Negation as failure
(Normal logic programs)

Marek Sergot
Department of Computing
Imperial College, London

January 2001; January 2010 v1.3f

Contents

- Negation as failure (NBF): Summary

  The original declarative semantics of NBF is in terms of the Clark completion, comp(D) of a logic program/database D. The SLDNF proof procedure, which is the most common way of executing normal logic programs (pure Prolog is a special case) is sound, but not complete, with respect to this semantics.

- Defaults and exceptions with NBF

  Normal logic programs (and negation by failure) provide a simple and practical formalism for expressing defaults and exceptions, and other forms of non-monotonic reasoning. There are some limitations however.

  (Another motivation: how do we build a deductive database incorporating a form of 'Closed World Assumption'?)

- Semantics for negation as failure (separate handouts)

  Various alternative semantics for negation by failure give a better account of the meaning of certain commonly occurring forms of logic programs than does the Clark completion, and also provide links to special-purpose non-monotonic reasoning systems. We cover:

  – stratified programs
  – stable models and answer sets

  There are various other semantics – 'perfect' models, 'well founded' models – which have connections to other non-monotonic reasoning methods, which we shall not cover.

These notes assume some familiarity with negation as failure and normal logic programs. Here is a summary.

Negation as failure: Summary

A *normal logic program* is a set of clauses (sometimes called 'extended definite clauses') of the form:

- $A \leftarrow L_1, \ldots, L_n \quad (n \geq 0)$

where A is an *atom* and each L_i is a *literal*. A literal is either an atom (a 'positive literal') or of the form `not` B where B is an atom. (`not` B is a 'negative literal').

A clause with no occurrences of `not` is called a *definite clause*.

The atom A is the *head* of the clause; the literals $L_1, \ldots, L_n$ are the *body* of the clause. When the body is empty ($n = 0$ above) the arrow $\leftarrow$ is usually omitted.

I will use (or try to remember to use) the Prolog convention that strings beginning with uppercase letters are variables.

`not` is **negation by failure** (NBF): `not` B succeeds when all attempts to prove B fail in finite time. (There are more precise ways of saying this, but this will do for the purposes of this brief summary.)

In addition to 'normal logic programs', there are also:

- *disjunctive logic programs*, in which clauses are of the form

  $$A_1 \vee \cdots \vee A_k \leftarrow B_1, \ldots, B_m, \texttt{not } C_1, \ldots, \texttt{not } C_n \qquad (k \geq 1, m \geq 0, n \geq 0)$$

  A_i, B_i, C_i all *atoms*.

- *extended logic programs* which combine classical, truth-functional negation $\neg$ with negation by failure `not` . Clauses have the form

  $$L \leftarrow L_1, \ldots, L_m, \texttt{not } L_{m+1}, \ldots, \texttt{not } L_n \qquad (m \geq 0, n \geq m)$$

  where L and each L_i are (classical) literals, i.e., of the form A or $\neg A$ where A is an atom.

We will look at extended logic programs later in the course.
We will not cover disjunctive logic programs.

In *Deductive databases*:

- *DATALOG* — definite clauses without function symbols

Sometimes *DATALOG* means no recursion allowed, and *DATALOG*$^+$ is *DATALOG* with recursion. Sometimes *DATALOGn* means *DATALOG* with negation (by failure).

Operational semantics

(Usually) the SLDNF proof procedure. There are other ways of executing normal logic programs, e.g., top-down (goal-directed) like SLDNF, top-down with tabulation of partial computations, 'all answers at a time', bottom-up, bottom-up with 'magic sets'. These will not be covered in this course, except for 'answer set programming' (and in passing, a brief mention of bottom-up methods).

SLDNF: The computation of a goal/query $G = L_1, \ldots, L_m$ is a series of derivation steps:

$\leftarrow G$

θ_1

$\leftarrow G_1$

θ_2

$\leftarrow G_2$

$\vdots$

$\leftarrow G_{n-1}$

θ_n

$\Box$ the empty goal

The θ_i are the unifiers (m.g.u.'s) produced by each derivation step. The answer computed θ is the composition of these unifiers $\theta = \theta_1 \cdot \cdots \cdot \theta_n$.

