

COMP4415 Lecture Notes on Logic Programming

Michael Maher

May 18, 2009

1 Non-monotonicity

Classical logic is *monotonic* in the following sense:

$$\text{if } P \vdash q \text{ then } P \cup Q \vdash q$$

In other words, from more formulas we can only infer more things – nothing becomes un-inferable when we add Q to P .

But this property does not apply to many of the reasoning steps we apply in our everyday lives. For example, when I come to the lecture I expect some students will come. I can formulate this as a kind of inference:

$$\textit{lecture_time}, \textit{lecture_room} \vdash \textit{students}$$

But on a public holiday, if I come to the lecture I do not expect to see students. In this case

$$\textit{lecture_time}, \textit{lecture_room}, \textit{public_holiday} \not\vdash \textit{students}$$

Thus the inference relation \vdash is non-monotonic.

Many attempts at non-monotonic reasoning are based on the Closed World Assumption.

Definition 1 *The Closed World Assumption (CWA) is the inference rule: if $P \not\vdash q$ then infer $\neg q$.*

This inference rule is also called *Negation as Failure* because it infers a negated conclusion from the failure to prove the unnegated formula.

Notice that this principle can be applied to any inference system that defines \vdash . It is not restricted to the inference rules of classical logic.

In these notes we will first discuss applying the ideas behind the CWA when P is a set of definite clauses, and later extend the ideas to more general classes.

2 CWA for Definite Clause Logic Programs

2.1 Consistency

If we apply the CWA to a definite clause logic program P then the result is always consistent. We can be more precise. Let \vdash_{CWA} be the inference relation defined by the inference rules of classical logic augmented by the CWA.

Proposition 1 *Let P be a set of definite clauses, and let A be a ground atom. Then*

- $P \vdash_{CWA} A$ iff $A \in lm(P)$
- $P \vdash_{CWA} \neg A$ iff $A \notin lm(P)$

Thus the result of applying CWA to definite clauses is always sensible. The result is never inconsistent and always complete: for every ground atom A , either A or $\neg A$ is inferred. This appears the best of all possible worlds, but there are some complications.

2.2 Computability

Sebelik and Stepanek showed that every partial recursive function can be expressed as a logic program, by encoding standard rules for defining partial recursive functions¹ into definite clauses. Others have done this for other formulations of the computable functions, but I like this one. (The textbook (Ben-Ari) has a simple construction of a program P for every two-register machine M that computes the same function as M . See Theorem 5.43.)

Definition 2 *A program P is constructed according to the Sebelik-Stepanek scheme if, for every predicate f , the collection of rules for f has one of the following forms, there is no mutual recursion among predicates in P , each auxiliary predicate r_f is not used elsewhere, and no auxiliary predicate r_f syntactically depends on another auxiliary predicate r_h .*

- $f(X, 0)$.
- $f(X, s(X))$.
- $f(X_1, \dots, X_m, X_i)$.

¹The functions definable by these rules are called the μ -recursive functions. They are equivalent to the partial recursive functions, the Turing computable functions, ... According to the Church-Turing thesis, this set of functions is precisely the computable functions.

- $f(\bar{X}, Z) : -g_1(\bar{X}, Y_1), \dots, g_k(\bar{X}, Y_k), h(Y_1, \dots, Y_k, Z)$.
- $f(0, X_2, \dots, X_m, Z) : -g(X_2, \dots, X_m, Z)$.
 $f(s(X_1), X_2, \dots, X_m, Z) : -f(X_1, \dots, X_m, Y), h(X_1, \dots, X_m, Y, Z)$.
- $f(\bar{X}, Z) : -g(\bar{X}, 0, Y), r_f(\bar{X}, 0, Y, Z)$.
 $r_f(\bar{X}, U, 0, U)$.
 $r_f(\bar{X}, U, s(V), Z) : -g(\bar{X}, s(U), Y), r_f(\bar{X}, s(U), Y, Z)$.

where g, g_1, \dots, g_k, h are previously defined predicates, \bar{X} denotes the sequence of variables X_1, \dots, X_m , and the variables $U, V, X, X_1, \dots, X_m, Y, Y_1, \dots, Y_k, Z$ occurring in a rule are distinct.

It should be clear that the different forms of predicate definition correspond respectively to the zero function, the successor function, the projection functions, composition of functions, primitive recursion, and minimization. (The minimization operation finds the smallest value of Z such that $g(\bar{X}, Z, 0)$. This is not immediately obvious from the clauses.) The final restriction on auxiliary predicates ensures that the predicate g used in minimization is not defined using minimization, and so g represents a total function.

All partial recursive functions can be expressed as logic programs.

Theorem 1 *For each partial recursive n -ary function F there is a program P , constructed according to the Sebelik-Stepanek scheme, with least model $lm(P)$ and defining a $n+1$ -ary predicate f where $F(a_1, \dots, a_n) = b$ iff $f(a_1, \dots, a_n, b) \in lm(P)$.*

As a consequence, applying the closed world assumption to definite clause logic programs in general is impossible, assuming the Church-Turing thesis.

Corollary 1 *The problem of determining whether $A \notin lm(P)$ for arbitrary atom A and definite clause program P is not decidable. Thus the CWA is not computable.*

If we restrict our attention to propositional definite clauses then we don't have this problem, of course. In this case we can compute $lm(P)$ and all other atoms can be designated *false* by the CWA. If there are no function symbols in P then the problem is essentially propositional, because we can consider all ground instances of the clauses as a (much larger) set of propositional clauses.

There are two ways we can react to the previous Corollary:

- Use a weaker form of the CWA
- Only consider propositional reasoning

2.3 Clark Completion

The *Clark completion* is a weak form of CWA obtained by syntactic transformation of the clauses in a logic program P .

If the set of all rules of P with p in the head is

$$\begin{aligned} p(\tilde{x}) &\leftarrow B_1 \\ p(\tilde{x}) &\leftarrow B_2 \\ &\dots \\ p(\tilde{x}) &\leftarrow B_n \end{aligned}$$

then the formula associated with p is

$$\forall \tilde{x} p(\tilde{x}) \leftrightarrow \begin{aligned} &\exists \tilde{y}_1 B_1 \\ &\vee \exists \tilde{y}_2 B_2 \\ &\dots \\ &\vee \exists \tilde{y}_n B_n \end{aligned}$$

where \tilde{y}_i is the set of variables in B_i except for variables in \tilde{x} . If p does not occur in the head of a rule of P then the formula is

$$\forall \tilde{x} \neg p(\tilde{x})$$

The collection of all such formulas is called the *Clark completion* of P , and is denoted by P^* .

