CS720 Class Notes

Steve Revilak

Jan 2007 – May 2007

This are Stephen Revilak's course notes from CS720, Logical Foundations of Computer Science. This course was taught by Professor Peter Fejer at UMass Boston, during the Spring 2007 Semester.

Part 1

Propositional Logic

1.1 Lecture – Logic Review – 1/29/2007

1.1.1 A review of Logic

What is the aim of logic? We want to develop formal languages to model situations encountered in computer science.

A few branches of logic:

- Propositional logic (traditional)
- First-order logic (traditional)
- Modal logic (developed by philosophers)

1.1.2 Propositional Logic

In propositional logica, formulas are meant to represent statements in a symbolic way. Examples:

- If it's raining, then the sidewalks are wet
- If you study hard, then you'll pass the course

Note that these examples are "if-then" constructs. In propositional logic, we write this as $p \to q$.

Let's apply symbols to one of our examples

1	If it's raining, the sidewalks are wet	$p \to q$
2	The sidewalks are not wet	$\neg q$
3	It is not raining	$\neg p$

1.1.3 Formulas in Propositional Logic

First we have the notion of *propositional atoms*. Propositional atoms are similar to variables. We'll typically use single letters, or single letters with subscripts: P, Q, R, P_1, P_2, P_3, etc.

The definition of a propositional logic formula is inductive. Note the use of parenthesis in the definitions below.

1. Every propositional atom is a formula

2. If ϕ is a formula, then so is $(\neg\phi)$.
3. If ϕ and ψ are formulas, then so is $(\phi \vee \psi)$.
4. If ϕ and ψ are formulas, then so is $(\phi \wedge \psi)$.
5. If ϕ and ψ are formulas, then so is $(\phi \rightarrow \psi)$.

Suppose we wanted to give a rigorous proof that

$$((p \rightarrow q) \vee (r \wedge s))$$

were a valid formula.

Proof

1	p is a formula	Rule 1
2	q is a formula	Rule 1
3	$(p \rightarrow q)$ is a formula	Rule 5, lines 1, 2
4	r is a formula	Rule 1
5	s is a forumla	Rule 1
6	$(r \wedge s)$ is a formula	Rule 4, Lines 4, 5
7	$((p \rightarrow q) \vee (r \wedge s))$ is a formula	Rule 3, lines 3, 6

An example of something that *isn't* a formula by the definitions given

$$(\neg p \vee q)$$

Informally, we'd treat this as $((\neg p) \vee q)$, but it doesn't fit the formal definition.

Precedence in Propositional Logic

The order of precedence is

$$\begin{array}{c} \neg \\ \vee,\ \wedge \\ \rightarrow \end{array}$$

Precedence allows us to write

$$\neg p \vee q \rightarrow r \wedge s$$

which is equivalent to the formal

$$(((\neg p) \vee q) \rightarrow (r \wedge s))$$

1.1.4 Syntax vs. Semantics

Convention: Upper-case greek letters represent sets of formulas, while lower-case greek letters represent single formulas.

Consider the following two notations

$$\Gamma \vdash \phi$$
$$\Gamma \vDash \phi$$

In each of these cases ϕ is a single formula and Γ is a set of formula. (Γ may be an empty set).

- $\Gamma \vdash \phi$ means that ϕ can be derived from from formulas in Γ by using some formal proof system.
- $\Gamma \vDash \phi$ means that ϕ follows logically from Γ. For every situation where Γ holds, ϕ holds.
- $\Gamma \vdash \phi$ is a syntactic definition
- $\Gamma \vDash \phi$ is a semantic definition

In many logic systems, we desire the following:

$$\Gamma \vdash \phi \text{ IFF } \Gamma \vDash \phi \tag{1.1.1}$$

Equation (1.1.1) can be broken into two components:

$$\text{If } \Gamma \vdash \phi, \text{ then } \Gamma \vDash \phi \qquad \text{This is called } \textbf{soundness} \tag{1.1.2}$$

$$\text{If } \Gamma \vDash \phi, \text{ then } \Gamma \vdash \phi \qquad \text{This is called } \textbf{completeness} \tag{1.1.3}$$

- **Soundness** is a syntactic quality, and the more important of the two. Soundness means that when you prove something, the result really does follow. Soundness allows you to trust the proof system.
- **Completeness** is a semantic quality. Completeness is useless without soundness.

1.1.5 Natural Deduction

Natural Deduction is a system due to Gentzen. It's a syntactic system ($\Gamma \vdash \phi$).

The following is referred to as a *sequent*

$$\phi_1, \phi_2, \ldots, \phi_n \vdash \psi \tag{1.1.4}$$

A sequent is "valid" if one can derive ψ from the premises $\phi_1, \phi_2, \ldots, \phi_n$ using the rules of natural deduction.

The general idea – we will have introduction rules and elimination rules for each connective. Introduction rules allow a connective to be used; elimination rules allow a connective to be removed.

Conjunction ($\wedge$)

$$\frac{\phi, \psi}{\phi \wedge \psi} \qquad \text{Rule: } \wedge_i \tag{1.1.5}$$

$$\frac{\phi \wedge \psi}{\phi} \qquad \text{Rule: } \wedge_{e_1} \tag{1.1.6}$$

$$\frac{\phi \wedge \psi}{\psi} \qquad \text{Rule: } \wedge_{e_2} \tag{1.1.7}$$

In the equations above, the subscript i denotes *introduction* and the subscript e denotes *elimination*.

Example 1.1.1: prove the following

$$P \wedge Q,\ R \vdash P \wedge R$$

Proof.

1	$P \wedge Q$	Premise
2	R	Premise
3	P	$\wedge_{e_1}$, Line 1
4	$P \wedge R$	$\wedge_i$, Lines 3, 2

It's also possible to represent such proofs as a tree.

(Note to self, try installing
`http://www.phil.cam.ac.uk/teaching_staff/Smith/LaTeX/nd.html` which does the tree representations natively.

Double Negation ($\neg\neg$)

$$\frac{\neg\neg\phi}{\phi} \qquad \text{Rule: } \neg\neg_e \tag{1.1.8}$$

$$\frac{\phi}{\neg\neg\phi} \qquad \text{Rule: } \neg\neg_i \tag{1.1.9}$$

Example 1.1.2: Prove

$$P, (\neg\neg Q \wedge R) \vdash \neg\neg P \wedge Q \tag{1.1.10}$$

PROOF:

1	P	Premise
2	$(\neg\neg Q \wedge R)$	Premise
3	$\neg\neg Q$	$\wedge_{e_1}$, Line 2
4	Q	$\neg\neg_e$, Line 3
5	$\neg\neg P$	$\neg\neg_i$, Line 1
6	$\neg\neg P \wedge Q$	$\wedge_i$, lines 5, 4

Implication

$$\frac{\phi,\ \phi \to \psi}{\psi} \qquad \text{Rule: } \to_e \text{, Modus ponens} \tag{1.1.11}$$

$$\frac{\neg\psi,\ \phi \to \psi}{\neg\phi} \qquad \text{Rule: Modus Tollens (MT)} \tag{1.1.12}$$

1.1.6 Proof Boxes

When doing proofs, we will use boxes to make temporal assumptions. For example

$$\frac{\boxed{\begin{array}{c}\phi \\ \vdots \\ \psi\end{array}}}{\phi \to \psi} \qquad \text{Rule: } \to_i \tag{1.1.13}$$

In (1.1.13), we assume that ϕ is true, and from this assumption derive ψ. The first and last lines of the box form an implication, $\phi \to \psi$.

Rules for boxes:

- The first line of the box must introduce a temporal assumption. This assumption is not a premise. It is only valid within the box.
- Boxes can be opened at any time (but they must nest properly)
- All boxes must be closed before the last line of the proof.
- In justifying a proof line, one cannot use a previous box that has closed already. (Think of it like this: assumptions have lexical scope in which they are valid)

- The line immediately following a closed box has to match the pattern of the conclusion of the rule that uses the box.

Example 1.1.3:

1 Outside the box

2	a line
3	First line of A
4	⋮
5	Last line of A
6	First line of B
7	⋮
8	Last Line of B
9	First line of C
10	⋮
11	Last Line of C

12 The end

In this example, lines inside C cannot reference lines inside A or B. However, C can reference lines 1 and 2.

(For typesetting, we make use of the `proofbox` package, `http://www.cs.man.ac.uk/~pt/proofs/`)

Example 1.1.4: Another example, but with a real proof

$$P \to Q \vdash \neg Q \to \neg P$$

PROOF:

1	$P \to Q$	premise
2	$\neg Q$	assumption
3	$\neg P$	Modus Tollens. 1,2
4	$\neg Q \to \neg P$	$\to_i$, 2-3

Example 1.1.5: Another example

$$\vdash P \to P$$

PROOF:

1	P	assumption
2	$P \to P$	$\to_i$. 1–1

These kinds of boxed assumptions will often be used to introduce implication.

1.1.7 Theorems

Given the form

$$\Gamma \vdash \phi$$

where Γ is an empty set, we have the construct

$$\vdash \phi$$

In this context ϕ is referred to as a *theorem*.

Example 1.1.6: Prove

$$\vdash (Q \to R) \to ((\neg Q \to \neg P) \to (P \to R))$$

Proof:

1	$Q \to R$	assumption
2	$\neg Q \to \neg P$	assumption
3	P	assumption
4	$\neg\neg P$	$\neg\neg i$. 3
5	$\neg\neg Q$	MT. 2,4
6	Q	$\neg\neg e$. 5
7	R	$\to e$ 1,6
8	$P \to R$	$\to i$, 3–7
9	$(\neg Q \to \neg P) \to (P \to R)$	$\to i$. 2–8
10	$(Q \to R) \to ((\neg Q \to \neg P) \to (P \to R))$	$\to i$. 1–9

In general, we can transform an equation like

$$\phi_1, \phi_2, \ldots, \phi_n \vdash \psi$$

Into

$$\vdash \phi_1 \to (\phi_2 \to \ldots \to (\phi_n \to \psi))$$

Think about this for the next lecture.

1.2 Lecture – 1/31/2007

1.2.1 Syntax and Semantics

The formula

$$\phi_1, \ldots, \phi_n \vdash \psi$$

is a *syntactic* representation. Starting with premises $\phi_1, \ldots, \phi_n$, we apply formal rules to derive ψ. This deriviation is done without regard to any assignment of truth values in the formulas.

Assignment of truth values is a *semantic* representation.

1.2.2 Implications and Assumptions

For proofs that involve implication, a general strategy is as follows:

- Use elimination rules to deconstruct assumptions that have been made.
- Use introduction rules to construct the final conclusion.

1.2.3 Disjunction ($\vee$)

For disjunction, we have two introduction rules

$$\frac{\phi}{\phi \vee \psi} \qquad \text{Rule: } \vee i_1 \tag{1.2.1}$$

$$\frac{\psi}{\phi \vee \psi} \qquad \text{Rule: } \vee i_2 \tag{1.2.2}$$

Eliminating disjunctions is a little harder. Suppose we have $\phi \vee \psi$ and wish to prove χ. Because we don't know *which* of ϕ or ψ is true, we have to show both cases. All told, there will be three parts to or-elimination.

- $\phi \vee \psi$
- ϕ true makes χ true
- ψ true makes χ true

The rule for disjunction elimination is

$$\frac{\phi \vee \psi \qquad \begin{array}{|c|}\hline \phi \\ \vdots \\ \chi \\ \hline \end{array} \qquad \begin{array}{|c|}\hline \psi \\ \vdots \\ \chi \\ \hline \end{array}}{\chi} \qquad \text{Rule: } \vee e \tag{1.2.3}$$

Example 1.2.1: $p \vee q \vdash q \vee p$

Proof:

1	$p \vee q$	premise
2	p	assumption
3	$q \vee p$	$\vee i_2$. 2
4	q	assumption
5	$q \vee p$	$\vee i_1$. 4
6	$q \vee p$	$\vee e$. 1, 2–3, 4–5

Above, note that line 1 is the disjunction that we want to eliminate, and that lines 3 and 5 are deriving the same thing, $q \vee p$.

Example 1.2.2: Prove

$$q \to r \vdash p \vee q \to p \vee r \tag{1.2.4}$$

PROOF:

1	$q \to r$	premise
2	$p \vee q$	assumption
3	p	assumption
4	$p \vee r$	$\vee i_1$. 3
5	q	assumption
6	r	$\to e$. 1, 5
7	$p \vee r$	$\vee i_2$. 6
8	$p \vee r$	$\vee e$. 2, 3–4, 5–7
9	$p \vee q \to p \vee r$	$\to i$. 2, 8

1.2.4 Things to remember about OR-elimination

- To have a sound argument, both of the conclusions (the χ formula) must be the same.
- The work done by the $\vee e$ rule is really combining the work of the two χ cases.
- In each case, you may *not* use temporary assumptions from the other case. Each case must be derived independently.
- When using $\vee e$, three things must be mentioned: the disjunction being eliminated, and the two χ cases used to eliminate it.

1.2.5 The Copy Rule

The copy rule allows you to repeat a line that appeared earlier in the proof, subject to box scoping rules.

Example 1.2.3: Prove $\vdash p \to (q \to p)$

PROOF:

1	p	assumption
2	q	assumption
3	p	copy rule. Line 1
4	$p \to q$	$\to i$. 2, 3
5	$p \to (q \to p)$	$\to i$. 1, 4

The use of the copy rule in line 3 allows us to meet the scoping requirements of $\to i$ in line 4.

1.2.6 Negation Rules (¬)

Let us introduce the symbol $\bot$, which we will refer to as 'bottom'. We will use this symbol to name specific contradictions, like $p \wedge (\neg p)$.

$$\frac{\bot}{\phi} \qquad \text{Rule: } \bot e \tag{1.2.5}$$

Equation (1.2.5) is really saying the following: you can conclude anything from a contradiction.

Another variation on this theme is:

$$\frac{\phi, \neg\phi}{\bot} \qquad \text{Rule: } \neg e \tag{1.2.6}$$

We can use contradictions to introduce negations.

$$\frac{\boxed{\begin{array}{c}\phi \\ \vdots \\ \bot\end{array}}}{\neg\phi} \qquad \text{Rule: } \neg i \tag{1.2.7}$$

Example 1.2.4: Prove $p \to q, p \to \neg q \vdash \neg p$

Proof:

1	$p \to q$	premise
2	$p \to \neg q$	premise
3	p	assumption
4	q	$\to e$. 3, 1
5	$\neg q$	$\to e$. 3, 2
6	$\bot$	$\neg e$. 4, 5
7	$\neg p$	$\neg i$. Lines 3–6

Tables 1.2.6 and 1.2.6 show a summary of natural deduction rules.

Operator	introduction	elimination
$\wedge$	$\dfrac{\phi, \psi}{\phi \wedge \psi}$	$\dfrac{\phi \wedge \psi}{\phi}$ $\quad$ $\dfrac{\phi \wedge \psi}{\psi}$
$\vee$	$\dfrac{\phi}{\phi \vee \psi}$ $\quad$ $\dfrac{\psi}{\phi \vee \psi}$	$\dfrac{\phi \vee \psi \quad \boxed{\begin{array}{c}\phi\\ \vdots\\ \chi\end{array}} \quad \boxed{\begin{array}{c}\psi\\ \vdots\\ \chi\end{array}}}{\chi}$
$\rightarrow$	$\dfrac{\boxed{\begin{array}{c}\phi\\ \vdots\\ \psi\end{array}}}{\phi \rightarrow \psi}$	$\dfrac{\phi, \phi \rightarrow \psi}{\psi}$
$\neg$	$\dfrac{\boxed{\begin{array}{c}\phi\\ \vdots\\ \bot\end{array}}}{\neg\phi}$	$\dfrac{\phi, \neg\phi}{\bot}$
$\bot$		$\dfrac{\bot}{\phi}$
$\neg\neg$		$\dfrac{\neg\neg\phi}{\phi}$

Table 1.1: Summary of Natural Deduction Rules

$\dfrac{\phi \rightarrow \psi, \ \neg\psi}{\neg\phi}$	Modus Tollens
$\dfrac{\phi}{\neg\neg\phi}$	$\neg\neg i$
$\dfrac{\boxed{\begin{array}{c}\neg\phi\\ \vdots\\ \bot\end{array}}}{\phi}$	PBC: Proof by Contradiction
$\dfrac{}{\phi \vee \neg\phi}$	LEM: Law of Excluded Middle

Table 1.2: Derived Natural Deduction Rules

Example 1.2.5: Deriving Modus Tollens

$$\frac{\phi \to \psi, \neg\psi}{\neg\phi} \tag{1.2.8}$$

PROOF:

1	$\phi \to \psi$	premise
2	$\neg\psi$	premise
3	ϕ	assumption
4	ψ	$\to e$. 3, 1
5	$\bot$	$\neg e$. 4, 2
6	$\neg\phi$	$\neg e$. 3–5

Example 1.2.6: Derive

$$\frac{\phi}{\neg\neg\phi} \tag{1.2.9}$$

PROOF:

1	ϕ	premise
2	$\neg\phi$	assumption
3	$\bot$	$\neg e$. 1, 2
4	$\neg\neg\phi$	$\neg i$. 2–3

1.2.7 Law of Excluded Middle

Formally, the law of excluded middle (*LEM*, for short) is stated as follows:

$$\frac{}{\phi \vee \neg\phi} \quad \text{Rule: LEM} \tag{1.2.10}$$

There are no premises. This is also known as an *axiom*.

In essense, this axiom is saying "either ϕ is true or it's not". (There's no in-between).

Example 1.2.7: Derivation of the law of excluded middle.

PROOF:

1	$\neg(\phi \vee \neg\phi)$	assumption
2	ϕ	assumption
3	$\phi \vee \neg\phi$	$\vee i_1$. Line 2
4	$\bot$	$\neg e$ 3, 1
5	$\neg\phi$	$\neg i$. 2–4
6	$\phi \vee \neg\phi$	$\vee i_2$. 5
7	$\bot$	$\neg e$. 6, 1
8	$\neg\neg(\phi \vee \neg\phi)$	$\neg i$. 1–7.
9	$\phi \vee \neg\phi$	$\neg\neg e$. 8

1.2.8 Provable Equivalence

We say that Two formulas, ϕ and ψ are *provably equivalent*

$$\phi \dashv\vdash \psi$$

if $\phi \vdash \psi$ and $\psi \vdash \phi$.

1.2.9 Intuitionism

Intuitionism is a set of mathematical beliefs. In a nutshell, the intuitionist view requires direct proofs. For example an intuitionist would not accept the notion of ϕ being true if it were proven by showing $\neg\phi$ were a contradiction.

'Classical' mathematicians accept proof by contradiction.

Consider the following example.

Theorem 1.2.8: There are irrational numbers a, b such that a^b is rational.

A classical proof of this would be as follows:

PROOF: Let $b = \sqrt{2}$, an irrational number.

Case 1 Assume that b^b is rational. If so, we simply let $a = b$

Case 2 Assume b^b is irrational. Let $a = b^b$; by assumption, a is still an irrational number. This assignment of a gives

$$a^b = (b^b)^b = b^{b^2} = \sqrt{2}^2 = 2 \tag{1.2.11}$$

and 2 is rational.

Since the above cases are exhaustive (either b^b is rational or it isn't), the proof is complete. □

The classical view would accept this proof. The intuitionist view would not.

1.2.10 Semantics

In logic, semantics come from truth values: T, F.

Definition 1.2.9 (Valuation)**:** A *valuation* (or *model*) for ϕ is an assignment of truth values to each variable in the formula ϕ.

Let v be a valuation for ϕ. Under v, ϕ has a truth value of $v(\phi)$.

Truth Tables are one way that we can represent a valuation.

Example 1.2.10: The following is a truth table for implication.

ϕ	ψ	$\phi \to \psi$
T	T	T
T	F	F
F	T	T
F	F	T

Another example of semantic notation:

$$\phi_1, \ldots, \phi_n \vDash \psi \tag{1.2.12}$$

Equation (1.2.12) is valid if every valuation to the variables $\phi_1, \ldots, \phi_n, \psi$ that makes $\phi_1 \ldots \phi_n$ true also makes ψ true.

