Complete Knowledge Assumption

- Often you want to assume that your knowledge is complete.
- Example: you can state what switches are up and the agent can assume that the other switches are down.
- Example: assume that a database of what students are enrolled in a course is complete.
- The definite clause language is monotonic: adding clauses can't invalidate a previous conclusion.
- Under the complete knowledge assumption, the system is non-monotonic: adding clauses can invalidate a previous conclusion.

Completion of a knowledge base

- Suppose the rules for atom a are

  $$a \leftarrow b_1.$$
  $$\vdots$$
  $$a \leftarrow b_n.$$

  equivalently $a \leftarrow b_1 \vee \ldots \vee b_n$.
- Under the Complete Knowledge Assumption, if a is true, one of the b_i must be true:

  $$a \rightarrow b_1 \vee \ldots \vee b_n.$$
- Under the CKA, the clauses for a mean Clark's completion:

  $$a \leftrightarrow b_1 \vee \ldots \vee b_n$$

Clark's Completion of a KB

- Clark's completion of a knowledge base consists of the completion of every atom.
- Clark's completion means that if there are no rules for an atom a, the completion of this atom is $a \leftrightarrow \mathit{false}$, which means that a is false.

Example of Clark's Completion

down-s1. up-s2.
live-l1 $\leftarrow$ live-w0.
live-w0 $\leftarrow$ live-w1 $\wedge$ up-s2.
live-w0 $\leftarrow$ live-w2 $\wedge$ down-s2.
live-w1 $\leftarrow$ live-w3 $\wedge$ up-s1.

The completion of these atoms is:

down-s1 $\leftrightarrow$ true.
up-s1 $\leftrightarrow$ false.
up-s2 $\leftrightarrow$ true.
down-s2 $\leftrightarrow$ false.
live-l1 $\leftrightarrow$ live-w0.
live-w0 $\leftrightarrow$ (live-w1 $\wedge$ up-s2) $\vee$ (live-w2 $\wedge$ down-s2).
live-w1 $\leftrightarrow$ live-w3 $\wedge$ up-s1.

This implies that up-s1 is false, and live-w1 is false.

Negation as Failure

- With the completion, the system can derive negations, and so it is useful to extend the language to allow negations in the body of clauses.
- The definition of a definite clause can be extended to allow literals in the body rather than just atoms.
- We write the negation of atom a under the complete knowledge assumption as $\sim a$ to distinguish it from classical negation that does not assume the completion.
- $\sim a$ means that a is false under the complete knowledge assumption. This is negation as failure.
- Clark's completion of an acyclic knowledge base is always consistent and always gives a truth value to each atom. We assume that the knowledge bases are acyclic.

Example

Consider the usual axiomatization of the circuits. Representing a domain can be made simpler by expecting the user to tell the system only what switches are up and by the system concluding that a switch is down if it has not been told the switch is up.

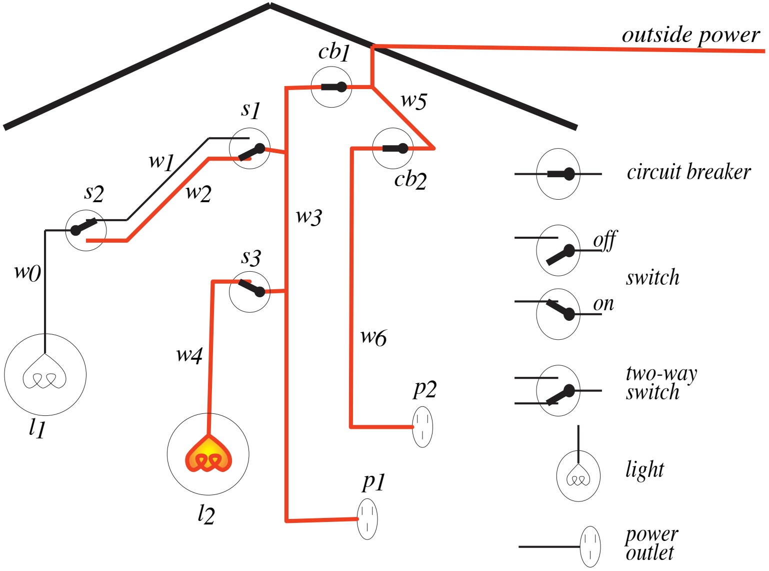
down-s1 $\leftarrow$ $\sim$up-s1.
down-s2 $\leftarrow$ $\sim$up-s2.
down-s3 $\leftarrow$ $\sim$up-s3.