Each formula in P^* involving a predicate symbol p on the left can be thought of as having two parts, corresponding to the two arrows in \leftrightarrow :

- A slightly rewritten subset of P where p occurs positively in a clause. This subsets can be interpreted as rules saying “if (any of) these conditions hold then p holds”.
- A kind of converse of these clauses saying “only if (one of) these conditions hold will p hold”.

It is the second part that implements a weak version of the CWA.

P^* contains uses of equality that are not conditions, whereas P uses equality only as conditions. As a result we also need to define explicitly what we mean by equality. In this case, we want the equality to reflect the kind of equality that unification computes. So we include the following (infinite) set of equality axioms which we used earlier to reason about the correctness of the unification algorithm. Σ is the set of function symbols.

For every $f \in \Sigma$

$$\forall \tilde{x} \forall \tilde{y} f(\tilde{x}) = f(\tilde{y}) \leftrightarrow \tilde{x} = \tilde{y}$$

For every $f, g \in \Sigma, f \neq g$

$$\forall \tilde{x} \forall \tilde{y} f(\tilde{x}) \neq g(\tilde{y})$$

For every term $t(x, \tilde{y})$ containing x (except the term x)

$$\forall x, \tilde{y} x \neq t(x, \tilde{y})$$

We will refer to these axioms as *Clark's axioms* for equality, and denote them by E^* . There are also the usual axioms for equality (reflexivity, symmetry, transitivity).

The least Herbrand model of P is also the least Herbrand model of P^* . That means the transformation from P to P^* does not add any positive consequences, but does, in general, add some negative consequences.

Proposition 2 *Let P be a set of definite clauses, and P^*, E^* be the Clark completion of P . Then*

$$lm(P) = lm(P^*)$$

The Clark completion is a first-order theory, and hence all consequences can be generated, unlike the pure CWA.

2.4 SLD Finite Failure

Another weak form of the CWA exploits finite failure of SLD resolution. As observed earlier, if the execution of a goal clause $\leftarrow G$ is finitely failed we must have $P \not\vdash G$. When G is ground, we can infer $\neg G$ by the CWA. Thus this is a weak form of the CWA. It has the advantage that we are using the same mechanism for computing answers and for performing CWA inference.

It turns out that this form of CWA is equivalent to the Clark completion for definite clause programs.

Theorem 2 *Let P be a definite clause logic program and A be a ground atom.*

$$P^*, E^* \models A \quad \text{iff } A \text{ has a successful SLD derivation wrt } P.$$

$$P^*, E^* \models \neg A \quad \text{iff } A \text{ is finitely failed wrt } P.$$

Thus, when P is a definite clause program we have a method to partially decide whether A and $\neg A$ can be inferred from P^*, E^* .

3 CWA for General Logic Programs

In most of the following discussion, we will assume that we can (at least in theory) generate the set of ground instances of the clauses in a set P . We denote this set of ground clauses by $gd(P)$. Thus we are reducing the clauses to a large (possibly infinite) set of propositional clauses, since we can think of each ground atom as a proposition.

3.1 Logic Programs

It is worthwhile thinking about the advantages of definite clauses have when we try to give a meaning to logic programs. With definite clauses:

- The intersection of two models is a model.
- There is a least model that defines all the atoms that are true in all models.
- The least model can be generated (by unit resolution).
- We can test whether a ground atom is in the least model (using SLD resolution).
- There is a single set (model) that defines the meaning of a program

In general these properties will not apply to general clauses.

As a consequence, there is a problem when we use the CWA on more general clauses: the inference relation can produce inconsistency. For example, if P is the clause

$$A \vee B$$

then we have $P \not\models A$ and $P \not\models B$. By the CWA we can infer $\neg A$ and $\neg B$, but this is inconsistent with $A \vee B$.

It turns out that \vdash_{CWA} has consistent consequences only if P has a least model. So, although the CWA can be applied beyond definite clause programs, it cannot be used reliably without knowing a great deal about P , and then only for a very restricted class of programs. For more details on this point, read the note on CWA on the course website. For this reason we will no longer (almost) consider a set of clauses as the starting point for applying the CWA. Instead we will (mostly) insist on using *rules*. A rule has an atom as its head and a set of literals as its body.

Definite clauses are simple in a sense: because there is only one positive literal, there is only one way to write them as rules. So it made no difference whether we wrote them as sets of literals or as rules.

When we turn to general clauses there may be several ways write them as rules. In the semantics that follow, the different ways we write clauses as rules will generally result in different meanings. That means we have to choose which way to write clauses. This is more like programming than specifying.

For example, the clause $A \vee B$ can be represented as the rule $A \leftarrow \neg B$ or as $B \leftarrow \neg A$. It is a good exercise to check, for each of the semantics that follows, whether these two logic programs mean the same thing.

For the moment, the only allowable rules in logic programs have the form

$$A \leftarrow B_1, \dots, B_n, \neg C_1, \dots, \neg C_m$$

where A , B_i and C_i are atoms.

In some presentations, a different symbol than \neg is used to represent negation in rules like this, because this kind of negation does not have the same properties as classical negation. (The fact that rules derived from the same clause will have different meanings is a reflection of this fact.) We will need to do this later but, for the moment, we will stick with the standard negation symbol. Just remember, this is not classical negation.

3.2 Meaning of Rules

We can think of rules as an explicitly deductive approximation to the original clause, but the deduction can only possibly produce one result (in the propositional case): the head of the rule might be produced. Given this, a required property for any interpretation to be a model of the rule is that if the body of the rule is true in the interpretation, then the head must be deduced and so must be true in the interpretation. That is, the interpretation must be deductively closed, when thought of as a set.

A second principle requires that every atom true in the model must have a deductive reason for being true. That is, the atom must be the head of a rule whose body is true in the interpretation.

In terms of T_P these requirements on an interpretation I are

- $T_P(I) \subseteq I$
- $T_P(I) \supseteq I$

That is, I must be a fixedpoint of T_P .

Unfortunately, when rule bodies contain negated literals T_P is not monotonic, so we cannot directly employ the results on fixedpoints established earlier. For example, if P consists of the rule $a \leftarrow \neg a$ then $T_P(\emptyset) = \{a\}$ but $T_P(\{a\}) = \emptyset$.

Clearly T_P is not monotonic. This should not be surprising, given that we are trying to define/characterize non-monotonic reasoning.

In what follows, some of the semantics of rules will be obtained by restricting the set of fixedpoints of T_P considered “valid”. The greater the restriction, the stronger the inferences that the semantics justifies.