There are two kinds of derivation steps.

(a) Computation rule selects a positive literal $L_i = B$:

$\leftarrow L_1, \ldots, L_{i-1}, \boxed{B}, L_{i+1}, \ldots L_n$

θ resolve with clause $B' \leftarrow M_1, \ldots, M_k$ where $B.\theta = B'.\theta$

$\leftarrow (L_1, \ldots, L_{i-1}, M_1, \ldots, M_k, L_{i+1}, \ldots L_n)\theta$

(b) Computation rule selects a negative literal $L_i = \texttt{not}\ B$:

$\leftarrow L_1, \ldots, L_{i-1}, \boxed{\texttt{not}\ B}, L_{i+1}, \ldots L_n$

ι sub-computation: all ways of computing B **fail** (in finite time)

$\leftarrow L_1, \ldots, L_{i-1}, L_{i+1}, \ldots L_n$

ι is the identity substitution (the NBF sub-computation does not generate bindings for variables).

The *computation rule* picks out one of the literals of the current goal (query). The computation rule must be *safe*: it must not pick a negative literal containing a variable. This is necessary for soundness. If the current goal contains only negative literals with variables then the computation cannot proceed: it 'flounders'.

Declarative semantics

In the original declarative semantics, the meaning of a normal logic program/database D is given by the *Clark completion*:

$$\text{comp}(D)$$

comp(D) is intended to capture the meaning of negation by failure `not` in terms of ordinary, classical, truth-functional negation $\neg$.
The content of the program/database is Th(comp(D)).

The Clark completion comp(D)

- replaces all occurrences of `not` in D by ordinary (truth-functional) negation $\neg$.
  A clause $A \leftarrow B_1, \ldots, B_n$ is shorthand for the universally quantified formula
  $$\forall(B_1 \wedge \cdots \wedge B_n \rightarrow A)$$
  that is, $\forall x_1 \forall x_2 \ldots \forall x_m(B_1 \wedge \cdots \wedge B_n \rightarrow A)$ where $x_1, x_2, \ldots, x_m$ are the variables appearing the clause. ($\rightarrow$ is material implication.)
- and adds to the clauses in D the following additional formulas:
  - 'only if' counterparts to the clauses in D;
  - the Clark equality theory (*CET*): (axioms for =)

'only if' The 'definition' of a predicate `p` is the set of clauses with `p` in their heads: these are the clauses defining `p`. To construct the 'only if' part of the definition of `p`:
First re-write every clause
$$\texttt{p}(t_1, \ldots, t_m) \leftarrow L_1, \ldots, L_n$$
in 'homogenised form'
$$\texttt{p}(x_1, \ldots, x_m) \leftarrow x_1 = t_1, \ldots, x_m = t_m, L_1, \ldots, L_n$$
$x_1, \ldots, x_m$ are new variables not occurring in any clause with predicate `p` in its head.
Suppose the homogenised clauses defining `p` are:
$$\begin{aligned} \texttt{p}(x_1, \ldots, x_m) &\leftarrow E_1 \\ \texttt{p}(x_1, \ldots, x_m) &\leftarrow E_2 \\ &\vdots \\ \texttt{p}(x_1, \ldots, x_m) &\leftarrow E_k \end{aligned}$$
The 'completed definition' of `p` is obtained by adding the sentence:
$$\forall x_1 \ldots \forall x_m (\texttt{p}(x_1, \ldots, x_m) \rightarrow (E'_1 \vee E'_2 \vee \cdots \vee E'_k))$$
Each E'_i is obtained from E_i by (1) replacing the commas separating literals by $\wedge$ and all occurrences of negation by failure `not` by classical, truth-functional negation $\neg$, and then (2) prefixing existential quantifiers $\exists y_1 \ldots \exists y_t$ for any remaining variables $y_1, \ldots, y_t$ in E_i besides those $x_1, \ldots, x_m$ already in the head.
The completion comp(D) of D contains such an 'only if' clause for every predicate defined in D.
Finally, for any predicate `q` of arity n appearing in the body of a clause of D but not defined in D, add the sentence:
$$\forall x_1 \ldots \forall x_n \neg \texttt{q}(x_1, \ldots, x_n)$$

