Logic Programming: From Underspecification to Undefinedness

Lee Naish, Harald Søndergaard and Benjamin Horsfall
Department of Computing and Information Systems
The University of Melbourne, Victoria 3010, Australia
{lee,harald,brho}@unimelb.edu.au

Abstract

The semantics of logic programs was originally described in terms of two-valued logic. Soon, however, it was realised that three-valued logic had some natural advantages, as it provides distinct values not only for truth and falsehood, but also for "undefined". The three-valued semantics proposed by Fitting and by Kunen are closely related to what is computed by a logic program, the third truth value being associated with non-termination. A different three-valued semantics, proposed by Naish, shared much with those of Fitting and Kunen but incorporated allowances for programmer intent, the third truth value being associated with underspecification. Naish used an (apparently) novel "arrow" operator to relate the intended meaning of left and right sides of predicate definitions. In this paper we suggest that the additional truth values of Fitting/Kunen and Naish are best viewed as duals. We use Fitting's later four-valued approach to unify the two three-valued approaches. The additional truth value has very little affect on the Fitting three-valued semantics, though it can be useful when finding approximations to this semantics for program analysis. For the Naish semantics, the extra truth value allows intended interpretations to be more expressive, allowing us to verify and debug a larger class of programs. We also explain that the "arrow" operator of Naish (and our four-valued extension) is essentially the information ordering. This sheds new light on the relationships between specifications and programs, and successive executions states of a program.

1 Introduction

Logic programming is an important paradigm. Computers can be seen as machines which manipulate meaningful symbols and the branch of mathematics which is most aligned with manipulating meaningful symbols is logic. This paper is part of a long line of research on what are good choices of logic to use with a "pure" subset of the Prolog programming language. We ignore the "non-logical" aspects of Prolog such as cut and built-ins which can produce side-effects, and assume a sound form of negation (ensuring in some way that negated literals are always ground before being called).

There are several ways in which having a well-defined semantics for programs is helpful. First, it can be helpful for implementing a language (writing a compiler, for example) — it forms a specification for answering "what should this program compute". Second, it can be helpful for writing program analysis and transformation tools. Third, it can be helpful for verification and debugging — it can allow application programmers to answer "does this program compute what I intend" and, when the answer is negative, "why not". There is typically imprecision involved in all three cases.

1. Many languages allow some latitude to the implementor in ways that affect observable behaviour of the program, for example by not specifying the order sub-expression evaluation (C is an example). Even in pure Prolog, typical approaches to semantics do not precisely deal with infinite loops and/or "floundering" (when a negative literal never becomes ground). Such imprecision is not necessarily a good thing, but there is often a trade-off between precision and simplicity of the semantics.

2. Program analysis tools must provide imprecise information in general if they are guaranteed to terminate, since the properties they seek to establish are almost always undecidable.

3. Programmers are often only interested in how their code behaves for some class of inputs. For other inputs they either do not know or do not care (this is in addition to the first point). Moreover, it is often convenient for programmers to reason about *partial* correctness, setting aside the issue of termination.

A primary aim of this paper is to reconcile two different uses of many-valued logic for understanding logic programs. The first use is for the provision of semantic definition, with the purpose of answering "what should this program compute?" The other use is in connection with program specification and debugging, concerned with answering "does this program compute what I intend" and similar questions involving programmer intent. Our main contributions are:

- We show how Belnap's four-valued logic enables a clean distinction between a formula/query which is undefined, or non-denoting, and one which is irrelevant, or inadmissible.
- We use this logic to provide a denotational semantics for logic programs which is designed to help a programmer reason about partial correctness in a natural way. This aim is different to the semanticist's traditional objective of reflecting runtime behaviour, or aligning denotational and operational semantics.
- We show how four-valued logic helps modelling the concept of modes in a *moded* logic programming language such as Mercury.

A version of this paper with a different copyright notice is to appear at 18th Computing: Australasian Theory Symposium (CATS 2012), Melbourne, Australia. Conferences in Research and Practice in Information Technology, Vol. 128. Julian Mestre, Ed. Reproduction for academic, not-for profit purposes permitted provided this text is included. This work was partially supported by ARC grant DP 110102579.

- We argue that the semantics fits well with established practice in program debugging and verification.

We assume the reader has a basic understanding of pure logic programs, including programs in which clause bodies use negation, and their semantics. We also assume the reader has some familiarity with the concepts of types and modes as they are used in logic programming.

The paper is structured as follows. We set the scene in Section 2 by revisiting the problems that surround approaches to logical semantics for pure Prolog. In Section 3 we introduce the three- and four-valued logics and many-valued interpretations that the rest of the paper builds upon. In Section 4 we provide some background on different approaches to the semantics of pure Prolog, focusing on work by Fitting and Kunen. In Section 5 we review Naish's approach to what we call specification semantics. In Section 6 we present a new four-valued approach which combines two three-valued approaches (Fitting and Naish). Section 7 establishes a "model intersection" principle. Section 8 shows how a four-valued approach helps modelling the concept of *modes* in a moded logic language such as Mercury. Section 9 sketches the application to declarative debugging. Section 10 recapitulates, offering a high-level justification for the four-valued approach, and Section 11 concludes.

2 Logic programs

Suppose we need Prolog predicates to capture the workings of classical propositional conjunction and negation. We may specify the behaviour exhaustively (we use `neg` for negation; `not` is used as a general negation primitive in Prolog):

```
or(t, t, t).          neg(t, f).
or(t, f, t).          neg(f, t).
or(f, t, t).
or(f, f, f).
```

yielding simple, correct predicates. If we also need a predicate for implication, we could define

```
implies(X, Y) :- neg(X, U), or(U, Y, t).
```

Variables in the head of a clause are universally quantified over the whole clause; those which only occur in the body are existentially quantified within the body.

Although Prolog programs explicitly define only what is true, it is also important that they implicitly define what is false. This is the case for most programs and is essential when negation is used. For example, `neg(t,t)` would be considered false and for it to succeed would be an error. Because (implicit) falsehood depends on the set of all clauses defining a predicate, it is often convenient to group all clauses into a single definition with distinct variables in the arguments of the clause head. This can be done in Prolog by using the equality (`=`) and disjunction (`;`) primitives. For example, `neg` could be defined

```
neg(X, Y) :- (X=t, Y=f ; X=f, Y=t).
```

Clark (1978) defined the *completion* of a logic program which explicitly groups clauses together in this way; others (Fitting (1991), Naish (2006)) assume the program contains a single clause per predicate from the outset. Henceforth we assume the same. The `:-` in single-clause definitions thus tells us about both the truth and falsehood of instances of the head. Exactly how `:-` is best viewed has been the topic of much debate and is a central focus of this paper. One issue is

```
p(a).
p(b) :- p(b).
p(c) :- not p(c).
p(d) :- not p(a).
```

Figure 1: Small program to exemplify semantics

the relationship between the truth values of the head and body — what set of truth values do we use, what constitutes a model or a fixed point, etc. Another is whether we consider one particular model/fixed point (such as the least one according to some ordering) as the semantics or do we consider any one of them to be a possible semantics or consider the set of all models/fixed points as the semantics.

Let us fix our vocabulary for logic programs and lay down an abstract syntactic form.