Example 1.2.11: $p,\ p \to q \vDash q$. This can be verified by looking at the truth table for implication.

Another rule

$$\bot \vDash \phi \qquad \text{for } \textbf{all } \phi \tag{1.2.13}$$

We also have *tautologies*

$$\vDash \phi \tag{1.2.14}$$

If ϕ is a tautology, then every truth assignment makes ϕ true.

1.2.11 Box Rules

The following is a list of rules for using boxes. We'll use this list when proving soundness and completeness.

BOX1 In a proof, all boxes must be closed *before* the last line of the proof.

BOX2 Boxes must be properly nested.

BOX3a Once a box closes, no line in the box can be referenced later

BOX3b Once a box is closed, no box strictly inside the closed box can be referenced later.

1.2.12 Soundness

Theorem 1.2.12 (Soundness Theorem)**:** If

$$\phi_1, \ldots, \phi_n \vdash \psi$$

then

$$\phi_1, \ldots, \phi_n \vDash \psi$$

Put another way, syntax implies semantics.

A proof of this appears in the lecture notes for the next class (page 20).

1.3 Soundness, Completeness, and CNF (Text Notes)

These are notes from Huth & Ryan, Chapter 1

1.3.1 Soundness And Completeness

$\vdash$ is a syntactic notion.
$\models$ is a semantic notion. $\models$ is also called *semantic entailment*.

Soundness If $\phi_1, \ldots, \phi_n \vdash \psi$ holds, then so does $\phi_1, \ldots, \phi_n \models \psi$.

Completeness Wherever $\phi_1, \ldots \phi_n \models \psi$ holds, there exists a natural deduction proof of $\phi_1, \ldots \phi_n \vdash \psi$

Consider the formula

$$\models \phi_1 \to (\phi_2 \to (\phi_3 \to \ldots \to (\phi_n \to \psi))) \tag{1.3.1}$$

Because (1.3.1) is a chain of implications, the formula will hold *unless* ψ is false.

Theorem 1.3.1 (Soundness and Completeness)**:** Let $\phi_1, \ldots, \phi_n, \psi$ be formulas of propositional logic. Then $\phi_1, \ldots, \phi_n \models \psi$ holds IFF the sequent $\phi_1, \ldots \phi_n \vdash \psi$ is valid.

Soundness means that whatever we prove will be a true fact, based on truth tables.

Completeness means that no matter what (semantically) valid sequents there are, they all have syntactic proofs in the system of natural deduction.

We define *equivalence of formulas* using $\models$. If ϕ *semantically entails* ψ and vice versa, then ϕ and ψ are the same as far as our truth table semantics are concerned.

Definition 1.3.2 (Semantic Equivalence)**:** ϕ and ψ are *semantically equivalent* if $\phi \models \psi$ and $\psi \models \phi$ hold. In this case, we write $\phi \equiv \psi$.

Definition 1.3.3 (Validity)**:** We say that ϕ is *valid* if $\models \phi$ holds.

Example 1.3.4: The following are valid formulas

$$p \to q \equiv \neg q \to \neg p$$
$$p \to q \equiv \neg p \vee q$$
$$p \wedge q \to p \equiv r \vee \neg r$$

Definition 1.3.5 (Tautology)**:** η is a *tautology* if $\models \eta$ holds.

Lemma 1.3.6: Given formulas of propositional logic $\phi_1, \ldots, \phi_n, \psi$

$$\phi_1, \ldots, \phi_n \models \psi$$

holds IFF

$$\models \phi_1 \to (\phi_2 \to (\phi_3 \to \ldots \to (\phi_n \to \psi)))$$

holds.

Definition 1.3.7 (Conjunctive Normal Form)**:** A formula C is in *conjunctive normal form* (CNF) if C is a conjunction of clauses, where each clause D is a disjunction of literals. Example:

$$(a \vee b) \wedge (c \vee d \vee e) \wedge (f)$$

Definition 1.3.8 (Satisfiable)**:** A formula ϕ in propositional logic is *satisfiable* if ϕ has a valuation such that $v(\phi) = \mathrm{T}$

Satisfiability is a weaker concept than validity. For example,

$$p \vee q \rightarrow p$$

is satisifiable; the formula will be true whenever $p = \mathrm{T}$. However, $p \vee q \rightarrow p$ is not valid – it evaluates to false when $p = \mathrm{F}$ and $q = \mathrm{T}$.

It is possible to specify a forumula ϕ by its truth table alone. In this case, we don't know how ϕ appears syntactically, but we know how ϕ is supposed to "behave".

1.4 Lecture – 2/5/2007

1.4.1 Soundness

As noted earlier, *soundness* is a quality whereby

$$\phi_1, \ldots, \phi_n \vdash \psi \text{ guarantees } \phi_1, \ldots, \phi_n \vDash \psi$$

This is referred to as the *soundness theorem*

1.4.2 Proof of the Soundness Theorem

Here's a partial proof, using course-of-values induction.[1]

Let us define $M(k)$

$M(k)$: If $\phi_1, \ldots, \phi_n \vdash \psi$ by a proof of length k, then $\phi_1, \ldots, \phi_n \vDash \psi$.

Let us fix $k \geq 1$, and assume $M(k')$ is true for all k' such that $1 \leq k' \leq k$.

Let $\phi_1, \ldots, \phi_n \vdash \psi$ be a proof of length k.

The following list of cases base the justification on the last line.

1. Premise. If the last line is a premise, then ψ is the same as some ϕ_i, and we need to show $\phi_1, \ldots, \phi_n \vDash \phi_i$.
2. Assumption. The last line of the proof cannot be an assumption, by rule **BOX1**.
3. Rule $\wedge e$. If the last line of the proof is $\wedge e$, then the last line cannot be part of a box. By the inductive hypothesis, $\phi_1, \ldots, \phi_n \vDash \phi \wedge \psi$, so $\phi_1, \ldots, \phi_n \vDash \psi$.
4. $\bot$. If the last line of the proof is $\bot$ from the application of $\neg e$, then by the inductive hypothesis, we have

   $$\phi_1, \ldots, \phi_n \vDash \phi$$
   $$\phi_1, \ldots, \phi_n \vDash \neg\phi$$

   There is no truth assignment that makes $\phi_1, \ldots, \phi_n$ true, so

   $$\phi_1, \ldots, \phi_n \vDash \bot$$

5. Implication. Suppose the last line is $\theta_1 \rightarrow \theta_2$. By the rules of natural deduction, there must be a box with

   $$\boxed{\begin{array}{c}\theta_1 \\ \vdots \\ \theta_2\end{array}}$$

   and this box must occur at the top level. By the inductive hypothesis,

   $$\phi_1, \ldots, \phi_n, \theta_1 \vDash \theta_2 \qquad \text{so}$$
   $$\phi_1, \ldots, \phi_n \vDash \theta_1 \rightarrow \theta_2$$

[1] See Huth and Ryan, pg. 43

6. Or-elimination. Suppose the last line of the proof was $\eta_1 \vee \eta_2$. Then, the proof will have the general structure

1	$\eta_1 \vee \eta_2$
2	η_1
3	$\vdots$
4	ψ
5	η_2
6	$\vdots$
7	ψ
8	ψ

where the final ψ cannot lie inside a box. By the inductive hypothesis

$$\phi_1, \ldots, \phi_n \vDash \eta_1 \vee \eta_2$$

so

$$\phi_1, \ldots, \phi_n, \eta_1 \vDash \psi \qquad \text{or}$$
$$\phi_1, \ldots, \phi_n, \eta_2 \vDash \psi$$

Thus

$$\phi_1, \ldots, \phi_n \vDash \psi$$

7. Not introduction. Suppose the last line came from an application of $\neg i$. Our proof will have the general form

$$\begin{array}{|c|}\hline \phi \\ \vdots \\ \bot \\ \hline\hline \end{array}$$
$$\neg\phi$$

Therefore $\psi = \neg\phi$. By the inductive hypotheis,

$$\phi_1, \ldots, \phi_n, \phi \vDash \bot \qquad \text{So,}$$
$$\phi_1, \ldots, \phi_n \vDash \neg\phi$$

1.4.3 Completeness of Natural Deduction

Completeness means the following:

If $\phi_1, \ldots, \phi_n \vDash \psi$ then $\phi_1, \ldots, \phi_n \vdash \psi$

1.4.4 Proof of the Completeness Theorem

The following comes from a class discussion of the proof that appears in Huth & Ryan, pg. 49–53.

There are three steps to the proof

Step 1 $\vDash \phi_1 \to (\phi_2 \to \ldots \to (\phi_n \to \psi))$

Step 2 $\vdash \phi_1 \to (\phi_2 \to \ldots \to (\phi_n \to \psi))$

Step 3 $\phi_1, \ldots, \phi_n \vdash \psi$

Steps 1 and 3 are easy. Step two takes some work.

Completeness Proof: Step 1

$$\vDash \phi_1 \rightarrow (\phi_2 \rightarrow \ldots \rightarrow (\phi_n \rightarrow \psi)) \tag{1.4.1}$$

is expressing a *tautology*. Because it is a nested implication, (1.4.1) can be false only if $\psi = \text{F}$. However, $\psi = \text{F}$ would contradict $\phi_1, \ldots, \phi_n \vDash \psi$.

Therefore (1.4.1) holds.

Completeness Proof: Step 2

Step 2 is really saying the following

Theorem 1.4.1: If $\vDash \eta$ holds, then $\vdash \eta$ is valid. In other words, if η is a *tautology*, then η is also a *theorem*.

Suppose η holds. Then η contains n distinct propositional atoms $p_1, \ldots, p_n$. Because η is a tautology, each of the 2^n lines in η's truth table evaluates to T. We'll devise an approach that allows us to take all 2^n sequents and assemble them into a proof for η.

Lemma 1.4.2: Let ϕ have propositional atoms $p_1, \ldots, p_n$. Let L be a line in ϕ's truth table. For $1 \leq i \leq n$, let $\hat{p_i} = p_i$ if $p_i = \text{T}$ in line L. Otherwise, let $\hat{p_i} = \neg p_i$. Then, we have

1. $\hat{p_1}, \ldots, \hat{p_n} \vdash \phi$ is provable if the entry for ϕ in line L is T.
2. $\hat{p_1}, \ldots, \hat{p_n} \vdash \neg\phi$ is provable if the entry for ϕ in line L is F.

Lemma 1.4.2 can be proven by induction on ϕ

Basis. Let ϕ be some variable p. Then we have one of two cases:

$$p \vdash p$$
$$\neg p \vdash \neg p$$

Inductive Step 1. Suppose $\phi = \neg\phi_1$, and assume the result is T for ϕ_1. There are two possibilities:

1. $\phi = \text{T}$ in line L. Then $\phi_1 = \text{F}$ in line L. By the inductive hypothesis,

   $$\hat{p_1}, \ldots, \hat{p_n} \vdash \neg\phi_1 = \phi$$

2. Suppose $\phi = \text{F}$ in line L. Then $\phi_i = \text{T}$ in line L. By the inductive hypothesis,

   $$\hat{p_1}, \ldots \hat{p_n} \vdash \phi_1$$
   $$\hat{p_1}, \ldots \hat{p_n} \vdash \neg\neg\phi_1 = \phi \qquad \text{by } \neg\neg i$$

Inductive Step 2. Here, ϕ has the form

$$\phi = \phi_1 \circ \phi_2 \qquad \text{for } \circ \in \{\vee, \wedge, \rightarrow\}$$

Let $q_1, \ldots, q_m$ be the variables of ϕ_1. Let $r_1, \ldots, r_k$ be the variables of ϕ_2. This gives

$$\{p_1, \ldots, p_n\} = \{q_1, \ldots, q_m\} \cup \{r_1, \ldots, r_k\}$$

Let L_1 be a line in ϕ_1's truth table corresponding to line L. Let L_2 be a line in ϕ_2's truth table corresponding to line L. We have

$\hat{q_1}, \ldots, \hat{q_m}$	for line L_1
$\hat{r_1}, \ldots, \hat{r_k}$	for line L_2

Therefore

$$\{\hat{p_1}, \ldots, \hat{p_n}\} = \{\hat{q_1}, \ldots, \hat{q_m}\} \cup \{\hat{r_1}, \ldots, \hat{r_k}\}$$

Suppose $\circ = \rightarrow$. Then we have $\phi = \phi_1 \rightarrow \phi_2$. If $\phi = \mathrm{F}$ in line L, then $\phi_1 = \mathrm{T}$ and $\phi_2 = \mathrm{F}$. By the inductive hypothesis,

$\hat{q_1}, \ldots, \hat{q_m} \vdash \phi_1$	
$\hat{r_1}, \ldots, \hat{r_k} \vdash \neg\phi_2$	
$\hat{p_1}, \ldots, \hat{p_n} \vdash \phi_1 \wedge \neg\phi_2$	by $\wedge i$

However, what we actually need to show is

$$\hat{p_1}, \ldots, \hat{p_n} \vdash \neg(\phi_1 \rightarrow \phi_2)$$

But we can make the transformation by natural induction.

1	$\phi_1 \wedge \neg\phi_2$	premise
2	ϕ_1	$\wedge e$
3	$\neg\phi_2$	$\wedge e$
4	$\phi_1 \rightarrow \phi_2$	assumption
5	ϕ_2	$\rightarrow e$
6	$\bot$	
7	$\neg(\phi_1 \rightarrow \phi_2)$	$\neg i$

Having $\phi = \phi_1 \rightarrow \phi_2 = \mathrm{F}$ is only one of four possible cases. Suppose ϕ's implication were true in line L. Then, we have three cases to consider (we'll just look at one of them).

Suppose $\phi_1 \rightarrow \phi_2 = \mathrm{T}$, $\phi_1 = \mathrm{T}$, and $\phi_2 = \mathrm{T}$. By the inductive hypothesis,

$$\hat{q_1}, \ldots, \hat{q_m} \vdash \phi_1$$
$$\hat{r_1}, \ldots, \hat{r_k} \vdash \phi_2$$
$$\hat{p_1}, \ldots, \hat{p_n} \vdash \phi_1 \wedge \phi_2$$

What we need to show is

$$\hat{p_1}, \ldots, \hat{p_n} \vdash \phi_1 \rightarrow \phi_2$$

Again, the transformation may be made by natural induction:

1	$\phi_1 \wedge \phi_2$	premise
2	ϕ_1	$\wedge e$
3	ϕ_2	$\wedge e$
4	ϕ_1	assumption
5	ϕ_2	copy line
6	$\phi_1 \rightarrow \phi_2$	$\rightarrow i$

All told there are 12 cases to consider for $\phi = \phi_1 \circ \phi_2$: all four truth table lines for $\circ \in \{\wedge, \vee, \rightarrow\}$. Above, we've done two – there are 10 more. Try working a few of them out.

This completes the proof of Lemma 1.4.2. We'll finish step 2 next.

Given $p_1, \ldots, p_n$, let $n = 2$, $p_1 = p$ and $p_2 = q$. Furthermore, let η be a tautology.

By Lemma 1.4.2

$$p, q \vdash \eta$$
$$\neg p, q \vdash \eta$$
$$p, \neg q \vdash \eta$$
$$\neg p, \neg q \vdash \eta$$

To finish step 2, we need to show this using natural deduction. The proof shown below isn't particular to any specific η, but it's the general form we'd need to use.

1	$p \vee \neg p$						LEM
2	p		assumption	$\neg p$			assumption
3	$q \vee \neg q$		LEM	$q \vee \neg q$			LEM
4	q	assumption	$\neg q$	assumption	q	assumption	$\neg q$	assumption
5	$\vdots$		$\vdots$		$\vdots$		$\vdots$	
6	η		η		η		η	
7	η		$\vee e$	η			$\vee e$
8	η						$\vee e$

Completeness Proof: Step 3

Given

$$\vdash \phi_1 \rightarrow (\phi_2 \rightarrow \ldots \rightarrow (\phi_n \rightarrow \psi)) \qquad \text{From step 2}$$

we need to show

$$\phi_1, \ldots, \phi_n \vdash \psi$$

This is a mechanical transformation. The general idea is to assume $\phi_1 = \mathrm{T}$, which forces

$$\phi_2 \rightarrow \ldots \rightarrow (\phi_n \rightarrow \psi) = \mathrm{T}$$

Continue doing this for each ϕ_i, and you'll eventually get to $\phi_n \rightarrow \psi$, and finally ψ.

1.4.5 Normal Forms

Definition 1.4.3 (Semantic Equivalence)**:** ϕ and ψ are *semantically equivalent* if $\phi \vDash \psi$ and $\psi \vDash \phi$. We write this as $\phi \equiv \psi$.

We say that ϕ is *valid* if $\vDash \phi$. "valid" is just another name for "tautology".

Lemma 1.4.4: Given forumulas $\phi_1, \ldots, \phi_n, \psi$,

$$\phi_1, \ldots, \phi_n \vDash \psi \text{ IFF } \vDash \phi_1 \rightarrow (\phi_2 \rightarrow \ldots \rightarrow (\phi_n \rightarrow \psi))$$

We've covered the forward case of this already – we'll cover the reverse here.

Suppose

$$\phi_1 \rightarrow (\phi_2 \rightarrow \ldots \rightarrow (\phi_n \rightarrow \psi))$$

is valid. If a truth assignment makes $\phi_1, \ldots, \phi_n$ true, then it also makes

$$\phi_1 \rightarrow (\phi_2 \rightarrow \ldots \rightarrow (\phi_n \rightarrow \psi))$$

true, and it will make ψ true as well. If $\phi_1, \ldots, \phi_n$ were true and $\psi = \text{F}$, this would contradict $\phi_1, \ldots, \phi_n \vDash \psi$.

This reduces entailment to a tautology.

Truth tables can be used to test validity. However, there's a disadvantage to this: given n propositional variables, the truth table will have 2^n rows. In the next lecture, we'll look at other ways of testing validity.

1.5 Lecture – 2/7/2007

1.5.1 Conjunctive Normal Form (CNF)

A formula ϕ is in CNF if ϕ is a conjunction of disjunctions of literals. We can define CNF as a grammar

$$\begin{array}{rcl} \text{Literal} & = & p \mid \neg p \\ \text{Disjunction} & = & \text{Literal} \mid \text{Literal} \vee \text{Disjunction} \\ \text{CNF} & = & \text{Disjunction} \mid \text{Disjunction} \wedge \text{CNF} \end{array}$$

Example 1.5.1: Formulas in CNF

$$(p \vee q) \wedge (\neg p \vee \neg q)$$
$$p \wedge q \wedge r$$

As the second line shows, it's okay for a Disjunction to consist of a single Literal.

If a formula ϕ is in CNF, then there is an easy way to check its validity.

$\vDash \phi_1 \wedge \ldots \wedge \phi_n$ IFF $\vDash \phi_1$ and $\ldots$ and $\vDash \phi_n$

$\phi_1 \wedge \ldots \wedge \phi_n$ is valid IFF every ϕ_i is valid.

We can state this more formally:

Theorem 1.5.2: A disjunction of literals $p_1, \ldots, p_n$ is valid IFF for some j, k, $1 \leq j, k \leq n$, $p_j = \neg p_k$.

Proof: If j, k exist, then the disjunction is valid (Law of Excluded Middle). However, if no such j, k exist, we can assign a value of F to each p_i and make $p_1 \vee \ldots \vee p_n$ false.

Truth tables can be used to test validity. Another way to test validity is to construct a proof by natural deduction (prove $\vdash \phi$). A third way to test validity is to convert an arbitrary formula into an equivalent CNF formula, and test the CNF formula.

Definition 1.5.3 (Satisfiable)**:** ϕ is *satisfiable* if there is some truth assignment that makes ϕ true.

validity $\rightarrow$ satisfiable
satisfiable $\nrightarrow$ validity

Note that ϕ is satisifiable IFF $\neg\phi$ is not valid. Therefore, if we can determine validity, then we can determine satisifiability.