Clark equality theory (*CET*) Axioms for =:

- unique name axioms
  - $c \neq d$ for each pair of distinct constants c and d

  and more generally (constants can be regarded as function symbols of arity 0):
  - $\forall x_1 \ldots \forall x_n \forall y_1 \ldots \forall y_m \, (f(x_1, \ldots, x_n) \neq g(y_1, \ldots, y_m))$ for each pair of distinct function symbols f and g.
- axioms for equality (=): reflexivity, symmetry, transitivity, substitutivity;

$$\forall x \, (x = x)$$
$$\forall x \forall y \, (x = y \rightarrow y = x)$$
$$\forall x \forall y \forall z \, (x = y \wedge y = z \rightarrow x = z)$$

  and (substitutivity) for every n-ary predicate p:

$$\forall x_1 \ldots \forall x_n \forall y_1 \ldots \forall y_n \, (x_1 = y_1 \wedge \cdots \wedge x_n = y_n \rightarrow (\, p(x_1, \ldots, x_n) \leftrightarrow p(y_1, \ldots, y_n)\,))$$

- an axiom schema corresponding to the *occurs check* of the unification algorithm: $x \neq t[x]$ for any term $t[x]$ containing variable x.

(These axioms make = correspond to *unification*.)

Note: You sometimes see references to the *domain closure axiom*. If the language contains constants $a_1, a_2, \ldots, a_m$, the domain closure axiom is

$$\forall x \, (\, x = a_1 \vee x = a_2 \vee \cdots \vee x = a_m\,)$$

The domain closure axiom is *not* part of the Clark completion.

Example Here is a little program D to illustrate:

```
p(Y) ← q(Y), not r(a,Y)
p(f(Z)) ← not p(Z)
p(b)
r(a,b) ← q(b)
```

In 'homogenised' form:

```
p(X) ← X=Y, q(Y), not r(a,Y)
p(X) ← X=f(Z), not p(Z)
p(X) ← X=b
r(X,Y) ← X=a, Y=b, q(b)
```


The completion comp(D) contains the clauses

```
p(Y) ← q(Y), ¬r(a,Y)
p(f(Z)) ← ¬p(Z)
p(b)
r(a,b) ← q(b)
```

and the following 'only if' sentences:

$$\texttt{∀X}\,(\texttt{p(X)} \rightarrow \texttt{∃Y}\,(\texttt{X=Y} \wedge \texttt{q(Y)} \wedge \texttt{¬r(a,Y)}) \;\vee\; \texttt{∃Z}\,(\texttt{X=f(Z)} \wedge \texttt{¬p(Z)}) \;\vee\; \texttt{X=b}\,)$$
$$\texttt{∀X∀Y}\,(\texttt{r(X,Y)} \rightarrow \texttt{X=a} \wedge \texttt{Y=b} \wedge \texttt{q(b)}\,)$$
$$\texttt{∀X ¬q(X)} \qquad \texttt{\% because q is not defined in } D$$

together with the Clark equality theory for D.

Example Propositional programs are much simpler, because there are no quantifiers, no need for 'homogenisation', no need for the equality theory. Consider this little example D:

```
p ← not q
p ← r
q
```

$$\text{comp}(D) = \{\, \texttt{p} \leftrightarrow (\texttt{¬q} \vee \texttt{r}),\ \texttt{q},\ \texttt{¬r}\,\}$$

Strictly speaking, according to the definition above, that is not correct. Strictly speaking:

$$\text{comp}(D) = \{\, \texttt{p} \leftarrow \texttt{¬q},\ \texttt{p} \leftarrow \texttt{r},\ \texttt{q}\,\} \cup \{\, \texttt{p} \rightarrow (\texttt{¬q} \vee \texttt{r}),\ \texttt{¬r}\,\}$$

But it is usual to write the completion in the equivalent, simpler form using 'iff's.

