Optimising Tabling Structures for Bottom-Up Logic Programming

Roger Clayton, John G. Cleary, Bernhard Pfahringer, and Mark Utting

Department of Computer Science
University of Waikato
Hamilton, New Zealand
{rjc4, jcleary, bernhard, marku}@cs.waikato.ac.nz

Abstract. In this paper we show how efficient data structures can be automatically introduced into programs that are written using flat database like relations. We describe a compilation process for Starlog, which is a pure logic programming language with temporal features that allow updates to be supported cleanly. Starlog programs are translated into an intermediate language (SDSL), which is optimised, and then analysed to determine which data structures are best for each relation. Then Java code is produced, using a library of possible data structures with known speed characteristics. The main benefit of this approach is that different data structures can be introduced without changing the source program. This makes programs more flexible and easily maintained.

1 Introduction

The art of programming is popularly described as the combining of program control with data structures [Wir76]. In logic programming this model has been refined to logic plus control [Kow79]. From this refinement one would assume that logic programs are freed from specifying how data is stored and manipulated. But by looking at most useful logic programs it can be seen that this is simply not true.

Most Prolog programs use lists and other functor-based data structures extensively, or import a data structure package from a library. A significant part of the program design effort goes into choosing the most appropriate data structures and implementing them. The rest of the program code becomes highly dependent upon those data structures. Changing to a different data structure requires large-scale rewriting of the program. The ultimate and ironic result is that Prolog programs are full of data structure manipulation code which obscures the original problem.

This heavy dependency upon data structures is perhaps even more pronounced in pure logic programming languages like Mercury [SHC96], where assert and retract are not available, so the mutable state of the program must be passed around as a data structure. Logic programming today, as it has been for more than 20 years, is still as concerned with defining data manipulation as other language paradigms.

Another field concerned with the representation of data structures at a high level is relational databases. These do not require users to understand the underlying implementation of how their data is stored yet allow users to successfully interact with them. As far as a user is concerned the data is just 'there', with a consistent and uniform structure, and can be accessed or added to at any time simply by referring to the appropriate table name.

We can apply the same type of abstraction to logic programming. When a programmer wants to access program data, instead of searching through a list or some other data structure, it would be better to access it by its name (or type) and specify the relevant data properties (similar to the way static or asserted data is used in Prolog systems). As long as data records are never modified this remains logically sound, but once it becomes necessary to assert or retract rules, the simple semantics is compromised, and efficient indexing becomes more difficult.

By removing control of how the data is stored from the programmer one would assume that the efficiency of the program would be reduced. However this need not be the case. We argue that by analysing the operations performed on the data it is possible to (1) optimise these operations to remove redundancy and (2) to choose an efficient underlying implementation specific to the expected data set. In many cases allowing the compiler to decide data related issues will improve the efficiency while leaving programs more readable and amenable to change.

To support these ideas we present the logic programming language Starlog. Starlog programs provide a data-structure-free environment where all program data is tabled in a global database. Data structures are never explicitly used in Starlog since all data can be expressed as relations[CUC00]. Starlog programs are evaluated bottom-up using the fix point semantics discussed in [VEK76]. A temporal ordering is used to control the evaluation order of rules.

Starlog is similar to DataLog and related deductive databases. However, its focus is on the efficient execution of *programs* rather than storing and retrieving persistent data on external memory. Also the explicit representation of time values in Starlog differs from almost all relational and deductive databases. The concept of explicit time allows program rules to refer to any events from the past (expanding the range of possible programs from simulations). Alternatively, destructive updates are allowed by referring only to previous values [CUC00].

For this paper we limit the execution model to naive evaluation [SU99,RSS92] (albeit optimised for stratified programs). We do not address alternative evaluation techniques such as semi-naive or pseudo-naive evaluation.

In this paper we further describe Starlog and show how programs can be compiled to an intermediate language (SDSL). This intermediate language allows high level optimisation of the program. We describe how data is efficiently stored at runtime and show how it is possible to statically choose an efficient data structure implementation based on how the data is used by the program. To support our argument we will introduce an example program that will be compiled throughout this paper.

2 Related Work

The most well known relation indexing technique used in logic programming is first argument indexing that has been used extensively on the WAM [HA91]. Indexing on a single term is a simple, effective solution for small programs however it does not scale well. In addition, programs need to be written with this indexing in mind if they are to achieve optimal performance.

Although the XSB deductive database uses a top-down engine with tabling to achieve bottom-up evaluation [ASW94] its indexing requirements are similar to Starlog. XSB uses a variety of indexing techniques such as tries or hashing on three or less fields [RRS95,ASW94]. It also allows multiple indexes to the same predicates although, unlike Starlog, these must be distinct. XSB is able to index variables and complex terms which is currently not possible (or necessary) in Starlog.

The Aditi deductive database is more flexible. In this system the creator of relations can choose between four storage structures and can specify one or more keys [VRK90]. However it is unclear whether the order of keys can be specified or is important.

3 Example Program

To demonstrate the ideas presented in this paper we provide an example program that is an implementation of Conway's Game of Life [Gar70]. In the game of life a cell on an infinite two dimensional board lives to the next generation if it has either two or three neighbouring cells and a new cell is produced if there are three neighbouring cells around an empty location.

The example program given in Fig. 1 is written using a slightly simplified Starlog notation which should be familiar to Prolog programmers. The reader is reminded that this is a bottom-up program and should not be approached using top-down techniques due to the infinite regression that may result.

