Working Paper Series
ISSN 1177-777X

DATALOG AS A PARALLEL GENERAL PURPOSE PROGRAMMING LANGUAGE

John G. Cleary, Mark Utting and Roger Clayton

Working Paper: 06/2010
August 27, 2010

Department of Computer Science
The University of Waikato
Private Bag 3105
Hamilton, New Zealand

Datalog as a Parallel General Purpose Programming Language

John G. Cleary, Mark Utting and Roger Clayton

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

Abstract. The increasing available parallelism of computers demands new programming languages that make parallel programming dramatically easier and less error prone. It is proposed that datalog with negation and timestamps is a suitable basis for a general purpose programming language for sequential, parallel and distributed computers.
This paper develops a fully incremental bottom-up interpreter for datalog that supports a wide range of execution strategies, with trade-offs affecting efficiency, parallelism and control of resource usage. Examples show how the language can accept real-time external inputs and outputs, and mimic assignment, all without departing from its pure logical semantics.

1 Introduction

Combining logic and programming has a long history. Kowalski [Kow74,Kow79] and Colmeraur [CKR73] introduced Prolog in the early 1970s. Codd [Cod70] introduced relational databases at a similar time. Since then there has been much work in extending and refining these approaches in the form of more advanced and efficient Prolog-like languages and dataflow languages [JHM04]. Relational database theory has also been extended to include deductive databases [Liu99].

The overall agenda of these efforts has been to maintain the best of the logical and procedural worlds. The logical world seeks a declarative reading of programs, with simple semantics, so that they can be reasoned about easily. If this agenda is successful then it should be easy to prove programs correct [CU01,DM05], debug programs [Sha82,Sil08], and transform and modify them while maintaining correctness [PP96]. The procedural world seeks programs that are efficient, where the programmer can reason informally about and control resource usage, and where programs can interface with the existing computing milieux (perform I/O, call existing libraries, display graphics, *etc.*).

We perceive the current attempts to merge logic and programming to be incomplete. Most logic languages need to move outside their pure logical foundations in order to include facilities such as I/O and to enable efficient execution [CM03]. Kowalski [Kow01] argues that this is a major reason for the limited adoption of logic programming. He points out that in a multi-agent reactive

world, pure logic programming is best suited for just the *think* phase of the observation-thought-action cycle shown in Fig. 1. It is not good at observing changes to its input environment, or at performing update actions that change the real world.

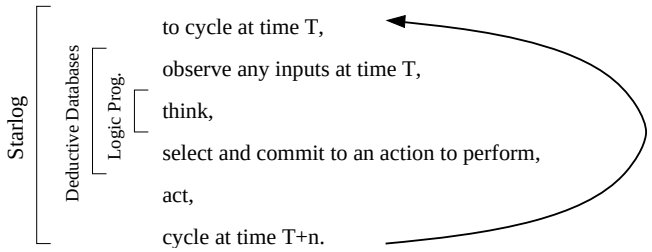

Fig. 1. The observation-thought-action cycle of multi-agent systems (Adapted from Kowalski, 2001)

While very successful and widely deployed, relational databases overlap with logic only for queries. Deductive databases and abductive logic programming extend this to include the observation and commit-to-an-action phases of Fig. 1, but it is still difficult to express the effects of the updates [Kow01].

Over the last decade, we have designed a language, called **Starlog**, that has a simple logical semantics, but is able to encompass traditionally difficult areas including input and output and mutation of the underlying database. The intention is that this be a general purpose language with (ultimately) good run time performance. We have developed several compilers for different subsets of the language, including one compiler that chooses data structures automatically and generates reasonably efficient sequential Java code whose execution speed is comparable to hand-coded Java programs [Cla04].

Recently, we have observed that this style of programming exposes a large amount of potential parallelism, and we have started developing a new compiler for parallel programming. The goal is to take advantage of the clean declarative semantics, and transform programs to run efficiently on various parallel architectures (many-core CPUs, cluster computers, GPUs, FPGAs, circuits etc.) with good scalability.

This paper describes the semantics of Starlog – that is, a logic programming language that has the following key features:

relational data: all data is stored in flat relations, as in the relational database model, rather than using lists or more complex data structures as is common in Prolog programs. This makes it easier to distribute data for parallel computation, and makes it possible to defer the choice of underlying data structures to the compiler [Cla04].

causality: a causality ordering is defined between tuples, to indicate which tuples depend on other tuples, and which are independent. We use causality

to ensure that negation is sound, to control the evaluation order of the program, to determine which computations can be performed in parallel, and to enable the logical effect of assignment and mutation.

temporal view: the causality ordering gives a temporal view of the data, which enables programs to observe time-varying inputs, react to those inputs by generating actions that update the external world, and to see the effects of those actions as new inputs arrive.

Section 2 defines syntax and terminology, then Section 3 makes a link to previous work on dependency graphs and Section 4 discusses some small example programs using dependency graphs. The results in Sections 3 and 4 are not used later in the main line of development. Section 5 defines a number of terms that are used in Section 6, which contains one of the major contributions of the paper – a direct (and hence potentially efficient) least fixpoint construction for Horn clause logic including negation. Section 7 presents a series of refinements of the fixpoint construction, giving more efficient interpreters for the language. Finally, Section 8 gives conclusions and further work. The website, http://www.cs.waikato.ac.nz/research/jstar, gives further information about the Starlog languages, including example programs from this paper and a reference interpreter for executing them.

2 Syntax and Notation

This section introduces the syntax of the Starlog language and the notation used throughout the paper.

By a *logic program* we mean a finite set of *clauses*, written as:

$$\mathbf{A} \leftarrow \bar{\mathbf{B}}$$

where $\mathbf{A}$ is referred to as the *head* of the clause and $\bar{\mathbf{B}}$ as its *body*. The head $\mathbf{A}$ is an *atom*, which is a predicate symbol applied to zero or more terms. Terms are constructed from constant and function symbols, plus variables, as usual. The body $\bar{\mathbf{B}}$ is a set of *literals* $\mathbf{B}_1, \mathbf{B}_2 ..., \mathbf{B}_m$. A literal is either a *positive literal*, which is just an atom, or a *negative literal*, which is a negated atom [Llo87,PP90].

A subset of the predicate symbols are identified as *built-in predicates* and may not appear in the head or in any negative literals of any program clause. Also, any variable which appears in a clause must appear in at least one positive literal (including built-in literals) in the body. Each clause is universally quantified over all the variables in the clause.

The *language* L of $\mathbf{P}$ consists of all the well-formed formulae of the first order theory obtained in this way. The *Herbrand base* $B_{\mathbf{P}}$ of $\mathbf{P}$ is the set of all ground atoms of the theory [Llo87]. $\mathbf{P}^*$ denotes the *ground instantiation* [Llo87] of the program $\mathbf{P}$. The convention is used that terms which may contain unbound variables will be written in boldface (for example $\mathbf{A} \leftarrow \bar{\mathbf{B}} \in \mathbf{P}$), whereas terms which are ground are written as Roman capitals (for example $A \leftarrow \bar{B} \in \mathbf{P}^*$). By an *interpretation* I of $\mathbf{P}$ we mean a subset of the Herbrand base $B_{\mathbf{P}}$. The

complete semantics of *built-in* predicates is represented by the interpretation $I_\circ$. For example, $2 < 3$ is a member of $I_\circ$, while $3 < 2$ is not.

Definition 1. (Reduction [PP90]) *The reduction of* $\mathbf{P}^*$ *modulo an interpretation* I *is the set of (ground) clauses*

$$\mathbf{P}^*/I \;\equiv\; \{A \leftarrow (\bar{B} - I) \mid A \leftarrow \bar{B} \in \mathbf{P}^* \wedge (I \not\models \neg\bar{B})\}$$

We will be particularly interested in $\mathbf{P}^*/I_\circ$, the reduction modulo the built-in predicates. This removes from $\mathbf{P}^*$ all clauses that contain a builtin that evaluates to false.

Given the body $\bar{\mathbf{B}}$ of a clause, we distinguish the following four subsets:

- $\bar{\mathbf{B}}^+$ the positive literals that are not built-in predicates.
- $\bar{\mathbf{B}}^-$ the negative literals.
- $\bar{\mathbf{B}}^\sim$ the negative literals with their negation stripped from them.
- $\bar{\mathbf{B}}^\circ$ the built-in predicates.

We will require all programs to be written in a *causal* style, so that there exists some causality ordering over all the tuples such that rules always add new tuples in the future, not in the past. We base this notion of causality on the idea of a well-founded ordering. Our goal is to prevent temporal contradictions, such as the Grandfather Paradox of time travel [Bar43], where someone travels back in time and kills his own biological grandfather before the latter met the traveler's grandmother, thus making it impossible for the traveler to exist. Informally, we want to ensure that the output (the head) of each rule is generated after (or at the same time) as the inputs of the rule, so that the output cannot modify the inputs and lead to a contradiction. We shall see that this is equivalent to *local stratification* from the database and logic programming literature [Prz88].

Definition 2. (Well-founded) *A binary relation* $<$ *over a set* X *is* well-founded *iff it has no infinite descending chains. Equivalently, if every non-empty subset of* X *has a minimal element with respect to* $<$*.*

Definition 3. (Causal pair $\lesssim, <$) *Throughout the paper we will be using a* causal pair *of orderings, one a well-founded strict partial order (irreflexive ordering)* $<$ *and the other a pre-order (reflexive transitive ordering)* $\lesssim$ *on the Herbrand base. (In general these depend on the program* $\mathbf{P}$*.)*

The two orderings are related by:

$$x < y \Rightarrow x \lesssim y$$
$$x < y \wedge y \lesssim z \Rightarrow x < z$$
$$x \lesssim y \wedge y < z \Rightarrow x < z$$

These orderings are extended to negative literals by adding the following axioms and forming the minimal transitive closure of the relations:

$$x \lesssim y \Rightarrow x < not(y)$$
$$x < y \Rightarrow not(x) < y$$

$\lesssim$ is also extended to ground clauses by adding $(A \leftarrow \bar{B}) \lesssim (C \leftarrow \bar{D})$ iff $A \lesssim C$ and forming the transitive closure.

To understand the ordering of negative literals, note that for any pair of positive tuples A_1, A_2 such that $A_1 < A_2$, we have $A_1 < not(A_1) < A_2$. This shows that $not(A_1)$ becomes known *immediately after* the calculation of A_1 has been completed. If that calculation did produce the tuple A_1, then $not(A_1)$ is false, whereas if the calculation failed to produce A_1, then $not(A_1)$ is true.

Our definition of causal programs makes use of the completion of a program, $comp(\mathbf{P})$ [ABW88,Llo87]. This allows us to use global invariants of the program to restrict attention to the instances that can actually occur during execution. This makes it easier to prove that individual rules are in fact causal. Calculating the completion of a whole program is inconvenient in practice, so in Section 3 we shall describe more practical ways of proving that a program is causal.

Definition 4. (Causal) *A program* $\mathbf{P}$ *is* causal *iff there is a causal pair* $\lesssim, <$ *such that for every rule instance* $A \leftarrow \bar{B} \in \mathbf{P}^*$ *where* $comp(\mathbf{P}), I_\circ \models \bar{B}$

$$\forall B \begin{pmatrix} B \in \bar{B}^+ \Rightarrow B \lesssim A \\ B \in \bar{B}^\sim \Rightarrow B < A \end{pmatrix}$$

Definition 5. (Strongly Causal) *A program* $\mathbf{P}$ *is* strongly causal *iff there is a causal pair* $\lesssim, <$ *such that for every rule instance* $A \leftarrow \bar{B} \in \mathbf{P}^*$ *where* $comp(\mathbf{P}), I_\circ \models \bar{B}$

$$\forall B \left(B \in \bar{B}^+ \cup \bar{B}^\sim \Rightarrow B < A\right)$$

Strong causality permits the later interpreters to be simplified and gives more precise control over execution order. However, it often makes it harder to actually write programs. The transitive closure example in section 4.4 illustrates this point.

Next we use our notion of causal programs to show that the input program is *locally stratified* [PP90], which means that it has the usual perfect model semantics [Prz88]. Local stratification requires the Herbrand universe to be partitioned into *strata*, $H_0, H_1, \ldots H_\beta$, where β is a countable ordinal, and for each instantiation of a rule $A \leftarrow B$, if $A \in H_i$ then all the positive literals of B must be in $\bigcup\{H_j | j \leq i\}$ and all the negative literals of B must be in strata $\bigcup\{H_j | j < i\}$ [Prz88, Defn. 5].

Theorem 6. *A causal program* $\mathbf{P}$ *that terminates is locally stratified, so has a unique perfect model [Prz88], which is also equal to the unique minimal model defined by Apt* et. al. *[ABW88].*

Proof. Since $\mathbf{P}$ is causal, it has a pre-order $\lesssim$ that is well-founded. From this pre-order, we can construct a partial order by taking the equivalence classes induced by $\lesssim$, that is, two atoms a and b are in the same equivalence class iff $a \lesssim b \lesssim a$. Then we can take a linear extension [DP02] of that partial order, to obtain a total order $H_0, H_1, \ldots$, which we use as the stratification order for $\mathbf{P}$. Note that this total order has a minimum element, since $\lesssim$ is well-founded. If $\mathbf{P}$ terminates

after a finite number of deductions, then there exists a countable ordinal bound β such that the last tuple produced is in H_β, so we have constructed a local stratification order $H_0, H_1, \ldots H_\beta$.

Thus **P** is a locally stratified logic program, and by Theorem 4 of Przymusinski [Prz88], **P** has a unique perfect model that coincides with the unique minimal model defined by Apt *et. al.* □

We have now established the semantics of a terminating causal program P. However, defining the semantics in this way does not give as much flexibility for parallel execution as we would like. It evaluates rules in stratification order, and this is unnecessarily restrictive. In Section 5 we will define more general evaluation operators that allow more parallelism, and we will prove that they give the same results as this standard semantics.

