TLP **3** (1): 1–59, January 2003.
DOI: 10.1017/S1471068402001540 Printed in the United Kingdom

The witness properties and the semantics of the Prolog cut

JAMES H. ANDREWS

Department of Computer Science, University of Western Ontario, London, Ontario, Canada N6A 5B7

Abstract

The semantics of the Prolog 'cut' construct is explored in the context of some desirable properties of logic programming systems, referred to as the witness properties. The witness properties concern the operational consistency of responses to queries. A generalization of Prolog with negation as failure and cut is described, and shown not to have the witness properties. A restriction of the system is then described, which preserves the choice and first-solution behaviour of cut but allows the system to have the witness properties. The notion of cut in the restricted system is more restricted than the Prolog hard cut, but retains the useful first-solution behaviour of hard cut, not retained by other proposed cuts such as the 'soft cut'. It is argued that the restricted system achieves a good compromise between the power and utility of the Prolog cut and the need for internal consistency in logic programming systems. The restricted system is given an abstract semantics, which depends on the witness properties; this semantics suggests that the restricted system has a deeper connection to logic than simply permitting some computations which are logical. Parts of this paper appeared previously in a different form in the *Proceedings of the 1995 International Logic Programming Symposium* (Andrews, 1995).

KEYWORDS: operational semantics, abstract semantics, depth-first search, negation as failure, cut

1 Introduction

Since the first widely-used Prolog implementations of the early 1980s, Prolog programmers have had access to some powerful constructs for controlling the backtracking behaviour of their programs. The best-known of these is the 'cut', written '!', which appears as a literal in the sequence of literals in a clause body. Cut allows programmers to direct the flow of control in a program by cutting away backtrack points which lead to unwanted execution paths.

Programmers have embraced cut enthusiastically. Most large Prolog programs now in use contain cuts, or related constructs such as the if-then-else construct (*A* -> *B* ; *C*). Cut is used mainly for choosing between clauses. However, it has other important uses, such as for obtaining the first solution to a subgoal and discarding others.

Unfortunately, the unrestricted use of cuts produces a program which has no direct logical interpretation. A cut does not even have an effect restricted to the

clause in which it appears; rather, it may affect all the clauses of the predicate which its clause is defining. It is therefore difficult to give a semantics to a program which uses cut, other than an operational semantics.

It seems therefore that the use of cut, and the constructs related to it, must be restricted in order to regain a logical interpretation for Prolog programs. Various approaches to this have been proposed, including the 'soft cut' and the mode and determinism restrictions of the Mercury system (Somogyi *et al.*, 1996). However, neither soft cut nor Mercury allow the behaviour of cut which allows us to choose the first solution to a subgoal and discard other solutions. This is a fundamental property often used by Prolog programmers, so it would be preferable to preserve it.

Like most logic programming researchers, we believe that Prolog's 'hard cut' cannot be salvaged from a logical point of view. However, we do not believe it is necessary to retreat all the way to soft cut. In this paper, we show how the hard cut of Prolog can be restricted to produce a cut, referred to as 'firm cut', which has important advantages over both soft and hard cut. Firm cut allows useful behaviours such as first-solution which are disallowed by soft cut. Modulo a run-time or compile-time mode restriction, firm cut is operationally identical to the more widely-used hard cut, which soft cut is not. However, firm cut disallows the most non-logical and anti-intuitive behaviours of hard cut, and while (like hard cut) it has no purely logical interpretation, it still satisfies some important consistency properties which hard cut does not.

We refer to the consistency properties which firm cut satisfies as the 'witness properties'. Because it satisfies these properties, firm cut and the systems incorporating it can be given abstract semantics based on compositional valuation functions (functions from goals to truth values). We demonstrate this by giving such an abstract semantics for the system with firm cut.

Along the way, we also introduce a form of formula, the *if* formula, which allows a Prolog program with cuts to be given a 'completed form' analogous to the Clark completion of a definite clause program. This form of program may have applications even when dealing with other forms of cut.

1.1 The witness properties

One of the central properties we like to prove about logic programming systems is the equivalence between the operational and logical semantics. The well-known equivalence of SLD-resolution and the least model semantics is the most obvious example. Such properties show that the logic programming system in question achieves some standard of expected behaviour.

But what if the logic programming system has no logical semantics? Is there any standard to which such a system can be held, any middle ground between a system with a full logical semantics and a system indistinguishable from imperative or functional programming systems? We believe that there is, and suggest the *witness properties* as a possible standard.

https://doi.org/10.1017/S1471068402001540 Published online by Cambridge University Press

The witness properties are as follows:

1. (Success property) If a goal formula G succeeds (returns an answer substitution), then some ground instance of G succeeds.
2. (Failure property) If a goal formula G fails (terminates without returning an answer substitution), then all ground instances of G fail.

The witness properties accord with our intuitions about the internal consistency of logic programming systems, and about the nature of formulas and the search for satisfying substitutions for them. They therefore provide a possible standard to which to hold logic programming systems. Their name comes from the notion of witness for an existentially-quantified formula: the formula $\exists x\ G$ is true if there is a witness term t such that $G[x := t]$ is true, and false otherwise. A goal formula such as $p(x)$ can be read as asking whether $\exists x\ p(x)$ is true.

In the success property, we insist on *ground* instances in particular, partly because otherwise it would always be vacuously true: G is an instance of G, so if G succeeds, some instance of it succeeds. We express the failure property in terms of ground terms as well for symmetry. Another reason for using ground terms in the statement of the properties is that it allows the success and failure of goals with free variables to be characterized in terms of the simpler notion of success and failure of ground goals. Many variants of these properties are possible and may be valuable for different applications.

Note that the converses of the witness properties are not necessarily enjoyed by logic programming systems. The converse of the success property (if an instance of G succeeds, then G succeeds) is not enjoyed by any deterministic definite clause resolution system (like Prolog) using a search rule which selects clauses in order, as the following example shows:

$p(0)$:- $p(0)$.
$p(1)$.

The goal $p(y)$ diverges even though its instance $p(1)$ succeeds. The converse of the failure property (if all ground instances of G fail, then G fails) is not enjoyed by any deterministic definite clause resolution system, regardless of search or selection rule, as the following example (based on that of Clark, Andreka and Nemeti) shows:

$p(f(x))$:- $p(x)$.

The goal $p(y)$ diverges even though every ground instance of it fails.

The witness properties also have theoretical significance. Generally, we may consider a logic programming system to be unsatisfying from a logical point of view if it can be given only operational semantics, as this leads us to suspect that the operational model is a 'hack' which is only logical in the sense that it permits some computations which can be viewed as logical. Of course, every operational semantics for an LP language can be converted to a denotational semantics if operational notions such as unification and substitution sequence are suitably 'reified' (i.e. represented explicitly by mathematical constructs). However, these semantics should not necessarily boost our confidence that the operational model is logical,

https://doi.org/10.1017/S1471068402001540 Published online by Cambridge University Press

any more than the operational semantics did. The existence of semantics which do not reify operational notions suggests that we are dealing with a system which has a deeper connection to logic than simply permitting logical computations. Evidence from past research and the present paper indicates that the witness properties lead to such semantics.

1.2 This paper

In this paper, we show how the hard cut of Prolog, as restricted to 'firm cut', retains the witness properties and can be given an abstract, non-reifying semantics. We believe that the resulting system is the best compromise yet found between the power and utility of the Prolog cut and the need for internal consistency in logic programming systems.

In section 2, we review background and related work in more detail. In section 3, we present the notation and syntax we will use for logic programs with cut and a new construct, *if*. In section 4, we present a first operational semantics for the extended programs. This operational semantics corresponds to Prolog, with its permissive, non-logical view of negation and cut; thus it is referred to as the 'liberal' semantics. In section 4, we also show that the *if* construct allows us to derive a convenient 'completed form' for every program, in which each predicate is defined by exactly one clause.

In section 5, we restrict the liberal operational semantics, and show that the restricted system has the witness properties. The new, restricted system is referred to as the 'conservative' semantics, and firm cut is defined as the cut associated with it. In section 6, we define a non-reifying abstract semantics for the system with firm cut, using the witness properties to prove soundness and completeness of the conservative semantics. Finally, in Section 7 we give some conclusions and suggestions for further research.

2 Background and related work

In this section, we introduce the background of this research and the other research related to it. We have grouped this material into three sections: one concerning the cut and other choice constructs like the if-then-else, one concerning the semantics of depth-first Prolog and cut, and one concerning the various different notions of termination of a logic program.

2.1 Cut and other choice constructs

Cut was introduced in the DECsystem-10/20 Prolog of 1982, written by David Warren, Fernando Pereira, Lawrence Byrd and Luis Pereira. It was recognized even at the time as a 'meta-theoretic' control construct, which could at best be read as making meta-level manipulations of the search tree. Cut was taken into the C-Prolog interpreter (Pereira *et al.*, n.d.), which became a very widely distributed early version of the language.

https://doi.org/10.1017/S1471068402001540 Published online by Cambridge University Press

Cut operates by cutting away previously-encountered alternatives. Consider the following program:

$p(a, y).$
$p(b, y)$:- $q(y), !, r(y).$
$p(x, y).$
$q(c).$
$q(d).$
$r(d).$

(x and y are variables, and a-e are constants.) With respect to this program, calls to the predicate p exhibit the following behaviour.

- Goals of the form $p(a, t)$ succeed for any term t.
- Goals of the form $p(b, t)$ succeed only if t is d, or if t is not unifiable with either c or d; otherwise they fail. For instance:
  - — The goal $p(b, y)$ fails, because y is unified with c by the first clause for q, the last clauses for p and q are cut away, and $r(c)$ fails.
  - — The goal $p(b, d)$ succeeds because $q(d)$ succeeds, only the last clause for p is cut away, and $r(d)$ succeeds.
  - — The goal $p(b, b)$ succeeds because $q(b)$ fails entirely, and so the third clause for p is used.
- Finally, goals of the form $p(s, t)$, where s is anything other than a and b, succeed.

Cut therefore cuts away not only the later clauses of the same predicate, but also the alternative clauses for subgoals that appear earlier in the clause. The former behaviour allows us to select clauses, but the latter behaviour allows us to choose the first solution to a subgoal (by stating the subgoal and following it by a cut). This may be used for various reasons: to discard solutions that we, the programmers, know to be equivalent to the first; to prevent backtracking because we know there will be no more successes; or simply to select the first solution because we know that is the one we are interested in (for instance, "$prime(x), x > 100, !$" for the first prime greater than 100).

We can see immediately that Prolog with the form of cut described above does not have the failure witness property, since $p(b, y)$ fails but $p(b, d)$ succeeds. (Examples can be constructed violating the success witness property as well.) The most common way to fix this problem with cut is to allow backtracking into the portion before the cut – that is, to cut away later clauses to the current clause but not alternative clauses to subgoals before the cut. This is generally referred to as the 'soft cut', and the more usual cut is referred to as the 'hard cut' to distinguish it. With soft cut, we can regain a logical interpretation: if ! in the above program is interpreted as soft cut, then the second and third clauses are equivalent to the classical formulas

$p(b, y) \leftarrow q(y) \& r(y).$
$p(x, y) \leftarrow (\neg(x = b) \vee (x = b \& \neg q(y))).$

However, we lose the ability to select the first solution with soft cut.

https://doi.org/10.1017/S1471068402001540 Published online by Cambridge University Press

A construct related to cut is the 'if-then-else' construct, usually written (G_1 -> G_2; G_3) and read "if G_1 then G_2 else G_3". This construct is often syntactic sugar for a hard-cut-like operation; that is, the evaluation of (G_1 -> G_2; G_3) is equivalent to the evaluation of a goal $p(x_1, \ldots, x_n)$ against the program

$p(x_1, \ldots, x_n)$:- G_1, !, G_2
$p(x_1, \ldots, x_n)$:- G_3

where $x_1, \ldots, x_n$ are the free variables in G_1, G_2, G_3.

The cut in the if-then-else construct is hard cut in most Prologs. The choice construct of the Mercury language (Somogyi *et al.*, 1996) is written in this way and uses soft cut; Mercury has no other choice construct.

2.2 Semantics of Prolog and cut

The least-model semantics (van Emden and Kowalski, 1976) is traditionally viewed as the standard one for pure logic programming as it was originally conceived. However, the depth-first search of Prolog and similar systems makes it difficult to fit them into the least-model framework, at least if we want a semantics with respect to which the system is sound and complete. Evidently some other form of semantics is needed to characterize depth-first logic programming systems precisely, whether taking cut into consideration or not.

The *operational* semantics of Prolog with cut was not formally defined in a self-contained system until Billaud's 1990 paper (Billaud, 1990). In Billaud's semantics, when a predicate is called, the current backtrack stack is stored; the execution of a cut corresponds to discarding the current backtrack stack and replacing it with the one stored by the current predicate.

Various authors have given *denotational* semantics for Prolog with cut (de Bruin and de Vink, 1989; Börger, 1990; Baudinet, 1992), including Billaud in his original paper (Billaud, 1990). Some of these approaches have proven equivalence with an operational semantics. These papers were based on earlier work in operational and denotational semantics of Prolog, including (Jones and Mycroft, 1984; Deransart and Ferrand, 1987; Arbab and Berry, 1987; Debray & Mishra, 1988; Nicholson and Foo, 1989).

The denotational approaches essentially view a Prolog program as a function from goals to sequences of answer substitutions, and 'reify' notions like unification and answer substitution sequence by giving abstract mathematical constructs corresponding to them. Such approaches are able to handle any operational model which transforms a goal into a sequence of substitutions using unification. This includes models with any conceivable sound or unsound strategy for negation and cut; for instance, sound soft cut, unsound negation as failure, or a negation operator which judges $\neg p(t)$ to be true iff t unifies with 42. Therefore, although a reifying semantics may be very useful for some purposes (for instance, to use as a guide for implementation of a standard computational model), the existence of such a semantics does not by itself suggest that the system thus characterized is any more than an operational superset (or superset of a subset) of pure logic programming.

In contrast, what may be called the 'non-reifying' semantic tradition (Andrews, 1991; Andrews, 1997; Stärk, 1998; Elbl, 1999) gives characterizations of the success and failure of Prolog goals not involving reified answer substitutions and unification. Andrews' earliest characterizations (Andrews, 1991) took account only of depth-first Prolog without builtins, negation or cut. Andrews (Andrews, 1997) and Stärk (Stärk, 1998) then extended this to systems with negation as failure, Andrews by characterizing floundering and Stärk by imposing a mode restriction. More recently, Elbl (Elbl, 1999) has given a semantics for depth-first logic programming which uses more abstract denotations to achieve compositionality, and extends this semantics to take account of negation with a similar mode restriction to Stärk's.

These more logical approaches draw their power from expressing the semantics of Prolog in a manner which allows them to avoid encoding operational notions such as unification into the semantics. Without such a property, proofs using Stärk's proof assistant (Stärk, 1998), for instance, would have to reason about unification at almost every step.

We should note that even reifying semantics can act as the basis of powerful theorem provers if they are automated. For example, Lindenstrauss, Sagiv and Serebrenik (Lindenstrauss and Sagiv, 1997; Lindenstrauss *et al*., 1997) discuss automatic proofs of strong termination based on term rewriting techniques. However, in proving termination and (especially) correctness properties, it is often necessary to have human intervention, in order to deduce generalizations to be proven by induction or norms for proving termination.

2.3 Termination

We seek an abstract semantics with respect to which some large subset of Prolog with cut is sound *and complete*. The soundness property allows us to argue that any outcome which a Prolog goal does return is consistent with the semantics. The completeness property, however, allows us to argue that the semantics does not judge a goal to be true (resp. false) unless it actually succeeds (resp. fails) according to the operational semantics; that is, that we have precisely captured *termination* of goals. We must therefore define exactly what we mean by termination of a goal. In this paper, we study *left-to-right* termination, which subsumes the more widely-studied notion of *strong* termination.

A Prolog query can have one of several outcomes. It can succeed or fail, or it can diverge (fail to terminate altogether). If a query succeeds, Prolog typically gives us the option of finding more solutions. If we keep asking for more solutions, there are three things that may happen: the query may eventually fail back to the top level and report no more solutions; the query may return a finite number of solutions and then diverge; or the query may return an infinite number of solutions. We may label these outcomes as:

1. Success:
   (a) Finite number of solutions, then failure.
   (b) Finite number of solutions, then divergence.
   (c) Infinite number of solutions.

https://doi.org/10.1017/S1471068402001540 Published online by Cambridge University Press

2. Failure.
3. Divergence.

These outcomes correspond to the shape of the resolution search tree for systems with a left-to-right subgoal selection rule, and the placement of solutions within that tree. (In the following, we assume that the leftmost subgoal is always selected, that the children of each node of the search tree correspond, left to right, to the sequence of clauses defining the selected subgoal's predicate, and that the search rule is also left-to-right.) If the tree is finite, we get outcome 1(a) or 2. If it has some infinite path, and there is a finite number of solutions to the left of the leftmost infinite path, we get outcome 1(b) or 3. Otherwise, there is an infinite number of solutions to the left of the leftmost infinite path (outcome 1(c)), and we can obtain only a finite prefix of the sequence of solutions by backtracking.

The two kinds of termination most often mentioned in the literature are *existential termination* and *universal termination*. A query existentially terminates either if it fails, or if there is a solution somewhere in the search tree. Knowing that a query existentially terminates is thus useful primarily if we are studying breadth-first implementations or nondeterministic operational semantics. A query universally terminates if the search tree is finite (i.e., a search on any path terminates). Universal termination therefore corresponds only to cases 1(a) and 2 above.

Most of the work on proving termination of Prolog programs (Plümer, 1990; Apt and Pedreschi, 1993; Bezem, 1993; Apt and Marchiori, 1994; Stärk, 1998) has concentrated on universal termination. Because of our interest in features of practical logic programming systems such as Prolog, in this paper we continue to study what we refer to as *depth-first termination*. A query depth-first terminates if it returns at least one solution, or if it fails. Depth-first termination thus encompasses outcomes 1(a)-(c) and 2 above, and thus identifies a larger set of queries as terminating than universal termination. It also corresponds to one of a Prolog user's intuitive notions of termination of a goal.

Depth-first termination is what we will have to characterize if we want to take account of the behaviour of cut. Cut cuts away all but the first solution returned from the portion of the clause before the cut, so all that is important to the semantics is that the portion before the cut returns at least one solution or fails. Note, however, that even in the absence of cut, a goal formula G universally terminates iff the query $(G \& false)$ (in Prolog parlance, `(G, fail)`) depth-first terminates. Depth-first termination is thus strictly more general than universal termination.

3 Notation and syntax of extended programs

In this section, we define the syntax of programs that we will use for the rest of the paper. It is a generalization of the subset of Prolog including cut (!), negation as failure, and defined predicates. It does not include problematic built-in predicates such as `assert` and `retract`, `var`, `nonvar`, and `setof`, each of which merits further study but whose inclusion might confuse the issues we study here.

https://doi.org/10.1017/S1471068402001540 Published online by Cambridge University Press

We use the following meta-variables: B, C, F, G and H for formulas, s and t for terms, and x, y and z for variables, all possibly primed or subscripted. We use $\vec{x}$, $\vec{t}$, etc. generally to stand for sequences of variables, terms, etc. We use $\exists\vec{x}$ as notation to stand for $\exists x_1 \ldots \exists x_n$, where $\vec{x} = (x_1, \ldots, x_n)$.

We define an extended notion of *goal formula* (or simply *formula*), representing a query or an element of a clause body. The BNF definition of a formula is as follows.

$$\begin{array}{rcl} G & ::= & (t = t) \mid p(t, \ldots, t) \mid G \& G \mid G \vee G \\ & \mid & \neg G \mid \exists x\ G \mid if\,[\vec{x}](G, G) \end{array}$$

All the connectives are standard except the *if* connective. $if\,[\vec{x}](B, C)$ is a variable binding construct, which binds all the variables in the list $\vec{x}$. $if\,[\vec{x}](B, C)$ is computed as follows: if $\exists\vec{x}(B)$ is false, so is $if\,[\vec{x}](B, C)$; otherwise, $if\,[\vec{x}](B, C)$ is equivalent to $C\theta$, where θ is the first substitution for $\vec{x}$ returned by the computation of B. This form of formula allows us to express a Prolog program with cuts in a 'completed' form (see section 4.3).

We assume a standard syntax of terms. We assume that the language of the program contains at least two terms, which we will refer to as 0 and 1. We define the formula *true* as $0 = 0$, and the formula $false$ as $0 = 1$.

Because we will be speaking of clauses with cut, we cannot use the standard logic-programming definition of clause. The BNF definitions of formula, clause, clause body, and clause body element used in this paper are as follows:

$$\begin{array}{lcl} clause & ::= & p(t, \ldots, t)\ \texttt{:-}\ body \\ body & ::= & \epsilon \mid bodyelt, body \\ bodyelt & ::= & G \mid\ ! \end{array}$$

(ϵ is the empty expression.) As in Prolog, we generally write a clause of the form $p(t_1, \ldots, t_n)$:- ϵ as simply $p(t_1, \ldots, t_n)$. Note that we restrict the cut to occurring 'at the top level' in clauses. In most Prologs it is possible to use cut within a complex formula (for instance, a disjunction), but such cuts are seldom used and their effect is generally said to be undefined[1].