Definition 1 (Syntax) An atom (or atomic formula) is of the form $p(t_1, \ldots, t_n)$, where p is a predicate symbol (of arity n) and $t_1, \ldots, t_n$ are terms. If $A = p(t_1, \ldots, t_n)$ then A's predicate symbol $pred(A)$ is p. There is a distinguished equality predicate $=$ with arity 2, written using infix notation. A *literal* is an *atom* A or the negation of an atom, written $\neg A$. A *conjunction* C is a conjunction of literals. A *disjunction* D is of the form $C_1 \vee \cdots \vee C_k$, $k > 0$, where each C_i is a conjunction. A *predicate definition* is a pair $(H, \exists W[D])$ where H is an atom in most general form $p(V_1, \ldots, V_n)$ (that is, the V_i are distinct variables), D is a disjunction, and $W = vars(D) \setminus vars(H)$. We call H the *head* of the definition and $\exists W[D]$ its *body*. The variables in H are the *head variables* and those in W are *local variables*. Finally, a *program* is a finite set P of predicate definitions such that if $(H_1, B_1) \in S$ and $(H_2, B_2) \in S$ then $pred(H_1) \neq pred(H_2)$.

We let $\mathcal{G}$ denote the set of ground atoms (for some suitably large fixed alphabet).

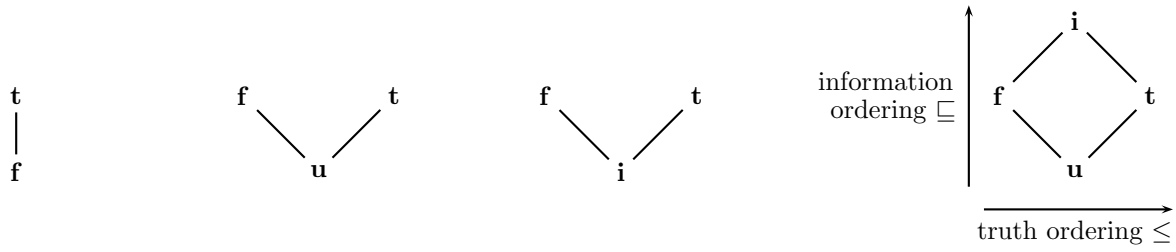
Definition 2 (Head instance) A *head instance* of a predicate definition $(H, \exists W[D])$ is an instance where all head variables have been replaced by ground terms and local variables remain unchanged.

3 Interpretations and models

In two-valued logic, an interpretation is a mapping from $\mathcal{G}$ to $\mathbf{2} = \{\mathbf{f}, \mathbf{t}\}$. To give meaning to recursively defined predicates, the usual approach is to impose some structure on $\mathcal{G} \to \mathbf{2}$, to ensure that we are dealing with a lattice, or a semi-lattice at least. Given the traditional "closed-world" assumption (that a formula is false unless it can be proven true), the natural ordering on $\mathbf{2}$ is this: $b_1 \leq b_2$ iff $b_1 = \mathbf{f} \vee b_2 = \mathbf{t}$. The ordering on interpretations is the natural extension of $\leq$, equipped with which $\mathcal{G} \to \mathbf{2}$ is a complete lattice.

Three-valued logic is arguably a more natural logic for the partial predicates that emerge from pure Prolog programs, and more generally, for the partial functions that emerge from programming in any Turing complete language. The case for three-valued logic as the appropriate logic for computation has been made repeatedly, starting with Kleene (1938) and pursued by the VDM school (see for example Barringer, Cheng & Jones (1984)), and others. The third value, $\mathbf{u}$, for "undefined", finds natural uses, for example as the value of `p(b)`, given the program in Figure 1.

With three- or four-valued logic, an interpretation becomes a mapping from $\mathcal{G}$ to $\mathbf{3} = \{\mathbf{u}, \mathbf{f}, \mathbf{t}\}$ or to $\mathbf{4} = \{\mathbf{u}, \mathbf{f}, \mathbf{t}, \mathbf{i}\}$ (we discuss the role of the fourth value $\mathbf{i}$ shortly.) For compatibility with the way equality is treated in Prolog, we constrain interpretations so $x = y$ is mapped to $\mathbf{t}$ if x and y are



(a) Classical order **2** (b) Kleene's order **3** (c) Naish's order (d) interlaced bilattice **4**

Figure 2: Partially ordered sets of truth values

identical (ground) terms, and **f**, otherwise. This is irrespective of the set of truth values used. There are different choices for the semantics of the connectives. Based on the natural "information content" orderings shown in Figure 2(b) and (d), the natural choices are the strongest monotone extensions of the two-valued connectives. This gives rise to Kleene's (strong) three-valued logic K_3 (Kleene 1938) and Belnap's four-valued logic (Belnap 1977). We denote the ordering depicted in Figure 2(b) by $\sqsubseteq$, that is, $b_1 \sqsubseteq b_2$ iff $b_1 = \mathbf{u} \vee b_1 = b_2$, and we overload this symbol to also denote the ordering in Figure 2(d) (that is, $b_1 \sqsubseteq b_2$ iff $b_1 = \mathbf{u} \vee b_1 = b_2 \vee b_2 = \mathbf{i}$), as well of the natural extensions to $\mathcal{G} \to \mathbf{3}$ or $\mathcal{G} \to \mathbf{4}$. We shall also use $\sqsupseteq$, the inverse of $\sqsubseteq$. In some contexts we disambiguate the symbol by using a superscript: $\sqsupseteq^{\mathbf{3}}$ or $\sqsupseteq^{\mathbf{4}}$. Similarly, we use $\geq^{\mathbf{2}}$ for the truth ordering with two values, and $=^{\mathbf{2}}$, $=^{\mathbf{3}}$ and $=^{\mathbf{4}}$ for equality of truth values in the different domains.

The structure in Figure 2(d) is the simplest of Ginsberg's bilattices (Ginsberg 1988). The diamond shape can be considered a lattice from two distinct angles. The ordering $\leq$ is the "truth" ordering, whereas $\sqsubseteq$ is the "information" ordering. For the truth ordering we denote the meet and join operations by $\wedge$ and $\vee$, respectively. For the information ordering we denote the meet and join operations by $\sqcap$ and $\sqcup$, respectively. The bilattice in Figure 2(d) is interlaced: Each meet and each join operation is monotone with respect to *either* ordering. The bilattice is also distributive in the strong sense that each meet and each join operation distributes over all the others.

An equivalent view of three- or four-valued interpretations is to consider an interpretation to be a pair of ground atom sets. That is, the set of interpretations $\mathcal{I} = \mathcal{P}(\mathcal{G}) \times \mathcal{P}(\mathcal{G})$. In this view an interpretation $I = (T_I, F_I)$ is a set T_I of ground atoms deemed true together with a set F_I of ground atoms deemed false. A ground atom A that appears in neither is deemed undefined. Such a truth value *gap* may arise from the absence of any evidence that A should be true, or that A should be false. In a four-valued setting, para-consistency is a possibility: A ground atom A may belong to $T_I \cap F_I$. Such a truth value *glut* may arise from the presence of conflicting evidence regarding A's truth value.

The concept of a *model* is central to many approaches to logic programming. A model is an interpretation which satisfies a particular relationship between the truth values of the head and body of each head instance. We now define how truth for atoms is lifted to truth for bodies of definitions.

Definition 3 (Made true) Let $I = (T_I, F_I)$ be an interpretation. Recall that ground equality atoms are in T_I or F_I, depending on whether their arguments are the same term.