3 Dependency Graphs

The definition of the causal orderings $\lesssim, <$ given above and their relationship to a program are abstract and it is not clear how such an ordering can be effectively realized. For example, it will be noted later that computable versions of the orderings are needed. This is especially so as the most precise version of the orderings requires knowledge of $comp(\mathbf{P})$ which is effectively what we are trying to compute.

This section links the definitions above to the idea of dependency graphs which have often been used to develop semantics for datalog with negation. It also shows how the $\lesssim, <$ orderings might be specified and used in practice. However, the material here is used only in the following Section 4, which gives a number of example programs. The main development from Section 5 onwards relies only on the abstract notion of a causal pair $\lesssim, <$, so can be used with dependency graphs, or with any other technique for finding a causal pair.

The remaining definitions in this section follow the order in which we can use dependency graphs to analyze and execute a program **P**:

1. We perform static analysis on **P** to deduce various facts about it, such as invariants, types and range information - we call such information a *theory* of the program.
2. We use that theory to calculate a conservative superset of the instances of the rules that may be true during the execution of the program, and we calculate a *dependency graph* from those rule instances. This dependency graph corresponds to a causality ordering between all the tuples that may be generated by the program.
3. We then check that the derived dependency graph is well-founded (contains no cycles through negations). This is not necessarily decidable, but if we cannot prove that the dependency graph is well-founded, we require the programmer to strengthen or correct the program.

3.1 Static Analysis Theories

Definition 7. (Theory)
$\mathbf{L}$ *is a theory of a program* $\mathbf{P}$ *iff*

$$comp(\mathbf{P}) \wedge I_\circ \Rightarrow \mathbf{L}$$

An example of a simple theory about a program that calculates primes is

$$\forall N(prime(N) \Rightarrow 2 \leq N)$$

A theory $\mathbf{L}$ gives us partial information about the behavior of the program. In particular, it allows us to deduce that some ground atoms $\mathbf{L}^+$ *will* be produced by the program, while other ground atoms $\mathbf{L}^-$ will *never* be produced by the program. For atoms not in $\mathbf{L}^+ \cup \mathbf{L}^-$, the theory is incomplete - it does not tell us whether or not they will be produced.

Definition 8. $(\mathbf{L}^+, \mathbf{L}^-)$

$$\begin{array}{lcl} \mathbf{L}^+ & \equiv & \{A \in B_\mathbf{P} \mid \mathbf{L} \models A\} \\ \mathbf{L}^- & \equiv & \{A \in B_\mathbf{P} \mid \mathbf{L} \models not(A)\} \end{array}$$

For example, our simple theory of primes tells us that $prime(1) \in \mathbf{L}^-$, so 1 cannot be a prime, whereas the status of $prime(2)$ is unknown according to this theory. We may be able to deduce a stronger theory, which tells us more about the possible behaviour of the program.

Definition 9. (stronger) *A theory* $\mathbf{L}_2$ *is* stronger *than a theory* $\mathbf{L}_1$ *iff*

$$\mathbf{L}_1^+ \subseteq \mathbf{L}_2^+ \wedge \mathbf{L}_1^- \subseteq \mathbf{L}_2^-$$

An example of a stronger theory than the one above is

$$prime(2) \wedge prime(3) \wedge \forall N(prime(N) \Rightarrow (N = 2 \vee N = 3 \vee 5 \leq N)$$

This tells us that $prime(2)$ and $prime(3)$ are in L^+ so are definitely primes, while $prime(1)$ and $prime(4)$ are in L^-, so cannot be primes. We can also use theories to capture information about types, functional dependencies, possible values of variables, etc. The strongest theory is $comp(\mathbf{P})$ itself.

Definition 10. (Restriction) *The* restriction *of a program* $\mathbf{P}$ *modulo* $\mathbf{L}$ *is the set of ground clauses:*

$$\mathbf{P} \,//\, \mathbf{L} \equiv \{A \leftarrow \bar{B} \mid A \leftarrow \bar{B} \in \mathbf{P}^*/I_\circ \wedge \bar{B}^+ \cap \mathbf{L}^- = \emptyset \wedge \bar{B}^\sim \cap \mathbf{L}^+ = \emptyset\}$$

So $\mathbf{P} \,//\, \mathbf{L}$ is the set of rule instantiations that are consistent with $\mathbf{L}$ and whose builtins are all true.

3.2 Dependency Graphs and Causality

We define dependency graphs in the usual way [ABW88,PP90], except that we define them only over the subset of the program that satisfies a given theory.

Definition 11. (Dependency Graph) *The vertices of the* dependency graph $G_{\mathbf{P},\mathbf{L}}$ *of a program* $\mathbf{P}$ *with respect to* $\mathbf{L}$ *are all the ground atoms appearing in* $\mathbf{P} \,//\, \mathbf{L}$. *The edges of* $G_{\mathbf{P},\mathbf{L}}$ *are defined as follows. For every clause* $A \leftarrow \bar{B} \in \mathbf{P} \,//\, \mathbf{L}$, *there is a* positive *directed edge from each* $B \in \bar{B}^+$ *to* A *and there is a* negative *directed edge from each* $B \in \bar{B}^\sim$ *to* A.

The dependency relations $\lesssim, <$ *between ground atoms of* $\mathbf{P}$ *are defined by:*

- $B \lesssim A$ *iff there is a directed path from* B *to* A, *or if* $A = B$.
- $B < A$ *iff there is a directed path from* B *to* A *that passes through at least one negative edge.*

Theorem 12. *A pair of orderings* $\lesssim, <$ *generated by a dependency graph* $G_{\mathbf{P},\mathbf{L}}$ *are a causal pair provided* $<$ *is well founded.*

Proof. The elementary properties of the orderings follow directly from the definition.

Theorem 13. *A program* $\mathbf{P}$ *is causal if the orderings generated by* $G_{\mathbf{P},\mathbf{L}}$ *for a theory* $\mathbf{L}$ *of* $\mathbf{P}$ *are causal, that is,* $<$ *is well-founded.*

If we use a simple (weak) theory about the program, we may derive a dependency graph whose $<$ ordering is not well-founded, perhaps because it contains loops. In this case we could try a stronger theory about the program, to obtain a smaller dependency graph that is more likely to have a well-founded $<$ ordering. If we cannot find any theory that leads to a well-founded $<$ ordering, then we consider the original program to be erroneous, and require the programmer to strengthen it so that it is possible to find a well-founded $<$ order.

Typically, a program is non-causal because two or more rules define opposing causality orderings between tuples. For example, the following program is not causal, because `a(2)` $\lesssim$ `b(2)`, but `b(2)` $<$ `a(2)` is also true (instantiating the first rule with `T=2`).

```
a(T) <-- 0 < T, T < 4, not(b(T)).
b(2) <-- a(2).
```

This program could be made causal by changing the head of the first rule to be `a(T+1)`, which would give an order of $a(2) \lesssim b(2) < a(3)$.

4 Example Programs

This section shows several example Starlog programs. These examples are intended to illustrate the style of the language and a range of different applications. For each of these programs, we investigate possible theories, their resultant orderings and proofs that these are well-founded.

4.1 Builtin Predicates

We assume a number of builtin predicates covering input, output, arithmetic and the generation of ranges of integers.

Input provided externally will appear as tuples `input(T, X)` where `T` is an integer timestamp $\texttt{T} \geq 0$ and `X` is the input itself.

Output is provided by the predicate `println(T, X)`. The programer generates tuples of this form, which are then output to some suitable external channel. `T` is a time stamp with $\texttt{T} \geq 0$ and `X` is the data to be output. To programmers used to languages such as Prolog this idiom may be somewhat startling, as the `println` tuples appear in the head of rules not the bodies. However, this is an important part of ensuring that the language has a pure semantics.

We assume that five builtin arithmetic predicates are available over the integers: $<$ and $\leq$, which provide ordering; addition written as `Z is X + Y` to follow standard Prolog practice; subtraction written as `Z is X - Y`; and multiplication written as `Z is X * Y`. In each case `X`, `Y` must be ground integers for these to be executed.

The final builtin predicate `range(N, Lo, Hi)` generates all the integers in the range `Lo`...`Hi`, potentially in parallel. Like the other arithmetic predicates `Lo`, `Hi` must be ground integers for this to be executed. The pragmatic reason for introducing this predicate is that it enables us to ignore issues around the efficient parallel generation of ranges of integers for these examples. It is trivial to write an implementation that is sequential and serializes all parts of the program that depend on it. It is less easy to write an implementation that generates integers in a way that does not restrict the available parallelism.

4.2 Finding Prime Numbers

```
01: max(5000) <-- true.

02: mult(M, P) <-- mult(N, P), M is N + P, max(Max), M < Max.
03: mult(M, P) <-- prime(P), M is P * P, max(Max), M < Max.

04: mult(M) <-- mult(M, _).

05: prime(N) <-- max(M), range(N, 2, M), not(mult(N)). % Deduce prime numbers

06: println(N, prime(N)) <-- prime(N).
```

Fig. 2. Sieve of Eratosthenes

Our first example program in Fig 2 generates all the prime numbers upto a given number specified by the predicate `max`, using the Sieve of Eratosthenes. It

generates all multiples of known primes in `mult(N)`, and uses negation to find numbers that are not multiples, so must be primes.

The different multiples `M` are generated in the predicate `mult(M, P)` for each prime `P` by starting at `P * P` and adding successive increments of `P`.

When this program is executed the `println` predicate generates the following output:

```
prime(2)
prime(3)
prime(5)
prime(7)
prime(11)
...
prime(4999)
```

The following table shows the details of the execution up to time 12, which is the first time that there are multiple `mult(_,_)` tuples at the same time.

mult/2	mult	prime	println
-	-	(2)	(2,prime(2))
-	-	(3)	(3,prime(3))
(4, 2)	(4)	-	-
-	-	(5)	(5,prime(5))
(6, 2)	(6)	-	-
-	-	(7)	(7,prime(7))
(8, 2)	(8)	-	-
(9, 3)	(9)	-	-
(10, 2)	(10)	-	-
-	-	(11)	(11,prime(11))
(12, 2)	(12)	-	-
(12, 3)			

Theory The following simple theory for the program constrains the range of the parameters to be 2 or greater. As well it contains some simple deductions about integers, which are useful for checking that the rules are causal under this theory.

$$\texttt{mult(M, P)} \Rightarrow \texttt{M} \geq 2, \texttt{P} \geq 2$$
$$\texttt{mult(M)} \Rightarrow \texttt{M} \geq 2$$
$$\texttt{prime(N)} \Rightarrow \texttt{N} \geq 2$$
$$\texttt{println(N, T)} \Rightarrow \texttt{N} \geq 2, \texttt{T = prime(N)}$$
$$\texttt{M is N+P, N} \geq 2, \texttt{P} \geq 2 \Rightarrow \texttt{M > N}$$
$$\texttt{M is P*P, P} \geq 2 \Rightarrow \texttt{M > P}$$

The ordering from the dependency graph that results from this theory divides all possible tuples into strata. There is one stratum for each integer $\texttt{N} \geq 2$, which

contains the tuples

$$\{\texttt{mult(N,\_)},\ \texttt{mult(N)},\ \texttt{num(N)},\ \texttt{prime(N-1)},\ \texttt{println(N-1,\_)}\}$$

Because the only negative edge in the causality graph is from `mult(N)` to `prime(N)`, `prime(N)` and `println(N,prime(N))` are 'pushed up' to the next stratum.

Execution of the program using this ordering and the Ev selection operator (see Definition 44) will be sequential, with all processing taking place for stratum `N`, followed by all processing for stratum `N+1` and so on.

We now introduce a stronger theory for this program. This places constraints on the ranges of the tuples (some start at 4 rather than 2) and adds the very significant fact that `mult(M,P)` implies $\texttt{M} \geq \texttt{P * P}$. As we will see, this makes a significant difference to the potential parallelism available in the program.

$$\begin{aligned}
\texttt{mult(M, P)} &\Rightarrow \texttt{M} \geq 4,\ \texttt{P} \geq 2,\ \texttt{M} \geq \texttt{P*P} \\
\texttt{mult(M)} &\Rightarrow \texttt{M} \geq 4 \\
\texttt{prime(N)} &\Rightarrow \texttt{N} \geq 2 \\
\texttt{println(N, T)} &\Rightarrow \texttt{N} \geq 2,\ \texttt{T = prime(N)} \\
\texttt{M is N+P},\ \texttt{N} \geq 2,\ \texttt{P} \geq 2 &\Rightarrow \texttt{M} > \texttt{N} \\
\texttt{M is P*P},\ \texttt{P} \geq 2 &\Rightarrow \texttt{M} > \texttt{P}
\end{aligned}$$

Fig. 3 shows the the initial part of the $\lesssim, <$ ordering that results from this theory (`println` and `max` are omitted for clarity). It is straightforward to show using the theory that these orderings are well founded and that they are a causal pair, thus showing that the program itself is causal.

From the figure it can be seen that $\texttt{prime(3)} \lesssim \texttt{mult(9)} < \texttt{prime(9)}$ rather than $\texttt{prime(8)} < \texttt{prime(9)}$ which was true in the earlier simple ordering. Informally $\texttt{prime(N)} > \texttt{prime}(\sqrt{\texttt{N}})$ which allows all the calculations between $\texttt{prime}(\sqrt{\texttt{N}})$ and `prime(N)` to occur in parallel (of course, there are positive dependencies between the `mult(N, P)` tuples but as soon as they are generated all consequent calculations can proceed without being restricted by causal considerations).

One of the requirements for a practical system (see Section 7.4) is that it be possible to compute the orderings $\lesssim$ and $<$. That is, given two ground tuples it must be possible to compute, preferably quickly, whether they are in fact ordered. To demonstrate that this is possible, the following is an explicit calculation of a causal pair that is slightly stronger than the ordering above. This code is written in Prolog, *not* Starlog.