A *program* is a sequence of clauses. It is clear that the syntax of programs, as defined here, generalizes the syntax of Prolog programs with only literals and cuts as body elements. For simplicity, we assume that each predicate is defined with a distinct arity in a given program; that is, that at every occurrence of a predicate name, it is given the same number of parameters. We say that a clause *defines* predicate p if the head of the clause has predicate p. We use $clauses(p, P)$ to stand for the sequence of clauses defining predicate p in program P.

As an example of a program in the extended syntax, consider the following standard definition of a 'delete' predicate:

$d(x, [\,], [\,])$
$d(x, [x|ys], zs)$:- $!, d(x, ys, zs)$
$d(x, [y|ys], [y|zs])$:- $d(x, ys, zs)$

[1] Billaud's operational semantics of cut (Billaud, 1990) defines a behaviour of cuts within complex formulas which is consistent with the operational semantics of some Prolog interpreters.

https://doi.org/10.1017/S1471068402001540 Published online by Cambridge University Press

The goal $d(x, y, z)$ deletes all occurences of the element x in the list y, resulting in the list z. As we will see, the following definition is equivalent:

$$
\begin{aligned}
&d(x, y, z) \ :- \\
&\quad (y = [\,] \ \& \ z = [\,]) \\
&\vee \ if\,[ys](y = [x|ys], d(x, ys, z)) \\
&\vee \ (\neg\exists ys(y = [x|ys]) \ \& \\
&\qquad \exists y'\exists ys\exists zs(y = [y'|ys] \& z = [y'|zs] \& d(x, ys, zs)))
\end{aligned}
$$

4 The liberal operational semantics

In order to define precisely the logic programming systems which will be the focus of our study, we must define precisely their operational, or procedural, semantics. In this section, we define two operational semantics (the second simpler than the first) for the extended logic programs defined in the last section. Because they share Prolog's rather lax, non-logical interpretation of negation and cut, they are referred to as 'liberal' operational semantics. The second of these semantics will be used as the basis of the more 'conservative' semantics of the next section, which regains the witness properties.

Traditionally, operational semantics of logic programming are given using variants of resolution, in particular SLD-resolution. However, in the presence of such features as depth-first search, negation as failure and cut, SLD-resolution-based operational semantics require an additional superstructure of definitions, for instance to define the order in which branches of the SLD-tree are searched. We therefore follow other researchers (Deransart and Ferrand, 1987; Billaud, 1990) in defining operational semantics for our system using the style which has come to be known as SOS, or Structured Operational Semantics (Plotkin, 1981).

The rules in this paper are presented in groups, which (following Abadi and Cardelli (1996)) are referred to as 'fragments', to emphasize that they are only parts of formal systems. We define various different operational semantics for various different purposes; each semantics will be made up of several of these fragments.

In this section, we first present some basic definitions in section 4.1. In section 4.2, we define the 'liberal general' operational semantics. This semantics takes its name from its liberal attitude and the fact that it can handle general programs (with multi-clause definitions and cut).

Traditional Prolog multi-clause predicate definitions turn out to be awkward to work with in the presence of cut. Predicates defined with a single clause are more convenient to work with; but is it always possible to transform a program with multi-clause definitions into one with single-clause definitions? In section 4.3 we answer this question in the affirmative, defining a 'completed form' for programs and giving an algorithm which transforms a program to completed form. In section 4.4, we give the 'liberal completed' semantics, which is defined only for completed-form programs and is much simpler than the liberal general semantics. It is this liberal completed semantics that we use as the basis of the safer, 'conservative' semantics of the rest of the paper.

https://doi.org/10.1017/S1471068402001540 Published online by Cambridge University Press

Finally, in section 4.5, we show formally that the liberal semantics, like the Prolog systems they characterize, are problematic from a logic programming point of view because they violate not only logic, but also the weaker witness properties.

4.1 Basic definitions

This section defines some basic notions of the operational semantics, namely goal stacks, results, judgments and computations.

The judgements of the operational semantics contain *goal stack elements* and *results*. A goal stack element represents a subgoal to solve, possibly with information about how to solve it. A goal stack element can be one of the following:

- a formula;
- an expression of the form $p(t_1, \ldots, t_n)$*using*(γ), where γ is a sequence of clauses; or
- an expression of the form *body*(η), where η is a clause body (i.e. a possibly empty sequence of body elements).

A goal stack element of the form $p(t_1, \ldots, t_n)$*using*(γ) represents a predicate call along with the sequence of clauses remaining to be used in its processing; a goal stack element of the form *body*(η) represents a predicate body, possibly containing cuts. (We distinguish a predicate body from a regular sequence of formulas in this way because a body with cuts demands some special treatment.) We define a *goal stack* as a sequence of goal stack elements.

In this paper, the *result* of a computation in the operational semantics can be one of four things:

- A substitution θ, indicating a successful computation returning θ as the solution;
- *fail*, indicating failure to find a substitution;
- *flounder*, indicating that a mode restriction has been violated (see section 5); or
- *diverge*, indicating that the operational semantics believes the computation to diverge (see section 6).

Only the first two results are possible with the semantics in this section, but the others will be possible in later semantics.

A *judgement* of an operational semantics is an expression of the form $(\theta : \alpha \Rightarrow_P \rho)$, where θ is a (finite representation of a) substitution, α is a goal stack containing no free variables in the domain of θ, P is a program, and ρ is a result. A judgement indicates that the computation of the goals in α, under the current substitution θ and the program P, has the result ρ.

A *computation* in a given operational semantics is a tree, written root-down, in which each node is a judgement, and where the relationship between each node and its children is defined by the rules in that operational semantics. Computing the outcome of a Prolog goal G with respect to program P corresponds to finding a

https://doi.org/10.1017/S1471068402001540 Published online by Cambridge University Press

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\theta' : \epsilon \;\Rightarrow\; \theta'}}{[x := a, ys := [\,]] : z = zs \;\Rightarrow\; \theta'}}{[x := a] : [a] = [a|ys], z = zs \;\Rightarrow\; \theta'}}{() : a = x, [a] = [x|ys], z = zs \;\Rightarrow\; \theta'} \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\theta'' : \epsilon \;\Rightarrow\; \theta''}}{\theta'[x' := a] : z = [\,] \;\Rightarrow\; \theta''}}{\theta'[x' := a] : [\,] = [\,], z = [\,] \;\Rightarrow\; \theta''}}{\theta' : a = x', [\,] = [\,], z = [\,] \;\Rightarrow\; \theta''}}{\theta' : d(a, [\,], z)\mathit{using}(C_1; C_2; C_3) \;\Rightarrow\; \theta''}}{\theta' : d(a, [\,], z) \;\Rightarrow\; \theta''} \qquad \dfrac{}{\theta' : \mathit{body}(d(a, [\,], z)) \;\Rightarrow\; \theta''}$$

$$() : d(a, [a], z)\mathit{using}(C_2; C_3) \;\Rightarrow\; \theta''$$

$$\dfrac{\dfrac{\dfrac{}{[x := a] : [a] = [\,], z = [\,] \;\Rightarrow\; \mathit{fail}}}{() : a = x, [a] = [\,], z = [\,] \;\Rightarrow\; \mathit{fail}} \qquad \dfrac{\text{(see above)}}{() : d(a, [a], z)\mathit{using}(C_2; C_3) \;\Rightarrow\; \theta''}}{() : d(a, [a], z)\mathit{using}(C_1; C_2; C_3) \;\Rightarrow\; \theta''}$$

$$() : d(a, [a], z) \;\Rightarrow\; \theta''$$

Fig. 1. An example computation in the liberal general semantics with respect to the first 'delete' program of section 3. The computation is split into two pieces in order to fit on the page.

result ρ and a computation whose root node is $(() : G \;\Rightarrow_P\; \rho)$, where () is the empty substitution. Generally, we will drop the P subscript where its value is clear.

In the operational semantics, we use α to stand for a goal stack, and η to stand for a sequence of body elements. We use γ to stand for a sequence of clauses; to distinguish sequences of clauses more clearly from sequences of goal stack or body elements, we separate clauses in a sequence by semicolons, and goal stack or body elements by commas.

4.2 The liberal general semantics

The first operational semantics we study, as described above, is the *liberal general* semantics. It is made up of the fragments [Basic] (figure 2), [Liberal Choice] (figure 3), and [General Predicates] (figure 4). The liberal general semantics corresponds to most common implementations of Prolog, which employ hard cut and unsound negation as failure. We begin this section by looking at an example computation, and then discuss the individual rules of the liberal general semantics in more detail.

4.2.1 Example computation

Figure 1 shows an example computation in the liberal general semantics. (The clauses C_1, C_2, C_3 are the clauses for d from the three-clause version defined in section 3). The substitution θ' is $[x := a, ys := [\,], zs := z]$, and the substitution θ'' is $[x := a, ys := [\,], zs := [\,], x' := a, z = [\,]]$.) This computation, like all computations, gives the result of the computation within the same judgement as the original goal. Therefore it may not be clear how to obtain a result from knowing only the goal we want to solve. The example illustrates how we can do so in a systematic fashion by applying rules bottom-up.

We start (at the bottom) with the goal formula $d(a, [a], z)$ and the empty substitution; our task is to determine the result expression, to the right of the $\Rightarrow$

https://doi.org/10.1017/S1471068402001540 Published online by Cambridge University Press

Unif/succ:
$$\frac{\theta\sigma : \alpha\sigma \;\Rightarrow\; \rho}{\theta : (s = t), \alpha \;\Rightarrow\; \rho}$$
where σ is an mgu of s and t

Unif/fail:
$$\frac{}{\theta : (s = t), \alpha \;\Rightarrow\; \mathit{fail}}$$
where s and t are not unifiable

Success:
$$\frac{}{\theta : \epsilon \;\Rightarrow\; \theta}$$

Conj:
$$\frac{\theta : B, C, \alpha \;\Rightarrow\; \rho}{\theta : B \& C, \alpha \;\Rightarrow\; \rho}$$

Disj/nofail:
$$\frac{\theta : B, \alpha \;\Rightarrow\; \rho}{\theta : B \vee C, \alpha \;\Rightarrow\; \rho}$$
where ρ is not *fail*

Disj/fail:
$$\frac{\theta : B, \alpha \;\Rightarrow\; \mathit{fail} \qquad \theta : C, \alpha \;\Rightarrow\; \rho}{\theta : B \vee C, \alpha \;\Rightarrow\; \rho}$$

Exists:
$$\frac{\theta : B[x := x'], \alpha \;\Rightarrow\; \rho}{\theta : \exists x(B), \alpha \;\Rightarrow\; \rho}$$
where x' does not occur in the conclusion

Fig. 2. [Basic], the operational semantics rules fragment for the basic logic programming connectives.

Not/succ:
$$\frac{\theta : B \;\Rightarrow\; \theta'}{\theta : \neg B, \alpha \;\Rightarrow\; \mathit{fail}}$$

Not/fail:
$$\frac{\theta : B \;\Rightarrow\; \mathit{fail} \qquad \theta : \alpha \;\Rightarrow\; \rho}{\theta : \neg B, \alpha \;\Rightarrow\; \rho}$$

If/succ:
$$\frac{\theta : B[\vec{x} := \vec{x}'] \;\Rightarrow\; \theta' \qquad \theta' : C[\vec{x} := \vec{x}']\theta', \alpha\theta' \;\Rightarrow\; \rho}{\theta : \mathit{if}\,[\vec{x}](B, C), \alpha \;\Rightarrow\; \rho}$$
where $\vec{x}'$ do not appear in the conclusion

If/fail:
$$\frac{\theta : B[\vec{x} := \vec{x}'] \;\Rightarrow\; \mathit{fail}}{\theta : \mathit{if}\,[\vec{x}](B, C), \alpha \;\Rightarrow\; \mathit{fail}}$$
where $\vec{x}'$ do not appear in the conclusion

Fig. 3. [Liberal Choice], the operational semantics rules fragment for dealing with 'not' and 'if' in a liberal manner.

symbol. Since $d(a, [a], z)$ is a predicate call, we know that the bottommost rule is a Pred rule, that the substitution in the premise is still empty, and that the goal stack in the premise is $d(a, [a], z)\mathit{using}(C_1; C_2; C_3)$. We therefore apply that rule at the bottom of the computation. We have now reduced the problem of finding the result of $(() : d(a, [a], z))$ to that of finding the result of $(() : d(a, [a], z)\mathit{using}(C_1; C_2; C_3))$.

https://doi.org/10.1017/S1471068402001540 Published online by Cambridge University Press

Pred: $$\frac{\theta : p(t_1, \ldots, t_n) using(\gamma), \alpha \Rightarrow \rho}{\theta : p(t_1, \ldots, t_n), \alpha \Rightarrow \rho}$$

where γ is $clauses(p, P)$, renamed apart from any free variables in the conclusion

Using/cut/succ: $$\frac{\theta : s_1 = t_1, \ldots, s_n = t_n, \eta_1 \Rightarrow \theta' \quad \theta' : body(\eta_2)\theta', \alpha\theta' \Rightarrow \rho}{\theta : p(s_1, \ldots, s_n) using(C, \gamma), \alpha \Rightarrow \rho}$$

where C is of the form $p(t_1, \ldots, t_n) :- \eta_1, !, \eta_2$, and η_1 contains no cuts

Using/cut/fail: $$\frac{\theta : s_1 = t_1, \ldots, s_n = t_n, \eta_1 \Rightarrow fail \quad \theta : p(s_1, \ldots, s_n) using(\gamma), \alpha \Rightarrow \rho}{\theta : p(s_1, \ldots, s_n) using(C, \gamma), \alpha \Rightarrow \rho}$$

where C is of the form $p(t_1, \ldots, t_n) :- \eta_1, !, \eta_2$, and η_1 contains no cuts

Using/nocut/succ: $$\frac{\theta : s_1 = t_1, \ldots, s_n = t_n, \eta, \alpha \Rightarrow \theta'}{\theta : p(s_1, \ldots, s_n) using(C, \gamma), \alpha \Rightarrow \theta'}$$

where C is of the form $p(t_1, \ldots, t_n) :- \eta$, and η contains no cuts

Using/nocut/fail: $$\frac{\theta : s_1 = t_1, \ldots, s_n = t_n, \eta, \alpha \Rightarrow fail \quad \theta : p(s_1, \ldots, s_n) using(\gamma), \alpha \Rightarrow \rho}{\theta : p(s_1, \ldots, s_n) using(C, \gamma), \alpha \Rightarrow \rho}$$

where C is of the form $p(t_1, \ldots, t_n) :- \eta$, and η contains no cuts

Using/empty: $$\frac{}{\theta : p(s_1, \ldots, s_n) using(\epsilon), \alpha \Rightarrow fail}$$

Body/cut/succ: $$\frac{\theta : \eta_1 \Rightarrow \theta' \quad \theta' : body(\eta_2)\theta', \alpha\theta' \Rightarrow \rho}{\theta : body(\eta_1, !, \eta_2), \alpha \Rightarrow \rho}$$

where η_1 contains no cuts

Body/cut/fail: $$\frac{\theta : \eta_1 \Rightarrow fail}{\theta : body(\eta_1, !, \eta_2), \alpha \Rightarrow fail}$$

where η_1 contains no cuts

Body/nocut: $$\frac{\theta : \eta, \alpha \Rightarrow \rho}{\theta : body(\eta), \alpha \Rightarrow \rho}$$

where η contains no cuts

Fig. 4. [General Predicates], the operational semantics rules fragment for dealing with general (multi-clause) predicate definitions.

At this point, we can apply either the Using/nocut/succ or the Using/nocut/fail rule; we do not know which is applicable. However, we know that if Using/nocut/succ is applicable, the substitution in the left-hand premise is the empty substitution and the goal stack in the left-hand premise is $(a = x, [a] = [\,], z = [\,])$; we also know that if Using/nocut/fail is applicable, then the substitution in the (only) premise is again empty and the goal stack in the premise is again $(a = x, [a] = [\,], z = [\,])$. If the result of this goal stack is $fail$, then Using/nocut/fail is applicable; if it is some substitution θ, then Using/nocut/succ is applicable. We therefore choose as our next task to find the result of $((() : a = x, [a] = [\,], z = [\,])$.

As it turns out, in two simple steps (a Unif/succ step and a Unif/fail step) we can determine that $((() : a = x, [a] = [\,], z = [\,] \Rightarrow fail)$. Therefore we choose Using/nocut/fail as the rule to apply. This choice determines the form of the sub-

https://doi.org/10.1017/S1471068402001540 Published online by Cambridge University Press

stitution (again, the empty substitution) and the goal stack ($d(a, [a], z)using(C_2; C_3)$) in the right-hand premise. We can repeat this process of finding results in order to obtain the result θ'' of $d(a, [a], z)using(C_2; C_3)$, which is inherited by our original goal $d(a, [a], z)$ as its result. θ'' contains the mapping $[z := [\,]]$; thus the computation has correctly told us that the result of deleting a from the list $[a]$ is the empty list.

In general, whenever we are faced with a choice of two rules, the above strategy will work. The form of substitution and goal stack in one of the premises can be uniquely determined, and the choice of rule and form of substitution and goal stack in the other premise (if another is needed) can be uniquely determined from the result of the first premise. Thus, information in a computation can be seen as 'flowing' in a clockwise manner around the perimeter of the computation.

4.2.2 The rules

We now describe the general significance of the rules in the liberal general semantics in terms of how the different kinds of goal stack elements are handled.

The equality rules in the [Basic] fragment describe the usual results of unification; if unification fails, the entire goal stack fails, but if it succeeds, the computation proceeds under the mgu. The first order connective rules in [Basic] express the usual operation of Prolog interpreters. We solve a conjunction by solving each of its conjuncts in turn, left to right. We solve a disjunction by attempting to solve its left-hand disjunct and the rest of the subgoals; if this is solvable, we can ignore the right-hand disjunct, but if not, we attempt to solve that disjunct with the rest of the subgoals. Finally, we solve an existential formula (corresponding to a free variable in a clause) by renaming its variable apart from the rest of the variables in the goal.

In the [Liberal Choice] rules, we solve a negation by solving the negated formula, inverting the sense of the result at the end. This is the usual unsound strategy, which will be corrected in the system with firm cut. Similarly, the formula $if[\vec{x}](B, C)$ is computed by first computing B and checking the result. If the result is a successful computation returning satisfying substitution θ, then θ is used to compute C; otherwise, the whole formula fails. This will also be modified in the system with firm cut, in order to achieve the witness properties.

The predicate call and clause selection rules of the [General Predicates] fragment reflect how Prolog backtracks over clauses and cuts away alternate solutions. We 'launch' the processing of a predicate call by collecting the clauses in the program defining the predicate into an initial *using* expression. Then, if the first clause contains a cut, we process first only the part before the cut. On success, we retain the substitution returned and discard the other clauses, but on failure, we discard that first clause and repeat the procedure. This characterizes the behaviour of Prolog clauses with cut.

Conversely, if the first clause does not contain a cut, we process the entire clause body *along with the rest of the subgoals*. Again, on success of the goal stack we discard the other clauses, and on failure of the goal stack, the first clause. However, because we have included the rest of the subgoals in the goal stack, we retain the

https://doi.org/10.1017/S1471068402001540 Published online by Cambridge University Press

option of returning to another clause if a subgoal fails later in the computation. This characterizes the behaviour of usual Prolog clauses without cut.

Finally, the predicate body rules reflect how cuts in a clause body after the first cut may prune the search tree. If a clause body has cuts, then the portion before the first cut is processed first; if it returns a solution, we process the rest of the body with that first solution, and otherwise the entire body fails. If the body has no cuts, however, it is processed just as a sequence of formulas.

4.3 Completed forms of programs

In this section, we show that it is possible to transform any program into one in a 'completed' form, in which every predicate is defined by a single clause without cuts. This is valuable because programs in completed form are much easier to work with in the proofs we need to do. We begin by giving the transformation algorithm, show an example of how it transforms a program, and then prove the required properties of the transformation algorithm.

We say that a program is *in completed form* when each of the following conditions hold:

1. The parameters in the clause head are distinct variables;
2. There is only one clause defining each predicate;
3. The body of each clause consists of a single formula; and
4. The free variables in the body are a subset of the parameters in the head.

Our transformation of programs into completed forms depends upon the fact that our definition of formula includes the *if* connective, which allows us to achieve the effect of cuts; in fact, this is the main reason why *if* was included in the syntax and operational semantics of our language.

4.3.1 Transformation algorithm

Here, we give an algorithm which progressively transforms a program into completed form, by replacing clauses with other clauses. The program, as it is being transformed, will progressively satisfy each of the following properties.

(A) The parameters in the clause head are distinct variables.
(B) Each clause body begins and ends with a formula, and alternates formulas and cuts.
(C) Each clause has at most one cut; that is, each clause body consists of either a singleton formula F, or a sequence $F, !, G$.
(D) The last clause defining each predicate has a body which is a single formula, having no free variables except those appearing in the head.
(E) Each predicate is defined by exactly one clause.

The algorithm is as follows.

1. Choose a countable sequence of variables not appearing in the program. We will refer to these variables as $x_1, x_2, \ldots$ in the rest of the algorithm.

https://doi.org/10.1017/S1471068402001540 Published online by Cambridge University Press

2. While there is some clause in the program not of the form $(p(x_1,\ldots,x_n)\ :-\ \eta)$:
   2.1. Choose one such clause C, of the form $p(t_1,\ldots,t_{k-1},t_k,x_{k+1},\ldots,x_n)\ :-\ \eta$, where t_k is not x_k.
   2.2. If t_k is a variable y distinct from $x_1,\ldots,x_n$, then replace C in the program by $C[y := x_k]$.
   2.3. Otherwise, replace C by $p(t_1,\ldots,t_{k-1},x_k,x_{k+1},\ldots,x_n)\ :-\ (x_k = t_k),\eta$.