3.3 Minimal Models

In general, a set of general clauses does not have a least Herbrand model. But, inspired by the way that the least model of a definite clause logic program is the desired meaning of the program, we can try to use minimal models. A Herbrand interpretation M is a *minimal model* of P if M is a model of P and no strict subset of M is a model of P .

The Generalized Closed World Assumption (GCWA) is the inference rule that allows us to derive $P \vdash \neg A$ if A is not in any minimal model of P . The GCWA is different from the other forms of CWA below because it does not rely on the way clauses are written as rules.

3.4 SLDNF Resolution

When we consider a program as a set of rules (rather than clauses), each atom in the goal can only resolve with the head of a rule. However, a goal can also contain negative literals. Hence to define a resolution-based execution mechanism like SLD resolution we must define how to execute negative literals.

SLDNF resolution (SLD resolution with Negation as Failure) extends SLD resolution. If the selected literal is an atom then the atom is resolved against the head of a rule, just like SLD resolution. If the selected rule is a variable-free negated atom $\neg A$ (or is once the condition on the goal is taken into account) then what happens depends on the execution of the goal $\leftarrow A$. If $\leftarrow A$ succeeds then the goal containing $\neg A$ fails. If $\leftarrow A$ fails then by the CWA we can infer $\neg A$; consequently the goal $\leftarrow G_1, \neg A, G_2$ in which $\neg A$ appears can be simplified to $\leftarrow G_1, G_2$.

Notice that there are cases not covered by the above definition. If the definition is extended to apply to negative literals whether or not they are variable-free, problems arise. Consider the goal $\leftarrow \neg p(x), q(x)$ with program consisting of two facts $p(a)$ and $q(b)$.

If $\neg p(x)$ is selected then the goal $\leftarrow p(x)$ is executed. This goal succeeds, so $\leftarrow \neg p(x), q(x)$ fails. On the other hand, if $q(x)$ is selected first then unification results in $x = b$ and resolution reduces the goal to $\leftarrow \neg p(b)$. At the next step

$\neg p(b)$ must be selected, and so the goal $\leftarrow p(b)$ is executed. This goal fails, so $\leftarrow \neg p(x), q(x)$ succeeds with answer $x = b$.

If SLDNF execution reaches a goal where no literal can be selected (or where a non-ground negative literal is selected) the execution is said to *flounder*.

3.5 Clark Completion (again)

The Clark completion equally well applies to logic programs where negated atoms occur in the bodies of rules. Unfortunately, few of the results we showed for the Clark completion of definite clauses extends to more general classes of rules. One property that does continue to hold is that the Herbrand models of P^* are exactly the fixedpoints of T_P .

SLDNF resolution is a sound (but not complete) implementation of Clark's completion. In other words, SLDNF is a weaker form of CWA than Clark's completion.

Proposition 3 *If SLDNF resolution succeeds for goal $\leftarrow L$ where L is a ground literal then $P^*, E^* \models L$.*

3.6 Stratified Programs and Perfect Models

The idea of stratification is to divide a program into layers (or strata) such that every predicate depends negatively only on predicates in lower strata. I will define the notion for propositional programs, but it applies equally to sets of ground rules obtained from a non-propositional program. The definition is a bit complicated.

We first define what it means for an atom to be directly dependent on another, both positively and negatively. Let A and B range over atoms. $A \sqsupseteq_{+1} B$ if A appears in the head of a rule and B is a positive literal in the body of that rule. $A \sqsupseteq_{-1} B$ if A appears in the head of a rule and $\neg B$ is a negative literal in the body of that rule.

We say an atom is *defined* by a program P if it appears in the head of a rule in P .

Definition 3 *A program P is stratifiable if it can be partitioned into $P_0 \cup P_1 \cup \dots \cup P_k \cup \dots$ such that, every atom is defined in only one P_i and, for each i ,*

- *each atom defined in P_i is directly dependent negatively only on atoms defined in lower numbered partitions (that is defined in $\cup_{i=0}^{i-1} P_i$)*
- *each atom defined in P_i is directly dependent positively only on atoms defined in lower or equal numbered partitions (that is defined in $\cup_{i=0}^i P_i$)*

In this definition, we assume that atoms that are not defined by any rule are considered to be defined in P_0 .

We say that a program is stratified by a particular partition that satisfies these conditions.

If we know the status of some atoms, we can simplify a program. The following transformation is called the *Gelfond-Lifschitz transformation*.

Definition 4 Let P be a ground program and I be an interpretation. Let P^I denote the program obtained by

- (a) eliminating all rules $A \leftarrow B_1, \dots, B_k, \neg D_1, \dots, \neg D_m$ where $D_i \in I$ for some i , and
- (b) eliminating all negated atoms $\neg D_i$ where $D_i \notin I$ in the resulting program.

The standard model of a stratified program is computed, based on the stratification, layer by layer.

Definition 5 Let P be stratified into $P_0 \cup P_1 \cup \dots \cup P_k \cup \dots$.

The standard model is constructed iteratively according to the stratification.

Let M_0 be the least model of P_0 . For each $i > 0$, let M_i be the least model of $P_i^{N_{i-1}}$ where $N_i = M_0 \cup \dots \cup M_i$.

Then the standard model of P is $\cup_{i=0}^{\infty} M_i$.

Notice that, because of the stratification, the only negated atoms in P_i have been defined in lower strata, and N_i determines the truth value of these literals. So, for each i , the simplified program $P_i^{N_{i-1}}$ in the above definition contains only definite clauses, and hence the simplified program has a least model.

A program may have different stratifications, but they all compute the same standard model. Thus every stratified program has a unique standard model.

I am calling this standard model the *perfect model* because the standard model has a property that we will not discuss here.

The perfect model semantics is a refinement of the minimal model semantics. It also has the advantage that there is a single canonical model. On the other hand, for some programs a perfect model is not defined.

Proposition 4 Every perfect model is a minimal model.

In this semantics, and also in the following stable model semantics, we are using the existence of a least model of definite clause logic programs to “bootstrap” a canonical model of more general logic programs.

3.7 Stable Models

Definition 6 M is a stable model of P iff the least model of P^M is M .

If P is not a propositional program, then the stable models of P are the stable models of $gd(P)$.

The stable models are closely related to the other semantics for logic programs.

Proposition 5 Let P be a logic program.

- Every stable model of P is a minimal model of P and a model of P^* .
- If P is stratified then the perfect model of P is the unique stable model of P .
- If P is a set of definite clauses then the least model of P is the unique stable model of P .