Similarly, ϕ is valid IFF $\neg\phi$ is not satisifiable.

CNF and Computability

Suppose we had a function `CNF`(ϕ) that took an arbitrary formula ϕ and converted it to CNF. `CNF` operates under the following conditions:

- `CNF`(ϕ) $\equiv \phi$
- `CNF`(ϕ) is in CNF.

With such a function, we can test if ϕ is valid by testing whether `CNF`(ϕ) is valid.

`CNF`(ϕ) could not run in polynomial time. There are formulas ϕ such that any equivalent formula in conjunctive normal form is exponentially larger. For example:

$$(X_1 \wedge Y_1) \vee (X_2 \wedge Y_2) \vee \ldots \vee (X_n \wedge Y_n) \tag{1.5.1}$$

Equation (1.5.1) is in disjunctive normal form. Converting it to CNF would require many applications of distributive laws, resulting in a much longer formula.

1.5.2 Truth Tables and Conjunctive Normal Form

Given a truth table for ϕ, it is easy to give an equivalent formula in conjunctive normal form. This can be done even if you don't know the syntactic structure of ϕ.

Consider the following truth table

p	q	r	ϕ
T	T	T	T
T	T	F	T
T	F	T	T
T	F	F	F
F	T	T	T
F	T	F	F
F	F	T	T
F	F	F	F

We turn this into CNF as follows:

1. Take all the lines where $\phi = \text{F}$
2. Form a disjunction of variables $\hat{p_i}$, corresponding to variables in the truth table: $\hat{p_i} = \neg p_i$
3. Combine the disjunctions with conjunctions.

In our example, lines 4, 6, 8 are those where $\phi = \text{F}$. Our three disjunctions are

$\neg p \vee q \vee r$	from line 4
$p \vee \neg q \vee r$	from line 6
$p \vee q \vee r$	from line 8

Combining these with $\wedge$ gives

$$(\neg p \vee q \vee r) \wedge (p \vee \neg q \vee r) \wedge (p \vee q \vee r)$$

There is a special case. If all truth table entries are true, then $(p \vee \neg p)$ is a perfectly good equivalent CNF.

Of course, because one can test validity directly from a truth table, there's not much sense in going from truth table to CNF to validity test.

1.5.3 Procedure for CNF Transformation

The general procedure for turning an arbitrary formula into CNF is as follows:

1. Remove implication, by transforming $p \rightarrow q$ to $\neg p \vee q$.
2. Push negation inward, using DeMorgan's laws. We want negation to apply to atoms, not clauses
3. Use distributivity to transform the formula into CNF.

Removing Implication

Let us define the procedure `IMPL_FREE`. (We only mentioned the first case in class – I'm adding the other three).

procedure `IMPL_FREE`(ϕ)
  POSTCONDITION: `IMPL_FREE`(ϕ) $\equiv \phi$
  POSTCONDITION: `IMPL_FREE`(ϕ) has no $\rightarrow$
  if $\phi = \phi_1 \rightarrow \phi_2$ **then**
    return $\neg$`IMPL_FREE`(ϕ_1) $\vee$ `IMPL_FREE`(ϕ_2)
  else if $\phi = \phi_1 \vee \phi_2$ **then**
    return `IMPL_FREE`(ϕ_1) $\vee$ `IMPL_FREE`(ϕ_2)
  else if $\phi = \phi_1 \wedge \phi_2$ **then**
    return `IMPL_FREE`(ϕ_1) $\wedge$ `IMPL_FREE`(ϕ_2)
  else if $\phi = \neg\phi_1$ **then**
    return $\neg$`IMPL_FREE`(ϕ_1)
  else if $\phi = p$ **then**
    return p
  end if
end procedure

Removing Negation

One of our itermediate goals will be turn turn ϕ into an equivalent formula in Negation Normal Form (NNF).

Definition 1.5.4 (Negation Normal Form)**:** ϕ is in NNF if every negation applies to an atom, and ϕ is implication-free.

Again, we'll write an algorithm:

procedure `NNF`(ϕ)
  PRECONDITION: ϕ is implication-free
  POSTCONDITION: `NNF`(ϕ) $\equiv \phi$
  POSTCONDITION: `NNF`(ϕ) is in `NNF`
  if $\phi = p$ **then**
    return p
  else if $\phi = \phi_1 \vee \phi_2$ **then**
    return `NNF`(ϕ_1) $\vee$ `NNF`(ϕ_2)
  else if $\phi = \phi_1 \wedge \phi_2$ **then**
    return `NNF`(ϕ_1) $\wedge$ `NNF`(ϕ_2)
  else if $\phi = \neg p$ **then**
    return $\neg p$
  else if $\phi = \neg\neg\phi$ **then**
    return `NNF`(ϕ)
  else if $\phi = \neg(\phi_1 \vee \phi_2)$ **then**
    return `NNF`($\neg\phi_1$) $\wedge$ `NNF`($\neg\phi_2$)
  else if $\phi = \neg(\phi_1 \wedge \phi_2)$ **then**
    return `NNF`($\neg\phi_1$) $\vee$ `NNF`($\neg\phi_2$)
  end if
end procedure

Distributing Subformulas

The procedure `DISTR` implements the distributive laws needed for CNF conversion.

procedure `DISTR`(η_1, η_2)
  PRECONDITION: η_1 and η_2 are in CNF
  POSTCONDITION: `DISTR`(η_1, η_2) is in CNF
  POSTCONDITION: `DISTR`$(\eta_1, \eta_2) \equiv \eta_1 \vee \eta_2$
  if $\eta_1 = \eta_{1_1} \wedge \eta_{1_2}$ **then**
    return `DISTR`$(\eta_{1_1}, \eta_2) \wedge$ `DISTR`(η_{1_2}, η_2)
  else if $\eta_2 = \eta_{2_1} \wedge \eta_{2_2}$ **then**
    return `DISTR`$(\eta_1, \eta_{2_1}) \wedge$ `DISTR`(η_1, η_{2_2})
  else
    return $\eta_1 \vee \eta_2$
  end if
end procedure

The `CNF` Procedure

Using `IMPL_FREE`, `DISTR`, and `NNF` as building blocks, we can now define a procedure `CNF` that takes an arbitrary formula ϕ as input and returns an equivalent formula in conjunctive normal form.[2]

procedure `CNF`(ϕ)
  PRECONDITION: ϕ is in NNF
  POSTCONDITION: `CNF`$(\phi) \equiv \phi$
  POSTCONDITION: `CNF`(ϕ) is in CNF
  if $\phi = p$ **then**
    return p
  else if $\phi = \neg p$ **then**
    return $\neg p$
  else if $\phi = \phi_1 \wedge \phi_2$ **then**
    return `CNF`$(\phi_1) \wedge$ `CNF`(ϕ_2)
  else if $\phi = \phi_1 \vee \phi_2$ **then**
    return `DISTR`$($`CNF`$(\phi_1),$ `CNF`$(\phi_2))$
  end if
end procedure

`CNF` carries the precondition that ϕ is in NNF. We do the actual CNF conversion with the following call

$$\texttt{CNF}(\texttt{NNF}(\texttt{IMPL\_FREE}(\phi))) \tag{1.5.2}$$

In the worst case (1.5.2) will take exponential time. However, there are some inputs where the running time will be polynomial.

By contrast, using truth tables *always* takes exponential time.

1.5.4 Satisfiability Problems

The satisifiability problem is as follows: given a propositional logic formula ϕ, is ϕ satisfiable? We'll refer to this as the *SAT Problem*

The SAT problem is NP-complete. If we were to find a polynomial-time algorithm for SAT, that would imply P = NP. Therefore, it is unlikely that we will find a polynomial time algorithm.

[2] In our lecture, we called this formula `CNF'`. I'm using `CNF` to be consistent with the text

Earlier, we related satisifiability to validity. ϕ is not satisifiable if $\neg\phi$ is valid, and ϕ is valid if $\neg\phi$ is not satisifiable. This means that validity cannot be done in polynomial time. (If we had a way to compute validity in polynomial time, then we'd have a way to compute SAT in polynomial time).

Let's consider another function: $\texttt{CNF}^*(\phi)$, whose properties are as follows:

- $\texttt{CNF}^*(\phi)$ is in conjunctive normal form
- ϕ is valid IFF $\texttt{CNF}^*(\phi)$ is valid.
- $\texttt{CNF}^*(\phi)$ is computable in polynomial time

Where ϕ is any formula of propositional logic.

Is such a $\texttt{CNF}^*$ likely? No, because it would give us a polynomial time test for validity.

Let's consider another function: $\texttt{CNF}^{**}(\phi)$:

- $\texttt{CNF}^{**}(\phi)$ is in conjunctive normal form
- ϕ is valid IFF $\texttt{CNF}^{**}(\phi)$ is satisfiable.
- $\texttt{CNF}^{**}(\phi)$ is computable in polynomial time

In this case, $\texttt{CNF}^{**}(\phi)$ is possible to compute in polynomial time (*Why?*).

Another example: $\texttt{CNF-SAT}(\phi)$. Given a formula in CNF, is ϕ satisfiable? This is still NP-complete.

In summary

- Validity is easy to check when ϕ is in CNF
- Satisifiability is not easy to check when ϕ is in CNF

1.5.5 Horn Clauses

Horn clauses are named after Alfred Horn.

Let's review some notation

$\bot$	bottom – a contradiction
$\top$	top – a tautology

$\top$ is equivalent to $(p \vee \neg p)$.

The structure of horn clauses is shown in the following grammar:

$$\begin{array}{lcl} \mathrm{P} & = & \top \mid \bot \mid p \\ \mathrm{A} & = & \mathrm{P} \mid \mathrm{P} \wedge \mathrm{A} \\ \mathrm{C} & = & \mathrm{A} \rightarrow \mathrm{P} \\ \mathrm{H} & = & \mathrm{C} \mid \mathrm{C} \wedge \mathrm{H} \end{array}$$

Example 1.5.5: Horn Clauses.

$(p_1 \wedge p_2 \wedge p_3 \rightarrow p_4) \wedge (p_3 \rightarrow p_5)$
$p_1 \wedge p_2 \rightarrow \bot$
$(\top \rightarrow p_2) \wedge (p_3 \rightarrow \bot)$
$\top \wedge \bot \wedge \top \rightarrow \bot$

Horn clauses are equivalent to CNF. For example

$$\begin{aligned} & p_1 \wedge p_2 \wedge p_3 \rightarrow p_4 \\ =& \neg(p_1 \wedge p_2 \wedge p_3) \vee p_4 \\ =& \neg p_1 \vee \neg p_2 \vee \neg p_3 \vee p_4 \end{aligned}$$

This is just a straightforward application of implication elimination and the DeMorgan laws.

When converted to CNF, there will be one atom which is not negated.

Translation of Horn Clauses to Disjunctions

Suppose we are given the formula

$$\phi = p_1 \wedge p_2 \wedge \ldots \wedge p_k \rightarrow p'$$

There are a few cases to consider

Case 1 If at least one of p_i is $\bot$, then ϕ is a tautology. $(p_i \vee \neg p_i)$ is a perfectly good equivalent (for any p_i).

Case 2 No p_i is $\bot$, p' is an atom. We translate this as

$$\neg p_{i_1} \vee \ldots \vee \neg p_{i_r} \vee p'$$

$\neg p_{i_1} \vee \ldots \vee \neg p_{i_r}$ are the atoms in $p_1 \ldots p_k$. In other words, we leave out $\top$. Example

$$\top \wedge p_1 \wedge p_2 \rightarrow p' \Rightarrow \neg p_1 \vee \neg p_2 \vee p_3$$

Case 3 No p_i is bottom, at least one p_i is an atom, and $p' = \bot$. We translate this as

$$\neg p_{i_1} \vee \ldots \vee \neg p_{i_r}$$

Case 4 All p_i are $\top$, $p' = \bot$. Example:

$$\top \wedge \top \wedge \top \rightarrow \bot$$

This is translated as $\Box$ ("Box"). Box is an empty disjunction, and it is always false.

Case 5 $p' = \top$. Here, we have a tautology. We can translate it as $(p_i \vee \neg p_i)$ for any p_i.

1.6 Lecture – 2/12/2007

1.6.1 Horn Formulas

To review, horn formulas are defined with the following grammar

$$P = \bot \mid \top \mid p$$
$$A = P \mid P \wedge A$$
$$C = A \rightarrow P$$
$$H = C \mid C \wedge H$$

Horn formulas are really just a special case of CNF, where each disjunction has at most one positive literal.

Example 1.6.1: Examples of horn formulas

$$\begin{aligned} & p_1 \wedge p_2 \wedge \top \rightarrow p_3 \\ \equiv & \neg p_1 \vee \neg p_2 \vee p_3 \end{aligned}$$

$$\begin{aligned} & \top \wedge \top \wedge \top \rightarrow \bot \\ \equiv & \Box \qquad \text{An unsatisifiable formula} \end{aligned}$$

We call the symbol $\Box$ "Box". Box is an empty disjunction that is not satisifiable. It's equivalent to $(p \wedge \neg p)$, but has some technical conveniences.

Satisifiablity for CNF formulas is an NP-Complete problem.
Validity for CNF forumulas is a P problem (there is an efficient solution).
Horn formulas have an efficient algorithm for satisfiability.

1.6.2 Algorithm For Horn Satisfiability

Below is a linear-time algorithm that determines the satisifiability of horn clauses:

procedure `HORN`(ϕ)
  PRECONDITION: ϕ is a horn formula
  POSTCONDITION: `HORN` returns 'satisifiable' if ϕ is satisfiable; `HORN` returns unsatisfiable otherwise.
  Mark $\top$
  while there is a clause $p_1 \wedge \ldots \wedge p_k \rightarrow p'$ such that $p_1 \wedge \ldots \wedge p_k$ are marked but p' is not marked **do**
    Mark p'
  end while
  if $\bot$ is marked **then**
    return "unsatisifiable"
  else
    return "satisfiable"
  end if
end procedure

Why is this correct?

Claim 1.6.2: If v satisifies ϕ and p is marked, then $v(p) = \mathrm{T}$.

PROOF (Horn Algorithm): Our proof is by induction on the number of iterations of the while loop.

$M(k)$: if p is marked after k iterations of the while loop, then $v(p) = \mathrm{T}$ for any v with $v(\phi) = \mathrm{T}$

Basis: $k = 0$. If p is unmarked before the while loop is entered, then $p = \top$, and $v(\top) = \mathrm{T}$.

Inductive Case: Suppose $M(k)$ is true and p is marked the the $k+1$ iteration of the while loop. If p is marked, then there is a clause $p_1 \wedge \ldots \wedge p_k \rightarrow p$ where $p_1, \ldots . p_k$ are marked after k iterations. By the inductive hypothesis, $v(p_1), \ldots, v(p_l) = \mathrm{T}$. Therefore $v(p_1 \wedge \ldots \wedge p_k \rightarrow p)$ is true and $v(p)$ is true.

We can also guarantee that the algorithm will terminate on correct input. Each time the loop executes, a new p' is marked. There are only a finite number of literals to mark, therefore the algorithm must terminate eventually.

If the output is "unsatisifiable", then $\bot$ is marked. Therefore, any satisfying assignment must make $\bot = \mathrm{T}$. This is not possible, so there is no satisfying assignment for ϕ.

If the output is "satisfiable", then we must define v by $v(p) = \mathrm{T}$ if p is a marked atom and $v(p) = \mathrm{F}$ if p is an unmarked atom.

To show that v satisifies ϕ, it is enough to how that v satisifies each clause $p_1 \wedge \ldots \wedge p_k \rightarrow p'$

If any $v(p_i) = \mathrm{F}$, then the clause is satisfied.

If $v(p_i) = \mathrm{T}$ for all i, then each p_i is either $\top$ or a marked atom. Therefore $p' \neq \bot$; p' is either (1) $\top$ or (2) a marked atom, so $v(p') = \mathrm{T}$ and $v(p_1 \wedge \ldots \wedge p_k \rightarrow p') = \mathrm{T}$. □

1.6.3 SAT Solvers

Next, we'll look at an algorithm which takes a formula ϕ and tries to determine whether ϕ is satisfiable.

Satisifiability is an NP-complete problem. If we have a P-time algorithm, this algorithm must (1) occassionally give an incorrect answer or (2) be unable to handle some of the problems that is is presented with.

We'll look at two variations of such an algorithm.

For our SAT solver, we will assume that the connectives $\neg$ and $\wedge$ are adequate. All formulas will be transformed to use these connectives. We define the transformation $T(\phi)$ below.

$$\begin{aligned} T(p) &= p \\ T(\neg\phi) &= \neg T(\phi) \\ T(\phi \wedge \psi) &= T(\phi) \wedge T(\psi) \\ T(\phi \vee \psi) &= \neg(\neg T(\phi) \wedge \neg T(\psi)) \\ T(\phi \rightarrow \psi) &= \neg(T(\phi) \wedge \neg T(\psi)) \end{aligned}$$

Our SAT solver will take a formula using $\neg$ and $\wedge$ and transform the formula into a DAG. The DAG will be similar to a parse tree, but each literal will appear only once.

Let's look at an example

$$\begin{aligned} \phi &= p \wedge \neg(q \vee \neg p) \\ T(\phi) &= p \wedge \neg\neg(\neg q \wedge \neg\neg p) \end{aligned}$$

Our first step was to transform ϕ into $T(\phi)$, which uses the desired set of connectives. In figure 1.1, the numbers denote the order in which nodes were visitied. T and F denote values that a node must have in order to be true. Of course, the root must be marked true.

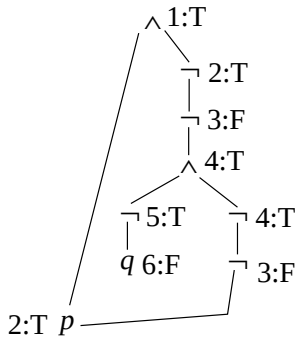

Figure 1.1: DAG for $T(\phi) = p \wedge \neg\neg(\neg q \wedge \neg\neg p)$

When marking these graphs, we don't necessarily stop after all nodes are marked. In this example, we started at the top and worked to the bottom. However, after reaching the bottom, we need to continue for as long as necessary to ensure that there are not conflicting assignments to any DAG node.

Similar to natural deduction, there are a series of rules that dictate how truth values propagate in the DAG.

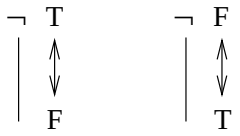

Figure 1.2: Rules ¬t and ¬f

Figure 1.2 shows that negation propagates true down to false, false down to true, true up to false, and false up to true.

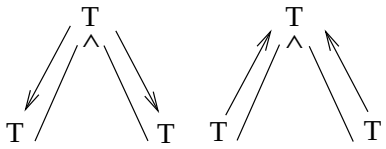

Figure 1.3: Rules ∧te and ∧ti

In figure 1.3, a true value at ∧ propagates true values to both of its child nodes. Likewise, if both children of ∧ are true, then ∧ must be true as well.

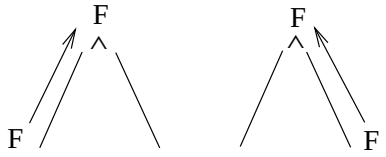

Figure 1.4: Rules ∧fl and ∧fr

In Figure 1.4, we see that if either child of ∧ is false, then ∧ must be false as well.

The rules in figure 1.5 state the following: if ∧ is false and one child of ∧ is true, then the other child must be false.

The general marking algorithm is as follows:

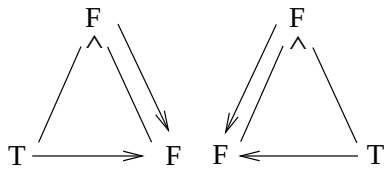

Figure 1.5: Rules ∧fll and ∧frr

- Mark the root node as T.
- Using rules, push values through the DAG, until no new values can be assigned.