The circuit breakers are okay unless it has been told they are broken:

ok-cb1 $\leftarrow$ $\sim$broken-cb1.
ok-cb2 $\leftarrow$ $\sim$broken-cb2.

The user has to specify only what is up and what is broken. This may save time if being down is normal for switches and being okay is normal for circuit breakers.

Electrical Environment



Representing the Electrical Environment (standard)

$light\_l_1.$

$light\_l_2.$

$down\_s_1.$

$up\_s_2.$

$up\_s_3.$

$ok\_l_1.$

$ok\_l_2.$

$ok\_cb_1.$

$ok\_cb_2.$

$live\_outside.$

$lit\_l_1 \leftarrow live\_w_0 \wedge ok\_l_1$

$live\_w_0 \leftarrow live\_w_1 \wedge up\_s_2.$

$live\_w_0 \leftarrow live\_w_2 \wedge down\_s_2.$

$live\_w_1 \leftarrow live\_w_3 \wedge up\_s_1.$

$live\_w_2 \leftarrow live\_w_3 \wedge down\_s_1.$

$lit\_l_2 \leftarrow live\_w_4 \wedge ok\_l_2.$

$live\_w_4 \leftarrow live\_w_3 \wedge up\_s_3.$

$live\_p_1 \leftarrow live\_w_3.$

$live\_w_3 \leftarrow live\_w_5 \wedge ok\_cb_1.$

$live\_p_2 \leftarrow live\_w_6.$

$live\_w_6 \leftarrow live\_w_5 \wedge ok\_cb_2.$

$live\_w_5 \leftarrow live\_outside.$

State with negation as failure

To represent the state, the user just specifies: up-s2. up-s3.

The system infers that s1 is down and both circuit breakers are okay.
The completion of the knowledge base is:

down-s1 $\leftrightarrow$ $\neg$up-s1.
down-s2 $\leftrightarrow$ $\neg$up-s2.
down-s3 $\leftrightarrow$ $\neg$up-s3.
ok-cb1 $\leftrightarrow$ $\neg$broken-cb1.
ok-cb2 $\leftrightarrow$ $\neg$broken-cb2.
up-s1 $\leftrightarrow$ false.
up-s2 $\leftrightarrow$ true.
up-s3 $\leftrightarrow$ true.
broken-cb1 $\leftrightarrow$ false.
broken-cb2 $\leftrightarrow$ false.

Non-monotonic Reasoning

- The (definite) clause logic is monotonic in the sense that anything that could be concluded before a clause is added can still be concluded after it is added; adding knowledge does not reduce the set of propositions that can be derived.
- A logic is non-monotonic if some conclusions can be invalidated by adding more knowledge.
- The logic of definite clauses with negation as failure is non-monotonic.
- Non-monotonic reasoning is useful for representing defaults.
- A default is a rule that can be used unless it overridden by an exception.

Defaults via absence of *anormality*

- To say that b is normally true if c is true, a knowledge base designer can write a rule of the form

$$b \leftarrow c \wedge \sim ab(a).$$

  where $ab(a)$ is an atom that means abnormal with respect to some aspect a.
- Given c, the agent can infer b unless it is told $ab(a)$. Adding $ab(a)$ to the knowledge base can prevent the conclusion of b.
- Rules that imply $ab(a)$ can be used to prevent the default under the conditions of the body of the rule.

Non-monotonic Reasoning: example

Suppose the purchasing agent is investigating purchasing holidays.
A resort may be adjacent to a beach or away from a beach.
If the resort was adjacent to a beach, the knowledge provider would specify this.

away-from-beach $\leftarrow$ $\sim$on-beach.

