Some Issues in the Operational Semantics of Logic Programs
(or: What do logic programs do?)

James Harland

December, 1995

Abstract

In this paper we discuss various issues in the operational semantics of logic programs. Such issues include the need for an operational semantics, the relationship between the operational semantics and the declarative semantics (which can involve a significant amount of subtlety), and various approaches to solving the technical problems faced when dealing with non-determinism.

Note: This paper is not intended as a complete survey of the field, but as a position paper exploring some ideas. Comments are most welcome.

1 Declarative Semantics

One of the key features of the logic programming paradigm is that programs have a declarative semantics, i.e. one that is given in terms of models, interpretations, formulae, and other well understood mathematical constructs, and not in terms of a sequence of operations. As Kowalski's famous equation of Algorithm = Logic + Control indicates [4], we may separate *what* the program computes from *how* it is computed. In particular we can consider the successful ground goals as a kind of abstract input-output relation, with the understanding that all the programmer has to do is to specify the logic of the algorithm, and allow the machine to take care of the control component. This allows the programmer to concentrate on what is important in the program rather than getting caught up in operational details.

For example, the semantics of the `append` predicate is usually given as a set of ground atoms, which will include the atoms below:[1]

[1]Note that we do not consider issues of polymorphism here; for simplicity, we just use lists of integers. A treatment that involved variables would be able to incorporate polymorphism, but similar remarks would still apply.

```
append([], [], [])
append([1], [2], [1,2])
append([1,2], [3], [1,2,3])
append([1,2], [3,4], [1,2,3,4])
...
```

From this we can then determine that the only answer to a query such as

```
?- append([], [2,3], X)
```

is `X = [2,3]`, that the query

```
append(X, Y, [1,2])
```

has solutions

```
X = [], Y = [1,2],
X = [1], Y = [,2],
X = [1,2], Y = [],
```

and that the query

```
?- append([1], [2,3], [2])
```

fails. Hence computation becomes a means of filling in gaps (where possible) in such a way that the programmer need not be concerned with the manner in which the answers are found, but only with the specification of correct programs.

2 The Need for Operational Semantics

Unfortunately, it seems that the declarative interpretation of the program, whilst an attractive and important feature of the logic programming paradigm, is often not specific enough, and the programmer is then forced to consider at least some aspects of the control component of the program. For example, pruning operators such as Prolog's cut, and other "extra-logical" constructs such as `var` and `nonvar` [10] are often used to enhance the programmers' control of the execution of the program. Whilst these are generally outside the semantics of logic programs, their existence may be seen as evidence that the operational interpretation of programs requires some attention from the programmer. In addition, more modern programming languages such as Mercury [8, 1] incorporate information such as modes and determinism into the language, which may be seen as a way of specifying the intended operational semantics of the program. In a nutshell, it would seem that we are often forced to care about the operational behaviour of logic programs.

An illustrative example here is sorting programs. If we allow the predicate `sort` to have two arguments, the first of which is a list of, say, integers, and the other the sorted form of the list, then the declarative semantics of any sorting program will include the atoms

```
sort([], [])
sort([1], [1])
sort([1,2], [1,2])
sort([2,1], [1,2])
sort([2,3,1], [1,2,3])
sort([3,1,2], [1,2,3])
...
```

Clearly this relation should hold no matter which sorting algorithm is used, and so we may think of this (infinite) list of atoms as specifying the behaviour of the program at a somewhat abstract level. However, programs with the same declarative semantics may have significant operational differences. For example, the sorting program may be implemented using naive sorting algorithms, such as *permutation sort*, or *bubblesort*, more sophisticated algorithms such as *insertion sort* or *shellsort*, or divide-and-conquer algorithms such as *quicksort* or *mergesort*. Each of these algorithms has a different operational behaviour, and are in some ways radically different. There is a significant operational difference between sorting a 50 element list with an $O(n!)$ algorithm such as *permutation sort* and sorting the same list with an $O(n \lg n)$ algorithm such as *mergesort*.

3 Levels of Operational Semantics

In this sense the *operational* semantics of a logic program, i.e. *how* the computation is performed, is deliberately somewhat more liberal than the declarative semantics, in that any method of computation that respects the declarative semantics of the program should be acceptable.[2] Hence, in principle at least, if the underlying execution model changes, then the same programs should return the same results, and so in this sense, the programmer could argue that the language has not changed, and hence he or she need not be concerned with the relative properties of the old and new implementation. This, however, does not seem to tally with experience, in that many would think that a significantly different execution model (such as the difference between a top-down interpreter and a bottom-up one) would correspond to a different programming language.