The initial state of the cells has not been included to leave the program as general as possible and so the effects of program optimisations on different starting states can be explored in later sections.

This is by no means the most efficient implementation of the game of life. We have other versions that employ builtin facilities to count neighbouring cells. However this implementation is compact, requires little knowledge of Starlog to understand and, above all, demonstrates how optimisations can improve the efficiency.

The execution process of this program will now be described. For a given starting state of cells at time T, this program generates all the neighbours of each cell using rule 3. Neighbours are distinguished from each other by the N parameter that lies between -4 and 4 and indicates the direction of the original cell. (N can not be equal to zero as this means the cell is its own neighbour). Each new neighbour tuple is inserted into a database at the current time T.

```
% Rule 1: New cell generated
cell(X,Y,T) <-- TPrev is T-1,
    neighbour(X,Y,N1,TPrev),
    neighbour(X,Y,N2,TPrev), N2 > N1,
    neighbour(X,Y,N3,TPrev), N3 > N2,
    not(neighbour(X,Y,N4,TPrev),
        N4 != N1, N4 != N2, N4 != N3).
% Rule 2: Cell survives to next generation
cell(X,Y,T) <-- TPrev is T-1,
    neighbour(X,Y,N1,TPrev),
    neighbour(X,Y,N2,TPrev), N2 > N1,
    not(neighbour(X,Y,N3,TPrev),
        N3 != N1, N3 != N2),
    cell(X,Y,TPrev).
% Rule 3: Adds 8 neighbours for each cell
neighbour(XNew,YNew,N,T) <-- cell(X,Y,T),
    near(DeltaX), near(DeltaY),
    N is (DeltaX*3)+DeltaY, N != 0,
    XNew is X+DeltaX, YNew is Y+DeltaY.
% Auxiliary Facts 1-3
near(-1).
near(0).
near(1).
```

Fig. 1. Starlog implementation of Conway's Game of Life

Rule 1 accesses the neighbour tuples database and finds all cases where there are exactly three neighbours for any X and Y location at time T-1. Rule 2 similarly accesses the database to find locations where there are exactly two neighbours for any X and Y location and checks if a cell already exists there. In both cases, a cell is generated at this X and Y location for time T. The time T is then advanced by 1 and the program repeats.

The order that Starlog rules are evaluated is crucial to the correct execution of programs that include negation. Using the stratified semantics discussed in [FBJ88] a partial stratification order is implicitly defined on the timestamps of tuples and their predicate names. The stratification order for the example program is:

$\forall T \; near(\_) < cell(\_,\_,T) < neighbour(\_,\_,\_,T)$.

That is, all near tuples are generated before any cell tuples, which must all be generated before any neighbour tuples for any given time T.

Under our naive evaluation model there exists a global current time variable. Each rule is considered individually in the stratification order of the rules' head predicates. For each rule the time variable in the head (in this paper the last parameter of each predicate) is bound to the global current time variable. In this way the search space of each rule is reduced and, by incrementing the current time variable after all rules have been processed, the program is guaranteed to make progress. The bodies of program rules are evaluated using a simple left to right *sideways information passing strategy* (SIPS) allowing the programmer some control of the order of queries. The effect of naive evaluation will be seen in later sections.

In the game of life the next generation of cells is only dependent on the current generation. In this and many other programs it is possible to delete (or garbage collect) tuples that can not contribute to the production of new tuples. Although we have implemented a tool that detects and discards tuples for some of the more obvious cases (such as this), garbage collection will not be considered in this paper due to limited space.

4 Indexes

Starlog programs must be transformed to a series of database operations. Before this can occur we need to define the properties of the database.

We need to know the indexes of each relation (predicate). These indexes determine the possible access paths for resolving each query of a given predicate. We require the programmer to specify one or more index paths for each predicate. For example, for the predicate $cell(X, Y, T)$, we might specify two index paths: $[T, X, Y]$, and $[T, Y, X]$. Then if a predicate contained the query $cell(X, Y, T)$, where T and X are known (ground) but Y is unbound, our query optimiser would use the first index, because it can index directly to the desired values of X and T, and then scan through all values of Y to obtain each solution to the query. [1].

We require each index path to be *unique*, which means that at most one tuple exists when all the variables in the index path are assigned values. In other words, for each index path, the fields in the predicate that are not in that index path must be functionally dependent on the fields that are in the index path. This is to ensure that tuples are not overwritten by others in the index. We are developing an automated tool to check this property for each index, similar to the determinism checker of Mercury [HSC96]. This is an undecidable problem in general, because a predicate can be generated by arbitrarily complex code, which can obscure the functional dependencies. However, the experience with Mercury shows that by restricting the source language slightly (e.g., requiring disjuncts to be written in a particular style), determinism checking can be made decidable, and that this is acceptable to programmers.

Just like in relational databases, the choice of indexes can have a major impact on the efficiency of the program – if there is no suitable index for a given query, an exhaustive search must be used, while if too many indexes are defined, there is extra overhead in updating them all each time a tuple is inserted or deleted. Although we currently require the programmer to design the indexes,

[1] A more sophisticated optimiser might take into account the actual distribution frequencies of each field, based on previous program runs or programmer hints, but our current optimiser just chooses an index with the longest prefix of ground values – $[T, X]$ in this case

an interesting research topic is to try inferring possible indexes from the program source code, based on the modes (groundness) of all the queries and the determinism information.

