Formalizing a logic for logic programming

John S. Schlipf [1]

Center for Intelligent Systems, Department of Computer Science, University of Cincinnati, Cincinnati, OH 45221-0008, USA

1. Introduction

Much research in the last few years has centered upon an idea called *negation as failure*. The basic idea of negation as failure is that, if an atomic "fact" (atomic sentence) is true, it must be demonstrably true – so if we cannot demonstrate that the atomic sentence is true, we should infer it to be false.

Negation as failure clearly is not logically sound. In classical logic, to infer a sentence to be false, we must demonstrate that the atomic sentence is false – not merely that it is undemonstrable. Nevertheless it has natural appeal. Most noticeably, as pointed out by McCarthy, Reiter, and others, it is invoked in a great many "common-sense" human inferences.

Example 1.1 (Reiter)

If a patient goes to a physician with a problem that could be caused by either pneumonia or a sprained ankle, the physician assumes – in the absence of evidence to the contrary – that the patient has only one of these problems.

Example 1.2

If company records do not show that Mr. Jones was ever a vice president, then management infers that he never was a vice president.

A major research topic concerning negation as failure is the attempt to find the "correct" semantics. We shall use the term *semantics* to mean what is usually called *declarative semantics*. The declarative semantics of a logic program tells *what* is to inferred – but does not prescribe how. Our point of view here is that goal of the study of semantics for negation as failure is to find a reasonable and fairly elegant mathematical formalism that captures *as much as possible* of ordinary human negation-as-failure type reasoning. There is of course

[1] This work was partially supported by NSF grants IRI-8705184 and IRI-8905166.

a danger: human negation-as-failure type reasoning often appears *ad hoc*, so it seems entirely possible that there is no entirely satisfactory way to capture it formally. But we believe the goal of finding one approach that captures much, or most, of "common sense reasoning" needs to be pursued.

Our starting point is not a search through ordinary human usage. Rather, we start by comparing various semantics people have already given for negation as failure. These semantics, we feel, all have clear "common sense" justification. We shall identify certain features of these semantics as *goals* of a semantics for a common sense negation as failure.

We start from the following vague statement of negations as failure:

> If it becomes *obvious* that an atomic sentence $R(t_1, t_2, \ldots, t_n)$ is not provable (*in whatever logical system is chosen as a paradigm*), infer that the sentence is false.

Of course, there are then two natural questions:

(1) When is it *obvious* that an atomic sentence is not provable?
(2) What logical system is chosen as the paradigm?

From our perspective here, these two questions distinguish many of the standard semantics for negation as failure. And we shall show that, by choosing a standard cautious interpretation of *obvious* and by creating an appropriate paradigm for the logic, we can meet all but one of the goals we suggest.

2. Logic programming background

The whole endeavor is simplified in the restricted realm of *logic programming*.

Example 2.1

The following is a logic program:

$$\{even(0),\ even(succ(succ(x))) \leftarrow even(x) \wedge \neg\, even(succ(x))\}$$

DEFINITION 2.1

A *literal* is an atomic formula $R(t_1, t_2, \ldots, t_n)$ (a *positive literal*) or a negated atomic formula $\neg\, R(t_1, t_2, \ldots, t_n)$ (a *negative literal*).

A *logic program* is a finite, or countably infinite, set of *rules*, (implicitly universally quantified) formulas of the form

$$\alpha \leftarrow \beta_1 \wedge \beta_2 \wedge \ldots \wedge \beta_n,$$

where α is a positive literal and the β_i's are all literals – positive or negative.

The literal α is called the *head* of the rule; each β_i is a *subgoal*, and the conjunction $\beta_1 \wedge \beta_2 \wedge \ldots \wedge \beta_n$ is called the body. The $\leftarrow$ symbol is some sort of implication.

Later on it will be convenient to treat a rule $p \leftarrow$ with no body as being implicitly $p \leftarrow$ **true**.

Attention is traditionally limited to *Herbrand models*:

DEFINITION 2.2

An Herbrand model is a model whose universe is the *Herbrand universe* of the program: the set of variable-free terms of the language. (Distinct terms are unequal elements.)

Example 2.2

For the program $\{even(0),\ even(succ(succ(x))) \leftarrow even(x) \wedge \neg\, even(succ(x))\}$ the Herbrand universe is the set of terms $\{0,\ succ(0),\ succ(succ(0)),\ succ(succ(succ(0))),\ \ldots\}$ – the natural model of the integers with the successor function.

Using the Herbrand universe blurs the distinction between the language and the model. This is a convenience in definitions.

DEFINITION 2.3

The *ground instantiation* of a logic program is formed by substituting elements of the Herbrand universe in for variables of the logic program in all possible ways.

Example 2.3

Consider again $\{even(0),\ even(succ(succ(x))) \leftarrow even(x) \wedge \neg\, even(succ(x))\}$. Its ground instantiation is:

$$\begin{aligned}
&\{even(0),\\
&even(succ(succ(0))) \leftarrow even(0) \wedge \neg\, even(succ(0)),\\
&even(succ(succ(succ(0)))) \leftarrow even(succ(0)) \wedge \neg\, even(succ(succ(0))),\\
&even(succ(succ(succ(succ(0)))))\\
&\quad \leftarrow even(succ(succ(0))) \wedge \neg\, even(succ(succ(succ(0)))),\\
&\ldots,\}
\end{aligned}$$

OBSERVATION 2.1 (WELL KNOWN)

An Herbrand model is a model of a logic program if and only if it is a model of the ground instantiation of the logic program.

Since the ground instantiation contains no variables at all, each atomic sentence $R(t_1, \ldots, t_k)$ of the instantiation may be treated as a proposition letter $p_{R(t_1, \ldots, t_k)}$. Hence, over its Herbrand universe, each logic program is equivalent to a propositional logic program. (Of course, a finite logic program in propositional logic usually has an infinite ground instantiation.) Hence from now on we may make all our definitions in terms of *infinite programs of propositional logic*. The terms *positive literal* and *negative literal* will be used for these proposition letters and their negations.

DEFINITION 2.4

A *partial interpretation* is a set of proposition letters and negated proposition letters. A partial interpretation I is *consistent*, or *3-valued*, if for no proposition letter a are a and $\neg a$ both in I. A partial interpretation I is *2-valued* if for each proposition letter a, either a or $\neg a$ is in I, but not both. A *model of a program* is a 2-valued interpretation which is a model in the ordinary sense of classical logic.

Think of a partial interpretation I as a set inferences made. In only one circumstance in this paper will inconsistent partial interpretations arise: when a logic programming semantics "considers" a logic program incoherent and infers *all* proposition letters, both positive and negative.

DEFINITION 2.5

A *logic programming semantics* is a function **S** which assigns, to each logic program **P**, a partial interpretation **S(P)**. **S(P)** is the set of literals which the semantics "tells us to infer" from **P**.

3. A semantics survey

We begin with a survey of some of the logic programming semantics. We do this, in part, because of our motivation: to try to find a formalism describing as much as possible of the way negation as failure has been applied. [2] It is part of this goal to accept, at least in part, and to try to reconcile, several of the major semantics for logic programming. But in addition, our suggested semantics is constructed identically to other semantics, except that it changes the logical paradigm, and many of the properties are analogous.

3.1. SEMANTICS BASED UPON CLASSICAL LOGIC

The most obvious semantics is that of classical logic: infer all proposition letters true in all classical models of the (ground instantiation of the) program.

[2] Of course, we present the various semantics from a point of view appropriate to this paper.

Steeped in classical logic as we are, we accept these inferences as valid. But these inferences totally miss negation as failure.