For a ground atom A,

I makes A true iff $A \in T_I$
I makes A false iff $A \in F_I$

For a ground negated atom $\neg A$,

I makes $\neg A$ true iff $A \in F_I$
I makes $\neg A$ false iff $A \in T_I$

For a ground conjunction $C = L_1 \wedge \cdots \wedge L_n$,

I makes C true iff $\forall i \in \{1 \cdot\cdot n\}$ I makes L_i true
I makes C false iff $\exists i \in \{1 \cdot\cdot n\}$ I makes L_i false

For a ground disjunction $D = C_1 \vee \cdots \vee C_n$,

I makes D true iff $\exists i \in \{1 \cdot\cdot n\}$ I makes C_i true
I makes D false iff $\forall i \in \{1 \cdot\cdot n\}$ I makes C_i false

For the existential closure of a disjunction $\exists W[D]$,

I makes $\exists W[D]$ true iff
    I makes some ground instance of D true
I makes $\exists W[D]$ false iff
    I makes all ground instances of D false

We use this to extend interpretations naturally, so they map $\mathcal{G}$ and existential closures of disjunctions to **2**, **3** or **4**. We freely switch between viewing an interpretation as a mapping and as a pair of sets. Thus, for any formula F,

$$I(F) = \begin{cases} \mathbf{u} & \text{if } I \text{ neither makes } F \text{ true nor false} \\ \mathbf{f} & \text{if } I \text{ makes } F \text{ false and not true} \\ \mathbf{t} & \text{if } I \text{ makes } F \text{ true and not false} \\ \mathbf{i} & \text{if } I \text{ makes } F \text{ true and also false} \end{cases}$$

Definition 4 ($\mathcal{R}^{\mathcal{D}}$-Model) Let $\mathcal{D}$ be **2**, **3** or **4** and $\mathcal{R}^{\mathcal{D}}$ be a binary relation on $\mathcal{D}$. An interpretation I is a $\mathcal{R}^{\mathcal{D}}$-model of predicate definition (H, B) iff for each head instance $(H\theta, B\theta)$, we have $\mathcal{R}^{\mathcal{D}}(I(H\theta), I(B\theta))$. I is a $\mathcal{R}^{\mathcal{D}}$-model of program P if it is a $\mathcal{R}^{\mathcal{D}}$-model of every predicate definition in P.

For example, a $=^{\mathbf{2}}$-model is a two-valued interpretation where the head and body of each head instance have the same truth value.

Another important concept used in logic programming semantics and analysis is the "immediate consequence operator". The original version, T_P, took a set of true atoms (representing a two-valued interpretation) and returned the set of atoms which could be proved from those atoms by using a clause for a single deduction step. Various definitions which generalise this to **3** and **4** have been given (see Apt & Bol (1994)). Here we give a definition based on how we define interpretations. We write Φ_P for the immediate consequence operator, following Fitting (1985).

Definition 5 (Φ_P) Given an interpretation I and program P, $\Phi_P(I)$ is the interpretation I' such that the truth value of an atom H in I' is the truth value of B in I, where (H, B) is a head instance of a definition in P.

Proposition 1 An interpretation I is a fixed point of Φ_P iff I is a $=^d$-model of P, for d in $\{\mathbf{2}, \mathbf{3}, \mathbf{4}\}$.

Proof Straightforward from the definitions. □

4 Logic program operational semantics

We first discuss some basic notions and how Clark's two-valued approach to logic program semantics fits with what we have presented so far. Then we discuss the Fitting/Kunen three-valued approach and Fitting's four-valued approach.

4.1 Two-valued semantics

There are three aspects to the semantics of logic programs: proof theory, model theory and fixed point theory (see Lloyd (1984), for example). The proof theory is generally based on resolution, often some variant of SLDNF resolution (Clark 1978). This gives a top-down operational semantics, which we don't consider in detail here. The model theory gives a declarative view of programs and is particularly useful for high level reasoning about partial correctness. The fixed point semantics, based on Φ_P or T_P, gives an alternative "bottom up" operational semantics (which has been used in deductive databases) and which is also particularly useful for program analysis.

The simplest semantics for pure Prolog disallows negation and treats a Prolog program as a set of definite clauses. Prolog's `:-` is treated as classical implication, $\leftarrow$, that is, $\geq^{\mathbf{2}}$-models are used. There is an important soundness result: if the programmer has an intended interpretation which is a model, any ground atom which succeeds is true in that model. The $(\leq)$ least model is also the least $=^{\mathbf{2}}$-model and the least fixed point of Φ_P, which is monotone in the truth ordering (so a least fixed point always exists). The atoms which are true in this least model are precisely those which have successful derivations using SLD resolution. For these reasons, this is the accepted semantics for Prolog programs without negation.

To support negation in the semantics, Clark (1978) combined all clauses defining a particular predicate into a single "if and only if" definition which uses the classical bi-implication $\leftrightarrow$. This is called the Clark completion *comp*(P) of a program P. Our definitions are essentially the same, but we avoid the $\leftrightarrow$ symbol. In this paper's terminology, Clark used $=^{\mathbf{2}}$-models, which correspond to classical fixed points of Φ_P. The soundness result above applies, and any finitely failed ground atom must also be false in the programmer's intended interpretation, if it is a model. However, because Φ_P is non-monotone in the truth ordering when negation is present, there may be multiple minimal fixed points/models, or there may be none. For example, using Clark's semantics for the program in Figure 1, there is no model and no fixed point due to the clause for `p(c)`, yet the query `p(a)` succeeds and `p(d)` finitely fails. Thus the Clark semantics does not align particularly well with the operational semantics.

4.2 Three-valued semantics

Even in the absence of negation, a two-valued semantics is lacking in its inability to distinguish failure and looping. Mycroft (1984) explored the use of many-valued logics, including **3**, to remedy this. Mycroft discussed this for Horn clause programs, and others, including Fitting (1985) and Kunen (1987), subsequently adapted Clark's work to a three-valued logic, addressing the problem of how to account properly for the use of explicit negation in programs.

In a two-valued setting the Clark completion may be inconsistent, witness the completion of the clause for `p(c)` in Figure 1. A $=^{\mathbf{3}}$-model always exists for a Clark-completed program; for example, `p(c)` takes on the third truth value. Moreover, since Φ_P is monotone with respect to the information ordering, a least fixed point always exists and coincides with the least $=^{\mathbf{3}}$-model. Ground atoms which are $\mathbf{t}$ in this model (such as `p(a)` in Figure 1) are those which have successful derivations, while ground atoms which are $\mathbf{f}$ (such as `p(d)`) are those which have finitely failed SLDNF trees. Atoms with the third truth value (`p(b)` and `p(c)`) must loop. If we were to delete the clause for `p(c)` in Figure 1, the Clark semantics would map `p(b)` to $\mathbf{f}$, even though it does not finitely fail. Atoms which are $\mathbf{t}$ or $\mathbf{f}$ in the Fitting/Kunen semantics may also loop if the search strategy or computation rule are unfair (even without negation, $\mathbf{t}$ atoms may loop with an unfair search strategy). However the Fitting/Kunen approach does align the model theoretic and fixed point semantics much more closely to the operational semantics than the approach of Clark.

Φ_P has a drawback, though: while monotone, it is not in general continuous. Blair (1982) shows that the smallest ordinal β for which $\Phi_P^\beta(\bot)$ is the least fixed point of Φ_P may not be recursive and Kunen (1987) shows that, with a semantics based on three-valued Herbrand models (all models or the least model), the set of ground atoms true in such models may not be recursively enumerable. Kunen instead suggests a semantics based on *any* three-valued model and shows that truth ($\mathbf{t}$) in all $=^{\mathbf{3}}$-models is equivalent to being deemed true by $\Phi_P^n(\bot)$ for some $n \in \mathbb{N}$. Hence Kunen proposes $\Phi_P^\omega(\bot)$ as the meaning of program P. For a given P and ground atom A, it is decidable whether A is $\mathbf{t}$ in $\Phi_P^n(\bot)$, so whether A is $\mathbf{t}$ in $\Phi_P^\omega(\bot)$ is semi-decidable.

For simplicity, in this paper we take (the possibly non-computable) $M = \mathit{lfp}(\Phi_P)$ to be the meaning of a program. However, since we shall be concerned with over-approximations to M, what we shall have to say will apply equally well if Kunen's $\Phi_P^\omega(\bot)$ is assumed.