To allow further optimisations, we merge all the indexes in the program into a single *index tree*[2]. Each node of this tree contains a set of subtrees indexed by values (this set typically expands and shrinks as the program executes), and another fixed set of named *branches* to other subtrees or leaves. At this stage, we do not specify exactly what data structure is used at each node of the index tree. For example, we could use a list, a hash table, a binary tree etc. The choice between these depends upon how the data at that node is used, and is discussed further in Section 8.

We can now be more precise about how each index is specified. An index is a sequence of *branch names* and *index nodes*, followed by a set of *named leaves*. Each index node corresponds to one parameter of the predicate being indexed, so carries the type and range information of that parameter. Similarly, each named leaf carries the type and range information of a non-indexed parameter. Sometimes we need to add an extra boolean leaf to reflect the existence of a predicate.

In our life example, the indexes are:

```
index cell(X,Y,Time)         [Time,X,Y,?]
index neighbour(X,Y,N,Time)  [Time,X,Y,N]
index near(P)                [branch('near'),P]
```

Each index becomes a path in the global index tree. Indexes with common prefixes are merged up to the first distinct branch name. So the $[Time, X, Y]$ prefixes of the `cell` and `neighbour` indexes above are merged, then the `?` at the end of the cell index generates a boolean leaf called 'cell', while the `neighbour` index carries on to the next level of the tree (N). This ability to share indexes can improve efficiency, by allowing related facts to be clustered together. If sharing is not wanted, a unique branch name at the beginning of each index prevents it. Note that it is an error to merge two index nodes that contain distinct types of data (e.g., integer and string).

Fig. 2 shows the final index tree for our life example. The thin lines represent the named branches/leaves, while the thick lines represent (a set of) index values, each leading to a different subtree node. For example, the top node of the tree has a single named branch (`'near'`), plus zero or more $Time$ values, each leading to a different subtree of (X, Y, N etc.) values.

This index structure has several advantages. Cells and neighbours share similar storage and usage properties for the Time, X and Y parameters. These have been collapsed together to reduce space and, as will be seen later, to aid with optimising.

The boolean leaf called `'cell'` is necessary, so that `cell` tuples can be distinguished from `neighbour` tuples with the same $Time, X, Y$ values. However,

[2] In fact, branches may rejoin to form a graph. However, this paper considers trees only.

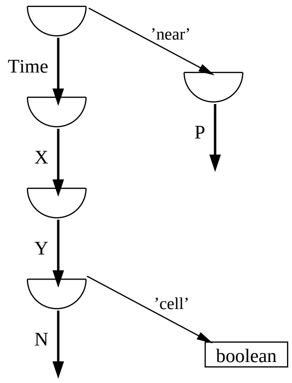

Fig. 2. Index tree for Game of Life example

these boolean flags are not necessary for the `neighbour` and `near` predicates, because their indexes end in a unique part of the tree.

The ability to have branch names makes it possible to provide multiple indexes to the same data (or tuples) when necessary. This can be used to optimise different types of searches on the same predicate such as when different variables are instantiated. The different paths may converge before they reach the end to reduce space (giving a graph rather than a tree). Of course when data is added down one path it must be added to all paths to maintain integrity.

5 Starlog Data Structure Language

Starlog programs can be represented as a series of database operations. The body of any Starlog program rule can be thought of as a set of database queries while the head is a database insert operation. To support this representation we have created an intermediate language called the Starlog Data Structure Language (SDSL) that consists of database operations and allows easy analysis and optimisation of programs.

In SDSL programs there is a distinction made between index variables and program variables. Program variables are variables from the original Starlog source code. Index variables can be thought of as pointers to index nodes within the global index structure. The first index that must be visited before any other is always the Root. This is a global index variable. All other index variables must be created and assigned by an SDSL operation. To avoid confusion index variables are always prefixed by a '$' symbol.

Operations supported by SDSL are given in table 1. Each operation (other than `builtin`) is performed on an input index. There are two types of operations: those to access or update the input index, and those that access or update elements associated with labeled branches. Program variables can be bound before the operation is encountered (in the case of a `look`, `insert`, `delete` or `set`) or will be bound by the operation itself (in the case of a `scan`, `minimum` or `get`). Mode inference will have indicated which type of operation is required. In most cases SDSL operations output a child index.

Table 1. SDSL operations list.

Operation	Supported Determinism
look(+<IndexVar>,+<ProgramVar>,–<ChildIndexVar>)	det, semidet
Finds child index associated with program variable in the input index.	
scan(+<IndexVar>,–<ProgramVar>:<Type>,–<ChildIndexVar>)	det, semidet, nondet
Finds all child indexes and the associated program variables and outputs them one at a time.	
insert(+<IndexVar>,+<ProgramVar>,–<ChildIndexVar>)	det
Inserts a value into input index if it does not already exist. Returns the potentially new child index.	
delete(+<IndexVar>,+<ProgramVar>)	det,semidet
Deletes a value from an input index if it exists. Used for garbage collection.	
minimum(+<IndexVar>,–<ProgramVar>,–<ChildIndexVar>)	det,semidet
Finds the minimum value in a input index. Used for pseudo-naive evaluation.	
get(+<IndexVar>,+<StaticLabel>,–<ProgramVar>:<Type>)	det,semidet
Returns value associated with the static label at the input index.	
set(+<IndexVar>,+<StaticLabel>,+<ProgramVar>)	det
Sets the value associated with the static label at the input index.	
branch(+<IndexVar>,+<StaticLabel>,–<ChildIndexVar>)	det
Creates a new child index associated with the static label if one does not already exist.	
trybranch(+<IndexVar>,+<StaticLabel>,–<ChildIndexVar>)	det,semidet
Outputs the child index associated with the static label at the input index.	
builtin(+<OpName>,+[<InputArg>,...],–[<OutputArg>:<Type>,...])	det,semidet
Performs a builtin operation, such as +,-,*,/,=,<,=<.	