OBSERVATION 3.1 (WELL KNOWN)

Consider any logic program **P**. Since every rule has a positive head, **P** is satisfied if all proposition letters are true. Hence the set of classical consequences of **P** may contain no negative literals.

The original declarative semantics for logic programs is the Van Emden–Kowalski semantics [17] for Horn clause programs. There, a positive literal is inferred if an only if it is true in all (2-valued) models of the program; otherwise, its negation is inferred. This interpretation is itself a model of the program. Moreover, a proposition letter is in this interpretation if and only if it can be inferred from the program using only modus ponens as a deduction rule. [3] The Van Emden–Kowalski semantics captures both classical consequence and negation as failure in the limited context of Horn clause programs. But attempts to extend its notions to other logic programs have produced varied generalizations.

3.1.1. Minimal model semantics and the GCWH

DEFINITION 3.1

For (2-valued) models $\mathcal{M}$, $\mathcal{N}$ of a program **P**, say $\mathcal{M} \subseteq \mathcal{N}$ if every proposition letter true in $\mathcal{M}$ is also true in $\mathcal{N}$.

A (2-valued) model $\mathcal{M}$ of **P** is *minimal* if it is minimal in the $\subseteq$ ordering.

Example 3.1

Let **P** be $\{hasPneumonial(MrJones) \leftarrow \neg\, hasSprainedAnkle(MrJones)\}$, which has the form $\{a \leftarrow \neg \mathrm{b}\}$. Then **P** has three models: $\{a, b\}$, $\{a, \neg b\}$, $\{\neg a, b\}$. The last two are minimal, the first is not.

THEOREM 3.2 (WELL KNOWN)

Every logic program has a minimal model. Given any model $\mathcal{N}$ for a logic program **Q**, there is a minimal model $\mathcal{M}$ of **Q** where $\mathcal{M} \subseteq \mathcal{N}$.

DEFINITION 3.2

The minimal model semantics for logic program **P**, **Min(P)**, is the set of all literals true in all minimal models of **P**.

[3] Actually, we are using a variant of *modus ponens*: from $a \leftarrow \beta_1 \wedge \ldots \wedge \beta_n$ and $\beta_1, \ldots, \beta_n$, infer a. Thus we are really making inferences using just *modus ponens* and the $\wedge$-introduction rule. We shall continue to refer to this variant as *modus ponens*.

Example 3.2

Let **Q** be

$$\{hasPneumonia(MrJones) \leftarrow \neg\, hasSprainedAnkle(MrJones)$$

$$willMakeMeRich(X) \leftarrow hasPneumonia(X) \wedge hasSprainedAnkle(X)\}$$

Its ground instantiation has the form $\{a \leftarrow \neg\, b,\ c \leftarrow a \wedge b\}$. That has two minimal models: $\{a, \neg\, b, \neg\, c\}$, $\{\neg\, a, b, \neg\, c\}$ so $\mathbf{Min(Q)} = \{\neg\, c\}$, i.e., $\{\neg\, willMakeMeRich(MrJones)\}$.

OBSERVATION 3.3

For p a proposition letter appearing in a program P,

(1) $p \in Min(\mathbf{P})$ if and only if p is provable from **P** in classical logic.

(2) $\neg\, p \in Min(\mathbf{P})$ if and only if, for no set $\{\neg\, q_1, \ldots, \neg\, q_k\}$ of negative literals consistent with **P**, is p provable from $\mathbf{P} \cup \{\neg\, q_1, \ldots, \neg\, q_k\}$.

DEFINITION 3.3

The Generalized Closed World Hypothesis (GCWH) is the inference rule that if an atomic sentence is false in all minimal models, it should be inferred to be false.

The GCWH was first formulated by Minker.

3.2. SEMANTICS BASED UPON A MORE CONSTRUCTIVE INTERPRETATION OF $\leftarrow$

Later semantics have rejected the classical logic interpretations as too weak. They adopt a more constructive-logic type of paradigm, approximately:

> The only way to infer an atomic sentence is to deduce it from known facts by *modus ponens*.

Alternatively, the rules with any given atomic sentence in their head may be thought of as a *definition* of that atomic sentence.

Example 3.3

Let **P** be $\{a \leftarrow \neg\, b\}$. Then b cannot be inferred by *modus ponens* alone, so these semantics infer $\neg\, b$. Then by *modus ponens*, they infer a.

Note that this point of view explicitly rejects proof by contradiction. This can also be thought of as a reinterpretation of $\leftarrow$, rejecting the classical notion. Fitting [5] used this to develop his semantics, discussed briefly below.

3.2.1. *Stratified semantics*

Example 3.4

Let program **P** contain the atomic formulas

$$married(A, B),\ married(C, D),\ parent(A, C),\ parent(B, C)$$

plus the rules

$$married(X, Y) \leftarrow married(Y, X),$$

$$relative(X, Y) \leftarrow parent(X, Y),$$

$$relative(X, Y) \leftarrow relative(Y, X)$$

$$relative(X, Y) \leftarrow relative(X, Z) \wedge relative(Z, Y),$$

$$inLaw(X, Y) \leftarrow married(X, Z) \wedge relative(Z, Y) \wedge \neg\, relative(X, Y),$$

$$inLaw(X, Y) \leftarrow inLaw(Y, X)$$

The stratified semantics chooses here among minimal models. It divides the program into two pieces called *strata*, $\mathbf{P}_2$, the last two lines, and $\mathbf{P}_1$, all the rest. It exploits properties of the program: (1) Each relation is defined (appears as the head of a rule) in only one of the pieces. (2) The relation, *inLaw*, defined in $\mathbf{P}_2$ does not appear at all in $\mathbf{P}_1$. (3) Any relation defined R defined in any stratum $\mathbf{P}_t$ is used only positively in the bodies of rules in $\mathbf{P}_t$, though relations defined in $\mathbf{P}_1$ may appear both positively and negatively in the bodies of rules in $\mathbf{P}_2$.

In the stratified semantics the two strata are treated separately. The minimal model I_1 is constructed for the first part, as in the Van Emden–Kowalski semantics. So $\neg\, relative(A, D) \in I_1$. Then a minimal model I_2 is chosen for the entire program where $I_1 \subseteq I_2$. Since $\neg\, relative(A, D) \in I_1$, $\neg\, relative(A, D) \in I_2$ also. Hence the stratified semantics prefers a minimal interpretation containing $\neg\, relative(A, D)$, $inLaw(A, D)$ to one containing $relative(A, D)$, $\neg\, inLaw(A, D)$.

A program is *stratified* if it can be broken into a finite set of strata obeying the positive and negative occurrence properties above. The stratified semantics assigns truth values stratum by stratum as described. It was developed independently in [1,3,18]. Przymusinski [13] considered the extension, called *local stratification*, where the ground instantiation (as a propositional logic program) is stratified (in a possibly infinite but well-ordered set of strata). The failing of the stratified semantics is that many interesting programs (an example will be mentioned below) are not stratified, nor even locally stratified.

We isolate a property of the stratified semantics below. This property seems to be a consequence of both of the justifications we presented for the constructive semantics.

DEFINITION 3.4

A *stratified pair* of programs is a pair of programs **P**, **Q** where for each relation R, if R occurs in the head of a rule in **Q**, it does not appear at all in **P**.

DEFINITION 3.5

A logic programming semantics **S** obeys the *Weak Principle of Stratification* if, for every stratified pair **P**, **Q** of logic programs, if a relation R appears in **P** at all, then an atomic sentence $R(t_1, t_2, \dots, t_n) \in \mathbf{S}(\mathbf{P} \cup \mathbf{Q})$ if and only if $R(t_1, t_2, \dots, t_n) \in \mathbf{S}(\mathbf{P})$.