Example Similarly, consider this little example D:

```
p(X) ← r(X), not q(X,Y)
p(X) ← not t(X)
q(a,b)
```

We would normally write (*CET* is the Clark equality theory):

$$\text{comp}(D) = \{\, \texttt{∀X}\,(\texttt{p(X)} \leftrightarrow ((\texttt{r(X)} \wedge \texttt{∃Y ¬q(X,Y)}) \vee \texttt{¬t(X)}),$$
$$\texttt{∀X∀Y}\,(\texttt{q(X,Y)} \leftrightarrow \texttt{X=a} \wedge \texttt{Y=b}),$$
$$\texttt{∀X ¬t(X)},\quad \texttt{∀X ¬r(X)}\,\} \cup\ \mathit{CET}$$

Small point: Note the quantification in $\texttt{∃Y ¬q(X,Y)}$. Prolog's unsound implementation of negation-by-failure (picking a `not` literal even if it contains a variable) would treat it as $\texttt{¬∃Y q(X,Y)}$ (which is wrong).

Thanks to Xu Zhao (MSc MAC, 2015–16) who spotted that many earlier versions of these notes stupidly omitted $\texttt{∀X ¬r(X)}$.

Soundness and completeness

Soundness (of SLDNF) For normal logic program/database D, if SLDNF computes an answer substitution θ (using a safe computation rule) for the goal $L_1, \ldots, L_n$, then

$$\text{comp}(D) \models \forall((L'_1 \wedge \cdots \wedge L'_n)\theta)$$

where each L'_i is obtained from L_i by replacing all occurrences of `not` by classical, truth-functional negation $\neg$. (The $\forall$ is necessary in these statements since the answer substitution θ may be not ground.)

Soundness (of NBF for SLDNF) For normal logic program/database D, if the SLDNF computation of goal $L_1, \ldots, L_n$ fails finitely (precise definition of this omitted here), then

$$\text{comp}(D) \models \forall(\neg(L'_1 \wedge \cdots \wedge L'_n)) \qquad \text{(equivalently: comp}(D) \models \neg\exists(L'_1 \wedge \cdots \wedge L'_n)\,)$$

where as usual each L'_i is obtained from L_i by replacing all occurrences of `not` by $\neg$.

But SLDNF is **not complete**

(1) because of the possibility of floundering;

(2) because of infinite computation trees ('loops').
Consider this program P:

```
r(a) ← p(a)
r(a) ← not p(a)
p(X) ← p(f(X))
```

Clearly comp(P) $\models$ `r(a)`. But no SLDNF computation of `?- r(a)` will fail finitely (they all get 'stuck in an infinite branch').

SLDNF is complete for some *special cases*: for example, for *definite programs*, logic programs where there is no negation `not` .

More pertinent for present purposes is the following problem.

comp(D) can be inconsistent

Consider

$$D = \{\texttt{p} \leftarrow \texttt{not p}\}$$

for which

$$\text{comp}(D) = \{\texttt{p} \leftrightarrow \neg\texttt{p}\}$$

which is clearly inconsistent (has no models).
(Strictly speaking, as defined above, comp(D) = $\{\texttt{p} \leftarrow \neg\texttt{p}, \texttt{p} \rightarrow \neg\texttt{p}\}$, which is equivalent to $\{\texttt{p} \leftrightarrow \neg\texttt{p}\}$.)

There are two problems:

(1) How can we determine, by looking at the form of program D, whether its completion comp(D) is consistent or inconsistent?
The completion comp(D) is guaranteed to be consistent for certain classes of program/database D, of which *stratified* is the most important. We shall examine those presently.

(2) There are intuitively well-behaved and coherent logic programs/databases which arise naturally in practice but for which the Clark completion is inconsistent. (Examples to follow.) Clearly, for those programs, the Clark completion is an inadequate formalisation of their intended meaning.