The process of translating a program rule into SDSL will now be discussed. Each body member in a rule is considered a query performed in an order according to the SIPS. The type (functor/arity) and mode of a query can be used to find the best index path through the database structure. Working through this path, a `look` operation is used when looking for a known (ground) value in an index node. A `scan` is used when looking for an unbound value in an index node. If the next member in the path is a static label then a `trybranch` operation is used. Finally a `get` instruction is used when looking for a particular leaf value that may exist at the end of an index path.

When all queries have been translated into SDSL then the rule's head must be asserted. For each member of the path (or paths) associated with the head predicate an `insert` operation is performed when an index node is reached, the `branch` operation is used to follow static labels and `set` operations update any leaf values.

During this process all new index variables are created with unique names to avoid clashes. All builtin Starlog operations are translated into their SDSL equivalent. An example of SDSL translation of a body goal is given in Fig. 3.

Given:
`index` $p(x_1, ..., x_n, y_1, ..., y_m)$ [x_1,`branch('p')`,$x_2, ..., x_n, y_1, ..., y_m$]

A body goal of type $p(+x_1, ..., +x_n, -y_1, ..., -y_m)$ (assuming $x_1 ... x_n$ are ground) translates to:

```
look:semidet($Root,x1,$IndexA),
trybranch:det($IndexA,'p',$IndexB),
look:semidet($IndexB,x2,$IndexC),... look:semidet($IndexJ,xn,$IndexK),
scan:nondet($IndexK,y1,$IndexL),... scan:nondet($IndexV,ym,$IndexW)
```

Fig. 3. Example translation of a body goal to SDSL.

By default the determinism of each SDSL operation is as loose as possible. For example, all `scan`s are initially nondeterministic, all `look`s are semidet, while all `set` operations are only ever deterministic. By observing how variables are used in a program it is sometimes possible to narrow the determinism. The different values of variables instantiated by a `scan` operation that are not used elsewhere in the program and are the last element in the index path are not important - only the presence of at least one element. In this case a `scan` that was nondeterministic can be optimised to become semi-deterministic. There are more cases of determinism optimisation but these have been omitted due to space restrictions.

Each Starlog rule is translated into an SDSL *code block*, which is simply a sequence of SDSL operations with an implicit *fail* at the end. The semantics of the sequence is the usual Prolog-like backtracking semantics, so the fail forces exploration of all execution paths of the non-deterministic instructions. More precisely, the code block `{A,B,...}` is equivalent to the Prolog code:

```
( A,B,...,fail ; true )
```

To support negation there is another variety of code block. Negation code blocks have exactly the opposite effect – if there is a solution to all the instructions in the block, then the block fails, otherwise it succeeds.

6 SDSL Optimisations

As with most SDSL programs, the example program contains many redundant and repeated operations. The following subsections describe two types of optimisations and their effects on the example program. These are variations on traditional peephole optimisation and common subexpression elimination techniques. However the effectiveness of these optimisations is dependent on the design of the index tree. Well designed index trees which share index nodes between related predicates (such as in Fig. 2) will allow for significantly greater optimisation.

6.1 Optimisation Within Rules

As a result of the Starlog to SDSL transformation, similar operations can be performed over the same index in the same rule. Removing all repetitions of operations will make the program more efficient.

Operations do not have to be identical to optimise them. Most similar operations that are performed on the same input index can be optimised together if they generate the same output. There are many combinations of operations that can be optimised together given that they use the same input index variable. Any `scan`, `look`, or `insert` operation can match any other of these provided the program variable parameters are the same. All `get` or `set` operations can match if both the static labels and program variable parameters are identical. Likewise, any `branch` or `trybranch` operations can match each other when the static labels are identical.

When two (or more) SDSL operations are optimised together, all output index variables must be unified. This ensures the output index for the first operation is used rather than the output indexes for the removed operations.

The following rule shows how a `look` operation can be deleted when a `scan` has previously been done on the same variable. Note that $S \longrightarrow T$ arrow means that S is rewritten to T, while $S[\$E := \$C]$ means that $\$E$ is replaced by $\$C$ everywhere in S.

$$\frac{scan(\$A, B, \$C) \in S_1}{\{S_1, look(\$A, B, \$E), S_2\} \longrightarrow \{S_1, S_2\}[\$E := \$C]}$$

This rule, and other similar rules, are repeatedly applied until no further optimisations are possible.

6.2 Optimisation Across Rules

From a global perspective it can be seen that different code blocks in SDSL programs may have similar sets of operations. These cases can sometimes be optimised by merging the common code sections, thus reducing the total number of operations performed by the program.

To perform this optimisation at least two code blocks must be found that share a common prefix. An example of this can be seen in the optimised example program shown in Fig. 4. Rules 1 and 2 share the first eight operations. In this case the program variables and index variables are identical however it is not required. Matching variables can be unified together if necessary.