   (After this while loop has been completed, we can assume that property (A) above is satisfied.)
3. While there is some clause of the form $p(x_1,\ldots,x_n)\ :-\ \eta_1,F,G,\eta_2$, where F and G are formulas: choose one such clause and transform it to the form $p(x_1,\ldots,x_n)\ :-\ \eta_1,(F\&G),\eta_2$.
4. While there is some clause with an empty body: choose one such clause and replace the body by the single formula *true* (i.e., $0 = 0$).
5. While there is some clause with two consecutive cuts: choose one such clause and replace the consecutive cuts by a single cut.
6. While there is some clause beginning with a cut: choose one such clause and insert the formula *true* before the first cut.
7. While there is some clause ending with a cut: choose one such clause and insert the formula *true* after the last cut. (We can now assume that property (B) above is satisfied.)
8. While there is some clause of the form $p(x_1,\ldots,x_n)\ :-\ \eta,!,F,!,G$:
   8.1. Select one such clause.
   8.2. Select a predicate name q not appearing in the program.
   8.3. Add a clause to the program of the form $q(\vec{y})\ :-\ F,!,G$, where $\vec{y}$ are all the free variables of F,G.
   8.4. Replace the original selected clause by $p(x_1,\ldots,x_n)\ :-\ \eta,!,q(\vec{y})$.

   (We can now assume that property (C) above is satisfied.)
9. Repeat until the last clause of all predicates is of the form $p(x_1,\ldots,x_n)\ :-\ G$, where all free variables of G appear in the head:
   9.1. Choose the last clause of one predicate for which this is not the case; let it be of the form $p(x_1,\ldots,x_n)\ :-\ \eta$.
   9.2. If η is some singleton formula G, replace the clause by $p(x_1,\ldots,x_n)\ :-\ \exists\vec{y}(G)$, where $\vec{y}$ are all the free variables of G not in $x_1,\ldots,x_n$.
   9.3. Otherwise, η is a sequence of the form $F,!,G$. Replace the clause by $p(x_1,\ldots,x_n)\ :-\ if[\vec{y}](F,G)$, where $\vec{y}$ are all the free variables of F,G not in $x_1,\ldots,x_n$.

   (We can now assume that property (D) above is satisfied.)
10. While there is some predicate which is defined by more than one clause:
   10.1. Choose one such predicate p. Let the second-last clause defining p be $p(x_1,\ldots,x_n)\ :-\ \eta$, and let the last clause defining p be $p(x_1,\ldots,x_n)\ :-\ H$.

https://doi.org/10.1017/S1471068402001540 Published online by Cambridge University Press

10.2. If η is some singleton formula G, replace the two clauses by the single clause $p(x_1,\ldots,x_n)$:- $\exists\vec{y}(G) \vee H$, where $\vec{y}$ are all the free variables of G not in $x_1,\ldots,x_n$.

10.3. Otherwise, η is a sequence of the form $F, !, G$. Replace the two clauses by the single clause $p(x_1,\ldots,x_n)$:- $if\,[\vec{y}](F,G) \vee ((\neg\exists\vec{y}(F))\&H)$, where $\vec{y}$ are all the free variables of F, G not in $x_1,\ldots,x_n$.

(We can now assume that property (E) above is satisfied.)

The effect of all these steps is that we have arrived at a program in completed form, i.e. in which all predicates are defined by a single clause of the form $p(x_1,\ldots,x_n)$:- G, where the free variables of G are among $x_1,\ldots,x_n$.

Given program P, we refer to the program resulting at the end of the sequence of transformations as the *augmented Clark completion* of P, or $acc(P)$. The augmented Clark completion of P serves essentially the same purpose as the Clark completion in Clark's original treatment of negation as failure (Clark, 1978); that is, it gives a closed form of the intended meaning of each predicate. We cannot truly consider it to be a logical completion, however; the *if* construct, while it can be given a semantics consistent with the witness properties (as we will see), cannot be given a logical interpretation.

4.3.2 Example

As an example, consider the first 'delete' program from section 3:

$d(x, [\,], [\,])$
$d(x, [x|ys], zs)$:- $!, d(x, ys, zs)$
$d(x, [y|ys], [y|zs])$:- $d(x, ys, zs)$

Assume that the variables selected in Step 1 are $x_1, x_2, x_3, \ldots$. The program is transformed, by the end of Step 2, to the form:

$d(x_1, x_2, x_3)$:- $(x_2 = [\,]), (x_3 = [\,])$
$d(x_1, x_2, x_3)$:- $(x_2 = [x_1|ys]), !, d(x_1, ys, x_3)$
$d(x_1, x_2, x_3)$:- $(x_2 = [y|ys]), (x_3 = [y|zs]), d(x_1, ys, zs)$

By the end of Step 7, the program has been transformed into:

$d(x_1, x_2, x_3)$:- $(x_2 = [\,]\ \&\ x_3 = [\,])$
$d(x_1, x_2, x_3)$:- $(x_2 = [x_1|ys]), !, d(x_1, ys, x_3)$
$d(x_1, x_2, x_3)$:- $(x_2 = [y|ys]\ \&\ x_3 = [y|zs]\ \&\ d(x_1, ys, zs))$

Step 8 has no effect because there is no clause with more than one cut (this is the case in most programs). However, Step 9 scopes the local variables in the last clause, making the whole program read as follows:

$d(x_1, x_2, x_3)$:- $(x_2 = [\,]\ \&\ x_3 = [\,])$
$d(x_1, x_2, x_3)$:- $(x_2 = [x_1|ys]), !, d(x_1, ys, x_3)$
$d(x_1, x_2, x_3)$:- $\exists y \exists ys \exists zs(x_2 = [y|ys]\ \&\ x_3 = [y|zs]\ \&\ d(x_1, ys, zs))$

https://doi.org/10.1017/S1471068402001540 Published online by Cambridge University Press

Let us refer to this new body of the third clause as B_3. Step 10 first combines the last two clauses into a single clause with *if*, resulting in a new program as follows:

$$d(x_1, x_2, x_3) \text{ :- } (x_2 = [\,] \;\&\; x_3 = [\,])$$
$$d(x_1, x_2, x_3) \text{ :-}$$
$$\quad if\,[ys](x_2 = [x_1|ys], d(x_1, ys, x_3)) \vee (\neg\exists ys(x_2 = [x_1|ys]) \;\&\; B_3)$$

Let us refer to this new body of the second clause as B_2. Step 10 then continues, and transforms the remaining two clauses to the single clause

$$d(x_1, x_2, x_3) \text{ :- } (x_2 = [\,] \;\&\; x_3 = [\,]) \vee B_2$$

The program is now in completed form.

4.3.3 Properties

We now prove the properties we want the algorithm to have: that is, that it terminates, that it produces a program in completed form, and that the completed-form result program actually does the same thing as the original program.

Theorem 1 (*Completion Algorithm Termination*)
The completion algorithm terminates.

Proof
Each loop in the algorithm continues while there is a clause in the program with a specified property. The effect of each loop, however, is to eliminate all clauses with the specified property. Therefore each loop in the algorithm terminates. □

Theorem 2 (*Completed Form Formation*)
The completion algorithm produces a program in completed form.

Proof
Once the program being transformed achieves each of the properties (A)-(E), as stated in the algorithm text, it never loses those properties. The conjunction of the properties (A)-(E) is the same as saying that the program is in completed form. □

To prove that the completion algorithm preserves the results of computations, it is technically necessary to prove by induction on the structure of computations that each transformation step preserves result. For brevity, we will prove this in detail for only one of the transformations, and then argue more informally in the main proof. The following is a lemma and a theorem to do with the transformation we will prove in detail. All proofs are contained in Appendix A.

Lemma 1
Let α be a goal stack. Let α' be α with any number of occurrences of a sequence B, C in a goal stack or clause body replaced by $B\&C$, where B and C are formulas. Then $(\theta : \alpha \Rightarrow_P \rho)$ in the liberal general semantics iff $(\theta : \alpha' \Rightarrow_P \rho)$ in the liberal general semantics.

Proof
See Appendix A. □

https://doi.org/10.1017/S1471068402001540 Published online by Cambridge University Press

Pred: $$\frac{\theta : B[x_1 := t_1, \ldots, x_n := t_n], \alpha \Rightarrow \rho}{\theta : p(t_1, \ldots, t_n), \alpha \Rightarrow \rho}$$

where $p(x_1, \ldots, x_n)$:- B is the clause defining p in the completed-form program P

Fig. 5. The predicate rule for the liberal completed semantics, the only rule in the [Completed Predicates] fragment.

Lemma 2
Let P' be P with some sequence B, C in a clause body replaced by $B\&C$. Then $\theta : \alpha \Rightarrow_P \rho$ in the liberal general semantics iff $\theta : \alpha \Rightarrow_{P'} \rho$ in the liberal general semantics.

Proof
See Appendix A. □

The main result preservation theorem is as follows.

Theorem 3 (*Result Preservation of Completion Algorithm*)
The completion algorithm preserves result according to the liberal general operational semantics. That is, if P' is the completion of P, then $\theta : \alpha \Rightarrow_P \rho$ in the liberal general semantics iff $\theta : \alpha \Rightarrow_{P'} \rho$ in the liberal general semantics.

Proof
We prove the theorem by proving that each of the transformations preserves result. The lemma is used in the proof of Step 3. The details of the proof can be found in Appendix A. □

Now that we know that the completion process preserves result, we can assume that the programs we deal with will be in completed form (since if not, we have an automatic process for transforming them to completed form). We will therefore assume this for the rest of this paper.

4.4 The liberal completed semantics

Due to the complex behaviour of the Prolog cut, the liberal general operational semantics contains nine rules for predicates. These rules exist mainly to manipulate the sequences of body elements that exist in the clauses of a general program, and to backtrack over multiple clauses defining a predicate. Since we now are assuming completed-form programs, we can discard these rules in favour of one simple rule. The resulting operational semantics is referred to as the *liberal completed* semantics. Its simplicity moves us to adopt it as the standard presentation of the liberal semantics for the rest of the paper.

The liberal general semantics' nine rules for predicates were contained in the fragments [General Predicates]. The one rule replacing them is the rule contained in figure 5. We refer to the proof system fragment containing only this rule as the [Completed Predicates] fragment. Thus, the liberal completed semantics consists of the fragments [Basic], [Liberal Choice], and [Completed Predicates].

https://doi.org/10.1017/S1471068402001540 Published online by Cambridge University Press

The following result proves that it is safe to use the liberal completed semantics when we have a completed program.

Theorem 4 (*Equivalence of General and Completed Semantics*)
If P is a program in completed form, then the liberal general and liberal completed semantics have the same result. That is, $\theta : \alpha \Rightarrow_P \rho$ in the liberal general semantics iff $\theta : \alpha \Rightarrow_P \rho$ in the liberal completed semantics.

Proof
The computation in the liberal general semantics may have portions ending in applications of the Using/nocut/succ and Pred rules, of the following form:

$$\dfrac{\dfrac{\dfrac{\theta\xi : G\xi, \alpha\xi \;\Rightarrow\; \rho}{\vdots}}{\theta : \vec{t} = \vec{x}, G, \alpha \;\Rightarrow\; \rho}}{\dfrac{\theta : p(\vec{t})using(p(\vec{x}) \;:-\; G), \alpha \;\Rightarrow\; \rho}{\theta : p(\vec{t}), \alpha \;\Rightarrow\; \rho}}$$

where ξ is the substitution $[x_1 := t_n, \ldots, x_n := t_n]$. (We assume without loss of generality that the free variables of the clause are distinct from those of the conclusion.) This portion of the computation in the liberal completed semantics will have the following form:

$$\dfrac{\theta : G\xi, \alpha \;\Rightarrow\; \rho'}{\theta : p(\vec{t}), \alpha \;\Rightarrow\; \rho'}$$

where ρ' differs from ρ only in that it does not contain substitutions for the renamed variables arising from clauses. Since the substitution ξ deals only with the x_i variables, which do not appear in α, the uppermost judgements in the two computations are essentially identical.

The computation in the liberal general semantics may also have portions ending in a sequence of applications of the Using/empty, Using/nocut/fail and Pred rules, of the following form:

$$\dfrac{\dfrac{\dfrac{\theta\xi : G\xi, \alpha\xi \;\Rightarrow\; fail}{\vdots}}{\theta : \vec{t} = \vec{x}, G, \alpha \;\Rightarrow\; fail} \qquad \dfrac{}{\theta : p(\vec{t})using(), \alpha \;\Rightarrow\; fail}}{\dfrac{\theta : p(\vec{t})using(p(\vec{x}) \;:-\; G), \alpha \;\Rightarrow\; fail}{\theta : p(\vec{t}), \alpha \;\Rightarrow\; fail}}$$

where ξ is the substitution $[x_1 := t_n, \ldots, x_n := t_n]$. This portion of the computation in the liberal completed semantics will have the following form:

$$\dfrac{\theta : G\xi, \alpha \;\Rightarrow\; fail}{\theta : p(\vec{t}), \alpha \;\Rightarrow\; fail}$$

Again, the substitution ξ does not affect α. □

https://doi.org/10.1017/S1471068402001540 Published online by Cambridge University Press

$$\dfrac{\dfrac{[ys := [\,]] : \epsilon \;\Rightarrow\; [ys := [\,]]}{() : [a] = [a|ys] \;\Rightarrow\; [ys := [\,]]} \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\theta'' : \epsilon \;\Rightarrow\; \theta''}{[ys := [\,]] : z = [\,] \;\Rightarrow\; \theta''}}{[ys := [\,]] : [\,] = [\,], z = [\,] \;\Rightarrow\; \theta''}}{[ys := [\,]] : ([\,] = [\,] \& z = [\,]) \;\Rightarrow\; \theta''}}{[ys := [\,]] : ([\,] = [\,] \& z = [\,]) \vee B_2 \vee B_3 \;\Rightarrow\; \theta''}}{[ys := [\,]] : d(a, [\,], z)) \;\Rightarrow\; \theta''}}{() : if\,[ys]([a] = [a|ys], d(a, ys, z)) \;\Rightarrow\; \theta''}}{() : if\,[ys]([a] = [a|ys], d(a, ys, z)) \vee B_3 \;\Rightarrow\; \theta''}$$

$$\dfrac{\dfrac{}{() : [a] = [\,] \& z = [\,] \;\Rightarrow\; fail} \qquad \dfrac{\text{(see above)}}{() : if\,[ys]([a] = [a|ys], d(a, ys, z)) \vee B_3 \;\Rightarrow\; \theta''}}{\dfrac{() : ([a] = [\,] \& z = [\,]) \vee B_2 \vee B_3 \;\Rightarrow\; \theta''}{() : d(a, [a], z) \;\Rightarrow\; \theta''}}$$

Fig. 6. A sample computation in the liberal completed semantics. $B_1 \vee B_2 \vee B_3$ is the body of the clause defining d from the second program in section 3, with parameters instantiated. Not all substitutions are listed in full.

$$\dfrac{\dfrac{\dfrac{\dfrac{[x := 0] : \epsilon \;\Rightarrow\; [x := 0]}{() : x = 0 \;\Rightarrow\; [x := 0]}}{() : \neg(x = 0) \;\Rightarrow\; fail} \qquad \dfrac{\dfrac{[x := 1] : \epsilon \;\Rightarrow\; [x := 1]}{() : x = 1 \;\Rightarrow\; [x := 1]}}{() : \neg(\neg(x = 0)), x = 1 \;\Rightarrow\; [x := 1]}}{() : \neg(\neg(x = 0))\&x = 1 \;\Rightarrow\; [x := 1]}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{() : \epsilon \;\Rightarrow\; ()}}{() : 0 = 0 \;\Rightarrow\; ()}}{() : \neg(0 = 0) \;\Rightarrow\; fail} \qquad \dfrac{}{() : 0 = 1 \;\Rightarrow\; fail}}{\dfrac{() : \neg(\neg(0 = 0)), 0 = 1 \;\Rightarrow\; fail}{() : \neg(\neg(0 = 0))\&0 = 1 \;\Rightarrow\; fail}}$$

$$\dfrac{\dfrac{\dfrac{}{() : a = 0 \;\Rightarrow\; fail} \qquad \dfrac{}{() : \epsilon \;\Rightarrow\; ()}}{() : \neg(a = 0) \;\Rightarrow\; ()}}{\dfrac{() : \neg(\neg(a = 0)), a = 1 \;\Rightarrow\; fail}{() : \neg(\neg(a = 0))\&a = 1 \;\Rightarrow\; fail}}$$

Fig. 7. Computations showing that the goal $\neg(\neg(x = 0))\&x = 1$ violates the success property in the liberal completed semantics. a is some arbitrary ground term not identical to 0.

Figure 6 shows a sample computation in the liberal completed semantics, using the second, one-clause version of the delete program from section 3. (θ'' is the substitution $[ys := [\,], z := [\,]]$.) Note that although the number of steps is similar to that of the liberal general computation, now the elements of a goal stack are simply formulas. This will simplify our analysis, since we can focus on formulas rather than having to deal with the interaction of formulas and sequences of clauses with cuts.

4.5 Inadequacy of liberal semantics

Because it is intended to capture the behaviour of Prolog programs with cut, the liberal completed semantics does not have either of the witness properties. Figure 7

https://doi.org/10.1017/S1471068402001540 Published online by Cambridge University Press

$$\dfrac{\dfrac{\dfrac{\dfrac{}{[x := 0] : \epsilon \;\Rightarrow\; [x := 0]}}{() : x = 0 \;\Rightarrow\; [x := 0]}}{() : \neg(x = 0), x = 1 \;\Rightarrow\; \mathit{fail}}}{() : \neg(x = 0) \& x = 1 \;\Rightarrow\; \mathit{fail}}$$

$$\dfrac{\dfrac{\dfrac{}{() : 1 = 0 \;\Rightarrow\; \mathit{fail}} \qquad \dfrac{\dfrac{}{() : \epsilon \;\Rightarrow\; ()}}{() : 1 = 1 \;\Rightarrow\; ()}}{() : \neg(1 = 0), 1 = 1 \;\Rightarrow\; ()}}{() : \neg(1 = 0) \& 1 = 1 \;\Rightarrow\; ()}$$

Fig. 8. Computations showing that the goal $\neg(x = 0)\&x = 1$ violates the failure property in the liberal completed semantics.

shows that the goal formula $G_1 \equiv \neg(\neg(x = 0))\&x = 1$ succeeds in the liberal completed semantics, even though $G_1[x := 0]$ fails and $G_1[x := a]$, where a is any arbitrary ground term not identical to 0, fails. Similarly, figure 8 shows that the goal formula $G_2 \equiv \neg(x = 0)\&x = 1$ fails in the liberal completed semantics, even though $G_2[x := 1]$ succeeds.

This is consistent with the behaviour of the usual unsound implementation of negation as failure. We can, of course, ban unsound NAF alone with a mode restriction similar to that of Stärk (1998); however, if we retain the general *if* construct (corresponding to the hard cut), we will still permit behaviour which violates the witness properties. This suggests that we need some further restriction to *if* analogous to Stärk's restriction on negation.

Note that these counterexamples also show that the liberal general semantics (a generalization of the liberal completed semantics) has neither of the witness properties.

5 The conservative operational semantics

In the last section, we gave operational semantics for programs which characterized Prolog computation, but were inadequate from a logic-programming point of view because they violated the witness properties. In this section, we repair the faults of the liberal semantics by placing simple restrictions on some of its rules. The result is the *conservative* semantics, which does enjoy the witness properties. We refer to the form of cut embodied in the conservative semantics as *firm* cut.

In section 5.1, we present and describe the rules for the conservative semantics, and in section 5.2 we prove useful properties of it, including the witness properties. Finally, in section 5.3 we show that the firm cut still permits the useful first-solution behaviour of the Prolog cut.

5.1 The conservative semantics rules

The conservative operational semantics restricts the computation of negation and *if*. Whereas the liberal completed semantics is made up of the rules fragments [Basic] (figure 2), [Liberal Choice] (figure 3), and [Completed Predicates] (figure 5), the

https://doi.org/10.1017/S1471068402001540 Published online by Cambridge University Press

Not/succ:
$$\frac{\theta : B \;\Rightarrow\; \theta'}{\theta : \neg B, \alpha \;\Rightarrow\; \mathit{fail}}$$
where B has no free variables

Not/fail:
$$\frac{\theta : B \;\Rightarrow\; \mathit{fail} \qquad \theta : \alpha \;\Rightarrow\; \rho}{\theta : \neg B, \alpha \;\Rightarrow\; \rho}$$
where B has no free variables

Not/flounder:
$$\frac{}{\theta : \neg B, \alpha \;\Rightarrow\; \mathit{flounder}}$$
where B has free variables

Not/sub:
$$\frac{\theta : B \;\Rightarrow\; \rho}{\theta : \neg B, \alpha \;\Rightarrow\; \rho}$$
where B has no free variables, and ρ is *flounder* or *diverge*

If/succ:
$$\frac{\theta : B[\vec{x} := \vec{x}'] \;\Rightarrow\; \theta' \qquad \theta' : C[\vec{x} := \vec{x}']\theta', \alpha \;\Rightarrow\; \rho}{\theta : \mathit{if}\,[\vec{x}](B, C), \alpha \;\Rightarrow\; \rho}$$
where $\exists\vec{x}(B)$ has no free variables, and $\vec{x}'$ do not appear in the conclusion

If/fail:
$$\frac{\theta : B[\vec{x} := \vec{x}'] \;\Rightarrow\; \mathit{fail}}{\theta : \mathit{if}\,[\vec{x}](B, C), \alpha \;\Rightarrow\; \mathit{fail}}$$
where $\exists\vec{x}(B)$ has no free variables, and $\vec{x}'$ do not appear in the conclusion

If/flounder:
$$\frac{}{\theta : \mathit{if}\,[\vec{x}](B, C), \alpha \;\Rightarrow\; \mathit{flounder}}$$
where $\exists\vec{x}(B)$ has free variables

If/sub:
$$\frac{\theta : B[\vec{x} := \vec{x}'] \;\Rightarrow\; \rho}{\theta : \mathit{if}\,[\vec{x}](B, C), \alpha \;\Rightarrow\; \rho}$$
where $\exists\vec{x}(B)$ has no free variables, and $\vec{x}'$ do not appear in the conclusion, and ρ is *flounder* or *diverge*

Fig. 9. The rules of the [Conservative Choice] fragment, for computing the choice constructs in a more restricted fashion.

$$\frac{\frac{}{() : \neg(x = 0), x = 1 \;\Rightarrow\; \mathit{flounder}}}{() : \neg(x = 0) \& x = 1 \;\Rightarrow\; \mathit{flounder}}$$

Fig. 10. The safe computation of $\neg(x = 0) \& x = 1$ in the conservative semantics.

conservative semantics is made up of the rules fragments [Basic], [Conservative Choice], and [Completed Predicates]. The rules for the new fragment, [Conservative Choice], are in figure 9.

