Logic Programming: From Underspecification to Undefinedness

Lee Naish, Harald Søndergaard and Benjamin Horsfall
Computing and Information Systems
University of Melbourne

http://www.cs.mu.oz.au/~lee/papers/sem4lp/

Outline

Motivation

What programs compute (in brief)

What we intend programs to compute (in brief)

What logic programs compute

What we intend logic programs compute

A unified view

What computation is all about

Motivation

We want to reason about (relationships between)

- What we intend to compute
- Formal specifications
- Programs
- Computations
- What is computed

Logic is a possible unifying paradigm

Note: we concentrate on partial correctness here

What programs compute

Classical logic, with two truth values, is not enough to describe what Prolog programs compute

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

`p(a)` and `p(b)` succeed and finitely fail, respectively, but `p(c)` loops

Gödel and Turing tell us we can't avoid "undefinedness" in formal systems which are reasonably expressive (eg, Turing-complete languages)

What we intend programs to compute

Two values is not enough to describe our intentions either, because often we don't care about the behaviour in all cases

Eg, for `merge`, we assume the input lists are sorted; we don't specify what is computed in other cases because it shouldn't arise

The idea of "garbage in, garbage out", preconditions, inadmissibility, programming by contract, etc is not new:

"Pray, Mr. Babbage, if you put into the machine wrong figures, will the right answers come out"?

There are multiple correct implementations *which compute different things*, so the relationship between what is computed and what is intended is not 1:1

What logic programs compute

When talking about semantics its convenient to have only a single clause for each predicate, using disjunction and equality, eg

```
p(X) :- (X=a ; X=b, not p(a) ; X=c, p(c)).
```

A key question: what does "`:-`" mean?

Van Emden and Kowalski developed a fixedpoint semantics for Horn clauses, using the "immediate consequence operator" T_P, treating `:-` as classical $\leftarrow$

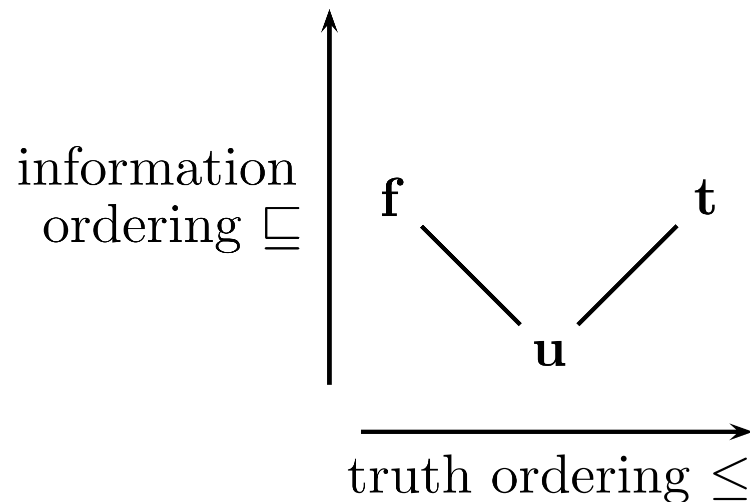
Clark combined clauses into single definitions to support negation and treated `:-` as classical $\leftrightarrow$

We use syntax $(H, \exists W[D])$, where W are the variables only in D

A head instance of (H, B) is an instance where variables in H are replaced by ground terms and other variables are unchanged

What logic programs compute (cont.)

Fitting and Kunen used **K3**, generalised T_P to Φ_P and treated `:-` as three-valued bi-implication/equivalence ($\cong$)



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

Conjunction is g.l.b. in truth order; similarly disjunction is l.u.b. and negation is reflection $\neg\mathbf{t} = \mathbf{f}$, $\neg\mathbf{f} = \mathbf{t}$, $\neg\mathbf{u} = \mathbf{u}$ (De Morgan's laws hold)

Generalises classical logic (**2**)

What logic programs compute (cont.)