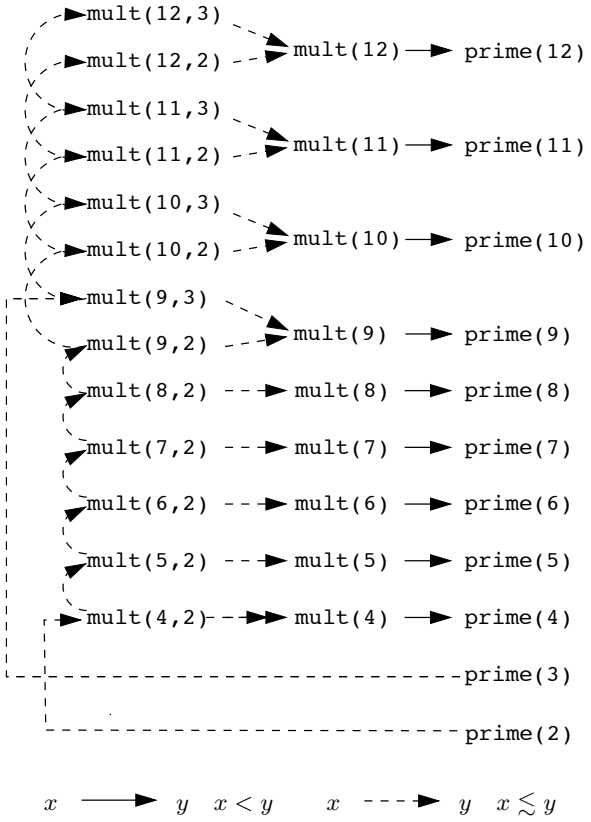

Fig. 3. Ordering for Primes Using Strong Theory

$$\texttt{mult(N1)} < \texttt{prime(N2)} \texttt{ :- } 4 \leq \texttt{N1}, \texttt{N1} \leq \texttt{N2}.$$
$$\texttt{mult(N1,\_P)} < \texttt{prime(N2)} \texttt{ :- } 4 \leq \texttt{N1}, \texttt{N1} \leq \texttt{N2}.$$
$$\texttt{prime(N1)} < \texttt{prime(N2)} \texttt{ :- } 2 \leq \texttt{N1}, \texttt{N1*N1} \leq \texttt{N2}.$$

$$\texttt{mult(N1)} < \texttt{mult(N2)} \texttt{ :- } 4 \leq \texttt{N1}, \texttt{N1*N1} \leq \texttt{N2}.$$
$$\texttt{mult(N1,\_P)} < \texttt{mult(N2)} \texttt{ :- } 4 \leq \texttt{N1}, \texttt{N1*N1} \leq \texttt{N2}.$$
$$\texttt{prime(N1)} < \texttt{mult(N2)} \texttt{ :- } 4 \leq \texttt{N1}, \texttt{N1*N1*N1*N1} \leq \texttt{N2}.$$

$$\texttt{mult(N1)} < \texttt{mult(N2,\_P)} \texttt{ :- } 4 \leq \texttt{N1}, \texttt{N1*N1} \leq \texttt{N2}.$$
$$\texttt{mult(N1,\_P)} < \texttt{mult(N2,\_P)} \texttt{ :- } 4 \leq \texttt{N1}, \texttt{N1*N1} \leq \texttt{N2}.$$
$$\texttt{prime(N1)} < \texttt{mult(N2,\_P)} \texttt{ :- } 4 \leq \texttt{N1}, \texttt{N1*N1*N1*N1} \leq \texttt{N2}.$$

$$\texttt{A} \lesssim \texttt{B} \texttt{ :- A} < \texttt{B}.$$

$$\texttt{mult(N1)} \lesssim \texttt{mult(N2)} \texttt{ :- } 4 \leq \texttt{N1}, \texttt{N1} \leq \texttt{N2}.$$
$$\texttt{mult(N1,\_P)} \lesssim \texttt{mult(N2)} \texttt{ :- } 4 \leq \texttt{N1}, \texttt{N1} \leq \texttt{N2}.$$
$$\texttt{mult(N1,P)} \lesssim \texttt{mult(N2,P)} \texttt{ :- } 4 \leq \texttt{N1}, \texttt{N1} \leq \texttt{N2}, 2 \leq \texttt{P}.$$

As noted above the tuple `prime(N)` is strictly dependent on `prime(`$\sqrt{\texttt{N}}$`)` — this and similar relationships are reflected in the code above by the appearance of the terms `N1*N1` in the bodies of the rules. In the orderings `prime(N1)` $<$ `mult(N2)` and `prime(N1)` $<$ `mult(N2,P)` the even more spectacular term `N1*N1*N1*N1` appears. The following chain when `N1 = 2` exemplifies the origins of this fourth power of `N1`:

$$\texttt{prime(2)} \lesssim \texttt{mult(4,2)} \lesssim \texttt{mult(4)} < \texttt{prime(4)} \lesssim \texttt{mult(16,4)} \lesssim \texttt{mult(16)}$$

4.3 A Running-Maximum Program

The next example program outputs the maximum of all input numbers seen so far. It illustrates external input (the `input(Time,Number)` relation is an input to this program), negation, assignment and how to make large jumps in time. The use of assignment here is particularly notable as it is often seen as being difficult or impossible in pure functional or logic languages. However, because the time ordering is explicit we are able to directly express the logic of assignment.

Lines 9-21 can be viewed as a library that implements assignment. Sending a `val(T,K)` request to the library causes a `value(T,K,M)` response to be returned, where `M` is the value associated with key `K` at time `T`. Sending an `assign(T,K,M)` tuple to the library sets the current value of `K` to `M`.

In practice, we often write negations like lines 14 and 16-19 in a sugared form,

```
not(exists U assign(U, K, _), T0 < U, U < T)
```

and omit the definition of auxiliary predicates such as `value_neg`. But to keep the semantics clear, we shall avoid such syntactic sugar in this paper.

```
01: println(T, max(T, M)) <-- assign(T, max, M).
02:
03: assign(T, max, N) <-- input(T, N), value(T, max, M), M < N.
04: assign(T, max, N) <-- input(T, N), not(value(T, max, _)).
05:
06: val(T, max) <-- input(T, _).
07:
08:
09: % This records the current assignment (when each input arrives).
10: value(T, K, M) <--
11:    val(T, K),
12:    assign(T0, K, M),
13:    T0 < T,
14:    not(value_neg(T, K, T0)).
15:
16: value_neg(T, K, T0) <--
17:    val(T, K),
18:    assign(T0, K, _),
19:    T0 < T,
20:    assign(U, K, _),
21:    T0 < U, U < T.
```

Fig. 4. Running Maximum

Here is an example execution with four input numbers arriving externally at various times. For real-time reactive programming, these arrival times might correspond to seconds or milliseconds. For non real-time programming, they might correspond to the line numbers of an input file that is read sequentially, where the missing line numbers correspond to input lines that are empty or do not contain a valid number. Given the following external inputs:

```
input(1, 13)
input(4, 11)
input(7, 23)
input(10, 17).
```

the program generates the following external outputs:

```
max(1, 13)
max(7, 23)
```

The tuples generated during execution are shown in the following table:

input	val	value	value_neg	assign	println
(1,13)	(1, max)	-	-	(1,max,13)	(1,max(1,13))
(4,11)	(4, max)	(4,max,13)	-	-	-
(7,23)	(7, max)	(7,max,13)	-	(7,max,23)	(7, max(7,23))
(10,17)	(10, max)	(10,max,17)	(10,max,1)	-	-

Theory and Ordering The only interesting theory for the program asserts that for each predicate the parameter `T` $\geq$ `0` and the key `K = max` (this parameter is included to make it clear that this assignment logic is easily extended to multiple keys). The constraint `T` $\geq$ `0` flows from the original constraint in the `input` predicate.

The ordering generated by the dependency graph for this theory divides the tuples into one stratum for each integer from 0 upward. Each such stratum is then further split into substrata in increasing order {`val`}, {`value_neg`}, {`value`}, and {`assign`, `println`, `input`}. It is straightforward to show that this ordering is well founded and strongly causal.

4.4 Transitive Closure of a Graph

```
01: t(X, Y) <-- r(X, Z), t(Z, Y).
02: t(X, Y) <-- r(X, Y).
```

Fig. 5. Simple Transitive Closure

The third example program computes the transitive closure `t(X,Y)` over a base relation `r(X,Y)`. The first version of this program is simple (and for some base relations, very inefficient). Termination relies on the fact that if a tuple is generated more than once then it only triggers further computation the first time. This program has a trivial ordering where all the tuples are equivalent to each other, that is, all tuples are grouped into a single stratum. With this ordering the program is causal but not strongly causal.

The second version of the program is given as an illustration of how to convert a weakly causal program to a strongly causal one. To do this a counter `I` is added for each iteration of the transitive closure in the tuples `tr(I, X, Y)` (which means that a new transitive link from `X` to `Y` has been computed during iteration `I`). An explicit check that tuples computed in earlier iterations are not repeated is made on line `04` using the predicate `tr_neg`. The negation can be replaced by the syntactically sugared construction `not(tr(K,X,Y), K =< I)` which obviates any need to define the predicate `tr_neg`.

The interaction between `tr` and `tr_neg` can be seen as a simpler variant of the assignment pattern used in the running-maximum example of the previous section. In this case each value is assigned only once, and the `tr_neg` tuples prevent later re-assignments of the values.

The following table shows the details of the execution of this second version of the program. Fig. 7 shows a diagram of the base relation `r(_,_)` used in the example.

```
01: t(X, Y) <-- tr(_, X, Y).

02: tr(J, X, Y) <--
03:        r(X, Y), tr(I, Z, Y), J is I+1
04:        not(tr_neg(I, X, Y)).

05: tr_neg(I, X, Y) <--
06:        r(X, Z), tr(I, Z, Y),
07:        tr(K, X, Y), K =< I.

08: tr(0, X, Y)   <-- r(X, Y).
```

Fig. 6. Strongly Causal Transitive Closure

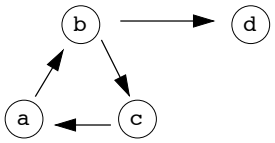

Fig. 7. Base relation for Transitive Closure Example

tr	tr_neg
(0, a, b) (0, b, c) (0, b, d) (0, c, a)	-
(1, c, b) (1, a, c) (1, a, d) (1, b, a)	-
(2, b, b) (2, c, c) (2, c, d) (2, a, a)	(2, a, b) (2, b, c) (2, b, d) (2, c, a)

Theory and Ordering Like the previous example, the theory for this program is straightforward, constraining the `I` of `tr(I,_,_)` and `tr_neg(I,_,_)` so that $\texttt{I} \geq 0$.

This leads to the program being split into three strata in the following order: the input tuples {`r(_,_)`}, the internal tuples {`tr(_,_,_)`, `tr_neg(_,_,_)`}, and the final result {`t(_,_)`}. The {`tr(_,_,_)`, `tr_neg(_,_,_)`} stratum is then split into substrata, one for each integer from 0 upward. Fig. 8 illustrates this ordering. The program is easily shown to be strongly causal, requiring only use of the results that `I < I+1` for line `03` and the transitivity of integer $\leq$ on line `07`.

5 Semantic Concepts

This section introduces several operators and relations that are needed to define the semantics of the language. We will be considering a number of different operators on the Herbrand universe $V : 2^{B_\mathbf{P}} \rightarrow 2^{B_\mathbf{P}}$. Many of them are taken

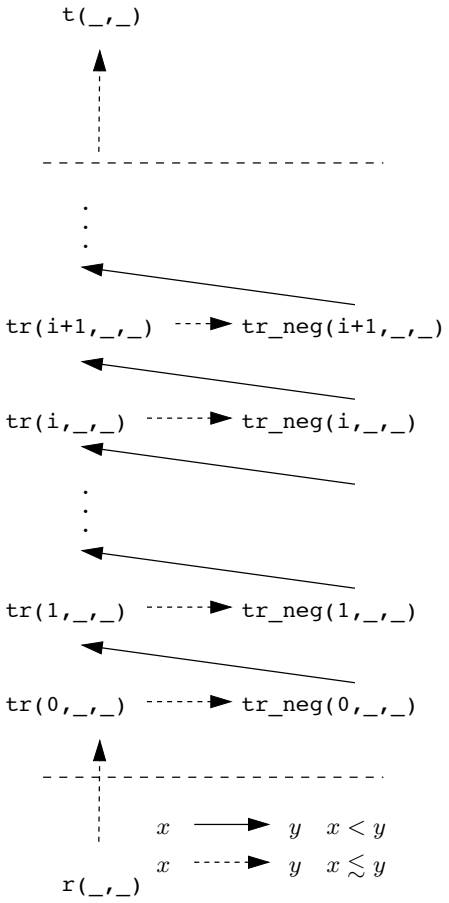

Fig. 8. Ordering for Transitive Closure Example

from the standard literature on the semantics of logic programming languages, but some, like the *selection operators* in Section 5.1, are new.

The *immediate consequence operator* $T_{\mathbf{P}}$ performs one bottom-up deductive step, deducing the heads of all rules whose bodies are true. That is, $T_{\mathbf{P}}(I)$ computes all consequences that are true given an interpretation I.

Definition 14. (Immediate consequence operator $T_{\mathbf{P}}$ [Llo87, p37]) *$T_{\mathbf{P}}(I)$ is the set of all atoms $A \in B_{\mathbf{P}}$ such that there is a clause $A \leftarrow \bar{B} \in \mathbf{P}^*$, where $\bar{B}$ follows from the interpretation I and the builtins $I_\circ$:*

$$T_{\mathbf{P}}(I) \;\equiv\; \{A \mid A \leftarrow \bar{B} \in \mathbf{P}^* \wedge (I, I_\circ \models \bar{B})\}$$

Definition 15. (Monotonic) *An operator V is monotonic (with respect to an ordering $\subseteq$) iff*

$$\forall I, J (I \subseteq J \Rightarrow V(I) \subseteq V(J))$$

In programs without negation, the immediate consequence operator $T_{\mathbf{P}}$ is monotonic with respect to the subset ordering. However, in the presence of negation it may not be. The technical work below is mainly concerned with finding a variant of $T_{\mathbf{P}}$ and an ordering on interpretations to restore monotonicity.

We will only ever need to consider one program at a time so we usually omit the subscript $\mathbf{P}$ from the operators in what follows. We also assume that $\mathbf{P}$ is at least causal.