Consider the rules Not/succ and Not/fail from [Conservative Choice]. These rules are the same as those of the [Liberal Choice] fragment, except that they have the restriction that B (the negated formula) must have no free variables. When B does have free variables, a new rule, Not/flounder, applies. Not/flounder states that a goal stack beginning with a negated formula with free variables immediately returns a new result, *flounder*, indicating that the computation cannot continue at this point.

Another new rule, Not/sub, defines what happens when B has no free variables, but the sub-computation itself flounders: the *flounder* result is passed on. Note that

https://doi.org/10.1017/S1471068402001540 Published online by Cambridge University Press

the rules in the [Basic] and [Completed Predicates] fragments are already described in such a way that they also automatically pass on the new *flounder* result. Hence, *flounder* acts as a kind of run-time exception, which causes the computation to terminate immediately.[2]

The conservative rules for the *if* connective are constructed from those of the liberal rules in a similar manner, modulo the bound variables of the *if*. As an example of a conservative computation, consider again the goal $\neg(x = 0)\&x = 1$, which was a problem for the liberal semantics. Figure 10 shows that the conservative semantics handles it in a sound way, by immediately stating that it flounders.

We should note at this point that there are other approaches to the problem of handling negation in a sound way. Loveland and Reed (1991), for example, define a resolution method by which queries against programs with negation can be evaluated in a sound and complete manner. Dahl (1980) defines an approach which delays the evaluation of a negated goal until it becomes ground, and an approach which, within a negated goal's computation, blocks only the unification of variables which are free outside the scope of the negation. Di Pierro *et al.* (1995) define an approach in which an existentially closed negated atom (a formula of the form $\exists[\neg A]$) succeeds iff all branches of the SLD-tree of the atom either fail or instantiate the atom. Some of these methods have been implemented in a variety of systems, for instance in Naish's NU-Prolog (Naish, 1986). Here we are motivated by our interest in the features implemented in the most widely-used Prolog systems. Most Prolog systems implement the simple negation as failure characterized by the liberal semantics and restricted by the conservative semantics.

5.2 Properties of the conservative semantics

In this section, we prove the properties of the conservative operational semantics that we wanted to hold. First, we prove the correspondence of computations in the liberal and the conservative semantics. Then, we prove the witness properties.

5.2.1 Correspondence of computations

First, we note that successful and failing computations in the conservative semantics correspond to successful and failing computations in the liberal completed semantics.

Theorem 5
If $\theta : \alpha \Rightarrow \rho$ in the conservative semantics, and ρ is not *flounder*, then $\theta : \alpha \Rightarrow \rho$ in the liberal completed semantics.

Proof
Any computation in the conservative semantics which contains applications of the Not/flounder or If/flounder rules must result in *flounder*, since the *flounder* outcomes of these rules descend through all other rules in the [Basic], [Conservative choice] and [Completed predicates] fragments. Therefore if a computation in the

[2] Not/sub also passes on the result *diverge*, which is not needed until section 6.2.2.

https://doi.org/10.1017/S1471068402001540 Published online by Cambridge University Press

conservative semantics does not result in *flounder*, it must not use those rules; rather, it uses only the other Not and If rules, which are restrictions of those in the liberal completed semantics, and the other rules, which are identical to those in the liberal completed semantics. Such a computation is, in fact, a computation in the liberal completed semantics. □

The converse does not hold, since successful and failing computations in the liberal completed semantics may flounder in the conservative semantics. However, all computations in the liberal completed semantics do correspond to some kind of computations in the conservative semantics, as the next theorem shows.

Theorem 6

If $\theta : \alpha \Rightarrow \rho$ in the liberal completed semantics, then there is some ρ' such that $\theta : \alpha \Rightarrow \rho'$ in the conservative semantics, and ρ' is either ρ or *flounder*.

Proof

By induction on the structure of the liberal completed computation. Cases are on the bottommost rule application.

All applications of rules with 0 premises correspond to rule applications in the conservative semantics.

If the bottommost rule is Disj/fail: the bottommost judgement is of the form $(\theta : B \vee C, \alpha \Rightarrow \mathit{fail})$, and its left-hand premise judgement is of the form $(\theta : B, \alpha \Rightarrow \mathit{fail})$. By the induction hypothesis (IH), either $(\theta : B, \alpha \Rightarrow \mathit{fail})$ in the conservative semantics, or $(\theta : B, \alpha \Rightarrow \mathit{flounder})$ in the conservative semantics. In the first case, the result follows directly from another application of the IH; in the second case, the result follows from one application of the Disj/nofail rule.

The cases for the Not and If rules are similar to that of Disj/fail. Applications of all other rules in the liberal completed computation have exactly one premise, and correspond to applications of the same rules in the conservative computation. □

Examples of goals whose outcomes differ in the liberal completed and conservative semantics are as follows:

- The goal $\neg\neg(x = 0)$ succeeds in the liberal completed semantics, but flounders in the conservative semantics.
- The goal $\neg(x = 0)$ fails in the liberal completed semantics, but flounders in the conservative semantics.
- The goal $\neg\neg(x = 0)\&\mathit{loop}(x)$, where the predicate *loop* is defined with the definition $\mathit{loop}(x)$:- $\mathit{loop}(x)$, diverges (does not have any finite computation) with respect to the liberal completed semantics; however, it flounders in the conservative semantics.

These examples, along with the witness properties to be proven next, show that although strictly fewer goals succeed or fail in the conservative semantics, strictly more goals terminate in the conservative semantics.

5.2.2 *The witness properties*

Finally, we show the witness properties of the conservative semantics. Most proofs are contained in full in Appendix A.

We begin with some useful definitions. We say that θ *is a specialization of* θ', in symbols $\theta \subseteq \theta'$, if there is some θ'' such that $x\theta \equiv x\theta'\theta''$, for all variables x in the domain of θ'. Given a set V of variables and a substitution θ, we say that a substitution ξ *grounds* V *consistent with* θ if $\xi \subseteq \theta$ and $x\xi$ is ground for every $x \in V$.

An inductive generalization of the failure property can be proven directly; the corresponding generalization of the success property requires a technical lemma. These three lemmas are as follows.

Lemma 3 (*General Failure Property of Conservative Semantics*)
Let θ, α be such that $(\theta : \alpha \Rightarrow \mathit{fail})$ in the conservative semantics. Then for any ξ, $(\theta : \alpha\xi \Rightarrow \mathit{fail})$ in the conservative semantics.

Proof
See Appendix A. □

Lemma 4 (*Substitution Monotonicity of Conservative Semantics*)
Let θ, α be such that $\alpha\theta \equiv \alpha$ and $\theta : \alpha \Rightarrow \theta'$ in the conservative semantics. Then $\theta' \subseteq \theta$.

Proof
See Appendix A. □

Lemma 5 (*General Success Property of Conservative Semantics*)
Let θ, α be such that $\theta : \alpha \Rightarrow \theta'$ in the conservative semantics. Let V be a subset of the free variables of α. Then for any ξ grounding V consistent with θ', $\theta : \alpha\xi \Rightarrow \theta'\xi$ in the conservative semantics.

Proof
See Appendix A. □

We can now state and prove the witness properties mentioned in the Introduction for the conservative semantics. First, we define more precisely what we mean by success and failure.

We say that a goal G *succeeds* (in the conservative semantics) if there is a computation with a conclusion of the form $() : G \Rightarrow \theta'$. We say that a goal G *fails* if there is a computation with a conclusion of the form $() : G \Rightarrow \mathit{fail}$.

Theorem 7 (*Witness Properties of the Conservative Semantics*)

(1) If a goal G succeeds, then some ground instance of G succeeds.
(2) If a goal G fails, then any ground instance of G fails.

Proof
(1) If G succeeds, this means there is a θ' such that $() : G \Rightarrow \theta'$. Let σ be the substitution which substitutes all free variables of $G\theta'$ by 0. Let ξ be the substitution

https://doi.org/10.1017/S1471068402001540 Published online by Cambridge University Press

which substitutes any variable $x \in FV(G)$ by $x\theta'\sigma$. Then ξ grounds $FV(G)$ consistent with θ'. By the General Success Property, we have that $() : G\xi \Rightarrow \theta'\xi$. Thus the ground instance $G\xi$ of G succeeds.

(2) If G fails, then $() : G \Rightarrow fail$. By the General Failure Property, for any ξ, including those grounding all variables in $FV(G)$, we have that $() : G\xi \Rightarrow fail$. Thus all ground instances of G fail. □

5.3 Implementation issues

In this section, we discuss some implementation-related issues. We show that the conservative semantics retains the desirable first-solution behaviour of the Prolog hard cut. We also discuss the possibility of turning the mode restriction of the conservative semantics into a static rather than a dynamic one.

5.3.1 First solution behaviour

When we have a formula of the form $if[\vec{x}](B, C)$, the conservative operational semantics allows the $\vec{x}$ variables to pass on to C, and allows free variables other than $\vec{x}$ in C; however, only the first successful substitution for $\vec{x}$ is passed on. The conservative semantics therefore still allows the useful 'first solution' behaviour which *if* has inherited from cut.

For an example of this behaviour, consider the following problem. We define an *association list* as a list of terms of the form $a(k, j)$, where k is a key and j is a value associated with it. A problem commonly encountered in symbolic programming is to extract the first value (and only the first value) associated with a key in an association list, which is taken as the 'current' value of the key. We can write the standard logic programming 'member' predicate as

$$m(x, y) \ :- \ \exists yh \exists yt(y = [yh|yt] \& (x = yh \vee m(x, yt)))$$

and then write a predicate which solves the first-value problem as follows:

$$v(x, y, z) \ :- \ if[w](m(a(y, w), x), z = w)$$

The predicate call $v(x, y, z)$, where x is an association list, y is a key, and z is any term, succeeds iff z is the first value associated with y in x.

The query $v([a(b, 0), a(b, 1)], b, z)$ to this program should result in the binding $[z := 0]$, since this is the first value returned by m as associated with the key b in the list. However, the query $v([a(b, 0), a(b, 1)], b, 1)$ to this program should fail; even though the value 1 is associated with b later in the list, *if* should select only the first solution. Figure 11 shows that this is indeed the behaviour of the conservative semantics.

We could evidently get closer to the liberal general semantics by allowing the first subformula of the *if* to be computed with free variables, as long as those variables do not get bound in the course of the computation, as suggested by one of Dahl's negation strategies (Dahl, 1980) and Di Pierro *et al.* (1995). Since this would complicate the operational semantics and our analysis, we have decided to stick with the conservative semantics as given.

https://doi.org/10.1017/S1471068402001540 Published online by Cambridge University Press

Computation of m subgoal:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{[yh := a(b,0), yt := [a(b,1)], w := 0] : \epsilon \;\Rightarrow\; [w := 0]}}{[yh := a(b,0), yt := [a(b,1)]] : a(b,0) = a(b,w) \;\Rightarrow\; [w := 0]}}{[yh := a(b,0), yt := [a(b,1)]] : a(b,0) = a(b,w) \vee m(a(b,w), [a(b,1)] \;\Rightarrow\; [w := 0]}}{() : [a(b,0), a(b,1)] = [yh|yt], (yh = a(b,w) \vee m(a(b,w), yt) \;\Rightarrow\; [w := 0]}}{() : [a(b,0), a(b,1)] = [yh|yt] \& (yh = a(b,w) \vee m(a(b,w), yt) \;\Rightarrow\; [w := 0]}}{() : \exists yt([a(b,0), a(b,1)] = [yh|yt] \& (yh = a(b,w) \vee m(a(b,w), yt))) \;\Rightarrow\; [w := 0]}}{() : \exists yh \exists yt([a(b,0), a(b,1)] = [yh|yt] \& (yh = a(b,w) \vee m(a(b,w), yt))) \;\Rightarrow\; [w := 0]}$$

$$() : m(a(b,w), [a(b,0), a(b,1)]) \;\Rightarrow\; [w := 0]$$

Successful computation:

$$\dfrac{\dfrac{\dfrac{\text{(see above)}}{() : m(a(b,w), [a(b,0), a(b,1)]) \;\Rightarrow\; [w := 0]} \qquad \dfrac{\dfrac{}{[w := 0, z := 0] : \epsilon \;\Rightarrow\; [z := 0]}}{[w := 0] : z = 0 \;\Rightarrow\; [z := 0]}}{() : if\,[w](m(a(b,w), [a(b,0), a(b,1)]), z = w) \;\Rightarrow\; [z := 0]}}{() : v([a(b,0), a(b,1)], b, z) \;\Rightarrow\; [z := 0]}$$

Failing computation:

$$\dfrac{\dfrac{\text{(see above)}}{() : m(a(b,w), [a(b,0), a(b,1)]) \;\Rightarrow\; [w := 0]} \qquad \dfrac{}{[w := 0] : 1 = 0 \;\Rightarrow\; fail}}{() : if\,[w](m(a(b,w), [a(b,0), a(b,1)]), 1 = w) \;\Rightarrow\; fail}$$

$$() : v([a(b,0), a(b,1)], b, 1) \;\Rightarrow\; fail$$

Fig. 11. Examples showing first-solution behaviour of conservative semantics. (Some substitutions are simplified for clarity.) Top: a computation returning the first solution to a call to the membership predicate. Middle: a computation showing that the first solution is selected by *if*. Bottom: a computation showing that subsequent solutions are not selected by *if*.

5.3.2 Static analysis

The conservative operational semantics restricts the behaviour of the logic programming system by essentially enforcing mode checks at run time. However, we do not believe that there is any obstacle to doing static mode checking (see, for example, Barbuti and Martelli (1990), Apt and Marchiori (1994), and Gabbrielli and Etalle (1999)) to catch programs at compile time which could result in floundering goals. (In Andrews (1999), a static analysis scheme is proposed which does a fine-grained analysis in order to reject as few programs as possible, at the expense of some complexity.)

Because the conservative semantics behaves identically to the liberal semantics on non-floundering goals, and because the liberal semantics characterizes Prolog, we believe that an implementation of firm cut is achievable simply by imposing static mode restrictions on a conventional logic programming system. For the sake of brevity, we do not explore this issue further here, but assume in the rest of the paper that such a static analysis system is possible.

6 The abstract semantics

In this section, we present an abstract semantics for the conservative operational semantics. The abstract semantics does not reify such notions as substitution sequence

https://doi.org/10.1017/S1471068402001540 Published online by Cambridge University Press

and unification; rather, the central element of the semantics which deals with free variables is the interpretation of the existential quantifier by a valuation function of the same form as those of classical truth theory (Kripke, 1975; Fitting, 1985). This suggests that the conservative semantics and firm cut have a deeper connection to logic than simply permitting some logical computations.

The abstract semantics is in the UNV (unfolding-normal-form-valuation) style (Andrews, 1997), and it depends upon the witness properties to achieve soundness and completeness. In UNV semantics, we associate a truth value to a goal; the truth value can be described as the maximally defined truth value among the valuations of the normal forms of the unfoldings of the goal. We doubt that it is possible to give such a semantics for the liberal semantics and thus for Prolog with hard cut, due to those systems' failure to achieve the witness properties.

We begin with an overview of UNV semantics in section 6.1 containing some basic definitions, including that of an (operational) *outcome* of a goal G with respect to a program P, $outcome_P(G)$. Section 6.1 also contains a 'roadmap' of the series of results that follow, referred to as the 'raising lemmas'. In sections 6.2 through 6.5 we proceed, through the raising lemmas, to systematically raise the characterizing expression for $outcome_P(G)$ to greater and greater levels of abstraction, until all operational notions have been abstracted away.

Finally, in section 6.5, we link the previous raising lemmas into a final characterization of outcome of a general goal with respect to a program, and give an expression describing the abstract denotation of a program. We conclude with an example, in section 6.6, and some discussion in section 6.7.

In this section, whenever we refer to a program P and a goal G, we assume that G does not yield the *flounder* result. It may also be possible to characterize the *flounder* result, as in, for instance, Andrews (1997). However, for simplicity, here we assume that programs will be subject to a static analysis which excludes those able to generate such a result, as discussed in section 5.3.2.

6.1 UNV semantics

Here we give an overview of UNV semantics and some basic definitions which will be used throughout the section. We also give a 'roadmap' of the results which will be proven.

6.1.1 Overview

The UNV semantics given here is based on six basic notions:

- The three *truth values* T, F and U, or 'true', 'false, and 'undefined'.
- The *definedness ordering* on truth values, which ranks T and F as being more defined than U.
- The *alethic* or *truth ordering* on truth values, which ranks U as 'more true' than F and T as 'more true' than U.
- The *unfoldings* of a goal, which are the formulas obtained from the goal by expanding zero or more predicate calls, possibly repeatedly.

https://doi.org/10.1017/S1471068402001540 Published online by Cambridge University Press

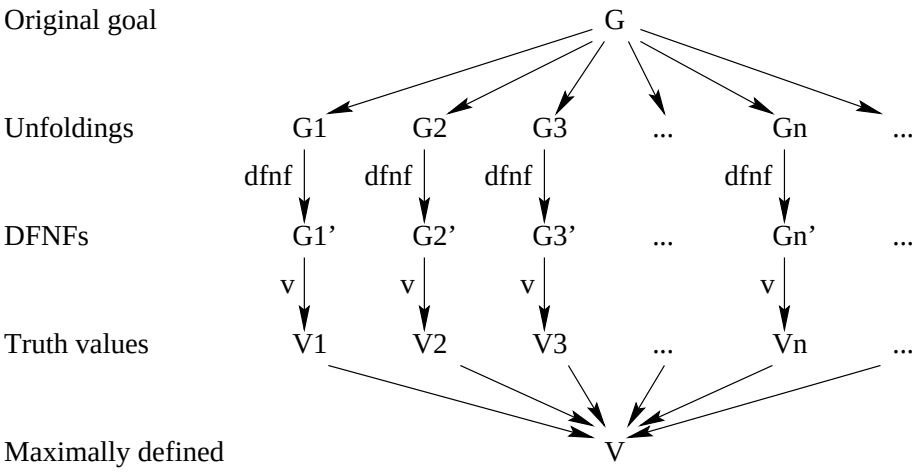

Fig. 12. Diagram of the basic notions of UNV (unfolding-normal-form-valuation) semantics.

- The *depth-first normal form*, or DFNF, of a goal, which is a formula closely related to the disjunctive normal form (DNF) of the goal.
- The *valuation* $v(G)$ of a goal G in DFNF, which is a compositional function from formulas to truth values.

The last three of these will be given more precise and detailed definitions in the course of this section.

A schematic diagram of the basic notions of UNV semantics is contained in figure 12. Given a goal G, we consider all the (possibly infinitely many) unfoldings $G_1, G_2, G_3, \ldots, G_n, \ldots$ of the goal. Then, we find the DFNFs of all the unfoldings, resulting in the normal-form goals $G'_1, G'_2, G'_3, \ldots, G'_n, \ldots$. We apply the valuation function v to the normal-form goals, getting a set $V_1, V_2, V_3, \ldots, V_n, \ldots$ of truth values, each of them equal to either T, F, or U. (The alethic ordering of truth values is used to compute the valuation of existentially-quantified goals.) There will be one unique maximally defined truth value in this set; this will be taken as the truth value of the original goal G.

6.1.2 Outcomes of goals

When we evaluate a goal in a logic programming system, we expect to receive a substitution (if one exists) as the result of the evaluation. However, when we prove properties of logic programs, we are more interested in proving whether a general pattern of goals succeeds or fails; we are less interested in obtaining substitutions, because there may be a different substitution for each different instance of the pattern. Hence, in this paper (as in Andrews (1991, 1997) and Stärk (1998)) we take the 'observable' of interest to be whether a goal succeeds, fails or diverges, linking these observables to the truth values T, F and U respectively.

https://doi.org/10.1017/S1471068402001540 Published online by Cambridge University Press

We therefore define the *outcome* of a goal G with respect to P, $outcome_P(G)$, as follows.

