Working Paper Series
ISSN 1177-777X

A SEMANTICS AND IMPLEMENTATION OF A CAUSAL LOGIC PROGRAMMING LANGUAGE

John G. Cleary, Mark Utting and Roger Clayton

Working Paper: 01/2009
February 11, 2009

©John G. Cleary, Mark Utting and Roger Clayton
Department of Computer Science
The University of Waikato
Private Bag 3105
Hamilton, New Zealand

A Semantics and Implementation of a Causal Logic 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 increasingly widespread availability of multicore and manycore computers demands new programming languages that make parallel programming dramatically easier and less error prone. This paper describes a semantics for a new class of declarative programming languages that support massive amounts of implicit parallelism.

The key idea is that rather than writing low-level imperative programs that define a sequence of state updates, we write a set of high-level rules that are all executed in parallel, acting on a global database of facts. A simple declarative semantics is possible, because rules can add new tuples to the database but cannot modify existing tuples, and because abstract timestamps are used to record causality relationships between tuples.

It turns out that negation and garbage collection are the two crucial features that enable us to recover the efficient mutable updates that are possible in imperative languages. This paper develops the semantics of negation, using a direct least-fix-point (LFP) construction, and shows that this semantics agrees with the well-founded, perfect and stable semantics. The paper develops an efficient bottom-up execution algorithm based directly on the LFP construction. It also gives a declarative formulation of the problem of garbage collection and describes an algorithm for doing garbage collection. Finally it is claimed that the programming language described can form the basis of a practical general purpose programming language.

1 Introduction

Combining logic and programming has a long history. Kowalski [19, 20] and Colmeraur [8] introduced Prolog in the early 1970s. Codd [7] 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 [17]. Relational database theory has also been extended to include deductive databases [22].

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, which as a result of simple semantics can be reasoned about easily. If this agenda is successful then it should be easy to prove programs correct [5, 11], debug programs [31, 32], and transform and modify them while maintaining correctness [26]. The procedural world seeks programs that are efficient, where the programmer can reason about and control resource usage, and where programs can interface with the existing computing milieux.

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 [6]. Kowalski [18] 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.

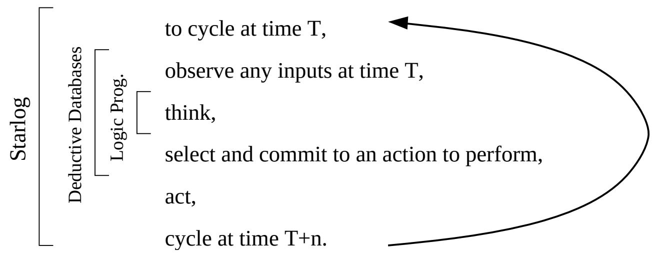

Fig. 1. The observation-thought-action cycle of multi-agent systems (Adapted from [18]).

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 [18].

Over the last decade, we have designed a language, called **Starlog**, that has a simple logical semantics, and which 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 [3].

Recently, we have observed that this style of programming exposes a large amount of potential parallelism, and we have started developing a new language called **JStar** for parallel programming. JStar has the same semantics as Starlog, but will use a more Java-like syntax. It will have a compiler that can transform programs to run efficiently on various parallel architectures (manycore CPUs, cluster computers, GPUs etc.) with good scalability. Our motivation is that the JStar language will be easy to formally reason about and will also be a compact, efficient and powerful programming language which can be easily retargeted to parallel computing platforms.

This paper describes the common semantics of Starlog and JStar – 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 [3].

timestamps: each tuple in a relation has an abstract *timestamp* associated with it. This 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.

causality: a causality relation is defined between timestamps, 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 (together with garbage collection) to enable destructive updating of data within the program, which is important for efficiency.

Section 2 defines syntax and terminology, then Section 3 discusses some small example programs. Section 4 defines a number of terms that are used in Section 5, which contains one of the major contributions of the paper – a direct (and hence potentially efficient) least fix point construction for Horn clause logic including negation. Section 6 presents a series of refinements of the fix point construction, giving more efficient interpreters for the language. Section 7 specifies the garbage collection challenge and Section 7.4 gives an algorithm for garbage collection that satisfies that specification. Finally, Section 8 gives conclusions and further work. The JStar website, http://www.cs.waikato.ac.nz/research/jstar, gives further information about the JStar and Starlog languages.

2 Notation and Definitions

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 [23, 28].

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 [23]. $\mathbf{P}^*$ denotes the *ground instantiation* [23] 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 semantics of *built-in* predicates is represented by the interpretation $I_\circ$.

Definition 1. [Reduction [28]] *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.

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.

Definition 2. [$\lesssim$] *Throughout the paper we will be using a pre-order (reflexive transitive ordering)* $\lesssim$ *on the Herbrand base. In general this will depend on the program* $\mathbf{P}$. *We will also use the strict partial order (irreflexive ordering)* $<$, *defined by:*

$$x < y \;\equiv\; x \lesssim y \wedge \neg(y \lesssim x)$$

and the equivalence relation $\sim$ *which is derived from* $\lesssim$ *as follows:*

$$x \sim y \;\equiv\; x \lesssim y \wedge y \lesssim x$$

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 x, y such that x strictly preceeds y, we have $x < \mathit{not}(x) < y$. This shows that $\mathit{not}(x)$ becomes known *immediately after* the calculation of x has been completed. If that calculation did produce the tuple x, then $\mathit{not}(x)$ is false, whereas if the calculation failed to produce x, then $\mathit{not}(x)$ is true. We use this ordering to define causality relationships between tuples and between rules in our programs.

Definition 3. [Causal] *A program* $\mathbf{P}$ *is* causal *if for every rule instance* $A \leftarrow \bar{B} \in \mathbf{P}^*$

$$\forall B \begin{pmatrix} B \in \bar{B}^{+} \wedge (I_\circ \models \bar{B}^\circ) \Rightarrow B \lesssim A \\ B \in \bar{B}^{\sim} \wedge (I_\circ \models \bar{B}^\circ) \Rightarrow B < A \end{pmatrix}$$

Definition 4. [Strongly Causal] *A program* $\mathbf{P}$ *is* strongly causal *if for every rule instance* $A \leftarrow \bar{B} \in \mathbf{P}^*$

$$\forall B \left(B \in \bar{B}^{+} \cup \bar{B}^{-} \wedge (I_\circ \models \bar{B}^\circ) \Rightarrow B < A \right)$$

The notion of causality is similar to *local stratification* and *weak stratification* which are at the basis of earlier work on the semantics of logic programs [28, 15]. If $\mathbf{P}$ is causal and $\mathbf{P}^*/I_\circ$ is Noetherian (contains no infinite descending chains) then $\mathbf{P}^*/I_\circ$ is locally stratified as defined by Przymusinski [29]. Then $\mathbf{P}^*$ is weakly-stratified [28].

3 Example Programs

This section shows several example Starlog programs, to give an overview of how this style of language can be used for several kinds of numerical and graph calculations. We have also used Starlog for other applications, such as controlling LEGO robots, composing MIDI music files, developing Java Swing GUI programs and graphical animations, but it is beyond the scope of this paper to discuss such applications.

3.1 Finding Prime Numbers

Our first example program generates all the prime numbers upto a given number `MAX`, using the Sieve of Eratosthenes. It generates all multiples of known primes, and uses negation to find numbers that are not multiples, so must be primes. The tuples in this program are all stratified by their first parameter (except for `println`, which uses the last parameter), then by the tuple name. The tuple names are ordered so that `num` and `mult` are before `prime`, and `prime` is before `println`. Note that, although `mult` and `num` are unordered, the rule on lines 04-05 is still stratified, because $M > N$.

```
01: num(2) <-- true.
02: num(N+1) <-- num(N), N < MAX.  % Generate all numbers 2..MAX
```

```

mult(M) <-- num(N), prime(P), N >= P, % Generate multiples of P
            M is N*P, M < MAX.

prime(N) <-- num(N), not(mult(N)).    % Deduce prime numbers

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

If this program is executed with MAX=5000, it produces the following output:

```
prime(2)
prime(3)
prime(5)
prime(7)
prime(11)
prime(13)
...
prime(4999)
```

Note that the predicate `println` is used to communicate results to the outside world. This idiom where output occurs in the *head* of a rule rather than the body may be startling to those used to logic programming but is an important part of preserving purity while interfacing with the real world.

3.2 Transitive Closure of a Graph

This program computes the transitive closure $t(X, Y)$ over a base relation $r(X, Y)$. The ordering is $t(\_,\_) > r(\_,\_)$, that is, all tuples $t(X, Y)$ are greater than all tuples $r(U, V)$. Also $t(\_,\_) \lesssim t(\_,\_)$, that is, all $t(\_,\_)$ tuples order as equal. This version is only causal, not strongly causal.

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

There is no claim that this is an efficient program. It relies for its termination on the fact that if a tuple is generated more than once then it only triggers further computation the first time.

The second version is strongly causal. 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) and $tp(I, X, Y)$ (which indicates a provisional result). Also, explicit code is added to check that tuples computed in earlier iterations are not repeated. This includes the tuple $su(I, X, Y)$, which indicates that a result at time I should be suppressed because there has been an earlier result with the same pair X, Y. It is assumed that the builtin predicates include integer comparison ($>$) and addition.