We will look at some alternative semantics for NBF later in the course.

Question Do we have comp(D) $\models D$, i.e. $D \subseteq$ Th(comp(D)) ?

Strictly speaking no, but only because we have to remember to replace negation-by-failure `not` in D by classical negation $\neg$.

We have the following properties of Th

- $A \subseteq \text{Th}(A)$
- $\text{Th}(A) \subseteq \text{Th}(A \cup B)$ (monotony)

and so

- $A \subseteq \text{Th}(A \cup B)$

So what we do have is

- $D' \subseteq \text{Th}(\text{comp}(D))$, i.e. comp($D$) $\models D'$

where D' is D with all occurrences of `not` replaced by $\neg$.

(A trivial point really. I just wanted to illustrate that remembering to convert `not` to $\neg$ is a nuisance.)

General rules and exceptions with NBF

Normal logic programs (and negation by failure) provide a simple and practical formalism for expressing defaults and exceptions, and other forms of non-monotonic reasoning.

Example

- Typically (by default, unless there is reason to think otherwise, ...) a bird can fly.
- Except that ostriches, which are birds, typically cannot fly.
- Also penguins, which are birds, cannot fly.
- Except that magic ostriches *can* fly (in general).

Formulation:

```
can_fly(X) ← bird(X), not abnormal_bird(X)
bird(X) ← ostrich(X)
abnormal_bird(X) ← ostrich(X), not abnormal_ostrich(X)
bird(X) ← penguin(X)
abnormal_bird(X) ← penguin(X), not abnormal_penguin(X)
ostrich(X) ← magic_ostrich(X)
abnormal_ostrich(X) ← magic_ostrich(X), not abnormal_magic_ostrich(X)
```

Check we get the intended behaviour:

- `bird(bill)`. We conclude (by NBF) `can_fly(bill)`.
- `ostrich(bill)`. We conclude (by NBF) `abnormal_bird(bill)`. And so (again by NBF) ¬`can_fly(bill)`.

  But note that this negative information is *implicitly* represented.
- `magic_ostrich(bill)`. We conclude (by NBF) `abnormal_ostrich(bill)`. And so now we do *not* have the conclusion `abnormal_bird(bill)`, and so `can_fly(bill)` again.

Example (Extension of the previous one)

- Typically (by default, unless there is reason to think otherwise, ...) a bird can fly.
- Except that ostriches, which are birds, typically cannot fly.
- Also penguins, which are birds, cannot fly.
- Except that magic ostriches *can* fly (in general).
- Jim is an ostrich (an ordinary ostrich) who can fly.
- Frank is a magic ostrich who cannot fly.
- No bird can fly if it is dead (no exceptions!).

Basic formulation (as before):

```
can_fly(X) ← bird(X), not abnormal_bird(X)
bird(X) ← ostrich(X)
abnormal_bird(X) ← ostrich(X), not abnormal_ostrich(X)
bird(X) ← penguin(X)
abnormal_bird(X) ← penguin(X), not abnormal_penguin(X)
ostrich(X) ← magic_ostrich(X)
abnormal_ostrich(X) ← magic_ostrich(X), not abnormal_magic_ostrich(X)
```

To represent that

- Jim is an ostrich (an ordinary ostrich) who can fly.

we have choices. We can write:

```
ostrich(jim)
abnormal_ostrich(jim)
```

or simply just

```
ostrich(jim)
can_fly(jim)
```

Which is better? Take your pick. (I think I might go for the second in this example, all things being equal. But that can also be dangerous — comments later.)

To represent that

- Frank is a magic ostrich who cannot fly.

we can write:

```
magic_ostrich(frank)
abnormal_magic_ostrich(frank)
```

Notice that we can't write directly (cf. `can_fly(jim)` above):

```
¬can_fly(frank)
```

because this is not a valid clause in a normal logic program. (It would be allowed in an *extended logic program*.)