The algorithm will end in one of three states:

1. Not all nodes are marked. In this case, we don't know whether $T(\phi)$ is satisifiable.
2. All notes are marked, and each node is marked with a single truth value. Here, $T(\phi)$ is satisfiable, and we have a satisfying assignment.
3. Some nodes are marked both T and F. Here, we know that $T(\phi)$ is not satisfiable.

If $T(\phi)$ has two or more satisfying assignments, then this algorithm is not sufficient to determine satisfiability. It will only work for formulas where there is a single satisfying assignment.

Consider the DAG in figure 1.6. Here, we have a formula of the form $\neg(\phi_1 \wedge \phi_2)$. We can assign a truth value to the root, and push a value one level below. However, after step 2 we're stuck; there are multiple

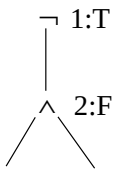

Figure 1.6: $\neg(\phi_1 \wedge \phi_2)$. We don't know

satisifying assignments for the children of $\wedge$ that will make $\wedge$ false.

Improvements to the SAT Solver

We can make some improvements to our P-time SAT solver. It still won't be perfect, but these improvements will allow it to find answers for a larger number of cases.

- After the first pass, if some nodes are unmarked, pick an unmarked test node n. Temporarily mark $n = \text{T}$ and push values around. Next, temporarily mark $n = \text{F}$ and push values around. If the two test values for n lead to contradictions, then declare the formula to be unsatisfiable.
- If both tests lead to application of the same mark to some previously unmarked node m, then we may mark m permanently.
- If one test value produces a contradiction and the other test value does not, then we can retain the test value that did not produce a contradition.
- If one test leaves every node marked but does not produce a contradiction, then we can declare the formula satisfiable and we have a satisfying assignment.

Again, these improvements will *NOT* yield a perfect algorithm. Satisfiability is an NP-complete problem, so we cannot hope to solve it with a polynomial time algorithm.

A related lesson: sometimes we are faced with the need to solve an NP-Complete problem. Sometimes, it's useful to approximate a best answer, if 'close enough' is sufficiently good.

Part 2

Predicate Logic

2.1 Lecture – 2/12/2007

Predicate logic is also known as *first-order logic*.

Predicate logic evolved from the need to express things that propositional logic could not express. Consider the following:

All men are mortal	(All A's are B's)
Socrates is a man	(s is an A)
Socrates is a mortal	(s is a B)

Propositional logic cannot decompose these statements.

Some building blocks for predicate logic

Predicates Predicates take one or more arguments and return a truth value. (Single argument predicates are called unary predicates, two-argument predicates are called binary predicates, etc).

Predicates represent properties of individuals

Constants A constant stands for a single individual.

Constants can also be thought of as *nullary functions* – functions that take no arguments, and always return a specific value.

Functions Functions take zero or more arguments, and return some single value.

Example 2.1.1: Unary predicates

The moon is green	$G(m)$
The wall is green	$G(w)$
π is irrational	$I(\pi)$

Example 2.1.2: Binary predicates

John and Peter are brothers	$B(j, p)$

Example 2.1.3: Functions

John's father	$f(j)$
John's father is an engineer	$E(f(j))$

2.2 Lecture – 2/21/2007

2.2.1 Predicate Logic

The ingredients of predicate logic are

- Predicate symbols (arity > 0)
- Functions (arity ≥ 0). Functions whose arity is zero are *constants*.
- Quantifiers – $\forall$, $\exists$.
- Variables

Example 2.2.1: John's father and George are brothers.

$$\mathsf{B}(f(j), g)$$

Example 2.2.2: Every cow is brown. In propositional logic, we express this as

$$\forall x(\mathsf{Cow}(x) \rightarrow \mathsf{Brown}(x))$$

Example 2.2.3: Some cows are brown.

$$\exists x(\mathsf{Cow}(x) \wedge \mathsf{Brown}(x))$$

This is **not** the same thing as $\exists x(\mathsf{Cow}(x) \rightarrow \mathsf{Brown}(x))$. With implication, the literal translation is "either x is not a cow or x is brown". (x could be a brown horse).

Example 2.2.4: Some birds don't fly.

$$\exists x(\mathsf{Bird}(x) \wedge \neg\mathsf{Fly}(x))$$

Example 2.2.5: Not every bird flies.

$$\neg(\forall x(\mathsf{Bird}(x) \rightarrow \mathsf{Fly}(x)))$$

We can manipulate this formula a little

$$\neg(\forall x(\neg\mathsf{Bird}(x) \vee \mathsf{Fly}(x)))$$
$$\neg(\forall x\neg(\mathsf{Bird}(x) \wedge \neg\mathsf{Fly}(x)))$$
$$\exists x(\mathsf{Bird}(x) \wedge \neg\mathsf{Fly}(x))$$

This example illustrates the relationship between $\forall$ and $\exists$.

Relationship between $\forall$ and $\exists$

$$\exists x\mathsf{P}(x) = \neg\forall x\neg\mathsf{P}(x) \tag{2.2.1}$$
$$\forall x\mathsf{P}(x) = \neg\exists x\neg\mathsf{P}(x) \tag{2.2.2}$$

2.2.2 Components of First Order Logic

Terms. Terms denote objects. Constants, variable, and functions are all examples of terms.

Formulas. Formulas denote truth values – they represent T or F, but not an individual thing.

Predicate Vocabulary = First order language. Our predicate vocabulary is

- A set $\mathcal{P}$ of predicate symbols, each having arity > 0.
- A set $\mathcal{F}$ of function symbols, each having arity ≥ 0. The set $\mathcal{F}$ includes constants.

We denote this vocabulary by $(\mathcal{F}, \mathcal{P})$.

Terms in $(\mathcal{F}, \mathcal{P})$:

- If x is a variable, then x is a term.
- If a is a nullary function, then a is a term.
- If f is an n-ary function and $t_1, \ldots, t_n$ are terms, then $f(t_1, \ldots, t_n)$ is also a term.

As a BNF, terms are

$$t ::= x \mid a \mid f(t_1, \ldots, t_n) \tag{2.2.3}$$

Formulas in $(\mathcal{F}, \mathcal{P})$:

- If P is a predicate symbol of arity > 0, and $t_1, \ldots, t_n$ are terms, then $P(t_1, \ldots, t_n)$ is a formula.
- If ϕ is a formula, then $\neg\phi$ is a formula.
- If ϕ, ψ are formulas, then $\phi \wedge \psi$, $\phi \vee \psi$, and $\phi \rightarrow \psi$ are also formulas.
- If ϕ is a formula and x is a variable, then $\forall x\phi$ and $\exists x\phi$ are formulas.
- If t_1, t_2 are terms, then $t_1 = t_2$ is a formula (= acts like a binary predicate).

As a BNF, formulas are:

$$\phi ::= P(t_1, \ldots, t_n) \mid \neg\phi \mid \phi \wedge \psi \mid \phi \vee \psi \mid \phi \rightarrow \psi \mid \forall x\phi \mid \exists x\psi \mid t_1 = t_2 \tag{2.2.4}$$

The binding rules for first order logic:

$\neg, \forall x, \exists x$
$\vee, \wedge$
$\rightarrow$ right associative

By *right-associative*, we mean that $q \rightarrow r \rightarrow s$ is interpreted as $q \rightarrow (r \rightarrow s)$, <u>not</u> $(q \rightarrow r) \rightarrow s$.

2.2.3 Bound Quantifiers and Free Quantifiers

Consider the formula

$$\exists x(P(x) \wedge \neg Q(x)) \vee R(y)$$

x is a *bound* variable, and y is a *free* variable. By analogy, in

$$\sum_{i=1}^{n} i^2$$

i is bound, and n is free.

The *scope* of a quantifier in a formula is the subformula immediately following the quantifier.

An occurrence of a variable x is bound if (1) it immediately follows a quantifier symbol, or (2) it is in the scope of a quantifer of the same variable. For example, in $\forall x P(x)$, there are two bound occurrences of x – (1) in $\forall x$ and (2) in $P(x)$.

If an occurrence of a variable is not bound, then it is said to be free.

Example 2.2.6: Consider the formula.

$$\forall x(P(x,y) \vee R(x)) \wedge \exists z Q(x,y,z)$$

Here

$P(x,y)$	x is bound, y is free
$R(x)$	x is bound
$Q(x,y,z)$	x, y are free; z is bound

Example 2.2.7: Condsider

$$\exists z(P(z) \wedge \forall z R(z))$$

Here, there are two different bindings for z. In $P(z)$, z is bound to $\exists z$. In $R(z)$, z is bound to $\forall z$. The following two formulas are equivalent:

$$\exists z(P(z) \wedge \forall y R(y))$$
$$\exists w(P(w) \wedge \forall z R(z))$$

With respect to binding, only the *innermost* quantifier matters.

2.2.4 Substitution in First-Order Logic

We notate substitution as

$$\phi[t/x] \tag{2.2.5}$$

This means "take ϕ, and replace all **free** occurrences of x with t".

Example 2.2.8: Substitution.

$$\begin{aligned}&((\exists x(R(x,y)) \wedge (\forall z P(x,z)))\,[f(y)/x]\\ =&(\exists x(R(x,y)) \wedge (\forall z P(f(y),z))\end{aligned}$$

Note that the bound occurrence of x was not replaced, but the unbound occurrence of x was.

Let's try to come up with an inductive definition for substitution:

$$\begin{aligned}R(t_1,\ldots,t_n)[t/x] &= R(t_1[t/x],\ldots,t_n[t/x])\\ (\neg\phi)[t/x] &= \neg\phi[t/x]\\ (\phi \circ \psi)[t/x] &= \phi[t/x] \circ \psi[t/x] && \circ \in \{\vee,\wedge,\rightarrow\}\\ (Qy\phi)[t/x] &= \begin{cases} Qy\phi & \text{if } x = y \text{ (no change)}\\ Qy(\phi[t/x]) & \text{otherwise}\end{cases} && \text{where } Q \in \{\forall,\exists\}\end{aligned}$$

The general idea behind substitution is as follows: when we substitute t for x, ϕ must say the same thing about t that it said about x.

Example 2.2.9: Some (correct) examples of substitution.

$\exists x(y = 2x)$	y is even
$(\exists x(y = 2x))[3z/y] = \exists x(3z = 2x)$	$3z$ is even

Example 2.2.10: An incorrect use of substitution.

$$(\exists x(y = 2x))[x + 2/y] = \exists x(x + 2 = x) \qquad 2 \text{ exists?}$$

Something went wrong here, we didn't preserve the meaning of ϕ as example 2.2.9 did.

The problem was that the substitution created a *new* bound occurrence of x (replacing the free variable y).

Definition 2.2.11: We say that t is *free for x in ϕ* if no free occurrence of x in ϕ is in the scope of a quantifer on a variable that occurs in t.

[It seems like we could equivalently say that substitution cannot create bound occurrences that did not exist prior to the substitution].

Let's try to formulate definition 2.2.11 inductively.

- Given $R(t_1, \ldots, t_n)$, then t is free for x. Terms don't have quantifiers.
- t is free for x in $\neg\phi$ IFF t is free for x in ϕ.
- t is free for x in $\phi \circ \psi$ IFF t is free for x in ϕ, and t is free for x in ψ. (As before, $\circ \in \{\vee, \wedge, \rightarrow\}$).
- t is free for x in $Qy\phi$ IFF
  - t does not contain y, **AND**
  - t is free for x in ϕ

  OR

  - x does not occur free in $Qy\phi$ (whereby substitution causes no change)

In general, when we write $\phi[t/x]$, we will assume that t is free for x in ϕ.

2.2.5 Natural Deduction Rules for First Order Logic

In this section, we'll cover a few of the natural deduction rules for first-order logic.

$$\frac{}{t = t} \qquad \text{=i. Equals introduction. This is an axiom} \tag{2.2.6}$$

$$\frac{t_1 = t_2, \phi[t_1/x]}{\phi[t_2/x]} \qquad \text{=e. Equals elimination} \tag{2.2.7}$$

Example 2.2.12: Prove $t_1 = t_2 \vdash t_2 = t_1$

1	$t_1 = t_2$	premise
2	$t_1 = t_1$	=i
3	$t_2 = t_1$	=e. Lines 1, 2. ϕ is $x = t_1$

Example 2.2.13: Prove $t_1 = t_2, t_2 = t_3 \vdash t_1 = t_3$

1	$t_2 = t_3$	premise
2	$t_1 = t_2$	premise
3	$t_1 = t_3$	$(t_1 = x)[t_2/x]$

An alternate proof:

1	$t_1 = t_2$	$(t_1 = x)[t_2/x]$
2	$t_2 = t_3$	premise
3	$t_1 = t_3$	=e. Lines 2, 1

With respect to introduction and elimination rules, $\forall$ will be similar to $\wedge$, while $\exists$ will be similar to $\vee$.

$$\frac{\forall x \phi}{\phi[t/x]}$$ Rule: $\forall$e

$$\frac{\boxed{\begin{array}{c} x_0 : \\ \vdots \\ \phi[x_0/x] \end{array}}}{\forall x \phi}$$ Rule: $\forall$i

In the $\forall$i rule, we are taking an *arbitrary* variable x_0 and proving that ϕ holds when x_0 is used for x. Because x_0 is an arbitrary variable, we infer that the substitution will hold for any arbitrary variable. x_0 has to be completely generic. It *cannot* occur anywhere else in the proof.

2.3 1st Order Logic Notes (H&R, Chapter 2)

Definition 2.3.1: Given a term t, a variable x, and a formula ϕ, we say that t is free for x in ϕ if no x leaf in ϕ occurs in the scope of $\forall y$ or $\exists y$ for any variable y occuring in t.

Restated: if t contains a variable y, then the substitution $[t/x]$ cannot cause y to become bound to a quantifier in ϕ.

In First-Order logic, the rules =i, and =e for equality are reflexive, symmetric, and transitive.

The rules for $\forall x$i and $\exists x$e involve the use of a dummy variable. (Huth and Ryan use x_0). The rules for dummy variables are as follows:

- x_0 must not exist outside the box.
- x_0 cannot be carried outside the box.

$\forall x\phi$ is similar to $\phi_1 \wedge \phi_2$.

- $\forall x\phi$ must prove that $\phi[x_0/x]$ will hold for *every* x_0 (though we use an arbitrary x_0 to prove it).
- $\phi_1 \wedge \phi_2$ must prove that ϕ_i holds for every $i = 1, 2$.

2.4 Lecture – 2/26/2007

2.4.1 Natural Deduction for Propositional Logic

In our last lecture, we covered the following rules:

$$\frac{\forall x \phi}{\phi[t/x]} \qquad \text{Rule: } \forall x\text{e}$$

$$\frac{\boxed{\begin{array}{c} x_0 \\ \vdots \\ \phi[x_0/x] \end{array}}}{\forall x \phi} \qquad \text{Rule: } \forall x\text{i}$$

Example 2.4.1: Prove $P(t), \forall x(P(x) \to \neg Q(x)) \vdash \neg Q(t)$.

1	$P(t)$	premise
2	$\forall x(P(x) \to \neg Q(x))$	premise
3	$P(t) \to \neg Q(t)$	$\forall$e. Line 2
4	$\neg Q(t)$	$\to$e. Lines 1, 3

Example 2.4.2: Prove $\forall x(P(x) \to Q(x)), \forall x(P(x)) \vdash \forall x Q(x)$.

	1	$\forall x(P(x) \to Q(x))$	premise
	2	$\forall x(P(x))$	premise
x_0	3	$P(x_0)$	$\forall$e. Line 2
	4	$P(x_0) \to Q(x_0)$	$\forall$e. Line 1
	5	$Q(x_0)$	$\to$e. Lines 3, 4
	6	$\forall x Q(x)$	$\forall$i. Lines 3–5

2.4.2 Existential Quantifiers

Where $\forall$ behaves similar to $\wedge$, $\exists$ behaves similar to $\vee$. The rules for $\exists$ are

$$\frac{\phi[t/x]}{\exists x \phi} \qquad \text{Rule: } \exists\text{i}$$

$$\frac{\boxed{\begin{array}{cc} x_0 & \phi[x_0/x] \\ & \vdots \\ & \chi \end{array}}}{\chi} \qquad \text{Rule: } \exists\text{e}$$

Example 2.4.3: Prove $\forall x \phi \vdash \exists x \phi$.

1	$\forall x \phi$	premise
2	$\phi[x/x]$	$\forall$e. Line 1
3	$\exists x \phi$	$\exists$i. Line 2

Example 2.4.4: Prove $\forall x(Q(x) \to R(x)),\ \exists x(P(x) \wedge Q(x)) \vdash \exists x(P(x) \wedge R(x))$.

	1	$\forall x(Q(x) \to R(x))$	premise
	2	$\exists x(P(x) \wedge Q(x))$	premise
x_0	3	$P(x_0) \wedge Q(x_0)$	assumption
	4	$Q(x_0)$	$\wedge$e. Line 3
	5	$Q(x_0) \to R(x_0)$	$\forall$e. Line 1
	6	$R(x_0)$	$\to$e. Lines 4–5
	7	$P(x_0)$	$\wedge$e. Line 3
	8	$P(x_0) \wedge R(x_0)$	$\wedge$i. Lines 7, 6
	9	$\exists x(P(x) \wedge R(x))$	$\exists$i. Line 8
	10	$\exists x(P(x) \wedge R(x))$	$\exists$e. Lines 2, 3–9

Example 2.4.5: Here, we show how violating the rules for x_0 allows us to derive an **incorrect** proof of

$\exists x P(x), \forall x(P(x) \to Q(x)) \vdash \forall y(Q(y))$.

	1	$\exists x P(x)$	premise
	2	$\forall x(P(x) \to Q(x))$	premise
x_0	3		
x_0	4	$P(x_0)$	assumption
	5	$P(x_0) \to Q(x_0)$	$\forall$e. Line 2
	6	$Q(x_0)$	$\to$e. Lines 4, 5
	7	$Q(x_0)$	$\exists$e. Lines 1, 4–6.
	8	$\forall y Q(y)$	$\forall$i. Lines 3–7

The problem occurs in line 7; the x_0 in lines 4–6 is being allowed to 'escape' from the box.

2.4.3 Provable Equivalences

A few lists of provable equivalences.

$$\neg\forall x\phi \dashv\vdash \exists x\neg\phi \tag{2.4.1}$$
$$\neg\exists x\phi \dashv\vdash \forall x\neg\phi \tag{2.4.2}$$

Assuming that x is not free in ψ:

$$\forall x\phi \wedge \psi \dashv\vdash \forall x(\phi \wedge \psi) \tag{2.4.3}$$
$$\forall x\phi \vee \psi \dashv\vdash \forall x(\phi \vee \psi) \tag{2.4.4}$$
$$\exists x\phi \wedge \psi \dashv\vdash \exists x(\phi \wedge \psi) \tag{2.4.5}$$
$$\exists x\phi \vee \psi \dashv\vdash \exists x(\phi \vee \psi) \tag{2.4.6}$$
$$\forall x(\psi \to \phi) \dashv\vdash \psi \to \forall x\phi \tag{2.4.7}$$
$$\exists x(\phi \to \psi) \dashv\vdash \forall x\phi \to \psi \tag{2.4.8}$$
$$\forall x(\phi \to \psi) \dashv\vdash \exists x\phi \to \psi \tag{2.4.9}$$
$$\exists x(\psi \to \phi) \dashv\vdash \psi \to \exists x\phi \tag{2.4.10}$$

$$\forall x\phi \wedge \forall x\psi \dashv\vdash \forall x(\phi \wedge \psi) \tag{2.4.11}$$
$$\exists x\phi \vee \exists x\psi \dashv\vdash \exists x(\phi \vee \psi) \tag{2.4.12}$$

$$\forall x\forall y\phi \dashv\vdash \forall y\forall x\phi \tag{2.4.13}$$
$$\exists x\exists y\phi \dashv\vdash \exists y\exists x\phi \tag{2.4.14}$$

One case that is *not* an equivalence:

$$\exists x\phi \wedge \exists x\psi \neq \exists x(\phi \wedge \psi) \tag{2.4.15}$$

The left side of (2.4.15) allows different values of x to make ϕ and ψ true. The right side requires the same value of x to make ϕ and ψ true.