```
t(X, Y)       <-- tr(_, X, Y)
tp(I+1, X, Y) <-- r(X, Y), tr(I, Z, Y).
su(I, X, Y)   <-- tp(I, X, Y), tr(K, X, Y), I > K.
tr(I, X, Y)   <-- tp(I, X, Y), not(su(I, X, Y)).
tr(0, X, Y)   <-- r(X, Y).
```

This program uses the following causality ordering:

```
  r(_,_) < tr(_,_) < t(_,_).
  tp(I,_,_) < su(I,_,_) < tr(I,_,_).
  tr(K,_,_) < tp(I,_,_) when K < I.
```

3.3 A Running-Maximum Program

The final program incrementally 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. All the tuples are stratified firstly by an integer timestamp (the first parameter of each tuple, except for `println`, where the last parameter is the timestamp), and then by the name of the tuple - these are ordered as follows:

```
  input < val < value_neg < value < assign < println
```

```
01: println(max(T, M), T) <-- 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.
```

In practice, we often write negations like line 14 in a sugared form as

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

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

Here is an example execution with four input numbers arriving 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.

T	input	val	value	assign	println
1	(1,13)	(1, max)	-	(1,max,13)	max(1,13)
4	(4,11)	(4, max)	(4,max,13)	-	-
7	(7,23)	(7, max)	(7,max,13)	(7,max,23)	max(7,23)
10	(10,42)	(10, max)	(10,max,23)	(10,max,42)	max(10,42)

4 Semantic Concepts

This section introduces several operators and relations that are needed to define the semantics of the language. Many of them are taken from the standard literature on the semantics of logic programming languages, but some, like the *selection operators* in Section 4.1 are specific to our language.

Definition 5. **[Preferable [30]]** *For two interpretations I, J, we say that I is* preferable to *J, written $I \sqsubseteq J$, iff*

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

Definition 6. **[Perfect [30]]** *A model M of the program $\mathbf{P}$ is* perfect *iff there is no other model K of $\mathbf{P}$ where $K \sqsubseteq M$.*

We will be considering a number of different operators on the Herbrand universe $V : 2^{B_{\mathbf{P}}} \to 2^{B_{\mathbf{P}}}$. These will include the immediate consequence operator, $T_{\mathbf{P}}$ as well as other selection operators that are related to $T_{\mathbf{P}}$ but are monotone.

Definition 7. **[Monotonic]** *An operator V is monotonic iff*

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

Definition 8. $[V^\alpha]$ *For all ordinals α and transformations V we define $V^\alpha(I)$ as follows:*

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

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

Definition 9. [Immediate consequence operator [23, 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 $I_\circ$:*

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

$T_{\mathbf{P}}(I)$ computes all consequences that are true given an interpretation I. In programs without negation $T_{\mathbf{P}}$ 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 10. [Δ]

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

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

Definition 11. [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 which 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.

Before proceeding it is necessary to put a weak constraint on the program (P) and its ordering $<$.

Definition 12. [Noetherian program] *A program $\mathbf{P}$ and an ordering $<$ together are* Noetherian *iff $\Pi(I) = I$ implies $\Delta(I)$ has no infinite descending chain according to $<$.*

This constraint is a very weak one. For example $\Delta(I)$ is always Noetherian whenever it is finite or the Herbrand Universe itself is Noetherian. In all practical cases where a program runs for only a finite (possibly unbounded) number of steps, $\Delta(I)$ will be finite. In all the following work we will assume that the program and ordering are Noetherian. Of course when computing the semantics of a program it will be necessary to first show that it satisfies the Noetherian condition.

To violate this condition it is necessary for a program to generate an infinite descending chain as it moves forward. The following example shows such a program.

Example:

$p(s(I)) \leftarrow p(I)$
$p(0) \leftarrow$
$q(I) \leftarrow p(I), not(q(s(I))$

together with the ordering

$p(I) \lesssim p(s(I))$
$p(I) \lesssim q(I)$
$q(s(I)) < q(I)$

Calculation from this program gives:

$\Pi^n = \{p(0), p(s(0)), \ldots p(s^n(0))\}$
and $\Pi^\omega = \{p(0), p(s(0)), \ldots\}$

However $\Delta(\Pi^\omega) = \{q(0), q(s(0), \ldots\}$, which contains an infinite descending chain.

4.1 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 flexibility in the choice of triggers may help 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 13. **[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 fix-point.

Fig. 2 illustrates the relationship between the selection operators and Π, Δ and the minimal model $M_{\mathbf{P}}$ defined below.

5 Semantics

In this section we will demonstrate that any selection operator has a least-fixpoint which is equal to the perfect model $M_{\mathbf{P}}$. Often such least fix-points are constructed by showing that the operator is monotone and then applying the Tarski-Knaster theorem. However as the following example shows this approach

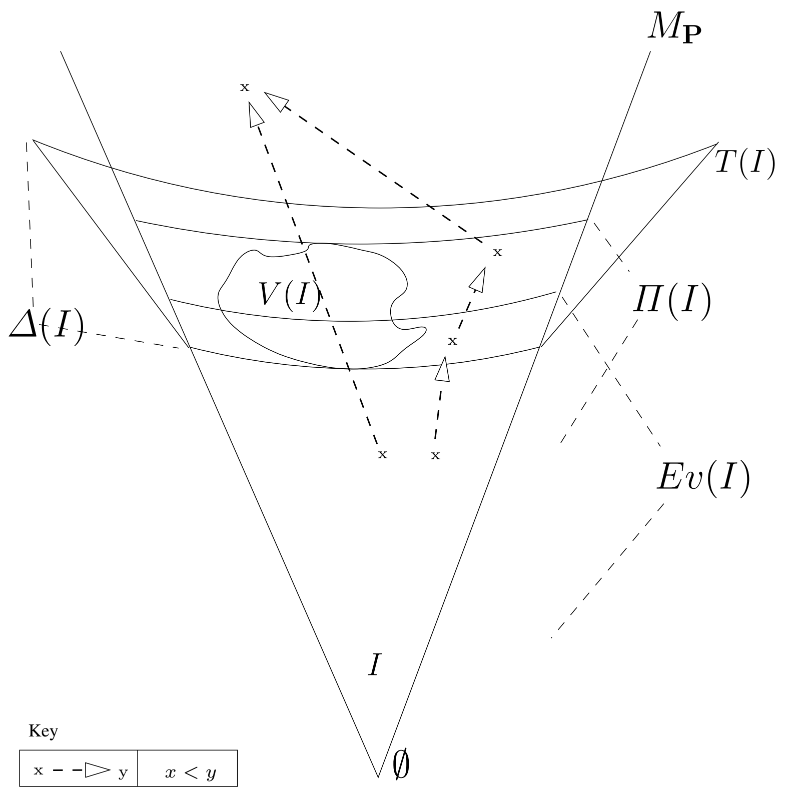

Fig. 2. Selection Operators

cannot be naively followed because Π is not monotone either in the $\subseteq$ ordering or 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 fix-point 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 fix-point and show that it is equal to $M_{\mathbf{P}}$.

Theorem 14. *For a selection operator V, $V(I) = I$ iff $T(I) = I$.*

Proof. Assume $T(I) = I$. From the definition of $\Delta, \Delta(I) = \emptyset$. From the definition of $\Pi, \Pi(I) = T(I) = I$ 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 $\Pi, \Pi(I) \subseteq T(I)$, 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 15. *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.

They 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 16. *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 17. *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 18. [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 19. [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*, $\bot$, such that $\bot \leq x$ 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 20. *For a selection operator V there is a least ordinal δ where V^δ is a fix-point.*

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 $\subseteq$ [10]. Directly from Theorem 17 V is monotonic on this restricted set. By the Tarski-Knaster theorem [33], V has a least fix-point on this CPO computed by an ordinal δ. □

Theorem 21. *From [28]: If the program $\mathbf{P}$ is weakly stratified then there is a unique minimal (under $\sqsubseteq$) perfect model, $M_\mathbf{P}$. $M_\mathbf{P}$ is also well-founded and unique stable.*

Theorem 22. *For a selection operator V with a least fix-point V^δ*

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

Proof. From theorem 14 V^δ is a model. Also from theorem 17 $V^\delta \sqsubseteq M_\mathbf{P}$ but $M_\mathbf{P}$ is a minimal model (wrt $\sqsubseteq$) so $V^\delta = M_\mathbf{P}$. □

5.1 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 below. 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 3 is significantly more complex and difficult to understand than the simple causal version (5 rules versus 2 rules).

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 [1, 23].

Theorem 23. *If the program $\mathbf{P}$ is strongly causal and $B_{\mathbf{P}}$ is Noetherian 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*($\mathbf{P}$) [1, 23].

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. □

Note that this theorem requires a stronger Noetherian condition on the entire Herbrand base not just V^δ.

6 Interpreters

Having established a semantics we will now define a sequence of algorithms for generating the least fix-point. The algorithms are given both a program, $\mathbf{P}$ and a selection operator V (see Defn. 13). 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.

6.1 Simple Least Fixpoint

The first interpreter (see Fig. 3) is a straightforward implementation of the least-fixpoint procedure which 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.

