Undefinedness and underspecification

Contributors

Based largely on Lee Naish, Harald Søndergaard and Benjamin Horsfall, "Logic Programming: From Underspecification to Undefinedness", CATS 2012, http://www.cs.mu.oz.au/~lee/papers/sem4lp/

Also above authors plus Graeme Gange, "Multiple-Valued Fixed Points and Cyclic Circuits" (submitted), plus ongoing work with Bernie Pope

Outline

Motivation

What programs compute

What we intend programs to compute

What logic programs compute

What we intend logic programs compute

A unified view of everything:-)

Conclusions

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, as is the lambda calculus

Note: we concentrate on partial correctness here

What programs compute

Even for computing Booleans, Gödel and Turing tell us that two values is not enough when we have an expressive language, eg

```
>-- Haskell                          --% Prolog (kind of)
>e n =                               --
>   if n == 0 then True else         -- e(0).
>   if n == 1 then False else        -- e(N) :- N\=0, N\=1,
>   e (n-2)                          --      N2 is N-2, e(N2).
```

This code is able to determine if a natural number is even or not

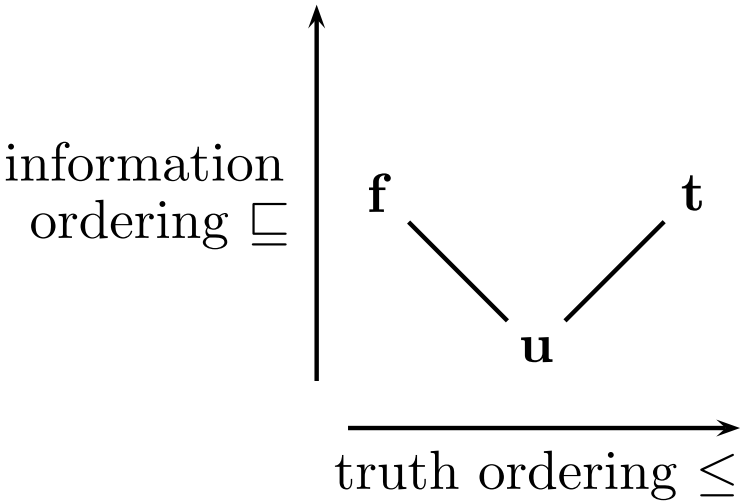
But when given a negative number, the code loops

In Haskell, `e (-1)` computes nothing and in Prolog, `e(-1)` neither succeeds nor finitely fails (it can't be proved or disproved)

We can't avoid "undefinedness" in formal systems which are reasonably expressive (eg, Turing-complete languages)

Kleene's strong three-valued logic (3)

As well as true and false there is a third "undefined" truth value; they can be arranged in a lattice



$\wedge$	**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 we intend programs to compute

In some contexts we only use natural numbers, and `e` is fine

When we define `merge`, we assume the input lists are sorted; we don't care what is computed otherwise because it shouldn't arise

There are multiple correct implementations *which compute different things* — there is not a 1:1 relationship between what is computed and what is intended

The idea of "garbage in, garbage out", preconditions, inadmissibility, contracts, etc is not new:

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

It seems we can't practically avoid "underspecification" either

What logic programs compute

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

```
e(N) :- (N=0 ;
         not N=0, not N=1, N2 is N-2, e(N2)).
```

What does ":-"mean? Is it classical $\leftarrow$, or $\leftrightarrow$ or something else?

What is the relationship between LHS and RHS of definitions?

We use abstract syntax $(H, \exists W[D])$, where W are the variables in D but not H

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

We (later) capture the relationship between heads and bodies by defining various kinds of *models*

What logic programs compute (cont.)

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 $\leftrightarrow$

Fitting and Kunen used **3**, generalised T_P to Φ_P and treated `:-` as three-valued bi-implication/equivalence ($\cong$)

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**

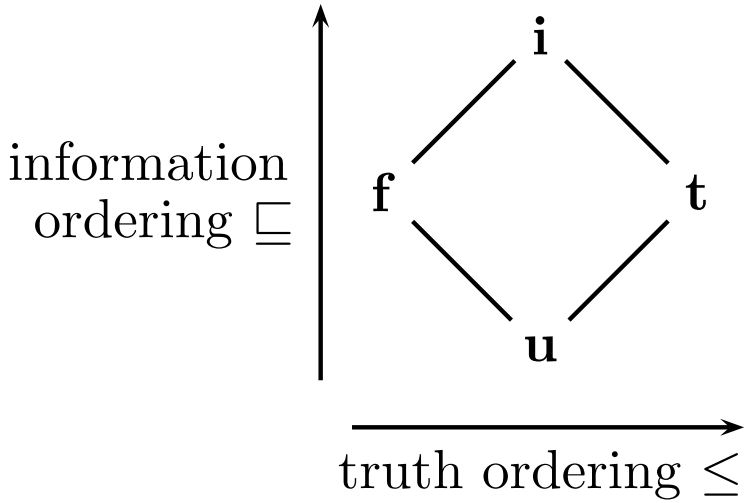
Clauses with heads **i** correspond to calls which are "inadmissible" or "garbage in" or "pre-conditions" are violated

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

Although same truth tables are used for $\vee$, $\wedge$ and $\neg$, **u** and **i** are really quite different

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

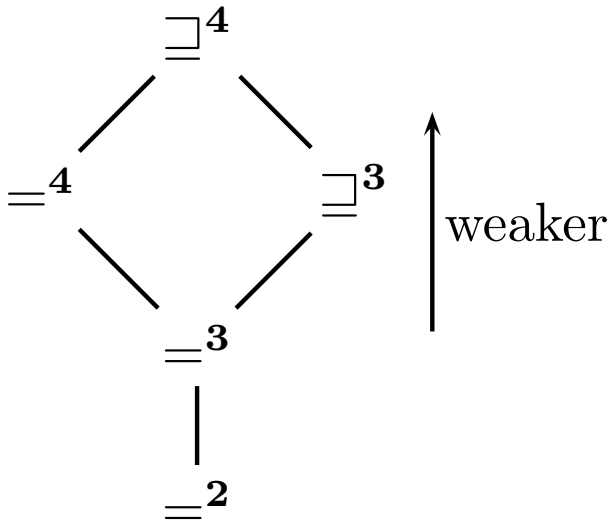
But how do we generalise the Naish (2006) "$\leftarrow$"?