4.3 Four-valued semantics

Subsequent to his three-valued proposal, Fitting recommended, in a series of papers including Fitting (1991, 2002), bilattices as suitable bases for logic program semantics. The bilattice **4** (Figure 2(d)) was just one of several studied for the purpose, and arguably the most important one.

Fitting's motivation for employing four-valued logic was, apart from the elegance of the interlaced bilattices and their algebraic properties, the application in a logic programming language which supports a notion of (spatially) distributed programs. In this context there is a natural need for a fourth truth value, $\top$ (our $\mathbf{i}$), to denote conflicting information received from different nodes in a distributed computing network.

In this language, the traditional logical connectives used on the right-hand sides of predicate definitions are explained in terms of the truth ordering. Negation is reflection in the truth ordering: $\neg\mathbf{u} = \mathbf{u}$, $\neg\mathbf{f} = \mathbf{t}$, $\neg\mathbf{t} = \mathbf{f}$ and $\neg\mathbf{i} = \mathbf{i}$, conjunction is meet ($\wedge$), disjunction is join ($\vee$), and existential quantification is the least upper bound ($\bigvee$) of all instances. The following tables give conjunction and disjunction in **4**.

$\wedge$	**u**	**t**	**f**	**i**
u	**u**	**u**	**f**	**f**
t	**u**	**t**	**f**	**i**
f	**f**	**f**	**f**	**f**
i	**f**	**i**	**f**	**i**

$\vee$	**u**	**t**	**f**	**i**
u	**u**	**t**	**u**	**t**
t	**t**	**t**	**t**	**t**
f	**u**	**t**	**f**	**i**
i	**t**	**t**	**i**	**i**

The operations $\sqcap$ and $\sqcup$ are similarly given by Figure 2(d). Fitting refers to $\sqcap$ (he writes $\otimes$) as *consensus*, since $x \sqcap y$ represents what x and y agree about. The $\sqcup$ operation (which he writes as $\oplus$) he refers to *gullibility*, since $x \sqcup y$ represents agreement with both x and y, whatever they say, including cases where they disagree.

The idea of a information (or knowledge) ordering is familiar to anybody who has used domain theory and denotational semantics. To give meaning to recursively defined objects we refer to fixed points of functions defined on structures equipped with some ordering — the information ordering. This happens in Fitting's three-valued semantics: That uses the same distinction between a truth ordering $\leq$ and an information ordering $\sqsubseteq$ but it does not expose it as radically as the bilattice. In Fitting's words, the three-valued approach, "while abstracting away some of the details of [Kripke's theory of truth] still hides the double ordering structure" (Fitting 2006).

The logic programming language of Fitting (1991) contains operators $\otimes$ and $\oplus$, reflecting the motivation in terms of distributed programs. We, on the other hand, deal with a language with traditional pure Prolog syntax. If the task was simply to model its operational semantics, having four truth values rather than three would offer little, if any, advantage. However, our motivation for using four-valued logic is very different to that of Fitting. We find compelling reasons for the use of four-valued logic to explain certain programming language features, as well as to embrace, semantically, such software engineering aspects as program correctness with respect to programmer intent or specification, declarative debugging, and program analysis. We next discuss one of these aspects.

5 Three-valued specification semantics

Naish (2006) proposed an alternative three-valued semantics. Unlike other approaches, the objective was not to align declarative and operational semantics. Instead, the aim was to provide a declarative semantics which can help programmers develop correct code in a natural way. Naish argued that intentions of programmers are not two-valued. It is generally intended that some ground atoms should succeed (be considered **t**) and some should finitely fail (be considered **f**) but some should never occur in practice; there is no particular intention for how they should behave and the programmer does not care and often does not know how they behave. An example is merging lists, where it is assumed two sorted lists are given as input: it may be more appropriate to consider the value of `merge([3,2],[1],[1,3,2])` *irrelevant* than to give it a classical truth value, since a *precondition* is violated. Or consider this program:

```
or2(t, _, t).          or3(_, t, t).
or2(f, B, B).          or3(B, f, B).
```

It gives two alternative definitions of `or` (defined in Section 2), both designed with the assumption that the first two arguments will always be Booleans. If they are not, we consider the atom is *inadmissible* (a term used in debugging (Pereira 1986, Naish 2000)) and give it the truth value **i**. Interpretations can be thought of as the programmer's understanding of a specification, where **i** is used for underspecification of behaviour. The same three-valued interpretation can be used with all three definitions of `or`, so a programmer can first fix the interpretation then code any of these definitions and reason about their correctness. In contrast, both the Clark and Fitting/Kunen semantics assign different meanings to the three definitions, with atoms such as `or3(4,f,4)` and `or2(t,[],t)` considered **t** and `or3(t,[],t)` considered **f**. In order for the programmer's intended interpretation to be a $=^{\mathbf{2}}$-model or $=^{\mathbf{3}}$-model, unnatural distinctions such as these must be made. Naish (2006) argues that it is unrealistic for programmers to use such interpretations as a basis for reasoning about correctness of their programs.

Although Naish uses **i** instead of **u** as the third truth value, his approach is structurally the same as Fitting/Kunen's with respect to the Φ_P operator and the meaning of connectives used in the body of definitions. The key difference is how Prolog's `:-` is interpreted. Fitting generalises Clark's classical $\leftrightarrow$ to $\cong$ or "strong equivalence", where heads and bodies of head instances must have the same truth values. Naish defined a different "arrow", $\leftarrow$, which is asymmetric. In addition to identical truth values for heads and bodies, Naish allows head instances of the form $(\mathbf{i}, \mathbf{f})$ and $(\mathbf{i}, \mathbf{t})$. The difference is captured by these tables (Fitting left, Naish right):

$\cong$	**t**	**f**	**u**
t	**t**	**f**	**f**
f	**f**	**t**	**f**
u	**f**	**f**	**t**

$\leftarrow$	**t**	**f**	**i**
t	**t**	**f**	**f**
f	**f**	**t**	**f**
i	**t**	**t**	**t**

Naish's reasoning is that if a predicate is called in an inadmissible way, it does not matter if it succeeds or fails. The definition of a model uses this weaker "arrow"; we discuss it further in Section 6. Naish (2006) shows that for any model, only **t** and **i** atoms can succeed and only **f** and **i** atoms can finitely fail. In models of the code in Figure 1, `p(b)` can be **t** or **f** or **i** but `p(c)` can only be **i**. For practical code, programmers can reason about partial correctness using intuitive models in which the behaviour of some atoms is unspecified.

6 Four-valued specification semantics

The Fitting/Kunen and Naish approaches all use three truth values, the Kleene strong three-valued logic for the connectives in the body of definitions, and the same immediate consequence operator. It is thus tempting to assume that the "third" truth value in these approaches is the same in some sense. This is implicitly assumed by Naish, when different approaches are compared (Table 1 of Naish (2006)). However, the third truth value is used for very different purposes in these approaches. Fitting uses it to make the semantics more precise than Clark — distinguishing success and finite failure from nontermination (neither success nor finite failure). Naish uses it to make the semantics *less* precise than Clark, allowing a truth value corresponding to success or finite failure. Thus we believe it is best to treat the third truth values of Fitting and Naish as *duals* instead of the same value. Because conjunction, disjunction and negation in **4** are symmetric in the information order, the third value in the Kleene strong three-valued logic can map to either the top or bottom element in **4**. This is why the third truth values in Fitting/Kunen

and Naish are treated in the same way, even though they are better viewed as semantically distinct.