The 'across rule' optimisation ensures that the operations in the common prefixes are performed only once. This optimisation is described by the rule:

$$\{Prefix, A\}, \{Prefix, B\} \longrightarrow \{Prefix, \{A\}, \{B\}\}$$

Like the within-rule optimisation, optimisations across code blocks are performed repeatedly. The process involves comparing all code blocks with each other in the same scope to find two with a common prefix. These two are combined using the rules outlined above and the process is repeated until no further optimisations are possible.

There are some safety conditions regarding reordering stratified programs that must be addressed during this optimisation, however space limitations prevent discussing further details of these.

The result of applying the across-rule optimisation to the game of life example is shown on the right side of Fig. 4. From just a cursory examination the fully optimised program shows a significant reduction in operations over the original SDSL translation of the program in Fig. 1, which contained 73 operations. The final optimised version has only 44. The actual improvement in efficiency depends greatly on the initial state of the game of life since some arrangements of cells result in higher numbers of nondeterministic alternatives and therefore greater improvements. The improved performance for a selection of initial states (see [Sil02]) is given in table 2. On average the execution time of one program iteration has improved by more than 40%.

Starting State:	Blinker	Glider	Rabbit	Traffic Light	Bakery
Number of live cells	3	5	9	12	28
Unoptimised (μs)	275	407	726	777	2,223
Optimised (μs)	178	259	424	427	1,084

Table 2. Performance results of optimisation on one iteration of example program.

7 Compilation to Java

For SDSL programs to execute we need to translate them to an executable language. The chosen target language is Java. Java has several advantages over other languages such as platform independence and automatic garbage collection. The object oriented nature of Java also assists with delaying selection of data structures till after code is generated.

The Java implementation of naive evaluation contains a loop that iterates a current time variable from zero to infinity. Within each iteration all program rules (or SDSL code blocks as they are now) appear in stratification order of the rules' heads.

SDSL operations involving updating or accessing data structures are translated into data structure procedure calls. These procedure calls do not always correspond to those in the Java standard libraries. For example nondeterministic `scans` through all data values are not supported. Therefore we have defined our own library of data structures that have the necessary functionality. Knowing

```
{ % Rule 1
  builtin:det(assign,[currentTime],[T:int])
  builtin:det(subtract,[T,1],[TPrev:int])
  look:semidet($Root,TPrev,$1)
  scan:nondet($1,X:int,$2)
  scan:nondet($2,Y:int,$3)
  scan:nondet($3,N1:int,$4)
  scan:nondet($3,N2:int,$8)
  builtin:semidet(greaterThan,[N2,N1],[])
  scan:nondet($3,N3:int,$12)
  builtin:semidet(greaterThan,[N3,N2],[])
  not{scan:nondet($3,N4:int,$16)
      builtin:semidet(notEquals,[N4,N1],[])
      builtin:semidet(notEquals,[N4,N2],[])
      builtin:semidet(notEquals,[N4,N3],[])
  not}
  insert:det($Root,T,$17)
  insert:det($17,X,$18)
  insert:det($18,Y,$19)
  set:det($19,'cell',true)
}
{% Rule 2
  builtin:det(assign,[currentTime],[T:int])
  builtin:det(subtract,[T,1],[TPrev:int])
  look:semidet($Root,TPrev,$1)
  scan:nondet($1,X:int,$2)
  scan:nondet($2,Y:int,$3)
  scan:nondet($3,N1:int,$4)
  scan:nondet($3,N2:int,$8)
  builtin:semidet(greaterThan,[N2,N1],[])
  not{scan:nondet($3,N3:int,$12)
      builtin:semidet(notEquals,[N3,N1],[])
      builtin:semidet(notEquals,[N3,N2],[])
  not}
  get:det($3,'cell',Tmp:boolean)
  builtin:semidet(Equals,[Tmp,true],[])
  insert:det($Root,T,$16)
  insert:det($16,X,$17)
  insert:det($17,Y,$18)
  set:det($18,'cell',true)
}


{% Rule 3
  builtin:det(assign,[currentTime],[T:int])
  look:semidet($Root,T,$1)
  scan:nondet($1,X:int,$2)
  scan:nondet($2,Y:int,$3)
  get:det($3,'cell',Tmp1:boolean)
  builtin:semidet(equals,[Tmp1,true],[])
  name:det($Root,'near',$4)
  scan:nondet($4,DeltaX:int,$5)
  scan:nondet($4,DeltaY:int,$7)
  builtin:det(multiply,[DeltaX,3],[Tmp2:int])
  builtin:det(add,[Tmp2,DeltaY],[N:int])
  builtin:semidet(notEquals,[N,0],[])
  builtin:det(add,[X,DeltaX],[NewX:int])
  builtin:det(add,[Y,DeltaY],[NewY:int])
  insert:det($1,NewX,$9)
  insert:det($9,NewY,$10)
  insert:det($10,N,$11)
}
```

```
| {% Rules 1/2/3 optimized
|   builtin:det(assign,[currentTime],[T:int])
|   {  builtin:det(subtract,[T,1],[TPrev:int])
|      look:semidet($Root,TPrev,$1)
|      scan:nondet($1,X:int,$2)
|      scan:nondet($2,Y:int,$3)
|      scan:nondet($3,N1:int,$4)
|      scan:nondet($3,N2:int,$8)
|      builtin:semidet(greaterThan,[N2,N1],[])
|      { scan:nondet($3,N3:int,$12)  % Rule 1
|        builtin:semidet(greaterThan,[N3,N2],[])
|        not{scan:nondet($3,N4:int,$16)
|            builtin:semidet(notEquals,[N4,N1],[])
|            builtin:semidet(notEquals,[N4,N2],[])
|            builtin:semidet(notEquals,[N4,N3],[])
|        not}
|        insert:det($Root,T,$17)
|        insert:det($17,X,$18)
|        insert:det($18,Y,$19)
|        set:det($19,'cell',true)
|      }
|      {% Rule 2
|
|
|
|
|
|
|
|
|        not{scan:nondet($3,N3:int,$12)
|           builtin:semidet(notEquals,[N3,N1],[])
|           builtin:semidet(notEquals,[N3,N2],[])
|        not}
|        get:det($3,'cell',Tmp:boolean)
|        builtin:semidet(Equals,[Tmp,true],[])
|        insert:det($Root,T,$16)
|        insert:det($16,X,$17)
|        insert:det($17,Y,$18)
|        set:det($18,'cell',true)
|       }
|     }
|
|     {% Rule 3
|
|       look:semidet($Root,T,$1)
|       scan:nondet($1,X:int,$2)
|       scan:nondet($2,Y:int,$3)
|       get:det($3,'cell',Tmp1:boolean)
|       builtin:semidet(equals,[Tmp1,true],[])
|       name:det($Root,'near',$4)
|       scan:nondet($4,DeltaX:int,$5)
|       scan:nondet($4,DeltaY:int,$7)
|       builtin:det(multiply,[DeltaX,3],[Tmp2:int])
|       builtin:det(add,[Tmp2,DeltaY],[N:int])
|       builtin:semidet(notEquals,[N,0],[])
|       builtin:det(add,[X,DeltaX],[NewX:int])
|       builtin:det(add,[Y,DeltaY],[NewY:int])
|       insert:det($1,NewX,$9)
|       insert:det($9,NewY,$10)
|       insert:det($10,N,$11)
|     }
|   }
| }
```