- If there is a θ' such that $((): G \Rightarrow_P \theta')$ in the conservative operational semantics, then $outcome_P(G) = T$.
- If $((): G \Rightarrow_P fail)$ in the conservative operational semantics, then $outcome_P(G) = F$.
- Otherwise (i.e. if there is no result ρ such that $((): G \Rightarrow_P \rho)$ in the conservative semantics), then $outcome_P(G) = U$.

This notion of outcome will be what is characterized by the abstract, UNV semantics.

For use in the raising lemmas, we will also need the closely-related notion of 'pessimistic outcome' $outcome^\frown(G)$ of a goal G. This is what the outcome of G would be, independent of the program, if we were to pessimistically assume that all predicates in the program would diverge (result in infinite computations). This notion will be defined more precisely below.

6.1.3 Roadmap

Here we present a guide to the characterization results that follow. The sequence of raising lemmas we will prove will be as follows:

1. The outcome of a goal G with respect to a program P can be obtained by inspecting all the pessimistic outcomes of all the unfoldings of G, and taking the maximally defined one. ($outcome_P(G) = max_k(\{outcome^\frown(G') \mid G' \text{ is an } P\text{-unfolding of } G\})$.)
2. The pessimistic outcome of a goal G is the same as the pessimistic outcome of its depth-first normal form. ($outcome^\frown(G) = outcome^\frown(dfnf(G))$.)
3. The pessimistic outcome of a goal G in depth-first normal form can be characterized by a compositional valuation function (function from goals to truth values), v. ($outcome^\frown(G) = v(G)$.)
4. Putting the previous three raising lemmas together, the outcome of G with respect to P, $outcome_P(G)$, can be alternatively characterized by the expression $max_k(\{v(dfnf(G')) \mid G' \text{ is a } P\text{-unfolding of } G\})$.

This final result gives an abstract view of the meaning of a program, which allows us to define the program's denotation, concluding the characterization.

6.2 Unfoldings and the pessimistic semantics

In this section, we define the notion of unfolding of a goal, and also define the pessimistic operational semantics, which treats all predicates as being divergent. We then show how the two notions are related by proving that every terminating goal has some unfolding which terminates even in the pessimistic semantics. This property is useful because it allows us to abstract away (into the notion of unfolding) all consideration of the program, and concentrate on characterizing outcomes under the program-independent pessimistic semantics.

https://doi.org/10.1017/S1471068402001540 Published online by Cambridge University Press

Pred: $$\frac{}{\theta : p(t_1, \ldots, t_n), \alpha \Rightarrow \mathit{diverge}}$$

Fig. 13. The predicate rule for the pessimistic semantics, the only rule in the [Pessimistic Predicates] fragment.

We then draw upon the standard notion of definedness ordering of truth values in order to get a succinct characterization of this relationship. The section concludes with the first raising lemma.

6.2.1 Unfoldings

Informally, an unfolding of a goal is the goal after some predicate calls are replaced by the corresponding predicate bodies, possibly repeatedly. The notion comes originally from Burstall and Darlington's corresponding functional programming notion (Burstall and Darlington, 1977), and is analogous to Tamaki and Sato's notion of unfolding of a program (Tamaki and Sato, 1984). Unfoldings are also used in the unfolding semantics of Gabbrieli and Levi (1992), and in other semantics such as Etalle's for modular general logic programs (Etalle, 1998).

More formally, given a program P in completed form, a formula G' is a *1-P-unfolding* of G if it is G with one occurrence of $p(t_1, \ldots, t_n)$ replaced by $B[x_1 := t_1, \ldots, x_n := t_n]$, where $(p(x_1, \ldots, x_n) \text{ :- } B)$ is a definition in P. A formula G' is a *P-unfolding* of G if it is either G itself, or a P-unfolding of a 1-P-unfolding of G. We will drop the program name P when it is unimportant or clear from context. Clearly, the P-unfolding operation, seen as a rewriting, is confluent.

For instance, let the program P consist of the definitions $(q \text{ :- } r)$ and $(p \text{ :- } q\&p)$. Then the goal $G = (q \vee p)$ has two 1-P-unfoldings, namely $(r \vee p)$ and $(q \vee (q\&p))$. G has an infinite number of P-unfoldings, including G itself, its two 1-P-unfoldings, and other unfoldings such as $(r \vee (q\&(r\&p)))$.

We define a P-unfolding of a *sequence* $G_1, \ldots, G_n$ of formulas as any sequence $G'_1, \ldots, G'_n$ of formulae in which G'_i is a P-unfolding of G_i, for all $1 \leqslant i \leqslant n$.

6.2.2 The pessimistic semantics

If we unfold a succeeding or failing goal enough, we obtain a goal which succeeds or fails without doing any predicate expansions. A divergent goal, however, cannot be unfolded to a point where it succeeds or fails without predicate expansions.

These facts suggest the following analytical framework. We define an operational semantics, the *pessimistic* semantics, which returns the result *diverge* on any predicate call. We can then characterize a successful goal as one with an unfolding which succeeds in the pessimistic semantics, a failing goal as one with an unfolding which fails in the pessimistic semantics, and a divergent goal as one with no unfolding which returns anything but *diverge* in the pessimistic semantics.

To this end, we define the pessimistic operational semantics as being made up of the the operational semantics fragments [Basic], [Conservative Choice], and

https://doi.org/10.1017/S1471068402001540 Published online by Cambridge University Press

[Pessimistic Predicates], where the latter fragment consists of the single rule shown in figure 13. Note that the rules in [Basic] and [Conservative Choice] are described in such a way that they pass on the *diverge* outcome. Thus, as soon as a predicate call is encountered in the course of computation, the pessimistic semantics effectively assumes that the computation will diverge. This means, for instance, that if there is a predicate call in a goal G to the left of the first disjunction in G, then G will diverge according to the pessimistic semantics.

We define the *pessimistic outcome* of a goal G, $outcome^{\frown}(G)$, as follows.

- If there is a θ' such that $(() : G \Rightarrow \theta')$ in the pessimistic operational semantics, then $outcome^{\frown}(G) = T$.
- If $(() : G \Rightarrow fail)$ in the pessimistic operational semantics, then $outcome^{\frown}(G) = F$.
- Otherwise (i.e. if $(() : G \Rightarrow diverge)$ in the pessimistic semantics), then $outcome^{\frown}(G) = U$.

Note that the program P is irrelevant to the pessimistic semantics, and that all computations in the pessimistic semantics are of bounded size because predicate calls are not expanded.

6.2.3 Results

Here we show the relationship between unfoldings and the pessimistic semantics.

Theorem 8
Let $\theta : \alpha \Rightarrow_P \rho$ in the conservative semantics. Then some P-unfolding α' of α is such that $\theta : \alpha' \Rightarrow_P \rho$ in the pessimistic semantics.

Proof
By induction on the structure of the conservative computation. Cases are on the bottommost rule, and all cases follow trivially from the induction hypothesis except the case in which the bottommost rule is a Pred rule. In this case, one additional predicate unfolding is necessary to obtain α' from the α' of the induction hypothesis. □

The converse of the above theorem is also the case:

Theorem 9
Let some P-unfolding α' of α be such that $\theta : \alpha' \Rightarrow_P \rho$ in the pessimistic semantics, where ρ is not *diverge*. Then $\theta : \alpha \Rightarrow_P \rho$ in the conservative semantics.

Proof
By induction on the number of 1-P-unfoldings needed to derive α' from α. The base case (0 unfoldings) is trivial. For the inductive case (n unfoldings), let α'' be a 1-P-unfolding of α such that α' is a P-unfolding of α'' after $n-1$ unfoldings. By the induction hypothesis, $\theta : \alpha'' \Rightarrow_P \rho$ in the conservative semantics.

It remains to prove that $\theta : \alpha \Rightarrow_P \rho$ as well. We do this by induction on the structure of the α'' computation. The cases are on the bottommost rule applied. In

https://doi.org/10.1017/S1471068402001540 Published online by Cambridge University Press

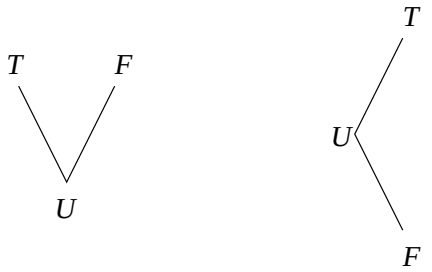

Fig. 14. Hasse diagrams of the 'definedness' ordering $<_k$ (left) and the 'truth' ordering $<_t$ (right) of truth values.

all cases, if α starts with a predicate call and α'' is derived from it by unfolding that call, then the computation of α can be derived from that of α'' by just adding an application of Pred. Otherwise, all cases follow directly from one or more applications of the induction hypothesis. □

This property of predicate unfoldings and the pessimistic semantics will be useful for the rest of the paper, because it allows us to abstract away from the unbounded computations of the non-pessimistic semantics and consider only the simpler, bounded computations of the pessimistic semantics.

6.2.4 The definedness ordering

The following definitions and theorem makes the connections between unfoldings and the pessimistic semantics more precise and concise by allowing us to give an expression corresponding to the outcome of a goal in terms of its pessimistic outcome.

We define the *definedness ordering* $<_k$ on truth values as the least partial order relation such that $U <_k T$ and $U <_k F$ (see figure 14). This is a standard ordering for these three truth values; see for example (Belnap, 1977). The expression $max_k(S)$, where S is a set of truth values, is undefined if $\{T, F\} \subseteq S$, and otherwise is defined as the unique truth value V such that $W \leqslant_k V$ for all $W \in S$.

Finally, we give the first raising lemma.

Lemma 6 (*Raising Lemma 1*)
For any goal G, $max_k(\{outcome^\frown(G') \mid G' \text{ is a } P\text{-unfolding of } G\})$ is well-defined and equal to $outcome_P(G)$.

Proof
Let the set S of truth values be $\{outcome^\frown(G') \mid G' \text{ is a } P\text{-unfolding of } G\}$. First assume that $outcome_P(G) = T$. By Theorem 8, $T \in S$; however, if $F \in S$, then by Theorem 9, $outcome_P(G) = F$, a contradiction. Therefore $max_k(S)$ is defined and must be T. Similarly, if $outcome_P(G) = F$ then $max_k(S)$ is defined and equal to F. If $outcome_P(G) = U$, then it cannot be the case that $T \in S$ or $F \in S$, because otherwise, by Theorem 9, $outcome_P(G) \neq U$. Therefore $S = \{U\}$, and $max_k(S)$ is defined and equal to U. □

https://doi.org/10.1017/S1471068402001540 Published online by Cambridge University Press

R1 $(B_1 \vee B_2)\&C \ \rhd \ (B_1\&C) \vee (B_2\&C)$
R2 $B\&(C_1 \vee C_2) \ \rhd \ (B\&C_1) \vee (B\&C_2)$, where B is negated-disjunction
R3 $\exists x(B_1 \vee B_2) \ \rhd \ (\exists x B_1) \vee (\exists x B_2)$
R4 $if\,[\vec{x}]((B_1 \vee B_2), C) \ \rhd \ if\,[\vec{x}](B_1, C) \vee (\neg(\exists \vec{x} B_1)\&if\,[\vec{x}](B_2, C))$
R5 $if\,[\vec{x}](B, C) \ \rhd \ \exists \vec{x}(B\&C)$, where B is negated-disjunction

Fig. 15. The rules of the term-rewriting relation $\rhd$.

6.3 Depth-first normal form

We now turn to the notion of Depth-First Normal Form (DFNF) in order to increase the level of abstraction of the semantics. The DFNF of a formula G is a formula which is operationally equivalent to G but whose outcome can be given a compositional characterization. In this section, we first define a term-rewriting system which rewrites formulas into formulas. We then prove that the system is locally confluent and terminating, and that it transforms every formula to a unique normal form (which we define as the DFNF). We then prove that each of the transformations of the rewriting system preserves pessimistic outcome. The conclusion is that each goal has a unique DFNF which has the same pessimistic outcome as the original goal.

The DFNF by itself does not directly raise the abstraction level of the semantics; however, it puts a goal in a form which can be given an abstract characterization, as we will see in the next section. The conclusion of this section is therefore referred to as the second raising lemma.

6.3.1 Term-rewriting system

The notion of DFNF, which is closely related to the notion of Disjunctive Normal Form (DNF), was introduced in (Andrews, 1997). Here we expand the notion to take account of *if* formulas.

The classes of *negated-disjunction* (N) and *outer-disjunction* (O) formulae are defined mutually recursively as follows. (Informally, an O formula has $\vee$s directly inside only $\neg$s or other $\vee$s.)

$$N \quad ::= \quad p(t_1, \ldots, t_n) \ \mid \ s = t \ \mid \ N\&N \ \mid \ \exists x N \ \mid \ \neg O$$

$$O \quad ::= \quad N \ \mid \ O \vee O$$

For example, $p \vee (\exists x(q(x))\&r)$ is an outer-disjunction formula but not a negated-disjunction formula; however, $\neg(p \vee (\exists x(q(x))\&r))$ is a negated-disjunction formula and thus automatically an outer-disjunction formula.

The notion of depth-first normal form is based on the five rules R1-R5 of the term-rewriting relation $\rhd$ (figure 15), which can be applied anywhere in a formula to rewrite it into another formula. Two of the rules refer to the notion of a negated-disjunction formula. We define a formula to be *in depth-first normal form* if none of R1-R5 can be applied anywhere in the formula.

For example, the formula $if\,[x](x = 0, p(x) \vee q(x))$ can be rewritten by one application of R5 to $\exists x(x = 0\&(p(x) \vee q(x)))$, and then by one application of R2 to

https://doi.org/10.1017/S1471068402001540 Published online by Cambridge University Press

$\exists x((x = 0 \& p(x)) \vee (x = 0 \& q(x)))$. It can then be rewritten by one application of R3 to $\exists x(x = 0 \& p(x)) \vee \exists x(x = 0 \& q(x))$. None of the rules R1-R5 apply to this latter formula, so it is in depth-first normal form.

6.3.2 Local confluence and termination

To prove that the rewriting process always leads to a single formula, we prove local confluence and termination of this rewriting system. The proofs are contained in Appendix A.

Theorem 10 (*Local Confluence of Rewriting System*)
If $A \rhd A_1$ and $A \rhd A_2$, then there is an A_3 such that $A_1 \rhd^* A_3$ and $A_2 \rhd^* A_3$.

Proof
See Appendix A. □

In preparation for proving termination of the rewriting system, we define the *depth* $d(G)$ of a formula G. It is the conventional notion of depth of a formula, expanded to take account of *if*.

$$d(s = t) = d(p(t_1, \ldots, t_n)) = 1$$
$$d(B \& C) = d(B \vee C) = max(d(B), d(C)) + 1$$
$$d(\neg B) = d(\exists x\ B) = d(B) + 1$$
$$d(if\,[x_1, \ldots, x_n](B, C)) = max(d(B), d(C)) + 1$$

We also define the *maximum potential depth* $pd(G)$ of a formula G. This is the depth that the formula might possibly attain after repeatedly being transformed with R1-R5.

$$pd(s = t) = pd(p(t_1, \ldots, t_n)) = 1$$
$$pd(B \& C) = pd(B \vee C) = max(pd(B), pd(C)) + 1$$
$$pd(\neg B) = pd(\exists x\ B) = pd(B) + 1$$
$$pd(if\,[x_1, \ldots, x_n](B, C)) = n + 2pd(B) + max(pd(B), pd(C))$$

Clearly $1 \leqslant d(G) \leqslant pd(G)$ for all formulas G.

The main lemma we need for termination is to prove that each application of R1-R5 maintains or decreases potential depth.

Lemma 7
If $G \rhd G'$, then $pd(G) \geqslant pd(G')$.

Proof
See Appendix A. □

Theorem 11 (*Termination of Rewriting System*)
For every G, there is an integer j such that for every sequence of formulas $G = G_0, G_1, G_2, \ldots, G_k$ such that $G_i \rhd G_{i+1}$ for all $1 \leqslant i < k$, we have that $k \leqslant j$.

Proof
Each of the rules R1-R5 increase the number of connectives in the formula, where *if* is counted as one connective. However, the Lemma shows that the depth of the resultant formula is bounded by $pd(G)$. Since the formula tree has a bounded depth and bounded branching factor, there is a limit j to how many nodes (connectives) it can contain. The rewriting process must stop at or before this limit. □

https://doi.org/10.1017/S1471068402001540 Published online by Cambridge University Press

6.3.3 Unique normal form and DFNF

Because of local confluence and termination, we are able to state the following corollary, which shows that every goal has a unique normal form under the rewriting rules R1-R5.

Corollary 12 (*Unique Normal Form*)
For every formula G not in normal form, there is a unique formula G'' in normal form, such that for all G' such that $G \rhd G'$, we have that $G' \rhd^* G''$.

Proof
See Appendix A. □

Because of this corollary, we are justified in making the following definition. The *depth-first normal form* of a formula, $dfnf(G)$, is the unique formula G' such that $G \rhd^* G'$ and there is no G'' such that $G' \rhd G''$. (For instance, the depth-first normal form of the example formula from Section 6.3.1, $if[x](x = 0, p(x) \vee q(x))$, is $\exists x(x = 0 \& p(x)) \vee \exists x(x = 0 \& q(x))$.) Clearly, despite the complexity of the proofs of confluence and termination, we can obtain $dfnf(G)$ in a straightforward fashion, by simply applying one of the rules R1–R5 to any suitable redex (say, the outermost one) until there are no more redexes.

We also note that $dfnf(G)$ is outer-disjunction, a fact which will be important soon.

Theorem 13
For all G, $dfnf(G)$ is outer-disjunction.

Proof
If $dfnf(G)$ were not outer-disjunction, it would have some disjunction as an immediate subformula of a conjunction, existential formula, or *if* formula. In all these cases, one of rules R1-R5 would apply. □

6.3.4 Outcome preservation

We now show that the depth-first normal form formation does not change the outcome of a goal under the pessimistic semantics.

Theorem 14 (*General Result Preservation of* $dfnf$)
If α' is α with some formulas transformed by applications of rules R1-R5, then $\theta : \alpha \Rightarrow \rho$ in the pessimistic semantics iff $\theta : \alpha' \Rightarrow \rho$ in the pessimistic semantics.

Proof
See Appendix A. □

We can now give the second raising lemma, by showing the specific result that we wanted to obtain.

Lemma 8 (*Raising Lemma 2*)
$outcome^\frown(G) = outcome^\frown(dfnf(G))$.

https://doi.org/10.1017/S1471068402001540 Published online by Cambridge University Press

Proof
By Theorem 14, with respect to the pessimistic semantics, $(() : G \Rightarrow \rho)$ iff $(() : dfnf(G) \Rightarrow \rho)$. Therefore, with respect to the pessimistic semantics, G succeeds (fails, diverges) exactly when $dfnf(G)$ succeeds (fails, diverges). □

Note that we have come one step closer to an abstract characterization of outcome, by reducing the problem of characterizing outcome of a general goal with respect to a general program to the problem of characterizing the outcome of an outer-disjunction goal with respect to the pessimistic semantics.

6.4 The valuation function

Finally we come to the definition of the valuation v, which characterizes the outcomes of outer-disjunction goals (e.g. goals in DFNF) with respect to the pessimistic semantics. This valuation is a compositional function from formulae to truth values, like valuations in standard theories of truth (Kripke, 1975; Fitting, 1985), and interprets the binary connectives in a manner consistent with the left-to-right search algorithm of Prolog. v is based on the similar valuation in Andrews (1997). The valuation in that paper is on a domain of four truth values, but we need only three truth values here because we do not consider the *flounder* outcome.

In this section, we first define the alethic ordering $<_t$ on truth values, and then the valuation function v which uses it. Then we show that the valuation of a goal in outer-disjunction form is the same as its pessimistic outcome.

6.4.1 Alethic ordering and valuation function

We define the *alethic ordering* $<_t$ on truth values as the least partial order relation such that $F <_t U$ and $U <_t T$. (See figure 14. This is another standard ordering on these truth values; see for instance (Belnap, 1977).) The expression $max_t(S)$, where S is a set of truth values, is defined as the unique truth value V such that $W \leqslant_t V$ for all $W \in S$. The alethic ordering is used in the valuation function to express the meaning of $\exists x\ G$ in terms of the meaning of the instances of G.

v, a *valuation function* mapping ground, outer-disjunction (O) formulae to truth values in $\{T, U, F\}$, is defined as follows.

- $v(t = t) = T$;
- $v(s = t) = F$, where s is not identical to t;
- $v(p(t_1, \ldots, t_n)) = U$;
- $v(B\&C) = \begin{cases} v(C) & \text{if } v(B) = T, \\ v(B) & \text{otherwise;} \end{cases}$
- $v(B \vee C) = \begin{cases} v(C) & \text{if } v(B) = F, \\ v(B) & \text{otherwise;} \end{cases}$
- $v(\exists x\ B) = max_t(\{v(B[x := t]) \mid t \text{ ground}\})$;
- $v(\neg B) = \begin{cases} F & \text{if } v(B) = T, \\ U & \text{if } v(B) = U, \\ T & \text{if } v(B) = F. \end{cases}$

https://doi.org/10.1017/S1471068402001540 Published online by Cambridge University Press

For instance, recall from section 3 that *true* is the formula $(0 = 0)$ and *false* is the formula $(0 = 1)$. By the definition of v, we have that $v(true) = v(0 = 0) = T$, and $v(false) = v(0 = 1) = F$, as expected. We also have that $v(\neg true) = F$, $v(true \& false) = F$, and $v(false \vee true) = T$. We have that $v(false \vee p(0))$ and $v(true \& p(0))$ are both U, but $v(false \& p(0)) = F$ and $v(true \vee p(0)) = T$, consistent with how the pessimistic semantics would execute the formulas as queries.