The four values **t**, **f**, **i** and **u** are associated with truth/success, falsehood/finite failure, inadmissibility (the Naish third value) and looping/error (the Fitting/Kunen third value). Inadmissibility can be seen as saying both success and failure are correct, so we can see it as the union of both. Atoms which are **u** in the Fitting semantics neither succeed nor fail. Thus the information ordering can also be seen as the set ordering, $\subseteq$, if we interpret the truth values as sets of Boolean values. In Naish, **i** is implicitly considered the bottom element so the ordering used in that work is the inverse of the ordering considered here.

We now show how Naish's semantics can be generalised to **4**. As discussed above, adding the truth value **i** to the Fitting semantics does not allow us to describe what is *computed* any more precisely, though it can be useful for *approximating* what is computed. However, adding the truth value **u** to the Naish semantics *does* allow us to describe more precisely what is *intended*. There are occasions when both the success and finite failure of an atom are considered incorrect behaviour and thus **u** is an appropriate value to use in the intended interpretation. We give three examples. The first is an interpreter for a Turing-complete language. If the interpreter is given (the representation of) a looping program it should not succeed and it should not fail. The second is an operating system. Ignoring the details of how interaction with the real world is modelled in the language, termination means the operating system crashes. The third is code which is only intended to be called in limited ways, but is expected to be robust and check its inputs are well formed. Exceptions or abnormal termination with an error message are best not considered success or finite failure. Treating them in the same way as infinite loops in the semantics may not be ideal but it is more expressive than using the other three truth values (indeed, "infinite" loops are never really infinite because resources are finite and hence some form of abnormal termination results).

Naish (2006) defines models in terms of the $\leftarrow$ described earlier, and his Proposition 7 relates models to the information ordering on interpretations. This is actually a key observation (though the significance is not noted by Naish (2006)): the $\leftarrow$ defines the information order on truth values! The classical arrow defines the truth ordering on two values; Naish's arrow defines the orthogonal ordering in the three-valued extension. It is therefore clear how Naish's arrow can be generalised to **4**. The models of Naish (2006) are $\sqsupseteq^{\mathbf{3}}$-models, which can be generalised to $\sqsupseteq^{\mathbf{4}}$-models.

Proposition 2 M is a $\sqsupseteq^{\mathbf{4}}$-model of P iff $\Phi_P(M) \sqsubseteq M$.

Proof M is a $\sqsupseteq^{\mathbf{4}}$-model iff, for every head instance (H, B) of P, $M(B) \sqsubseteq M(H)$. This is equivalent to stating that if M makes B true then M makes H true, and also, if M makes B false then M makes H false. But this is the case iff $\Phi_P(M) \sqsubseteq M$, by the definition of Φ_P. $\square$

It is easy to see that if M is a $\sqsupseteq^{\mathbf{3}}$-model of P then M is a $\sqsupseteq^{\mathbf{4}}$-model of P. However, the converse is not necessarily true, so the results of Naish (2006) cannot be used to show properties of four-valued models. However, such properties can be proved directly, using properties of the lattice of interpretations.

Proposition 3 If M is a $\sqsupseteq^{\mathbf{4}}$-model of P then $\mathit{lfp}(\Phi_P) \sqsubseteq M$.

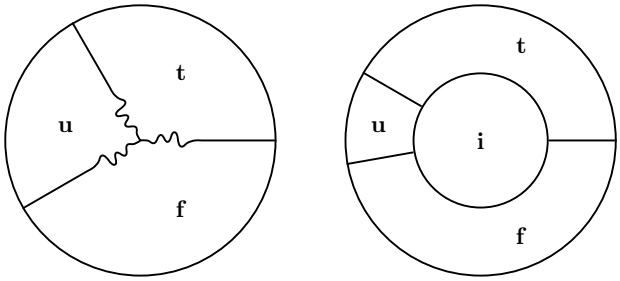


Figure 3: Least vs typical intended $\sqsupseteq^{\mathbf{4}}$-model

Proof The proof is by transfinite induction. Given program P, define

$$I^\beta = \begin{cases} \Phi_P(I^{\beta'}) & \text{if } \beta \text{ is a successor ordinal } \beta'+1 \\ \bigsqcup_{\alpha<\beta} I^\alpha & \text{if } \beta \text{ is a limit ordinal} \end{cases}$$

Assume $I^\alpha \sqsubseteq M$ for all ordinals $\alpha < \beta$. We show that $I^\beta \sqsubseteq M$.

First consider the case $\beta = \beta'+1$. By the induction hypothesis, $I^{\beta'} \sqsubseteq M$. Since Φ_P is monotone, $I^\beta = \Phi_P(I^{\beta'}) \sqsubseteq \Phi_P(M)$. By Proposition 2, $\Phi_P(M) \sqsubseteq M$. Hence $I^\beta \sqsubseteq M$.

Now consider the case of limit ordinal $\beta = \bigsqcup_{\alpha<\beta} \alpha$. By definition, $I^\beta = \bigsqcup_{\alpha<\beta} I^\alpha$. By the induction hypothesis, $I^\alpha \sqsubseteq M$, for each $\alpha < \beta$. But then by properties of the least upper bound operation, $I^\beta = \bigsqcup_{\alpha<\beta} I^\alpha \sqsubseteq M$. $\square$

Proposition 4 The least $\sqsupseteq^{\mathbf{4}}$-model of P is $\mathit{lfp}(\Phi_P)$.

Proof This follows from Proposition 3 and the fact that fixed points are $=^{\mathbf{4}}$-models. $\square$

For reasoning about partial correctness, the relationship between truth values in an interpretation and operational behaviour is crucial.

Theorem 1 If M is a $\sqsupseteq^{\mathbf{4}}$-model of P then no **t** atoms in M can finitely fail, no **f** atoms in M can succeed and no **u** atoms in M can finitely fail or succeed.

Proof Finitely failed atoms are **f** in $\mathit{lfp}(\Phi_P)$, successful atoms are **t** in $\mathit{lfp}(\Phi_P)$, and **u** atoms in $\mathit{lfp}(\Phi_P)$ must loop, from Kunen. From Proposition 3 and the $\sqsubseteq$ ordering, **f** atoms in M can only be **f** or **u** in $\mathit{lfp}(\Phi_P)$, **t** atoms in M can only be **t** or **u** in $\mathit{lfp}(\Phi_P)$, and **u** atoms in M can only be **u** in $\mathit{lfp}(\Phi_P)$. $\square$

These results about the behaviour of **t** and **f** atoms are essentially the two soundness theorems, for finite failure and success, respectively, of Naish (2006). The result for **u** atoms is new. The relationship between the operational semantics and various forms of three-valued model-theoretic semantics was summarised by Table 1 of Naish (2006). However, it assumed the Fitting/Kunen third truth value was the same as Naish's. We can now refine it using the four values, as follows (the last row summarises Theorem 1):

operational	**succeed**	**loop**	**fail**
least $=^{\mathbf{4}}$-model	**t**	**u**	**f**
any $=^{\mathbf{4}}$-model	**t**	**t/u/i/f**	**f**
any $\sqsupseteq^{\mathbf{4}}$-model	**t/i**	**t/u/i/f**	**i/f**

Figure 3 gives a graphical representation of how the least model compares with a typical intended model.

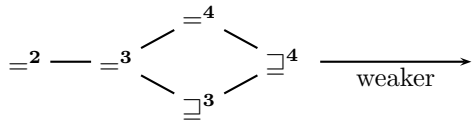


Figure 4: Relationship between model definitions

In the least model, no atoms are **i**, and there is a correspondence between the truth values of atoms, **t**, **f** and **u**, and their behaviour — success, finite failure, and looping, respectively. However, the distinction between these categories can be subtle and counter-intuitive (hence the wiggly lines). In a typical intended interpretation there are atoms which are **i** (they may have any other truth value in the least model). This allows the distinction between the categories to be more intuitive and allows a single interpretation to be a model of many different programs with different behaviours for the **i** atoms. The set of **u** atoms in a typical intended interpretation is a subset of the **u** atoms in the minimal model. Atoms which are **u** in the minimal model can have any truth value in the intended model. The case where the intended model has no **u** atoms corresponds to a three-valued model of Naish (2006).

