a single clause such as the universal `print/debug` program described earlier can force all the relations in a program to be tabled and consume memory for ever.

Constraints

The results from the assignment program earlier include facts such as:

$$\text{value}(x,a)@T \text{ :- } 2 \leq T, T \leq 5.$$

which says that the key x has the value a from 2 to 5. This could be represented inefficiently as a sequence of facts `value(x,a)@2`, `value(x,a)@3`, `value(x,a)@4`, `value(x,a)@5`. Such a representation would totally defeat the purpose of the language as a program which had a small number of assignments with long (possibly infinite) time gaps between would need to generate a new fact for each time step. The solution to this is to allow facts to contain constraints such as $2 \leq T$. This approach ensures that the cost of assignments is proportional only to the number of assignments not to the time gap between them.

Current Starlog implementations allow general and powerful arithmetic constraints [Cle87]. While these fix some large-scale problems they can be expensive to maintain and to propagate through the bottom-up deduction process. This is particularly true when it is necessary to perform constructive negation on such constraints. The results in [Cle97] provide an effective but relatively expensive algorithm for doing this. To alleviate these costs work is in progress aimed at performing a static type analysis of Starlog programs that is capable of deducing the forms of the possible constraints at different parts of the program. If successful this would allow compile time generation of code for relaxing constraints and would remove any need for dynamic execution of constructive negation.

Selection of Data Structures

Crucial to the efficient execution of Starlog programs is the selection of indices, that is data structures, for relations and suspensions. We believe that deciding which data structure is most suitable can be automated within a compiler. To support this claim, this section will analyse assignment and the Hamming number program.

Hamming Numbers

The first basic observation about the Hamming program is that it can be treated as an event list. On each cycle of execution the earliest time is selected from the event list and three new ground integer values are inserted into the event list. This analysis requires only recognition that the new results are generated as ground integers and that there are no other parameters in the `hamming` relation that might need indexing. The data structure problem then requires selection of an appropriate event list structure. Generic solutions include a calendar queue or balanced binary trees such as AVL tree or splay tree.

How the Hamming numbers are used also has an effect on the data structure. As the program is given, with only a print statement appended, the event list description suffices. If, say, a debug statement that allowed selection of earlier time values was included then it would also be necessary to retain all values on the event list so that it grew continually and the lowest selected values were not discarded. It might also be necessary to provide an index on the time value so that arbitrary values could be selected. This would force a more general and much less optimised structure than the ones described below.

There are two data structures however, that are particularly efficient. The first of these contains three distinct linked lists, each with a pointer to the root and the end elements:

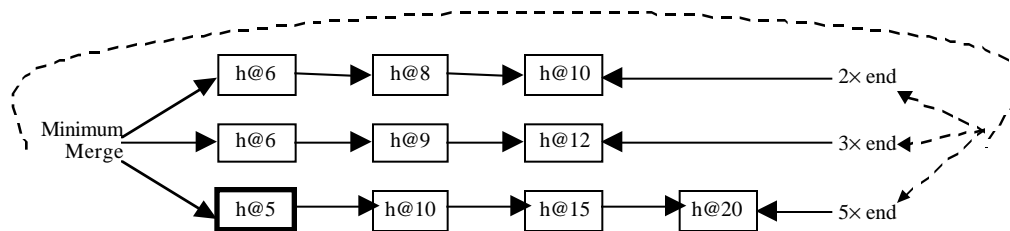


Figure 2. Triple List data structure at time 5.

Each of these lists corresponds to one of the rules in the Hamming program that generates a new Hamming number. Consequently, these are referred to as the $2\times$, $3\times$ and $5\times$ lists. When the program is running, the resulting fact generated from each of these rules is inserted at the end of their respective list. Because these facts are generated in order, adding these to the end of the lists will result in an ordered list with no comparisons required. The minimum time value on each step is found by merging the three lists. It is possible for duplicate facts to exist across the three lists. As a result, it is sometimes necessary to consume more than one root member if a duplicate is found in the merge process. Depending on the implementation of this data structure, it is possible to make this a very efficient representation of the Hamming program data. For example, depending on the ordering of operations the lists may never become empty and so no null pointer checks are required. One implementation of this revealed that no comparisons were required for the insert operation and only two comparisons were required per merge operation to determine the minimum fact. Therefore a total of two comparisons are required per program cycle.

This representation requires the compiler to understand the following about the program:

- that the results are generated by precisely three different rules. This follows from the structure of the program and the fact that none of the rules fails at any point;
- that the generated numbers within each rule are monotonic (that is, $I > J > 0$ implies $2 * I > 2 * J$, etc.);

- that each new hamming value generates at least one new value for each rule (that is, $2 \times I$ exists for every value of I). This condition ensures that the individual lists never become empty.
- that the event list is never empty. This follows from the observation that 1 is inserted initially and thereafter a new value is inserted in each list at every time step. This observation allows the omission of checks for an empty event list.

Any duplicate facts that are generated are removed at the merge step.

This data structure can be further refined to a linked list with a single root that has three pointers describing where the next outputs of the $2 \times$, $3 \times$ and $5 \times$ rules will be inserted.

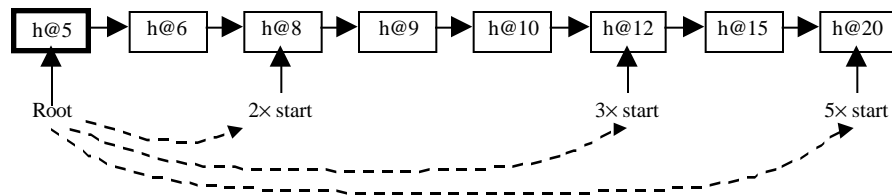


Figure 3. Multiple Insert Points List at time 5.