In fact, while $v(0 = 0) = T$, we have that $v(s = 0) = F$ for any term s other than 0. Therefore the set $\{v(t = 0) \mid t \text{ ground}\}$ is the set $\{v(0 = 0)\} \cup \{v(t = 0) \mid t \text{ ground and } t \neq 0\}$, i.e. $\{T\} \cup \{F\}$, or $\{T, F\}$. As a consequence, $v(\exists x(x = 0)) = \{v(t = 0) \mid t \text{ ground}\} = T$, since T is the maximally true truth value in the set $\{T, F\}$.

6.4.2 *Equivalence of valuation and pessimistic outcome*

The valuation function v characterizes precisely the behaviour of outer-disjunction formulae with respect to the pessimistic semantics. In preparation for this result, we state a proposition which is a weaker form of the converse of the witness properties, applying only to N formulas.

Proposition 15
Let α be a sequence of negated-disjunction (N) formulas, such that $\theta : \alpha \Rightarrow \rho$ in the pessimistic semantics. Let V be a subset of the free variables of α. Then:
(1) If for some substitution ξ grounding V consistent with θ, $(\theta : \alpha\xi \Rightarrow \theta')$ in the pessimistic semantics, then ρ is some θ''.
(2) If for all substitutions ξ grounding V consistent with θ, $(\theta : \alpha\xi \Rightarrow fail)$ in the pessimistic semantics, then ρ is *fail*.

The fragment of the pessimistic semantics dealing with negated-disjunction formulas is identical to the fragment of the semantics of Andrews (1997) dealing with negated-disjunction formulas with respect to the empty program. The proof of this Proposition is thus a simple adaptation of the proof of Lemma 4.5 from Andrews (1997). Intuitively, the Proposition applies only to N formulas because instantiating an N formula will either cause it to fail or will not change the outcome its computation. In contrast, for example, $B \vee C$ may diverge because B diverges, but $B\theta \vee C\theta$ may succeed because $B\theta$ fails and $C\theta$ succeeds. We cannot draw any conclusions about the behaviour of $B \vee C$ from the behaviour of its instances.

We are now in a position to state the third raising lemma, continuing our process of abstraction. Note that this lemma relates an operational notion (pessimistic outcome) to an entirely abstract one (valuation).

Theorem 16 (*Raising Lemma 3*)
If G is ground and outer-disjunction, then $v(G) = outcome^{\frown}(G)$.

Proof
By induction on the structure of G. Cases are on the outermost connective. We note only the three subcases of the case in which $G = \exists x\ B$.

If $outcome^{\frown}(G) = T$, there must be some θ' such that $(() : \exists x\ B \Rightarrow \theta')$ in the pessimistic semantics. In this case, we also have that $(() : B[x := x'] \Rightarrow \theta')$,

https://doi.org/10.1017/S1471068402001540 Published online by Cambridge University Press

and by the witness properties, there must be some ground t and θ'' such that $(() : B[x := x'][x' := t] \Rightarrow \theta'')$. Thus for some t, $outcome^\frown(B[x := t]) = T$. By the induction hypothesis, $v(B[x := t]) = T$; and by the definition of max_t, $v(G) = T$.

If $outcome^\frown(G) = F$, then $(() : \exists x\ B \Rightarrow fail)$ in the pessimistic semantics. In this case, we also have that $(() : B[x := x'] \Rightarrow fail)$, and by the witness properties, for all ground t, $(() : B[x := x'][x' := t] \Rightarrow fail)$. Thus for all t, $outcome^\frown(B[x := t]) = F$. By the induction hypothesis, $v(B[x := t]) = F$; and by the definition of max_t, $v(G) = F$.

Otherwise, $outcome^\frown(G) = U$. By Prop. 15, there cannot be any t such that $outcome^\frown(B[x := t]) = T$, because otherwise $outcome^\frown(G)$ would be T; and again by Prop. 15, it cannot be the case that for all t, $outcome^\frown(B[x := t]) = F$, because otherwise $outcome^\frown(G)$ would be F. Thus for some t, $outcome^\frown(B[x := t]) = U$, so the set $\{v(B[x := t]) \mid t \text{ is ground }\}$ of truth values is either $\{U\}$ or $\{U, F\}$. Thus by the definition of max_t, $v(G) = U$. □

6.5 The denotation of a program

In this section, we give the final raising lemma which summarizes the previous ones. This lemma gives an expression which is an abstract characterization of the outcome of a goal; we therefore give a definition of the denotation of a program which uses this expression.

Lemma 9 (*Raising Lemma 4*)
For any ground goal G,
$outcome_P(G) = max_k(\{v(dfnf(G')) \mid G' \text{ is a } P\text{-unfolding of } G\})$.

Proof
By Raising Lemma 1, $outcome_P(G) = max_k(\{outcome^\frown(G') \mid G' \text{ is a } P\text{-unfolding of } G\})$. By Raising Lemma 2, $outcome^\frown(G') = outcome^\frown(dfnf(G'))$ for any G'. But by Theorem 13, $dfnf(G')$ is in outer-disjunction form for any G'; therefore by Raising Lemma 3, $outcome^\frown(dfnf(G')) = v(dfnf(G'))$. Putting this all together, we conclude that $outcome_P(G) = max_k(\{v(dfnf(G')) \mid G' \text{ is a } P\text{-unfolding of } G\})$. □

We therefore make the following definition. The *denotation* v_P of a program P is a valuation function defined by:

$$v_P(G) = max_k(\{v(dfnf(G')) \mid G' \text{ is an unfolding of } G\}).$$

We have the following trivial theorem.

Theorem 17 (*Denotation*)
For any ground goal G, $outcome_P(G) = v_P(G)$.

Proof
By Raising Lemma 4 and the definition of v_P. □

Note that the restriction to ground goals does not decrease the generality of the denotation result, since a goal G with free variables $\vec{x}$ has the same outcome as the goal $\exists \vec{x} G$.

https://doi.org/10.1017/S1471068402001540 Published online by Cambridge University Press

6.6 Example

As a further example of how the denotation of a program defines the correct truth value of a goal, we derive the value obtained by applying the denotation of a program to a goal.

Let the program P be the second 'delete' program from Section 3:

$$
\begin{aligned}
&d(x, y, z) \ :- \\
&\quad (y = [\,] \ \& \ z = [\,]) \\
&\vee \ if\,[ys](y = [x|ys], d(x, ys, z)) \\
&\vee \ (\neg \exists ys(y = [x|ys]) \ \& \\
&\qquad \exists y' \exists ys \exists zs(y = [y'|ys] \& z = [y'|zs] \& d(x, ys, zs)))
\end{aligned}
$$

Consider the goal $G = \exists z \ d(a, [\,], z)$. This goal asks whether there is a z which is obtained by deleting a everywhere from the empty list $[\,]$. It has the outcome T in the conservative semantics, since there does exist a z, namely the empty list $[\,]$ itself, which is obtained that way.

We take as our objective to derive the value of $v_P(G)$. From the definition of v_P, we have that $v_P(G) = max_k(\{v(dfnf(G')) \mid G' \text{ is an unfolding of } G\})$. Let S be the set $\{v(dfnf(G')) \mid G' \text{ is an unfolding of } G\}$; then $v_P(G) = max_k(S)$. As discussed in the proof of Raising Lemma 1, if $\{U, T\} \subseteq S$, then $F \notin S$; so if we can find one unfolding of G whose DFNF valuation is U and another whose DFNF valuation is T, then we know $S = \{U, T\}$.

In fact, we can find such unfoldings. The subsequent sections show that G itself is such that $v(dfnf(G)) = U$, and that the first unfolding G_1 of G is such that $v(dfnf(G_1)) = T$. Hence $v_P(G) = max_k(S) = max_k(\{U, T\}) = T$.

First, we show that $v(dfnf(G)) = U$. Then, we find the expression for G_1 and for $dfnf(G_1)$. Finally, we show that $v(dfnf(G_1)) = T$.

6.6.1 $v(dfnf(G)) = U$

G is $\exists z \ d(a, [\,], z)$. This formula contains no disjunctions or *if*s, so none of the DFNF rewriting rules applies to it; hence $dfnf(G)$ is G itself. By the definition of v, $v(dfnf(G)) = v(G) = v(\exists z \ d(a, [\,], z))$, which is the expression $max_t(\{d(a, [\,], z) \mid t \text{ is a ground term}\})$; that is, the maximally true truth value amongst the valuations of all the formulas of the form $d(a, [\,], t)$, where t is a ground term.

However, by the definition of v, the valuation of any predicate call formula is U (since v correctly characterizes the pessimistic semantics). Hence $max_t(\{d(a, [\,], z) \mid t \text{ is a ground term}\}) = max_t(\{U\}) = U$. Since this was the expression for $v(dfnf(G))$, we have that $v(dfnf(G)) = U$.

6.6.2 First unfolding and its DFNF

G is $\exists z \ d(a, [\,], z)$. The first unfolding of G, G_1, can be obtained by replacing the predicate call within it by the body of the definition of the predicate d, replacing formal by actual parameters. Therefore:

https://doi.org/10.1017/S1471068402001540 Published online by Cambridge University Press

$$
\begin{aligned}
G_1 = \exists z(& \\
& ([\,] = [\,] \;\&\; z = [\,]) \\
\vee\ & if\,[ys]([\,] = [a|ys], d(a, ys, z)) \\
\vee\ & (\neg\exists ys([\,] = [a|ys]) \;\& \\
& \quad \exists y'\exists ys\exists zs([\,] = [y'|ys] \& z = [y'|zs] \& d(a, ys, zs)))
\end{aligned}
$$

We abbreviate this formula as $\exists z(G'_1 \vee G'_2 \vee (G'_3 \& G'_4))$.

The DFNF rewriting rule R3 can be applied twice to G_1, to yield the formula $(\exists z(G'_1) \vee \exists z(G'_2) \vee \exists z(G'_3 \& G'_4))$. G'_2 is an *if* formula, $if\,[ys]([\,] = [a|ys], d(a, ys, z))$, whose first subformula $([\,] = [a|ys])$ is a negated-disjunction formula; hence, the DFNF rewriting rule R5 can be applied to it, yielding the subformula $G'_5 = \exists ys([\,] = [a|ys] \& d(a, ys, z))$. At this point, no more of the DFNF rewriting rules can be applied to the formula, so it is in depth-first normal form.

Hence, $dfnf(G_1) = (\exists z(G'_1) \vee \exists z(G'_5) \vee \exists z(G'_3 \& G'_4))$, where:

- $G'_1 = ([\,] = [\,] \;\&\; z = [\,])$;
- $G'_5 = \exists ys([\,] = [a|ys] \& d(a, ys, z))$;
- $G'_3 = \neg\exists ys([\,] = [a|ys])$; and
- $G'_4 = \exists y'\exists ys\exists zs([\,] = [y'|ys] \& z = [y'|zs] \& d(a, ys, zs)))$.

6.6.3 $v(dfnf(G_1)) = T$

$v(dfnf(G_1)) = v(\exists z(G'_1) \vee \exists z(G'_5) \vee \exists z(G'_3 \& G'_4))$. We can therefore obtain the value of $v(dfnf(G_1))$ by first obtaining the values of its disjuncts. By the definition of v, we have that $v(\exists z(G'_1))$ is the value of the expression $max_t(\{v([\,] = [\,] \;\&\; t = [\,]) \mid t$ is a ground term$\})$. The value of $v([\,] = [\,] \;\&\; t = [\,])$ is T if the values of both $v([\,] = [\,])$ and $v(t = [\,])$ are T, and it is F otherwise. However, $v([\,] = [\,])$ is always T; and $v(t = [\,])$ is T if t is $[\,]$, and otherwise is F.

The set $\{v([\,] = [\,] \;\&\; t = [\,]) \mid t$ is a ground term$\}$ therefore consists of the two truth values $\{T, F\}$. The maximally true member of this set is T; hence, $v(\exists z(G'_1)) = max_t(\{T, F\}) = T$. Now, $dfnf(G_1)$ is of the form $(\exists z(G'_1) \vee H)$; so $v(dfnf(G_1)) = v(\exists z(G'_1) \vee H)$. By the definition of v, and because $v(\exists z(G'_1)) = T$, $v(\exists z(G'_1) \vee H) = T$; hence $v(dfnf(G_1)) = T$.

We conclude the example by reiterating the value of $v_P(G)$. Because $v(dfnf(G)) = U$ and $v(dfnf(G_1)) = T$, the set $\{v(dfnf(G')) \mid G'$ is an unfolding of $G\}$ is just $\{U, T\}$. Therefore:

$$
\begin{aligned}
v_P(G) &= max_k(\{v(dfnf(G')) \mid G' \text{ is an unfolding of } G\}) \\
&= max_k(\{U, T\}) \\
&= T
\end{aligned}
$$

This result accords with the fact that the original goal G did succeed under the conservative semantics.

6.7 Discussion

Note that the abstract semantics is based on six basic, relatively simple notions: the notion of truth value, the two orderings of the truth values, the notion of predicate unfolding, the notion of depth-first normal form, and the logical valuation. The

https://doi.org/10.1017/S1471068402001540 Published online by Cambridge University Press

notion of depth-first normal form, in turn, is based on a rewriting system of five rules. The predicate unfolding and normal form constructions essentially do local meaning-preserving transformations to prepare the goal in question for characterization, and the valuation actually performs that characterization.

In some sense, the crucial element of the abstract semantics, the element which allows it not to reify such notions as substitutions and unification, is the $\exists$ clause of the definition of v. Rather than view a variable operationally, as a placeholder in a term which at some future point can be replaced by another term, the $\exists$ clause allows us to view it as a true variable ranging over a fixed domain of discourse. This, in turn, has been enabled by the witness properties of the conservative semantics. Without the witness properties, we would not have been able to prove that the value of $v(\exists xG)$ could be derived directly from the consideration of the values of $v(G[x := t])$, for any ground t. Hence, the witness properties are useful not only from the point of view of intuitively justifying the behaviour of a logic programming system, but also on theoretical grounds.

7 Conclusions

The main contributions of this paper are as follows.

- We have defined an extension of Prolog with hard cut and negation as failure in which programs can provably be put in a convenient 'completed' form. This completion has been achieved by using a variable-binding choice construct, *if*.
- We have identified the witness properties as important properties intermediate between the strict logicalness of pure Horn clause programming and the unrestricted freedom of typical Prolog implementations.
- We have defined restrictions on the computation of extended programs which allow the resulting system to achieve the witness properties. We have referred to the resulting notion of cut as *firm cut*, insofar as it is intermediate between hard and soft cut.
- We have defined an abstract semantics for the restricted system (taking depth-first termination, rather than universal termination, as its observable), which uses the witness properties in order to avoid reifying the concepts of unification and substitution.

Long investigations by the author have not resulted in any semantics for Prolog which allow the full range of behaviour of hard cut while rising in any meaningful way above the level of an operational semantics. We do not believe at this point that such a semantics is possible. We believe that the system with firm cut, as defined in this paper, is the best compromise yet found between the power of the hard cut and the logical rigour of the soft cut. We believe that the behaviours of hard cut excluded by firm cut are unlikely to be missed by Prolog programmers, and that the witness properties achieved by firm cut capture the core of programmers' desiderata about a logic programming system, even though they are not in complete harmony with logic. However, these are merely beliefs. We invite readers to decide whether they agree or disagree based on their experience.

The more theoretically substantiated conclusions we draw from this work are as follows.

- The widely-held view that features such as cut and negation as failure entirely destroy the declarative interpretation of logic programming systems seems to be too strong. While firm cut cannot be interpreted as a logical construct, the abstract semantics developed here suggest that a system with firm cut is more declarative than one with hard cut, while still retaining behaviour of hard cut which is useful in practice.
- If a logic programming language does not achieve soundness with respect to traditional logical interpretations, it might still be possible for it to achieve the witness properties. Given that practical, widely-used languages often implement pragmatic features which depart from well-defined semantics, insisting on the witness properties might be an acceptable alternative to insisting on soundness with respect to first order logic.
- The Prolog syntax and clause-based operational semantics is difficult to work with in an abstract setting when taking cut into consideration. We have found it easier to study semantic issues with programs in 'completed' program form, and the structured operational semantics, described in this paper. The syntax of the Mercury language (Somogyi *et al*., 1996) is already closer to the completed form described here, since it uses an efficient 'if' formula (though the 'if' of Mercury corresponds to soft cut, not firm cut).

There are several interesting open questions suggested by this research.

- Are other 'non-logical' features of Prolog able to be given a form which allows the witness properties to be preserved? Obviously there is no hope for the `var` and `nonvar` predicates, which check the instantiation of their arguments, but what about `assert`, `retract`, `bagof`, and so on?
- What is the largest subset of the liberal general semantics with the witness properties? That is, can we define an operational semantics analogous to the conservative semantics, but with respect to which all goals with the witness properties do not flounder? The answer to this question may lie with different strategies for coping with negation.
- Can a mode *inference* system be devised which ensures non-floundering of goals? That is, can we automate the process of defining modes for a program that will guarantee that no goal consistent with the inferred modes of the program's predicates will flounder?

We have implemented the ideas contained in this paper in an experimental proof assistant program called SVP (Spreadsheet Verifier for Prolog), whose user interface has been described in Andrews (1998). SVP transforms a Prolog program with cuts into completed form, and then assists the user in proving theorems in an assertion language similar to those defined in Andrews (1991) and Stärk (1998). We hope to report on this work in the future.

Acknowledgements

Thanks to Verónica Dahl and the Logic and Functional Programming Laboratory at Simon Fraser University for the use of their facilities in preparing this material. Thanks also to Kai Salomaa for clarification on terminology. Michel Billaud, Robert Stärk, and Torkel Franzen helped with the original conference version of this paper, and the anonymous journal referees contributed valuable comments and corrections. This research is supported by an NSERC (Natural Sciences and Engineering Research Council of Canada) Individual Research Grant.

A Proofs of results

A.1 Completion algorithm properties

Lemma 1

Let α be a goal stack. Let α' be α with any number of occurrences of a sequence B, C in a goal stack or clause body replaced by $B\&C$, where B and C are formulas. Then $(\theta : \alpha \Rightarrow_P \rho)$ in the liberal general semantics iff $(\theta : \alpha' \Rightarrow_P \rho)$ in the liberal general semantics.

Proof

By induction on the number of replacements of B, C by $B\&C$. The base case (0 replacements) is trivial. For the inductive case, it suffices to demonstrate the case where α' is derived from α by one replacement of B, C by $B\&C$. This in turn we prove by induction on the structure of the computation of α. If α begins with B, C and α' begins with $B\&C$, then the computation of α' can be derived from that of α with one Conj step. Otherwise, either the first formulas in the two goal stacks are identical, or they have the same top-level connective; in either case, regardless of the bottommost rule applied, the result follows straightforwardly from the induction hypothesis. □

Lemma 2

Let P' be P with some sequence B, C in a clause body replaced by $B\&C$. Then $\theta : \alpha \Rightarrow_P \rho$ in the liberal general semantics iff $\theta : \alpha \Rightarrow_{P'} \rho$ in the liberal general semantics.

Proof

By the lemma, we can add new rules to the operational semantics as follows:

$$(1)\frac{\theta : \alpha' \Rightarrow_P \rho}{\theta : \alpha \Rightarrow_P \rho} \qquad (2)\frac{\theta : \alpha \Rightarrow_P \rho}{\theta : \alpha' \Rightarrow_P \rho}$$

where α' is α with any number of occurrences of a sequence B, C in a goal stack or clause body replaced by $B\&C$. Moreover, by the lemma, we can essentially insert applications of these rules anywhere in a computation and derive a computation of the premise from the computation of the conclusion.

Therefore the $(\rightarrow)$ direction of the theorem can be proven as follows. Given the computation of $\theta : \alpha \Rightarrow_P \rho$, insert an application of (1) above each Pred rule involving the clause transformed in P', obtaining the computation of the premise from the lemma. The transformed proof will have sections of the form:

$$\dfrac{\dfrac{\theta : p(\vec{t})using(\gamma'), \alpha \ \Rightarrow_P \ \rho}{\theta : p(\vec{t})using(\gamma), \alpha \ \Rightarrow_P \ \rho}}{\theta : p(\vec{t}), \alpha \ \Rightarrow_P \ \rho}$$

To obtain the computation of $\theta : \alpha \ \Rightarrow_{P'} \ \rho$, replace each such section by

$$\dfrac{\theta : p(\vec{t})using(\gamma'), \alpha \ \Rightarrow_{P'} \ \rho}{\theta : p(\vec{t}), \alpha \ \Rightarrow_{P'} \ \rho}$$

and replace P by P' everywhere else. The other direction of the theorem can be proven by inverting this operation. □

Theorem 3

The completion algorithm preserves result according to the liberal general operational semantics. That is, if P' is the completion of P, then $\theta : \alpha \ \Rightarrow_P \ \rho$ in the liberal general semantics iff $\theta : \alpha \ \Rightarrow_{P'} \ \rho$ in the liberal general semantics.

Proof

We prove the theorem by proving that each of the transformations preserves result. In what follows, we will refer to the original program as P and the program after the single transformation in question as P'.

Step 2.2

Clearly the two computations are equivalent up to a renaming of some of the variables involved in the computations.