Example 2.4.6: Prove $\neg\exists x\phi \vdash \forall x\neg\phi$.

	1	$\neg\exists x\phi$	premise
x_0	2		
	3	$\phi[x_0/x]$	assumption
	4	$\exists x\phi$	$\exists$ i. Line 3
	5	$\bot$	$\neg$e. Lines 4, 1
	6	$\neg\phi[x_0/x]$	$\neg$i. Lines 3–5
	7	$\forall x\neg\phi$	$\forall$i. Lines 2–6

Example 2.4.7: Prove $\forall x\neg\phi \vdash \neg\exists x\phi$.

	1	$\forall x\neg\phi$	premise
	2	$\exists x\phi$	assumption
x_0	3	$\phi[x_0/x]$	assumption
	4	$\neg\phi[x_0/x]$	$\forall$e. Line 1
	5	$\bot$	$\neg$e. Lines 3, 4
	6	$\bot$	$\exists$e. Lines 2, 3–5
	7	$\neg\exists\phi$	$\neg$i. Lines 2–6

Example 2.4.8: Prove $\forall x\phi \vee \psi \vdash \forall x(\phi \vee \psi)$, assuming x is not free in ψ.

	1	$\forall x\phi \vee \psi$	assumption
	2	$\forall x\phi$	assumption
x_0	3	$\phi[x_0/x]$	$\forall$e. Line 2
	4	$\phi[x_0/x] \vee \psi$	$\vee$i. Line 3
	5	$\forall x(\phi \vee \psi)$	$\forall$i. Lines 3–4
	6	ψ	assumption
x_0	7	$\phi[x_0/x] \vee \psi$	$\vee$i. Line 6
	8	$\forall x(\phi \vee \psi)$	$\forall$i. Line 7
	9	$\forall x(\phi \vee \psi)$	$\vee$e. Lines 1, 2–5, 6–8

In lines 4, 7 note that $\phi[x_0/x] \vee \psi = (\phi \vee \psi)[x_0/x]$, because x is not free in ψ.

Example 2.4.9: Prove $\forall x(\phi \vee \psi) \vdash \forall x\phi \vee \psi$.

	1	$\forall x(\phi \vee \psi)$	premise
	2	$(\forall x\phi) \vee \neg(\forall x\phi)$	LEM
	3	$\forall x\phi$	assumption
	4	$\forall x\phi \vee \psi$	$\vee$i. Line 4
	5	$\neg\forall x\phi$	assumption
	6	$\vdots$	
	7	$\exists x\neg\phi$	Proven earlier. Omitted here.
x_0	8	$\neg\phi[x_0/x]$	assumption
	9	$(\phi \vee \psi)[x_0/x]$	$\forall$e. Line 1
	10	$\phi[x_0/x]$	assumption
	11	$\bot$	$\neg$e. Lines 10, 8
	12	ψ	$\bot$e. Line 11
	13	ψ	assumption
	14	ψ	$\vee$e. Lines 9, 10–12, 13
	15	ψ	$\vee$e. *Why??*
	16	$\forall x\phi \vee \psi$	$\vee$i. Line 15
	17	$\forall x\phi \vee \psi$	$\vee$i. Lines 2, 3–4, 5–16

More Proofs involving non-free variables

Example 2.4.10: Prove $\phi \dashv\vdash \forall x\phi$, where x is not free in ϕ.

Because x is not free in ϕ, we can do substitutions without changing ϕ.

	1	ϕ	premise
x_0	2	ϕ	Copy. Line 1
	3	$\forall x\phi$	$\forall$i. Line 2

In the other direction

	1	$\forall x\phi$	premise
	2	ϕ	$\forall$e. Line 1

We can do the same thing with the existential quantifier.

Example 2.4.11: Prove $\phi \dashv\vdash \exists x\phi$, if x is not free in ϕ.

	1	ϕ	premise
	2	$\exists x\phi$	$\exists$i. $(\phi = \phi[x/x])$

In the other direction

	1	$\exists x\phi$	premise
x_0	2	$\phi[x_0/x]$	assumption $(\phi[x_0/x] = x)$
	3	ϕ	$\exists$e.

If x is not free in ϕ, we could even prove

$$\exists x\phi \dashv\vdash \forall x\phi$$

2.4.4 Semantics in First-Order Logic

Suppose we are given

$$\exists x\forall y\, P(x,y) \vDash \forall y\exists x\, P(x,y)$$

x and y are independant, but both must come from some domain of values. Similarly P also has a meaning.

For first-order logic, "valid" means that the truth semantics hold under any model.

2.5 Lecture – 2/28/2007

Example 2.5.1: This is a different version of the proof that appears on page 120, that doesn't use the 'questionable' reference to a boxed variable.

Prove: $\forall x(\phi \wedge \psi) \vdash (\forall x\phi) \wedge \psi$

	1	$\forall x(\phi \wedge \psi)$	premise
x_0	2		
	3	$(\phi \wedge \psi)[x_0/x]$	$\forall$e. Line 1. (Note: $= \phi[x_0/x] \wedge \psi[x_0/x]$)
	4	$\phi[x_0/x]$	$\wedge e_1$. Line 3
	5	$\forall x\phi$	$\forall$i. Lines 2–5
	6	$\phi \wedge \psi$	$\forall$e. Line 1
	7	ψ	$\wedge e_2$. Line 6
	8	$(\forall x\phi) \wedge \psi$	$\wedge$i. Lines 5, 7

2.5.1 Semantics of First-Order Logic

Suppose we are given the formula $\forall x\, P(f(x), y)$, how do we assign a truth value to this?

We need a *context* in which to evaluate the formula. Traditionally, this context is referred to as a *universe*.

Note that $\forall x\, P(f(x), y)$ says something about y (the free variable), but it says nothing about x (the bound variable).

Definition 2.5.2: A *model* for $(\mathcal{F}, \mathcal{P})$ consists of the following elements. Recall that $\mathcal{F}$ is a set of functions and $\mathcal{P}$ is a set of predicates.

1. A non-empty set A. This is the universe of values.
2. For any nullary function symbol $f \in \mathcal{F}$, we need a meaning: $f^{\mathcal{M}} \in A$. $f^{\mathcal{M}}$ is the *meaning* of f in the model $\mathcal{M}$.
3. For every n-ary function symbol $f \in \mathcal{F}$ having arity $n > 0$, we have a meaning $f^{\mathcal{M}} \colon A^n \to A$.
4. For every n-ary predicate $P \in \mathcal{P}$, our meaning is $P^{\mathcal{M}} \subseteq A^n$. ($P^{\mathcal{M}}$ is essentially a set of sets of terms that satisfy P under $\mathcal{M}$).
5. The equality predicate, $=$. In general, $=^{\mathcal{M}}$ can be interpreted as $\{(a, a) | a \in A\}$.

Our definition for model covers only functions and predicates (and constants, since constants are really nullary functions). This definition does *not* cover free variables. To handle free variables, we need a lookup table.

Definition 2.5.3: A *lookup table* for $\mathcal{M}$ is a function

$$l \colon \mathsf{var} \to A$$

Very simply, l gives values to variables.

We write $l[x \mapsto a]$ for the lookup table that maps x to a and $y \neq x$ to $l(y)$

In $l[x \mapsto a]$, x is a free variable and $a \in A$.

Lookup tables behave like functions.

2.5.2 Evaluating Formulas With Models

In first-order logic, the symbol $\models$ is overloaded. One meaning is semantic entailment. The other meaning has to do with the evaluation of a formula in a model.

We write

$$\mathcal{M} \models_l \phi$$

to mean that ϕ evaluates to T in the model $\mathcal{M}$ under the lookup function l.

Again, $\mathcal{M}$ is a model, l is a lookup table, and ϕ is a formula.

Let's build up a definition of evaluation.

- Let $t^{\mathcal{M},l}$ denote the value of the term t in $\mathcal{M}$ under l.
- $x^{\mathcal{M},l} = l(x)$, where x is a (free) variable.
- If f is a nullary function (constant), then $f^{\mathcal{M},l} = f^{\mathcal{M}}$.
- $f(t_1, \ldots, t_n)^{\mathcal{M},l} = f^{\mathcal{M}}(t_1^{\mathcal{M},l}, \ldots, t_n^{\mathcal{M},l})$

That covers evaluation of terms. Now, lets move on to formulas.

- $\mathcal{M} \models_l P(t_1, \ldots, t_n)$ IFF $(t_1^{\mathcal{M},l}, \ldots, t_n^{\mathcal{M},l}) \in P^{\mathcal{M}}$.
- $\mathcal{M} \models_l \neg\phi$ IFF not $\mathcal{M} \models_l \phi$.
- $\mathcal{M} \models_l \phi \vee \psi$ IFF $\mathcal{M} \models_l \phi$ or $\mathcal{M} \models_l \psi$
- $\mathcal{M} \models_l \phi \wedge \psi$ IFF $\mathcal{M} \models_l \phi$ and $\mathcal{M} \models_l \psi$
- $\mathcal{M} \models_l \phi \rightarrow \psi$ IFF not $\mathcal{M} \models_l \phi$ or $\mathcal{M} \models_l \psi$
- $\mathcal{M} \models_l \forall x\phi$ IFF for all $a \in A$, $\mathcal{M} \models_{l[x \mapsto a]} \phi$
- $\mathcal{M} \models_l \exists x\phi$ IFF for some $a \in A$, $\mathcal{M} \models_{l[x \mapsto a]} \phi$

Let's go back to our earlier question: what does $\forall x\, P(f(x), y)$ mean? Using our definitions:

> For all $a \in A$, $\mathcal{M} \models_{l[x \mapsto a]}$ makes $P(f(x), y)$ true.
> For all $a \in A$, $(f^{\mathcal{M}}(a), l(y)) \in P^{\mathcal{M}}$.

Suppose we had two lookup tables l, l', and that these two tables agree on all free variables of ϕ. Then

$$\mathcal{M} \models_l \phi \text{ IFF } \mathcal{M} \models_{l'} \phi$$

Definition 2.5.4: ϕ is a *sentence* if ϕ has no free variables.

Suppose ϕ is a sentence. Then, either $\mathcal{M} \models_l \phi$ for all l, or $\mathcal{M} \not\models_l \phi$ for all l.

Why is this the case? Lookup tables apply to free variables only. If ϕ is a sentence, then ϕ has no free variables, so it doesn't matter which lookup table we use. Every lookup table is equivalent!

Put another way $\mathcal{M} \models_l \phi$ is equivalent to $\mathcal{M} \models \phi$. If ϕ is a sentence, then the lookup table is irrelevant.

2.5.3 Semantic Entailment in First-Order Logic

As noted, $\models$ is also used to represent semantic entailment in first-order logic.

We write $\Gamma \models \psi$ to mean that "Gamma entails psi". As before, we use the convention of using Γ to represent a set of formulas.

If $\mathcal{M} \models_l \phi$ for all $\phi \in \Gamma$, then $\mathcal{M} \models_l \psi$.

To reiterate the overloaded notation:

$\Gamma \vDash \psi$	LHS is a set of formulas
$\mathcal{M} \vDash_l \psi$	LHS is a model

Definition 2.5.5: We say that ψ is *satisfiable* if there are $\mathcal{M}$, l such that $\mathcal{M} \vDash_l \psi$.

Definition 2.5.6: We say that ψ is valid if $\mathcal{M} \vDash_l \psi$ for **all** $\mathcal{M}$, l.

This is a very big statement. In the realm of models, 'all' really means 'every thing you could possibly think of'.

Definition 2.5.7: $\Gamma = \phi_1, \ldots, \phi_n$ is *satisfiable* if there is $\mathcal{M}$, l such that $\mathcal{M} \vDash_l \phi$ for all $\phi \in \Gamma$.

Question: if $\Gamma = \emptyset$, is Γ still satisfiable? Yes, $\Gamma = \emptyset$ is satisfiable.

2.5.4 Soundness and Completeness of First-Order Logic

Recall,

Soundness A syntactic notion. If $\Gamma \vdash \phi$, then $\Gamma \vDash \phi$. Note that Γ may be a finite set of formulas, or Γ may be an infinite set of formulas.

Completeness If $\Gamma \vDash \phi$, then $\Gamma \vdash \phi$.

Soundness is the more important quality. Soundness allows us to trust what we have proven syntactically.

Soundness and Completeness hold for First-Order logic. (We're not going to explore a proof here, but the principles hold).

2.5.5 The Validity Problem

The validity problem is as follows: Given a formula ϕ, is ϕ valid? (i.e. is $\vDash \phi$ true?)

In propositional logic, validity is a decidable problem. We can construct a truth table for the given formula and see if all rows evaluate to T. Although there's no efficient way to determine validity for propositional logic, it's still a computable problem.

In first-order logic, validity is *not* decidable. Huth and Ryan give a proof by reduction from the Post Correspondence Problem.

Post Correspondence Problem

In the post correspondence problem, we are given a n dominos

$$\boxed{\begin{array}{|c|}\hline s_1 \\ \hline t_1 \\ \hline \end{array}} \cdots \boxed{\begin{array}{|c|}\hline s_n \\ \hline t_n \\ \hline \end{array}} \tag{2.5.1}$$

Each s_i and t_i is a binary string. No s_i or t_i can be the empty string.

The goal is to arrange the dominos so that the same word appears on the top and bottom halves. However, we are allowed to reuse dominos as many times as we'd like. This is the part that makes PCP undecidable — there's no bound on the computation.

PCP is reducible to the halting problem.

To show that validity is not decidable, we will reduce PCP to validity. This will show validity for first-order logic to be undecidable. (A solution to the validity problem would allow us to solve PCP by transforming an instance of the PCP into the validity problem).

Given an instance of the PCP (2.5.1), we will construct a formula ψ such that the PCP instance has a solution IFF ψ is valid.

Our model description for ψ

$$
\begin{aligned}
A &= \{0,1\}^* \\
f_0(w) &= w0 && \text{append 0 to } w \\
f_1(w) &= w1 && \text{append 1 to } w \\
P(s,t) &= s = t && s = s_1, \ldots, s_n,\ t = t_1, \ldots, t_n
\end{aligned}
$$

Representing a string with f_0 and f_1 will involve a fair amount of recursion. As a notational convenience, let

$$f_{b_1 \ldots b_k} = f_{b_k}(f_{b_{k-1}}(f_{b_{k-2}}(\cdots f_{b_1}(e)\cdots)))$$

[to be continued next lecture]

2.6 Lecture – 3/5/2007

2.6.1 First-Order Logic and the Post Correspondence Problem

Example 2.6.1: Here's an instance of the PCP that has a solution:

b	a	ca	abc
ca	ab	a	c

The solution uses dominos $\{2, 1, 3, 2, 4\}$.

If a solution to an instance of PCP exists, then finding that solution is a *recursively enumerable* problem. (The term *Turing recognizable* means the same thing as recursively enumerable). If a solution exists, then brute force search will find it eventually. However, you cannot tell when a solution does not exist.

Our goal for this section is as follows: given a PCP instance C, we will produce a formula ϕ such that C has a solution IFF ϕ is logically satisfiable.

Our formula makes use of the following elements

e	A constant, which denotes the empty string ϵ
f_0, f_1	Unary functions that append symbols to a given string
P	A binary predicate

If s is a string then

$f_0(s) = s0$	Append 0
$f_1(s) = s1$	Append 1

Given strings s, t, the meaning of $P(s, t)$ is as follows: there exists a sequence of indexes $i_1, \ldots, i_k$ such that s is the term representing $s_{i_1}, \ldots, s_{i_k}$ and t is the term representing $t_{i_1}, \ldots, t_{i_k}$.

P only checks to see if s and t can be formed by a sequence of dominos. P does *not* check if s and t are the same string. P means "s, t are constructible from dominos".

Finally, we'll also use a notational shortcut

$$f_{b_1 \ldots b_l} = f_{b_l}(f_{b_{l-1}}(\cdots(f_{b_2}(f_{b_1}(e)))\cdots))$$

The sentence describing the PCP is

$$\phi = \phi_1 \wedge \phi_2 \rightarrow \phi_3 \tag{2.6.1}$$

where

$$\phi_1 = \bigwedge_{i=1}^{k} P(f_{s_i}(e), f_{t_i}(e)) \tag{2.6.2}$$

$$\phi_2 = \forall u \forall v \left(P(u, v) \rightarrow \bigwedge_{i=1}^{k} P(f_{s_i}(u), f_{t_i}(v)) \right) \tag{2.6.3}$$

$$\phi_3 = \exists z \, P(z, z) \tag{2.6.4}$$

Intuitively,

- ϕ_1 allows us to build strings from sequences of dominos, starting from the empty string.
- ϕ_2 says that when we have strings s, t, then we can append to them only strings that can be formed by adding to the sequence of dominos.
- ϕ_3 says that z can be formed from a sequence of dominoes, where z appears on both the top half and the bottom half.

A Proof that This Reduction Works

Part One: Suppose that $\vDash \phi$ is valid (where $\phi = \phi_1 \wedge \phi_2 \rightarrow \phi_3$). Then ϕ must be valid for our model of the Post Correspondence Problem. We need to give a formal defintion of our model that demonstrates its ϕ's validity.

Let

$$\mathcal{M} = (A, e^{\mathcal{M}}, f_0^{\mathcal{M}}, f_1^{\mathcal{M}}, P^{\mathcal{M}})$$

where

$$\begin{aligned} A &= \{0,1\}^* \\ e^{\mathcal{M}} &= \epsilon \\ f_0^{\mathcal{M}}(w) &= w0 \\ f_1^{\mathcal{M}}(w) &= w1 \\ P^{\mathcal{M}}(s,t) = \; & \{(s,t) \mid \text{ there is a sequence of indices } i_1, \ldots, i_m \text{ such that } s = s_{i_1}, \ldots, s_{i_m} \\ & \quad \text{and } t = t_{i_1}, \ldots, t_{i_m}\} \end{aligned}$$

If $\vDash \phi$ is valid, then $\mathcal{M} \vDash \phi$ is valid.

If t is a variable-free term, then $(f_s(t))^{\mathcal{M}} = t^{\mathcal{M}} \cdot s$. So $\mathcal{M} \vDash \phi_1$.

If the pair $(s,t) \in P^{\mathcal{M}}$, then the pair $(ss_i, tt_i) \in P^{\mathcal{M}}$ for $i = 1, \ldots, k$. So, $\mathcal{M} \vDash \phi_2$.

Since $\phi_1 \wedge \phi_2 \rightarrow \phi_3$, and $\phi_1 \wedge \phi_2$ holds, ϕ_3 must hold as well. The definitions of ϕ_3 and $P^{\mathcal{M}}$ says that there is a solution to the PCP instance C.

So, if ϕ is valid, then then C has a solution.

Part Two. Suppose C has a solution. We must show that ϕ is valid for *any* model $\mathcal{M}'$

By our definition of ϕ, if $\mathcal{M}' \nvDash \phi_1$ or $\mathcal{M}' \nvDash \phi_2$ then we are done — ϕ is vacuously valid. The harder part is handling when $\mathcal{M}' \vDash \phi_1 \wedge \phi_2$. In that case, we must verify $\mathcal{M}' \vDash \phi_3$ as well.