Figure 4 shows the relationship between the five different definitions of a model we have considered. Any interpretation which is a model according to one definition is also a model according to all definitions to the right. Weaker definitions of models allow more flexibility in how we think of our programs, yet still guarantee partial correctness.

7 A "model intersection" property

With the classical logic approach for definite clause programs, we have a useful model intersection property: if M and N are (the set of true atoms in) models then $M \cap N$ is (the set of true atoms in) a model. Proposition 1 of Naish (2006) generalises this result using the truth ordering for three-valued interpretations, and Proposition 2 of Naish (2006) gives a similar result which mixes the truth and information orderings. However, none of these results hold for logic programs with negation. Here we give a new analogous result, using the information ordering, which holds even when negation is present. This will be utilised in the next section, on modes.

Proposition 5 If M and N are $\sqsupseteq^4$-models of program P then $M \sqcap N$ is a $\sqsupseteq^4$-model of P.

Proof By Proposition 2, $\Phi_P(M) \sqsubseteq M$ and $\Phi_P(N) \sqsubseteq N$, since M and N are models. By monotonicity, $\Phi_P(M \sqcap N) \sqsubseteq \Phi_P(M) \sqsubseteq M$, and $\Phi_P(M \sqcap N) \sqsubseteq \Phi_P(N) \sqsubseteq M$. It follows that $\Phi_P(M \sqcap N) \sqsubseteq M \sqcap N$, so by Proposition 2, $M \sqcap N$ is a model of P. □

This result does not hold for $=^4$-models. For example:

```
p :- p.
q :- q.
r :- p ; q ; s.
s :- p ; q ; not r.
```

Let M be the interpretation which maps (p,q,r,s) to (**t**,**f**,**t**,**t**), respectively, and N be the interpretation (**f**,**t**,**t**,**t**). Both M and N are $=^4$-models. The meet, $M \sqcap N$, is (**u**,**u**,**t**,**t**) but Φ_P applied to this interpretation is (**u**,**u**,**t**,**u**). So $M \sqcap N$ is a $\sqsupseteq^4$-model but not a $=^4$-model.

8 Types and modes

We now discuss the motivation for type and mode systems in logic programming and show how $\sqsupseteq^4$-models could have a role to play in mode systems. The lack of restrictions on what constitutes an acceptable Prolog program means that is it is easy for programmers to make simple mistakes which are not immediately detected by the Prolog system. A typical symptom is the program unexpectedly fails, leading to rather tedious analysis of the complex execution in order uncover the mistake. One approach to avoid some runtime error diagnosis is to impose additional discipline on the programmer, generally restricting programming style somewhat, in order to allow the system to statically classify certain programs as incorrect. Various systems of "types" and "modes" have been proposed for this. An added benefit of some systems is that they help make implementations more efficient. Here we discuss such systems at a very high level and argue that four-valued interpretations potentially have a role in this area, particularly in mode systems such as that of Mercury (Somogyi, Henderson & Conway 1995).

Type systems typically assign a type (say, Boolean, integer, list of integers) to each argument of each predicate. This allows each variable occurrence in a clause to also be assigned a type. One common error is that two occurrences of the same variable have different types. For example, consider a predicate `head` which is intended to return the head of a list of integers but is incorrectly defined as: `head([_|Y],Y)`. The first occurrence of `Y` is associated with type list of integer and the other is associated with type integer. If `head` is called with both arguments instantiated to the expected types, it must fail. But `head` can succeed if it is called in different ways. For example, with only the first argument instantiated it will succeed, albeit with the wrong type for the second argument (and this in turn may cause a wrong result or failure of a computation which calls `head`).

Type systems can be refined by considering the "mode" in which predicates are called, or dependencies between the types of different arguments. This can allow additional classes of errors to be detected. For example, we can say the first argument of `head` is expected to be "input" and the second argument can be "output". Alternatively (but with similar effect), we could say if the first argument is a list of integers, the second should be an integer. For a definition such as `head([_|Y],X)` there is a consistent assignment of types to variables but it does not satisfy this mode/type-dependency constraint. One high level constraint of several mode systems is that if input arguments are well typed then output arguments should be well typed for any successful call. In fact, we want the whole successful derivation to be well typed (otherwise we have a very dubious proof). Typically, well typed inputs in a clause head imply well typed inputs in the body, which implies well typed outputs in body, which implies well typed outputs in the head. This idea is present in the directional types concept (Aiken & Lakshman 1994, Boye & Małuszynski 1995), the mode system of Mercury (Somogyi et al. 1995), and the view of modes proposed in Naish (1996). Here we show the relevance of four-valued interpretations to this idea, ignoring the details of what constitutes a type (which differs in the different proposals) and what additional constraints are imposed (neither Mercury or directional types support cyclic dataflow and Mercury has additional interactions between types, modes and determinism).

```
% Type t has a single constructor, 'g'

:- type t ---> g.
:- pred p1(t, t).   % other preds similar

:- mode p1(in, in) is nondet.   % OK
% :- mode p1(out, in) is nondet.  % OK
% :- mode p1(in, out) is nondet.  % Error
% :- mode p1(out, out) is nondet. % Error
p1(g, _).

:- mode p2(in, out) is nondet. % one/both
:- mode p2(out, in) is nondet. % modes OK
p2(A, A).

:- mode p3(in, out) is nondet. % both modes
:- mode p3(out, in) is nondet. % needed
p3(A, B) :- p3(B, A).

:- mode p4(in, out) is nondet. % OK
% :- mode p4(in, in) is nondet.  % Error
p4(A, A) :- p4(A, _).

:- mode p5(in, out) is nondet. % OK
p5(g, _) :- error("...").
```

Figure 5: Mercury well modedness

We will use Mercury in our examples. Mercury allows types to be defined using `type` declarations and declared for predicate arguments using `pred` declarations. Modes are declared using `mode` declarations, which also declare determinism (in our examples we use `nondet`, which imposes no constraint and can be ignored). Figure 5 gives some very simple examples. It defines a type `t`, containing a single constant `g`, which we use for all arguments of all predicates. We use `g` because any well typed argument must be ground whereas other arguments may be non-ground in an answer computed by Prolog (Mercury rejects any programs where this could occur). The predicate definitions do not compute anything sensible, and two loop in all cases. Additional arguments could be added to remedy these defects, but our aim is to concentrate on modes (particularly some of the less intuitive aspects).

The definition of `p1` constrains the first arguments to be `g` but does not constrain the second argument. Thus the first argument can be input or output, but the second argument must be input. Mercury allows multiple mode declarations for a single predicate, as shown in `p2`; the predicate must be well moded for each mode. In this case the program is considered well moded with either or both the mode declarations, which is typical. However, there are cases, such as `p3`, which are well moded with both modes but neither single mode is sufficient, because the predicate has a recursive call which typically has a different instantiation pattern from the top-level call. With only a single mode declaration, the inputs of the clause head are not passed on to the inputs of the recursive call, potentially resulting in a dubious proof. Another example of such recursive calls is shown in `p4`. Although `p4` is well moded with respect to mode `(in,out)`, it is not well moded with the mode `(in,in)`, even though the latter mode imposes more constraints on the way in which `p4` is used. The last example, `p5`, illustrates another feature of the Mercury mode system. Mercury has an `error` primitive which leads to abnormal termination. Where it is used, the constraints of the mode system are not enforced (since no proof is possible), so it is well moded with mode `(in,out)`, whereas `p1` (which has a similar definition) is not.