Step 2.3

It suffices to show that any application of any of the four *using* rules with P correspond to parts of computations with P'. Consider an application of the Using/nocut/succ rule with P, where the formula being considered is an application of predicate p. The bottommost portion of the computation is:

$$\dfrac{\dfrac{\dfrac{\theta\xi : \eta\xi, \alpha\xi \ \Rightarrow \ \theta'}{\vdots}}{\theta : (s_1 = t_1), \ldots, (s_k = t_k), \ldots, (s_n = x_n), \eta, \alpha \ \Rightarrow \ \theta'}}{\theta : p(s_1, \ldots, s_n)using(p(t_1, \ldots, t_k, \ldots, x_n) \ :- \ \eta), \gamma), \alpha \ \Rightarrow \ \theta'}$$

where ξ is the substitution resulting from the unifications. With P', the bottommost portion of the computation is the following:

$$\dfrac{\dfrac{\dfrac{\theta\xi' : \eta\xi', \alpha\xi' \ \Rightarrow \ \theta'}{\vdots}}{\theta : (s_1 = t_1), \ldots, (s_k = x_k), \ldots, (s_n = x_n), (x_k = t_k), \eta, \alpha \ \Rightarrow \ \theta'}}{\theta : p(s_1, \ldots, s_n)using(p(t_1, \ldots, x_k, \ldots, x_n) \ :- \ (x_k = t_k), \eta), \gamma), \alpha \ \Rightarrow \ \theta'}$$

where ξ' is the substitution resulting from the unifications. However, by the properties of unification, we can rearrange the equality formulas in the judgement second from the bottom to read: $(s_1 = t_1), \ldots, (x_k = s_k), (x_k = t_k), \ldots, (s_n = x_n)$. This sequence makes it clear that the result substitution ξ' is identical to ξ. The cases of the other Using rules are proven similarly.

https://doi.org/10.1017/S1471068402001540 Published online by Cambridge University Press

Step 3
See Lemma 2 just before this theorem.

Step 4
Let α be a goal stack, and let α' be α with the formula *true* inserted anywhere in a sequence of goal stack elements or body elements. Then $(\theta : \alpha \Rightarrow_P \rho)$ iff $(\theta : \alpha' \Rightarrow_P \rho)$, by a simple structural induction. We can then follow the same line of reasoning as in Lemma 2 to conclude that inserting *true* anywhere in a clause body preserves result.

Step 5
When a clause with two consecutive cuts appears, instances of the Body/cut/succ rule will arise in which η_1 is empty; that is, a portion of some computations will be of the form

$$\frac{\overline{\theta : \epsilon \Rightarrow \theta} \qquad \theta : body(\eta_2), \alpha \Rightarrow \rho}{\theta : body(!, \eta_2), \alpha \Rightarrow \rho}$$

where the Success rule has been used at the left-hand premise. When the program is transformed to remove the second cut, this portion of the computation will be replaced by the single judgement $(\theta : body(\eta_2), \alpha \Rightarrow \rho)$.

Step 6
See Step 4 above.

Step 7
See Step 4 above.

Step 8
The original computation may have applications of the Body/cut/succ rules of the following form:

$$\frac{\theta : \eta_1 \Rightarrow \theta' \qquad \theta' : body(\eta_2)\theta', \alpha\theta' \Rightarrow \rho}{\theta : body(\eta_1, !, \eta_2), \alpha \Rightarrow \rho}$$

This part of the computation is replaced in the new computation by the following sequence:

$$\cfrac{\cfrac{\cfrac{\cfrac{\theta[\vec{y} := \vec{y}'] : \eta_1 \Rightarrow \theta'}{\vdots}}{\theta : \vec{y} = \vec{y}', \eta_1[\vec{y} := \vec{y}'] \Rightarrow \theta'} \qquad \theta' : body(\eta_2)\theta', \alpha\theta' \Rightarrow \rho}{\theta : q(\vec{y}) using (q(\vec{y}') \text{ :- } \eta_1[\vec{y} := \vec{y}'], !, \eta_2[\vec{y} := \vec{y}']), \alpha \Rightarrow \rho}}{\theta : q(\vec{y}), \alpha \Rightarrow \rho}}{\theta : body(q(\vec{y})), \alpha \Rightarrow \rho}$$

Note that the substitution $[\vec{y} := \vec{y}']$ has the effect of restoring η_1, η_2 to their original naming. We do not show $[\vec{y} := \vec{y}']$ elsewhere since the computations are equivalent up to renaming.

The original computation may also have applications of Body/cut/fail, which are transformed similarly.

https://doi.org/10.1017/S1471068402001540 Published online by Cambridge University Press

Step 9.2

In computations with P, variables in the clause are renamed apart at the appropriate applications of the Pred rule. In computations with P', the $\vec{y}$ variables are bound and therefore not renamed apart. However, they become renamed apart in Exists rule applications above the application of the Using or Body rule in which they become part of the goal stack.

Step 9.3

The original computation may have portions ending with applications of the Using/cut/succ rule, of the form

$$\dfrac{\dfrac{\dfrac{\theta\xi : F\xi \;\Rightarrow\; \theta'}{\vdots}}{\theta : \vec{t} = \vec{x}, F \;\Rightarrow\; \theta'} \qquad \dfrac{\theta' : G\theta', \alpha\theta' \;\Rightarrow\; \rho}{\theta' : body(G)\theta', \alpha\theta' \;\Rightarrow\; \rho}}{\theta : p(\vec{t}) using (p(\vec{x}) \;:\!-\; F, !, G), \alpha \;\Rightarrow\; \rho}$$

where ξ is the substitution resulting from the unification of $\vec{t}$ with $\vec{x}$. (Without loss of generality, to avoid confusion, we assume that the free variables of the clause are different from those in $\vec{t}$ and α, and do not require renaming apart.) The computation with respect to P' will have this portion of the computation replaced by the following:

$$\dfrac{\dfrac{\dfrac{\dfrac{\theta\xi : F\xi \;\Rightarrow\; \theta' \qquad \theta' : G\xi\theta', \alpha\xi \;\Rightarrow\; \rho}{\theta\xi : if\,[\vec{y}](F, G)\xi, \alpha\xi \;\Rightarrow\; \rho}}{\vdots}}{\theta : \vec{t} = \vec{x}, if\,[\vec{y}](F, G), \alpha \;\Rightarrow\; \rho}}{\theta : p(\vec{t}) using (p(\vec{x}) \;:\!-\; if\,[\vec{y}](F, G)), \alpha \;\Rightarrow\; \rho}$$

However, because the $\vec{x}$ are distinct and different from the variables in α, $\alpha\xi$ is just α; and because θ' has arisen from $\theta\xi$, $\xi\theta' = \theta'$. Thus the two judgements at the top of this portion of this computation are the same as the two at the top of the portion of the computation with respect to P.

The original computation may also have applications of Using/cut/fail, which are transformed similarly.

Step 10.2

The original computation may have portions ending in applications of the Using/nocut/succ rule, of the form

$$\dfrac{\dfrac{\dfrac{\theta\xi : G\xi, \alpha\xi \;\Rightarrow\; \rho}{\vdots}}{\theta : \vec{t} = \vec{x}, G, \alpha \;\Rightarrow\; \rho}}{\theta : p(\vec{t}) using (p(\vec{x}) \;:\!-\; G; p(\vec{x}) \;:\!-\; H), \alpha \;\Rightarrow\; \rho}$$

where ξ is the substitution resulting from the unification of $\vec{t}$ with $\vec{x}$. (Again, without loss of generality we assume the free variables of the clauses are different from those of the conclusion.) The computation with respect to P' will have this portion of the computation replaced by the following:

https://doi.org/10.1017/S1471068402001540 Published online by Cambridge University Press

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\theta\xi : G\xi, \alpha\xi \;\Rightarrow\; \rho}{\vdots}}{\theta\xi : \exists\vec{y}(G)\xi, \alpha\xi \;\Rightarrow\; \rho}}{\theta\xi : \exists\vec{y}(G)\xi \vee H\xi, \alpha\xi \;\Rightarrow\; \rho}}{\vdots}}{\theta : \vec{t} = \vec{x}, \exists\vec{y}(G) \vee H, \alpha \;\Rightarrow\; \rho}}{\theta : p(\vec{t})using(p(\vec{x}) \;:-\; \exists\vec{y}(G) \vee H), \alpha \;\Rightarrow\; \rho}$$

The topmost judgements of these portions of the proof are the same.

The original computation may also have applications of Using/nocut/fail, which are transformed similarly.

Step 10.3

The original computation may have portions ending in applications of the Using/cut/succ rule, of the form

$$\dfrac{\dfrac{\dfrac{\theta\xi : F\xi \;\Rightarrow\; \theta'}{\vdots}}{\theta : \vec{t} = \vec{x}, F \;\Rightarrow\; \theta'} \qquad \dfrac{\theta' : G\theta', \alpha\theta' \;\Rightarrow\; \theta'}{\theta' : body(G)\theta', \alpha\theta' \;\Rightarrow\; \theta'}}{\theta : p(\vec{t})using(p(\vec{x}) \;:-\; F, !, G; p(\vec{x}) \;:-\; H), \alpha \;\Rightarrow\; \rho}$$

where ξ is the substitution resulting from the unification of $\vec{t}$ and $\vec{x}$. (Throughout, we assume the variables of the clauses are distinct from the other variables in the computation.) The computation with P' will have this portion replaced by the following:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\theta\xi : F\xi \;\Rightarrow\; \theta' \quad \theta' : G\xi\theta', \alpha\xi\theta' \;\Rightarrow\; \rho}{\theta\xi : if[\vec{y}](F, G)\xi, \alpha\xi \;\Rightarrow\; \rho}}{\theta\xi : if[\vec{y}](F, G)\xi \vee ((\neg\exists\vec{y}(F)\&H)\xi, \alpha\xi \;\Rightarrow\; \rho}}{\vdots}}{\theta : \vec{t} = \vec{x}, if[\vec{y}](F, G) \vee ((\neg\exists\vec{y}(F)\&H), \alpha \;\Rightarrow\; \rho}}{\theta : p(\vec{t})using(p(\vec{x}) \;:-\; if[\vec{y}](F, G) \vee ((\neg\exists\vec{y}(F)\&H)), \alpha \;\Rightarrow\; \rho}$$

As in Step 9.3, because of the way the substitutions were formed, the topmost judgements in this portion of the P' computation are the same as those at the top of the portion of the P computation.

The original computation may also have portions ending in applications of the Using/cut/fail rule, of the form

$$\dfrac{\dfrac{\dfrac{\theta\xi : F\xi \;\Rightarrow\; fail}{\vdots}}{\theta : \vec{t} = \vec{x}, F \;\Rightarrow\; fail} \qquad \dfrac{\dfrac{\theta\xi : H\xi, \alpha\xi \;\Rightarrow\; \rho}{\vdots}}{\theta : \vec{t} = \vec{x}, H, \alpha \;\Rightarrow\; \rho}}{\theta : p(\vec{t})using(p(\vec{x}) \;:-\; F, !, G; p(\vec{x}) \;:-\; H), \alpha \;\Rightarrow\; \rho}$$

where ξ is the substitution renaming the variables of the first clause apart, and ξ' is

https://doi.org/10.1017/S1471068402001540 Published online by Cambridge University Press

the substitution resulting from the unification of $\vec{t}$ and $\vec{x}$. The computation with P' will have this portion replaced by the following:

$$\dfrac{\dfrac{\dfrac{\theta\xi : F\xi \;\Rightarrow\; fail}{\vdots}}{\theta\xi : \exists\vec{y}(F)\xi \;\Rightarrow\; fail} \qquad \theta\xi : H\xi, \alpha\xi \;\Rightarrow\; \rho}{\theta\xi : \neg\exists\vec{y}(F)\xi, H\xi, \alpha\xi \;\Rightarrow\; \rho}$$

$$\dfrac{\theta\xi : F\xi \;\Rightarrow\; fail}{\theta\xi : if[\vec{y}](F,G)\xi, \alpha\xi \;\Rightarrow\; fail} \qquad \dfrac{\theta\xi : (\neg\exists\vec{y}(F)\&H)\xi, \alpha\xi \;\Rightarrow\; \rho}{\theta\xi : if[\vec{y}](F,G)\xi \vee (\neg\exists\vec{y}(F)\&H)\xi, \alpha\xi \;\Rightarrow\; \rho}$$

$$\dfrac{\dfrac{\dfrac{\theta\xi : if[\vec{y}](F,G)\xi \vee (\neg\exists\vec{y}(F)\&H)\xi, \alpha\xi \;\Rightarrow\; \rho}{\vdots}}{\theta : \vec{t} = \vec{x}, if[\vec{y}](F,G) \vee (\neg\exists\vec{y}(F)\&H), \alpha \;\Rightarrow\; \rho}}{\theta : p(\vec{t}) using(p(\vec{x}) \;:-\; if[\vec{y}](F,G) \vee (\neg\exists\vec{y}(F)\&H)), \alpha \;\Rightarrow\; \rho}$$

The three judgements at the top of this P' computation portion consist of two instances of one of the judgements at the top of the P portion, and one instance of the other one.

Since all the individual transformations preserve result, we conclude that the entire transformation process preserves result. □

A.2 Witness properties of conservative semantics

Lemma 3

Let θ, α be such that $(\theta : \alpha \;\Rightarrow\; fail)$ in the conservative semantics. Then for any ξ, $(\theta : \alpha\xi \;\Rightarrow\; fail)$ in the conservative semantics.

Proof

By induction on the structure of the computation of $(\theta : \alpha \;\Rightarrow\; fail)$. Cases are on the bottommost rule applied.

Unif/succ: Let σ be the mgu found in the rule. If $\xi \subseteq \sigma$, then $s\xi$ and $t\xi$ are identical, and the result follows from the induction hypothesis (IH). Otherwise, if $s\xi$ and $t\xi$ have mgu σ', then since σ is an mgu of s and t, there must be some ξ' such that $\xi\sigma' = \sigma\xi'$. The result then follows from the IH. Otherwise, $s\xi$ and $t\xi$ do not unify, and the computation fails with a single Unif/fail step.

Unif/fail: If $s\xi$ and $t\xi$ had a unifier σ, then s and t would have a unifier $\xi\sigma$. Since s and t have no unifier, the computation of $\theta : \alpha\xi \;\Rightarrow\; fail$ also consists of just one Unif/fail step.

Success: Cannot occur.

Conj, Disj/nofail, Disj/fail: Directly from the IH.

Exists: We have not required that the substitution ξ substitutes a term for x'. Therefore the result follows from the IH.

Not/succ: B has no free variables, so the computation $\theta : B\xi \;\Rightarrow\; fail$ is the same as that for $\theta : B \;\Rightarrow\; fail$.

Not/fail: Again, B has no free variables, so the computation of the left-hand premise is the same. The result then follows from the IH.

https://doi.org/10.1017/S1471068402001540 Published online by Cambridge University Press

Not/flounder, Not/sub: cannot occur.

If/succ: We must prove that $\theta : if[\vec{x}](B, C\xi), \alpha\xi \Rightarrow fail$. ($B$ has no free variables other than $\vec{x}$, and if binds the variables $\vec{x}$. We assume without loss of generality that $dom(\xi) \cap \{\vec{x}\} = \emptyset$.) For this, it suffices to prove that, for some θ', $\theta : B[\vec{x} := \vec{x}'] \Rightarrow \theta'$ (which it does by assumption), and that $\theta' : C\xi[\vec{x} := \vec{x}']\theta', \alpha\xi \Rightarrow fail$. Because $\vec{x}'$ do not appear in the conclusion, $C\xi[\vec{x} := \vec{x}']\theta'$ is the same thing as $C[\vec{x} := \vec{x}']\theta'\xi$. The result therefore follows from the induction hypothesis.

If/fail: B has no free variables other than the $\vec{x}$ variables, so the computation $\theta : B[\vec{x} := \vec{x}']\xi \Rightarrow fail$ is the same as that for $\theta : B[\vec{x} := \vec{x}'] \Rightarrow fail$. By the hypothesis, this computation fails.

If/flounder, If/sub: Cannot occur.

Pred: Directly from the IH. □

Lemma 4

Let θ, α be such that $\alpha\theta \equiv \alpha$ and $\theta : \alpha \Rightarrow \theta'$ in the conservative semantics. Then $\theta' \subseteq \theta$.

Proof

By induction on the structure of the computation. The only rule which modifies the substitution in the judgements is the Unif/succ rule, which obviously produces a more specific substitution. All other cases are straightforward consequences of the induction hypothesis. □

Lemma 5

Let θ, α be such that $\theta : \alpha \Rightarrow \theta'$ in the conservative semantics. Let V be a subset of the free variables of α. Then for any ξ grounding V consistent with θ', $\theta : \alpha\xi \Rightarrow \theta'\xi$ in the conservative semantics.

Proof

By induction on the structure of the computation. Cases are on the bottommost rule.

Unif/success: Let σ be the mgu found in the rule. By substitution monotonicity, any ξ grounding V consistent with θ' must also ground V consistent with σ. Thus $\alpha\xi\sigma$ is the same as $\alpha\sigma\xi$, and the result follows from the induction hypothesis (IH).

Unif/fail: Cannot occur.

Success: Trivial.

Conj, Disj/nofail: Directly from the IH.

Disj/fail: From the General Failure Property, we have that $\theta : B\xi, \alpha\xi \Rightarrow fail$. From the IH, we have that $\theta : C\xi, \alpha\xi \Rightarrow \theta'\xi$. The result follows in one Disj/fail step.

Exists: Because V is also a subset of the free variables of $B[x := x']$, the result follows from the IH.

Not/succ: Cannot occur.

Not/fail: Because B has no free variables, $B\xi$ is the same as B. The result follows from the original left-hand premise and from the IH.

Not/flounder, Not/sub: Cannot occur.

https://doi.org/10.1017/S1471068402001540 Published online by Cambridge University Press

If/succ: We assume without loss of generality that $dom(\xi) \cap \{\vec{x}\} = \emptyset$. (We can do this because the $\vec{x}$ variables are renamed and thus can be prevented from appearing in θ'.) We must therefore prove that $(\theta : if[\vec{x}](B, C\xi), \alpha\xi \Rightarrow \theta'\xi)$. By assumption, $\theta : B[\vec{x} := \vec{x}'] \Rightarrow \theta''$ for some θ''. By the IH, $\theta'' : C[\vec{x} := \vec{x}']\theta''\xi, \alpha\xi \Rightarrow \theta'\xi$. By substitution monotonicity, ξ must ground V consistent with θ'' as well. Thus $C[\vec{x} := \vec{x}']\theta''\xi$ is the same as $(C\xi)[\vec{x} := \vec{x}']\theta''$, and the result follows in one If/succ step.

If/fail, If/flounder, If/sub: Cannot occur.

Pred: Directly from the IH. □

A.3 DFNF results

Theorem 10

If $A \rhd A_1$ and $A \rhd A_2$, then there is an A_3 such that $A_1 \rhd^* A_3$ and $A_2 \rhd^* A_3$.

Proof

There are five cases, one for each of the rules R1-R5 applied to derive A_1 from A. We will give only the argument for R1, since the arguments for the rest are similar or simpler. We write $A[B_1, \ldots, B_n]$ for a formula A with distinguished subformulas $B_1, \ldots, B_n$, and $A[C_1, \ldots, C_n]$ for that formula with the distinguished $B_1, \ldots, B_n$ replaced by $C_1, \ldots, C_n$.

Let A be $A[(B_1 \vee B_2)\&C]$, and A_1 be $A[(B_1\&C) \vee (B_2\&C)]$. If A_2 is derived from applying R1 to the same location, the result is trivially true. A_2 cannot be derived from applying R2 to the same location, because $(B_1 \vee B_2)$ is not negated-disjunction. A_2 also cannot be derived from applying R3-R5 to the same location. We therefore have four subcases. In the first three subcases, A_2 may be one of $A[(B_1' \vee B_2)\&C]$, $A[(B_1 \vee B_2')\&C]$, or $A[(B_1 \vee B_2)\&C']$. In the first subcase, one step from either A_1 or A_2 will lead to $A[(B_1'\&C) \vee (B_2\&C)]$. The second subcase is similar. In the third subcase, two steps from A_1 and one from A_2 will lead to $A[(B_1\&C') \vee (B_2\&C')]$. The final subcase is when A can be written as $A[(B_1 \vee B_2)\&C, D]$, A_1 is $A[(B_1\&C) \vee (B_2\&C), D]$, and A_2 is $A[(B_1 \vee B_2)\&C, D']$. In this case, one step from either A_1 or A_2 will lead to $A[(B_1\&C) \vee (B_2\&C), D']$. □

Lemma 7

If $G \rhd G'$, then $pd(G) \geqslant pd(G')$.

Proof

Clearly rules R1–R3 maintain potential depth; the difficult cases are R4 and R5.