We do this by *interpreting* binary strings in the domain of values A' for the model $\mathcal{M}'$. Let us define a function

$$\mathsf{interpret}\colon \{0,1\}^* \rightarrow A$$

$$\begin{aligned} \mathsf{interpret}(\epsilon) &= e^{\mathcal{M}'} \\ \mathsf{interpret}(s0) &= f_0^{\mathcal{M}'}(\mathsf{interpret}(s)) \\ \mathsf{interpret}(s1) &= f_1^{\mathcal{M}'}(\mathsf{interpret}(s)) \\ \mathsf{interpret}(b_1 \ldots b_l) &= f_{b_l}^{\mathcal{M}'}(f_{b_{l-1}}^{\mathcal{M}'}(\cdots(f_{b_1}^{\mathcal{M}'}(e^{\mathcal{M}'}))\cdots)) \end{aligned}$$

Since, $\mathcal{M}' \vDash \phi_1$, we can conclude that

$$(\mathsf{interpret}(s_i), \mathsf{interpret}(t_i)) \in P^{\mathcal{M}'} \qquad \text{for } i = 1, \ldots, k$$

Since $\mathcal{M}' \vDash \phi_2$, we can conclude that

$$(\mathsf{interpret}(ss_i), \mathsf{interpret}(tt_i)) \in P^{\mathcal{M}'} \qquad \text{for } i = 1, \ldots k$$

Starting with $(s,t) = (s_{i_1}, t_{i_1})$, we can repeatedly apply the previous formula to obtain

$$(\mathsf{interpret}(s_{i_1} \dots s_{i_n}), \mathsf{interpret}(t_{i_1} \dots t_{i_n})) \in P^{\mathcal{M}'}$$

Since $s_{i_1} \dots s_{i_n}$ and $t_{i_1} \dots t_{i_n}$ form a solution to C, they are equal. Therefore, $\mathsf{interpret}(s_{i_1} \dots s_{i_n})$ and $\mathsf{interpret}(t_{i_1} \dots t_{i_n})$ are the same elements in A'. Therefore $\exists z\, P(z,z)$ is in $\mathcal{M}'$ and $\mathcal{M}' \vDash \phi_3$.

This shows a reduction from PCP to the validity problem for first-order logic. Because PCP is not decidable, the validty problem for first-order logic is not decidable.

2.6.2 Implications of the Undecidability of the Validity Problem for First-Order Logic

Having shown that validity is undecidable for first-order logic, we may conclude two additional things:

1. ϕ is satisfiable IFF $\neg\phi$ is valid. Because we can't compute validity, we can't compute satisifiability either.
2. $\vDash \phi$ IFF $\vdash \phi$. Because validity is not computable, provability is not computable either.

Validity is recursively enumerable. To check validity, one must consider all finite models and all infinite models. $\vDash \phi$ is R.E. because of soundness and completeness. We can try all possible proofs. If we find a proof, soundness assures us that the proof will hold.

The set of non-valid formulas is not recursively enumerable. If the set of non-valid formulas were RE, then validity would be recursive (aka, Turing Computable). A set is recursive IFF the set and its compliment are recursively enumerable.

As an aside, there are formulas of first-order logic that are not valid, but are true for all finite structures. Simply enumerating all possible finite structures is not sufficient to test for validity.

2.6.3 Expressiveness of First-Order Logic

First-Order logic is pretty powerful, but there are concepts it cannot express.

Theorem 2.6.2 (Compactness Theorem)**:** If Γ is a set of sentences and all finite subsets of Γ are satisfiable, then Γ is satisfiable.

Proof: Suppose that Γ were unsatisfiable. Because Γ is unsatisfiable, then $\Gamma \vDash \bot$ and $\Gamma \vdash \bot$ (by completeness).

Let Δ be some finite subset of Γ. We have $\Delta \vdash \bot$, since proofs have finite length. Because $\Delta \vdash \bot$, soundness and completeness tell us that $\Delta \vDash \bot$.

$\Delta \vDash \bot$ contradict the assumption that all sentences in Γ are consistent; Γ must be satisifiable. □

Theorem 2.6.3 (Löwenheim-Skolem Theorem)**:** If Γ is a set of sentences, and for all $n \geq 1$, Γ has a model $\mathcal{M}$ with at least n elements, then Γ has a model with infinitely many elements.

The phrase "for all $n \geq 1$, Γ has a model $\mathcal{M}$" refers to $\mathcal{M} \vDash \phi$ for all $\phi \in \Gamma$.

The phrase "a model $\mathcal{M}$ with at least n elements", means that the universe A of $\mathcal{M}$ has at least n elements.

2.7 Chapter 2.4 Notes

2.7.1 Semantics of Predicate Logic

It's generally easy to show $\Gamma \vdash \psi$ — just write a proof. It's much harder to show that no proof exists.

$\Gamma \vDash \psi$ is just the opposite. It's generally easy to show a counterexample where $\Gamma \nvDash \psi$. Showing $\Gamma \vDash \psi$ for all possible models $\mathcal{M}$ is usually difficult.

The evaluation of any first-order logic formula requires a *universe* of values.

Definition 2.7.1 (Model)**:** A *model* for $(\mathcal{F}, \mathcal{P})$ is

$$\mathcal{M} = (A, c^{\mathcal{M}}, f^{\mathcal{M}}, P^{\mathcal{M}})$$

where

- A is a universe of concrete values
- $c^{\mathcal{M}}$. For each nullary function (constant) we need a concrete $a \in A$ that the constant stands for.
- $f^{\mathcal{M}}$. For each n-ary function $f^{\mathcal{M}} \colon A^n \to A$, we need a concrete function definition.
- $P^{\mathcal{M}}$. For each n-ary predicate, we need a set of n-ary tuples that satisfy the predicate. For an n-ary predicate P, $P^{\mathcal{M}} \subseteq A^n$.

f and P are simply symbols. By contrast, $f^{\mathcal{M}}$ is a concrete function and $P^{\mathcal{M}}$ is a concrete relation. The notion of "model" is very liberal and very open-ended.

There is one area that our definition of model does not address: free variables. To handle free variables, we need a *lookup table*

$$l \colon \mathsf{var} \to A$$

Lookup tables are also denoted by $l[x \mapsto a]$. $l[x \mapsto a]$ means that l maps x to $a \in A$, and for all $y \neq x$, l maps y to $l(y)$.

If a first-order logic formula is a *sentence*, then validity holds (or does not hold) irrespective of the look-up table. If ϕ is a sentence, then ϕ has no free variables; if ϕ has no free variables, then the lookup table is irrelevant.

The semantics of first-order logic impose a special meaning on the symbol $=$, which we denote $=^{\mathcal{M}}$. (a, b) is in the relationship $=^{\mathcal{M}}$ IFF a and b are the same elements of A.

2.7.2 Semantic Entailment (Section 2.4.2)

Definition 2.7.2 (Semantic Entailment)**:** Let $\Gamma = \phi_1, \ldots, \phi_n$. We say that Γ semantically entails ψ, or

$$\phi_1, \ldots, \phi_n \vDash \psi$$

IFF ψ is true whenever all $\phi_1, \ldots, \phi_n \in \Gamma$ are true.

Some useful rules for semantic entailment.

1. $\Gamma \vDash \psi$ holds IFF for all models $\mathcal{M}$ and lookup tables l, whenever $\mathcal{M} \vDash_l \phi$ holds for all ϕ in Γ, then $\mathcal{M} \vDash_l \psi$ holds as well.
2. Formula ψ is satisfiable IFF there is some model $\mathcal{M}$ and some environment l such that $\mathcal{M} \vDash_l \psi$ holds.

3. Formula ψ is valid IFF $\mathcal{M} \models_l \psi$ holds for all models $\mathcal{M}$ and environments l in which we can check ψ.
4. The set Γ is *consistent* or satisfiable IFF there is a model $\mathcal{M}$ and a lookup table l such that $\mathcal{M} \models_l \phi$ holds for all $\phi \in \Gamma$.

2.7.3 Decidability of First-Order Logic

Establishing $\mathcal{M} \models \psi$ is undecidable if the universe of $\mathcal{M}$ is infinite. If the universe is infinite, there's no way that we can check each value.

Determining whether $\phi_1 \ldots \phi_n \models \psi$ is also an undecidable problem. Validity requires entailment to hold under all possible models. Because there are an infinite number of potential models, we cannot check this programatically either.

By contrast, validity and satisfiability are decidable problems in propositional logic: construct a truth table and examine it. This is not an efficient approach, but it can be done programatically.

2.7.4 Soundness and Completeness

As with predicate logic,

$$\phi_1, \ldots, \phi_n \vdash \psi \text{ IFF } \phi_1, \ldots, \phi_n \models \psi \tag{2.7.1}$$

Soundess and completeness hold for first-order logic.

2.8 Lecture – 3/7/2007

2.8.1 Löwenheim-Skolem Theorem

Last time we discussed the Löwenheim-Skolem Theorem (See Theorem 2.6.3, page 55). If Γ is a set of sentences and for all $n \geq 1$ Γ has a model of size n, then Γ has an infinite model.

The phrase "Γ has a model of size n" means that there is a model $\mathcal{M} = (A, \ldots)$ and for all $\phi \in \Gamma$, $\mathcal{M} \vDash \phi$ and $|A| \geq n$.

Indirectly, this theorem is saying that first-order logic gives us no way to say "the universe is finite".

Proof (Löwenheim-Skolem Theorem): For all $n \geq 1$, let ϕ_n be the formula

$$\exists x_1 \exists x_2 \ldots \exists x_n \left(\bigwedge_{i \neq j}^{n} \neg(x_i = x_j) \right) \tag{2.8.1}$$

Equation (2.8.1) states that there are *at least* n elements in our universe.

Let $\Delta = \Gamma \cup \{\phi_n \mid n \geq 1\}$. Δ is adding n formulas to the set Γ.
Let Δ' be a finite subset of Δ.
Let $m = \max\{n \mid \phi_n \in \Delta'\}$.
Let $\mathcal{M}$ be a model for Γ with at least m elements.

Because $\mathcal{M}$ is a model for Δ, $\mathcal{M}$ is also a model for Δ'.

By the compactness theorem, Δ has a model $\mathcal{M}'$ which is an infinite model of Γ. □

2.8.2 First-Order Logic and Directed Graphs

The compactness theorem also allows us to prove some things about first-order logic and directed graphs.

A directed graph is a set of vertices and edges: $G = (V, E)$.

E is equivalent to a binary predicate that states whether two nodes are connected. Let us represent the set E with the logical predicate R.

A specific instance of a graph will serve as a model.

Graphs are commonly used in program verification. Nodes represent states and edges represent state transitions. Typical verification problems ask the question "is there a path from a good state to a bad state?".

We define our relation R

$$R = \{(u, v) \mid \text{there is a path from } u \text{ to } v\}$$

We refer to R as the *reachability relation*.

R is the reflexive, transitive closure of the set of edges. (Reflexive, because we consider (u, u) as a path of length zero from u to itself).

The question: *is reachability expressible in predicate logic?*

More formally, is there a formula ϕ with a single binary predicate R and free variables u, v such that for all models $\mathcal{M}$ and lookup tables l, $\mathcal{M} \vDash_l \phi$?

(Note: because we're working with free variables, we're actually concerned about paths between $l(u)$ and $l(v)$).

The answer: no such formula exists. There is no formula in predicate logic that can show reachability for all directed graphs.

If such a formula did exist, it would have the form

$$\phi = (u = v) \vee R(u, v) \vee \exists x \, (R(u, x) \wedge R(x, v)) \wedge \ldots$$

The formula would be infinitely long, which predicate logic doesn't allow. Allowing infinitely long formulas would invalidate the compactness theorem, and invalidating the compactness theorem would invalidate soundness and completeness.

Theorem 2.8.1: Reachability is not expressible in predicate logic: there is no formula ϕ, whose only free variables are u and v and whose only predicate symbol (of arity 2) is R, such that ϕ holds IFF there is a path in that graph from the node associated to u to the node associated to v.

PROOF: Suppose there were such a formula ϕ. Let c and c' be constants. Let ϕ_n be the formula expressing that there is a path of length n from u to v.

$$\begin{aligned}
\phi_0 &= c = c' \\
\phi_1 &= R(c, c') \\
\phi_2 &= \exists x \, (R(u, x) \wedge R(x, v)) \\
&\vdots \\
\phi_n &= \exists x_1 \ldots \exists x_{n-1} \, (R(c, x_1) \wedge R(x_1, x_2) \wedge \cdots \wedge R(x_{n-1}, c'))
\end{aligned}$$

Let us define

$$\Delta = \{\neg \phi_i \mid i \geq 0\} \cup \{\phi[c/u][c'/v]\}$$

Δ is unsatisfiable: $\{\neg \phi_i \mid i \geq 0\}$ says that no path exists, while $\{\phi[c/u][c'/v]\}$ says that a path does exist.

However, every finite subset of Δ is satisfiable since there are paths of any finite length. Therefore, by the compactness theorem, Δ is satisfiable. This is a contradiction, since Δ was designed to be an unsatisfiable formula. Therefore, no such ϕ exists. □

So we see that first-order logic is powerful, but it has its shortcomings. It's powerful enough to enter the realm of undecidability (the validity and satisfiability problems), but there are some simple problems that it cannot express (reachability).

2.8.3 Second-Order Logic

First-order logic cannot express reachability, but second-order logic can. Where first-order logic gives us freedom over individual objects, second-order logic gives us freedom over predicates.

Second-order logic has no compactness theorem.

2.8.4 Existential Second-Order Logic

Existential Second-Order Logic formulas have the form

$$\exists \, P \, \phi$$

Where P is a predicate symbol and ϕ is a propositional logic, or first-order logic formula.

Let $\mathcal{M}$ be a model for the language of $(\phi - P)$.

Let $T \subseteq A \times A$. $\mathcal{M}_T$ will be a full model for ϕ, where $P^{\mathcal{M}_T} = T$.

We will have $\mathcal{M} \vDash_l \exists P\, \phi$ IFF there is a $T \in A \times A$ with $\mathcal{M}_T \vDash_l \phi$.

Non-Reachability can be expressed in second-order logic as

$$\exists P \forall x \forall y \forall z \, (C_1 \wedge C_2 \wedge C_3 \wedge C_4)$$

where each C_i is a horn clause

$$C_1 = P(x, x) \tag{2.8.2}$$
$$C_2 = P(x, y) \wedge P(y, z) \rightarrow P(x, z) \tag{2.8.3}$$
$$C_3 = P(u, v) \rightarrow \bot \tag{2.8.4}$$
$$C_4 = R(x, y) \rightarrow P(x, y) \tag{2.8.5}$$

C_1 states that P is reflexive.
C_2 states the P is transitive.
C_3 ensures that there is no P path from u to v.
C_4 says that any R edge is also a P edge.

Which is a good definition of non-reachability in a directed graph. It expresses reflexivity and transitivity, and also that you cannot get from u to v.

2.8.5 Universal Second-Order Logic

Formulas of universal second-order logic have the form

$$\forall\, P\, \phi$$

Where existential second-order logic allowed to express non-reachability, universal second-order logic allows us to express reachability.

$$\phi = \forall P \exists x \exists y \exists z (\neg C_1 \vee \neg C_2 \vee \neg C_3 \vee \neg C_4) \tag{2.8.6}$$

where each C_i is as defined in (2.8.2) - (2.8.5).

Theorem 2.8.2: Let $\mathcal{M} = (A, R^{\mathcal{M}})$ be any model. Then formula (2.8.6) holds under lookup table l in $\mathcal{M}$ IFF $l(v)$ is R-reachable from $l(u)$ in $\mathcal{M}$.

PROOF: **Case 1**. Suppose that $\mathcal{M} \vDash \phi$ (for ϕ in (2.8.6)) holds for all interpretations of $\mathcal{M}$. Then it also holds for the model where P is the reflexive transitive closure of R.

In the model where P is the reflexive transitive closure of R, only the clause $\neg C_3$ will hold. But this means that $\mathcal{M} \vDash P(u, v)$ has to hold, so there is path of finite length from $l(u)$ to $l(v)$.

Case 2. Let $l(v)$ be R-reachable from $l(u)$ in $\mathcal{M}$. For any interpretation where P is not reflexive, not transitive, or does not contain R, ϕ will still hold. One of the clauses $\neg C_1$, $\neg C_2$ or $\neg C_4$ will be true.

If the interpretation has P being the reflexive transitive closure of R, then P will certainly contain R. The clauses $\neg C_1$, $\neg C_2$, $\neg C_4$ will be false, but $\neg C_3$ will be true, because $(l(u), l(v))$ is in the reflexive transitive closure by assumption. Therefore ϕ still holds. □

2.8.6 Program Verification

There are two schools of program verification: model-based and proof-based.

Model-Based Verification Model-based verification uses $\mathcal{M} \vDash \phi$, but assumes that $\mathcal{M}$ is finite. If $\mathcal{M}$ is finite, then $\mathcal{M} \vDash \phi$ is decidable. You can try every possible combination.

Model-based verification is *retrospective*. You do it after you've constructed the model.

Proof-Based Verification Proof-based verification relies on soundness and completeness. We try to establish $\phi_1, \ldots, \phi_n \vDash \phi$ by constructing a syntactic proof for $\phi_1, \ldots, \phi_n \vdash \phi$.

Proof-based verification is *prospective*. You can do it before the model has been constructed. This can be help you decide when the model is sufficient. If you can prove ϕ, you're done.

Part 3

Program Verification

3.1 Lecture – 3/12/2007

3.1.1 An Introduction to Verification

There are three ingredients in program verification:

1. A means of modeling the system. Typically this is mathematical model.
2. Specifying a language for system properties.
3. A way of verifying that the model matches (or fails to match) the desired program behavior. (A verification method).

Verification methods can be classified according to several criteria:

- **Proof-based vs. Model Based**.

  Proof-based verification models a system by a set of formulas, Γ. The specification is another formula ϕ. Verification involves finding a proof for $\Gamma \vdash \phi$ in some formal system. Assuming that the formal system is sound, we can determine whether $\Gamma \vDash \phi$.

  Model-based methods use a model $\mathcal{M}$. The specification is a formula ϕ. Verification involves showing that $\mathcal{M} \vDash \phi$.

  If $\mathcal{M}$ is a finite model, then the verification problem is decidable (but not necessarily efficient).

- **Degree of Automation**.

  The verification process may be (1) fully-automatic, (2) computer assisted or (3) manual.

  Fully-automatic verification works better for model-based systems.

  Fully-manual verification works better for proof-based systems.

- **Full Verification vs. Property Verification**.

  Full Verification tries to develop a complete specification of the program: what must be true about its inputs, what must be true about its outputs, as well as the details of the program's behavior. While this method may be practical for small programs, it is usually impractical for medium to large programs.

  Property Verification does not try to produce a complete description of the program. Instead, it focuses on certain aspects of the program's behavior.

Property Verification is weaker than full verification, but easier to do. With property verification, you can at least make assertions about some aspects of the system.

- **Intended Domain**.

  What type of program are you verifying? Is it sequential or concurrent? Is it reactive or terminating?

- **Pre-development vs. Post-development**.

  When is the verification process applied? It's preferably to apply verification earlier in the development process, when mistakes are less costly to rectify.

3.1.2 Model-Checking

We will focus on model checking as a verification mechanism. Some characteristics of model checking are:

- It's model-based
- Fully-automated
- Uses property verification
- It applies to the domain of concurrent, reactive systems
- It's done in Post-development
- It's based in *linear temporal logic* (LTL). LTL is a form of logic that involves time.
- The model will be based on a *transition system*.
- If the model is inconsistent (i.e., if $\mathcal{M} \not\models \phi$), then the verification process will produce a counterexample.

Alloy is a Hybrid model verifier. It limits the size of the model search space, under the assumption that counterexamples can usually be found in small models.

3.1.3 Temporal Logic Systems

Temporal logics are formal logic systems that involve time. We'll be concerned with two temporal logic systems:

1. **Linear Temporal Logic**. LTL assumes that time proceeds linearly along a single path. LTL allows a single 'next' step in time.

   ```
   S1 -> S2 -> S3 -> ...
   ```

2. **Branching Temporal Logic**. Branching temporal logic allows multiple 'next' steps. For example:

   ```
                         -------> S4 ------> ...
                        |
        -----> S2 ---->|
       |                |
   S1-->|                -------> S3 ------> ...
       |
        ----------- S5 -------> ...
   ```

We'll focus on Linear Temporal Logic first.

3.1.4 Linear Temporal Logic

LTL views time as a sequence of states, extending (perhaps infinitely far) into the future. There may be several possible paths, but only path will be realized.