```
$\alpha := 0$;
$Gamma := \emptyset$ ;
do
    assert $Gamma = V^\alpha$;
    $new := V(Gamma)$;
    $delta := new - Gamma$;
    assert $delta = \Delta(V^\alpha)$;
    $Gamma := new$;
    $\alpha := \alpha + 1$;
until $delta = \emptyset$;
assert $Gamma = M_{\mathbf{P}}$;
```

Fig. 3. Simple Interpreter

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

Theorem 24. *The assertions in the program are true.*

Proof. See definitions 13(V^α), 10(Δ) and the theorems in Section 5. □

6.2 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. 11 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 and incremental variants of Π and the selection operator V.

Definition 25. [T']

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

Theorem 26.

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

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

Definition 27. $[\Delta']$

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

Theorem 28.

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

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

Π' is defined as an incremental version of Π.

Definition 29. $[\Pi']$

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

Theorem 30.

$$\Pi'(I) = \Pi(I) - I$$

and if $T(I) \supseteq I$ *then* $\Pi(I) = \Pi'(I) \cup I$.

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

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

In an important result, which enables incremental calculation, Π' can be computed using only Δ'.

Theorem 31.

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

Proof. Directly from the definitions of Π' (Defn. 29), Π (Defn. 11), Δ (Defn. 10) and Δ' (Defn. 27).

Because the theorem above uses only Δ' in the calculation of Π' we can reformulate the calculation in terms of the operator Π''.

Definition 32. $[\Pi'']$

$$\Pi''(\Delta'(I)) \;\equiv\; \Pi'(I)$$

The final redefinition is an incremental form for the selection operators V.

Definition 33. $[V']$

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

Theorem 34. *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. The is the efficient incremental form that will eventually be used in the interpreter.

Definition 35. $[V'(I, \Delta')]$

```
$V'(I, \Delta')$ :
    **if** $\Delta' = \emptyset$ **then**
        **return** $\emptyset$;
    **else**
        **return** *a non-empty subset of* $\Pi''(\Delta')$;
    **fi**;
```

Theorem 36. *The two forms of V' are related as follows:*

$$V'(I) = V'(I, \Delta'(I))$$

In general the calculation of the non-empty subset 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 Δ'. For example, when the most general selection operator is used $V(I) = \Pi(I)$ and then $V'(I, \Delta') = \Pi''(\Delta')$.

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 form Ev'.

Definition 37. [Ev']

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

It is easily verified that for any interpretation I

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

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

Ev' 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 $\lesssim$ and the execution order, which can be useful when resource consumption is important and it is necessary to restrict the amount of parallel execution.

Combining these definitions and adapting the previous interpreter we arrive at the interpreter in Fig. 4, 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.

delta is then used in line 9 for the incremental calculation of *Gamma* using V' (Defn. 33).

```
1.   $Gamma := \emptyset$ ;
2.   $\alpha := 0$;
3.   do
4.       assert $Gamma = V^\alpha$;
5a.      $delta :=$
5b.              $\{(\mathbf{E} \leftarrow \bar{\mathbf{F}}^-)\theta \mid$
5c.                $\mathbf{E} \leftarrow \bar{\mathbf{F}} \in \mathbf{P} \wedge$
5d.                $\bar{\mathbf{F}}^+\theta \subseteq Gamma \wedge$
5e.                $\bar{\mathbf{F}}^\circ\theta \subseteq I_\circ \wedge$
5f.                $Gamma \cap \bar{\mathbf{F}}^\sim\theta = \emptyset \wedge$
5g.                $\mathbf{E}\theta \notin Gamma$
5h.              $\}$;
6.       assert $delta = \Delta'(V^\alpha)$;
9.       $Gamma := V'(Gamma, delta) \cup Gamma$;
10.      $\alpha := \alpha + 1$;
11.  until $delta = \emptyset$;
12.  assert $Gamma = M_\mathbf{P}$;
```

Fig. 4. Interpreter which Computes Gamma Incrementally

6.3 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 in Fig. 5 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.4 and explicitly finds all rules that have no positive goals (although they may contain builtin calculations and negations which 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.

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 39).

The variable *new* is broken out of the incremental calculation of *Gamma*. It holds the items which 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 (eg 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

```
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.       $new := V'(Gamma, Delta);$
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. 5. Interpreter which Computes Delta Incrementally

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 $F \in \bar{\mathbf{F}}^{+}\theta \cap \mathit{new}$).

Of course there can be multiple rules that all give the same answer, this is a problem for the programmer not the interpreter. Also there can be partial results placed in *Delta* which 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 computed early enough to eliminate the result at the point where it is generated.

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 [4, 3].

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 [3] 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 [3].

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 which can match it. This can be done by a static index across the rules or in many cases generating explicit code to call the execution of the rule. 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*, which may be strongly program dependent.

Later we will examine a framework for doing rule dependent optimizations of line 12.

The following theorems establish the correctness of this interpreter.

Definition 38. $[W_\alpha]$

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

Theorem 39.

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

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

$$\begin{aligned}
&& A &\notin V^{\alpha+1}\\
&& V^{\alpha+1} &\models \bar{B}\\
&\text{that is} & \bar{B}^{+} &\subseteq V^{\alpha+1}\\
&\text{and} & \bar{B}^{\sim} \cap V^{\alpha+1} &= \emptyset
\end{aligned}$$

.

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

Case (I): $V^{\alpha} \subseteq V^{\alpha+1}$ so $\bar{B}^{+} \subseteq V^{\alpha}$ and thus $V^{\alpha} \models \bar{B}$. 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 $\mathbf{E} \leftarrow \bar{\mathbf{F}} \in \mathbf{P}$, $\mathbf{F} \in \bar{\mathbf{F}}$, and binding θ where $\bar{B} = \bar{\mathbf{F}}\theta$ and $A \leftarrow \bar{B} = (\mathbf{E} \leftarrow \bar{\mathbf{F}})\theta$. That is, $A \leftarrow \bar{B}$ is included in term (4).

To show the converse consider all ground clauses $A \leftarrow \bar{B} \in \mathbf{P}^*$ which 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 $V^{\alpha} \cap \bar{B}^{\sim} = \emptyset$. 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 $V^{\alpha+1} \cap \bar{B}^{\sim} = \emptyset$. Combining these results $V^{\alpha+1} \models \bar{B}$, the second of the required conditions. Also from the last condition in the term $A = E\theta \notin V^{\alpha+1}$ the first of the required conditions. □

Theorem 40. *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 fix-point of V. □

6.4 Explicit Bindings

```
12a.  $d_2 := \{(\mathbf{E} \leftarrow \bar{\mathbf{F}}^-)\theta \mid$
12b.          $\mathbf{E} \leftarrow \bar{\mathbf{F}} \in \mathbf{P} \wedge$
12c.          $\exists \mathbf{F}, \psi\ ($
12d.            $\mathbf{F} \in \bar{\mathbf{F}}^+ \wedge$
12e.            $\mathbf{F}\psi \in new \wedge$
12f.            $\theta \in conseq(\psi,\ \bar{\mathbf{F}}^+ - \mathbf{F})) \wedge$
12g.          $) \wedge$
12h.          $\bar{\mathbf{F}}^\circ\theta \subseteq I_\circ \wedge$
12i.          $Gamma \cap \bar{\mathbf{F}}^\sim\theta = \emptyset \wedge$
12j.          $\mathbf{E}\theta \notin Gamma$
12k.          $\}$

20.   $conseq(\theta, \bar{\mathbf{F}})$
21.       if $\bar{\mathbf{F}} = \emptyset$ then
22.           return $\{\theta\}$;
23.       else
24.           select some $\mathbf{F} \in \bar{\mathbf{F}}$;
25.           $\Psi := \emptyset$ ;
26.           for each $\psi$ where $\exists D(D \in Gamma \wedge D = \mathbf{F}\theta\psi)$
27.               $\Psi := \Psi \cup conseq(\theta\psi,\ \bar{\mathbf{F}} - \{\mathbf{F}\})$;
28.           end for;
29.           return $\Psi$;
30.       fi;
```

Fig. 6. Explicit Calculation of Bindings

The final version of the algorithm replaces the calculation of d_2 on line 12 of Fig. 5 with an explicit sequential calculation of the binding θ.

The matching of positive goals is done in two places. On line 12b a rule is selected and then on line 12e a chosen positive goal from the rule is matched against the newly computed tuples (we refer to this process as *triggering* and to the matching tuple in *new* as the *trigger*). Later, in the *conseq* function which computes all possible bindings for the selected rule, the remaining positive goals are matched one by one against previously computed tuples in *Gamma*.

This version of the interpreter also makes clearer how indexing can be used to improve performance. The triggering in line 12e can be done by constructing a static index over the positive goals in the rules. Such indexing allows tuples to be immediately paired with an appropriate rule and positive goal as they are placed into *new*.

An index over *Gamma* can potentially improve execution speed in the matching of positive goals in line 26 and in the checks of the negations in line 12i and of the newly generated head tuple in line 12j.

6.5 Optimization

It is possible to significantly optimize the interpreter above using static information from particular programs [3]. These optimizations depend on the orderings between the goals in individual rules and operate by allowing certain important execution steps to be omitted for particular cases.

There are four points in the interpreter where these optimizations can apply:

1. where tuples in *new* trigger the execution of a rule (line 12e Fig. 6 and line 12c Fig. 5)
2. where a lookup is done of tuples in *Gamma* (line 26 Fig. 6)
3. where negated tuples in *new* are checked against the partial rules in *Delta*, by analogy with triggering we refer to this as the *negation trigger* point (line 11 Fig. 5)
4. where negations are checked against *Gamma* when new tuples are being generated, by analogy with the lookup point we refer to this as the *negation lookup* point (line 12i Fig. 6 and line 12e Fig. 5)

The guard conditions for the optimizations are given in terms of two ideas. That of one goal *dominating* another, that is, the dominating goal occurs later than the other in all instantiations of the rule where the builtin goals are true. Domination is extended to a *maximal* goal which dominates all other positive goals.

Definition 41. [Domination]
Given a clause $\mathbf{R} = \mathbf{A} \leftarrow \bar{\mathbf{B}}$
$\mathbf{B} \in \bar{\mathbf{B}}$ *is* dominated *by* $\mathbf{C} \in \bar{\mathbf{B}}$ *iff* $I_\circ \models (\bar{\mathbf{B}}^\circ \Rightarrow \mathbf{B} < \mathbf{C})$.