The definition of the stable model does at one go what the definition of the perfect model does in stages. So it is clear that the stable models are a kind of generalization of perfect models. On the other hand, when it exists a perfect model is unique while a program may have several stable models. Furthermore, the definition of perfect model is constructive, whereas the definition of stable models is only by a property. That makes it hard to generate the stable models.

In the remainder of these notes we will write *not* A in place of $\neg A$.

4 Semantics with Partial Models

In general, we know that some atoms are true, others are false, and still others are unknown. So it seems reasonable to allow partial interpretations. We will represent these as a consistent set of literals (that is, no atom and its negation appear in such a set). Partial interpretations are ordered by set containment. A partial interpretation can be viewed as a 3-valued interpretation which maps atoms to either *true*, *false* or *unknown* (in symbols, **t**, **f** or **u**).

A literal L is true in an interpretation I iff $L \in I$, and is false in I iff $\sim L \in I$, where $\sim A$ is *not* A and \sim *not* A is A . A set of literals $\{L_1, \dots, L_n\}$ is true (respectively, false) in interpretation I if $\{L_1, \dots, L_n\} \subseteq I$ ($\{\sim L_1, \dots, \sim L_n\} \cap I \neq \emptyset$).

The meaning of the logical symbols \wedge , \vee , *not*, and \leftrightarrow must be extended to the 3-valued logic. Following from these definitions we can define what it means to be a 3-valued model, and to be a 3-valued logical consequence (\models_3).

\wedge	t	f	u
t	t	f	u
f	f	f	f
u	u	f	u

\vee	t	f	u
t	t	t	t
f	t	f	u
u	t	u	u

\leftrightarrow	t	f	u
t	t	f	f
f	f	t	f
u	f	f	t

\neg (<i>not</i>)	
t	f
f	t
u	u

4.1 Clark Completion (yet again)

Clark's completion can be viewed in 3-valued logic. There are many similarities to the treatment of Clark's completion in conventional 2-valued logic. An interpretation I is a partial model of ψ if ψ evaluates to **t** in I . We write $\psi \models_3 \phi$ if ϕ evaluates to **t** in every model of ψ .

We can define a version of the function T_P that applies to partial interpretations.

$$\Phi_P(I) = \begin{aligned} & \{A \mid \text{there is a rule } A \leftarrow B \text{ in } P \text{ where } B \text{ is true in } I\} \\ & \cup \{not A \mid \text{for every rule } A \leftarrow B \text{ in } P \text{ with head } A, B \text{ is false in } I\} \end{aligned}$$

Φ_P is monotonic, but not continuous. For example, let P be the program $q \leftarrow r(X); r(s(X)) \leftarrow r(X)$.

Now, let $I_n = \{not r(s^i(a)) \mid 0 \leq i \leq n\}$, where we assume, for convenience, that there is only one unary function symbol s and one constant a . Then $I = \cup_{n=0}^{\infty} I_n = \{not r(s^i(a)) \mid 0 \leq i\}$. $\Phi_P(I_n) = I_{n+1}$ and hence $\cup_{n=0}^{\infty} \Phi_P(I_n) = I$ but $\Phi_P(I) \neq I$ (in fact, $\Phi_P(I) = I \cup \{not q\}$).

Φ_P reflects a single iteration of unit resolution on the Clark completion P^* , in the same way that T_P reflects an iteration of unit resolution on P . The least fixedpoint $lfp(\Phi_P)$ is the smallest set of literals closed under unit resolution with P^* . Unfortunately, this set is not computable, in general, for non-propositional programs, although the subset $\Phi_P \uparrow \omega$ is computable. The least fixedpoint semantics is called the *Fitting* semantics, while a semantics based on $\Phi_P \uparrow \omega$ is called the *Kunen* semantics. Kunen's semantics characterizes 3-valued logical consequence and an idealized form of SLDNF resolution.

Theorem 3 *Let P be a logic program and A a ground atom.*

- *I is a 3-valued Herbrand model of P^* iff I is a fixedpoint of Φ_P*
- *$lfp(\Phi_P)$ is the least 3-valued Herbrand model of P^**
- *For every formula ψ , $\exists n \Phi_P \uparrow n \models \psi$ iff $P^*, E^* \models_3 \psi$*
- *$P^*, E^* \models_3 A$ iff the goal $\leftarrow A$ has a successful idealized SLDNF derivation*
- *$P^*, E^* \models_3 \text{not } A$ iff the goal $\leftarrow A$ has a finitely failed idealized SLDNF derivation*

4.2 Well-Founded Semantics

Semantics like Fitting's and Kunen's reflect unit resolution, but they are not able to capture inferences like: if P contains only $p \leftarrow p$ then conclude *not* p , which is the kind of inference that the CWA allows. The well-founded semantics extends the Fitting semantics with such inferences.

Definition 7 *Let I be an interpretation and P be a program. A set S of atoms is an unfounded set wrt I if for every atom $A \in S$, and every rule of the form $A \leftarrow B$ either*

- *B contains a literal that is false in I , or*
- *a positive literal in B occurs in S*

The idea of an unfounded set is a generalization of an unproductive recursive loop or infinite recursion. If P is $p \leftarrow q; q \leftarrow p$ then $\{p, q\}$ is an unfounded set for any interpretation I . If P is $p(X) \leftarrow p(s(X))$ then $\{p(s^n(a)) \mid n \geq 0\}$ is an unfounded set for any interpretation I . If P is $p \leftarrow q; q \leftarrow p; q \leftarrow r$ and I contains *not* r then $\{p, q\}$ is an unfounded set for I .

The union of any two unfounded sets is also unfounded. Thus there is a greatest unfounded set (wrt P and I). Denote this set by $U_P(I)$.

The function W_P is used to define the well-founded semantics.

$$W_P(I) = \begin{aligned} & \{A \mid \text{there is a rule } A \leftarrow B \text{ in } P \text{ where } B \text{ is true in } I\} \\ & \cup \{\text{not } A \mid A \in U_P(I)\} \end{aligned}$$

It is easy to see that W_P is monotonic.

Definition 8 *The well-founded semantics WFS_P is the least fixedpoint of W_P .*

Well-founded semantics is closely related to some of the other semantics we have studied.

Proposition 6 *Let P be a logic program.*

- *If P is stratified then the well-founded semantics is equivalent to the perfect model.*
- *$lfp(\Phi_P)$ is a subset of WFS_P .*
- *All stable models, when written as 3-valued models, contain the well-founded model.*

The last point suggests that the well-founded semantics is the part that is common to all stable models. This is not quite true, however. Consider the program P containing the rules $p \leftarrow \text{not } p; p \leftarrow q; r \leftarrow \text{not } q; q \leftarrow \text{not } r$. The well-founded semantics is the empty set, but P has a unique stable model $M = \{p, q, \text{not } r\}$.