Definition 16. (Δ)

$$\Delta(I) \;\equiv\; T(I) - I$$

Δ computes all the *new* consequences that are derivable from I.

We will need to apply operators repetitively to generate a fix point.

Definition 17. (V^α) *For all ordinals α and operators V we define $V^\alpha(I)$ as follows:*

$$\begin{aligned} V^0(I) &= \emptyset \\ V^{\alpha+1}(I) &= V(V^\alpha(I)) \\ V^\alpha(I) &= \bigcup_{\beta<\alpha} V^\beta(I) \text{ where } \alpha \text{ is a limit ordinal.} \end{aligned}$$

For the special case $V^\alpha(\emptyset)$, we write V^α.

Next, we define an aggressively parallel operator Π, that will allow us to support a range of alternative parallel evaluation strategies.

Definition 18. (Π)

$$\Pi(I) \;\equiv\; \{A \mid A \leftarrow \bar{B} \in \mathbf{P}^* \wedge (I, I_\circ \models \bar{B}) \wedge \nexists y, z (y \in \Delta(I) \wedge z \in \bar{B}^\sim \wedge y \lesssim z)\}$$

$\Pi(I)$ approximates the largest set of consequences that can be 'safely' deduced from I, that is, consequences that can not be later contradicted by new consequences that invalidate the negations in rules. Π includes all the derivations in T except where the generating rule contains a negation which is foreshadowed by tuples which are earlier in the ordering and in the newly derived results.

It is possible to directly specify Π only because of the existence of the $\lesssim$ ordering. The major contribution of this part of the paper is to show how Π can be used both to directly specify a semantics and to effectively compute it.

Π can be somewhat surprising in its effect. For example, it can permit tuples that are in the future to be used for further computation, that is, it does not force execution to proceed in a stratified ordering, except where this is forced by negations. This can be a mixed blessing, on the one hand it gives maximal parallelism, on the other it does not give precise control over the order of execution or of the resource consumption implied by that. The following section generalizes Π to a set of *selection operators*, which can give finer control over execution order.

5.1 Selection Operators

During program execution we want flexibility about what newly deduced facts trigger further computation. For example, in a sequential execution it may be more efficient to select one tuple at a time, or in distributed execution the flexibility may help to avoid excessive latency. *Selection operators* provide room to do this. They choose a subset of $\Pi(I)$ (including I itself). Π itself is the most inclusive selection operator.

Definition 19. (Selection Operator) *An operator V is a* selection operator *iff*

$\Pi(I) \cap I \subseteq V(I) \subseteq \Pi(I)$ *and*
$V(I) = I \Rightarrow \Pi(I) = I$.

The first line of this definition ensures that $V(I)$ contains all safe tuples that are already in I, and that it does not choose any unsafe facts—that is, it is bounded above by $\Pi(I)$, which is the set of all safe consequences. The second line ensures that $V(I)$ does not stop choosing new facts too early. It will be shown that any selection operator can be safely used to compute the least fixpoint. So the choice of selection operator gives implementors of Starlog significant freedom to choose different parallel evaluation strategies.

Fig. 9 illustrates the relationship between the selection operators and Π, Δ and the minimal model $M_{\mathbf{P}}$ defined below.

6 Semantics

In this section we will demonstrate that any selection operator has a least-fixpoint which is equal to the perfect model $M_{\mathbf{P}}$, as defined by Przymusinski [Prz88]. We need a couple of definitions before embarking on this.

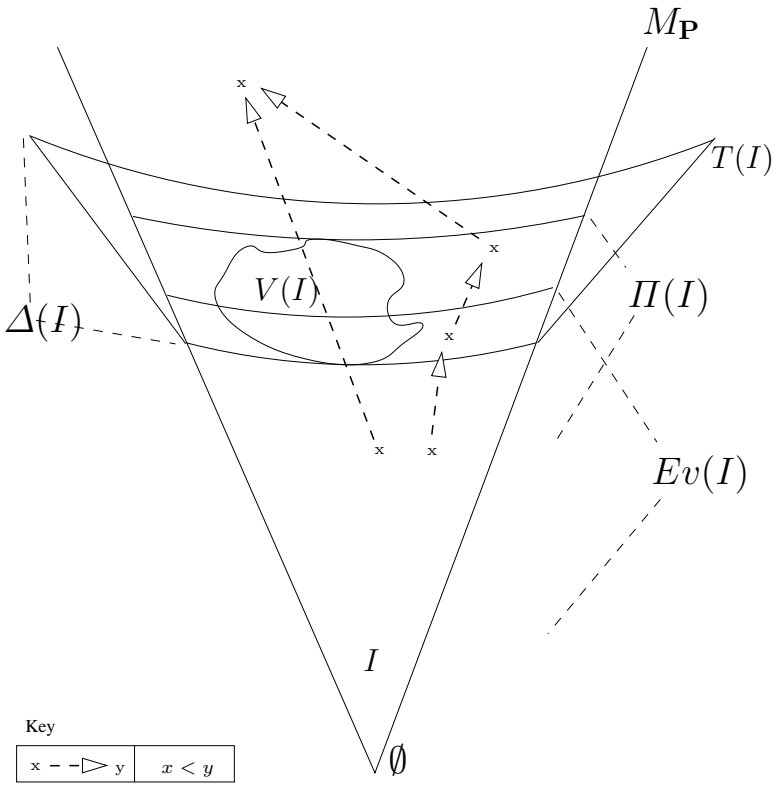

Fig. 9. Selection Operators

Definition 20. (Preferable [Prz88]) *For two interpretations I, J, we say that I is* preferable to *J, written $I \sqsubseteq J$, iff*

$$\forall x\,(x \in I - J \Rightarrow \exists y(y \in J - I \wedge y < x))$$

Definition 21. (Perfect [Prz88]) *A model M of the program $\mathbf{P}$ is* perfect *iff there is no other model K of $\mathbf{P}$ where $K \sqsubseteq M$.*

Often least fixpoints are constructed by showing that the operator is monotone and then applying the Tarski-Knaster theorem. However as the following example shows, this approach cannot be naively followed, because Π is not monotone, neither in the $\subseteq$ ordering nor the $\sqsubseteq$ ordering.

Example:

Consider the following single clause program:

$p \leftarrow \neg q$

together with the ordering $p > q$.

To check the montonicity of Π consider the following cases:

$\emptyset \subseteq \{q\}$, $\emptyset \sqsubseteq \{q\}$ and

$\Pi(\emptyset) = \{p\}$, $\Pi(\{q\}) = \emptyset$ but

$\{p\} \not\subseteq \emptyset$ and $\{p\} \not\sqsubseteq \emptyset$,

showing that Π is not monotone on either ordering.

The least-fixpoint will be constructed in two stages. First we establish the following three conditions on any selection operator V:

1. For all ordinals α, if $I = V^\alpha$ then $V(I) = I$ iff $T(I) = I$
2. $\alpha \leq \beta \Rightarrow V^\alpha \subseteq V^\beta$
3. For any model K of the program $\mathbf{P}$ and ordinal α then $V^\alpha \sqsubseteq K$

Note that these conditions apply only to the interpretations V^α, *not* to all interpretations. As shown by the example earlier, the conditions do not hold in general and require the construction of the least fixpoint to occur in the space only of the sets V^α, not the space of all possible interpretations.

Secondly we use these results to construct a least fixpoint and show that it is equal to $M_{\mathbf{P}}$.

Theorem 22. *For a selection operator V, $V(I) = I$ iff $T(I) = I$.*

Proof. Assume $T(I) = I$. From the definition of Δ, $\Delta(I) = \emptyset$. From the definition of Π, $\Pi(I) = T(I) = I$ which in turn implies $V(I) = I$.

Assume $V(I) = I$. From the definition of selection operator $V(I) \subseteq \Pi(I)$ and from the definition of Π, $\Pi(I) \subseteq T(I)$, thus $I = V(I) \subseteq T(I)$. Conversely, $V(I) - I = \emptyset$ and from the definition of selection operator $\Delta(I) = \emptyset$, which implies $T(I) \subseteq I$. □

Theorem 23. *Given a selection operator V, then for all ordinals α,*

$$V^\alpha \subseteq V(V^\alpha)$$

and

$$x \in \Delta(V^\alpha) \Rightarrow \forall\beta(\beta < \alpha \Rightarrow \exists y(y \in \Delta(V^\beta) \wedge y \lesssim x))$$

Proof. The proof will proceed by a transfinite induction on both hypotheses in concert. Both hypotheses are trivially true for $\alpha = 0$.

Consider the case when α is a sucessor ordinal and let $\alpha = \beta + 1$. Note that by the induction hypothesis $V^\beta \subseteq V^\alpha$.

First establish that for $x \in \Delta(V^\alpha)$ there exists $y \in \Delta(V^\beta), y \lesssim x$. This establishes the more general condition by recursion on β. From the definition of Δ, $x \in \Delta(V^\alpha)$ implies $x \notin V^\alpha$ and that there is some ground clause $x \leftarrow \bar{B} \in \mathbf{P}^*/I_\circ$ where $V^\alpha \models \bar{B}$. By the induction hypothesis $x \notin V^\beta$. We now split into a number of subcases.

First consider the case when $V^\beta \models \bar{B}$. Because $x \notin V^\beta$ then $x \in \Delta(V^\beta)$ and as $x \lesssim x$, x supplies a value for y.

Second consider the case when $V^\beta \not\models \bar{B}$. There are two possible reasons for this: either $y \in \bar{B}^+$ and $y \notin V^\beta, y \in V^\alpha$, that is, $y \in \Delta(V^\beta)$ but by causality $y \in \bar{B}^+$ implies $y \lesssim x$ and thus y satisfies the condition; or $y \in \bar{B}^\sim$ and $y \in V^\beta, y \notin V^\alpha$ which contradicts the induction hypothesis that $V^\beta \subseteq V^\alpha$.

Continuing the successor case consider a counter example x for the subset condition, a member of V^α which satisfies the condition $x \in V^\alpha, x \notin V(V^\alpha)$. From the definition of a selection operator this implies that there is a ground clause $x \leftarrow \bar{B} \in \mathbf{P}^*/I_\circ$ where $V^\beta \models \bar{B}$ and $\nexists y, z(y \in \Delta(I) \wedge z \in \bar{B}^- \wedge y \lesssim z)$. Given that $x \in V^\alpha, x \notin V(V^\alpha))$ and the constraint $\Pi(V^\alpha) \cap V^\alpha \subseteq V(V^\alpha)$ then $x \notin \Pi(V^\alpha)$. There are two possible reasons for this: either $V^\alpha \not\models \bar{B}$ or $V^\alpha \models \bar{B}$ and $\exists y, z(y \in \Delta(V^\alpha) \wedge z \in \bar{B}^- \wedge y \lesssim z)$.

Consider first $V^\alpha \not\models \bar{B}$. There are two possible reasons for this: either $\exists y(y \in \bar{B}^+ \wedge y \in V^\beta \wedge y \notin V^\alpha)$, but this contradicts the hypothesis that $V^\beta \subseteq V^\alpha$; or $\exists y(y \in \bar{B}^- \wedge y \notin V^\beta \wedge y \in V^\alpha)$, which implies that $y \in \Delta(V^\beta)$, but this contradicts the selection of the ground clause $x \leftarrow \bar{B}$.

Consider second $V^\alpha \models \bar{B}$ and $\exists y, z(y \in \Delta(I) \wedge z \in \bar{B}^- \wedge y \lesssim z)$. Using the first result for the successor case this implies that $z \in \Delta(V^\beta)$ which implies that $x \notin \Pi(V^\beta)$ and because $V^\beta \subseteq V^\alpha$ this contradicts the assumption that $x \in V^\alpha$.

This completes the proof of both the induction hypotheses for the successor case.

Consider the case when α is a limit ordinal, that is, $V^\alpha = \bigcup_{\beta<\alpha} V^\beta(I)$. First we will show that given $x \in \Delta(V^\alpha)$ then $\forall\beta(\beta < \alpha \Rightarrow \exists y(y \in \Delta(V^\beta) \wedge y \lesssim x))$. From the definition of Δ, $x \in \Delta(V^\alpha)$ implies $x \notin V^\alpha$ and that there is some ground clause $x \leftarrow \bar{B} \in \mathbf{P}^*/I_\circ$ where $V^\alpha \models \bar{B}$. Consider some $\beta < \alpha$ and note that $x \notin V^\beta$. We now split into a number of subcases.

First, consider the case when $V^\beta \models \bar{B}$. Because $x \notin V^\beta$ then $x \in \Delta(V^\beta)$ and as $x \lesssim x$, x supplies a value for y.

Second, consider the case when $V^\beta \not\models \bar{B}$. There are two possible reasons for this. The first reason is that $y \in \bar{B}^+$ and $y \notin V^\beta, y \in V^\alpha$. These conditions

imply that there is some ordinal $\gamma > \beta$ such that $y \notin V^{\gamma} \wedge y \in V^{\gamma+1}$, which implies $y \in \Delta(V^{\gamma})$. From the induction hypotheses this implies there is some $z \in \Delta(V^{\beta})$ such that $z \lesssim y$. Thus $z \lesssim x$ and this supplies the value of y we are seeking. The second possible reason is that $y \in \bar{B}_{\wedge} y \in V^{\beta} \wedge y \notin V^{\alpha}$ but this contradicts the induction hypothesis that $V^{\beta} \subseteq V^{\alpha}$.

Continuing the limit case consider a counter example x for the subset condition, a member of V^{α} which satisfies the condition $x \in V^{\alpha} \wedge x \notin V(V^{\alpha})$. There is an ordinal $\beta < \alpha$ where $x \notin V^{\beta}$ and $x \in V^{\beta+1}$. This implies that there is a ground clause $x \leftarrow \bar{B} \in \mathbf{P}^*/I_{\circ}$ where $V^{\beta} \models \bar{B}$ and $\nexists y, z(y \in \Delta(I) \wedge z \in \bar{B}^{-} \wedge y \lesssim z)$. Given that $x \in V^{\alpha} \wedge x \notin V(V^{\alpha}))$ and the constraint $\Pi(V^{\alpha}) \cap V^{\alpha} \subseteq V(V^{\alpha})$, then $x \notin \Pi(V^{\alpha})$. There are two possible reasons for this: either $V^{\alpha} \not\models \bar{B}$ or $V^{\alpha} \models \bar{B}$ and $\exists y, z(y \in \Delta(V^{\alpha}) \wedge z \in \bar{B}^{-} \wedge y \lesssim z)$.