So the weak principle of stratification asserts that, if **P**, **Q** is a stratified pair of programs, **P** ∪ **Q** is a conservative extension of **P**.

From a programming point of view, one way to think of the weak principle of stratification is to consider a stratified pair **P**, **Q** of logic programs as two program modules. Think of each module as defining the relations which appear in the heads of its clauses. Thus module **P** defines certain relations, and **Q** defines other relations using the relations defined in **P**. If a semantics for logic programming obeys the weak principle of stratification, then these modules operate more or less independently: in particular, module **Q** will not alter the definitions made by **P**.

OBSERVATION 3.4

The minimal model semantics does not obey the weak principle of stratification.

Proof

Consider the programs $\mathbf{P} = \{a \leftarrow b\}$, and $\mathbf{Q} = \{c \leftarrow \neg a\}$. **P** has one minimal model: $\{\neg a, \neg b\}$. So the minimal model semantics will infer both those formulas from **P**. On the other hand, $\mathbf{P} \cup \mathbf{Q}$ has two minimal models: $\{\neg a, \neg b, c\}$ and $\{a, \neg b, \neg c\}$, so the minimal model semantics for $\mathbf{P} \cup \mathbf{Q}$ can infer only $\{\neg b\}$. □

3.2.2. Program completions: 2-valued and 3-valued

Clark proposed treating the rules explicitly as definitions. Hence an atomic formula $R(t_1, \dots, t_n)$ is true *if and only if* it is implied by one of the rules.

DEFINITION 3.6

For a propositional program **P**, for each proposition letter p occurring in **P**, the *completion* of **P** contains the formula

$$p \leftrightarrow \bigvee \{q_1 \wedge \dots \wedge q_k : p \leftarrow q_1 \wedge \dots \wedge q_k \text{ is a rule of } P\}.$$

The completion contains no other formulas.

(Note that, in general, the disjunction $\bigvee$ above may well be a disjunction over an infinite set of formulas. The disjunction of 0 formulas is defined to be the constant **false**.)

Example 3.5

The completion of $\{a \leftarrow \neg b, b \leftarrow c, b \leftarrow \neg c, d\}$ is

$$\{c \leftrightarrow \mathbf{false}, b \leftrightarrow c \vee \neg c, a \leftrightarrow \neg b, d \leftrightarrow \mathbf{true}\}$$

which has one 2-valued model, $\{\neg a, b, \neg c, d\}$.

DEFINITION 3.7

The 2-valued program completion semantics $\mathbf{PC}_2(\mathbf{P})$ of a program $\mathbf{P}$ is the set of all literals true in all 2-valued models of the program completion.

Example 3.6

The logic program $\mathbf{P} = \{p \leftarrow \neg p\}$ in classical logic is equivalent to $\{p\}$, but its completion, $\{p \leftrightarrow \neg p\}$, is inconsistent. So $\mathbf{PC}_2(\mathbf{P}) = \{p, \neg p\}$.

Fitting and Kunen [5,9] suggested a 3-valued interpretation, where the third truth value is "undefined" (denoted $\perp$), corresponding to neither a proposition letter nor its negation being in the partial interpretation.

Given a consistent partial interpretation, they use Łukasiewicz's truth tables to define the notion of a partial interpretation being a 3-valued model of the completion of a program. A conjunction is true if all the conjuncts are true, false if one conjunct is false, and undefined otherwise, disjunctions are dual. And $\neg T = F$, $\neg F = T$, and $\neg \perp = \perp$. Łukasiewicz's truth table for $\leftrightarrow$ (the operator he called $\cong$) is:

	$\leftrightarrow$	b: T	F	$\perp$
a:	T	T	F	F
	F	F	T	F
	$\perp$	F	F	T

The propositional constants **true** and **false**, of course, evaluate to T and F.

DEFINITION 3.8

The semantics $\mathbf{PC}_3(\mathbf{P})$ is the set of all literals true in all 3-valued partial models of the completion of $\mathbf{P}$.

Example 3.7

The completion of the logic program $\mathbf{Q} = \{p \leftarrow \neg p, a, b \leftarrow \neg c\}$ is

$$\{p \leftrightarrow \neg p, a \leftrightarrow \mathbf{true}, ,b \leftrightarrow \neg c, c \leftrightarrow \mathbf{false}\},$$

which is inconsistent, so $\mathbf{PC}_2(\mathbf{Q}) = \{p, \neg p, a, \neg a, b, \neg b, c, \neg c\}$. But $\mathbf{PC}_3(\mathbf{Q}) = \{a, b, \neg c\}$.

The semantics $\mathbf{PC}_3$, like all three-valued semantics discussed in this paper, has an inductive construction. For a program $\mathbf{P}$, define a knowledge extension operator $\mathbf{K_P}$ as follows: For I a set of literals,
(1) p is in $\mathbf{K_P}(I)$ if there is a rule $p \leftarrow \beta_1 \wedge \ldots \wedge \beta_k$ in $\mathbf{P}$ where $\beta_1 \wedge \ldots \wedge \beta_k$ is true in I;
(2) $\neg p$ is in $\mathbf{K_P}(I)$ if for every rule $p \leftarrow \beta_1 \wedge \ldots \wedge \beta_k$ in $\mathbf{P}$ with head p, $\beta_1 \wedge \ldots \wedge \beta_k$ is false in I. Now define by transfinite induction

$$I_{<\eta} = \bigcup_{\nu<\eta} I_\nu$$

$$I_\eta = \mathbf{K_P}(I_{<\eta}).$$

$\mathbf{PC}_3(\mathbf{P})$ is the least fixed point of the operator $\mathbf{K_P}$, i.e., the first I_η where $I_\eta = I_{<\eta}$.

THEOREM 3.5 [5]

For any logic program $\mathbf{P}$, $\mathbf{PC}_3(\mathbf{P})$ is itself a (consistent) 3-valued model of $\mathbf{P}$.

Proof (sketch)

It is straightforward to show that if a partial interpretation I is consistent, so is $\mathbf{K_P}(I)$. It then can be shown by transfinite induction, on the stages of the construction above, that the least fixed point is consistent. Finally, it is straightforward to show that a 3-valued interpretation is a fixed point of $\mathbf{K_P}$ if and only if it is a 3-valued model of $\mathbf{P}$. □

(Analogous results will hold for the other 3-valued semantics discussed in this paper.)

OBSERVATION 3.6

The 3-valued program completion semantics obeys the weak principle of stratification; the 2-valued does not.

Proof

The inductive construction of the 3-valued version makes the weak principle obvious.

We construct a counterexample for the 2-valued version from an example of Van Gelder [20]: $\mathbf{P}_1 = \{a \leftarrow \neg b,\ b \leftarrow \neg a\}$; $\mathbf{P}_2 = \{p \leftarrow \neg p,\ p \leftarrow a\}$. The completion of $\mathbf{P}_1$ is $\{a \leftrightarrow \neg b,\ b \leftrightarrow \neg a\}$ – a or b but not both - and it gives no way to choose between them. So, if the semantics satisfied the weak principle of stratification, it would infer neither a nor $\neg a$ when applied to $\mathbf{P}_1 \cup \mathbf{P}_2$.

The rule $p \leftarrow \neg p$ is classically equivalent to p, so it forces p to be true in all 2-valued models, but its completion is $p \leftrightarrow \neg p$, which is inconsistent. The completion of the two rules in P_2 is $p \leftrightarrow \neg p \vee a$, which amounts to a way to force a to be true without using a in the head of a clause. So a is inferable from the completion of $\mathbf{P}_1 \cup \mathbf{P}_2$. □

The program completion approach handles many unstratified programs nicely. One example is the well known Yale Shootout example. Game trees provide another example: Consider the program

$$\{winningPos(X) \leftarrow move(X, Y) \wedge \neg\, winningPos(Y)\}$$

describing a game where the player making the last legal move loses. To this is added a set of rules describing the positions of the game and the legal moves from position to position.