To represent that

- No bird can fly if it is dead (no exceptions!)

we have to be careful. One possible solution:

```
abnormal_bird(X) ← dead(X)
```

Why do we have to be careful? Because if we chose to represent that Jim is an ostrich who can fly as `abnormal_ostrich(jim)` then `dead(jim)` means that Jim cannot fly (which seems right, given the way the rule was formulated). But if we had written `can_fly(jim)` then Jim still flies even if dead. We would have to revise the clause to say instead:

```
can_fly(jim) ← not dead(jim)
```

There are other solutions. One can see that

- it is reasonably straightforward to formulate these general rule and exception structures using NBF, but
- the formulation gets complicated and unwieldy quickly, and then it becomes easy to make a mistake.

Is this difficulty inherent in exception structures or is it a side-effect of using something as simple as NBF? We shall look at other formalisms presently.

Further complications A dead bird is abnormal from the point of view of flying (and singing) but not necessarily from the point of view of having feathers.

```
feathers(X) ← bird(X), not abnormal_bird_feathers(X)
can_sing(X) ← bird(X), not abnormal_bird_singing(X)
% frank the magic ostrich can't fly because he lost his feathers
abnormal_bird_feathers(frank)
% ostriches can't sing (typically)
abnormal_bird_singing(X) ← ostrich(X), not abnormal_ostrich_singing(X)
% except that Italian ostriches can sing (in general)
abnormal_ostrich_singing(X) ← italian_ostrich(X), not ab_it_ost_sing(X)
                    ⋮
```


Further Examples

(On Tutorial Exercise sheet)

Example 1 (from an example by Bob Kowalski)

(r1) Except as provided for by r2, all thieves should be punished.

(r2) Except as provided for by r3, thieves who are minors should be rehabilitated and not punished.

(r3) Any thief who is violent should be punished.

Reasonably straightforward:

```
punish(X) ← thief(X), not exception_by_r2(X)
rehab(X) ← thief(X), minor(X), not exception_by_r3(X)
exception_by_r2(X) ← thief(X), minor(X), not exception_by_r3(X)
exception_by_r3(X) ← thief(X), violent(X)
```

Much the same comments apply to this example as to the abnormal ostriches discussed earlier. In particular, note that the formulation of rule (r2) only gives the conclusion 'not punish' implicitly.

Example 2

- Europeans are typically civilised in behaviour. Except that ...
- Football supporters are typically not civilised in behaviour, unless they are educated.
- Even educated football supporters, on the other hand, are not civilised in behaviour if they are drunk.

These statements are quite ambiguous and can be interpreted in different ways.

Assuming that 'unless' in these rules is to be read as 'unless it is known that ...' (as suggested by the qualification 'typically') the first two rules could be formulated like this:

```
civil(X) ← european(X), not ab_civil_european(X)
ab_civil_european(X) ← football_supporter(X), not educated(X)
```

Again, 'not civilised' is here represented only implicitly: the query `civil(X)` will fail for any `X` who is a football supporter not known to be educated, or who is educated and drunk.

Now for the third rule. My reading of the third rule is that it is intended to give an exception to the second. So one possibility is to re-formulate like this:

```
civil(X) ← european(X), not ab_civil_european(X)
ab_civil_european(X) ← football_supporter(X), not ab_football_supporter(X)
ab_football_supporter(X) ← educated(X), not drunk(X)
```

The last clause could be made more general and more flexible re-written as:

```
ab_football_supporter(X) ← educated(X), not ab_educ_football_supporter(X)
ab_educ_football_supporter(X) ← drunk(X)
```

What about educated football supporters who are not European? Are they (typically) civilised or not? The rules as expressed in their natural language formulation are not clear on this point. Perhaps it is intended that all educated football supporters, European or not, are typically civilised. If so we could add the following clause:

```
civil(X) ← football_supporter(X), educated(X),
              not ab_educ_football_supporter(X)
```

What if 'unless' is not intended to be read as 'unless it is known that ...' but as ordinary, classical 'not'? Cannot be represented in a normal logic program, since there is no other kind of negation but negation by failure `not` .