Consider firstly $V^{\alpha} \not\models \bar{B}$. There are two possible reasons for this: either $\exists y(y \in \bar{B}^{+} \wedge y \in V^{\beta} \wedge y \notin V^{\alpha}$, but this contradicts the hypothesis that $V^{\beta} \subseteq V^{\alpha}$; or $\exists y(y \in \bar{B}^{-} \wedge y \notin V^{\beta} \wedge y \in V^{\alpha}$, which implies that $y \in \Delta(V^{\beta})$, but this contradicts the selection of the ground clause $x \leftarrow \bar{B}$.

Consider secondly $V^{\alpha} \models \bar{B}$ and $\exists y, z(y \in \Delta(I) \wedge z \in \bar{B}^{-} \wedge y \lesssim z)$. Using the first result for the limit case this implies that $\exists w(w \in \Delta(V^{\beta}) \wedge w \lesssim y \lesssim z)$, which implies that $x \notin \Pi(V^{\beta})$ and because $V^{\beta} \subseteq V^{\beta+1}$ this contradicts the assumption that $x \in V^{\beta+1}$.

This completes the proof of both the induction hypotheses for the limit case. □

Theorem 24. *Given a selection operator V then for all ordinals α, β, $\alpha \leq \beta$ implies $V^{\alpha} \subseteq V^{\beta}$.*

Proof. Do a trans-finite induction on all ordinals using the previous theorem and the definition of V^{α}. □

Theorem 25. *Given a selection operator V then for all ordinals α and a model K of $\mathbf{P}$, $V^{\alpha} \sqsubseteq K$.*

Proof. The proof proceeds by trans-finite induction on α, using the induction hypothesis:

$$\forall x(x \in V^{\alpha} \wedge x \notin K \Rightarrow \exists y(y < x \wedge y \notin V^{\alpha} \wedge y \in K))$$

The result holds trivially for $\alpha = 0$.

For the case when α is a successor ordinal, let $\alpha = \beta + 1$. There will be at least one ground clause $x \leftarrow \bar{B} \in \mathbf{P}^*/I_{\circ}$ where $V^{\beta} \models \bar{B} \wedge \nexists y, z(y \in \Delta(V^{\beta}) \wedge z \in \bar{B}^{-} \wedge y \lesssim z$ and $K \not\models \bar{B}$.

There are two possible conditions where this will hold. Firstly, $y \in \bar{B}^{+} \wedge y \in V^{\beta} \wedge y \notin K$. By the previous theorem this implies $y \in V^{\alpha}$. So by the induction hypothesis $\exists z(z < y \wedge z \notin V^{\alpha} \wedge z \in K$, but $y \lesssim x$ so $z < x$ and z is a witness for y in the induction hypothesis.

Secondly, $y \in \bar{B}^{-} \wedge y \notin V^{\beta} \wedge y \in K$. From causality $y < x$. If $Y \in V^{\alpha}$ then $y \in \Delta(V^{\beta})$, which contradicts the assumption about the rule $x \leftarrow \bar{B}$. So $y \notin V^{\alpha}$, and y satisfies the hypothesis.

For the case when α is a limit ordinal then $V^\alpha = \bigcup_{\beta<\alpha} V^\beta$. There will be at least one ground clause $x \leftarrow \bar{B} \in \mathbf{P}^*/I_\circ$ and ordinal $\beta < \alpha$ where $V^\beta \models \bar{B} \wedge \nexists y, z(y \in \Delta(V^\beta) \wedge z \in \bar{B}^- \wedge y \lesssim z$ and $K \not\models \bar{B}$.

There are two possible conditions where this will hold. Firstly, $y \in \bar{B}^+ \wedge y \in V^\beta \wedge y \notin K$. By the previous theorem this implies $y \in V^\alpha$. So by the induction hypothesis $\exists z(z < y \wedge z \notin V^\alpha \wedge z \in K$, but $y \lesssim x$ so $z < x$ and z satisfies the hypothesis.

Secondly, $y \in \bar{B}^- \wedge y \notin V^\beta \wedge y \in K$. From causality $y < x$. If $y \in V^\alpha$ then $\exists\gamma(\gamma < \alpha \wedge \beta < \gamma$ where $y \notin V^\gamma \wedge y \in \gamma$ thus $y \in \Delta(V^\gamma)$. From the previous theorem this implies $\exists z(z \lesssim y \wedge z \in \Delta(V^\beta)$ which contradicts the assumption about the rule $x \leftarrow \bar{B}$, so $y \notin V^\alpha$ and y satisfies the hypothesis. □

Definition 26. (Chain) *An ordered set C is a* chain *iff $\forall x \in C, y \in C$ either $x \leq y$ or $y \leq x$.*

Definition 27. (CPO) *A set C is a* chain complete partial order *over the ordering $\leq$ if:*

1. *C is partially ordered by $\leq$;*
2. *there is a* bottom element*, $\perp$, such that $\perp \leq x$ for all $x \in C$;*
3. *for all chains $(S_i)_{i\in I}$ there is a least upper bound $lub_{i\in I}(S_i) \in C$.*

Theorem 28. *For a selection operator V there is a least ordinal δ where V^δ is a fixpoint.*

Proof. Construct a CPO using $\subseteq$ as the ordering. Consider the interpretations V^α for all ordinals α. These form a *chain complete partial order* (CPO) using the ordering $\subseteq$ [DP02]. Directly from Theorem 25, V is monotonic on this restricted set. By the Tarski-Knaster theorem [Tar55], V has a least fixpoint on this CPO computed by an ordinal δ. □

Theorem 29. *For a selection operator V with a least fixpoint V^δ*

$$V^\delta = M_{\mathbf{P}}$$

Proof. From theorem 22 V^δ is a model. Also from theorem 25 $V^\delta \sqsubseteq M_{\mathbf{P}}$ but $M_{\mathbf{P}}$ is a minimal model (wrt $\sqsubseteq$) so $V^\delta = M_{\mathbf{P}}$. □

6.1 Strong Causality

The work above has been carried out using only the weak notion of causality. This permits new literals to be added 'at the same time' as other literals which cause them. Strongly causal programs, however, only permit the conclusions to be added at a strictly later time. Assuming strong causality has two advantages: firstly it gives a simpler semantics (shown below) where $M_{\mathbf{P}}$ is the unique model of the programs completion; and secondly it permits a small simplification of the interpreters described later. This is achieved at some cost when writing programs, as it may be necessary to add both parameters and rules in order to

achieve strong causality. For example, the strong causality version of the transitive closure program in Section 4.4 is significantly more complex and difficult to understand than the simple causal version.

We now show that strongly causal programs have only a single model. This provides an exact semantics similar to that for logic programs without negation. It uses the notion of the completion of a program, $comp(\mathbf{P})$, which is defined in [ABW88,Llo87].

Theorem 30. *If the program* $\mathbf{P}$ *is strongly causal then the perfect model* $M_{\mathbf{P}}$ *is a model of* $comp(\mathbf{P})$ *and is the only model of* $comp(\mathbf{P})$.

Proof. $M_{\mathbf{P}}$ is a model of $comp(\mathbf{P})$ [ABW88,Llo87].

Let M, N be models of $comp(\mathbf{P})$. Assume $M \neq N$ and choose a minimal A whose membership of M is different from its membership of N. That is, $A \in M \wedge A \notin N$ or $A \notin M \wedge A \in N$. But from the definition of $comp(\mathbf{P})$ there will be a ground clause $A \leftarrow \bar{B} \in \mathbf{P}^*/I_\circ$ where either $M, I_\circ \models \bar{B}$, and $N, I_\circ \not\models \bar{B}$ or $M, I_\circ \not\models \bar{B}$, and $N, I_\circ \models \bar{B}$. But this imples that there is some member B of $\bar{B}$ where either $B \in M - N$ or $B \in N - M$. However $B < A$ (from strong causality) which contradicts the assumption that A is minimal. That is, the assumption that M and N are different leads to a contradiction. Thus given that $M_{\mathbf{P}}$ is a model of $comp(\mathbf{P})$ it is the only model. □

7 Interpreters

Having established a semantics we will now define a sequence of algorithms for generating the least fixpoint. The algorithms are given both a program, $\mathbf{P}$, and a selection operator V (see Defn. 19). The aim is to produce an efficient algorithm that avoids re-computing earlier results. The selection operator that is used will determine the resource usage of the algorithm and how much potential parallelism is available. We give versions of the algorithm that become successively more explicit and efficient, and we prove their correctness with respect to the semantics.

7.1 Simple Least Fixpoint

The first interpreter (see Fig. 10) is a straightforward implementation of the least-fixpoint procedure that introduces the notation used in the later versions. It uses the following variables (we use the convention that variables that are held over between iterations of the main loop are capitalised (*Gamma*) and those that are local to one iteration of the loop are lower case (*delta*)):

1. *Gamma* - the set of all computed literals. This becomes the fixpoint model of the program $\mathbf{P}$ when the algorithm terminates.
2. α - the number of iterations (used only to provide a link to the correctness results).
3. *new* - a complete recalculation of the current set of results.

1. $\alpha := 0$;
2. $Gamma := \emptyset$;
3. **do**
4. **assert** $Gamma = V^{\alpha}$;
5. $delta := T(Gamma) - Gamma$;
6. **assert** $delta = \Delta(V^{\alpha})$;
7. $new := V(Gamma) - Gamma$;
8. $Gamma := new \cup Gamma$;
9. $\alpha := \alpha + 1$;
10. **until** $delta = \emptyset$;
11. **assert** $Gamma = M_{\mathbf{P}}$;

Fig. 10. Simple Interpreter

4. *delta* - the computed results that have not been seen before, used to detect termination.

Theorem 31. *The assertions in the program are true.*

Proof. See definitions 19(V^{α}), 16(Δ) and the theorems in Section 6. □

7.2 Incremental *Gamma*

The aim of the following interpreters is to avoid as much re-computation of results as possible. In the final version we will recompute both the set *Gamma* and (a variant of) *delta* fully incrementally. To do this it is necessary to generalize some of our earlier definitions to fit in with the new algorithms.

From Defn. 18 the definition of Π is:

$$\Pi(I) = \{A \mid A \leftarrow \bar{B} \in \mathbf{P}^* \wedge (I, I_\circ \models \bar{B}) \wedge \nexists y, z(z \in \bar{B}^{\sim} \wedge y \in \Delta(I) \wedge y \lesssim z)\}$$

This definition references both the set Δ and the negations $\bar{B}^{\sim}$ that occur in the rules. We want to make Π computable directly from Δ, but it does not contain quite enough information as it lacks the information about the negations. To provide this information we define variants of the operators T and Δ that contain both the head of rules and the (ground) negations in the rules. We also define incremental variants of Π and the selection operator V.

Definition 32. (T')

$$\begin{aligned} T'(I) \quad \equiv \quad & \{A \leftarrow \bar{B}^{-} \mid A \leftarrow \bar{B} \in \mathbf{P}^* \wedge (I, I_\circ \models \bar{B}) \\ & \wedge \nexists y, z(z \in \bar{B}^{\sim} \wedge y \in \Delta(I) \wedge y \lesssim z\} \end{aligned}$$

Theorem 33.

$$T(I) = \{A \mid A \leftarrow \bar{B} \in T'(I)\}$$

Proof. Directly from the definitions of T' and T.

Definition 34. (Δ')

$$\Delta'(I) \equiv \{A \leftarrow \bar{B}^- \mid A \leftarrow \bar{B} \in \mathbf{P}^* \wedge (I, I_\circ \models \bar{B}) \wedge A \notin I\}$$

Theorem 35.

$$\Delta(I) = \{A : A \leftarrow \bar{B} \in \Delta'(I)\}$$

Proof. Directly from the definitions of Δ' and Δ.

Π' is defined as an *incremental* version of Π.

Definition 36. (Π')

$$\begin{aligned}\Pi'(I) \equiv \{&A \mid A \leftarrow \bar{B} \in \mathbf{P}^* \wedge (I, I_\circ \models \bar{B}) \\ &\wedge \nexists y, z(z \in \bar{B}^\sim \wedge y \in \Delta(I) \wedge y \lesssim z \wedge A \notin I)\}\end{aligned}$$

Theorem 37.

$$\begin{aligned}\Pi'(I) &= \Pi(I) - I, \quad \textit{and} \\ T(I) \supseteq I &\Rightarrow \Pi(I) = \Pi'(I) \cup I\end{aligned}$$

Proof. Directly from the definitions of Π' and Π.

Note that $T(V^\alpha) \supseteq V^\alpha$, so the above theorem applies to the calculations in the interpreter.

An important result, which enables incremental calculation, is that Π' can be computed using only Δ'.

Theorem 38.

$$\begin{aligned}\Pi'(I) = \{&A \mid A \leftarrow \bar{B} \in \Delta'(I) \\ &\wedge \nexists x, y, z(z \in \bar{B}^\sim \wedge y \leftarrow x \in \Delta'(I) \wedge y \lesssim z)\}\end{aligned}$$

Proof. Directly from the definitions of Π' (Defn. 36), Π (Defn. 18), Δ (Defn. 16) and Δ' (Defn. 34).

Because the theorem above uses only Δ' in the calculation of Π', we define a version of Π, called Π^Δ, that requires only the delta tuples as input, rather than all the delta and gamma tuples.

Definition 39. (Π^Δ)

$$\Pi^\Delta(X) \equiv \{A \mid A \leftarrow \bar{B} \in X \wedge \nexists x, y, z(z \in \bar{B}^\sim \wedge y \leftarrow x \in X \wedge y \lesssim z)\}$$

Theorem 40.

$$\Pi^{\Delta}(\Delta'(I)) = \Pi'(I)$$

Proof. Directly from the definitions of Π' and Π^{Δ}.