[2] It is assumed that only "reasonable" implementations are considered; the interpreter for which every query succeeds, and its twin for which every query fails, are neither particularly reasonable nor useful.

Another point to note is that there is a degree of flexibility in SLD-resolution [5], which is generally taken as the de facto standard for the operational semantics of a logic program. Note that there is generally more detail given than in the declarative semantics: unifiers, variables, clause variants, computation rules and so forth, which is perhaps only to be expected. However, there is still some freedom of choice within the notion of SLD-resolution, such as the choice of clause to be used, and the order in which sub-goals are evaluated. In Prolog the first matching clause is used, as well as the leftmost sub-goal, although it is well known that these are not the only choices. Hence it would seem that at least two levels of operational semantics may be distinguished: one, a more abstract "schema" of how things should be done, such as SLD-resolution, and the other a particular setting of the various parameters in a given implementation. Clearly there may be a formal relationship between the two, in that one may be thought of as an implementation of the other, but the main point to note is that different levels of abstraction may be useful for different purposes.

For example, consider the sorting example again. Whilst all sorting algorithms should have the same declarative semantics (i.e. all the results are the same), the operational semantics may differ in several ways. There is clearly a significant difference between the way that *permutation sort*, *shellsort* and *quicksort* operate, and it would seem that such significant differences should be somehow captured in the semantics of the program. In addition, there are differences in the way that similar sorting algorithms operate, such as *quicksort* and *mergesort*. The classic way to describe the difference is that while they both use divide-and-conquer, the *quicksort* algorithm uses an expensive partition and a cheap recombination, whereas the *mergesort* algorithm uses a cheap partition and an expensive recombination. One could pose, for example, another version of a divide-and-conquer sorting algorithm, say *thicksort*, which used both the expensive partition and the expensive recombination; apart from being intuitively a little silly, we should be able to reason about this program in a manner similar to that for *quicksort* and *mergesort*.

We can make even finer levels of distinction; in particular, *quicksort* can be implemented in various ways, in that different partition strategies may lead to different behaviour. Clearly we would expect different operational behaviour from each of the different partition strategies.

For these reasons it seems reasonable to propose that the notions of declarative and operational semantics should not be thought of as a dichotomy, but as part of a broad spectrum of levels at which we wish to inspect the behaviour of programs. Another perspective may be that rather than separate specifications and programs, we often need (at least operationally) a mixture of the two. For example, when comparing quicksort and mergesort, we need to consider the operational differences between the way that each algorithm performs the original partitioning of the list, but at this level the precise way in which the list is partitioned or recombined is not of direct interest. In this sense we seem to require the ability to "abstract away" from the parts of the program which are not of current interest.

It should be noted that a similar notion of abstracting away from details is often the way that the properties of programs are informally described. Often the first question that a novice computer user will ask about a given piece of software is "What does this program *do*?" or perhaps "How do I get it work?" or "What happens if I press this key now?". Once a little more experience has been gained, the novice user may begin to classify programs by function, dividing up the software world into editors, word processors, mail programs, newsreaders, compilers, games and so forth. Of course he or she will often still want to know the details of particular programs, but a simple description such as "a windows package" will convey a substantial proportion of the desired information. This grouping and abstraction activity may thought of as a step in the direction of a declarative semantics, in that it seeks to identify the key features of a program rather than the mass minutiae of the operational details. Thus the provision of a declarative semantics not only provides a standard against which various implementations may be compared, but also formalises a particular abstract view of programs.

In general, the amount of detail needed will depend on the context; sometimes one will want a very high-level, abstract view of what the program is, such as when describing its basic features to a potential customer; at other times, one will want a medium level of detail, such as two experts discussing the implementation techniques used in two similar systems, or a large amount of concrete information, such as when debugging a complex system. Ideally, then, the semantic framework should be able to be "scaled" appropriately, so that the desired level of abstraction may be found within a single framework. Of course, this may not be possible; however, we would clearly settle for a sufficiently close approximation.

It would also seem useful to interpret different operational semantics within the same framework. In other words, there does not seem to be one universal execution model for logic programs — top-down, bottom-up, top-down with memoing, depth-first vs. breadth-first vs. iterative deepening search, the Andorra model, etc., and yet it would seem to be useful to reason about the behaviour of programs in a variety of execution models.