Φ_P maps interpretations to interpretations (interpretations map ground atoms to truth values)

If (H, B) is a head instance, the truth value of B in I is the truth value of H in $\Phi_P(I)$

Φ_P is monotonic in the *information ordering* and its least fixedpoint gives the atoms which succeed/finitely fail/loop in Prolog (more or less)

What we intend logic programs compute

Naish (2006) proposed an approach similar to Fitting/Kunen using **3**, but `:-` is interpreted as the following "$\leftarrow$"

$\cong$	**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**

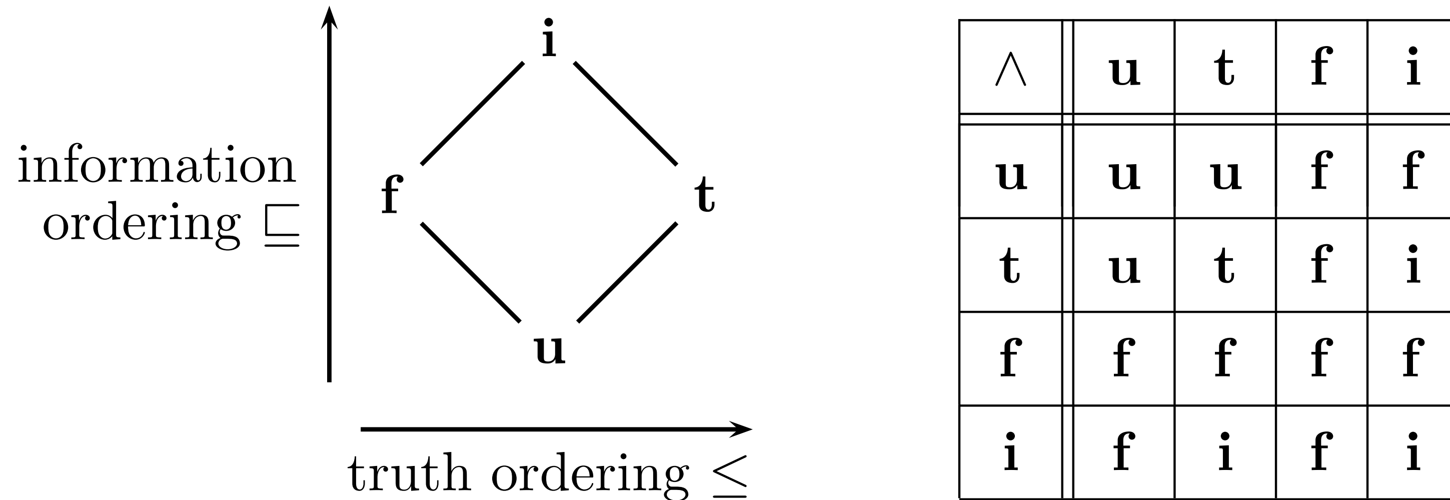
Clauses with heads **i** correspond to calls which are "inadmissible" or "garbage in" or "pre-conditions" are violated

Hence the body can have any truth value, we don't care

Although the same truth tables are used for $\vee$, $\wedge$ and $\neg$, **u** and **i** serve two quite different purposes

Belnap's 4

There are 4 truth values which can be arranged in a bilattice



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

Conjunction is g.l.b. in truth order; similarly disjunction is l.u.b. and negation is reflection $\neg\mathbf{t} = \mathbf{f}$, $\neg\mathbf{f} = \mathbf{t}$, $\neg\mathbf{i} = \mathbf{i}$, $\neg\mathbf{u} = \mathbf{u}$

Kleene's **3** is embedded in two ways

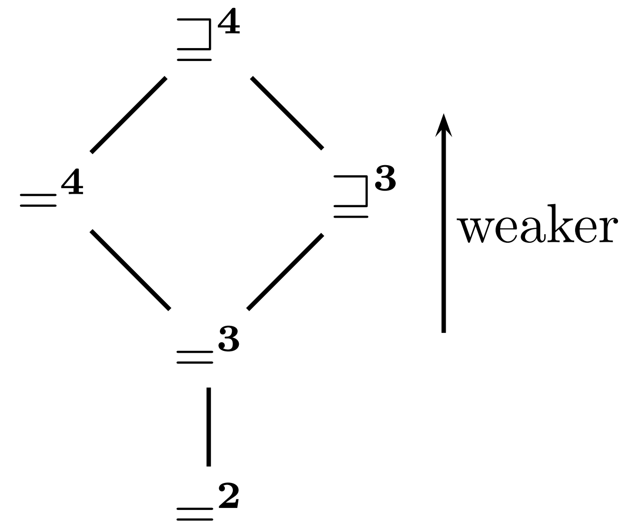
But how do we generalise the Naish (2006) "$\leftarrow$"?

Its the information ordering, rather than the truth ordering!