LTL makes use of atoms: p, q, r, etc. Atoms stand for facts that may be true or false in the system. For example, "printer lj2 is busy", "process X is suspended", etc.

In EBNF, the syntax for LTL is as follows:

$$
\begin{array}{lll}
\phi ::= \top & & \\
\quad | \ \bot & & \\
\quad | \ p & & \text{where } p \text{ is an atom} \\
\quad | \ (\neg\phi) & & \text{where } \phi \text{ is any LTL formula} \\
\quad | \ (\phi \wedge \phi) & & \text{and the other } \phi \text{ is any other LTL formula} \\
\quad | \ (\phi \vee \phi) & & \\
\quad | \ (\phi \rightarrow \phi) & & \\
\quad | \ (\mathrm{X}\ \phi) & & \\
\quad | \ (\mathrm{F}\ \phi) & & \\
\quad | \ (\mathrm{G}\ \phi) & & \\
\quad | \ (\phi\ \mathrm{U}\ \phi) & & \\
\quad | \ (\phi\ \mathrm{W}\ \phi) & & \\
\quad | \ (\phi\ \mathrm{R}\ \phi) & &
\end{array}
$$

The symbols X, F, G, U, W, and R are called *temporal connectives*. Their names are

Connective	Name
X	Next
F	Future
G	Global
U	Until
W	Weak Until
R	Release

The precedence order of LTL connectives is

$\neg, \mathrm{X}, \mathrm{F}, \mathrm{G}$	unary connectives
$\mathrm{U}, \mathrm{W}, \mathrm{R}$	LTL binary connectives
$\wedge, \vee$	
$\rightarrow$	

3.1.5 Semantics of Linear Temporal Logic

LTL is based on a transition system. A transition system consists of *states* (static structures) and *transitions* (dynamic structures).

Definition 3.1.1 (Transition System)**:** A transition system, $\mathcal{M} = (S, \rightarrow, L)$ is a set of States S, a binary transition relation $\rightarrow$, and a labeling function L.

$\rightarrow$ is a binary relation on S. For every $s \in S$, there will be some $s' \in S$ such that $s \rightarrow s'$. Dead ends are not allowed in this transition system.

The labeling function, $L\colon S \rightarrow \mathcal{P}(atoms)$. For a state s, $L(s)$ contains the set of atoms that are true in s.

Example 3.1.2: Figure 3.1 shows an LTL transition system. In this system,

- S_0 is the starting state.

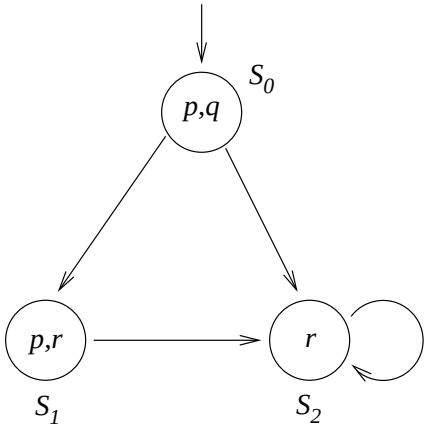

Figure 3.1: Example of an LTL Transition System

- $L(S_0) = \{p, q\}$. $L(S_1) = \{p, r\}$. $L(S_2) = \{r\}$.
- The transition relation contains $S_0 \to S_1$, $S_0 \to S_2$, $S_1 \to S_2$ and $S_2 \to S_2$.

Definition 3.1.3 (Path in an LTL)**:** A path in a model $\mathcal{M} = (S, \to, L)$ is a set of states s_1, s_2, s_3, ... in S such that for each $i \geq 1$, $s_i \to s_{i+1}$. We write such a path as $s_1 \to s_2 \to \ldots$

By convention, we will denote paths with π.

π^i denotes the path starting at time tick i. (s_i, s_{i+1}, s_{i+2}, ...).

Definition 3.1.4 (Satisfaction of an LTL Formula)**:** Let $\mathcal{M} = (S, \to, L)$ be a model and let $\pi = s_1 \to s_2 \to \ldots$ be a path in $\mathcal{M}$. Whether π satisfies an LTL formula is defined by the *satisfaction relation* $\vDash$, as follows.

1. $\pi \vDash \top$
2. $\pi \nvDash \bot$
3. $\pi \vDash p$ IFF $p \in L(s_1)$
4. $\pi \vDash \phi_1 \wedge \phi_2$ IFF $\pi \vDash \phi_1$ and $\pi \vDash \phi_2$.
5. $\pi \vDash \phi_1 \vee \phi_2$ IFF $\pi \vDash \phi_1$ or $\pi \vDash \phi_2$.
6. $\pi \vDash \phi_1 \to \phi_2$ IFF $\pi \vDash \phi_2$ whenever $\pi \vDash \phi_1$.
7. $\pi \vDash \mathrm{X}\,\phi$ IFF $\pi^2 \vDash \phi$. (π^2 is the path starting with the second element).
8. $\pi \vDash \mathrm{G}\,\phi$ IFF for all $i \geq 1$, $\pi^i \vDash \phi$. (Globally in the future)
9. $\pi \vDash \mathrm{F}\,\phi$ IFF there is some $i \geq 1$ such that $\pi^i \vDash \phi$. (At some point in the future. Note that "future" includes "now".)
10. $\pi \vDash \phi\,\mathrm{U}\,\psi$ IFF there is some $i \geq 1$ such that $\pi^i \vDash \psi$, and for all $1 \leq j < i$, we have $\pi^j \vDash \phi$.

    This is the *until* case. ϕ is true until ψ is true. We assume that ψ becomes true at some point.
11. $\pi \vDash \phi\,\mathrm{W}\,\psi$ IFF either (1) there is some $i \geq 1$ such that $\pi^i \vDash \psi$ and for all $1 \leq j < i$ we have $\pi^j \vDash \phi$ OR (2) for all $k \geq 1$, we have $\pi^k \vDash \phi$.

    The is 'weak until'. It's similar to U, but ψ might not become true. That's okay, as long as ϕ stays true.
12. $\pi \vDash \phi\,\mathrm{R}\,\psi$ IFF either (1) there is some $i \geq 1$ such that $\pi^i \vDash \phi$ and for all $1 \leq j \leq i$, we have $\pi^j \vDash \psi$, OR (2) for all $k \geq 1$ we have $\pi^k \vDash \psi$.

Here, we say that "ϕ releases ψ". ψ must be true up until (and including) the moment where ϕ becomes true.

R is the dual of U. $\phi \,\mathrm{R}\, \psi \equiv \neg(\neg\phi \,\mathrm{U}\, \neg\psi)$. □

NOTE: in LTL, "future" includes all time steps $\geq i$, including i itself.

Also note: $\phi \,\mathrm{W}\, \psi \equiv \phi \,\mathrm{U}\, \psi \vee \mathrm{G}\, \phi$.

If $s \in S$, $\mathcal{M}, S \vDash \phi$ IFF $\pi \vDash \phi$ for all paths π starting at state S.

3.1.6 Some Examples of LTL formulas

- $\vDash \mathrm{G}\, p \rightarrow p$. This formula is valid. If p holds globally, then p holds in the first state.
- $\vDash p \rightarrow \mathrm{F}\, p$. This is valid, because F includes the present, as well as future states.
- $\vDash p \rightarrow q \,\mathrm{U}\, p$. Valid, but vacuously. If p holds at s_1, then we can say q holds until s_1. (Vacuous, because there are no points before s_i).
- $\nvDash p \rightarrow p \,\mathrm{R}\, q$. Always false. If p is the first state in the path, then q cannot hold before p does.
- $\mathrm{G}\, \neg(\mathsf{started} \wedge \neg\mathsf{ready})$. Globally, we cannot be in a started state unless we are in a ready state.
- $\mathrm{G}(\mathsf{requested} \rightarrow \mathrm{F}\, \mathsf{enabled})$. If a service is requested, then it must become enabled. This includes becoming enabled at the time of the request.
- $\mathrm{G}\, \mathsf{enabled}$. Enabled forever, starting now
- $\mathrm{F}\, \mathrm{G}\, \mathsf{enabled}$. Enabled forever, starting at some point in the future.
- $\mathrm{G}\, \mathrm{F}\, \mathsf{enabled}$. Enabled infinitely often. (Allows transitions from enabled to not enabled, back to enabled again).

For next class, try to find an LTL expression that means "If running, then enabled at some point before". (i.e. - the service goes from not running to running, and becomes enabled at the point where it becomes running).

3.2 Lecture – 3/14/2007

3.2.1 Linear Temporal Logic

Last lecture, we asked how to say "if you're running at a state (but not at the previous state), then you're enabled". In LTL, we say this as

$$\mathrm{G}((\neg \mathsf{running} \wedge \mathrm{X}\,\mathsf{running}) \rightarrow \mathrm{X}\,\mathsf{enabled})$$

Review:

- $\mathrm{F}\,\mathrm{G}\,\phi$ means "ϕ will be true forever, starting at some point in the future".
- $\mathrm{G}\,\mathrm{F}\,\phi$ means "ϕ is true infinitely often".
- $\mathrm{F}\,\mathrm{G}\,\phi \rightarrow \mathrm{G}\,\mathrm{F}\,\phi$

What can't be expressed in linear temporal logic? LTL considers only one path through time; it does not have a concept for branching.

Suppose you wanted to say "From any reachable state, you can reach a restart state". This is like saying "From any reachable state, there exists a path to a restart state". Consider the transition system in Figure 3.2. This transition system contains two reachable states; the state where $\mathsf{restart} = \mathrm{T}$ is reachable

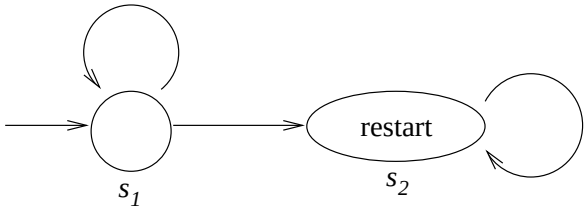

Figure 3.2: A simple Transition System

from either.

We could try to describe this as $\mathrm{G}(\mathrm{F}\,\mathsf{restart})$. But that doesn't quite work. We might stay at S_1 forever without reaching S_2.

Statements of the form "There exists a path" are usually not expressible in LTL. But sometimes one can get around this by using negation ($\exists x \phi \equiv \neg \forall x \neg \phi$).

When we say an LTL expression is true, it must be true regardless of any branch chosen.

3.2.2 Equivalences in LTL

G acts like $\forall$ (or $\wedge$).
F acts like $\exists$ (or $\vee$).

Some equivalences:

$$\neg\,\mathrm{G}\,\phi \equiv \mathrm{F}\,\neg\phi \qquad \text{G, F are duals} \qquad (3.2.1)$$

$$\neg\,\mathrm{F}\,\phi \equiv \mathrm{G}\,\neg\phi \qquad (3.2.2)$$

$$\neg\,\mathrm{X}\,\phi \equiv \mathrm{X}\,\neg\phi \qquad \text{X doesn't have a dual} \qquad (3.2.3)$$

$$\neg(\phi\,\mathrm{U}\,\psi) \equiv \neg\phi\,\mathrm{R}\,\neg\psi \qquad \text{U and R are duals} \qquad (3.2.4)$$

$$\neg(\phi\,\mathrm{R}\,\psi) \equiv \neg\phi\,\mathrm{U}\,\neg\psi \qquad (3.2.5)$$

$$\neg(\phi\,\mathrm{W}\,\psi) \equiv? \qquad \text{There's no dual for W.} \qquad (3.2.6)$$

Later, we'll see how W can be expressed using other connectives.

We give a proof of (3.2.4).

PROOF: $\neg(\phi \,\mathrm{U}\, \psi) \equiv \neg\phi \,\mathrm{R}\, \neg\psi$

1. Suppose $\pi \not\models \phi \,\mathrm{U}\, \psi$. For all $i \geq 1$, either $\pi^i \not\models \psi$ or there is a $j < i$ such that $\pi^j \not\models \phi$.

   In other words, $\pi^i \models \neg\psi$ or $\pi^j \models \neg\phi$. There are two cases to consider.

   (a) If for all $i \geq 1$, $\pi^i \models \neg\psi$, then $\pi \models \neg\phi \,\mathrm{R}\, \neg\psi$.

   (b) If there is an $i \geq 1$ such that $\pi^i \models \psi$, then take the least such i. There is a $j > i$ where $\pi^j \models \neg\phi$. For $i \leq k \leq j$, $\pi^k \models \neg\psi$, so $\pi \models \neg\phi \,\mathrm{R}\, \neg\psi$

2. Suppose $\pi \models \neg\phi \,\mathrm{R}\, \neg\psi$.

   Either (1) $\exists i \geq i$ such that $\pi^i \models \neg\phi$ and for all $1 \leq j \leq i$, $\pi^j \models \neg\psi$; or (2) for all $i \geq 1$, $\pi^i \models \neg\psi$.

   (a) In the first case, any k with $\pi^k \models \psi$ is $\geq i$. But at i, $\pi^i \models \neg\phi$, so $\pi \models \phi \,\mathrm{U}\, \psi$.

   (b) In the second case, ψ never becomes true, so $\pi \models \neg(\phi \,\mathrm{U}\, \psi)$.

□

More equivalences:

$$\mathrm{F}(\phi \vee \psi) \equiv \mathrm{F}\,\phi \vee \mathrm{F}\,\psi \tag{3.2.7}$$

$$\mathrm{G}(\phi \wedge \psi) \equiv \mathrm{G}\,\phi \wedge \mathrm{G}\,\psi \tag{3.2.8}$$

$$\mathrm{G}(\phi \vee \psi) \neq \mathrm{G}\,\phi \vee \mathrm{G}\,\psi \qquad \text{does NOT hold} \tag{3.2.9}$$

It's pretty easy to see why (3.2.9) doesn't hold. Consider Figure 3.3.

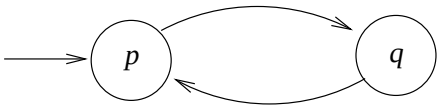

Figure 3.3: Illustration for (3.2.9)

$\mathrm{G}(p \vee q)$ holds, because one of p, q is true at each point in time. $\mathrm{G}\,p \vee \mathrm{G}\,q$ does not hold, because p and q keep alternating between true and false.

In LTL, we could say that

$$(\mathrm{G}\,\phi \vee \mathrm{G}\,\psi) \rightarrow \mathrm{G}(\phi \vee \psi)$$

Continuing with equivalences:

$$\mathrm{F}\,\phi \equiv \top \,\mathrm{U}\, \phi \tag{3.2.10}$$

$$\mathrm{G}\,\phi \equiv \bot \,\mathrm{R}\, \phi \tag{3.2.11}$$

$$\mathrm{F}\,\neg\phi \equiv \neg\top \,\mathrm{U}\, \neg\phi \tag{3.2.12}$$

$$\neg\,\mathrm{F}\,\neg\phi \equiv \neg(\top \,\mathrm{U}\, \neg\phi) \equiv \neg\top \,\mathrm{R}\, \neg\neg\phi \equiv \bot \,\mathrm{R}\, \phi \tag{3.2.13}$$

$$\phi \,\mathrm{U}\, \psi \equiv (\phi \,\mathrm{W}\, \psi) \wedge \mathrm{F}\,\psi \tag{3.2.14}$$

$$\phi \,\mathrm{W}\, \psi \equiv (\phi \,\mathrm{U}\, \psi) \vee \mathrm{G}\,\phi \tag{3.2.15}$$

$$\phi \,\mathrm{W}\, \psi \equiv \psi \,\mathrm{R}(\phi \vee \psi) \tag{3.2.16}$$

$$\phi \,\mathrm{R}\, \psi \equiv \psi \,\mathrm{W}(\phi \wedge \psi) \tag{3.2.17}$$

A proof of (3.2.16):

PROOF: $\phi \,\mathrm{W}\, \psi \equiv \psi \,\mathrm{R}(\phi \vee \psi)$

1. Suppose $\pi \vDash \phi \,\mathrm{W}\, \psi$.
   (a) If there exists an $i \geq 1$ such that $\pi^i \vDash \psi$ and for all $j \leq i$, $\pi^j \vDash \phi$.

   So, for all $1 \leq j \leq i$, $\pi^j \vDash \phi \vee \psi$. This is exactly $\psi \,\mathrm{R}(\phi \vee \psi)$.

   (b) Suppose, for all $k \geq 1$, $\pi^k \vDash \phi$. Then for all $k \geq 1$, $\pi^k \vDash \phi \vee \psi$. So, $\pi^k \vDash \psi \,\mathrm{R}(\phi \vee \psi)$.
2. Suppose $\pi \vDash \psi \,\mathrm{R}(\phi \vee \psi)$.
   (a) Suppose there is an $i \geq 1$ where $\pi^i \vDash \psi$, and $\pi^j \vDash \phi \vee \psi$ for $j \leq i$.

   Take the least such i; for all $j \leq i$, $\pi^j \vDash \phi$, so $\pi \vDash \phi \,\mathrm{W}\, \psi$.

   (b) Suppose for all $i \geq 1$, $\pi^i \vDash \phi \vee \psi$.

      i. Suppose there is an $i \geq 1$. Then $\pi^i \vDash \psi$. This is like case (2a).

      ii. Suppose for all $i \geq 1$, $\pi^i \nvDash \psi$. So, $\pi^i \vDash \phi$, and $\pi \vDash \phi \,\mathrm{W}\, \psi$

One last equivalence:

$$\phi \,\mathrm{U}\, \psi \equiv \neg(\neg\psi \,\mathrm{U}(\neg\phi \wedge \neg\psi)) \wedge \mathrm{F}\, \psi \tag{3.2.18}$$

We can show a derivation for the right side of (3.2.18).

$$\begin{aligned}
&\neg(\neg\psi \,\mathrm{U}(\neg\phi \wedge \neg\psi)) \wedge \mathrm{F}\, \psi \\
\equiv&(\neg\neg\psi \,\mathrm{R}\, \neg(\neg\phi \wedge \neg\psi)) \wedge \mathrm{F}\, \psi \\
\equiv&(\psi \,\mathrm{R}(\phi \vee \psi)) \wedge \mathrm{F}\, \psi \\
\equiv&\phi \,\mathrm{W}\, \psi \vee \mathrm{F}\, \psi \\
\equiv&\phi \,\mathrm{U}\, \psi
\end{aligned}$$

3.2.3 Adequate Connectives for LTL

In linear temporal logic, the following sets of connectives are adequate:

$\{\mathrm{U}, \mathrm{X}\}$
$\{\mathrm{R}, \mathrm{X}\}$
$\{\mathrm{W}, \mathrm{X}\}$

The connectives F, G can be derived from U, R.

U can derive R. R can derive W.
R can derive U. U can derive W.
W can derive R. R can derive U.

The set $\{\mathrm{F}, \mathrm{X}\}$ is not adequate. F and X are unary connectives; with only unary connectives, we cannot express the binary ones.

We cannot define G with $\{\mathrm{U}, \mathrm{F}\}$.
We cannot define F with $\{\mathrm{R}, \mathrm{G}\}$.
We cannot define F with $\{\mathrm{W}, \mathrm{G}\}$.

An adequate set of connectives for LTL must include X.

3.2.4 LTL Case Study: The Mutual Exclusion Problem

Huth and Ryan devotes a section to this. We'll start discussing it now, but read it over for next time.

The Problem: We have a set of concurrent processes that need to access a shared resource. We don't want multiple processes to access the same shared resource at the same time.

We will introduce critical sections into our code. The shared resource will be accessed from (and only from) these critical sections. It is not permissible for two processes to be in a critical section simultaneously.

Here are some of the qualities that we desire from our system.

Safety No more than one process in a critical section at any given time.

Liveness If a process wishes to enter a critical section, it will eventually be allowed to do so.

Non-Blocking A process can always request permission to enter a critical section.

No Strict Sequencing Processes need not enter their critical section in strict sequence. For example, if process 1 needs access to its critical section more often than process 2, we shouldn't impose an ordering of $\{1, 2, 1, 2, 1, 2 \ldots\}$ for entry to the critical section.