For example, consider a program containing the rule

```
p :- q, p.
```

In Prolog, the goal `?- p` will run forever. In a system such as XSB[12] or Aditi[11], this query will terminate with failure. Hence we may see this as a "bad" Prolog program (it fails to terminate) and a "good" XSB or Aditi program (it terminates). However, there may be cases in which the non-termination is desirable, such as in a command processor or operating system, in which the verified ability not to come to a halt is clearly an important property of the system.

Hence, the precise formulation of an operational semantics for logic programs involves the consideration of a number of issues, one of which, as we shall see, is the precise relationship between it and the declarative semantics.

4 Equivalence and Full Abstraction

From the above it would seem reasonable to deduce that programs with the same declarative semantics may have differing operational semantics, reflecting the fact that there are many different ways to implement the "same" program. However, when the operational semantics of the two programs are the same, what should that imply about the declarative semantics? If we are to take the notion of *logic* programming seriously, it would seem that programs which behave in the same manner under all operational conditions should be declaratively, and hence logically, equivalent. In other words, operationally equivalent programs should be logically equivalent.

Unfortunately, this property is far from holding for Horn clauses under the usual T_P semantics. For example, the two programs below have the same declarative semantics: the success set is $\{p\}$ and the failure set is $\{q, r\}$.

```
p.          p.
q :- r      r :- q
```

However, it seems unnatural to conclude that these two programs are in fact the same or equivalent, even though they have the same declarative semantics. This is because these two programs will behave differently in different contexts, in that their declarative semantics will differ if r is added to each program. In this way, regarding the declarative semantics of the program as a set or sets of atoms may not fully capture the subtleties of the program, and hence may not be a fully satisfactory semantics. In particular, this example shows that the declarative semantics is not fully abstract with respect to logical equivalence, in that there are logically inequivalent programs which have the same declarative semantics.

Hence we may ask whether there is a way to bridge this gap, and find some form of the declarative semantics which is more in tune with the subtleties of the operational semantics. Whilst there are many possible approaches here, including adding more structure into the models of the program, one way that has some interesting consequences is to allow a broader class of goals to be used. In the usual definition of Horn clauses, goals are restricted to be of the form

$$\exists \vec{x} A_1 \wedge A_2 \wedge \ldots \wedge A_n$$

where each of the A_i are atoms.

However, as we have seen, this class of goals does not allow us to inspect the structure of the program. Note that the same properties hold if we allow goals to be defined via the grammar below:

$$G ::= A \mid G \wedge G \mid G \vee G \mid \exists x G$$

In other words, allowing free occurrences of $\wedge$, $\vee$, and $\exists$ does not seem to help with this problem. However, if we extend the class of goals to include embedded implications, the picture changes, in that the two programs above become operationally different. Hence it seems useful to investigate the notions of declarative and operational equivalence for the class of formulae below:

$$D ::= A \mid D \wedge D \mid \forall x D \mid G \supset A$$
$$G ::= A \mid G \wedge G \mid G \vee G \mid \exists x G \mid D \supset G$$

Note that the D formulae basically provide the Horn clause structure, when the goals are just existentially quantified conjunctions of atoms. This class of formulae is a substantial fragment of the class known as *hereditary Harrop formulae* [7] [3].

The investigation of the properties of operational equivalence for this language has been given in [2]. In summary, if we define the notion of operational equivalence for programs in a "black box" manner, i.e. that

$$P_1 \equiv_o P_2 \text{ iff for all goal formulae } G,\ P_1 \vdash_o G \text{ iff } P \vdash_o G$$

where the relation $\vdash_o$ characterizes our execution model, then we find that operational equivalence implies not equivalence in intuitionistic logic, as we might have expected, but in a slightly stronger logic. This logic, which is significantly weaker than classical logic, is obtained from intuitionistic logic by adding the *Independence of Premise* rules; in particular, we need the following equivalences to hold:

$$D \supset (G_1 \vee G_2) \equiv (D \supset G_1) \vee (D \supset G_2)$$
$$D \supset \exists x\ G \equiv \exists x\ D \supset G$$

where x is not free in D. Note that the $\Leftarrow$ direction of both equivalences hold in intuitionistic logic.

From this we may see that in order to make good the seemingly simple notion that logically equivalent programs may be operationally different, but operationally identical programs should be logically equivalent, it seems that some sophisticated logical tools are called for, as this assertion contains significantly more subtlety than might appear on first sight.