Kolaitis [8] showed that no such program is locally statifiable. But if the digraph of positions and moves between positions is finite and acyclic, it is easily shown that $\mathbf{PC}_3$ correctly identifies winning and nonwinning positions.

3.2.3. 2-valued versus 3-valued intuitions

There are significant intuitive differences between 2-valued and 3-valued semantics. The comments we make here, though made of the program completion semantics, apply to all the 2-valued and 3-valued pairs in this paper.

One approach to negation as failure comes from epistemic logic: the theory is describing, not what is true, but what is known to an intelligent observer. The knowledge extension operator $\mathbf{K}_\mathbf{P}$ described above corresponds to the inference process of the observer. An observer's knowledge tends to be partial, justifying the 3-valued approach. It can be argued that Van Gelder's example, which was the source of observation 3.6 and helped motivate the principle of stratification, illustrates a paradoxical inference in the 2-valued program completion semantics. One way to think of this is that epistemic logic, geared toward knowledge and belief, is naturally 3-valued, if not 4-valued, and an attempt to force it into a 2-valued system creates paradoxical results. The 3-valued approach allows inferring that information which seems clearly to follow according to whatever logical paradigm is used, and stopping there.

On the other hand, think of a logic program as representing causality. Then a program like $\{p \leftarrow \neg\, p,\ a \leftarrow \neg\, b\}$ seems not just strange, but incoherent – and contradictory. The 2-valued program completion semantics essentially proscribes the program. Unfortunately, it is possible to get around proscribed constructs. The seemingly "anomalous", or "paradoxical", examples produced to violate the weak principle of stratification start with these proscribed programs and modify them so that they are consistent but have consequences violating the principle.

Also recall our original statement of negation as failure:

> If it becomes *obvious* that an atomic sentence $R(t_1, t_2, \ldots, t_n)$ is not provable (in whatever logical system is chosen as a paradigm), infer that the sentence is false.

Here the rule in $\mathbf{PC}_3$ for putting $\neg\, p$ into $\mathbf{KP}(I)$ can be thought of as coming from a very conservative interpretation of the word *obvious*. A positive literal p

is inferred *only* when it follows by *modus ponens* from other literals already inferred. A negative literal $\neg p$ is inferred *only* when all possible ways to derive p have been rejected - have had their hypotheses inferred false. The semantics $\mathbf{PC}_2$ can thus be thought of as more radical, since it permits other inferences, as in observation 3.6.

3.2.4. *The stable and well-founded semantics*

The stable and well-founded explicitly enforce another paradigm, analogous to observation 3.3:

> If an atomic sentence is not asserted directly (as in $p \leftarrow \mathbf{true}$), the only way to derive it is to prove it, using program rules and *modus ponens* alone, from the *negated atomic sentences* already established and the atomic sentences which are asserted directly.

The following definitions of the stable and well-founded semantics are not the original definitions. For the stable semantics, the equivalence of the definition here to the original definition is easy to show using the Gelfond–Lifschitz transformation of a logic program [7]. The definition here of the well-founded semantics is a small modification of a definition worked out in a discussion with Allen van Gelder, and is inspired by [2,14,15,19].

Example 3.8

Let $\mathbf{P}$ be the program

$$\{a \leftarrow b,\ b \leftarrow a,\ c \leftarrow d,\ d \leftarrow c,\ a \leftarrow \neg c\}.$$

The program completion is

$$\{a \leftrightarrow b \vee \neg c,\ b \leftrightarrow a,\ c \leftrightarrow d,\ d \leftrightarrow c\}.$$

But there is no way to derive c from *negative literals*. On the other hand, a can be derived from $\neg c$, and b can be derived from a, which can be derived from $\neg c$.

DEFINITION 3.9

For $\mathbf{P}$ a logic program, a *transitive rule* of $\mathbf{P}$ is a rule $\alpha \overset{*}{\leftarrow} \beta_1 \wedge \ldots \wedge \beta_k$ where α can be derived from $\beta_1 \wedge \ldots \wedge \beta_k$ by 0 or more (but a finite number of) applications of *modus ponens* and 0 or more of rules of $\mathbf{P}$.

As before, in the special case where a transitive rule has empty body, we treat its body as being the propositional constant **true**.

For a propositional program $\mathbf{P}$, for each proposition letter p occurring in $\mathbf{P}$, the *stable completion* of $\mathbf{P}$ contains the formula

$$p \leftrightarrow \bigvee \left\{ \neg q_1 \wedge \ldots \wedge \neg q_k : p \overset{*}{\leftarrow} \neg q_1 \wedge \ldots \wedge \neg q_k \text{ is a transitive rule of } \mathbf{P} \right\}.$$

The stable completion contains no other formulas.

CONVENTION

If **P** has a transitive rule such as $p \overset{*}{\leftarrow} a \wedge b$, it will also have a transitive rule $p \overset{*}{\leftarrow} a \wedge b \wedge \gamma$ for any literal γ: the derivation of p just does not happen to use γ. So a stable completion would be of the form $p \leftrightarrow (a \wedge b) \vee (a \wedge b \wedge \gamma) \vee \ldots$. When we write the stable completion, we shall include only disjuncts, such as $a \wedge b$, which have minimal sets of conjuncts.

Example 3.9

The stable completion of

$$\{a \leftarrow b,\ b \leftarrow a,\ c \leftarrow d,\ d \leftarrow c,\ a \leftarrow \neg c,\ e,\ e \leftarrow a \wedge b,\ f \leftarrow \neg e\}$$

is

$$\{a \leftrightarrow \neg c,\ b \leftrightarrow \neg c,\ c \leftrightarrow \textbf{false},\ d \leftrightarrow \textbf{false},\ e \leftrightarrow \textbf{true} \vee \neg c,\ f \leftrightarrow \neg e\}.$$

DEFINITION 3.10

A *stable model* of a program **P** is a model of the stable completion. If there is a unique stable model of this "completion", the stable semantics infers that.

We shall define here the stable semantics **ST(P)** of a program **P** to be the set of literals true in all models of the stable completion. (So if there is no stable model, the stable semantics infers inconsistent information.)

Hence the stable semantics for the example above infers $\{a, b, \neg c, \neg d, e, \neg f\}$.

DEFINITION 3.11

The well-founded semantics **WF(P)** of a logic program **P** is the set of all literals true in all 3-valued models of the stable completion of **P** (where, again, Łukasiewicz's truth table is used for $\leftrightarrow$).

In the example above, it makes the same inferences that the stable semantics does. But in general it makes fewer inferences.

Like the 3-valued program completion semantics, the well-founded semantics has an inductive definition with a knowledge-extension operator. And just as with Fitting's work, this can be used to prove that:

THEOREM 3.7 (VAN GELDER–ROSS–SCHLIPF)

For any logic program **P**, **WF(P)** is itself a (consistent) 3-vaued model of **P**.

The stable semantics is due to Gelfond and Lifschitz [6,7]. The well-founded semantics is due to Van Gelder–Ross–Schlipf [20]. Przymusinski first noticed that they were just 2-valued and 3-valued versions of the same semantics, though our explanation with the stable completion is quite different from his.

OBSERVATION 3.8

The well-founded semantics obeys the weak principle of stratification. The stable semantics does not.