If the resort is on the beach, we expect that resort users would have access to the beach.
If they have access to a beach, we would expect them to be able to swim at the beach.

beach-access $\leftarrow$ on-beach $\wedge$ $\sim$ab(beach-access).
swim-at-beach $\leftarrow$ beach-access $\wedge$ $\sim$ab(swim-at-beach).

If there is an enclosed bay and a big city, then there is no swimming, by default:

ab(swim-at-beach) $\leftarrow$ enclosed-bay $\wedge$ big-city $\wedge$ $\sim$ab(no-swimming-near-city).

We could say that British Columbia is abnormal with respect to swimming near cities:

ab(no-swimming-near-city) $\leftarrow$ in-BC $\wedge$ $\sim$ab(BC-beaches).

Given only the preceding rules, an agent infers away-from-beach. If it is then told on-beach, it can no longer infer away-from-beach, but it can now infer beach-access and swim-at-beach. If it is also told enclosed-bay and big-city, it can no longer infer swim-at-beach. However, if it is then told in-BC, it can then infer swim-at-beach.

Computing negation as failure with bottom-up procedure

- The bottom-up procedure for negation as failure is a modification of the bottom-up procedure for definite clauses.
- The difference is that it can add literals of the form $\sim p$ to the set C of consequences that have been derived; $\sim p$ is added to C when it can determine that p must fail.
- Failure can be defined recursively:
  - p fails when every body of a clause with p as the head fails.
  - A body fails if one of the literals in the body fails.
  - An atom b_i in a body fails if $\sim b_i$ has been derived.
  - A negation $\sim b_i$ in a body fails if b_i has been derived.

Negation as failure example

- Consider the following clauses:

  $p \leftarrow q \wedge \sim r.$ $\quad p \leftarrow s.$ $\quad q \leftarrow \sim s.$

  $r \leftarrow \sim t.$ $\quad t.$ $\quad s \leftarrow w.$
- Suppose the query is **ask p**.
- Initially G={p}.
- Using the first rule for p, G becomes {q, ∼r}.
- Selecting q, and replacing it with the body of the third rule, G becomes {∼s, ∼r}.
- It then selects ∼s and starts a proof for s. This proof for s fails, and thus G becomes {∼r}.
- It then selects ∼r and tries to prove r. In the proof for r, there is the subgoal ∼t, and thus it tries to prove t. This proof for t succeeds. Thus, the proof for ∼t fails and, because there are no more rules for r, the proof for r fails. Thus, the proof for ∼r succeeds.
- G is empty and so it returns yes as the answer to the top-level query.

Bottom-up negation as failure interpreter

```
$C := \{\};$
repeat
    either
        select $r \in KB$ such that
            $r$ is "$h \leftarrow b_1 \wedge \ldots \wedge b_m$"
            $b_i \in C$ for all $i$, and
            $h \notin C$;
        $C := C \cup \{h\}$
    or
        select $h$ such that for every rule "$h \leftarrow b_1 \wedge \ldots \wedge b_m$" $\in KB$
                either for some $b_i, \sim b_i \in C$
                or some $b_i = \sim g$ and $g \in C$
        $C := C \cup \{\sim h\}$
until no more selections are possible
```

Top-Down negation as failure proof procedure

- The top-down procedure for the complete knowledge assumption proceeds by negation as failure.
- This is a non-deterministic procedure that can be implemented by searching over choices that succeed:
  - When a negated atom $\sim a$ is selected, a new proof for atom a is started.
    - If the proof for a fails, $\sim a$ succeeds.
    - If the proof for a succeeds, the algorithm fails and must make other choices.
  - Suppose you have rules for atom a:

    $$\begin{array}{l} a \leftarrow b_1 \\ \quad\vdots \\ a \leftarrow b_n \end{array}$$

    - If each body b_i fails, a fails.
    - A body fails if one of the conjuncts in the body fails.
- Note that you need *finite* failure. Example $p \leftarrow p$.