5 Circumscription

Circumscription is a more sophisticated form of the CWA, and is similar in many ways to the Clark completion. Like these methods, circumscription attempts to minimize the extent of predicates, that is, minimize the number of tuples in the relation for each predicate. Clark's completion applies only to formulas in a limited form, and uses a logic program-based meaning of a rule, rather than a purely classical logic meaning. On the other hand, circumscription applies to any classical logic formula.

In outline, the circumscription of a predicate p in a formula ϕ expresses that any relation that satisfies the formula ϕ and is stronger than p , is equivalent to p . In symbols,

$$\forall r (\phi[p/r] \wedge \forall \tilde{x} r(\tilde{x}) \rightarrow p(\tilde{x})) \rightarrow \forall \tilde{x} p(\tilde{x}) \rightarrow r(\tilde{x})$$

We denote this formula by $Circ(\phi, p)$. This is a formula of second-order logic, because we are quantifying over all relations r . That makes it difficult to compute with directly, but we can find some of the consequences by choosing appropriate formulas defining possible values of r .

For example, let ϕ be $p(a) \vee p(b)$. Suppose we substitute $x = a$ in place of $r(x)$ in $Circ(\phi, p)$. Then we have the formula

$$((a = a \vee b = a) \wedge \forall x x = a \rightarrow p(x)) \rightarrow \forall x (p(x) \rightarrow x = a)$$

Simplifying, we have $p(a) \rightarrow \forall x (p(x) \rightarrow x = a)$. Similarly, we have $p(b) \rightarrow \forall x (p(x) \rightarrow x = b)$. So, as one consequence of $\phi \wedge \text{Circ}(\phi, p)$ we have

$$\forall x (p(x) \rightarrow x = a) \vee \forall x (p(x) \rightarrow x = b)$$

This says that the only possible relations that p can be are the relation with single tuple $\langle a \rangle$ or the relation with single tuple $\langle b \rangle$. In contrast, the CWA produces inconsistency on this example.

As another example, let ϕ be $(p(a) \leftarrow q(b)) \wedge (p(b) \leftarrow q(a))$. Suppose we substitute $(x = a \wedge q(b)) \vee (x = b \wedge q(a))$ in place of $r(x)$ in $\text{Circ}(\phi, p)$. Then in place of $\phi[p/r]$ we have the formula

$$((a = a \wedge q(b)) \vee (a = b \wedge q(a))) \leftarrow q(b) \wedge ((b = a \wedge q(b)) \vee (b = b \wedge q(a))) \leftarrow q(a)$$

which simplifies to *true*. Thus, $\text{Circ}(\phi, p)$ becomes

$$(true \wedge \forall x (x = a \wedge q(b)) \vee (x = b \wedge q(a)) \rightarrow p(x)) \rightarrow (\forall x p(x) \rightarrow (x = a \wedge q(b)) \vee (x = b \wedge q(a)))$$

Thus a consequence of $\phi \wedge \text{Circ}(\phi, p)$ is

$$\forall x p(x) \leftrightarrow (x = a \wedge q(b)) \vee (x = b \wedge q(a))$$

This is exactly what the Clark completion produces.

When the formula is a logic program that is not definite clauses then circumscription will produce weaker consequences than the Clark completion because the Clark completion is using the rule syntax to extract a more explicit meaning (and hence a stronger logical formula) than circumscription can.

The consequences of a circumscribed formula are sound wrt a form of the minimal model semantics.

Proposition 7 *Every model of ϕ that is minimal with respect to p is also a model of $\text{Circ}(\phi, p)$*

Circumscription can be extended in a number of ways: we can circumscribe several predicates simultaneously, we can let uncircumscribed predicates vary (rather than stay fixed), and we can put priorities on the circumscribed predicates, so that some predicates are minimized first, and others later.

6 Comparison

For definite clause logic programs the several semantics we developed were equivalent. For general logic programs, however, the different semantics are inequivalent. We can compare these semantics on several properties.

6.1 Definite Programs

The semantics we developed for general logic programs apply, in particular, to definite clause programs. In the first lecture on logic programming, we found

Theorem 4 *Let P be a definite clause program.*

$$SLD_P = U_P = lm(P) = lfp(T_P) = T_P \uparrow \omega$$

These equivalences describe only the positive consequences of a program. By the CWA, we can infer that the remaining atoms are false. We refer to this semantics as the *least model semantics*.

However, when we have general logic programs we are not always able to formulate their semantics as extensions of the least model semantics. For example, the program

$$p \leftarrow p$$

under the Clark completion, Kunen, and Fitting semantics does not entail that $\neg p$ is true, whereas the least model semantics does entail $\neg p$.

Several of the semantics reduce to the least model semantics in the following sense.

Theorem 5 *Let P be a definite clause program.*

$$lm(P) = \text{perfect model of } P = \text{unique stable model of } P = WFS_P$$

When the program consists of finitely many propositional definite clause there are more equivalences although, as the previous example shows, these semantics are not equivalent to the least model semantics. The Kunen, Fitting and Clark completion semantics agree on propositional definite programs.

Theorem 6 *If P is a finite propositional definite clause program. Let L be a literal.*

$$P^*, E^* \models L \text{ iff } P^*, E^* \models_3 L \text{ iff } L \in lfp(\Phi_P)$$

6.2 Inferential Power

Inferential power refers to the ability to infer conclusions from a program. Several of the semantics have already been established as refinements of another semantics. The refined semantics are stronger in the sense that they are able to infer more (or the same) conclusions from a given program than the other semantics.

For every semantics X we have defined, we can specify an inference relation $P \vdash_X L$ where P is a program and L is a literal. For example, we can define

inference in the Fitting semantics by $P \vdash_F L$ iff $L \in lfp(\Phi_P)$, and inference in the stable model semantics by $P \vdash_{SM} L$ iff L is true in all stable models of P . We say that a semantics S_1 has *greater (or equal) inferential power* than S_2 if

$$P \vdash_{S_2} L \text{ implies } P \vdash_{S_1} L$$

In this case S_1 can infer everything that S_2 can infer from P , and perhaps more.

Comparison of inference power is complicated by the fact that sometimes a semantics is undefined for a program (for example, if there is no stratification of P then the perfect model semantics is undefined; similarly, the Clark completion of P might be inconsistent). The perfect model semantics agrees with the well-founded and stable semantics, and has greater inferential power than the Kunen, Fitting, and GCWA semantics, but only provided P is stratified. The other semantics are defined on programs where the perfect model semantics is not defined.