Definition 42. [Maximal Element]
Given a clause $\mathbf{R} = \mathbf{A} \leftarrow \bar{\mathbf{B}}$
$\mathbf{B} \in \bar{\mathbf{B}}^+$ *is the* maximal element *of* $\mathbf{R}$ *iff* $\forall \mathbf{C}(\mathbf{C} \in (\bar{\mathbf{B}}^+ - \{\mathbf{B}\}) \Rightarrow I_\circ \models (\bar{\mathbf{B}}^\circ \Rightarrow \mathbf{B} > \mathbf{C}))$.

The optimizations are:

1. If a positive goal, **B**, is dominated by another positive goal, **A**, then **B** need never be used as a trigger. This follows because if **B** was used as a trigger then the later lookup of **A** in *Gamma* will fail because it is later than **B** and thus cannot be in *Gamma*. The result of this is to reduce the work done when a tuple matching **B** is placed in *new*.

2. If a negation, *not*(**C**), is dominated by a positive goal, **B**, then *not*(**C**) need not be checked again at the negation trigger point. This follows because **B** is later than **C** and so if **B** is in *Gamma* then it is already known if **C** will be in *Gamma* and the negation lookup is a sufficient check. The result of this is that the goal *not*(**C**) can be omitted from a partial rule when placing it in *Delta*. This reduces the complexity of *Delta* and any indexes over it, as well as reducing execution time at the negation trigger point.
3. If a negation, *not*(**C**), dominates all positive goals then there is no point in checking **C** at the negation lookup point as a tuple matching **C** cannot be in *Gamma* at that point. The result of this is to reduce the amount of computation at the negation lookup point.
4. If the head of a rule, **A**, can only be matched with maximal goals (in other rules) then it need never be added to *Gamma*. The is because it will never be matched at the lookup point only at the trigger point. The result is that any indexing structure over *Gamma* will be simplified, time need not be taken to insert **A** into *Gamma* and the memory used by *Gamma* will be reduced.
5. If the head of a rule, **A**, can only be matched with positive goals that are dominated by another goal then it can never be used at the trigger point. Also if any negative goals in the rule are dominated by a positive goal then **A** will never be part of any partial rule that needs a negation trigger. The result is that **A** need never be inserted into *Delta* but only into *Gamma*. This reduces the complexity of any index over *Delta*, time need not be taken to insert into or retrieve from *Delta* and the memory used by *Delta* is reduced.

The importance of these optimizations is that they are significant steps on the way to demonstrating that Starlog programs can be compiled to execute as efficiently as any other language. In addition, there are many well-known query optimization techniques for relational systems [34, 9] that can be applied to Starlog.

7 Garbage Collection

The various interpreters all contain the monotonically increasing set of computed results *Gamma*. In practice it is untenable to retain all tuples as this may unboundedly increase the memory needed to run a program. There are three reasons why we might want to retain a particular tuple in *Gamma*:

1. because it contributes to the future computation of the program. That is, it may match a positive or negative literal in a rule that may generate more tuples.
2. because it is an externally-visible tuple that must be printed or that causes some other real-world action such as writing to a file or displaying a shape on a screen.
3. because we want to record it for documentation or debugging purposes. For example, a declarative debugger could use *Gamma* to display and analyze the complete execution of the program, or of a particular rule.

In practice, the first reason is the challenging one that leads us to define garbage collection. The externally-visible tuples are typically acted upon at the time that tuples are added to *Gamma*, and it is not necessary to retain them once their external actions have been performed (unless reason 1 also applies). The third reason for retaining tuples is easily satisfied by saving old tuples in *Gamma* into an external file or database for later analysis. So the only tuples in *Gamma* that we must retain during the computation are those that influence the computation of future tuples. The rest can be garbage collected. A sufficiently accurate garbage collection algorithm should allow programs to execute with a memory usage that is proportional to the same program in an imperative language.

One example where garbage collection is particularly important is where the program uses only the most recently-timestamped value of a tuple, and all earlier values can be discarded. Detection of this case can make it possible to update the tuple using a destructive assignment, thus recovering the efficiency of imperative programs.

This garbage collection issue is also encountered in other languages such as Lisp and Java, where data that has no references (or only circular references) is periodically removed by a garbage collection process. Such garbage collection approximates the removal of items that will never be used in the future, as clearly if items have no reference pointers then they will never be used. This is only an approximation as it is possible to have items that will never be used but which are still referenced.

A related problem is that solved by 'fossil collection' in optimistic simulation [16, 12]. There as time advances through the parallel execution of the simulation some items will become no longer referenceable, so can be removed.

The results of this section incorporate aspects of both of these types of collection. The initial results allow for a wide range of different approximations and we anticipate much follow-on work to establish good practical algorithms that balance execution time, complexity and efficiency at removing garbage.

In this section we will first give a logical specification of the set of items that must be retained in *Gamma* at each iteration of the interpreter(s). We will then give an exemplary algorithm that is capable of computing this set, and prove it correct with respect to the specification.

7.1 Definition of Garbage

The definition of the set of items that should be kept, κ, is based on two sets: Γ the set of results which have been computed so far; and Ψ, a superset of the values to be computed in the future. Ψ provides different approximations to the final result $M_{\mathbf{P}}$ and thus leads to different approximations of the set of items that should be kept. The more accurate that Ψ is, the more garbage we can remove.

Definition 43. [Future Set]
Ψ is a future set *of Γ iff* $\Psi \supseteq \{x \mid x \in M_{\mathbf{P}} \land \exists z(z \in \Delta(\Gamma) \land x \gtrsim z)\}$.

Definition 44. [Keep Set] *Given Ψ, a future set of Γ, the* keep set *is defined as:*

$$
\begin{aligned}
\kappa(\Gamma,\Psi) = \{x \mid\ & x \in \Gamma \wedge \\
& \exists A, \bar{B}(A \leftarrow \bar{B} \in \mathbf{P}^* \wedge \\
& \quad A \in \Psi \wedge \\
& \quad \bar{B}^\circ \subseteq I_\circ \wedge \\
& \quad \exists \bar{D}, \bar{E}(\bar{D} \cup \bar{E} = \bar{B}^+ \wedge \bar{D} \cap \bar{E} = \emptyset \wedge \\
& \qquad \bar{D} \subseteq \Gamma \wedge \\
& \qquad \bar{E} \subseteq \Psi \wedge \bar{E} \neq \emptyset \wedge \\
& \qquad ((x \in \bar{D} \cup \{A\} \wedge \bar{B}^\sim \cap \Gamma = \emptyset) \vee \\
& \qquad x \in \bar{B}^\sim \cap \Gamma) \\
& \quad) \\
&) \\
\}\quad &
\end{aligned}
$$

This definition checks each rule in the program looking for tuples in Γ that will be used in future applications of the rule. Such tuples can be either positive goals that will be used to fire future rules, heads that will prevent duplication of results or negated goals that will prevent future rules from firing.

The positive goals in the rules are partitioned into two disjoint sets in all possible ways: $\bar{D}$, the goals which occur in the current Γ and which will need to be kept; and $\bar{E}$, the goals which will occur in the future (this last must be nonempty or else the rule will never fire in the future). If the tuple has negated goals which occur in the current Γ then it is sufficient to keep those and not keep the positive goals or the head (the future rule instance must fail, this can be ensured by keeping the negated tuples, remembering that the other positive tuples may be kept for other reasons in other rules). As the head of the rule, A, can be generated by multiple rules or by different firings of the current rule then it is necessary to also retain A in Γ.

In addition correctness of the rule instance is checked by ensuring that the head lies in the future, $A \in \Psi$, and that all the builtins are correct, $\bar{B}^\circ \subseteq I_\circ$.

Fig. 7 illustrates this definition. It shows a ground instance of a rule $A \leftarrow B_1, B_2, B_3$ where B_3 occurs in the future set (note that $x \lesssim B_3$ where x is in the current Δ). Thus B_1 and B_2 must be kept because they may participate in the future calculation of A. In this case B_3 is in the approximation Ψ but not in the model $M_{\mathbf{P}}$. So if Ψ had been a better approximation to the future computation then keeping this instance could have been avoided and B_1 and B_2 could be garbage collected (provided they did not need to be kept for another reason).

There is a sense in which the definitions above are not optimal, because there are programs where the set of tuples that should be kept is not unique. For example in the following program

$p \leftarrow q$
$p \leftarrow r$

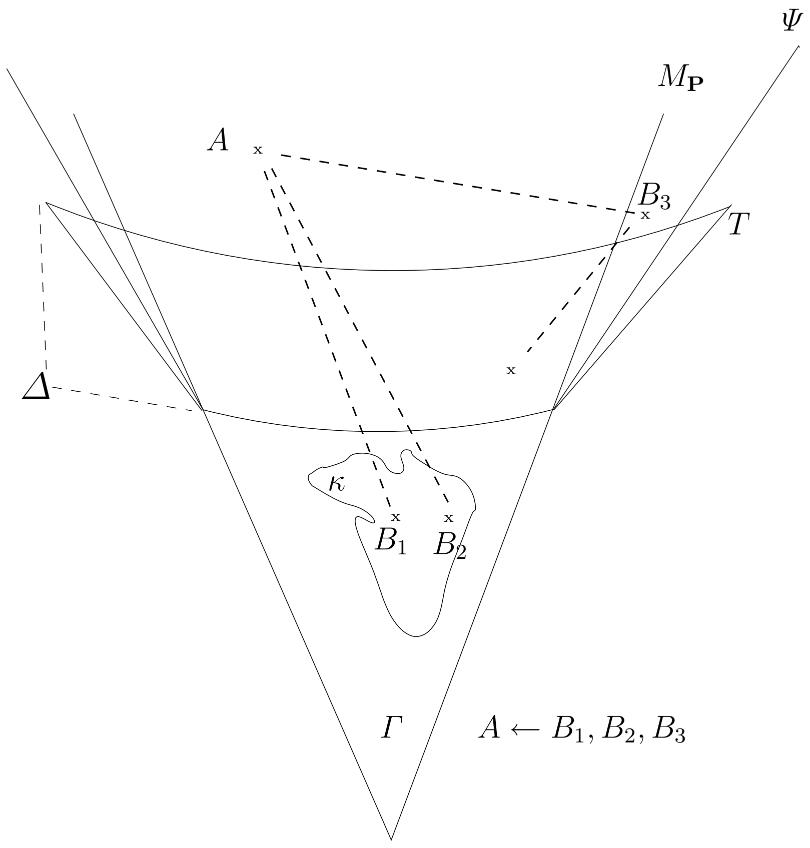

Fig. 7. Garbage Collection

where $q < p$ and $r < p$, if $\Gamma = \{q, r\}$ then either $\{q\}$ or $\{r\}$ could be retained and still lead to the correct model $M = \{p, q, r\}$. The definitions above only approximate this case and both q and r are retained ($\kappa(\{q, r\}, \{p\}) = \{q, r\}$).