Models

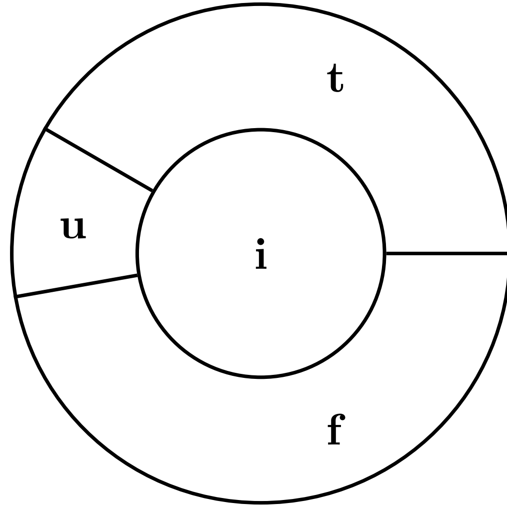
An interpretation I is a $\mathcal{R}^{\mathcal{D}}$-model, where $\mathcal{D}$ is a truth value domain (**2**, **3**, or **4**) and $\mathcal{R}$ is a relation over $\mathcal{D}$, iff for each head instance (H, B), we have $\mathcal{R}^{\mathcal{D}}(I(H), I(B))$

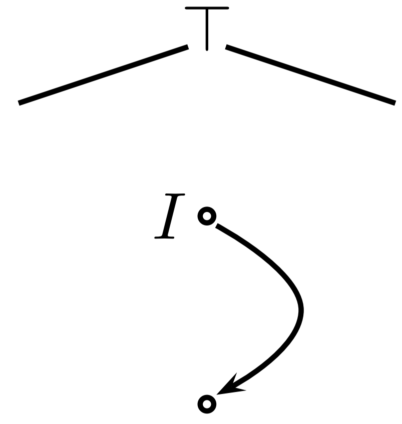
Semantics	`:-`	$\mathcal{R}^{\mathcal{D}}$
Van Emden/Kowalski	$\leftarrow$	$\geq^{\mathbf{2}}$
Clark	$\leftrightarrow$	$=^{\mathbf{2}}$
Fitting/Kunen	$\cong$	$=^{\mathbf{3}}$
Naish (2006)	"$\leftarrow$"	$\sqsupseteq^{\mathbf{3}}$
This work	$\sqsupseteq$	$\sqsupseteq^{\mathbf{4}}$

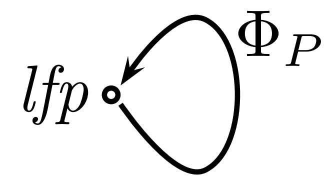


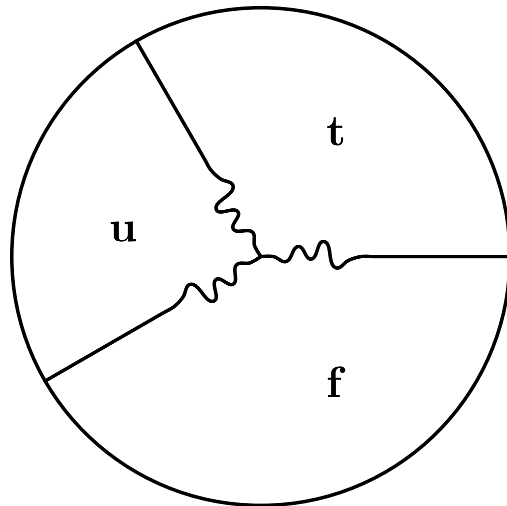
Our intended interpretation being a $\sqsupseteq^{\mathbf{4}}$-model is a sufficient condition for (partial) correctness

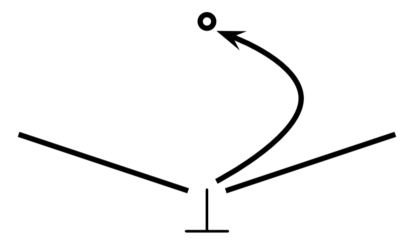
What is computed versus what we intend











Computation is the information ordering!

What should be the relationship between

- what we intend and what is computed?
- specifications and programs?
- type/mode declarations and definitions?
- heads and bodies of definitions within programs?
- successive execution states in a computation?

The information ordering, $\sqsupseteq$

The connectives in the bodies of definitions use the truth ordering

But computation operates in the information ordering: from underspecification to undefinedness