In general, the Kunen semantics has weaker inferential power than the Fitting semantics which is, in turn, weaker than the well-founded semantics. These relationships follow almost directly from the definitions. In addition, when it is defined, the stable semantics has greater inferential power than the well-founded semantics.

GCWA has greater inferential power than circumscription. All other semantics require the program to be in rule form, but those rules can be interpreted as clauses so circumscription and GCWA apply indirectly. The perfect and stable models are also minimal models, so GCWA has weaker inferential power than these, when they are defined. Similarly, the (2-valued) Clark completion semantics is weaker than these semantics, but it is stronger than the Fitting semantics.

These are, I think, the only cases where there are inferential power relationships among the semantics we have discussed. Remaining pairs of semantics are incomparable, that is, for some programs S_1 infers literals that S_2 doesn't and for other programs S_2 infers literals that S_1 doesn't.

6.3 Computability and Complexity

For logic programs with negation, and for logic formulas in general, consider the different non-monotonic semantics we have discussed. The following table summarises the computability of these semantics when addressing first-order programs, the complexity of determining whether a given atom is true or false in a propositional program, and whether the semantics is guaranteed to be consistent or not.

Semantics	First-order Computable?	Propositional Complexity	Consistency Guaranteed?
Clark completion	Yes	co-NP-complete	No
Kunen	Yes	$O(n)$	Yes
Fitting	No	$O(n)$	Yes
Well-founded	No	$O(n^2)$	Yes
Stable	No	co-NP-complete	No
Perfect Models	No	$O(n)$	Yes
CWA	No	NP-complete	No
GCWA	No	Π_2^p -complete	No (FO)
Circumscription	No	NP-hard	No (FO)
Default Logic	No	Π_2^p -complete	No
Defeasible Logic	Yes	$O(n)$	No

In this table, the Clark completion is interpreted in 2-valued logic, and perfect models are only applicable to stratified programs (and we assume the stratification is given). n refers to the size of a program (the number of symbols in the program). The stable model semantics in general produces several models; the complexity refers to the complexity of determining whether a literal is true in all stable models. It is NP-hard to determine whether a stable model exists.

The GCWA infers $\neg A$ if A does not appear in any minimal model. If we only consider inferring atoms A true in all minimal models the propositional complexity is “only” co-NP-complete. The CWA, GCWA, and Circumscription apply to arbitrary first-order formulas, rather than logic programs. The GCWA and Circumscription are consistent on propositional formulas, but can be inconsistent on first-order formulas.

Default Logic and Defeasible Logic will be discussed in the next lecture. Inconsistency in Defeasible Logic can only arise from the monotonic part of that logic; the non-monotonic part will never derive both an atom and its negation unless the monotonic part has derived such literals.

Complexity of Clark Completion We reduce the SAT problem to finding a model of the Clark completion of a program. For each clause $C = l_1 \vee l_2 \vee l_3$, include in the program: $p_C \leftarrow \neg p_C$ and $p_C \leftarrow l_i$, for $i = 1, 2, 3$. For each propositional variable q in the 3-SAT problem, add to the program $q \leftarrow q$. The completion of p_C is satisfied iff $l_1 \vee l_2 \vee l_3$. The completion of the q 's does not restrict their possible values.

Complexity of Stable Models Determining whether a program has a stable model is NP-complete. We reduce graph 3-colourability to the problem of finding

a stable model. Given a graph, we represent it by unit clauses. For each vertex named v , we have the unit clause $\text{vertex}(v)$ and for each edge (v_1, v_2) we have the unit clause $\text{edge}(v_1, v_2)$.

We also use unit clauses to specify the colours.

```
colour(red).
colour(green).
colour(yellow).
```

Let U denote the set of all these unit clauses. The following rules represent the set of all their ground instances.

```
% every vertex is coloured
coloured(V, red) ← vertex(V),
    not coloured(V, green), not coloured(V, yellow).
coloured(V, green) ← vertex(V),
    not coloured(V, red), not coloured(V, yellow).
coloured(V, yellow) ← vertex(V),
    not coloured(V, green), not coloured(V, red).

% no vertex has 2 (or more) colours
← vertex(V), colour(C1), colour(C2),
    C1 ≠ C2, coloured(V, C1), coloured(V, C2).

% adjacent vertices have different colours
← vertex(V1), vertex(V2), edge(V1, V2), colour(C),
    coloured(V1, C), coloured(V2, C).
```

The entire program is P .

Let $S = U \cup X$ where X is a set of coloured atoms. If X defines a 3-colouring of the graph then every vertex is coloured by a single colour and adjacent edges have different colours. Hence, for every vertex v coloured red, say, P^S contains $\text{coloured}(v, \text{red}) \leftarrow \text{vertex}(v)$ and so every atom in X is in $\text{lm}(P^S)$. The bodies of rules with empty head are not satisfied, since X defines a 3-colouring, and hence S is a stable model.

If S is a stable model then it is easy to see that X defines a 3-colouring of the graph, following the comments in the program.

7 Extensions of Logic Programs

We can extend the idea of stable models to programs that include rules without heads.

$$\leftarrow B_1, \dots, B_k, \text{not } C_{k+1}, \dots, \text{not } C_n$$

Such rules get simplified by an interpretation in the same way as other rules, and result in negative clauses. There are two possibilities

- there is no model of P^I
In this case, I is not a stable model.
- there is a model of P^I
Then the least model of P^I satisfies the negative clauses. In this case, I is a stable model iff $I = \text{lm}(P^I)$, as usual.

Rules without heads are convenient for programming, because they allow us to say explicitly that certain formulas must not be true.

Alternatively, we can consider such a rule as an abbreviation of

$$Q \leftarrow \text{not } Q, B_1, \dots, B_k, \text{not } C_{k+1}, \dots, \text{not } C_n$$

where Q is a new symbol, used only in this rule.

This rule can only be satisfied when Q is false, and then only if $B_1, \dots, B_k, \text{not } C_{k+1}, \dots, \text{not } C_n$ evaluates to false. Thus this implementation is equivalent to the meaning described above.

7.1 Extended Logic Programs

The negation as failure is only a weak form of negation: an atom is false if it cannot be proved to be true. We now introduce a second kind of negation that is closer to classical negation. We will use \neg for the second kind, while continuing to use *not* for negation as failure. The second kind of negation is sometimes called *explicit negation* or *classical negation*.