[3] The full class of hereditary Harrop formulae includes the case $\forall x G$ as a goal.

Whilst on such matters, the notion of declarative semantics per se may need to be extended and/or improved. For example, are the two programs below really so different?

```
p(a)                          q(b).
p(f(X)) :- p(X).              q(g(Y)) :- q(Y).
```

Whilst there are various techniques which can be used to exploit the similarity of structure between the programs, it would seem important to incorporate some notion of similarity into the declarative semantics of logic programs, and probably also into the operational semantics [9].

Another way to think of this is that if there are only 12 original jokes, how many original logic programs are there? It can often feel, when writing accumulator programs, or programs which manipulate lists in various ways, that there is a central methodology involved, and an awful lot of detail. A systematic manner in which to organise such programs, and hence a way of comparing similar programs, would clearly be a step in the right direction.

5 Nondeterminism and Systematic Search

In many ways, one can see the problems with the operational semantics of programs as finding ways to deal systematically, but not dogmatically, with non-determinism. As an example of one approach to this, consider the problem below. One of the key inference rules of linear logic is the rule for $\otimes$, given below.

$$\frac{\Gamma \vdash F_1, \Delta \qquad \Gamma' \vdash F_2, \Delta'}{\Gamma, \Gamma' \vdash F_1 \otimes F_2, \Delta, \Delta'}$$

This rule is known as a *multiplicative rule*, in that the two contexts from each branch are combined into a single one in the conclusion. The effect of this rule is that when searching for a proof of a sequent $\Gamma \vdash F_1 \otimes F_2, \Delta$, which corresponds to executing a goal containing $\otimes$, it is necessary to find a systematic way of dividing the context up into two mutually exclusive and exhaustive parts such that each branch is provable.

There are (at least) two approaches to this problem. One is to use a lazy approach, first advocated by Hodas and Miller [3] and implemented in the languages Lolli[3] and Lygon[13]. In this approach, the first conjunct is given all of the formulae, from which it extracts all the formulae needed, and the remaining formulae are passed to the second branch. We can express this using hypersequents as follows:

$$\frac{\Gamma \vdash F_1, \Delta \Rightarrow \Gamma' \vdash \Delta' \qquad \Gamma' \vdash F_2, \Delta' \Rightarrow \Gamma'' \vdash \Delta''}{\Gamma, \vdash F_1 \otimes F_2, \Delta \Rightarrow \Gamma'' \vdash \Delta''}$$

The intuition behind a hypersequent $\Gamma \vdash \Delta \Rightarrow \Gamma' \vdash \Delta'$ is that the sequent $\Gamma \vdash \Delta$ is provable provided that the resources Γ' and Δ' are removed from the sequent.

As exemplified in Lolli and Lygon, this technique leads to an execution model for which novel and useful logic programs can be written, as this execution model introduces a localised notion of state. However, whilst this is a natural and useful technique, it is not the only one. In particular, this is basically a depth-first approach, and there is no apparent reason why a more breadth-first approach may be used. In such an approach, rather than evaluating one sequent first and then the other, both sequents may be executed in parallel, with some well-known technique for the control of concurrent access to resources (such as semaphores or monitors) to allocate each formula to a unique branch. This gives us a form of co-routining, with the interaction between the two processes driven by the need to balance the allocation of formulae between branches.

Clearly both of these approaches deal with the non-determinism inherent in the $\otimes$ rule, but in very different ways, and in ways which correspond to different styles of programming. In addition, there is no obvious reason why the programmer would need to use only one of these techniques; a system which allowed both such search strategies to be used (or even some hybrid combination of them) would clearly be of significant interest. This would require a means of being able to reason about different strategies in the same framework, which appears to be a non-trivial task.

One possible approach to this is to allocate a boolean variable to each occurrence of a formula in a sequent, which allows us to express the mutual exhaustion and exclusion constraints algebraically, and then the different strategies correspond to different ways of solving the equations, such as completely solving all the equations on one branch before doing so for another (the lazy approach) or solving all the equations in the entire tree at once. The details of this approach are still underdevelopment, but it would seem that the crucial step is to discover the appropriate algebraic representation of the possible strategies.

6 Contribution of this Workshop

As stated above, it would seem that the central problem is to deal with non-determinism appropriately, and that algebraic tools may be of help. The paper by Zhang and Rounds in this workshop is one such approach, in which the non-determinism of logic programs is expressed in terms of a powerdomain. This seems a natural approach, and one that holds much promise.