Proof

The proof is the same as for observation 3.6. □

A major strength of the well-founded semantics is that, of all the semantics discussed in this paper, it is the strongest semantics which can be constructed in polynomial time (as a function of the size of a finite propositional program), assuming $\mathscr{P} \neg = \mathscr{NP}$. (That it is polynomial time is proved in [20]; that the stable semantics is co-$\mathscr{NP}$-complete is proved in [12].) When fixed, finite, first order logic programs are used to deduce information from finite databases, these results translate to: answering queries based upon the well-founded semantics is polynomial time in the size of the database; answering queries based upon the stable semantics is co-$\mathscr{NP}$-complete.

4. Goals for a "common sense" semantics

As we stated earlier, our aim here is to attempt to understand what "common sense" negation as failure reasoning is and to see to what extent that can be formalized. We take the point of view that all the semantics discussed above have a certain "common sense" justification, and that the aim of a negation as failure semantics is to reconcile, as far as is possible in a reasonable formalism, all the underlying motives. We suggest the following as goals of a "common sense" semantics:

(1) *Pure declarativeness*: If program $\mathbf{P}_1$ is formed from $\mathbf{P}$ by rearranging the rules in $\mathbf{P}$, by rearranging the subgoals (hypotheses) within a rule of $\mathbf{P}$, or by performing some alphabetic variation such as substituting one name for another throughout the program, then the semantics should infer the same sentences from the two (up to the alphabetic variation). This goal rejects some of the procedural features of PROLOG.

(2) The motivation for these goals is that semantics should extend classical logic, not replace it.

(a) *Non-contradiction*: The semantics can be applied to any program; no program is rejected as being meaningless. Furthermore, the set of sentences inferred by the program, together with the program itself, should be consistent in classical logic (with the classical interpretation of $\leftarrow$).

(b) *Classical completeness*: If an atomic sentence a holds in all models of $\mathbf{P}$ (in classical logic), then the semantics should infer a from $\mathbf{P}$.

   (c) *Factoring into cases*: If from a program the semantics can infer a sentence p assuming a and also infer p assuming $\neg a$, it can infer p. Similarly, if it can infer p assuming each of the 2^n truth assignment to $a_1, a_2, \ldots, a_n$, then it can infer p.

(3) The motivation for these goals is that the minimal model semantics makes correct conclusions, but it does not make enough.
   (a) *Minimality*: If I is the set of inferences made from program $\mathbf{P}$, then there should be a minimal model $\mathcal{M}$ of P where each formula in I is true in $\mathcal{M}$.
   (b) *GCWH*: If an atomic sentence $R(t_1, t_2, \ldots, t_n)$ is false in all minimal models of a program $\mathbf{P}$, then $\neg R(t_1, t_2, \ldots, t_n)$ is inferable.

(4) The motivation for these goals is that certain semantics, though not strong enough in general, seem always to make reasonable inferences.
   (a) *Extending Van Emden–Kowalski*: The semantics should agree with the Van Emden–Kowalski semantics on Horn clause programs.
   (b) *Extending program completion*: The semantics should extend the 3-valued program completion semantics.
   The justification for this goal is that the 3-valued program completion semantics, for example in its handling of winning games and of the Yale Shootout example, seems to capture a very significant notion of negation as failure.

(5) *Principle of stratification*: The semantics should obey the weak principle of stratification.

We have argued that all the goals above are in some sense "common-sense" goals for logic programming. Accordingly, it would, from a "common-sense" point of view, be desirable to achieve them all.

Table 1
Showing how previous semantics meet these goals [a]

Goal:	Minimal model	Strati- fied	2-valued completion	3-valued completion	Stable	Well- founded
Pure declarativeness	Yes	Yes	Yes	Yes	Yes	Yes
Non-contradiction	Yes	No	No	Yes	No	Yes
Classical completeness	Yes	*	Yes	No	Yes	No
Factoring into cases	Yes	*	†	No	†	No
Minimality	Yes	*	*	Yes	*	Yes
GCWH	Yes	*	No	No	†	No
Extend V. Emden–Kowal.	Yes	Yes	No	No	Yes	Yes
Extend program compl.	No	*	Yes	Yes	Yes	Yes
Princ. stratification	No	*	No	Yes	No	Yes

[a] * true *if* the semantics is defined on the program and gives non-contradictory results.
† the desired literal α is inferred, but $\neg \alpha$ may be also.

We justify these goals as satisfying some sort of "common sense" criteria. We make no attempt here to ask whether it makes good deep philosophical or linguistic sense to try to achieve all these goals at once. Our question here is merely whether there is any reasonable formalism that achieves these goals.

It is easy to construct an *ad hoc* semantics that satisfies all these goals. We do not know whether any non-*ad hoc* semantics satisfies them all. In the remainder of the paper we show that it is possible to satisfy all except the GCWH goal.

5. A logical paradigm permitting argument by cases

Recall our first intuitive characterization of negation as failure:

> If it becomes *obvious* that an atomic sentence $R(t_1, t_2, \dots, t_n)$ is not provable (*in whatever logical system is chosen as a paradigm*), infer that the sentence is false.

Our approach to generalizing previous semantics is, not to change the construction, but again to change the paradigm logic used for deriving positive literals.

Consider the "weaknesses" of the well-founded semantics. First, it does not make all inferences classical logic would. Second, it does not make all the inferences GCWH does. And third, it does not allow argument by factoring into cases. Now if we repeat the inductive construction of the well-founded semantics but replace derivation by *modus ponens* only with full classical logic, we get inference under GCWH (by observation 3.3), and we no longer meet some of the goals met by the well-founded semantics. So it is natural to attempt to find a logical paradigm for negation as failure that is stronger than argument by *modus ponens* alone but weaker than full classical logic. Is it possible to do this in such a way as to meet most of the "common sense" goals proposed above? It turns out it is at least possible to do so and meet all the goals except GCWH. The extension to the *modus ponens*-only paradigm is just strong enough explicitly to allow arguments by cases.

Example 5.1

It is tempting to try to infer and store information about possible cases. This turns out to be quite risky. For example, consider the program **P**

$$\{a \leftarrow \neg b,\ b \leftarrow \neg c,\ c \leftarrow \neg a,\ s \leftarrow a \wedge b,\ s \leftarrow b \wedge c,\ s \leftarrow c \wedge a\}.$$

First note that s holds in all (classical) models of **P**. From the first three rules it is tempting to infer, as in classical logic, $a \vee b$, $b \vee c$, and $c \vee a$. Now recall the intuition of the well-founded semantics: that positive literals can only be derived from known negative literals. The transitive rules with head s and with negative subgoals are (discarding unnecessary subgoals)

$$s \overset{*}{\leftarrow} \neg b \wedge \neg c,\ s \overset{*}{\leftarrow} \neg c \wedge \neg a,\ s \overset{*}{\leftarrow} \neg a \wedge \neg b.$$

But the inferences of $a \vee b$, etc., above contradict the bodies of all those rules. Hence an obvious generalization of the well-founded semantics would infer $\neg s$, violating the goal of classical consistency.

DEFINITION 5.1

A *derived rule* of $\mathbf{P}$ is a rule

$$a \Leftarrow \beta_1 \wedge \ldots \wedge \beta_n,$$

where, for some proposition letters $d_1, \ldots, d_k$ and for each of the 2^k ways to choose each $\delta_i = d_i$ or $\neg d_i$,

$$a \overset{*}{\leftarrow} \beta_1 \wedge \ldots \wedge \beta_n \wedge \delta_1 \wedge \ldots \wedge \delta_k$$

is a transitive rule of $\mathbf{P}$ (with a possibly non-minimal set of subgoals).