Thus program rules have the form

$$L \leftarrow L_1, \dots, L_k, \text{not } L_{k+1}, \dots, \text{not } L_n$$

where L and L_1, \dots, L_n are either atoms A or negated atoms $\neg A$. Programs with such rules are called *extended logic programs*.

We will follow the same approach as the stable models to give a semantics to extended logic programs P . First, if there are no occurrences of negation as failure (*not*), we can treat the rules as inference rules, and take the deductive closure

generated by these rules. This can be viewed as unit resolution on the rules, but where we treat A and $\neg A$ as different names for atoms, and only the *not* negation is used for resolution. The only issue is that (unlike definite clauses in the definition of stable models) it is possible to generate an inconsistency (i.e. both A and $\neg A$, for some atom A). If the deductive closure contains an inconsistency, then we replace it by the set of all literals, both positive and \neg -negative. We call this set *Lit*. In any case, the resulting set of literals is called the *answer set* of P , and denoted by $\alpha(P)$. By definition, any program without negation as failure has a unique answer set.

Second, if there is negation as failure in the rules then we can simplify the rules to eliminate negation as failure. Given a set $S \subset \text{Lit}$, the simplification of P wrt S , denoted by P^S , is obtained from P by

- (a) eliminating all rules $L \leftarrow L_1, \dots, L_k, \text{not } L_{k+1}, \dots, \text{not } L_n$ where $L_i \in S$ for some i in $k+1, \dots, n$, and
- (b) eliminating all negated literals $\text{not } L_i$ where $L_i \notin S$ in the resulting program.

Definition 9 *Let P be a logic program with both kinds of negation. S is an answer set of P iff $S = \alpha(P^S)$.*

The definition of answer sets closely parallels the definition of stable models so that we can compute answer sets by computing stable models of the related program where literals like $\neg A$ are translated into atoms not_A .

7.2 Disjunctive Logic Programs

A further extension is to allow disjunction in the head of rules. In this case rules have the form

$$L_1 | \dots | L_k \leftarrow L_{k+1}, \dots, L_m, \text{not } L_{m+1}, \dots, \text{not } L_n$$

where L_1, \dots, L_n are either atoms A or negated atoms $\neg A$. Here we use the vertical bar $|$ for disjunction.

For programs without negation as failure, an answer set is a minimal subset S of *Lit* such that

- for a rule $L_1 | \dots | L_k \leftarrow L_{k+1}, \dots, L_m$ if $\{L_{k+1}, \dots, L_m\} \subseteq S$ then at least one of L_1, \dots, L_k is in S
- if S contains A and $\neg A$, for some atom A , then $S = \text{Lit}$

This time there may be several answer sets for programs without negation as failure. For example, the one-rule program $A|B \leftarrow$ has two answer sets: $\{A\}$ and $\{B\}$, corresponding to the two minimal models of $A \vee B$.

The general definition of answer sets simply extends earlier definitions. Given a set $S \subset Lit$, the simplification of P wrt S , denoted by P^S , is obtained from P by

- (a) eliminating all rules $L_1 | \dots | L_k \leftarrow L_{k+1}, \dots, L_m, not L_{m+1}, \dots, not L_n$ where $L_i \in S$ for some i in $k+1, \dots, n$, and
- (b) eliminating all negated atoms $not L_i$ where $L_i \notin S$ in the resulting program.

Definition 10 *Let P be a logic program with disjunction and both kinds of negation. S is an answer set of P iff S is an answer set of P^S .*

8 Default Logic

Default logic is an extension of classical logic with default rules. A *default theory* is a pair (W, D) where W is a set of first-order formulas and D is a set of default rules. A *default rule* (or simply a *default*) has the form

$$\frac{p : q_1, \dots, q_n}{r}$$

where p, q_1, \dots, q_n, r are first-order formulas. p is called the *prerequisite*, q_1, \dots, q_n are called the *justifications* and r is called the *consequent*. The intended meaning of a default is that if p is true and if q_i is consistent with current knowledge, for each i , then we can infer r .

Although default logic is a first-order logic, we will discuss it mainly as a propositional logic.

Given a set E of formulas, a default rule is applicable if $E \vdash p$ and $E \not\vdash \neg q_i$ for $i = 1, \dots, n$, where \vdash is classical inference.

Starting from a base of knowledge W , we can infer more formulas using the default rules. Repeatedly applying default rules until no more can be inferred, and closing under classical consequence, we reach a set E . However, we want to be sure that the justifications that are consistent when a default is applied, are still consistent after more defaults have been applied. Thus we also require that all justification used in deriving E are consistent with E . Any set that satisfies these criteria is called an *extension*. The semantics of a default theory is its set of extensions.

A default theory does not necessarily have an extension. Consider the default theory (\emptyset, D) , where D contains the single default:

$$\frac{\text{true} : q}{\neg q}$$

This default theory has no extension because the generated set of formulas $\{\neg q\}$ is not consistent with the justification q .

The order in which default rules are applied influences which extension is generated. For example, suppose $W = \{p\}$ and D contains

$$\frac{p : q}{q} \qquad \frac{p : \neg q}{\neg q}$$

then, depending on which default rule is applied first, the extension generated is either $\{p, q\}$ or $\{p, \neg q\}$.

If we wish to have several prerequisites or several consequents we can use a conjunction of formulas. But there is a difference between the use of several justifications and a single justification that is a conjunction of the formulas. Consider the defaults

$$\frac{\text{true} : \neg p, \neg q}{r} \qquad \frac{\text{true} : \neg p \wedge \neg q}{r}$$

and let $W = \{p \vee q\}$. Both $\neg p$ and $\neg q$ are consistent with W , so r can be inferred by the first default. However $\neg p \wedge \neg q$ is not consistent with W , so r cannot be inferred by the second default.

For each logic programming rule

$$A \leftarrow B_1, \dots, B_k, \text{not } C_{k+1}, \dots, \text{not } C_n$$

there is a corresponding default rule that means something similar

$$\frac{B_1 \wedge \dots \wedge B_k : \neg C_{k+1}, \dots, \neg C_n}{A}$$

There is a close relationship between stable models and extensions.

Proposition 8 *Given a program P , let D be the set of corresponding defaults. Let S be a set of atoms.*

S is a stable model of P iff S is an extension of the default theory (\emptyset, D) .

A normal default has the form

$$\frac{p : q}{q}$$

where the justification and the consequent are the same. Thus a normal default uses the justification only to ensure that it is consistent to infer the consequent. A default theory is normal if every default is normal.