Fig. 4. SDSL life program before and after across-rule optimisation.

which of these data structures to use at each node of the global index is not necessary yet. Since the selection of data structures is the last action performed by the compiler it will be discussed in the next section.

The Java implementation of most SDSL operations is straight forward. `Look` operations become data structure searches, `insert` operations add (potentially) new data. `Scan` operations however require more attention.

Scanning through a data structure nondeterministically is quite unnatural in imperative programming. The problem involves keeping track of where you currently are in the data structure (so you can efficiently resume the search during backtracking) and allowing for concurrent scans to operate on the same data structure. The solution we have used is to have an iterator class for every type of data structure. When multiple `scan`s are required then multiple iterators can be used independently.

The determinism of operations indicates the types of control structures to be used. For a semi-deterministic operation an 'if' statement is added to check if the output of any operation is valid and therefore succeeds. Any nondeterministic operation will contain a 'while' loop that iterates through all results before failing.

From the example program, the first six lines from the program in Fig. 4 have been compiled to Java in Fig. 5.

```
...
{
   final int T = currentTime;
   {
      final int TPrev = T-1;
      final Node node1 = nodeRoot.look(TPrev);
      if (node1 != null){
         final NodeIterator iterator1 = node1.getIterator();
         while(iterator1.hasNext()){
            final int X = iterator1.nextValue_int();
            final Node node2 = iterator1.nextNode();
            final NodeIterator iterator2 = node2.getIterator();
            while(iterator2.hasNext()){
               final int Y = iterator2.nextValue_int();
               final Node node3 = iterator2.nextNode();
...
```

Fig. 5. Example of Java implementation of first six lines of life program.

8 Selection of Data Structures

This section describes work in progress on ways to select the data structures used at each node in the tree. We have implemented a suite of different low level data structures that permit indexing on integers or strings. The results below use five of these data structures: flexible arrays, sorted lists, unsorted lists, red-black trees, and hash tables. All implement a common interface so it is possible to defer the selection of the concrete implementing class till run time (that is after the Java code has been generated).

We have investigated the following six ways that this selection can be done:

- the user can select (this is what is required by most programming languages and data-base systems);
- some generic data structure can be used (this is the approach of Prolog for its internal rule base);
- an adaptive data structure can be used that collects statistics about its usage and switches dynamically;
- a model of the performance of each type of data structure can be developed and used together with static or measured information about the program to predict the performance of each data structure at each node;
- the program can be run a number of different times and the performance of each selection at each position inferred by a regression analysis;
- all possible combinations of data structures can be run and the fastest selected.

We will now discuss the advantages and disadvantages of each approach in the light of some experiments that we have carried out. (These results are preliminary as they have only been assessed on a the single program described earlier).

User selection is of course what is usually done in programming. The Starlog system provides the advantage that it is easy to switch data structures without altering the logic of the program.

The most generic of the data structures available is the red-black tree. Using this binary tree at each node gives an execution time almost twice the best measured result (6,941ms versus 3,628ms).

We have implemented a data structures wrapper which keeps statistics about what methods are being called and the number of elements held. If it detects that it is consistently used in a sub-optimal way then it will switch to another more appropriate representation. This data structure incurs additional overhead for recording and analysing statistics and for copying data from one representation to another. We have found it difficult to write a robust implementation that does not spend more time collecting statistics than it saves by selecting the appropriate representation.