Example 5.2

(1) Let $\mathbf{P} = \{p \leftarrow \neg p\}$. Then $p \overset{*}{\leftarrow} \neg p$ (in 1 application of *modus ponens*), and $p \overset{*}{\leftarrow} p$ (in 0 applications of *modus ponens*, so $p \Leftarrow$ is a derived rule of $\mathbf{P}$.

(2) Let $\mathbf{P} = \{a \leftarrow \neg b, b \leftarrow \neg a, c \leftarrow a, c \leftarrow b\}$. Then $c \overset{*}{\leftarrow} a$ (in one step), and $c \overset{*}{\leftarrow} \neg a$ (in two steps), so $c \Leftarrow$ is a derived rule of $\mathbf{P}$.

THEOREM 5.1

(*Completeness theorem for literals*) Let $\mathbf{P}$ be a logic program and let a be proposition letter occurring in $\mathbf{P}$. Then a holds in all models of $\mathbf{P}$ (i.e., a is a classical consequence of $\mathbf{P}$) if and only if $a \Leftarrow$ is a derived rule of $\mathbf{P}$.

Proof

It is easy to show that if $a \Leftarrow$ is a derived rule of $\mathbf{P}$ then a holds in all models of $\mathbf{P}$, so prove the converse. Assume that $a \Leftarrow$ is not a derived rule of $\mathbf{P}$, and construct a model of $\mathbf{P}$ where a is false.

Enumerate the proposition letters of $\mathbf{P}$: $a_0, a_1, a_2, \ldots$, with $a = a_0$. For each natural number i, we shall set α_i to be either a_i or $\neg a_i$ below. The final interpretation will be $I = \{\alpha_0, \alpha_1, \ldots, \alpha_n, \ldots\}$.

Set $\alpha_0 = \neg a_0$. Note that $a_0 \Leftarrow \alpha_0$ is *not* a derived rule of $\mathbf{P}$: if it were, since $a_0 \overset{*}{\leftarrow} a_0$ is a trivial transitive rule of $\mathbf{P}$, $a_0 \Leftarrow$ would be a derived rule of $\mathbf{P}$.

To define α_{i+1}: By inductive hypothesis, $a_0 \Leftarrow \alpha_0 \wedge \alpha_1 \wedge \ldots \alpha_i$ is not a derived rule of $\mathbf{P}$. Now if both $a_0 \Leftarrow \alpha_0 \wedge \alpha_1 \wedge \ldots \alpha_i \wedge a_{i+1}$ and $a_0 \Leftarrow \alpha_0 \wedge \alpha_1 \wedge \ldots \alpha_i \wedge \neg a_{i+1}$ were derived rules of $\mathbf{P}$, then $a_0 \Leftarrow \alpha_0 \wedge \alpha_1 \wedge \ldots \alpha_i$ would be a derived rule also. Hence at least one of the latter is not. Set

$$\alpha_{i+1} = \begin{cases} \neg a_{i+1} & \text{if } a_0 \Leftarrow \alpha_0 \wedge \alpha_1 \wedge \ldots \alpha_i \wedge a_{i+1} \text{ is a derived rule of } \mathbf{P}, \\ a_{i+1} & \text{otherwise.} \end{cases}$$

By construction, there is no derived rule $a_0 \Leftarrow \alpha_1 \wedge \ldots \wedge \alpha_n$ of $\mathbf{P}$.

Now show that if there is a derived rule

$$a_i \Leftarrow \alpha_1 \wedge \ldots \wedge \alpha_n, \qquad (1)$$

of **P** then $\alpha_i = a_i$. For suppose not. Since the construction picked $\alpha_i = \neg a_i$,

$$a_0 \Leftarrow \alpha_0 \wedge \alpha_1 \wedge \ldots \alpha_i \wedge a_{i+1} \qquad (2)$$

is a derived rule of **P**. Composing the derived rules 1 and 2 gives a derived rule

$$a_0 \Leftarrow \alpha_0 \wedge \alpha_1 \wedge \ldots \alpha_k, \qquad (3)$$

where k is the maximum of i and n.

Since I satisfies all the derived rules of **P**, and hence all the rules of **P**, I is a model of **P**. □

COROLLARY 5.2

(*Extended completeness for literals*) For any logic program **P**, any set I of *positive* literals, and any proposition letter a, a holds in all models of $\mathbf{P} \cup I$ (i.e., a is a classical consequence of $\mathbf{P} \cup I$) if and only if, for some $b_1, \ldots, b_n \in I$, $a \Leftarrow b_1 \wedge \ldots \wedge b_n$ is a derived rule of **P**.

Observe, however, that the corollary fails if I is allowed to contain negative literals. For example, for $\mathbf{P} = \{b \leftarrow \neg a\}$ and $I = \{\neg b\}$, a holds in all (classical) models of **P**. But since there is no rule with head a, every derived rule with head a must contain a in its body.

5.1. THE STABLE-BY-CASE SEMANTICS

We now have another logical paradigm: a positive literal can be proved only by applying *modus ponens* to derived rules a program and negative literals already established.

DEFINITION 5.2

For a propositional program **P**, for each proposition letter p occurring in **P**, the *stable-by-case completion* of **P** contains the formula

$$p \leftrightarrow \bigvee \{\neg q_1 \wedge \ldots \wedge \neg q_k : p \Leftarrow \neg q_1 \wedge \ldots \wedge \neg q_k \text{ is a derived rule of } \mathbf{P}\}.$$

The stable-by-case completion contains no other formulas.

DEFINITION 5.3

A 2-valued model $\mathcal{M}$ for a logic program **P** is *stable-by-case* if it is a model of the stable-by-case completion of **P**.

The *stable-by-case* semantics $\mathbf{ST}_{bc}(\mathbf{P})$ of a program **P** is the set of all literals true in all stable-by-case models of **P**.

The stable-by-case semantics eliminates some of the “paradoxes” of the stable semantics.

Example 5.3

Let **P** be Van Gelder's program $\{a \leftarrow \neg b, b \leftarrow \neg a, p \leftarrow \neg p, p \leftarrow a\}$. The stable-by-case completion is [4]

$$\{p \Leftarrow \mathbf{true} \vee a,\ a \Leftarrow \neg b,\ b \Leftarrow \neg a\}.$$

The first formula is enough to derive p, and the "paradoxical" behavior of the example is lost. Both $\{a, \neg b, p\}$ and $\{\neg a, b, p\}$ are stable-by-case.

But the stable-by-case semantics still has some of what we have been considering "paradoxical" behavior:

Example 5.4

Let $\mathbf{Q} = \{a \leftarrow \neg b, b \leftarrow \neg c, c \leftarrow \neg a\}$. The stable-by-case completion is $\{a \leftrightarrow \neg b, b \leftrightarrow \neg c, c \leftrightarrow \neg a\}$, which is inconsistent. So $\mathbf{ST}_{bc}(\mathbf{Q}) = \{a, \neg a, b, \neg b, c, \neg c\}$.

Example 5.5

Let $\mathbf{R} = \{a \leftarrow \neg b, b \leftarrow \neg c, c \leftarrow \neg a, s \leftarrow a \wedge b, s \leftarrow c \wedge a\}$. The stable-by-case completion is inconsistent. So $\mathbf{ST}_{bc}(\mathbf{R}) = \{a, \neg a, b, \neg b, c, \neg c, s, \neg s\}$.

THEOREM 5.3

Let **P** be a logic program.

(1) If I is 2-valued, I is a model of **P** if and only if I satisfies all the derived rules of **P**.

(2) Every stable model of **P** is a stable-by-case model of **P**.

(3) $\mathbf{ST}_{bc}(\mathbf{P}) \subseteq \mathbf{ST}(\mathbf{P})$.