Similarly in the following program

$s \leftarrow \neg t, \neg u$

where $t < s$ and $u < s$, if $\Gamma = \{t, u\}$ then either $\{t\}$ or $\{u\}$ could be retained and still lead to the correct model $M = \{t, u\}$. Again the definitions above only approximate this case and both t and u are retained ($\kappa(\{t, u\}, \emptyset) = \{t, u\}$).

It is unclear whether in practice such extra reductions in the size of a kept set would be useful.

One way of obtaining a better approximations for the future computation is to do a greatest fix-point calculation starting from $\Gamma = \Psi \cup Gamma$. Because $M_{\mathbf{P}} \subseteq T(\Gamma)$ then $T(\Gamma) - \Gamma$ can potentially be used as an improved approximation to Ψ. (This is similar to the alternating fix-point used in the calculation of the well-founded semantics [13]). For example, it seems that the reference-following algorithms of classical garbage collection only appear when such a greatest fix-point strategy is pursued. We do not explore this issue further in this paper.

The following theorem shows that future sets are an adequate approximation of future computation when applied to the monotone increasing results generated by a selection operator.

Theorem 45. *For any selection operator V and any Ψ (a future set of V^α):*

$$V^\alpha \cup \Psi \supseteq M_P$$

Proof. Consider some $x \in M_P$. If $x \in V^\alpha$ then the result trivially holds. If $x \notin V^\alpha$ then there will be a $\beta > \alpha$ where $x \in V^{\beta+1}$ and $x \notin V^\beta$ which implies that $x \in \Delta(V^\beta)$. From Theorem 3 this implies that $\exists y (y \in \Delta(V^\alpha) \wedge y \lesssim x)$. The definition of future set then implies $x \in \Psi$. □

7.2 Garbage Collection Algorithm

Fig. 8 shows modifications to the interpreter of Fig. 5 to include garbage collection. Line 9 is modified to include reference to a garbage collection function and because *Gamma* now no longer includes all tuples computed earlier the assertions on lines 5 and 16 are weakened.

Lines 20 through 24 specify a function which is applied on each cycle to (optionally) do garbage collection. The selection of the sets Ψ and *res* are non-deterministic and can cover a range of strategies and implementations. Later we investigate using Δ to compute Ψ and efficient calculation of κ. The most likely choices for *res* are some approximate superset of *mustkeep* when garbage collection is done or *Gamma* when no garbage collection is done.

7.3 Correctness

The basic issue with correctness is to show that the future computation after garbage collection is the same as what would have happened if there was no

```
5'.      assert  $Gamma \subseteq \bigcup_{\beta<\alpha} new_\beta = V^\alpha$;

9'.      $Gamma := keep(Gamma, Delta) \cup new$;

16'. assert  $Gamma \subseteq \bigcup_\alpha new_\alpha = M_{\mathbf{P}}$;

20. $keep(Gamma, Delta)$ :
21.      select $\Psi \supseteq \{x \mid x \in M \wedge \exists z(z \in \Delta(Gamma) \wedge z \lesssim x)\}$;
22.      $mustkeep := \kappa(Gamma, \Psi)$;
23.      select $res$ where $mustkeep \subseteq res \subseteq Gamma$;
24.      return $res$;
```

Fig. 8. Garbage Collection Interpreter - modifications from Fig. 5

garbage collection. The theorem below considers two parallel executions of the algorithm, one with the original line 9 of Fig. 5 and the second using the modified line 9' of Fig. 8. We will use unprimed values for the variables at different iterations for the first algorithm, $Delta_\alpha, new_\alpha, d_{2\alpha}$, and primed versions for the second modified algorithm $Delta'_\alpha, new'_\alpha, d'_{2\alpha}$.

The theorem below shows that the primed and unprimed versions of all the variables except $Gamma_\alpha$ are the same.

Theorem 46. *$\forall\alpha$ the variables $Delta_\alpha$, new_α, $d_{0\alpha}$, $d_{1\alpha}$, $d_{2\alpha}$ are the same as $Delta'_\alpha$, new'_α, $d'_{0\alpha}$, $d'_{1\alpha}$, $d'_{2\alpha}$ respectively and $Gamma_\alpha \supseteq Gamma'_\alpha$.*

Proof. The proof will be by induction on α.

First we observe that if the theorem holds up to β then the variables $new_{\beta+1}$, $d_{0\beta+1}$, $d_{1\beta+1}$ will equal $new'_{\beta+1}$, $d'_{0\beta+1}$, $d'_{1\beta+1}$ respectively. This follows directly from the fact that the calculation of $new_{\beta+1}$ depends only on $Delta_\beta$ and the calculations for $d_{0\beta+1}, d_{1\beta+1}$ depend only on $Delta_\beta$ and $new_{\beta+1}$. Also $Gamma_{\beta+1} \supseteq Gamma'_{\beta+1}$ follows from the induction hypotheses and the equality of $new_{\beta+1}$ and $new'_{\beta+1}$. If the equality of $d_{2\alpha}$ and $d'_{2\alpha}$ can be proven then the equality of $Delta_\alpha$ and $Delta'_\alpha$ follows immediately.

We first assume that the hypothesis holds for all $\beta < \alpha$ and show that assuming $d_{2\alpha} \neq d'_{2\alpha}$ leads to a contradiction. Given some $\mathbf{E} \leftarrow \bar{\mathbf{F}} \in \mathbf{P}$ and an F from line 12c we split into two cases.

Case I: $(\mathbf{E} \leftarrow \bar{\mathbf{F}}^-)\theta \in d_{2\alpha}$ and $(\mathbf{E} \leftarrow \bar{\mathbf{F}}^-)\theta \notin d'_{2\alpha}$.

For this to happen one of the conditions for the calculation of d_2 must fail in the primed case but not the original case. There are three possible sub-conditions that might fail: $(\bar{\mathbf{F}}^+\theta - \{F\}) \subseteq Gamma'_\alpha$, $Gamma'_\alpha \cap \bar{\mathbf{F}}^\sim\theta = \emptyset$, or $\mathbf{E}\theta \notin Gamma'_\alpha$. But because $Gamma_\alpha \supseteq Gamma'_\alpha$ the last two conditions must succeed for the primed case. Thus it is the condition $(\bar{\mathbf{F}}^+\theta - \{F\}) \subseteq Gamma'_\alpha$ that must fail. This implies that there is some $G \in (\bar{\mathbf{F}}^+\theta - \{F\})$

where $G \in Gamma_\alpha$ and $G \notin Gamma'_\alpha$. Thus there will be a $\gamma < \alpha$ where $G \in new_\gamma$ and thus $G \in Gamma_\gamma$ and $G \in Gamma'_\gamma$. Thus there will be an earliest β where $\gamma \leq \beta < \alpha$ where $G \in Gamma_\beta, G \in Gamma'_\beta$ and $G \in Gamma_{\beta+1}, G \notin Gamma'_{\beta+1}$.

We will now show that $G \in \kappa(Gamma'_\beta, \Psi'_\beta)$ for any future set Ψ'_β of $Gamma'_\beta$. and thus that it will be a member of *res* in Fig. 8. (This contradicts the conclusion $G \notin Gamma'_{\beta+1}$). To do this we equate $A \leftarrow \bar{B}$ in the definition of a keep set with $(\mathbf{E} \leftarrow \bar{\mathbf{F}})\theta$ and G with x.

Consider each term of the definition of the keep set and show that each is satisfied for $\kappa(Gamma'_\beta, \Psi'_\beta)$.

From line 12d of the algorithm $\bar{\mathbf{F}}^\circ\theta = \bar{B}^\circ \subseteq I^\circ$.

From lines 9 and 12c of the algorithm $\bar{B}^+ = \bar{\mathbf{F}}^+\theta \subseteq Gamma_\alpha$ and thus because $\beta < \alpha$, $\bar{B}^+ = \bar{\mathbf{F}}^+\theta \subseteq Gamma_\beta \cup \Psi_\beta$. From line 12c $F \in new_\alpha$ and because $\beta < \alpha$ $F \notin Gamma'_\beta$ which implies that $F \in \Psi'_\beta$. Also recall that $G \in Gamma_\beta$ and that $G \in (\bar{\mathbf{F}}^+\theta - \{F\}$. Together these imply that it is possible to partition $\bar{B}^+$ into subsets $\bar{D}$ and $\bar{E}$ where $G \in \bar{D}$, $\bar{D} \subseteq Gamma_\beta$, $\bar{E} \subseteq \Psi'_\beta$ and $F \in \bar{E}$ so that $\bar{E} \neq \emptyset$.

From line 12g $\mathbf{E}\theta = A \in Gamma_{\alpha+1}$ and thus $A \in \Psi'_\beta$.