Table 3 shows a simple model of the execution time for different data structures. The constants were obtained by running each data structure in isolation and measuring the results. This model then needs to be combined with estimates for the number of times each operation is performed as well as estimates of various values such as N, $log_2(N)$ and $\frac{1}{N}$. These estimates can be obtained three ways:

- statically from the program source

Data Structure:	look	scan	insert	Prerequisites
Destructive Variable	10	150	10	1 data value (reserved use)
Array [L = length]	54	$150 + \frac{33L}{N}$	67	Integer data from 0-L
Flexible Array [M $\approx v_{max} - v_{min}$]	80	$150 + \frac{150M}{N}$	210	Integer data
Unsorted List	$25N + 75$	$110 + \frac{300}{N}$	$28N + 75$	none
Sorted List	$18N + 75$	$110 + \frac{300}{N}$	$36N + 75$	none
Balanced Tree	$40 \log_2(N+1) + 75$	$800 + \frac{800}{N}$	$200 \log_2(N+1) + 75$	none
Hash Table [S = tablesize]	$\frac{30N}{S} + 310$	$150 + \frac{66S}{N}$	$\frac{5N}{S} + 400$	none

Table 3. Cost of Data structure Operations. N = number of elements

- by running the program once to collect counts of the number of operations and estimates of the data distribution dependent values.
- by having the programmer supply estimates of the size of the data structures and number of operations.

Fig. 6 (a) and (b) show results from using a static analysis and an analysis using measured data respectively. Each figure plots the rank assigned by the technique against the measured execution time of the program for all 625 possible combinations of data structures. The fastest combination of data structures was a sorted list, flexible array, flexible array and a sorted list for indexes T, X, Y and N respectively. The slowest combination used an unsorted list, sorted list, red-black tree and a hash table.

The static analysis we used counts one call for each occurrence in the program. That is, it ignores all issues of repeated execution. Also default estimates for the data structure sizes of 100 were used. As might be expected this crude analysis does not give an accurate ranking but even so the selected data structure was within 17% of the best possible combination. It seems that such static estimates can avoid very bad combinations even if they cannot select the best ones.

One run was used to measure the actual counts and values for N, $log_2(N)$ and $\frac{1}{N}$. The resulting rankings are shown in Fig. 6 (b). They slightly improve on the static rankings and give a selected combination within 16% of the best and on average these model the actual results more accurately. Of course this was at the expense of having had to do a statistics collection run which also makes the result less reliable for other data patterns. Not shown here are the actual estimated times using this technique. Although the rankings tracked the results reasonably well the estimates were under by a factor of 2 at the low end and high by a factor of 4 at the largest execution times. It is unclear why this is so and we are further investigating (possible issues include the costs of Java's garbage collection and interference between the different data structures). It is also possible that the model used was not sophisticated enough. For example, the index on T is accessed only at the current time or current time minus 1. The performance of a sorted list is much better than indicated by the estimate which assumes items are accessed at random. To improve this will require communicating more detailed information from the earlier stages of compilation.

Fig. 6 (c) shows the results obtained from timing many different runs. The total time can then be represented as the sum of the time taken by each data structure at each node. There are 4 index nodes and five possible data structures at each. However, there are in fact only 16 = 4*(5-1) different linearly separable values which can be extracted plus a constant (because a data structure is always chosen for each index only the differences between different data structures are accessible not the absolute values). By carefully selecting which combinations to use only 16 measurement runs are essential (more can be used to improve accuracy). Performing linear regression using all 625 runs gives a very good fit against the actual values. The selected value is within 2% of the best value. In general this approach will require at least $N*(DS-1)$ measurement runs where N is the number of index nodes and DS is the number of different types of data structures.

The ultimate is to run all possible combinations. This will seldom be feasible as it requires DS^N measurement runs and even then the results are only strictly applicable to the particular data pattern used.

9 Conclusion

The process of compiling a bottom-up logic program has been discussed with emphasis on optimising the way data is accessed and stored. There are five main steps involved with this method of compilation: (1) defining a global index structure, (2) compiling bottom-up programs to use the global index (by way of the SDSL language), (3) optimising the SDSL program, (4) compiling to an executable form and, (5) selecting data structures for each node in the global index. In this paper we have addressed automating all these steps except the first. Although hints have been given on the type of analysis needed to define an efficient global index structure, we have no formal inference method yet and believe that this may dominate future research.

Steps 3 and 5 are the optimisation steps where performance improvement is possible. For step 3, we have shown that our SDSL transformations give around 40% speedup. For step 5, the best possible choice of data structures is 3.6 times faster than the worst choice. Our model-based choice algorithms obtained most of this speedup (static prediction chose a data structure that was within 17% of the