When a new element is to be inserted, the pointer corresponding to the rule that generated it (eg. the $2 \times$ rule) is followed. The fact is inserted using a linear search forward from that point. Since there is only a single ordered list, finding the next fact to be consumed is trivial since the minimum timestamp will always belong to the root node. Because the insert pointers are updated on every insert operation, this process is efficient due to the low number of comparisons required. Also, the insert operation can be adapted to remove duplicate tuples as they are generated so that these will not take up space unlike the first data structure.

If the compiler can deduce that $5 \times I > 3 \times I > 2 \times I$ then the code can be refined so that the $2 \times$ and $3 \times$ pointers never have to worry about testing for the end of list and the $5 \times$ case knows that it is always at the end of the list.

The second version works well because only a small number of subsequent elements need be skipped on each insertion (this relies only on the fact that the individual clauses generate new facts in order). Also it consumes less space than the first as duplicates are removed as part of the insertion process not later as part of the merge. The conditions for its use are weak and we expect that this general form of merged list will apply to a wide range of programs.

We expect that in general there will be multiple potential data structures for a program and that it will need deeper expertise to select between them. So a programmer might be given the option of choosing which to use or performance data from test runs might be used to make the choice.

Assignment

The ideal situation with the assign/value program would be to compile it to use in place destructive assignment. This programming idiom is very prevalent in Starlog programs and so it is worth investing significant effort in detecting such cases. There are two aspects to this the index structure needed for the key in `value()` and the index structure needed for the time value.

In the example as given there are only two keys used: “x” and “y”. A compiler should be able to recognise this and create two unique global objects corresponding to these cases. Of course the general case could be much more complex. It is important for the compiler to deduce that the associated values are functional. That is, given a specific time and key, there is a single associated value. This follows from the logic of the program, although the analysis is subtle and requires recognising that the negation of `delete` forces failure at the next assignment. Other than this, the analysis required is just a type and mode analysis to determine which possible values are used as indices and their binding state at the point where `value()` is called as a body literal.

Clearly the usage of `value()` is important here and, as in Hamming, if it is possible to inquire about prior values then it will be necessary to maintain a persistent record of all the values for each key. Also, it may be necessary to deal with the special case where the value for a particular key does not exist. If this is possible then it may be necessary to maintain a valid flag with each key. In some cases it will be possible to deduce that a key is only read if the `value()` exists and the flag can be dispensed with.

Finally, to compile each assignment to a destructive overwrite of memory requires the following conditions to be met for all usages of `value()` as a call. It is sufficient that it occur in a fragment with the following form... $q(\dots)@T, \text{value}(K,V)@T$, .and that:

- T is bound to a ground integer after the $q(\dots)@T$ call;
- assign is statically stratified earlier than $q(\dots)@T$;

Summary

A pure logic programming language has been proposed that uses ordering on a “temporal” variable to sequence computation. The purity of the language leads to simple and standard semantics including a straightforward operational semantics. The remainder of the paper has described work in progress that aims to compile the data-structure-neutral relational style of the programming language to highly tuned and efficient data structures. The choice of data structure depends on how relations are used, as well as on how they are generated. If this research can be carried through then we believe it will provide a significant and important new programming tool.

Acknowledgements

This work has been supported by grant UOW-605 of the New Zealand Marsden Fund. We would like to thank Don Smith and Lunjin Lu for many stimulating hours of discussion about logic programming.

References

- [Cle87] Cleary, J.G.(1987) “*Logical Arithmetic*,” *Future Computing Systems*, 2(2), pp. 125-149.
- [Cle97] Cleary, J.G., (1997). *Constructive Negation of Arithmetic Constraints Using Data-Flow Graphs*. *Constraints*, 2: pp. 131-162.
- [LC99] Lunjin Lu, and Cleary, J.G., (1999). “*An Operational Semantics of Starlog*”,. *Proc. Principles and Practice of Declarative Programming 1999*, Paris, pp. 131-162. Springer-Verlag.
- [OM94] Orgun, Mehmet A. and Ma, Wanli (1994). “*An overview of temporal and modal logic programming*”. In Gabbay, D. M. and Ohlbach, H. J., editors, *Temporal Logic: First International Conference*. Springer-Verlag, LNCS 827.
- [RSS92] Ramakrishnan, Raghu, Srivastava, Divesh, and Sudarshan, S. (1992). “*Efficient Bottom-up Evaluation of Logic Programs*”. In Vandewalle, J., editor, *The State of the Art in Computer Systems and Software Engineering*. Kluwer Academic.
- [SHC96] Zoltan Somogyi, Fergus Henderson and Thomas Conway (1996). “*The execution algorithm of Mercury: an efficient purely declarative logic programming language*”. *Journal of Logic Programming*, volume 29, number 1-3, pages 17-64.
- [SU99] Donald A. Smith and Mark Utting (1999). “*Pseudo-Naive Evaluation*”. In *Tenth Australasian Database Conference, ADC'99*, Auckland, New Zealand, Jan 1999, Springer-Verlag.
- [TCG93] Tansel, A., Clifford, J, Gadia, S., Jajodia, S., Segev, A., and R. Snodgrass, editors (1993). “*Temporal Deductive Databases: Theory, Design, and Implementation*”. Benjamin/Cummings.
- [Wad95] Philip Wadler (1995). “*Monads for Functional Programming*”. In *Advanced Functional Programming*, LNCS 925. Springer-Verlag.
- [Wun96] Wunderwald, J. (1996). “*Adding Bottom-up Evaluation to Prolog*”. PhD thesis, Technical University of Munich.