Another thread is to develop more sophisticated logical tools to study the operational properties of logic programs. One such approach is the paper by

Andrews, Dahl and Tarau, in which a certain kind of grammar is used as the operational model, and it is shown how various paradigms may be mapped into this general model. Another approach is given in the paper by Staerk, in which a sophisticated and detailed model of a logic programming interpreter is given, in such a way that it is possible to prove various properties of the interpreter. Both approaches strengthen the ability to reason about the operational behaviour of logic programs.

A similar approach is given in the paper by Matsuoka, in which the notion of a proof-net in linear logic is used to provide stronger technical tools for reasoning about derivations and rule systems. Again, this would appear to only strengthen our ability to reason about the operational behaviour of logic programs.

The paper by Delzanno and Martelli addresses the problem of incorporating object-oriented programming facilities into a logic programming language, and shows how this may be done by mapping such facilities into a linear logic system known as Forum [6]. That the operational semantics is rich enough to allow the encoding of such a paradigm indicates that it is of significant power.

An important aspect of many logic programs is the use of recursion, and the paper by Liang discusses a method of proving the equivalence of recursive definitions. This seems particularly important for program transformation and optimisation, which are naturally related to the question of operational semantics.

It is well-known that the use of Negation as Failure can lead to somewhat idiosyncratic behaviour, and so a systematic operational account of the behaviour of programs which use this connective is particularly helpful. The paper by Momigliano and Ornaghi addresses this issue.

Hence, it would seem that the papers in this workshop are all addressing important aspects of the operational semantics of logic programs, and showing the diversity of possible approaches.

7 Conclusion

The intent of this paper was to discuss some issues, in the hope that this will lead to further discussion at the workshop. It would seem that the field of operational *semantics* for logic programs (as distinct from operational *techniques*) is still in its infancy, and there are many interesting directions to be followed. It would seem, though, that the ultimate aim is to be able to reason about non-determinism in a flexible and appropriate manner, and that the fulfilment of this aim is still some significant distance away.

References

[1] Thomas Conway, Fergus Henderson, and Zoltan Somogyi. Code generation for Mercury. *Proceedings of the International Logic Programming Symposium*, Portland, December, 1995.

[2] James Harland. On Normal Forms and Equivalence for Logic Programs. *Proceedings of the Joint International Conference and Symposium on Logic Programming* 146-160, Washington DC, November, 1992.

[3] J. Hodas and D. Miller. Logic Programming in a Fragment of Intuitionistic Linear Logic. *Journal of Information and Computation* 110:2:327-365, 1994.

[4] R.A. Kowalski. Predicate Logic as a Programming Language. *Information Processing 74*, North-Holland, Amsterdam, 1974.

[5] John Lloyd. *Foundations of Logic Programming.* Springer-Verlag, Berlin, 1984.

[6] Dale Miller. A multiple-conclusion meta-logic. Proc. of the IEEE Symposium on *Logic in Computer Science* 272–281, Paris, June, 1994.

[7] Dale Miller, Gopalan Nadathur, Frank Pfenning and Andre Scedrov. Uniform Proofs as a Foundation for Logic Programming. *Annals of Pure and Applied Logic* 51:125-157, 1991.

[8] Zoltan Somogyi, Fergus Henderson and Thomas Conway. Mercury: An Efficient Purely Declarative Logic Programming Language. *Proceedings of the Australian Computer Science Conference* 499-512, Adelaide February 1995.

[9] Leon Sterling and John Power. A Notion of Map Between Logic Programs, *Proceedings of the International Conference on Logic Programming* 390-404, Jerusalem, June, 1990.

[10] Leon Sterling and Ehud Shapiro. *The Art of Prolog* (2nd. edition). MIT Press, 1994.

[11] J. Vaghani, K. Ramamohanarao, D. Kemp, Z. Somogyi, P. Stuckey, T. Leask and J. Harland. *The Aditi Deductive Database System.* VLDB J. 3:2:245–288, 1994.

[12] D.S. Warren. Programming the PTQ Grammar in XSB. in *Applications of Logic Databases*, Raghu Ramakrishna (ed.), Kluwer Academic, 1994.

[13] Michael Winikoff and James Harland. Implementing the Linear Logic Programming Language Lygon. Proceedings of the International Logic Programming Symposium, Portland, December, 1995.