Finally from line 12e we deduce that $\bar{B}^\sim = \bar{\mathbf{F}}^\sim\theta \cap Gamma_\alpha = \emptyset$ and because $Gamma'_\beta \subseteq Gamma_\beta \subseteq Gamma_\alpha$ then $\bar{B}^\sim \cap Gamma_\beta = \emptyset$. This last implies that $\bar{D}$ is included in κ and thus that $G \in \kappa$.

This completes Case I.

Case II: $(\mathbf{E} \leftarrow \bar{\mathbf{F}}^-)\theta \notin d_{2\alpha}$ and $(\mathbf{E} \leftarrow \bar{\mathbf{F}}^-)\theta \in d'_{2\alpha}$. Following similar logic as for Case I one of the conditions for the calculation of d_2 must fail in the original case but not the primed case. There are two possibilities: the failure of the condition on line 12e, $Gamma_\alpha \cap \bar{\mathbf{F}}^\sim\theta = \emptyset$, or the failure of the condition on line 12f, $\mathbf{E} \notin Gamma_\alpha$.

Consider the first possibility, it implies that there is a some $G \in Gamma_\alpha, G \notin Gamma'_\alpha, G \in \bar{\mathbf{F}}^\sim\theta$. From the induction hypothesis there will be a first β where $G \in new_\beta, G \in Gamma_\beta, G \in Gamma'_\beta, G \in Gamma_{\beta+1}, G \notin Gamma'_{\beta+1}$. Following similar logic logic to case I we will deduce that $G \in \kappa(Gamma'_\beta, \Psi'_\beta)$.

Consider each term of the definition of the keep set and show that each is satisfied for $\kappa(Gamma'_\beta, \Psi'_\beta)$.

From line 12d of the algorithm $\bar{\mathbf{F}}^\circ\theta = \bar{B}^\circ \subseteq I^\circ$.

From lines 9 and 12c of the algorithm $\bar{B}^+ = \bar{\mathbf{F}}^+\theta \subseteq Gamma_\alpha$ and thus because $\beta < \alpha$, $\bar{B}^+ = \bar{\mathbf{F}}^+\theta \subseteq Gamma_\beta \cup \Psi_\beta$. From line 12c $F \in new_\alpha$ and because $\beta < \alpha$ $F \notin Gamma'_\beta$ which implies that $F \in \Psi'_\beta$. Also recall that $G \in Gamma_\beta$ and that $G \in (\bar{\mathbf{F}}^+\theta - \{F\}$. Together these imply that it is possible to partition $\bar{B}^+$ into subsets $\bar{D}$ and $\bar{E}$ where $G \in \bar{D}$, $\bar{D} \subseteq Gamma_\beta$, $\bar{E} \subseteq \Psi'_\beta$ and $F \in \bar{E}$ so that $\bar{E} \neq \emptyset$.

From line 12g $\mathbf{E}\theta = A \in Gamma_{\alpha+1}$ and thus $A \in \Psi'_\beta$.

We know that $G \in Gamma'_\beta$ and thus that $\bar{\mathbf{F}}^\sim\theta \cap Gamma'_\beta \neq \emptyset$ that is $\bar{B}^\sim\theta \cap Gamma'_\beta \neq \emptyset$. which implies that $\bar{B}^\sim\theta \cap Gamma'_\beta$ is included in κ and thus that G is included in κ.

This completes the first possibility for Case II.

Consider the second possibility for Case II and let $G = \mathbf{E}\theta$. This implies that $G \in Gamma_{\alpha}, G \notin Gamma'_{\alpha}$. There will be a first $\beta < \alpha$ where $G \in new_{\beta}$ and thus $G \in Gamma_{\beta}, G \in Gamma'_{\beta}$. There will be a first $\gamma, \beta < \gamma \leq \alpha$ where $G \in Gamma_{\gamma}, G \in Gamma'_{\gamma}, G \in Gamma_{\gamma+1}, G \notin Gamma'_{\gamma+1}$.

We again applying the same logic as case I. This implies that the expression $x \in \bar{D} \cup \{A\} \wedge \bar{B}^{\sim} \cap \Gamma = \emptyset$ is satisfied and that $G = A$ is included in κ.

This completes the second possibility for Case II.

□

7.4 Explicit Calculation of the Keep Set

This section develops one of many possible ways of calculating the keep set. The algorithm given in Fig. 9 explicitly iterates over all the rules and computes the partition of $\bar{\mathbf{B}}^{+}$ into two subsets $\bar{\mathbf{D}}$ and $\bar{\mathbf{E}}$ where the members of $\bar{\mathbf{D}}$ are ground tuples in Γ and the members of $\bar{\mathbf{E}}$ may not be ground and are intended to lie in the future. This done using the recursive procedure *partition*.

Throughout Ψ is a future set for *Gamma*.

Lines 1 through 3 of Fig. 9 iterate over all rules and for each rule call the method *partition*. This method recursively checks each member of $\bar{\mathbf{P}}$ (which was initialized to $\bar{\mathbf{B}}^{+}$ on line 2. For each $\mathbf{B} \in \bar{\mathbf{P}}$ two possibilities are explored. First a series of recursive calls is made to *partition* with each possible binding θ of $\mathbf{B}$ to a member of *Gamma*. In these cases $\mathbf{B}$ is added to $\bar{D}$. These bindings will eventually be placed in the keep set if all other conditions can be meet. Second *partition* will be recursively called with $\mathbf{B}$ placed in $\bar{\mathbf{E}}$ the set of tuples which are to occur in the future. Finally when all members of $\bar{\mathbf{P}}$ have been considered (that is $\bar{\mathbf{B}}^{+}$ has been partitioned into $\bar{D}$ and $\bar{\mathbf{E}}$) a call is made to the *check* method.

The real work of the algorithm is deferred to this procedure *check* which is given an abstract definition in Fig. 10 and whose implementation is deferred to Fig. 11. The abstract definition of *check* exactly reproduces the original definition of $\kappa(Gamma, Delta)$.

In order to construct code for *check* two issues must be dealt with; which future set to use, and how to effectively compute the builtin calls.

The result will be an algorithm which in general returns a superset of the definition of κ. Such an approximation is safe because the algorithm of Fig. 8 (and theorem 46) is happy with *res* being a superset of the keep set.

The first issue is settled by letting the future set Ψ be composed of all tuples that lie in the future of the current *Delta* set. In the algorithm we need to check if a (possibly non-ground) term $\mathbf{E}$ could (after instantiation) lie in this future set. We use the expression *Delta* **mayCause E** to indicate this.

Definition 47. [_ **mayCause** _]

$$Delta\ \mathbf{mayCause}\ \mathbf{E} = \exists x, \theta (x \in Delta \wedge x \lesssim \mathbf{E}\theta)$$

```
0.   $keep(Gamma, Delta)$ :
1.       return $\bigcup_{\mathbf{A} \leftarrow \bar{\mathbf{B}} \in \mathbf{P}} partition(Gamma, Delta, A, \bar{\mathbf{B}}^{+}, \emptyset, \emptyset, \bar{\mathbf{B}}^{\sim}, \bar{\mathbf{B}}^{\circ})$;

10.  $partition(Gamma, Delta, A, \bar{\mathbf{P}}, \bar{D}, \bar{\mathbf{E}}, \bar{\mathbf{N}}, \bar{\mathbf{I}})$ :
11.  if
12.    $[]_{\mathbf{B} \in \bar{\mathbf{P}}}$ $true \rightarrow$
13a.     return $partition(Gamma, Delta, A, \bar{\mathbf{P}} - \mathbf{B}, \bar{D}, \bar{\mathbf{E}} \cup \mathbf{B}, \bar{\mathbf{N}}, \bar{\mathbf{I}}) \cup$
13b.            $\bigcup_{\theta | \mathbf{B}\theta \in Gamma} partition(Gamma, Delta, A, (\bar{\mathbf{P}} - \mathbf{B})\theta, \bar{D} \cup \mathbf{B}\theta, \bar{\mathbf{E}}\theta, \bar{\mathbf{N}}\theta, \bar{\mathbf{I}}\theta)$
14.  else
15.      return $check(Gamma, Delta, A, \bar{D}, \bar{\mathbf{E}}, \bar{\mathbf{N}}, \bar{\mathbf{I}})$;
16.  fi
```

Fig. 9. Calculation of Keep Set

```
$check(Gamma, Delta, A, \bar{D}, \bar{\mathbf{E}}, \bar{\mathbf{N}}, \bar{\mathbf{I}})$ :
 $= \{x \mid x \in Gamma \wedge$
      $\exists \theta(A\theta \in \Psi \wedge$
        $\bar{\mathbf{I}}\theta \subseteq I_{\circ} \wedge$
        $\bar{\mathbf{E}} \subseteq \Psi \wedge$
        $((x \in \bar{D} \cup \{A\} \wedge \bar{\mathbf{N}}\theta \cap \Gamma = \emptyset) \vee x \in \bar{\mathbf{N}}\theta \cap \Gamma)$
      $)$
    $\}$
```

Fig. 10. Abstract Definition of check

Naively this would require matching each member of *Delta* with $\mathbf{E}$. However, as in the interpreters, this can be optimized by a suitable index over *Delta*. Also it is possible to be optimistic about this and return true even if it is not clear that there is a suitable x or θ. In fact a correct (but not perhaps useful) implementation would be to always return true. A full investigation of implementation techniques for this is not undertaken in this paper.