Next we define an incremental form of each selection operator V.

Definition 41. (V')

$$V'(I) \;\equiv\; V(I) - I$$

Theorem 42. *If* $V(I) \supseteq I$ *then* $V(I) = V'(I) \cup I$.

Proof. Directly from the definitions of V and V'.

As $V(V^{\alpha}) \supseteq V^{\alpha}$, this theorem applies to the calculations in the interpreters.

Now we further recast the calculation of V' so that it uses Δ' directly. This the efficient incremental form that will eventually be used in the interpreter.

Definition 43. $(V^{\Delta}(I, \Delta'))$ *Given a selection function* V*, a function* $V^{\Delta} : 2^{B_{\mathbf{P}}} \times 2^{B_{\mathbf{P}}} \rightarrow 2^{B_{\mathbf{P}}}$ *is an* incremental delta *version of* V *iff:*

$$\begin{aligned} V^{\Delta}(I, \Delta'(I)) &= V'(I) \\ &= V(I) - I \end{aligned}$$

In general the calculation of $V^{\Delta}(I, X)$ can depend on I, although in practice this seems not to be an interesting or useful thing to do. Usually the calculation need involve only consideration of the second parameter, X, which is $\Delta'(I)$. For example, when the most general selection operator is used $V(I) = \Pi(I)$ and then $V^{\Delta}(I, \Delta'(I)) = \Pi^{\Delta}(\Delta'(I)) = \Pi'(I)$.

Combining these definitions and adapting the previous interpreter we arrive at the interpreter in Fig. 11, which calculates *Gamma* incrementally.

Line 5 of this interpreter uses the definition of Δ' and expands it to an explicit calculation on the set *Gamma*. Note that the expression $I, I_{\circ} \models \bar{B}$ in the definition of $T(I)$ is expanded into explicit conditions on the variable binding θ applied to the rule selected from $\mathbf{P}$. The process of generating the binding θ has not yet been made explicit.

In lines 7-8, *delta* is then used for the incremental calculation of *Gamma* using V^{Δ} (Defn. 43).

Event List There is one selection operator that is of significant interest in practice. It selects all the minimal elements in Δ. This is similar to what is done in discrete event simulation where the lowest event(s) on the current event list are selected next for execution. It is formulated here in its incremental delta form Ev^{Δ}.

Definition 44. (Ev^{Δ})

$$Ev^{\Delta}(I, X) \;\equiv\; \{A \mid A \leftarrow B \in X \wedge \nexists C, D(C \leftarrow D \in X \wedge C < A)\}$$

```
1.   Gamma := ∅ ;
2.   α := 0;
3.   do
4.      assert  Gamma = V^α;
5a.     delta :=
5b.           {(E ← F̄⁻)θ |
5c.            E ← F̄ ∈ P ∧
5d.            F̄⁺θ ⊆ Gamma ∧
5e.            F̄°θ ⊆ I_∘ ∧
5f.            Gamma ∩ F̄~θ = ∅∧
5g.            Eθ ∉ Gamma
5h.           };
6.      assert  delta = Δ'(V^α);
7.      new := V^Δ(Gamma, delta);
8.      Gamma := new ∪ Gamma;
9.      α := α + 1;
10.  until delta = ∅;
11.  assert  Gamma = M_P;
```

Fig. 11. Interpreter that Computes Gamma Incrementally

It is easily verified that for any interpretation I

$$\emptyset \subseteq Ev^{\Delta}(I, \Delta'(I)) \subseteq \Pi^{\Delta}(\Delta'(I)) = \Pi'(I)$$

and hence that the operator $Ev(I) \equiv Ev^{\Delta}(I, \Delta'(I)) \cup I$ is a selection operator.

Ev^{Δ} is interesting for both its simplicity and computational efficiency and its ability to deliver multiple tuples for execution, thus making it suitable for parallel and distributed execution. It also provides a tight coupling between the ordering $<$ and the execution order, which can be useful when resource consumption is important and it is necessary to restrict the amount of parallel execution.

7.3 Incremental *Delta*

Although *Gamma* is now being incrementally calculated, *delta* is still being recomputed from the full set *Gamma* on each iteration. The next version of the interpreter, shown in Fig. 12, is modified so that *delta* is recomputed incrementally from the previous value of *delta*.

The first modification to the previous interpreter maintains *Delta* (now capitalized) between the iterations and computes its initial value on line 2. This computation is a specialization of line 5 of Fig. 11 and explicitly finds all rules that have no positive goals (although they may contain builtin calculations and negations that always succeed because there are no earlier results). This modification also requires a slight re-adjustment of the loop with the check at the top of the loop and a resulting re-arrangement of the calculations.

1. $\alpha := 0$;
2. $Delta := \{(\mathbf{E} \leftarrow \bar{\mathbf{F}}^-)\theta : \mathbf{E} \leftarrow \bar{\mathbf{F}} \in \mathbf{P}, \bar{\mathbf{F}}^+ = \emptyset, \bar{\mathbf{F}}^\circ\theta \subseteq I_\circ\}$;
3. $Gamma := \emptyset$;
4. **while** $Delta \neq \emptyset$ do
5. **assert** $Gamma = \bigcup_{\beta<\alpha} new_\beta = V^\alpha$;
6. **assert** $Delta = \Delta'(V^\alpha)$;
7. $new := V^\Delta(Gamma, Delta)$;
8. **assert** $new = V(V^\alpha) - V^\alpha = V'(V^\alpha)$;
9. $Gamma := Gamma \cup new$;
10. $d_0 := \{A \leftarrow \bar{B} \in Delta \mid A \in new\}$;
11. $d_1 := \{A \leftarrow \bar{B} \in Delta \mid new \cap \bar{B}^\sim \neq \emptyset\}$;

12a. $d_2 := \{(\mathbf{E} \leftarrow \bar{\mathbf{F}}^-)\theta \mid$

12b. $\mathbf{E} \leftarrow \bar{\mathbf{F}} \in \mathbf{P} \wedge$

12c. $\exists F(F \in \bar{\mathbf{F}}^+\theta \cap new \wedge (\bar{\mathbf{F}}^+\theta - \{F\}) \subseteq Gamma) \wedge$

12d. $\bar{\mathbf{F}}^\circ\theta \subseteq I_\circ \wedge$

12e $Gamma \cap \bar{\mathbf{F}}^\sim\theta = \emptyset \wedge$

12f. $\mathbf{E}\theta \notin Gamma$

12g. $\}$;

13. $\alpha := \alpha + 1$;
14. $Delta := (Delta - d_0 - d_1) \cup d_2$;
15. **end while**;
16. **assert** $Gamma = \bigcup_\alpha new_\alpha = M_\mathbf{P}$;

Fig. 12. Interpreter that Computes Delta Incrementally

The core of the incremental calculation is the calculation of *Delta* on lines 10 through 14. Showing the correctness of these lines requires a non-trivial proof (Theorem 46).

The variable *new* is broken out of the incremental calculation of *Gamma*. It holds the items that have been selected from *Delta* as being safe (members of Π) and whch trigger the next round of computation. In the assertions we label the values of the variables by the iteration that they occur in (e.g., new_α is the value assigned to *new* in iteration α). From $V(V^\alpha) \supseteq V^\alpha$ and the assertion $new = V(V^\alpha) - V^\alpha$ the sequence new_α is a disjoint partition of the model $M_{\mathbf{P}}$. So nothing is ever included in *new* more than once. From this it can be seen that a rule $\mathbf{E} \leftarrow \bar{\mathbf{F}} \in \mathbf{P}$ will generate a result $(\mathbf{E} \leftarrow \bar{\mathbf{F}}^-)\theta$ at most once (this follows from the condition $F \in \bar{\mathbf{F}}^+\theta \cap new$).

Of course there can be multiple rules that all give the same answer, this is an efficiency issue for the programmer not the interpreter. Also there can be partial results placed in *Delta* that contain a negation, which are later eliminated on line 11. Again we view this as an issue for the programmer who may be able to manipulate the rules and the ordering so that the negation is resolved early enough to eliminate it before it needs to be stored.

Examination of this interpreter can tell us a lot about its potential efficiency when implemented. Significant experience in implementing versions of this interpreter has been reported [CCPU02,Cla04].

The execution time of line 7 depends on the actual selection operator used. In practice it requires an ordered event list over the set *Delta*. At one extreme, the selection operator can be *Ev* (or a subset), which requires being able to find one or more minimal elements in *Delta*. At the other extreme, when Π is the selection operator the negations in *Delta* can be included in the ordering data structure over *Delta*, allowing a fast check of whether the negations can still potentially fail.

Line 9 is the inclusion of *new* into *Gamma*. *Gamma* will in practice require some form of indexing [Cla04] and this step requires insertion into whatever indexing has been chosen (the indexes may be highly dependent on the structure of the program).

Line 10 (and 14) requires the removal of the selected elements in *new* from *Delta*, which necessitates removal of the new items from the event list.

Line 11 (and 14) requires removal of items from *Delta* whose negations have been selected. The best way of doing this will depend on which selection operator is used. If *Ev* is the selection operator then line 11 can be omitted and replaced by a check that the negations of elements in *new* are not currently in *Gamma*. It is this variant of the interpreter that has been used elsewhere [Cla04].

The calculation in line 12 requires matching atoms in rules to both *new* and *Gamma*. The first of these is on line 12c. For each item in *new* it requires finding a rule that can match that item. This can be done by a static index across the rules, or in many cases by generating explicit code to trigger the execution of the matching rules. The fact that such optimization can be done is crucial for fast execution of Starlog programs.

Lines 12e, 12f and 12g all require finding items in *Gamma*, which are matched against partially instantiated atoms from the current rule. This can be done by providing suitable indexing on *Gamma*.

The following theorems establish the correctness of this interpreter. The main theorem 46 establishes the assertion on line 6, by relating the constructions of lines 10, 11, 12 and 14 back to V'.

Definition 45. (W_α)

$$W_\alpha \;\equiv\; V(V^\alpha) - V^\alpha \quad = \quad V'(V^\alpha)$$

Theorem 46.

$$\begin{aligned}
\Delta'(V^{\alpha+1}) &= \Delta'(V^\alpha) && (1)\\
&- \{A \leftarrow \bar{B} \in \Delta'(V^\alpha) \mid A \in W_\alpha\} && (2)\\
&- \{A \leftarrow \bar{B} \in \Delta'(V^\alpha) \mid W_\alpha \cap \bar{B}^- \neq \emptyset\} && (3)\\
&\cup \{\, (\mathbf{E} \leftarrow \bar{\mathbf{F}}^-)\theta \mid \mathbf{E} \leftarrow \bar{\mathbf{F}} \in \mathbf{P} \wedge \\
&\qquad \exists F(F \in \bar{\mathbf{F}}^+\theta \cap W_\alpha \wedge (\bar{\mathbf{F}}^+\theta - \{F\}) \subseteq V^{\alpha+1}) \wedge \\
&\qquad \bar{\mathbf{F}}^\circ\theta \subseteq I_\circ \wedge V^{\alpha+1} \cap \bar{\mathbf{F}}^\sim\theta = \emptyset \wedge \mathbf{E}\theta \notin V^{\alpha+1}\,\} && (4)
\end{aligned}$$

Proof. First consider the ground clauses $A \leftarrow \bar{B} \in \mathbf{P}^*$ such that $A \leftarrow \bar{B}^- \in \Delta'(V^{\alpha+1})$ and show that they are in the RHS of the equation. From the definition of Δ' recall that

$$\begin{aligned}
& & A &\notin V^{\alpha+1}\\
& \text{and} & I_\circ, V^{\alpha+1} &\models \bar{B}.\\
\text{The latter implies that} & & \bar{B}^+ &\subseteq V^{\alpha+1}\\
& \text{and} & \bar{B}^\sim \cap V^{\alpha+1} &= \emptyset\\
& \text{and} & \bar{B}^\circ &\subseteq I_\circ.
\end{aligned}$$

.

Now consider two cases: (I) $\bar{B}^+ \subseteq V^\alpha$; and (II) $\bar{B}^+ \not\subseteq V^\alpha$

Case (I): From $I_\circ, V^{\alpha+1} \models \bar{B}$ and $\bar{B}^+ \subseteq V^\alpha$ we have $I_\circ, V^\alpha \models \bar{B}$. Also $A \notin V^\alpha \subseteq V^{\alpha+1}$. Combining these results shows that $A \leftarrow \bar{B}^- \in \Delta'(V^\alpha)$, term (1) on the RHS. Also $A \notin W_\alpha$, excluding $A \leftarrow \bar{B}$ from term (2). Finally, $W_\alpha \subseteq V^{\alpha+1}$ and $\bar{B}^\sim \cap V^{\alpha+1} = \emptyset$ so that $W_\alpha \cap \bar{B}^\sim = \emptyset$ and thus $A \leftarrow \bar{B}^-$ is not in term (3).

Case (II): show that $A \leftarrow \bar{B}$ is in term (4) of the RHS. From the premise for this case there must be some $B \in \bar{B}^+$ where $B \in V^{\alpha+1}$ and $B \notin V^\alpha$. This implies that $B \in W_\alpha$. Using the notation of the term (4), there will be a (possibly non-ground) clause $\mathbf{E} \leftarrow \bar{\mathbf{F}} \in \mathbf{P}$, atom $\mathbf{F} \in \bar{\mathbf{F}}^+$, and binding θ such that $A \leftarrow \bar{B} = (\mathbf{E} \leftarrow \bar{\mathbf{F}})\theta$, and $B = \mathbf{F}\theta$ and $\bar{B}^\circ \subseteq I_\circ$. That is, $A \leftarrow \bar{B}$ is included in term (4).

To show the converse we consider all ground clauses $A \leftarrow \bar{B} \in \mathbf{P}^*$ that occur in the RHS, and show that they also occur in the LHS. That is we need to show that a ground clause $A \leftarrow \bar{B}$ satisfies $A \notin V^{\alpha+1}$ and $V^{\alpha+1} \models \bar{B}$.