The point of this example is to show (a) that normal logic programs can be used to express general rules and exceptions but (b) it is sometimes awkward and often not particularly clear. It is easy to make a mistake, i.e., express something other than what was intended.

We will look at some better — higher-level and more expressive — formalisms later.

Example 3

- A person who is big is assumed to be strong, unless there is reason to think (s)he is weak.
- A person who is small is assumed to be weak, unless there is reason to think that (s)he is strong, except that ...
- A person who is small and muscular is assumed to be strong unless there is reason to think that (s)he is weak.

This example is straightforward to formulate in the same style, but then difficult to figure out what is being said.
One possibility (1):

```
strong(X) ← big(X), not weak(X)
weak(X) ← small(X), not strong(X), not abnormal_small(X)
abnormal_small(X) ← small(X), muscular(X), not weak(X)
strong(X) ← small(X), muscular(X), not weak(X)
```

Another possibility (2):

```
strong(X) ← big(X), not weak(X)
weak(X) ← small(X), not strong(X)
strong(X) ← small(X), muscular(X), not weak(X)
```

Possibly even (3):

```
strong(X) ← big(X), not weak(X)
weak(X) ← small(X), not muscular(X), not strong(X)
strong(X) ← small(X), muscular(X), not weak(X)
```

I don't like this last one — it is not close to the way the original rules were expressed, and it does not generalise well.

What conclusions do these representations give for the various combinations of big/small and muscular/not muscular?

Let us assume that we never have, for any x, `big(`x`)` and `small(`x`)` at the same time. (How might one formalize such an important assumption?)

In each case, we have only two tools available (so far): the operational semantics (what will be computed by SLDNF); the Clark completion of the representation. Of these, the latter is very awkward to apply.

Suppose first that `big(jim)` (say) is added to the program (but not `small(jim)`). In all three formulations the computation is straightforward. This is because `small(jim)` fails so the other clauses play no role.

Suppose now that `small(mary)` is added to the program, but not `muscular(mary)` (and not `big(mary)`). Again it seems straightforward because the only clause that needs to be considered is the one defining `weak`: the others all have conditions which must fail for this set of data.

Suppose finally that both `small(jack)` and `muscular(jack)` are added to the program (but not `big(jack)`).

For the first formulation (1): the computation loops so we must look at the completion to determine what the answers would be.
One can see that the following are all implied by the completion (of program + data):

$$\texttt{weak(jack)} \leftrightarrow \neg\texttt{strong(jack)} \wedge \neg\texttt{abnormal\_small(jack)}$$
$$\texttt{abnormal\_small(jack)} \leftrightarrow \neg\texttt{weak(jack)}$$
$$\texttt{strong(jack)} \leftrightarrow \neg\texttt{weak(jack)}$$

(and the first is implied by the other two).
This suggests that the completion has two kinds of models: one in which `strong(jack)` is true and `weak(jack)` is false (and `abnormal_small(jack)` is true), and another in which `strong(jack)` is false and `weak(jack)` is true (and `abnormal_small(jack)` is false). This is easy to confirm. So we can reach no conclusion from this program about whether `jack` is strong or weak.

The second formulation (2): the computation loops. The completion (of program + data) implies:

$$\texttt{weak(jack)} \leftrightarrow \neg\texttt{strong(jack)}$$
$$\texttt{strong(jack)} \leftrightarrow \neg\texttt{weak(jack)}$$

It is easy to confirm that the completion has two kinds of models, one in which `jack` is weak and the other in which `jack` is strong.

The third formulation (3) is not very interesting, but for the sake of completeness: this one can be computed straightforwardly, because now only one clause (the third one) can play any role for `jack`: the first two all have conditions which fail for small muscular `jack`. (Which shows that formulation (3) is not equivalent to (1) or (2), according to the completion semantics).

Can we do any better than this kind of informal analysis? Perhaps the three formulations can be related in some way to one another? Alternative semantics for NBF comes next.