Its the information ordering, rather than the truth ordering!

Models

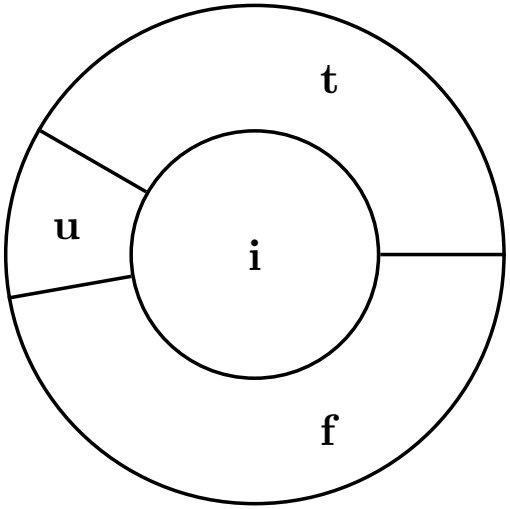
An interpretation I is a $\mathcal{R}^{\mathcal{D}}$-model, where $\mathcal{D}$ is a truth value domain (**2**, **3**, or **4**) and $\mathcal{R}$ is a relation over $\mathcal{D}$, iff for each head instance (H, B), we have $\mathcal{R}^{\mathcal{D}}(I(H), I(B))$

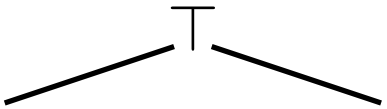
Semantics	:-	$\mathcal{R}^{\mathcal{D}}$
Van Emden/Kowalski	$\leftarrow$	$\geq^{\mathbf{2}}$
Clark	$\leftrightarrow$	$=^{\mathbf{2}}$
Fitting/Kunen	$\cong$	$=^{\mathbf{3}}$
Naish (2006)	"$\leftarrow$"	$\sqsupseteq^{\mathbf{3}}$
This work	$\sqsupseteq$	$\sqsupseteq^{\mathbf{4}}$

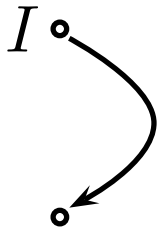


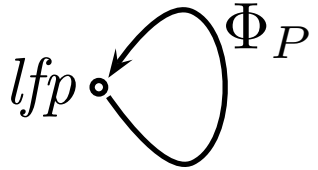
Our intended interpretation being a $\sqsupseteq^{\mathbf{4}}$-model is a sufficient condition for (partial) correctness

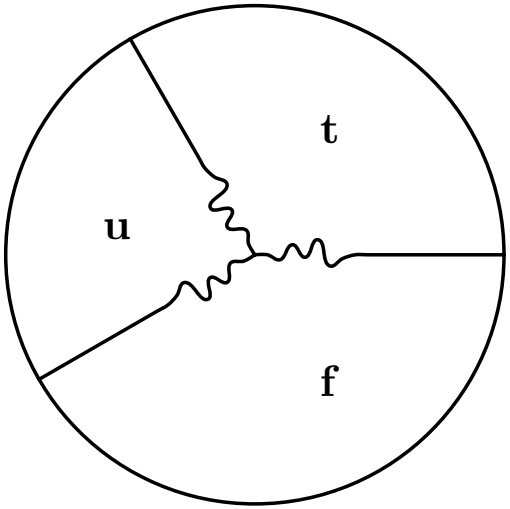
What is computed versus what we intend

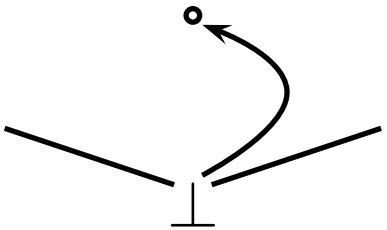












Computation is the information ordering!

What should be the relationship between

- what we intend and what is computed?
- specifications and programs?
- 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

Cyclic circuits (submitted to ISMVL12)

Logic circuits correspond to sets of Boolean equations; cyclic circuits correspond to recursive equations

Allowing cycles can reduce the size of circuits but some cyclic circuits are not well-behaved (eg, they may oscillate or have "memory" when its not desired)

Sometimes we don't care about the behaviour for some sets of inputs (eg, the seven segment display for decimal digits)

The same four-valued logic can be used to analyse such circuits (**u** = ill-behaved, **i** = don't care)

Functional programming (also in preparation)

Using a semi-lattice with $\perp = \mathbf{u}$ to define what is computed is well established

The Naish (2006) approach to what is intended has been adapted to functional programming: a semi-lattice with $\top = \mathbf{i}$

There are some advantages of the FP view, eg when we represent the same abstract value in more than one way

The two semi-lattices can be put together to form a complete lattice (work in progress)

Conclusions

Both undefinedness and underspecification are inconvenient to avoid

We can define a complete lattice with $\bot$ denoting undefined and $\top$ denoting unspecified, and a monotonic operator which corresponds to a computation step

This is an elegant framework for reasoning about *many* important aspects of computation which relate to correctness