Proposition 9 *Every normal default theory has an extension.*

9 Defeasible Logic

Defeasible logic is a rule-based logic that makes a clear distinction between conclusions that are drawn with certainty and conclusions that are defeasible (i.e. may be overturned as a result of more information). It also is explicit about failure-to-prove whereas the logics we have seen so far use negation-as-failure without representing failure itself. Finally, there are many variants of defeasible logic; we will only define one. To keep things simple, we will consider only propositional defeasible logic and a simplified form of the inference rules.

Rules have a head (which is a literal), a body (a set of literals), and an arrow (which is either \rightarrow or \Rightarrow). A rule may also have a label, so we can refer to the rule. Rules built using \Rightarrow are *defeasible* rules while rules built using \rightarrow are *strict* rules. Strict rules are used for inferences that are without doubt (i.e. traditional deductions) involving absolute facts and statements true by definition. Defeasible rules are used when the inference is plausible/likely/expected/usual/... but might not always be valid.

For example, we may have the program

$$\begin{array}{l}
 hawk hawk \\
 hawk bird \\
 brokenWing \\
 r : bird flies \\
 r' : brokenWing \neg flies
 \end{array}$$

Notice that we have some definite information about the object under discussion (that it is a hawk, has a broken wing, and that all hawks are birds) and some defeasible information (that birds generally fly, and things with broken wings generally do not fly). Rules r and r' potentially conflict, so defeasible logic also allows us to state which rule should be given preference if they do conflict. We write

$$r' > r$$

to say that r' over-rides r . This preference ordering of rules is required to be a partial order (no cycle of preferences) and, in general, is not a linear/total order.

From the above program we should be able to deduce that the object under discussion is most likely unable to fly. But we need to be able to express such a statement and know the inference rules first.

A conclusion is a literal with a tag. The tag expresses the kind of conclusion we come to about the literal. There are four kinds:

$+\Delta q$ which is intended to mean that q is definitely provable in D (i.e., using only strict rules).

$-\Delta q$ which is intended to mean that it is proved that q is not definitely provable in D .

$+\partial q$ which is intended to mean that q is defeasibly provable in D .

$-\partial q$ which is intended to mean that it is proved that q is not defeasibly provable in D .

So we would expect conclusions $+\partial\neg flies$ and $-\Delta flies$, among others, as a result of the above program.

9.1 Inference Rules

There is an inference rule for each tag.

$$\begin{array}{ll}
 +\Delta: & -\Delta: \\
 \text{Conclude } +\Delta q \text{ if} & \text{Conclude } -\Delta q \text{ if} \\
 \exists r \in R_s[q] & \forall r \in R_s[q] \\
 \forall a \in B(r) : +\Delta a & \exists a \in B(r) : -\Delta a
 \end{array}$$

These are the inference rules for tags concerning definite conclusions. The inference rule for $+\Delta$ is standard rule application, while the inference rule for $-\Delta$ is the strong negation of the one for $+\Delta$. Strong negation uses DeMorgan-like laws on logical operators and quantifiers and replace a tagged literals by the literal tagged with the opposite polarity (in this case replaces $+\Delta a$ by $-\Delta a$). $R_s[q]$ denotes the set of all strict rules with head q .

The inference rules for defeasible conclusions are more complicated because the inference rules balance different possibilities. Again, the rule for $+\partial$ is the strong negation of the rule for $-\partial$. In these inference rules we ignore the possibility of using the preferences among rules.

$$\begin{array}{ll}
 +\partial: & -\partial: \\
 \text{Conclude } +\partial q \text{ if either} & \text{Conclude } -\partial q \text{ if} \\
 (1) +\Delta q \text{ or} & (1) -\Delta q \text{ and} \\
 (2.1) \exists r \in R[q] \forall a \in B(r) & (2.1) \forall r \in R[q] \exists a \in B(r) : \\
 \quad +\partial a \text{ and} & \quad -\partial a \text{ or} \\
 (2.2) -\Delta \sim q \text{ and} & (2.2) +\Delta \sim q \text{ or} \\
 (2.3) \forall s \in R[\sim q] & (2.3) \exists s \in R[\sim q] \text{ such that} \\
 \quad \exists a \in B(s) : -\partial a & \quad \forall a \in B(s) : +\partial a
 \end{array}$$

$R[q]$ denotes the set of all strict or defeasible rules with head q . \sim is defined by $\sim\neg q = q$, $\sim q = \neg q$. The inference rule for $+\partial$ says that we can conclude $+\partial q$ (that q is defeasibly true) if either

- we have already concluded that q is definitely true ($+\Delta q$), or
- - there is a rule we can use to conclude q , and
 - we have already concluded that $\sim q$ is not definitely true, and
 - for every rule s for $\sim q$ we have already discovered a reason ($-\partial a$) why that rule cannot be used to conclude $\sim q$

We can consider defeasible logic as a kind of argumentative logic in the sense that rules provide different arguments for a literal to be considered *true/false* and the inference rules formulate the criteria for a winning argument.

9.2 Preferences on Rules

When there are several reasons to conclude q and several reasons to conclude $\sim q$ the logic must decide which reasons prevail (or that none prevail). The preference on rules can be used. One approach is that if one rule for q is preferred over all applicable rules for $\sim q$ then q is concluded.

Another approach is to consider the two sets of rules as *teams*. The team that “wins” is one that, for every rule of the opposite team, there is a more preferred rule on the winning team.

It turns out that preferences on rules can be encoded with defeasible rules, which is part of the reason why we use only the simplified inference rules above.

9.3 Ambiguity

When there are reasons to conclude q and reasons to conclude $\sim q$ we say q is *ambiguous*. There is a tricky question of how much we can conclude further from an ambiguous literal. There are two schools of thought: *ambiguity propagating* and *ambiguity blocking*. Consider the following example.

$$\begin{array}{ll} \Rightarrow a & \Rightarrow b \\ \Rightarrow \neg a & a \Rightarrow \neg b \end{array}$$

Here a is ambiguous since we have applicable rules for both a and $\neg a$, and we have no means to decide between them. In a setting where the ambiguity is blocked, b is not ambiguous because we have an applicable rule for b and, at the same time, the rule for $\neg b$ is not applicable since we cannot prove its antecedent. On the other hand, in an ambiguity propagating setting, b is ambiguous because there are rules for both b and $\neg b$, the antecedent of the rule for $\neg b$ is ambiguous and hence the ambiguity is propagated to b .

Using the above inference rules, we have proofs in this program for $-\partial a$, $-\partial\neg a$, $+\partial b$, and $-\partial\neg b$, thus showing the ambiguity blocking behavior of defeasible logic.