OBSERVATION 5.4

The stable-by-case semantics does not satisfy the weak principle of stratification.

Proof

Let $\mathbf{P} = \{p \leftarrow \neg q, q \leftarrow \neg p\}$. Let $\mathbf{Q} = \{a \leftarrow p \wedge \neg b, b \leftarrow p \wedge \neg c, c \leftarrow p \wedge \neg a\}$. Then the pair **P**, **Q** is stratified. **P** has two stable-by-case models: $\{p, \neg q\}$ and $\{q, \neg p\}$. So $\mathbf{ST}_{bc}(\mathbf{P}) = \emptyset$. But $\mathbf{P} \cup \mathbf{Q}$ has just one stable-by-case model: $\{\neg p, q, \neg a, \neg b, \neg c\}$. □

[4] Or is equivalent to – recall our convention on dropping extra conjuncts from the formulas.

5.2. THE WELL-FOUNDED-BY-CASE SEMANTICS

DEFINITION 5.4

The *well-founded-by-case* semantics, $\mathbf{WF}_{bc}(\mathbf{P})$, of a program $\mathbf{P}$ is the set of all literals true in all 3-valued models of the stable-by-case completion of $\mathbf{P}$.

Look again at the same examples used to introduce the stable-by-case semantics:

Example 5.6

Again let $\mathbf{P}$ be Van Gelder's program $\{a \leftarrow \neg b,\ b \leftarrow \neg a,\ p \leftarrow \neg p,\ p \leftarrow a\}$. The stable-by-case completion is

$$\{p \leftrightarrow \mathbf{true} \vee a,\ a \leftrightarrow \neg b,\ b \leftrightarrow \neg a\}.$$

And $\mathbf{WF}_{bc}(\mathbf{P}) = \mathbf{ST}_{bc}(\mathbf{P}) = \{p\}$.

Example 5.7

Let $\mathbf{Q} = \{a \leftarrow \neg b,\ b \leftarrow \neg c,\ c \leftarrow \neg a\}$. The stable-by-case completion is

$$\{a \leftrightarrow b,\ b \leftrightarrow c,\ c \leftrightarrow a\}.$$

$\mathbf{WF}_{bc}(\mathbf{Q}) = \emptyset$.

Example 5.8

Let $\mathbf{R} = \{a \leftarrow \neg b,\ b \leftarrow \neg c,\ c \leftarrow \neg a,\ s \leftarrow a \wedge b,\ s \leftarrow b \wedge c,\ s \leftarrow c \wedge a\}$. The stable-by-case completion is inconsistent, but the well-founded-by-case semantics captures the fact that s is logical consequence of $\mathbf{R}$. So $\mathbf{WF}_{bc}(\mathbf{R}) = \{s\}$. By contrast, $\mathbf{WF}(\mathbf{R}) = \emptyset$.

The inductive construction of $\mathbf{WF}_{bc}(\mathbf{P})$ is analogous to the inductive construction of $\mathbf{PC}_3(\mathbf{P})$.

THEOREM 5.5

For every logic program $\mathbf{P}$, $\mathbf{WF}_{bc}(\mathbf{P})$ is a (consistent) 3-valued model of the stable-by-case completion of $\mathbf{P}$.

Proof

The proof is analogous to the proof of theorem 3.5. □

THEOREM 5.6

For every logic program $\mathbf{P}$, there is a 2-valued model I of $\mathbf{P}$ where $\mathbf{WF}_{bc}(\mathbf{P}) \subseteq I$.

Proof

Let $I = \mathbf{WF}_{bc}(\mathbf{P}) \cup \{p\colon \neg p \notin \mathbf{WF}_{bc}(\mathbf{P})\}$. Suppose I is not a model of $\mathbf{P}$. Then there is a rule

$$a \leftarrow b_1 \wedge \ldots \wedge b_k \wedge \neg c_1 \wedge \ldots \wedge \neg c_n \qquad (4)$$

of **P** where all the subgoals are in I but $\neg a \in I$. In particular, since each $b_i \in I$, no $\neg b_i \in \mathbf{WF}_{bc}(\mathbf{P})$. So for each b_i there is a derived rule

$$b_i \Leftarrow \neg d_1^i \wedge \ldots \wedge \neg d_{m_i}^i, \tag{5}$$

where no $d_{m_i}^i \in \mathbf{WF}_{bc}(\mathbf{P})$. Furthermore, since $\mathbf{WF}_{bc}(\mathbf{P})$ is consistent, I is consistent, so no $c_i \in \mathbf{WF}_{bc}(\mathbf{P})$. Now substitute each derived rule 5 into rule 4 to get a derived rule with head a and body a set of negative literals $\neg d_j^i$ and $\neg c_k$ where no d_j^i or c_k is in $\mathbf{WF}_{bc}(\mathbf{P})$. But then $\neg a \notin \mathbf{WF}_{bc}(\mathbf{P})$, contradicting the assumption. □

COROLLARY 5.7

The well-founded-by-case semantics satisfies the goals of non-contradiction and minimality.

Proof

The existence of the model I above proves non-contradiction. By theorem 3.2, there is a minimal model $J \subseteq I$. By theorem 5.3, J satisfies all derived rules of **P**. Every positive literal in $\mathbf{WF}_{bc}(\mathbf{P})$ is derivable, via some derived rule, from negative literals in $\mathbf{WF}_{bc}(\mathbf{P})$. Hence every positive literal in $\mathbf{WF}_{bc}(\mathbf{P})$ must also be in J. Thus J is a minimal model as desired. □

THEOREM 5.8

If a positive literal a holds in every model of $\mathbf{P} \cup \mathbf{WF}_{bc}(\mathbf{P})$, then $a \in \mathbf{WF}_{bc}(\mathbf{P})$.

The relationship between 3-valued models of the stable completion and 3-valued models of the stable-by-case completion is a bit more tricky than with 2-valued models. For the logic program $\{p \leftarrow \neg p\}$, $\emptyset$ is a model of the stable completion but not of the stable-by-case completion, and $\{p\}$ is a model of the stable-by-case completion but not the stable completion. However, for the *least* fixed points, the behavior is reasonable.

THEOREM 5.9

For any logic program **P**, $\mathbf{WF}(\mathbf{P}) \subseteq \mathbf{WF}_{bc}(\mathbf{P})$.

Proof

Consider the inductive construction of the well-founded semantics, analogous to the inductive construction of the 3-valued program completion semantics: For J a set of literals,

$$\begin{aligned}\mathbf{W}_{\mathbf{P}}(J) = \Big\{p\text{: there is a transitive rule } p \overset{*}{\leftarrow} \neg c_1 \wedge \ldots \wedge \neg c_k \\ \text{of } \mathbf{P} \text{ where } \neg c_1, \ldots \neg c_k \in J\Big\} \\ \cup \Big\{\neg p\text{: for every transitive rule } p \overset{*}{\leftarrow} \neg c_1 \wedge \ldots \wedge \neg c_k \text{ of } \mathbf{P}, \\ \text{some } c_i \in J\Big\}.\end{aligned}$$

Then define by transfinite recursion $J_{<\eta} = \bigcup_{\mu<\eta} J_\mu$, and $J_\eta = \mathbf{WP}(J_{<\eta})$. **WF(P)** is the first J_η where $J_\eta = J_{<\eta}$.