The second issue is that the algorithm in Fig. 9 may leave the negated goals $\bar{\mathbf{N}}$ and the builtin goals $\bar{\mathbf{I}}$ only partially bound. The precise check implied by the definitions of κ and *check* requires that they be fully grounded. In the case of $\bar{\mathbf{N}}$ this can be done by matching against *Gamma*. However for the builtins this cannot necessarily be done finitely. The problem with the builtins is that it is easy to construct a set of builtin calls (eg simple arithmetic, say $X > Y$) that have an infinite set of solutions. In the actual execution of the program this is dealt with by placing restrictions on the rules so that they have only a finite number of solutions and by saying that it is the programmer's responsibility to ensure that they can be effectively computed. However, we must be more careful here as we will not necessarily be grounding all of the positive goals (some are in $\bar{\mathbf{E}}$ which is in the future set which can be infinite). Thus a builtin calculation that is perfectly well-behaved during normal execution might yield an infinite number of solutions when checked by the garbage collector.

This is dealt with by using a predicate $\mathit{finiteGoal}(\mathbf{B})$ which can be applied to a (possibly non-ground) builtin goal $\mathbf{B}$. It should return true only if there are a finite number of possible ground solutions for $\mathbf{B}$. It is free to return false if it is ever in doubt and a correct (but probably not useful) implementation is to always return false. One example technique for arithmetic is to return true whenever the arguments are suitably ground. Thus $\mathit{finiteGoal}(\mathit{add}(X, Y, Z))$ can return true whenever two or more of X, Y, Z are ground. We do not pursue this discussion further in this paper.

In Fig. 11 two methods *check* and *negations* are used recursively to ensure that the members of $\bar{\mathbf{E}}$ lie in the future (lines 2, 3 and 22, 23) and that the builtin calls have been satisfied (lines 4, 5 and 28, 29).

check recursively checks the future values and whenever possible evaluates builtin goals until no further execution or future checks are possible. Then it checks if any of the negations, $\bar{\mathbf{N}}$, are in *Gamma* (recall that this implies that they are ground). If this is so then we are guaranteed that the condition $\bar{\mathbf{N}}\theta \cap \Gamma = \emptyset$ in the expression $((x \in \bar{D} \cup \{A\} \wedge \bar{\mathbf{N}}\theta \cap \Gamma = \emptyset) \vee x \in \bar{\mathbf{N}}\theta \cap \Gamma)$ in Fig. 10 will fail and thus the terms $\bar{D} \cup \{A\}$ can be omitted from the result (see also $((x \in \bar{D} \cup \{A\} \wedge \bar{B}^{\sim} \cap \Gamma = \emptyset) \vee x \in \bar{B}^{\sim} \cap \Gamma)$ in Definition 44).

In either case possible values for the terms $\bar{\mathbf{N}}\theta \cap \Gamma$ may need to be included. This possibility is explored by the recursive procedure *negations*. It ensures that four conditions hold by the time it returns a result. It checks that the members of $\bar{\mathbf{E}}$ always lie in the future (lines 22 and 23). It checks that any safe builtins are executed (line 28 and 29). It omits any ground negations that are not in *Gamma* from the results (lines 24 and 25). It ensures that all the negations are

```
1.  $check(Gamma, Delta, A, \bar{D}, \bar{\mathbf{E}}, \bar{\mathbf{N}}, \bar{\mathbf{I}})$ :
2.  **if** $\exists \mathbf{E}(\mathbf{E} \in \bar{\mathbf{E}} \wedge \neg Delta$ **mayCause** $\mathbf{E}) \rightarrow$
3.      **return** $\emptyset$;
4.   $\underset{\mathbf{I}\in\bar{\mathbf{I}}}{[]}\ finiteGoal(\mathbf{I}) \rightarrow$
5.      **return** $\bigcup_{\theta|\mathbf{I}\theta\in I_\circ} check(Gamma, Delta, A, \bar{D}, \bar{\mathbf{E}}\theta, \bar{\mathbf{N}}\theta, (\bar{\mathbf{I}} - \mathbf{I})\theta)$;
6.  **else**
7.      **if** $\exists N(N \in \bar{\mathbf{N}} \wedge N \in Gamma)$
8.          **return** $negations(Gamma, Delta, \bar{\mathbf{E}}, \bar{\mathbf{N}}, \bar{\mathbf{I}})$;
9.      **else**
10.         **return** $\bar{D} \cup \{A\} \cup negations(Gamma, Delta, \bar{\mathbf{E}}, \bar{\mathbf{N}}, \bar{\mathbf{I}})$;
11.     **fi**
12. **fi**

21. $negations(Gamma, Delta, \bar{\mathbf{E}}, \bar{\mathbf{N}}, \bar{\mathbf{I}})$ :
22. **if** $\exists \mathbf{E}(\mathbf{E} \in \bar{\mathbf{E}} \wedge \neg Delta$ **mayCause** $\mathbf{E}) \rightarrow$
23.     **return** $\emptyset$;
24.  $\underset{\mathbf{N}\in\bar{\mathbf{N}}}{[]}\ ground(\mathbf{N}) \wedge \mathbf{N} \notin Gamma \rightarrow$
25.     **return** $negations(Gamma, Delta, \bar{\mathbf{E}}, \bar{\mathbf{N}} - \mathbf{N}, \bar{\mathbf{I}})$;
26.  $\underset{\mathbf{N}\in\bar{\mathbf{N}}}{[]}\ \neg ground(\mathbf{N}) \rightarrow$
27.     **return** $\bigcup_{\theta|\mathbf{N}\theta\in Gamma} negations(Gamma, Delta, \bar{\mathbf{E}}\theta, \bar{\mathbf{N}}\theta, \bar{\mathbf{I}}\theta)$;
28.  $\underset{\mathbf{I}\in\bar{\mathbf{I}}}{[]}\ finiteGoal(\mathbf{I}) \rightarrow$
29.     **return** $\bigcup_{\theta|\mathbf{I}\theta\in I_\circ} negations(Gamma, Delta, \bar{\mathbf{E}}\theta, \bar{\mathbf{N}}\theta, (\bar{\mathbf{I}} - \mathbf{I})\theta)$;
30. **else**
31.     **return** $\bar{\mathbf{N}}$;
32. **fi**
```

Fig. 11. Calculation of check

ground by taking any that are not and binding them in turn to all matching terms in *Gamma* (lines 26 and 27).

The overall algorithm is an approximation because there may be builtins that never become safe and so are never checked. This will result in more terms being returned than are required by the abstract definition.

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 the clock speeds of all major CPU families ceased to increase [2, 25] 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 [2] because of the difficulty of solving these 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 fix-point semantics for a pure logic programming language that explicitly incorporates a general ordering across the tuples of the language. This semantics has then been modified to give a fully incremental and hence efficient interpreter. 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 incremental interpreter, by demonstration of the feasibility of garbage collection, by the discussion of program specific optimizations, and by reference to other work [3] where these have been used together with techniques to automatically select data structures for implementing relational tables. 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. Such time stamps permit the data in the program to be updated and garbage collected. 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 good candidate to address the problems inherent in increasingly diverse and parallel modern computational hardware.

8.2 Future Work

An implementation of Starlog for sequential execution has been reported in [3]. This implementation is preliminary and a number of aspects were incomplete and need further work.

The major lack in the system was that no automatic garbage collection was built into the system. At the time the system was written the theoretical underpinnings of garbage collection were not understood and *ad hoc* techniques were used where necessary to get programs to run to completion. Thus, a major piece of work that needs to be done 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 P2 dataflow language [24].

The Starlog system includes a way of specifying the ordering. 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 [27].

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 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 implementation is to map programs to circuit based technologies such as FPGAs and ASICs. This will require separating the static logic, which can be mapped directly to circuits, from the data, which needs to be mapped to RAM or other external memory.

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 [31], automated unit testing [35], integrity constraints [21], and abstract interpretation [14].

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.

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

1. 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, 1988.
2. 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, EECS Department, University of California, Berkeley, Dec 2006.
3. Roger Clayton. *Compilation of Bottom-Up Evaluation for a Pure Logic Programming Language*. PhD thesis, Department of Computer Science, The University of Waikato, Hamilton, New Zealand, 2004.
4. Roger Clayton, John Cleary, Bernhard Pfahringer, and Mark Utting. Optimising tabling structures for bottom up logic programming. In Michael Leuschel and Francisco Bueno, editors, *LOPSTR 2002: Preproceedings of the International Workshop on Logic Based Program Development and Transformation, Madrid 17-20 Sep 2002*, pages 57–74. Facultad de Informática de Madrid, 2002.
5. 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. QUT Printing Services, Queensland University of Technology, 2001.
6. W.F. Clocksin and C.S. Mellish. *Programming in Prolog: Using the ISO Standard*. Springer, fifth edition edition, 2003.
7. E. F. Codd. A relational model of data for large shared data banks. *CACM*, 13(6):377–387, 1970.
8. 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.
9. Tyson Condie, David Chu, Joseph M. Hellerstein, and Petros Maniatis. Evita raced: Metacompilation for declarative networks. In *Proceedings of the 34th International Conference on Very Large Data Bases (VLDB), Auckland, New Zealand*, 2008.
10. B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, second edition, 2002.
11. Włodzimierz Drabent and Mirosława Miłkowska. Proving correctness and completeness of normal programs – a declarative approach. *Theory Pract. Log. Program.*, 5(6):669–711, 2005.
12. R. M. Fujimoto. Parallel discrete event simulation. In *WSC '89: Proceedings of the 21st conference on Winter simulation*, pages 19–28, New York, NY, USA, 1989. ACM.
13. Allen Van Gelder. The alternating fixpoint of logic programs with negation. In *PODS '89: Selected papers of the eighth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 185–221, San Diego, CA, USA, 1993. Academic Press Professional, Inc.
14. Francois Gobert. *Towards Putting Abstract Interpretation of Prolog into Practice*. VDM Verlag, 2008.
15. P. Hitzler and M. Wendt. A uniform approach to logic programming semantics. *Theory and Practice of Logic Programming*, 5(1-2):93–121, January 2005.