Case R4: If R4 was applied at the top level, then we have $G = if[\vec{x}]((B_1 \vee B_2), C)$ and $G' = if[\vec{x}](B_1, C) \vee (\neg(\exists\vec{x}B_1)\&if[\vec{x}](B_2, C))$. Let the length of $\vec{x}$ be n. Now we

https://doi.org/10.1017/S1471068402001540 Published online by Cambridge University Press

have that

$$\begin{array}{rcll} pd(G') & = & pd(if\,[\vec{x}](B_1, C) \vee (\neg(\exists\vec{x}B_1)\&if\,[\vec{x}](B_2, C)) \\ & = & max(& pd(if\,[\vec{x}](B_1, C) + 1, \\ & & & pd(\neg(\exists\vec{x}B_1)\&if\,[\vec{x}](B_2, C)) + 1) \\ & = & max(& 1 + n + 2pd(B_1) + max(pd(B_1), pd(C)), \\ & & & pd(\neg(\exists\vec{x}B_1)\&if\,[\vec{x}](B_2, C)) + 1) \\ & = & max(& 1 + n + 3pd(B_1), 1 + n + 2pd(B_1) + pd(C), \\ & & & pd(\neg(\exists\vec{x}B_1) + 2, \\ & & & pd(if\,[\vec{x}](B_2, C)) + 2) \\ & = & max(& 1 + n + 3pd(B_1), 1 + n + 2pd(B_1) + pd(C), \\ & & & 3 + n + pd(B_1), \\ & & & 2 + n + 2pd(B_2) + max(pd(B_2), pd(C)) \\ & = & max(& 1 + n + 3pd(B_1), 1 + n + 2pd(B_1) + pd(C), \\ & & & 3 + n + pd(B_1), \\ & & & 2 + n + 3pd(B_2), 2 + n + 2pd(B_2) + pd(C)) \\ & = & max(& 1 + n + 3pd(B_1), 1 + n + 2pd(B_1) + pd(C), \\ & & & 2 + n + 3pd(B_2), 2 + n + 2pd(B_2) + pd(C)) \end{array}$$

There are now two subcases. Subcase 1: if $pd(B_1) > pd(B_2)$, then

$$\begin{array}{rcl} pd(G) & = & pd(if\,[\vec{x}]((B_1 \vee B_2), C) \\ & = & n + 2pd(B_1 \vee B_2) + max(pd(B_1 \vee B_2), pd(C)) \\ & = & n + 2 + 2pd(B_1) + max(1 + pd(B_1), pd(C)) \\ & = & max(3 + n + 3pd(B_1), 2 + n + 2pd(B_1) + pd(C)) \end{array}$$

and $pd(G')$ simplifies to $max(1 + n + 3pd(B_1), 1 + n + 2pd(B_1) + pd(C))$. Thus if $pd(C) > pd(B_1)$, we have

$$pd(G) = (2 + n + 2pd(B_1) + pd(C)) > (1 + n + 2pd(B_1) + pd(C)) = pd(G')$$

and otherwise ($pd(C) \leqslant pd(B_1)$) we have

$$pd(G) = (3 + n + 3pd(B_1)) > (1 + n + 3pd(B_1)) = pd(G')$$

so in both cases, $pd(G) > pd(G')$.

Subcase 2: otherwise, $pd(B_2) \geqslant pd(B_1)$. We have:

$$pd(G) = max(3 + n + 3pd(B_2), 2 + n + 2pd(B_2) + pd(C))$$

and $pd(G')$ simplifies to $max(2 + n + 3pd(B_2), 2 + n + 2pd(B_2) + pd(C))$. Thus if $pd(C) > pd(B_2)$, we have

$$pd(G) = (2 + n + 2pd(B_2) + pd(C)) = (2 + n + 2pd(B_2) + pd(C)) = pd(G')$$

and otherwise ($pd(C) \leqslant pd(B_1)$) we have

$$pd(G) = (3 + n + 3pd(B_2)) > (2 + n + 3pd(B_2)) = pd(G')$$

so in both cases, $pd(G) \geqslant pd(G')$.

Similarly, if R4 was applied not at the top level, $pd(G) \geqslant pd(G')$, since if any subformula is transformed to have lower potential depth, the whole formula has lower potential depth.

https://doi.org/10.1017/S1471068402001540 Published online by Cambridge University Press

If R5 was applied at the top level, we have $pd(G) = pd(if[\vec{x}](B,C)) = n + 2pd(B) + max(pd(B), pd(C)) = max(n + 3pd(B), n + 2pd(B) + pd(C))$, and $pd(G') = pd(\exists\vec{x}(B\&C)) = 1 + n + max(pd(B), pd(C)) = max(1 + n + pd(B), 1 + n + pd(C))$. If $pd(B) > pd(C)$, then

$$pd(G) = n + 3pd(B) > 1 + n + pd(B) = pd(G')$$

and otherwise

$$pd(G) = n + 2pd(B) + pd(C) > 1 + n + pd(C) = pd(G')$$

Thus in both cases $pd(G) > pd(G')$.

Similarly, if R5 was applied not at the top level, $pd(G) > pd(G')$. □

Corollary 12.

For every formula G not in normal form, there is a unique formula G″ in normal form, such that for all G′ such that $G \rhd G'$, we have that $G' \rhd^* G''$.

Proof

Let k be the length of the longest chain of rewritings that starts with G (by Theorem 11 we know that this bound exists). We prove the corollary by induction on k. In the base case ($k = 1$), we know from Theorem 10 that there can be at most one unique G' such that $G \rhd G'$; hence, G'' is this G'. In the inductive case, if there is a unique G' such that $G \rhd G'$, the result follows from the induction hypothesis. If there is more than one, then for each pair G_1 and G_2 such that $G \rhd G_1$ and $G \rhd G_2$, by Theorem 10 there is some G_3 such that $G_1 \rhd^* G_3$ and $G_2 \rhd^* G_3$. But by the induction hypothesis, there is some unique normal form not only of G_3 but also of G_1 and G_2. Because $G_1 \rhd^* G_3$, the normal form of G_3 must be the same as that of G_1, and similarly for G_2. Hence all G' such that $G \rhd G'$ must have some unique normal form G''. This therefore is the unique normal form of G. □

Theorem 14.

If α' is α with some formulas transformed by applications of rules R1–R5, then $\theta : \alpha \Rightarrow \rho$ in the pessimistic semantics iff $\theta : \alpha' \Rightarrow \rho$ in the pessimistic semantics.

Proof

By induction on the structure of the assumption computation. If the application of the rules has not changed the top-level connective of the first formula in α, then the result follows by the induction hypothesis. Otherwise, we have cases according to which of R1-R5 was used to transform the top-level connective of the first formula.

Cases R1–R3 are very similar to the proof in Andrews (1997), and will not be repeated here.

Case R4: The two computations are $(\theta : if[\vec{x}((B_1 \vee B_2), C), \alpha \Rightarrow \rho)$ and $(\theta : if[\vec{x}](B_1, C) \vee (\neg(\exists\vec{x}B_1)\&if[\vec{x}](B_2, C)), \alpha \Rightarrow \rho)$; we must show that each implies the other. There are several subcases.

If $(\theta : B_1[\vec{x} := \vec{x}'] \Rightarrow \theta')$ and $(\theta' : C\theta', \alpha \Rightarrow \rho)$, where ρ is either some θ'' or *diverge*, then we have the following original computation:

https://doi.org/10.1017/S1471068402001540 Published online by Cambridge University Press

$$\dfrac{\dfrac{\theta : B_1[\vec{x} := \vec{x}'] \;\Rightarrow\; \theta'}{\theta : (B_1 \vee B_2)[\vec{x} := \vec{x}'] \;\Rightarrow\; \theta'} \qquad \theta' : C\theta', \alpha \;\Rightarrow\; \rho}{\theta : if[\vec{x}((B_1 \vee B_2), C), \alpha \;\Rightarrow\; \rho}$$

The corresponding computation with the transformed formula is:

$$\dfrac{\dfrac{\theta : B_1[\vec{x} := \vec{x}'] \;\Rightarrow\; \theta' \qquad \theta' : C\theta', \alpha \;\Rightarrow\; \rho}{\theta : if[\vec{x}](B_1, C)\alpha \;\Rightarrow\; \rho}}{\theta : if[\vec{x}](B_1, C) \vee (\neg(\exists\vec{x}B_1)\&if[\vec{x}](B_2, C)), \alpha \;\Rightarrow\; \rho}$$

If $(\theta : B_1[\vec{x} := \vec{x}'] \;\Rightarrow\; \theta')$ but $(\theta' : C\theta', \alpha \;\Rightarrow\; fail)$, then we have the following original computation:

$$\dfrac{\dfrac{\theta : B_1[\vec{x} := \vec{x}'] \;\Rightarrow\; \theta'}{\theta : (B_1 \vee B_2)[\vec{x} := \vec{x}'] \;\Rightarrow\; \theta'} \qquad \theta' : C\theta', \alpha \;\Rightarrow\; fail}{\theta : if[\vec{x}((B_1 \vee B_2), C), \alpha \;\Rightarrow\; fail}$$

The corresponding computation with the transformed formula is:

$$\dfrac{\dfrac{}{\theta : if[\vec{x}](B_1, C)\alpha \;\Rightarrow\; fail} \qquad \dfrac{\dfrac{\dfrac{\dfrac{\theta : B_1[\vec{x} := \vec{x}'] \;\Rightarrow\; \theta'}{\vdots}}{\theta : \exists\vec{x}B_1 \;\Rightarrow\; \theta'}}{\theta : \neg(\exists\vec{x}B_1), if[\vec{x}](B_2, C), \alpha \;\Rightarrow\; fail}}{\theta : (\neg(\exists\vec{x}B_1)\&if[\vec{x}](B_2, C)), \alpha \;\Rightarrow\; fail}}{\theta : if[\vec{x}](B_1, C) \vee (\neg(\exists\vec{x}B_1)\&if[\vec{x}](B_2, C)), \alpha \;\Rightarrow\; fail}$$

where the computation of the left-hand premise of the bottommost judgement is as follows:

$$\dfrac{\theta : B_1[\vec{x} := \vec{x}'] \;\Rightarrow\; \theta' \qquad \theta' : C\theta, \alpha \;\Rightarrow\; fail}{\theta : if[\vec{x}](B_1, C)\alpha \;\Rightarrow\; fail}$$

We have very similar cases when $(\theta : B_1[\vec{x} := \vec{x}'] \;\Rightarrow\; fail)$ but $(\theta : B_2[\vec{x} := \vec{x}'] \;\Rightarrow\; \theta')$, depending on the result of $(\theta' : C\theta', \alpha)$.

When both $(\theta : B_1[\vec{x} := \vec{x}'] \;\Rightarrow\; fail)$ and $(\theta : B_2[\vec{x} := \vec{x}'] \;\Rightarrow\; fail)$, we have the following original computation:

$$\dfrac{\dfrac{\theta : B_1[\vec{x} := \vec{x}'] \;\Rightarrow\; fail \qquad \theta : B_2[\vec{x} := \vec{x}'] \;\Rightarrow\; fail}{\theta : (B_1 \vee B_2)[\vec{x} := \vec{x}'] \;\Rightarrow\; fail}}{\theta : if[\vec{x}((B_1 \vee B_2), C), \alpha \;\Rightarrow\; fail}$$

The corresponding computation with the transformed formula is:

$$\dfrac{\dfrac{\theta : B_1[\vec{x} := \vec{x}'] \;\Rightarrow\; fail}{\theta : if[\vec{x}](B_1, C), \alpha \;\Rightarrow\; fail} \qquad \theta : (\neg(\exists\vec{x}B_1)\&if[\vec{x}](B_2, C)), \alpha \;\Rightarrow\; fail}{\theta : if[\vec{x}](B_1, C) \vee (\neg(\exists\vec{x}B_1)\&if[\vec{x}](B_2, C)), \alpha \;\Rightarrow\; fail}$$

where the computation of the right-hand premise of the bottommost judgement is:

$$\dfrac{\dfrac{\dfrac{\theta : B_1[\vec{x} := \vec{x}'] \;\Rightarrow\; fail}{\vdots}}{\theta : \exists\vec{x}B_1 \;\Rightarrow\; fail} \qquad \dfrac{\theta : B_2[\vec{x} := \vec{x}'] \;\Rightarrow\; fail}{\theta : if[\vec{x}](B_2, C), \alpha \;\Rightarrow\; fail}}{\theta : \neg(\exists\vec{x}B_1), if[\vec{x}](B_2, C), \alpha \;\Rightarrow\; fail}$$
$$\dfrac{}{\theta : (\neg(\exists\vec{x}B_1)\&if[\vec{x}](B_2, C)), \alpha \;\Rightarrow\; fail}$$

https://doi.org/10.1017/S1471068402001540 Published online by Cambridge University Press

The subcases in which a result of *diverge* arises are similar to those in which a result of *fail* arises.

Case R5: The two computations are $(\theta : if[\vec{x}](B,C),\alpha \Rightarrow \rho)$ and $(\theta : \exists\vec{x}(B\&C),\alpha \Rightarrow \rho)$; we must show that one implies the other. We also know that B is negated-disjunction. There are two subcases.

If $(\theta : B[\vec{x} := \vec{x}'] \Rightarrow \theta')$, then we have the following original computation:

$$\frac{\theta : B[\vec{x} := \vec{x}'] \Rightarrow \theta' \qquad \theta' : C[\vec{x} := \vec{x}']\theta', \alpha \Rightarrow \rho}{\theta : if[\vec{x}](B,C),\alpha \Rightarrow \rho}$$

However, because B is negated-disjunction, every computation in the pessimistic semantics with substitution and goal stack $(\theta : B[\vec{x} := \vec{x}'],\alpha')$ must contain a substitution and goal stack $(\theta' : \alpha')$. (See Lemma 4.6 of Andrews (1997).) Thus we have the following computation with the transformed formula:

$$\cfrac{\cfrac{\cfrac{\theta' : C[\vec{x} := \vec{x}']\theta', \alpha\theta' \Rightarrow \rho}{\vdots}}{\theta : B[\vec{x} := \vec{x}'], C[\vec{x} := \vec{x}'], \alpha \Rightarrow \rho}}{\cfrac{\theta : (B\&C)[\vec{x} := \vec{x}'],\alpha \Rightarrow \rho}{\vdots}}$$
$$\theta : \exists\vec{x}(B\&C),\alpha \Rightarrow \rho$$

However, since θ' applies only to the free variables of $B[\vec{x} := \vec{x}']$, which are $\vec{x}'$, and α does not contain these variables, the topmost judgement is equivalent to $(\theta' : C[\vec{x} := \vec{x}']\theta', \alpha \Rightarrow \rho)$.

Otherwise, $(\theta : B[\vec{x} := \vec{x}'] \Rightarrow \rho)$ where ρ is *fail* or *diverge*. In this subcase, the bottom of the original computation is as follows:

$$\frac{\theta : B[\vec{x} := \vec{x}'] \Rightarrow \rho}{\theta : if[\vec{x}](B,C),\alpha \Rightarrow \rho}$$

The bottom of the computation with the transformed formula is as follows:

$$\cfrac{\cfrac{\theta : B[\vec{x} := \vec{x}'], C[\vec{x} := \vec{x}'], \alpha \Rightarrow \rho}{\theta : (B\&C)[\vec{x} := \vec{x}'],\alpha \Rightarrow \rho}}{\vdots}$$
$$\theta : \exists\vec{x}(B\&C),\alpha \Rightarrow \rho$$

The presence of the extra formulas ($C[\vec{x} := \vec{x}']$ and α) has no effect on the computation. □

References

Abadi, M. and Cardelli, L. (1996) *A Theory of Objects*. Springer.

Andrews, J. H. (1991) *Logic Programming: Operational semantics and proof theory*. Distinguished Dissertation Series. Cambridge University Press.

Andrews, J. H. (1995) A paralogical semantics for the Prolog cut. *Proceedings International Logic Programming Symposium*, pp. 591–605. MIT Press.

https://doi.org/10.1017/S1471068402001540 Published online by Cambridge University Press

Andrews, J. H. (1997) A logical semantics for depth-first Prolog with ground negation. *Theor. Comput. Sci.* **184**(1–2), 105–143.

Andrews, J. H. (1998) On the spreadsheet presentation of proof obligations. In: Backhouse, R. (ed.), *Proceedings Workshop on User Interfaces for Theorem Provers (UITP)*, pp. 34–41. Computing Science Report 98-08, Department of Mathematics and Computing Science, Eindhoven University of Technology.

Andrews, J. H. (1999) *The witness properties and the semantics of the Prolog cut*. Technical report 542, Department of Computer Science, University of Western Ontario.

Apt, K. and Pedreschi, D. (1993) Proving termination of general Prolog programs. *Infor. & Computation*, **106**, 109–157.

Apt, K. R. and Marchiori, E. (1994) Reasoning about Prolog programs: From modes through types to assertions. *Formal Aspects of Computing*, **6A**, 743–764.

Arbab, B. and Berry, D. M. (1987) Operational and denotational semantics of Prolog. *J. Logic Program.* **4**, 309–329.

Barbuti, R. and Martelli, M. (1990) Recognizing non-floundering logic programs and goals. *Int. J. Foundations of Comput. Sci.* **1**(2), 151–163.

Baudinet, M. (1992) Proving termination properties of Prolog programs: A semantic approach. *J. Logic Program.* **14**(1), 1–29.

Belnap, N. D. Jr. (1977) A useful four-valued logic. In: Dunn, J. M. and Epstein, G. (eds.), *Modern Uses of Multiple-valued Logic*, pp. 8–37. Reidel.

Bezem, M. (1993) Strong termination of logic programs. *J. Logic Program.* **15**, 79–97.

Billaud, M. (1990) Simple operational and denotational semantics for Prolog with cut. *Theor. Comput. Sci.* **71**, 193–208.

Börger, E. (1990) *A logical operational semantics of full Prolog*. Technical report IWBS Report 111, IBM Wissenschaftliches Zentrum, Institut für Wissensbasierte Systeme, Heidelberg, Germany.

Burstall, R. M. and Darlington, J. (1977) A transformation system for developing recursive programs. *J. ACM*, **24**(1), 44–67.

Clark, K. L. (1978) Negation as failure. *Logic and Data Bases*, pp. 293–322. Plenum Press.

Dahl, V. (1980) Two solutions for the negation problem. In: Tärnlund, S.-Å. (eds.), *Second International Workshop on Logic Programming*, pp 61–72. Debrecen, Hungary.

de Bruin, A. and de Vink, E. P. (1989) Continuation semantics for Prolog with cut. In: *Theory and Practice of Software Engineering: Lecture Notes in Computer Science 351*, pp. 178–192. Springer-Verlag.

Debray, S. and Mishra, P. (1988) Denotational and operational semantics of Prolog. *J. Logic Program.* **5**, 61–91.

Deransart, P. and Ferrand, G. (1987) *An operational formal definition of Prolog*. Technical report RR763, INRIA.

Elbl, B. (1999) A declarative semantics for depth-first logic programs. *J. Logic Program.* **41**(1), 27–66.

Etalle, S. (1998) A semantics for modular general logic programs. *Theor. Comput. Sci.* **206**(1-2), 51–80.

Fitting, M. (1985) A Kripke-Kleene semantics for logic programs. *J. Logic Program.* **4**, 295–312.

Gabbrieli, M. and Levi, G. (1992) Unfolding and fixpoint semantics of concurrent constraint logic programs. *Theor. Comput. Sci.* **105**, 85–128.

Gabbrielli, M. and Etalle, S. (1999) Layered modes. *J. Logic Program.* **39**(1–3), 225–244.

Jones, N. D. and Mycroft, A. (1984) Stepwise development of operational and denotational

semantics for Prolog. *Proceedings International Symposium on Logic Programming*, pp. 281–288. IEEE Computer Society.

Kripke, S. (1975) Outline of a theory of truth. *J. Philosophy*, **72**, 690–716.

Lindenstrauss, N. and Sagiv, Y. (1997) Automatic termination analysis of logic programs. In: Naish, L. (ed.), *Proceedings 14th International Conference on Logic Programming*. MIT Press.

Lindenstrauss, N., Sagiv, Y. and Serebrenik, A. (1997) Termilog: A system for checking termination of queries to logic programs. In: Grumberg, O. (ed.), *Computer Aided Verification, 9th International Conference: Lecture Notes in Computer Science 1254*, pp. 444–447. Springer-Verlag.

Loveland, D. W. and Reed, D. W. (1991) A near-Horn Prolog for compilation. *Computational Logic: Essays in honor of Alan Robinson*, pp. 542–564. MIT Press.

Naish, L. (1986) *Negation and control in Prolog: Lecture Notes in Computer Science 238*. Springer-Verlag.

Nicholson, T. and Foo, N. (1989) A denotational semantics for Prolog. *ACM Trans. Program. Lang. & Syst*. **11**, 650–665.

Pereira, F., Warren, D., Bowen, D., Byrd, L. and Pereira, L. (n.d.) *C-Prolog user's manual*. Technical report, EdCAAD, Department of Architecture, Univ. of Edinburgh, Edinburgh.

Pierro, A., Martelli, M. and Palamidessi, C. (1995) Negation as instantiation. *Infor. & Computation*, **120**(2), 263–278.

Plotkin, G. (1981) *A structural approach to operational semantics*. Technical report DAIMI FN-19, Computer Science Department, Aarhus University, Aarhus.

Plümer, L. (1990) *Termination proofs for logic programs: Lecture Notes in Artificial Intelligence 446*. Springer-Verlag.

Somogyi, Z., Henderson, F. and Conway, T. (1996) The execution algorithm of Mercury, an efficient purely declarative logic programming language. *J. Logic Program.* **29**(1-3), 17–64.

Stärk, R. (1998) The theoretical foundations of LPTP (a logic program theorem prover). *J. Logic Program.* **36**(3), 241–269.

Tamaki, H. and Sato, T. (1984) Unfold/fold transformations of logic programs. *Proceedings 2nd International Logic Programming Conference*. Uppsala, Sweden.

van Emden, M. H. and Kowalski, R. A. (1976) The semantics of predicate logic as a programming language. *J. ACM* **23**(4), 733–742.