Type and mode declarations document some aspects of how predicates are intended to be used and how they are intended to behave. We define a subset of possible interpretations which are consistent with these declarations. We assume there is a notion of well typedness for each argument of each predicate in program P.

Definition 6 (Mode and mode interpretation) A *mode* for predicate p is an assignment of "input" or "output" to each of p's argument positions. Each predicate has a set of modes. A *mode interpretation* of P is a four-valued interpretation M such that the truth value of an atom A in M is

1. **i**, if there is no mode of the predicate for which all input arguments are well typed, and
2. **f**, if there is a mode of the predicate for which all input arguments are well typed but some (output) argument is not well typed.

Other atoms may take any truth value.

In typical automated mode analysis there is no additional information about other user-defined atoms and it can be assumed they are **t**. The (builtin) `error` atoms should be **u** for a language like Mercury. In the mode interpretation corresponding to Figure 5, `p1(g,g)` is **t** and {`p1(n,n)`, `p1(n,g)`, `p1(g,n)`} are all **i** (we use `n` as a representative ill-typed term; it also corresponds to non-ground computed answers). If the mode of `p1` was changed to `(in,out)`, `p1(g,n)` would be **f**. Changing the modes of a predicate so it can be used in more flexible ways corresponds to changing the truth value of some atoms from **i** to **f**. For `p2`, with two mode declarations, only `p1(n,n)` is **i**. Mode interpretations which are $\sqsupseteq^{\mathbf{4}}$-models give us the high level properties of well modedness:

Lemma 1 If a mode interpretation M of a program P is a $\sqsupseteq^{\mathbf{4}}$-model and A is a successful atom which, for some mode of the predicate, has all input arguments well typed, then A has all arguments well typed.

Proof By Theorem 1, since M is a $\sqsupseteq^{\mathbf{4}}$-model and A succeeds, A must be **t** or **i** in M. By the definition of mode interpretations, since A is not **f** and all input arguments are well typed for some mode, all output arguments must be well typed as well. □

Lemma 2 If a mode interpretation M of a program P is a $\sqsupseteq^{\mathbf{4}}$-model and A, with $M(A) = \mathbf{t}$, succeeds, then A is well typed and there is a ground clause instance A `:-` B_1 `;` $\ldots$ `;` B_n such that all atoms in some B_i are well typed and assigned **t**.

Proof Since M is a mode interpretation and A is not **i**, A has all input arguments well typed for some mode, so by Lemma 1 all arguments are well typed. Since A is **t** and M is a $\sqsupseteq^{\mathbf{4}}$-model, the clause body must be **t** or **u**. Because it succeeds it cannot be **u**, so it must be **t**. By the definition of disjunction and conjunction, all atoms in some B_i must be **t**. Since M is a mode interpretation, each of these atoms must have well typed inputs for some mode (otherwise they would be **i**). They all succeed, so by Lemma 1 they must be well typed. □

Theorem 2 If a mode interpretation M of a program P is a $\sqsupseteq^{\mathbf{4}}$-model and A is a **t** atom which succeeds, then there is a proof in which all atoms are well typed.

Proof By induction on the depth of the proof and Lemma 2. □

The mode interpretation corresponding to the code in Figure 5 is a $\sqsupseteq^{\mathbf{4}}$-model of the program. The same holds when a mode declaration is replaced by any of those "commented out" variants that are labelled OK. Conversely, the interpretations corresponding to ill-moded variants are not $\sqsupseteq^{\mathbf{4}}$-models. For example, predicate `p1` with mode `(in,out)` has a clause instance `p(g,n) :- true`, of the form **f** `:-` **t** (whereas `p5` is well moded with this mode; its instance is of the form **f** `:-` **u**). For `p3` with `(in,out)` as the only mode we have clause instance `p3(g,n) :- p3(n,g)`, of the form **t** `:-` **i**. For `p4` with mode `(in,in)` we have the clause instance `p4(g,g) :- p4(g,n)`, of the same form.

Although mode interpretations do not capture all the complexities of the Mercury mode system, they do give us a high level view and some additional insights. For any predicate definition, we know there is a lattice of mode interpretations, some of which are typically $\sqsupseteq^{\mathbf{4}}$-models. Each one corresponds to a set of mode declarations. Models higher in the information order place more restrictions on how we use a predicate — more atoms are **i** and more arguments must be input. Proposition 5 tells us the meet of two $\sqsupseteq^{\mathbf{4}}$-models is a $\sqsupseteq^{\mathbf{4}}$-model. This corresponds to taking the union of the sets of mode declarations (the set of **i** atoms in the meet is the intersection of the **i** atoms in the two $\sqsupseteq^{\mathbf{4}}$-models). One way the Mercury mode system could potentially be extended is by allowing the programmer to specify several sets of mode declarations for a predicate (corresponding to several $\sqsupseteq^{\mathbf{4}}$-models). A predicate such as `p2` could have two singleton sets of modes declared (with the union implicit due to Proposition 5), whereas `p3` would need a single set of two mode declarations. This potentially could allow more errors to be detected and perhaps greater efficiency (avoiding some modes of a predicate appearing in the object code if they are not required). An understanding of the lattice of mode interpretations may also be helpful for mode inference.

9 Declarative debugging

The semantics of Naish (2006) is closely aligned with declarative debugging (Shapiro 1983) and the term "inadmissible" comes from this area (Pereira 1986). In particular, it gives a formal basis for the three-valued approach to declarative debugging of Naish (2000), as applied to Prolog. This debugging scheme represents the computation as a tree; sub-trees represent sub-computations. Each node is classified as correct, erroneous or inadmissible. The debugger searches the tree for a *buggy* node, which is an erroneous node with no erroneous children. If all children are correct it is called an *e-bug*, otherwise (it has an inadmissible child) it is called an *i-bug*. Every finite tree with an erroneous root contains at least one buggy node and finding such a node is the job of a declarative debugger.

To diagnose wrong answers in Prolog a proof tree (see Lloyd (1984)) is used to represent the computation. Nodes containing **t**, **f** and **i** atoms are correct, erroneous and inadmissible, respectively. To diagnose computations that *miss* answers, a different form of tree is used, and nodes containing finitely failed **t**, **f** and **i** atoms are erroneous, correct, and inadmissible, respectively. There are some additional complexities, such as non-ground wrong answers and computations which return some but not all correct answers; we skip the details here. Four-valued interpretations could be used in place of three-valued interpretations in this scheme. For wrong answer diagnosis, **u** should be treated the same as **f** and for missing answer diagnosis **u** should be treated the same as **t**. Naish (2000) also discusses diagnosis of abnormal termination and (suspected) non-termination, but assumes only **i** atoms should loop or terminate abnormally. With four-valued interpretations this restriction can be lifted.

10 Computation and information ordering

The logic programming paradigm introduced the view of computation as deduction (Kowalski 1980). Classical logic was used and hence computation was identified with the truth ordering. Similarly, there was much early work discussing the relationship between specifications (written in classical first order logic) and programs (Hogger 1981, Kowalski 1985). This work generally overlooked what we call inadmissibility. For example, Kowalski (1985) gives a specification for the `subset(SS,S)` predicate, $\forall E[member(E,S) \rightarrow member(E,SS)]$ (sets are represented as lists and *member* is the Prolog list membership predicate), and shows that a common Prolog implementation of `subset` is a logical consequence. However, `subset(true,42)` is true according to the specification and if the specification is modified to restrict both arguments to be lists, the program is no longer a logical consequence (it has `subset([],_)` as a base case). When negation is considered, or even the fact that logic programs implicitly define falsehood of some atoms, it becomes clear that approaches based on the truth ordering are unworkable.