First, consider the members of the first term on the RHS: $\Delta'(V^\alpha)$. It is sufficient to consider just those members not also in terms (2) or (3). From term (2) $A \notin W_\alpha = V^{\alpha+1} - V^\alpha$. Also $A \leftarrow \bar{B} \in \Delta'(V^\alpha)$ implies $A \notin V^\alpha$. Together these imply $A \notin V^{\alpha+1}$ the first required condition.

$A \leftarrow \bar{B} \in \Delta'(V^\alpha)$ implies $V^\alpha \models \bar{B}$, which implies $V^\alpha \models \bar{B}^+$, and because $V^\alpha \subseteq V^{\alpha+1}$, $V^{\alpha+1} \models \bar{B}^+$. Also $V^\alpha \models \bar{B}^-$, that is $V^\alpha \cap \bar{B}^\sim = \emptyset$, and $I_\circ \models \bar{B}^\circ$. From term (3) $W_\alpha \cap \bar{B}^\sim = \emptyset$, that is, $(V^{\alpha+1} - V^\alpha) \cap \bar{B}^\sim = \emptyset$. Combined with $V^\alpha \cap \bar{B}^\sim = \emptyset$ this implies $V^{\alpha+1} \cap \bar{B}^\sim = \emptyset$, that is $V^{\alpha+1} \models \bar{B}^\sim$. Together these all imply $V^{\alpha+1} \models \bar{B}$, the second required condition.

Second, consider the members of term (4) on the RHS. Using the notation from that term $B = F\theta \in W_\alpha$, that is, $B \notin V^\alpha$ and $B \in V^{\alpha+1}$. Thus $V^{\alpha+1} \models \bar{B}^+$ and from the definition of the term $V^{\alpha+1} \cap \bar{B}^\sim = \emptyset$ and $\bar{B}^\circ \subseteq I_\circ$. Combining these results with $V^{\alpha+1} \models \bar{B}$ establishes the second of the two required conditions. From the last part of the term, we see that $A = E\theta \notin V^{\alpha+1}$, which establishes the first of the two required conditions. □

Theorem 47. *The assertions in the interpreter hold.*

Proof. The assertion $new = V(V^\alpha) - V^\alpha = V'(V^\alpha)$ follows directly from the definition of V'. The two assertions about *Gamma* and *Delta* follow from that and the theorem above. The terminating assertion follows in the event that the while loop finitely terminates when *Gamma* is the least fixpoint of V. □

7.4 Finite Interpreter

The final version of the interpreter in Fig. 13 ensures that all calculations are finite, and as part of this, makes the calculations of bindings explicit.

Finite Builtin Calls So far all the proofs and interpreters have implicitly allowed the *Delta* and *Gamma* sets to be infinite. Such infinite sets can actually occur with rules such as:

$$p(X, Y) \leftarrow X > Y$$

where built in calls return an infinite set of answers. Clearly, in practice this is untenable.

This is dealt with by using a predicate $finiteGoal(\mathbf{B})$, which can be applied to a (possibly non-ground) builtin goal **B**. It should return true only if there are a finite number of possible ground solutions for **B**. It is free to return false if it is ever in doubt and a correct (but not very useful) implementation is to always return false for a non-ground argument. One example technique for arithmetic is to return true whenever the arguments are suitably ground. Thus $finiteGoal(add(X, Y, Z))$ can return true whenever two or more of X, Y, Z are ground. The result of this is that on line 41 in Fig. 13 it is possible to reach a position where there are remaining unresolvable built in calls. This constitutes an error on the programmers part. In many cases it will be possible to statically check that this cannot happen, but in general this runtime check is needed.

2'. $Delta := \bigcup_{\mathbf{E} \leftarrow \bar{\mathbf{F}} \in \mathbf{P} \mid \bar{\mathbf{F}}^+ = \emptyset} lookup(\mathbf{E}, \bar{\mathbf{F}})$;

12'. $d2 := \bigcup_{\mathbf{E} \leftarrow \bar{\mathbf{F}} \in \mathbf{P}} trigger(\mathbf{E}, \bar{\mathbf{F}})$;

20. $trigger(\mathbf{E}, \bar{\mathbf{F}})$:

21. **return** $\bigcup_{\mathbf{F} \in \bar{\mathbf{F}}^+} \left(\bigcup_{\theta \mid \mathbf{F}\theta \in new} lookup(\mathbf{E}\theta, (\bar{\mathbf{F}} - \mathbf{F})\theta) \right)$;

30. $lookup(\mathbf{E}, \bar{\mathbf{F}})$:
31. **if** $\mathbf{E} \in Gamma \rightarrow$
32. **return** $\emptyset$
33. $[]_{\mathbf{F} \in \bar{\mathbf{F}}^\sim}$ $\mathbf{F} \in Gamma \rightarrow$
34. **return** $\emptyset$
35. $[]_{\mathbf{F} \in \bar{\mathbf{F}}^\circ}$ $finiteGoal(\mathbf{F}) \rightarrow$
36. **return** $\bigcup_{\theta \mid \mathbf{F}\theta \in I_\circ} lookup(\mathbf{E}\theta, (\bar{\mathbf{F}} - \mathbf{F})\theta)$
37. $[]_{\mathbf{F} \in \bar{\mathbf{F}}^+}$ $true \rightarrow$
38. **return** $\bigcup_{\theta \mid \mathbf{F}\theta \in Gamma} lookup(\mathbf{E}\theta, (\bar{\mathbf{F}} - \mathbf{F})\theta)$
39. **else** $\rightarrow$
40. **if** $\bar{\mathbf{F}}^\circ = \emptyset$
41. **assert** $ground(\bar{\mathbf{F}}) \wedge \bar{\mathbf{F}} = \bar{\mathbf{F}}^-$
42. **return** $\{\mathbf{E} \leftarrow \bar{\mathbf{F}}\}$
43. **else**
44. **error floundered**
45. **fi**
46. **fi**;

Fig. 13. Finite Interpreter (Lines 2' and 12' replace lines 2 and 12 of Fig 12), and lines 20 onwards are added.

Explicit Bindings The calculation of d_2 on line 12 of Fig. 12 is not explicit about how the bindings are to computed. The final interpreter replaces this with an explicit sequential calculation. It starts by iterating over all rules in the program and calling the method *trigger*. This checks if there are any positive goals in the rule that match against *new*. For each such match a *lookup* is done to evaluate the remaining goals in the rule. A similar call is used to compute the initial *Delta* set on line 2 where the rules with no positive goals are scanned.

lookup is a non-deterministic recursive routine that repeatedly checks the remaining goals against *Gamma* and the built in results. It is written using non-deterministic guards in an **if** of the form $[]\,\alpha \rightarrow \beta$. Any one of the guards (α) that evaluates to true can be selected non-deterministically and the corresponding body (β) is executed. The full form of the guard syntax $[]_{x \in S}\,\alpha \rightarrow \beta$ allows choice over the members x of some set S. Any member of the set S can be selected. This allows flexibility in the order in which goals are checked, matched and evaluated. For example, an implementation might use a strict left-to-right order or a more dynamic run-time selection of the next goal. The **else** guard is true only when all earlier guards are false.

This version of the interpreter makes it clearer how indexing can be used to improve performance. The rule selection in line 12 and the triggering on members of *new* in line 21 can be done by constructing a static index over the positive goals in the rules. This means it is not actually necessary to iterate over all the rules, but rather that a direct selection of both a rule and a suitable positive goal can be done given some member of *new*.

An index over *Gamma* can potentially improve execution speed in the lookup of positive goals in line 37, and in the checks of the negations in line 33 and of the newly generated head tuple in line 31.

Finite Execution To be sure that each iteration of the interpreter terminates with a finite execution a number of assumptions need to be true:

1. *Gamma* and *Delta* are finite;
2. the program consists of a finite number of rules with a finite number of goals;
3. $finitegoal(\mathbf{B})$ is computable in all cases;
4. the calculation of $V^{\Delta}(Gamma, Delta)$ is finite.

Given these assumptions it is elementary to verify that the execution of one iteration is finite and that if these are true at the start of an iteration then *Gamma* and *Delta* will be finite at the start of the next iteration. A couple of these do warrant some commentary.

In an actual system it will be necessary to specify (either directly from the programmer or automatically) an executable version of $X < Y$ or $X \lesssim Y$ (depending on the selection operator). It is outside the scope of this paper to detail how this specification might be done. However, our experiences with various implementations show that it is possible to have powerful classes of orderings that can be efficiently executed [Cla04].

It is possible to concoct a version of V' that is not computable. However, it can be verified from Definitions 18 and 39 that Π^{Δ} is finitely computable (given that *Delta* is finite and that $\lesssim$ is computable for ground arguments). Also, all the different possible versions V' are subsets of Π^{Δ}. The other important selection operator, Ev^{Δ}, (Definition 44), is also finitely computable (given that *Delta* is finite and that $<$ is computable for ground arguments).

7.5 Example Execution

The subsidiary information at http://www.cs.waikato.ac.nz/research/jstar includes both a reference interpreter written in Prolog and example programs and traces of their execution. Three versions of the interpreter are used corresponding to those in Figs. 10, 12 and 14. Multiple execution traces are included for each program in Section 4. These illustrate the effects of the different ordering functions given for the programs and the use of different selection functions including Π, Ev and a subset of Ev that selects one tuple at a time.

8 Conclusions

This paper is intended to be a first step on the way to a programming language that combines the best of logic programming and imperative programming and as well addresses the challenges laid down by the recent switch of performance growth from faster processors to more parallel processors.

Logic programming in the broad sense, encompassing relational databases and their query languages, has been very successful in enterprise computing but has not significantly penetrated the practice of general purpose programming. Its strengths are a strong ability to reason about program correctness and a programming expressiveness that reduces the size of programs and the software engineering burden.

Imperative programming is ubiquitous in general purpose programming. Its perceived strengths are its execution time and memory usage efficiency, together with an ability to reason informally about these resource requirements, interfaces to real time and hardware systems, and large and complex libraries which interface to *de facto* and standards based external systems.

Since 2004, when CPU clock speeds ceased to increase [ABC$^+$06,Osk08], the entire computing world has been forced to confront an increasingly diverse and parallel hardware regime for cost effective and high performance computing. This includes multi-core CPUs, general purpose graphic processing units and circuit based technologies such as FPGAs and ASICs. Unfortunately, existing programming languages and their parallel programming semantics find this regime challenging and expensive. There is evidence, for example, that the whole thrust of hardware development is being called into question [ABC$^+$06] because of the difficulty of solving these software problems.

In the next section we summarize the steps that this paper has taken toward fulfilling these aspirations and then consider the next steps necessary.

8.1 Summary

The first accomplishment of this paper has been the specification of a simple least fixpoint semantics for a pure logic programming language that explicitly incorporates a general ordering across the tuples of the language. We have proved that our fixpoint semantics agrees with the usual perfect model semantics for datalog with negation, and have developed a fully incremental and hence efficient interpreter that implements the semantics. The real importance of this is that we have also demonstrated that this pure logic programming language can directly deal with mutations and updates to data, as well as interfacing with external data streams, without moving outside its pure logical framework.

The potential efficiency of the language is made plausible by the fully incremental interpreter. Other work [Cla04] describes a scheme to automatically select data structures for implementing relational tables. Using the kind of incremental bottom-up fixpoint evaluation as this paper, that work showed that a variety of Starlog benchmark programs could be compiled to code whose execution time was comparable with fully imperative implementations. This was accomplished by automatic estimation of the usage of each relational table within each program, then using selection algorithms to choose efficient representations for each table and each index.

The major technical challenge of this paper has been showing how to use explicit time stamps on all tuples in the program. This allows the data to be held and manipulated in relational tables rather than in the list and functor intensive data structures of classical logic programming. The use of tables, which are highly abstract, permits the efficient manipulation and optimization of the runtime environment.

The execution order is explicitly determined by the ordering between tuples. Thus the base assumption is that execution is parallel unless explicitly constrained by the programmer or by the data causality of the algorithm. This highly parallel basis for execution, together with the ability to retarget the highly abstract data representations of relational tables, makes the language a plausible candidate to address the problems inherent in increasingly diverse and parallel modern computational hardware.

8.2 Related Work

John McCarthy's unpublished Elephant 2000 language proposal [McC92] had several similarities to Starlog. Elephant had the ideas of a time-stamped history, interacting with the real-world via input and output tuples, data-structure-free programming, and a compiler that chooses data structures. Starlog has a more general notion of timestamps (any well-founded partial order), but in many other ways follows a philosophy that is similar to that of Elephant.

An even more similar set of languages is the OverLog, Dedalus and (forthcoming) Bloom languages from the Declarative Networking group at Berkeley [CCHM08,AMC+09]. These are *declarative networking* languages, intended for specifying and implementing distributed protocols and algorithms. OverLog

was based loosely on datalog, but with ad-hoc aspects to its semantics, while Dedalus is closer to pure datalog with negation. Dedalus can be viewed as being a subset of Starlog, where timestamps are restricted to positive integers and rules are restricted so that timestamps can increase only by 0, 1, or an unspecified amount of time for the case where a tuple moves between two different nodes on a distributed network. Programs are also written in a style that explicitly partitions the data tuples across the nodes of a distributed network.

There has been quite a lot of research on bottom-up evaluation strategies for datalog, including naive evaluation, semi-naive evaluation, and pseudo-naive evaluation [SU99], as well as various kinds of top-down tabled evaluation like that used in XSB [ZS03]. Our final interpreter in this paper is more efficient than semi-naive evaluation because it calculates the delta set incrementally, as well as the gamma set, and it generalizes pseudo-naive evaluation by allowing a wide range of selection operators, thus supporting many different parallelization strategies.

There has been much research in the past on parallel implementations of logic programming languages, particularly Prolog [GPA+01]. Most of this work deals with top-down evaluation strategies rather than bottom-up, but some of the underlying techniques will nevertheless be relevant for parallel implementations of Starlog. Zhang *et. al.* describe a bottom-up evaluation strategy that improves on semi-naive evaluation by partitioning the data tuples of a datalog program rather than the rules [ZWC95]. Partitioning the program in this way is similar to the parallel evaluation strategy used by the Berkeley languages, and is one of the parallelization strategies that we plan to use for Starlog.