16. David R. Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems*, pages 404–425, 1985.
17. Wesley M. Johnston, J. R. Paul Hanna, and Richard J. Millar. Advances in dataflow programming languages. *ACM Comput. Surv.*, 36(1):1–34, 2004.
18. R. A. Kowalski. Logic programming and the real world. *Logic Programming Newsletter*, 14(1):9–11, February 2001.
19. Robert Kowalski. Predicate logic as programming language. In *Proceedings IFIP Congress, Stockholm*, pages 569–574. North Holland Publishing Co., 1974. Reprinted in Computers for Artificial Intelligence Applications, IEEE Computer Society Press, Los Angeles, 1986, pp. 68-73.
20. Robert Kowalski. Algorithm = Logic + Control. *Communications of the ACM*, 22(7):424–436, July 1979.
21. Martin Leucker, editor. *Runtime Verification, 8th International Workshop, RV 2008, Budapest, Hungary, March 30, 2008*, volume 5289 of *LNCS*. Springer-Verlag, Heidelberg, 2008.
22. Mengchi Liu. Deductive database languages: problems and solutions. *ACM Computing Surveys*, 31(1):27–62, 1999.
23. J. W. Lloyd. *Foundations of logic programming.* Springer-Verlag, New York, Inc., New York, NY, USA, second edition, 1987.
24. 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.
25. Mark Oskin. The revolution inside the box. *CACM*, 51(7):70–78, July 2008.
26. Alberto Pettorossi and Maurizio Proietti. Rules and strategies for transforming functional and logic programs. *ACM Computing Surveys*, 28(2):360–414, 1996.
27. Benjamin C. Pierce. *Types and Programming Languages.* The MIT Press, 2002.
28. H. Przymusinska and T. C. Przymusinski. Weakly stratified logic programs. *Fundamenta Informaticae*, 13:51–65, 1990.
29. T. C. Przymusinski. Every logic program has a natural stratification and an iterated least fixed point model. In *Proceedings of the eighth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 11–21. ACM Press, 1989.
30. 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, 1988.
31. 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.
32. Josep Silva. Debugging techniques for declarative languages: Profiling, program slicing and algorithmic debugging. *AI Commun.*, 21(1):91–92, 2008.
33. A. Tarski. A lattice theoretical fixed point theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.
34. J. D. Ullman. *Principles of Database and Knowledge-Based Systems: Volume II: The New Technologies.* W. H. Freeman, New York, NY, 1990.
35. Mark Utting and Bruno Legeard. *Practical Model-Based Testing: A Tools Approach.* Elsevier, 2007.

A Complete Interpreter

The following is a complete consolidated version of the interpreter of Fig. 5 including the modifications of Fig. 6, the addition of garbage collection, Fig. 8, and the keep function of Figs. 9 and 11.

```
$\alpha := 0$;
$Delta := \{(\mathbf{E} \leftarrow \bar{\mathbf{F}}^-)\theta : \mathbf{E} \leftarrow \bar{\mathbf{F}} \in \mathbf{P}, \bar{\mathbf{F}}^+ = \emptyset, \bar{\mathbf{F}}^\circ\theta \subseteq I_\circ\}$;
$Gamma := \emptyset$ ;
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 $new = V(V^\alpha) - V^\alpha = V'(V^\alpha)$;
    $Gamma := keep(Gamma, Delta) \cup new$;
    $d_0 := \{A \leftarrow \bar{B} \in Delta \mid A \in new\}$;
    $d_1 := \{A \leftarrow \bar{B} \in Delta \mid new \cap \bar{B}^\sim \neq \emptyset\}$;
    $d_2 := \{(\mathbf{E} \leftarrow \bar{\mathbf{F}}^-)\theta \mid$
            $\mathbf{E} \leftarrow \bar{\mathbf{F}} \in \mathbf{P} \wedge$
            $\exists \mathbf{F}, \psi$ (
              $\mathbf{F} \in \bar{\mathbf{F}}^+ \wedge$
              $\mathbf{F}\psi \in new \wedge$
              $\theta \in conseq(\psi,\ \bar{\mathbf{F}}^+ - \mathbf{F})) \wedge$
            $) \wedge$
            $\bar{\mathbf{F}}^\circ\theta \subseteq I_\circ \wedge$
            $Gamma \cap \bar{\mathbf{F}}^\sim\theta = \emptyset \wedge$
            $\mathbf{E}\theta \notin Gamma$
          $\}$
    $\alpha := \alpha + 1$;
    $Delta := (Delta - d_0 - d_1) \cup d_2$;
end while;
assert   $Gamma \subseteq \bigcup_\alpha new_\alpha = M_\mathbf{P}$;

    $conseq(\theta, \bar{\mathbf{F}})$
        if $\bar{\mathbf{F}} = \emptyset$ then
            return $\{\theta\}$;
        else
            select some $\mathbf{F} \in \bar{\mathbf{F}}$;
            $\Psi := \emptyset$ ;
            for all $\psi$ where $\exists D(D \in Gamma \wedge D = \mathbf{F}\theta\psi)$
                $\Psi := \Psi \cup conseq(\theta\psi,\ \bar{\mathbf{F}} - \{\mathbf{F}\})$;
            end for;
            return $\Psi$;
        fi;

    $keep(Gamma, Delta)$ :
        return $\bigcup_{\mathbf{A} \leftarrow \bar{\mathbf{B}} \in \mathbf{P}} partition(Gamma, Delta, A, \bar{\mathbf{B}}^+, \emptyset, \emptyset, \bar{\mathbf{B}}^\sim, \bar{\mathbf{B}}^\circ)$;
```

```
$partition(Gamma, Delta, A, \bar{\mathbf{P}}, \bar{D}, \bar{\mathbf{E}}, \bar{\mathbf{N}}, \bar{\mathbf{I}})$ :
if
 $\underset{\mathbf{B}\in\bar{\mathbf{P}}}{[]}$ $true \rightarrow$
    return $partition(Gamma, Delta, A, \bar{\mathbf{P}} - \mathbf{B}, \bar{D}, \bar{\mathbf{E}} \cup \mathbf{B}, \bar{\mathbf{N}}, \bar{\mathbf{I}}) \cup$
        $\bigcup_{\theta|\mathbf{B}\theta\in Gamma} partition(Gamma, Delta, A, (\bar{\mathbf{P}} - \mathbf{B})\theta, \bar{D} \cup \mathbf{B}\theta, \bar{\mathbf{E}}\theta, \bar{\mathbf{N}}\theta, \bar{\mathbf{I}}\theta)$
else
    return $check(Gamma, Delta, A, \bar{D}, \bar{\mathbf{E}}, \bar{\mathbf{N}}, \bar{\mathbf{I}})$;
fi

$check(Gamma, Delta, A, \bar{D}, \bar{\mathbf{E}}, \bar{\mathbf{N}}, \bar{\mathbf{I}})$ :
if $\exists\mathbf{E}(\mathbf{E} \in \bar{\mathbf{E}} \wedge \neg Delta$ mayCause $\mathbf{E}) \rightarrow$
    return $\emptyset$;
 $\underset{\mathbf{I}\in\bar{\mathbf{I}}}{[]}$ $finiteGoal(\mathbf{I}) \rightarrow$
    return $\bigcup_{\theta|\mathbf{I}\theta\in I_\circ} check(Gamma, Delta, A, \bar{D}, \bar{\mathbf{E}}\theta, \bar{\mathbf{N}}\theta, (\bar{\mathbf{I}} - \mathbf{I})\theta)$;
else
    if $\exists N(N \in \bar{\mathbf{N}} \wedge N \in Gamma)$
        return $negations(Gamma, Delta, \bar{\mathbf{E}}, \bar{\mathbf{N}}, \bar{\mathbf{I}})$;
    else
        return $\bar{D} \cup \{A\} \cup negations(Gamma, Delta, \bar{\mathbf{E}}, \bar{\mathbf{N}}, \bar{\mathbf{I}})$;
    fi
fi

$negations(Gamma, Delta, \bar{\mathbf{E}}, \bar{\mathbf{N}}, \bar{\mathbf{I}})$ :
if $\exists\mathbf{E}(\mathbf{E} \in \bar{\mathbf{E}} \wedge \neg Delta$ mayCause $\mathbf{E}) \rightarrow$
    return $\emptyset$;
 $\underset{\mathbf{N}\in\bar{\mathbf{N}}}{[]}$ $ground(\mathbf{N}) \wedge \mathbf{N} \notin Gamma \rightarrow$
    return $negations(Gamma, Delta, \bar{\mathbf{E}}, \bar{\mathbf{N}} - \mathbf{N}, \bar{\mathbf{I}})$;
 $\underset{\mathbf{N}\in\bar{\mathbf{N}}}{[]}$ $\neg ground(\mathbf{N}) \rightarrow$
    return $\bigcup_{\theta|\mathbf{N}\theta\in Gamma} negations(Gamma, Delta, \bar{\mathbf{E}}\theta, \bar{\mathbf{N}}\theta, \bar{\mathbf{I}}\theta)$;
 $\underset{\mathbf{I}\in\bar{\mathbf{I}}}{[]}$ $finiteGoal(\mathbf{I}) \rightarrow$
    return $\bigcup_{\theta|\mathbf{I}\theta\in I_\circ} negations(Gamma, Delta, \bar{\mathbf{E}}\theta, \bar{\mathbf{N}}\theta, (\bar{\mathbf{I}} - \mathbf{I})\theta)$;
else
    return $\bar{\mathbf{N}}$;
fi
```

Fig. 12. Complete Incremental Interpreter Including Garbage Collection