Prove by transfinite induction that $J_\eta \subseteq \mathbf{WF}_{bc}(\mathbf{P})$. Assume that $J_{<\eta} \subseteq \mathbf{WF}_{bc}(\mathbf{P})$. It is easy to show that every positive literal in J_η is in $\mathbf{WF}_{bc}(\mathbf{P})$. So suppose $\neg p \in J_\eta$ but $\neg p \notin \mathbf{WF}_{bc}(\mathbf{P})$. Then there is a derived rule

$$p \Leftarrow \neg c_1 \wedge \ldots \wedge \neg c_n$$

of **P** where no $c_i \in \mathbf{WF}_{bc}(\mathrm{P})$. In particular, no $c_i \in J_{<\eta}$. So there are proposition letters $d_1, \ldots, d_k$ such that, for every way to choose δ_i equal to d_i or $\neg d_i$, $i = 1, \ldots, n$,

$$p \overset{*}{\leftarrow} \neg c_1 \wedge \ldots \wedge \neg c_n \wedge \delta_1 \wedge \ldots \delta_k \tag{6}$$

is an iterated rule of **P**.

Now for each d_i, choose $\delta_i = d_i$ if $d_i \in J_{<\eta}$, and $\delta_i = \neg d_i$ otherwise. By construction of $J_{<\eta}$, for each d_i where $\delta_i = d_i$ there is an iterated rule

$$d_i \overset{*}{\leftarrow} \neg e_1^i \wedge \ldots \wedge \neg e_{h_i}^i \tag{7}$$

of **P** where each $\neg e_j^i \in J_{<\eta}$, and hence, by the consistency of the well-founded semantics, $e_j^i \notin J_{<\eta}$. Composing iterated rule 6 with the iterated rule 7 for each positive δ_i, we get an iterated rule with head p, all negative subgoals, and no subgoal false in $J_{<\eta}$. Hence $\neg p$ cannot be in J_η after all. □

Since the well-founded-by-case semantics extends the well-founded semantics, it also extends the 3-valued program completion semantics and the Van Emden–Kowalski semantics.

OBSERVATION 5.10

The well-founded-by-case semantics obeys the weak principle of stratification.

Proof

Again, this follows easily from the inductive constructuion of the semantics. □

The main weakness, from our "common sense" point of view, of the well-founded-by-case semantics, is that it does not obey the GCWH goal.

THEOREM 5.11

There is a program **P** and a proposition letter s occurring in **P** where $\neg s$ holds in every minimal model of **P** but $\neg s$ is not inferred by the well-founded-by-case semantics.

Proof

Let $\mathbf{P} = \{a \leftarrow \neg b,\ b \leftarrow \neg c,\ c \leftarrow \neg a,\ s \leftarrow a \wedge b \wedge c\}$. The stable-by-case completion of **P** is

$$\{a \leftrightarrow \neg b,\ b \leftrightarrow \neg c,\ c \leftrightarrow \neg a,\ s \leftrightarrow \neg b \wedge \neg c \wedge \neg a\}.$$

The well-founded-by-case semantics can make no inferences about a, b, and c, so it cannot infer $\neg s$. $\square$

Another weakness of the well-founded-by-case semantics is that even deciding whether $s \Leftarrow$ is a derived rule of a program $\mathbf{P}$ is co-$\mathcal{NP}$-hard. But with our goals of extending classical logic, this is to be expected. This merely says that any attempt to use the semantics practically – on any nontrivial scale – would have to use some approximation.

6. Conclusion

We have presented a group of "common sense" goals for a negation as failure semantics for logic programming. This "common sense" approach has the goal of, to the greatest extent possible, describing ordinary "common sense" inference, not proscribing forms of inference. Description of "common sense reasoning" was, after all, one of the sources of negation as failure in the first place. We do not know whether these goals are philosophically correct, but they do have a good deal of plausibility, and we think the attempt to isolate and satisfy "common sense" goals must be pursued.

The well-founded-by-case semantics, with its mixture of constructive and classical constructs, seems to meet these goals better than any other currently known semantics. It extends current logic programming intuitions, and it seems to have no "paradoxical" conclusions. Whether it is possible to meet all the goals listed, including the GCWH goal, should be a matter for further development.

Table 2
Showing how the new semantics meet the "common-sense" goals [a]

Goal:	Stable-by-case	Well-founded-by-case
Pure declarativeness	Yes	Yes
Non-contradiction	No	Yes
Classical completeness	Yes	Yes
Factoring into cases	†	Yes
Minimality	*	Yes
GCWH	†	No
Extend Van Emden–Kowalski	Yes	Yes
Extend program completion	Yes	Yes
Principle of stratification	No	Yes

[a] * true *if* the semantics is defined on the program and gives non-contradictory results.
† The desired literal α is inferred, but $\neg\alpha$ may be also.

References

[1] K.R. Apt, H. Blair and A. Walker, Towards a theory of declarative knowledge, in: *Foundations of Deductive Databases and Logic Programming*, ed. J. Minker (Morgan Kaufmann, Los Altos, CA, 1988).

[2] F. Bry, Logic programming as constructivism: a formalization and its application to databases, in: *8th ACM Symp. on Principles of Database Systems* (1989) pp. 34–50.

[3] A. Chandra and D. Harel, Horn clause queries and generalizations, J. ACM 29 (1982) 841–862.

[4] K.L. Clark, Negation as failure, in: *Logic and Databases*, eds. Gallaire and Minker (Plenum Press, New York, 1978) pp. 293–322.

[5] M. Fitting, A Kripke–Kleene semantics for logic programs, J. Logic Progr. 2 (1985) 295–312.

[6] M. Gelfond, On stratified autoepistemic theories, in: *Proc. AAAI* (1987).

[7] M. Gelfond and V. Lifschitz, The stable model semantics for logic programming, *Proc. 5th Int. Conf. Symp. on Logic Programming* (1988).

[8] P. Kolaitis and C. Papadimitriou, Why not negation by fixpoint?, in: *Proc. 7th Symp. on Principles of Database Systems* (1988).

[9] K. Kunen, Negation in logic programming, J. Logic Progr. 4 (1987) 289–308.

[10] K. Kunen, Some remarks on the completed database, Technical Report 775, Univ. of Wisconsin, Madison, WI 53706.

[11] V. Lifschitz, On the declarative semantics of logic programs with negation, in: *Foundations of Deductive Databases and Logic Programming*, ed. J. Minker (Morgan Kaufmann, Los Altos, CA, 1988) pp. 177–192.

[12] W. Marek and M. Truszczynski, Autoepistemic logic, J. ACM 38 (1991) 588–619.

[13] T.C. Przymusinski, On the declarative semantics of deductive databases and logic programs, in: *Foundations of Deductive Databases and Logic Programming*, ed. J. Minker (Morgan Kaufmann, Los Altos, CA, 1988) pp. 193–216.

[14] T.C. Przymusinski, Every logic program has a natural stratification and an iterated fixed point model, *8th ACM Symp. on Principles of Database Systems* (1989) pp. 11–21.

[15] K.A. Ross, A procedural semantics for well-founded negation in logic programs, in: *8th ACM Symp. on Principles of Database Systems* (1989) pp. 22–33.

[16] J.C. Shepherdson, Negation as failure, II, J. Logic Progr. 2 (1985) 185–202.

[17] M.H. Van Emden and R.A. Kowalski, The semantics of predicate logic as a programming language, J. ACM 23 (1976) 733–742.

[18] A. Van Gelder, Negation as failure using tight derivations for general logic programs, in: *Proc. 3rd IEEE Symp. on Logic Programming*, Salt Lake City, Utah (Springer, New York, 1986).

[19] A. Van Gelder, The alternating fixpoint of logic programs with negation, in: *8th ACM Symp. on Principles of Database Systems* (1989) pp. 1–10. Available from UC Santa Cruz as UCSC-CRL-88-17.

[20] A. Van Gelder, K.A. Ross and J.S. Schlipf, The well-founded semantics for general logic programs, J. ACM 38 (1991) 620–650.