8.3 Future Work

An implementation of Starlog for sequential execution was been reported in Clayton [Cla04]. This implementation was preliminary, and a number of aspects were incomplete and need further work.

The various interpreters all contain the monotonically increasing set of computed results *Gamma*. In practice it is untenable to retain all tuples as this set may grow unboundedly, so *ad hoc* techniques are used where necessary to get programs to run to completion. We are currently investigating more systematic ways of implementing a correct and efficient garbage collector for Starlog [CU09]. We have defined a logical specification of what garbage collection means in the Starlog context, and described one possible algorithm for garbage collection. Thus, a major piece of work that remains is to implement a garbage collector and to demonstrate that it can achieve sufficient memory compaction sufficiently quickly that practical programs can run to completion.

We expect this to require an investigation of the tradeoffs between execution time, compaction, and the complexity and sophistication of the techniques used. It is also plausible that the user may need to provide guidance to the garbage collector, similar to how programmers can specify the maximum time that tuples should be retained in the Overlog language [LCH+05].

The Starlog system includes a way of specifying the causal orderings. However, experience with using this indicates that it may be overly general. Also it can be wordy for the programmer to specify the ordering, for example, some programs have as many lines devoted to specifying the ordering as to the logic of the code. Further investigation is needed of compact ways of specifying the ordering, balancing the need to allow flexibility and parallelism, as well as ensuring that execution time is not affected by the complexity of the ordering. One interesting possibility is to provide ways of automatically inferring the ordering, similar to the type inference of some programming languages [Pie02].

The system includes ways of specifying the data structures to be used. These can be specified by the user or derived automatically. This is a rich and complex area and much more work can be done on extending the range of underlying data structures that can be used and on techniques for selecting them automatically. One thorny problem here is what to do about situations where the best data structure is data or size dependent.

Implementing this language efficiently on parallel and other special purpose architectures will require a lot of work. Particular problems will be how to partition the data across the distributed resources and the communication algorithms between the partitioned data. We anticipate that, like the data structures, this will require a mixture of user specification and automatic techniques, coupled with performance feedback from actual execution. One extreme challenge is to compile programs to circuit based technologies such as FPGAs and ASICs.

Another area that still needs research and experience with real problems is interfacing Starlog to external interfaces and APIs. Areas of particular interest include: relational databases, file systems, operating systems, and libraries provided by host languages.

Because of the lack of widespread experience with the syntax of logic programs as compared with popular imperative languages such as C or Java we see a need to provide syntactic sugar to ease the transition. Areas of particular promise include: looping constructs, call and return patterns, and assignment.

Given that Starlog has a pure semantics and does not need to step outside them to deal with practical matters, there is an opportunity to use some of the powerful logical tools that this makes possible, including: algorithmic debugging [Sha82], automated unit testing [UL07], integrity constraints [Leu08], and abstract interpretation [Gob08].

However, the most important next step is getting more experience with using the language in a wide range of programs. We need to find out if programs can be run efficiently in practice and if programmers can efficiently write and maintain the programs. We are looking forward to the experience.

A Complete Interpreter

Fig. 14 shows a complete consolidated version of the interpreter of Fig. 12 including the modifications of Fig. 13. This interpreter assumes that the following methods have been provided:

$\alpha := 0;$
$Delta := \bigcup_{\mathbf{E} \leftarrow \bar{\mathbf{F}} \in \mathbf{P} \mid \bar{\mathbf{F}}^+ = \emptyset} lookup(\mathbf{E}, \bar{\mathbf{F}});$
$Gamma := \emptyset\ ;$
while $Delta \neq \emptyset$ do
  assert $Gamma \subseteq \bigcup_{\beta<\alpha} new_\beta = V^\alpha;$
  assert $Delta = \Delta'(V^\alpha);$
  $new := V'(Gamma, Delta);$
  assert $new = V(V^\alpha) - V^\alpha = V'(V^\alpha);$
  $Gamma := keep(Gamma, Delta) \cup new;$
  $d_0 := \{A \leftarrow \bar{B} \in Delta \mid A \in new\};$
  $d_1 := \{A \leftarrow \bar{B} \in Delta \mid new \cap \bar{B}^\sim \neq \emptyset\};$
  $d2 := \bigcup_{\mathbf{E} \leftarrow \bar{\mathbf{F}} \in \mathbf{P}} trigger(\mathbf{E}, \bar{\mathbf{F}});$
  $\alpha := \alpha + 1;$
  $Delta := (Delta - d_0 - d_1) \cup d_2;$
end while;
assert $Gamma \subseteq \left(\bigcup_\alpha new_\alpha\right) = M_{\mathbf{P}};$

$trigger(\mathbf{E}, \bar{\mathbf{F}})$:
  return $\bigcup_{\mathbf{F} \in \bar{\mathbf{F}}^+} \left(\bigcup_{\theta \mid \mathbf{F}\theta \in new} lookup(\mathbf{E}\theta, (\bar{\mathbf{F}} - \mathbf{F})\theta) \right);$

$lookup(\mathbf{E}, \bar{\mathbf{F}})$:
  if $\mathbf{E} \in Gamma \rightarrow$
      return $\emptyset$
  $[]_{\mathbf{F} \in \bar{\mathbf{F}}^\sim}$ $\mathbf{F} \in Gamma \rightarrow$
      return $\emptyset$
  $[]_{\mathbf{F} \in \bar{\mathbf{F}}^\circ}$ $finiteGoal(\mathbf{F}) \rightarrow$
      return $\bigcup_{\theta \mid \mathbf{F}\theta \in I_\circ} lookup(\mathbf{E}\theta, (\bar{\mathbf{F}} - \mathbf{F})\theta)$
  $[]_{\mathbf{F} \in \bar{\mathbf{F}}^+}$ $true \rightarrow$
      return $\bigcup_{\theta \mid \mathbf{F}\theta \in Gamma} lookup(\mathbf{E}\theta, (\bar{\mathbf{F}} - \mathbf{F})\theta)$
  else $\rightarrow$
      if $\bar{\mathbf{F}}^\circ = \emptyset$
        assert $ground(\bar{\mathbf{F}}) \wedge \bar{\mathbf{F}} = \bar{\mathbf{F}}^-$
        return $\{\mathbf{E} \leftarrow \bar{\mathbf{F}}\}$
      else
        error floundered
      fi
  fi;

Fig. 14. Complete Incremental Interpreter (see Figs. 12 and 13).

1. $finitegoal(\mathbf{B})$, which is computable for all (possibly non-ground) goals $\mathbf{B}$;
2. a selection function $V'(Gamma, Delta)$, which is computable for all finite $Gamma$ and $Delta$ (depending on the selection function, this may require that one or other of $\lesssim$ or $<$ be computable for ground arguments).

Acknowledgements

Thanks to the researchers who helped develop earlier versions of Starlog and its semantics, including Xiao Zhonge, Don Smith, Lunjin Lu and Bernhard Pfahringer. Part of this work was supported by the New Zealand Marsden Fund, under grant UOW-605.

References

[ABC+06] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The landscape of parallel computing research: A view from Berkeley. Technical Report UCB/EECS-2006-183, Univ. of California, Berkeley, Dec 2006.

[ABW88] Krzysztof R. Apt, Howard A. Blair, and Adrian Walker. Towards a theory of declarative knowledge. In *Foundations of Deductive Databases and Logic Programming.*, pages 89–148. Morgan Kaufmann, San Francisco, CA, 1988.

[AMC+09] Peter Alvaro, William Marczak, Neil Conway, Joseph M. Hellerstein, David Maier, and Russell C Sears. Dedalus: Datalog in time and space. Technical Report UCB/EECS-2009-173, EECS Department, University of California, Berkeley, Dec 2009.

[Bar43] René Barjavel. *Le Voyageur Imprudent (The Imprudent Traveller).* Arthéme Fayard, France, 1943. Published in the journal *Je suis partout.*

[CCHM08] Tyson Condie, David Chu, Joseph M. Hellerstein, and Petros Maniatis. Evita raced: Metacompilation for declarative networks. In *Proc. of the 34th Int. Conf. on Very Large Data Bases (VLDB), Auckland, NZ*, U.S.A., 2008. Very Large Data Base Endowment Inc.

[CCPU02] Roger Clayton, John Cleary, Bernhard Pfahringer, and Mark Utting. Optimising tabling structures for bottom up logic programming. In *LOPSTR 2002: Preproceedings of the Int. Workshop on Logic Based Prog. Dev. and Transformation, Madrid 17-20 Sep 2002*, pages 57–74, Madrid, Spain, 2002. Facultad de Informática de Madrid.

[CKR73] A. Colmerauer, H. Kanoui, and P. Roussel. Un système de communication homme-machine en francais. Technical report, Groupe de Recherche en Intelligence Artificielle, Université d'Aix-Marseille II, 1973.

[Cla04] Roger Clayton. *Compilation of Bottom-Up Evaluation for a Pure Logic Programming Language.* PhD thesis, Dept. of Computer Science, The Univ. of Waikato, Hamilton, NZ, 2004.

[CM03] W.F. Clocksin and C.S. Mellish. *Programming in Prolog: Using the ISO Standard.* Springer-Verlag, Berlin, fifth edition, 2003.

[Cod70] E. F. Codd. A relational model of data for large shared data banks. *CACM*, 13(6):377–387, 1970.

[CU01] John Cleary and Mark Utting. Verification of Starlog programs. In Grigoris Antoniou and Guido Governatori, editors, *Proceedings of the 2nd Australasian Workshop of Computational Logic, Gold Coast, Australia, January 31 – February 1, 2001*, pages 31–45, Brisbane, Australia, 2001. QUT Printing Services, Queensland University of Technology.

[CU09] John G. Cleary and Mark Utting. A semantics and implementation of a causal logic programming language. Technical Report 01/2009, Department of Computer Science, The University of Waikato, Hamilton, New Zealand, January 2009. Available from http://www.cs.waikato.ac.nz/pubs/wp.

[DM05] Włodzimierz Drabent and Mirosława Miłkowska. Proving correctness and completeness of normal programs – a declarative approach. *Theory Pract. Logic Program.*, 5(6):669–711, 2005.

[DP02] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, Cambridge, UK, second edition, 2002.

[Gob08] Francois Gobert. *Towards Putting Abstract Interpretation of Prolog into Practice*. VDM Verlag, Saarbrücken Germany, 2008.

[GPA+01] Gopal Gupta, Enrico Pontelli, Khayri A.M. Ali, Mats Carlsson, and Manuel V. Hermenegildo. Parallel execution of Prolog programs: a survey. *ACM Trans. Program. Lang. Syst.*, 23(4):472–602, 2001.

[JHM04] Wesley M. Johnston, J. R. Paul Hanna, and Richard J. Millar. Advances in dataflow programming languages. *ACM Comput. Surv.*, 36(1):1–34, 2004.

[Kow74] Robert Kowalski. Predicate logic as programming language. In *Proceedings IFIP Congress, Stockholm*, pages 569–574, Stockholm, Sweden, 1974. North Holland Publishing Co. Reprinted in Computers for Artificial Intelligence Applications, IEEE Computer Society Press, Los Angeles, 1986, pp. 68-73.

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

[Kow01] R. A. Kowalski. Logic programming and the real world. *Logic Programming Newsletter*, 14(1):9–11, February 2001.

[LCH+05] Boon Thau Loo, Tyson Condie, Joseph M. Hellerstein, Petros Maniatis, Timothy Roscoe, and Ion Stoica. Implementing declarative overlays. *SIGOPS Oper. Syst. Rev.*, 39(5):75–90, 2005.

[Leu08] Martin Leucker, editor. *Runtime Verification, 8th International Workshop, RV 2008, Budapest, Hungary, March 30, 2008*, volume 5289 of *LNCS*. Springer-Verlag, Heidelberg, 2008.

[Liu99] Mengchi Liu. Deductive database languages: problems and solutions. *ACM Computing Surveys*, 31(1):27–62, 1999.

[Llo87] J. W. Lloyd. *Foundations of logic programming*. Springer-Verlag, New York, Inc., New York, NY, USA, second edition, 1987.

[McC92] John McCarthy. Elephant 2000. Available from http://www-formal.stanford.edu/jmc/elephant/elephant.html, 1992.

[Osk08] Mark Oskin. The revolution inside the box. *CACM*, 51(7):70–78, July 2008.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, Cambridge, MA, 2002.

[PP90] H. Przymusinska and T. C. Przymusinski. Weakly stratified logic programs. *Fundamenta Informaticae*, 13:51–65, 1990.

[PP96] Alberto Pettorossi and Maurizio Proietti. Rules and strategies for transforming functional and logic programs. *ACM Computing Surveys*, 28(2):360–414, 1996.

[Prz88] Teodor C. Przymusinski. On the declarative semantics of deductive databases and logic programs. In *Foundations of Deductive Databases and Logic Programming.*, pages 193–216. Morgan Kaufmann, San Francisco, CA, 1988.

[Sha82] Ehud Y. Shapiro. Algorithmic program diagnosis. In *POPL '82: Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 299–308, New York, NY, USA, 1982. ACM.

[Sil08] Josep Silva. Debugging techniques for declarative languages: Profiling, program slicing and algorithmic debugging. *AI Commun.*, 21(1):91–92, 2008.

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

[Tar55] A. Tarski. A lattice theoretical fixed point theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.

[UL07] Mark Utting and Bruno Legeard. *Practical Model-Based Testing: A Tools Approach.* Morgan Kaufmann, San Francisco, CA, 2007.

[ZS03] Neng-Fa Zhou and Taisuke Sato. Efficient fixpoint computation in linear tabling. In *PPDP '03: Proceedings of the 5th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 275–283, New York, NY, USA, 2003. ACM.

[ZWC95] Weining Zhang, Ke Wang, and Siu-Cheung Chau. Data partition and parallel evaluation of datalog programs. *IEEE Trans. on Knowledge and Data Engineering*, 7:163–176, 1995.