(As observed on the tutorial question sheet, the last example has the same logical form as some common patterns that arise in temporal reasoning, though this might not be apparent.)

Default reasoning with NBF: Other examples

Example

```
innocent(X) ← not guilty(X)
```

The default reasoning here has the same form as the examples of general rules and exceptions, but

- we would not say a person is *typically* innocent
- we would not say that there is a (defeasible) general rule that everyone is innocent.

In the context of determining whether a given `X` is innocent or guilty, the rule above says that we conclude `X` is innocent whenever it cannot be shown that `X` is guilty. The NBF here is a simple way of modelling *burden of proof*.
Some examples can be read (meaningfully) both ways, depending on context:

```
entitled_to_soc_security(X) ← not disqualified(X)
```

could be read, depending on context, as either (or both) of:

- a defeasible general rule with a possible exception, or
- a representation of what is required to establish entitlement (the 'burden of proof' is to show that `X` is disqualified).

The point is that for all these readings, and others — at this level of detail — the reasoning is technically the same.

Note also that 'typically', 'by default', . . . , is not necessarily the same as *most*.

Example A simple treatment of default persistence ('inertia') of facts over time.

Let `holds(F,T)` represent that fact (or 'fluent') `F` holds (is true) at time `T`.

Let `Ev initiates F` represent that an event of type `Ev` initiates a period of time for which fact ('fluent') `F` holds. `initiates` is a binary predicate written in infix form. For example:

```
birth(X) initiates alive(X)
hire(X) initiates employee(X)
```

Let `Ev terminates F` represent that an event of type `Ev` terminates a period of time for which fact ('fluent') `F` holds. `terminates` is a binary predicate written in infix form. For example:

```
death(X) terminates alive(X)
fire(X) terminates employee(X)
```

We might want to add something like

```
death(X) terminates employee(X) when employee(X)
```

but I don't want to get into such details now.

Let `happens(Ev,T)` represent that an event of type `Ev` happens at time `T`. For example:

```
happens(birth(jim), 0)
happens(hire(jim), 18)
happens(fire(jim), 19)
happens(hire(jim), 22)
happens(fire(jim), 45)
```

Here is a general formulation of what facts hold at what times using NBF as the device for dealing with default persistence ('inertia') of facts.

```
holds(F,T) ← happens(Ev,Ts), Ev initiates F, Ts < T, not broken(F,Ts,T)
broken(F,Ts,T) ← happens(Ev*,T*), Ev* terminates F, Ts < T* < T
```

This is a simplified form of the *event calculus*. More about event calculus (perhaps) and other temporal reasoning formalisms (definitely) later in the course.

Example Here is another possible treatment of default persistence ('inertia').

(This is just an example. It is not necessarily a good way of expressing default persistence 'inertial'.)

When `F` is a time-varying fact (or 'fluent'), let $\overline{\texttt{F}}$ represent the negation of `F`.
Let `given(F,T)` represent that fact `F` is known to hold at time `T`. (This could be because we know that an event which initiates `F` happens at time `T`, or because we observe that `F` is true at time `T` — whatever. `F` here could be a fluent or the negation of a fluent.)

A possible rule of default persistence ('inertia') — a 'frame axiom':

```
holds(F,T) ← given(F,T)
holds(F,T) ← holds(F,T'), next(T',T), not given(F̄,T)
```

For integer time, `next(T',T)` means `T = T' + 1`. `next` is more general.

Here is another possible rule:

```
holds(F,T) ← given(F,T)
holds(F,T) ← holds(F,T'), next(T',T), not holds(F̄,T)
```

(Compare the 'strong if big and not weak, weak if small and not strong' example earlier.)

Are these two formulations of persistence ('inertia') equivalent (in some sense)? If not, how are they related? (Ans: it depends in part on the structure of time — on what properties we assume for `next`.)

To answer these (and many other) questions we need better ways of characterising NBF. We consider some of the approaches to semantics of NBF next. Then we shall consider some more general approaches to formalisation of defeasible reasoning.