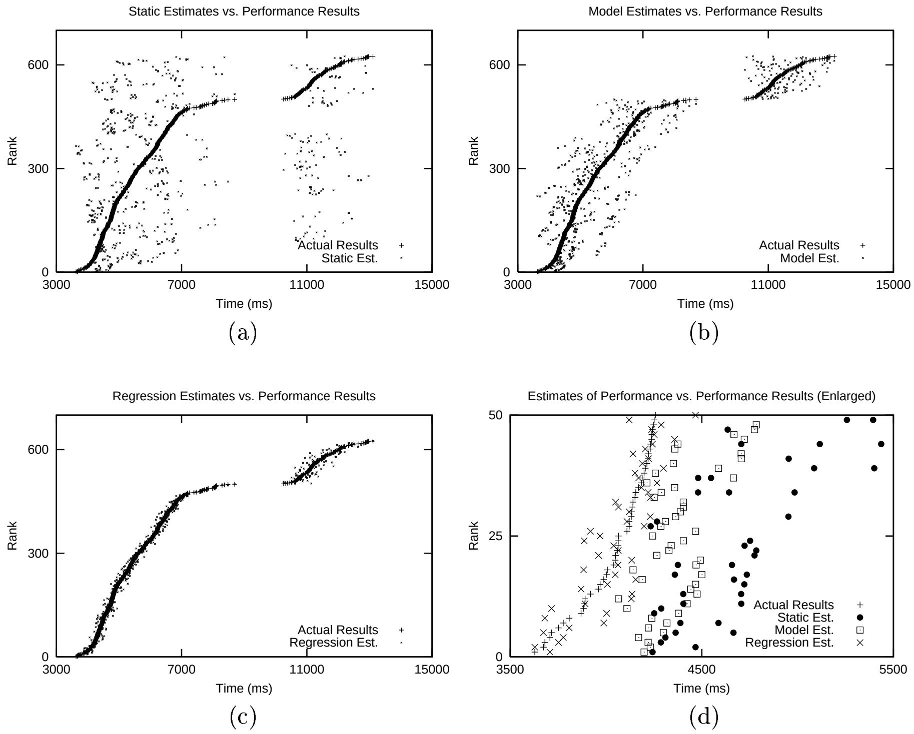

Fig. 6. Actual Performance compared with Static, Model and Regression Estimates.

fastest, while using runtime characteristics chose one within 16% of the fastest). However, these results are somewhat fragile (small changes to the program or the prediction data can give much worse choices), so we are continuing to improve our model to make it more robust. More research is needed to see how well the model generalises to a wider variety of programs and data sets.

An advantage of the approach that we have taken is that the binding of data to its representation can be deferred out of the program design stage until code has been written. Premature decisions on data representation can be costly to fix so we expect that the overall life cycle cost of code can be reduced. It is also possible to very easily change the representations used, significantly reducing the cost of maintaining and improving code.

References

[ASW94] K. Sagonas, T. Swift, and D. S. Warren. XSB as an efficient deductive database engine. In *R. T. Snodgrass and M. Winslett, editors, Proc. of the 1994 ACM SIGMOD Int. Conf. on Management of Data (SIGMOD'94)*, pages 442–453, 1994.

[CUC00] John G. Cleary, Mark Utting, and Roger J. Clayton. Data structures considered harmful. In *Australasian Workshop on Computational Logic, Canberra, Australia, Feb 2000.*

[FBJ88] Melvin Fitting and Marion Ben-Jacob. Stratified and Three-valued Logic Programming Semantics. *Proceedings of the 5th International Conference and Symposium on Logic Programming*, pages 1054-1069, August, 1988.

[Gar70] Martin Gardner. Mathematical games. *Scientific American*, 223(4):120–123, October 1970. Describes Conway's Game of Life.

[HA91] Hassan Ait-Kaci. Warren's Abstract Machine: A Tutorial Reconstruction. *MIT Press, Series in Logic Programming*, 1991.

[HSC96] Fergus Henderson, Zoltan Somogyi, and Thomas Conway. Determinism analysis in the mercury compiler. In *Proceedings of the Australian Computer Science Conference (ACSC), Melbourne, Australia, January 1996*, pages 337–346, 1996. Available from `http://www.cs.mu.oz.au/research/mercury/`.

[Kow79] Robert Kowalski. Algorithm = Logic + Control. *Communications of the ACM*, 22(7):424–436, July 1979.

[RRS95] I. V. Ramakrishnan, P. Rao, K. F. Sagonas, T. Swift, and D. S. Warren. Efficient tabling mechanisms for logic programs. In *L. Sterling, editor, Logic Programming - Proceedings of the Twelfth International Conference on Logic Programming*, pages 697–711, Massachusetts Institute of Technology, 1995. The MIT Press.

[RSS92] Raghu Ramakrishnan, Divesh Srivastava, and S. Sudarshan. Efficient Bottom-up Evaluation of Logic Programs. In *J. Vandewalle, editor, The State of the Art in Computer Systems and Software Engineering.* Kluwer Academic Publishers, 1992.

[SHC96] Zoltan Somogyi, Fergus Henderson, and Thomas Conway. The execution algorithm of Mercury: an efficient purely declarative logic programming language. *Journal of Logic Programming*, 29(1-3):17–64, October-December 1996.

[Sil02] Stephen A. Silver. Life lexicon. `http://www.argentum.freeserve.co.uk/lex.htm` Release 20, Feb 8, 2002.

[SU99] Donald A. Smith and Mark Utting. Pseudo-naive evaluation. In *Tenth Australasian Database Conference, ADC'99, Auckland, New Zealand, Jan 1999.* Springer-Verlag, 1999.

[VEK76] van Emden, M. H. and Kowalski R. A. The Semantics of Predicate Logic as a Programming Language. Journal of the ACM 23:733–742 (1976).

[VRK90] Jayen Vaghani, Kotagiri Ramamohanarao, David B. Kemp, Zoltan Somogyi, and Peter J. Stuckey. The Aditi Deductive Database System. In *Jan Chomicki, editor, Proceedings of the NACLP'90 Workshop on Deductive Databases.* Kansas State University Technical Report TR-CS-90-14, 1990.

[Wir76] N. Wirth. *Algorithms + Data Structures = Programs.* Prentice-Hall, Englewood Cliff, 1976. 1st edition.