Four-valued logic enables us to identify computation with the *information ordering* rather than the truth ordering. Specifications can be identified with intended interpretations, and inadmissibility with underspecification. There can be different logic programs, with different behaviours, which are correct according to a specification — they can be seen as refinements of the specification. The behaviour of a program is given by its least $\sqsupseteq^{\mathbf{4}}$-model, and it is (partially) correct if and only if the least model is less than or equal to the specification, in the information ordering. The specification being a $\sqsupseteq^{\mathbf{4}}$-model is a sufficient condition for correctness.

The same ordering applies to successive states of a computation using a correct program. Because $H \sqsupseteq B$ for each head instance, replacing a subgoal by the body of its definition (a basic step in a logic programming computation) gives us a new goal which is lower (or equal) in the information ordering, in the following sense. Given a top-level Prolog goal, the intended interpretation gives a truth assignment for each ground instance. Subsequent resolvents can also be given a a truth assignment for each ground instance of the variables in the top level goal (with local variables considered existentially quantified). As the computation progresses, the truth value assignment for each instance often remains the same, but can become lower in the information ordering. For example, consider the goal `implies(X,f)`. Our interpretation will map `implies(f,f)` to **t** and `implies(t,f)` to **f**, but may map `implies(42,f)` to **i**, if the first argument is expected to be input. After one step of the computation we have the conjunction `neg(X,U)` $\wedge$ `or(U,f,t)`. If our intended interpretation allows any mode for `neg`, the instance where `X` $= 42$ is then mapped to **f**.

We believe that having a complete lattice using the information ordering provides an important and fundamental insight into the nature of computation. At the top of the lattice we have an element which corresponds to underspecification in the mind of a person. At the bottom of the lattice we have an element which corresponds to a the inability of a machine or formal system to compute or define a value. The transitions

between the meanings we attach to specifications and correct programs, and successive execution states of a correct program, follow the information ordering, rather than the truth ordering.

11 Conclusion

We have been aware of the limitations of formal systems since well before the invention of electronic computers. Gödel showed the impossibility of a complete proof procedure for elementary number theory, hence important gaps between truth and provability, and in any Turing-complete programming language there are programs which fail to terminate — undefinedness is unavoidable. Our awareness of the limitations of humans in their interaction with computing systems goes back even further. Babbage (1864) claims to have been asked by members of the Parliament of the United Kingdom, "Pray, Mr. Babbage, if you put into the machine wrong figures, will the right answers come out"? The term "garbage in, garbage out" was coined in the early days of electronic computing and concepts such as "preconditions" have always been important in formal verification of software — underspecification is also unavoidable in practice.

Using a special value to denote undefinedness is the accepted practice in programming language semantics. Using a special value to denote underspecification is less well established, but has been shown to provide elegant and natural reasoning about partial correctness, at least in the logic programming context. In this paper we have proposed a domain for reasoning about Prolog programs which has values to denote both undefinedness and underspecification — they are the bottom and top elements of a bilattice. This gives an elegant picture which encompasses both humans not making sense of some things and computers being unable to produce definitive results sometimes. The logical connectives Prolog uses in the body of clauses operate within the truth order in the bilattice. However, the overall view of computation operates in the orthogonal "information" order: from underspecification to undefinedness.

References

Aiken, A. & Lakshman, T. K. (1994), Directional type checking of logic programs, *in* 'Static Analysis', Vol. 864 of *Lecture Notes in Computer Science*, Springer, pp. 43–60.

Apt, K. R. & Bol, R. N. (1994), 'Logic programming and negation: A survey', *Journal of Logic Programming* **19/20**, 9–71.

Babbage, C. (1864), *Passages from the Life of a Philosopher*, Longman and Co., London.

Barringer, H., Cheng, J. H. & Jones, C. B. (1984), 'A logic covering undefinedness in program proofs', *Acta Informatica* **21**, 251–269.

Belnap, N. D. (1977), A useful four-valued logic, *in* J. M. Dunn & G. Epstein, eds, 'Modern Uses of Multiple-Valued Logic', D. Reidel, pp. 8–37.

Blair, H. (1982), 'The recursion-theoretic complexity of the semantics of predicate logic as a programming language', *Information and Control* **54**, 25–47.

Boye, J. & Małuszynski, J. (1995), Two aspects of directional types, *in* L. Sterling, ed., 'Proc. Twelfth International Conf. Logic Programming', MIT Press, pp. 747–761.

Clark, K. L. (1978), Negation as failure, *in* H. Gallaire & J. Minker, eds, 'Logic and Data Bases', Plenum Press, pp. 293–322.

Fitting, M. (1985), 'A Kripke-Kleene semantics for logic programs', *Journal of Logic Programming* **2**(4), 295–312.

Fitting, M. (1989), Negation as refutation, *in* R. Parikh, ed., 'Proc. LICS 1989', IEEE, Cambridge, MA, pp. 63–70.

Fitting, M. (1991), 'Bilattices and the semantics of logic programming', *Journal of Logic Programming* **11**(2), 91–116.

Fitting, M. (2002), 'Fixpoint semantics for logic programming – a survey', *Theoretical Computer Science* **278**(1–2), 25–51.

Fitting, M. (2006), Bilattices are nice things, *in* T. Bolander, V. Hendricks & S. A. Pedersen, eds, 'Self-Reference', CSLI, Stanford, CA, pp. 53–77.

Ginsberg, M. (1988), 'Multivalued logics: A uniform approach to reasoning in artificial intelligence', *Computational Intelligence* **4**(3), 265–316.

Hogger, C. (1981), 'Derivation of logic programs', *JACM* **28**(2), 372–392.

Kleene, S. C. (1938), 'On notation for ordinal numbers', *The Journal of Symbolic Logic* **3**, 150–155.

Kowalski, R. A. (1980), *Logic for Problem Solving*, North Holland, New York.

Kowalski, R. A. (1985), The relation between logic programming and logic specification, *in* C. Hoare & J. Shepherdson, eds, 'Mathematical Logic and Programming Languages', Prentice-Hall, pp. 11–27.

Kunen, K. (1987), 'Negation in logic programming', *Journal of Logic Programming* **4**(4), 289–308.

Lloyd, J. W. (1984), *Foundations of Logic Programming*, Springer.

Mycroft, A. (1984), Logic programs and many-valued logic, *in* M. Fontet & K. Mehlhorn, eds, 'Symp. Theoretical Aspects of Computer Science', Vol. 166 of *Lecture Notes in Computer Science*, Springer, pp. 274–286.

Naish, L. (1996), A declarative view of modes, *in* 'Proc. 1996 Joint Int. Conf. Symp. Logic Programming', MIT Press, pp. 185–199.

Naish, L. (2000), 'A three-valued declarative debugging scheme', *Australian Computer Science Communications* **22**(1), 166–173.

Naish, L. (2006), 'A three-valued semantics for logic programmers', *Theory and Practice of Logic Programming* **6**(5), 509–538.

Pereira, L. M. (1986), Rational debugging in logic programming, *in* E. Shapiro, ed., 'Proc. Third Int. Conf. Logic Programming', Vol. 225 of *Lecture Notes in Computer Science*, Springer, pp. 203–210.

Shapiro, E. Y. (1983), *Algorithmic Program Debugging*, MIT Press, Cambridge MA.

Somogyi, Z., Henderson, F. J. & Conway, T. (1995), Mercury: An efficient purely declarative logic programming language, *in* 'Proc. Australian Computer Science Conf.', Glenelg, Australia, pp. 499–512.