3.2.5 To-Do

Look into a tool called NuSMV. This is a tool that implements linear temporal logic. It's introduced in Chapter 3.

3.3 Lecture – 3/26/2007

3.3.1 A Brief Review of LTL Formula Structure

An EBNF for LTL formulas is as follows:

$$
\begin{aligned}
\phi ::= & \top \\
& \mid \bot \\
& \mid p \\
& \mid (\neg \phi) \\
& \mid (\phi \wedge \phi) \\
& \mid (\phi \vee \phi) \\
& \mid (\phi \rightarrow \phi) \\
& \mid (\mathrm{X}\, \phi) \\
& \mid (\mathrm{F}\, \phi) \\
& \mid (\mathrm{G}\, \phi) \\
& \mid (\phi\, \mathrm{W}\, \phi) \\
& \mid (\phi\, \mathrm{R}\, \phi) \\
& \mid (\phi\, \mathrm{U}\, \phi)
\end{aligned}
$$

3.3.2 The Mutual Exclusion Problem

We'll continue with the mutual exclusion problem that we began looking at during our last lecture. (See page 70).

The general idea is as follows: we wish to control access to a resource. We identify a critical section for each process. Processes may only access the shared resource from within their critical sections. Only one process may be in a critical section at any given time.

The qualities we'd like to have:

Safety No more than one process can be in a critical section at any instant in time.

Liveness If a process wishes to enter its critical section, it will eventually be able to do so.

Non-Blocking A process may ask to enter its critical section at any time.

No Strict Sequencing Processes do not have to enter their critical sections in a strict sequence. We cannot pre-define the order in which processes enter their critical sections.

No Strict Sequencing does *not* follow from Liveness and Non-Blocking.

Figure 3.7 on Page 181 of Huth & Ryan shows a first attempt at modeling Mutual exclusion. In this figure:

- n_i – process i is not in its critical section
- t_i – process i is trying to enter its critical section
- c_i – process i is in its critical section.

Which of our qualities can be modeled using LTL?

Safety Safety can be modeled in LTL.

$$\mathrm{G}(\neg(c_1 \wedge c_2)) \tag{3.3.1}$$

Liveness For two processes, we might try to model liveness with the following pair of formulas:

$$\mathrm{G}(t_1 \rightarrow \mathrm{F}\, c_1)$$
$$\mathrm{G}(t_2 \rightarrow \mathrm{F}\, c_2)$$

This doesn't hold for H&R's figure 3.7. A path that violates these formulas is

$$s_1 \rightarrow s_3 \rightarrow s_7 \rightarrow s_1 \rightarrow s_3 \rightarrow s_7 \rightarrow \ldots$$

This path shows that it's possible for t to ask to enter its critical section, but never enters the critical section.

Non-Blocking For non-blocking, what we really want to say is "for a state where n_1 is true, there is a successor where t_1 is true". Unfortunately, LTL does not give us a way to express the existence of a state. Therefore, Non-Blocking cannot be expressed in LTL.

Another attempt is $\mathrm{G}(n_1 \rightarrow \mathrm{F}\, t_1)$. This also doesn't work; it forces process 1 to request entry to its critical section. If process 1 never wishes to enter its critical section, we shouldn't force it to.

Another attempt that doesn't work is $\mathrm{G}(\mathrm{F}\, t_1 \vee \neg \mathrm{F}\, t_1)$. This doesn't buy us anything because it's a tautology.

No Strict Sequencing No Strict Sequencing cannot be expressed in LTL, but it's negation can.

The negation of No Strict Sequencing says the following: all paths that have a c_1 period that ends cannot have another c_1 period until there is a c_2 period. (Strict sequence: alternating c_1, c_2).

In LTL, we can write the negation as

$$\mathrm{G}(c_1 \rightarrow c_1 \,\mathrm{W}(\neg c_1 \wedge \neg c_1 \,\mathrm{W}\, c_2))$$

Figure 3.7 contains a counter example to this formula:

$$s_0 \rightarrow s_5 \rightarrow s_3 \rightarrow s_4 \rightarrow s_5 \ldots$$

Because the negation is false, we conclude that No Strict Sequencing holds.

3.3.3 Computation Tree Logic (CTL)

Computation Tree Logic (CTL) is a branching time logic. It allows quantifiers to be applied to paths. (So, you can say "there exists a path").

EBNF for CTL

An EBNF for CTL appears below:

$$
\begin{aligned}
\phi ::= & \bot \\
& \mid \top \\
& \mid p \\
& \mid (\neg \phi) \\
& \mid (\phi \wedge \phi) \\
& \mid (\phi \vee \phi) \\
& \mid (\phi \rightarrow \phi) \\
& \mid (\mathrm{AX}\, \phi) \\
& \mid (\mathrm{EX}\, \phi) \\
& \mid (\mathrm{AF}\, \phi) \\
& \mid (\mathrm{EF}\, \phi) \\
& \mid (\mathrm{AG}\, \phi) \\
& \mid (\mathrm{EG}\, \phi) \\
& \mid (\mathrm{A}[\phi\, \mathrm{U}\, \phi]) \\
& \mid (\mathrm{E}[\phi\, \mathrm{U}\, \phi])
\end{aligned}
$$

Binding order for CTL Operators and Quantifiers

The precedence order for CTL is as follows:

$$
\begin{aligned}
& \neg, \mathrm{AG}, \mathrm{EG}, \mathrm{AF}, \mathrm{EF}, \mathrm{AX}, \mathrm{EX} \\
& \wedge, \vee \\
& \rightarrow, \mathrm{A[U]}, \mathrm{E[U]}
\end{aligned}
$$

Before we get further into definitions, let's look at a some examples to get an intuitive feel for how CTL works.

Example 3.3.1: $\mathrm{EF}\, q$. There is reachable state satisfying q.

Example 3.3.2: $\mathrm{AG}(p \rightarrow \mathrm{E}[p\, \mathrm{U}\, q])$. From all reachable states satisfying p, it is possible to maintain p until we reach a state satisfying q.

Example 3.3.3: $\mathrm{AG}(p \rightarrow \mathrm{EG}\, q)$. Whenever a state satisfying p is reached, there is a path where q will be true forevermore.

Example 3.3.4: $\mathrm{EF}\, \mathrm{AG}\, p$. There is a reachable state from which all reachable states satisfy p.

The quantifiers U, F, G, and X retain their meaning from LTL.

The quantifiers A and E mean "for all paths", and "for some path", respectively.

3.3.4 Semantics of CTL

Definition 3.3.5 (Transition System)**:** A *transition system* $\mathcal{M} = (S, \rightarrow, L)$ is a set of states S, endowed with a transition relation $\rightarrow$, and a labeling function $L \colon S \rightarrow \mathcal{P}(atoms)$

This is the same definition we saw with LTL. There are nodes, edges, and truth values (labels) assigned to nodes.

CTL formulas are also interpreted over transition systems.

Let $\mathcal{M} = (S, \rightarrow, L)$ be a model, let s be a state $s \in S$, and let ϕ be a CTL formula. Whether $\mathcal{M}, s \vDash \phi$ can be understood as follows:

1. If ϕ is atomic, satisfaction is determined by L.
2. If the top-level connective of ϕ is a boolean connective ($\wedge$, $\vee$, $\rightarrow$, etc), then the answer comes from (1) the usual truth-table style of evaluation applied to the operator and (2) recursively evaluating each of the operands.
3. If the top connective is an operator beginning with 'A', then satisfaction holds if all paths satisfy the 'LTL Formula' resulting from removing the leading A symbol.
4. If the top-level connective begins with 'E', then satisfaction holds if some path from s satisfies the 'LTL Formula' that results from removing the leading E.

These rules are a little loose – removing a leading A or E does not necessarily produce a formula that is strictly LTL.

CTL Semantics, more formally

1. $\mathcal{M}, s \vDash \top$ and $\mathcal{M}, s \nvDash \bot$
2. $\mathcal{M}, s \vDash p$ IFF $p \in L(s)$
3. $\mathcal{M}, s \vDash \neg\phi$ IFF $\mathcal{M}, s \nvDash \phi$
4. $\mathcal{M}, s \vDash \phi_1 \wedge \phi_2$ IFF $\mathcal{M}, s \vDash \phi_1$ and $\mathcal{M}, s \vDash \phi_2$.
5. $\mathcal{M}, s \vDash \phi_1 \vee \phi_2$ IFF $\mathcal{M}, s \vDash \phi_1$ or $\mathcal{M}, s \vDash \phi_2$.
6. $\mathcal{M}, s \vDash \phi_1 \rightarrow \phi_2$ IFF $\mathcal{M}, s \nvDash \phi_1$ or $\mathcal{M}, s \vDash \phi_2$.
7. $\mathcal{M}, s \vDash \mathrm{AX}\,\phi$ IFF for all s_1 such that $s \rightarrow s_1$ we have $\mathcal{M}, s_1 \vDash \phi$. AX says "In every next state"
8. $\mathcal{M}, s \vDash \mathrm{EX}\,\phi$ IFF for some s_1 such that $s \rightarrow s_1$ we have $\mathcal{M}, s_1 \vDash \phi$. EX says "In some next state". EX and AX are duals.
9. $\mathcal{M}, s \vDash \mathrm{AG}\,\phi$ holds IFF for all paths $s_1 \rightarrow s_2 \rightarrow s_3 \ldots$ where s_1 equals s and s_i along the path, we have $\mathcal{M}s_i \vDash \phi$.

   For all paths beginning with s, ϕ holds globally. This includes s itself.
10. $\mathcal{M}, s \vDash \mathrm{EG}\,\phi$ holds IFF there is a path paths $s_1 \rightarrow s_2 \rightarrow s_3 \ldots$ where s_1 equals s and s_i along the path, we have $\mathcal{M}s_i \vDash \phi$.

    There exists a path beginning with s such that ϕ holds globally along that path.
11. $\mathcal{M}, s \vDash \mathrm{AF}\,\phi$ holds IFF for all paths $s_1 \rightarrow \ldots$, there is some s_i such that $\mathcal{M}, s_i \vDash \phi$.

    For all computation paths beginning with s, there will be some future state where ϕ holds.
12. $\mathcal{M}, s \vDash \mathrm{EF}\,\phi$ holds IFF for some path $\mathcal{M}, s_i \vDash \phi$.

    There exists a computation path beginning with s, such that ϕ holds in some future state.
13. $\mathcal{M}, s \vDash \mathrm{A}[\phi_1\,\mathrm{U}\,\phi_2]$ holds IFF for all paths, that path satisfies $[\phi_1\,\mathrm{U}\,\phi_2]$.

    There is some s_i along the path such that $\mathcal{M}, s_i \vDash \phi_2$ holds, and for all $1 \leq j < i$, we have $\mathcal{M}, s_j \vDash \phi_1$.

    All computation paths beginning with s satisfy ϕ_1 until ϕ_2 becomes true.

14. $\mathcal{M}, s \vDash \mathrm{E}[\phi_1 \,\mathrm{U}\, \phi_2]$ holds IFF there is some path that satisfies $[\phi_1 \,\mathrm{U}\, \phi_2]$.

    There exists a computation path beginning with s such that ϕ_1 holds until ϕ_2 holds.

More Examples

Example 3.3.6: EF(started $\wedge$ $\neg$ready). You can reach a state where started is true and ready is false.

Example 3.3.7: AG(requested $\rightarrow$ AF acknowledged).

For all states (globally), if requested is true, then for every path acknowledged will become true at some point. This is a good property for a service protocol.

Example 3.3.8: AG AF enabled $\rightarrow$ AG AF running. On every path, if enabled is true infinitely often, then on every path, running will be true infinitely often. The left side of the implication is strong, which makes it a weak assertion (easy for the LHS to be false).

Contrast this with the LTL formula G F enabled $\rightarrow$ G F running. This means, on every path, if enabled is true infinitely often, then running is true infinitely often.

The LTL version is more particular; it is also inexpressible in CTL.

Example 3.3.9: AF AG ϕ. On every path, you'll reach a point where ϕ is globally true.

Example 3.3.10: AG EF ϕ. From every path, you can reach a state where ϕ will become true in the future.

Example 3.3.11 (Non-Blocking in CTL)**:** In CTL we can express our non-blocking property as AG($n_1 \rightarrow$ EX t_1).

3.3.5 LTL vs CTL

Implicitly, there is an A quantifier in every LTL formula. LTL requires the formula to be true on every path.

Not all LTL formulas can be expressed in CTL.

Not all CTL formulas can be expressed in LTL.

Sometimes one can turn an LTL formula into a CTL formula by adding the A quantifier.

3.4 Lecture – 3/28/2007

3.4.1 Mutual Exclusion in CTL and LTL

We'll finish our analysis of mutual exclusion with a comparison of how CTL and LTL handle the problem.

Safety
In LTL: $\mathrm{G}\,\neg(c_1 \wedge c_2)$
In CTL: $\mathrm{AG}\,\neg(c_1 \wedge c_2)$

Liveness
In LTL: $\mathrm{G}(n_1 \rightarrow \mathrm{F}\,c_1)$
In CTL: $\mathrm{AG}(n_1 \rightarrow \mathrm{AF}\,c_1)$

Non-Blocking
In LTL: Not expressible
In CTL: $\mathrm{AG}(n_1 \rightarrow \mathrm{EX}\,t_1)$

No Strict Sequencing
In LTL: We could express the negation, $\mathrm{G}(c_1 \rightarrow c_1\,\mathrm{W}(\neg c_1 \wedge \neg c_1\,\mathrm{W}\,c_2))$
In CTL: $\mathrm{EF}(c_1 \wedge \mathrm{E}[c_1\,\mathrm{U}(\neg c_1 \wedge \mathrm{E}[\neg c_2\,\mathrm{U}\,c_1])])$

Figure 3.4 gives a visual representation of the CTL formula for no strict sequencing.

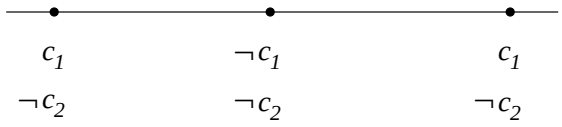

Figure 3.4: Visual representation of No Strict Sequencing under CTL

3.4.2 CTL Equivalences

Some of CTL's equivalences are like standard quantifier negation.

$$\neg\,\mathrm{AF}\,\phi \equiv \mathrm{EG}\,\neg\phi$$
$$\neg\,\mathrm{EF}\,\phi \equiv \mathrm{AG}\,\neg\phi$$
$$\neg\,\mathrm{AX}\,\phi \equiv \mathrm{EX}\,\neg\phi$$
$$\neg\,\mathrm{EX}\,\phi \equiv \mathrm{AX}\,\neg\phi$$
$$\mathrm{AF}\,\phi \equiv \mathrm{A}[\top\,\mathrm{U}\,\phi]$$
$$\mathrm{EF}\,\phi \equiv \mathrm{E}[\top\,\mathrm{U}\,\phi]$$

$$\mathrm{AX}\,\phi \equiv \neg\,\mathrm{EX}\,\neg\phi$$
$$\mathrm{AG}\,\phi \equiv \neg\,\mathrm{EF}\,\neg\phi$$
$$\mathrm{EG}\,\phi \equiv \neg\,\mathrm{AF}\,\neg\phi$$

3.4.3 Adequate Connectives in CTL

Theorem 3.4.1: A set of CTL connectives is adequate IFF it contains

- one of $\{\mathrm{AX}, \mathrm{EX}\}$
- one of $\{\mathrm{EG}, \mathrm{AF}, \mathrm{A\,U}\}$
- and $\mathrm{E\,U}$

Example 3.4.2: The set of connectives $\{\mathrm{EX}, \mathrm{A\,U}, \mathrm{E\,U}\}$ is adequate for CTL. From the equivalences given earlier, it's not difficult to see how the 8 CTL connectives can be derived.

Example 3.4.3: The set of connectives $\{\mathrm{EX}, \mathrm{EG}, \mathrm{E\,U}\}$ is adequate. This set is like the one appearing in Example 3.4.2. The only differences is EG instead of $\mathrm{A\,U}$ – we just need to show that $\mathrm{A\,U}$ can be expressed using the connectives we have.

We have the equivalence

$$\mathrm{A}[\phi\,\mathrm{U}\,\psi] \equiv \mathrm{A}[\neg(\neg\psi\,\mathrm{U}(\neg\phi \wedge \neg\psi)) \wedge \mathrm{F}\,\psi]$$

This isn't strictly a CTL formula, but it is a CTL* formula. We can manipulate this as follows:

$$\begin{aligned}
&\equiv \neg\,\mathrm{E}\,\neg[\neg(\neg\psi\,\mathrm{U}(\neg\phi \wedge \neg\psi)) \wedge \mathrm{F}\,\psi] \\
&\equiv \neg\,\mathrm{E}[\neg\psi\,\mathrm{U}(\neg\phi \wedge \neg\psi) \vee \neg\,\mathrm{F}\,\psi] \\
&\equiv \neg\,\mathrm{E}(\neg\psi\,\mathrm{U}(\neg\phi \wedge \neg\psi)) \vee \mathrm{E}[\neg\,\mathrm{F}\,\psi] \\
&\equiv \neg\,\mathrm{E}(\neg\psi\,\mathrm{U}(\neg\phi \wedge \neg\psi)) \vee \mathrm{E\,G}\,\neg\psi
\end{aligned}$$

From the last line, we have expressed $\mathrm{A\,U}$ with EG.

Example 3.4.4: The connectives $\{\mathrm{EX}, \mathrm{AF\,E\,U}\}$ are another adequate set.

3.4.4 What's Not A Connective in CTL

CTL doesn't adopt all of LTL's connectives. For example there's no AR, ER, AW, or EW.

This doesn't limit the expressiveness of CTL. In LTL, we saw that R and W could be derived. The same thing could be done in CTL.

$$\begin{aligned}
\mathrm{A}[\phi\,\mathrm{R}\,\psi] &\equiv \mathrm{A}[\neg(\neg\phi\,\mathrm{U}\,\neg\psi)] \\
&\equiv \neg\,\mathrm{E}[\neg\phi\,\mathrm{U}\,\neg\psi] \\
\mathrm{E}[\phi\,\mathrm{R}\,\psi] &\equiv \neg\,\mathrm{A}[\neg\phi\,\mathrm{U}\,\neg\psi] \\
\mathrm{A}[\phi\,\mathrm{W}\,\psi] &\equiv \mathrm{A}[\psi\,\mathrm{R}(\phi \vee \psi)] \\
\mathrm{E}[\phi\,\mathrm{W}\,\psi] &\equiv \mathrm{E}[\psi\,\mathrm{R}(\phi \vee \psi)]
\end{aligned}$$

3.4.5 More CTL Equivalences

The equivalences that follow are recursive. Instead of simply making an assertion from the current state, they say something about the current state and the state that follows. Studying them gives a good idea

about how CTL works.

$$\begin{aligned}
\mathrm{AG}\,\phi &\equiv \phi \wedge \mathrm{AX}\,\mathrm{AG}\,\phi \\
\mathrm{EG}\,\phi &\equiv \phi \wedge \mathrm{EX}\,\mathrm{EG}\,\phi \\
\mathrm{AF}\,\phi &\equiv \phi \wedge \mathrm{AX}\,\mathrm{AF}\,\phi \\
\mathrm{EF}\,\phi &\equiv \phi \wedge \mathrm{EX}\,\mathrm{EF}\,\phi \\
\mathrm{A}[\phi\,\mathrm{U}\,\psi] &\equiv \psi \vee (\phi \wedge \mathrm{AX}\,\mathrm{A}[\phi\,\mathrm{U}\,\psi]) \\
\mathrm{E}[\phi\,\mathrm{U}\,\psi] &\equiv \psi \vee (\phi \wedge \mathrm{EX}\,\mathrm{E}[\phi\,\mathrm{U}\,\psi])
\end{aligned}$$

3.4.6 LTL + CTL = CTL*

CTL* is a unification of LTL and CTL. Where CTL requires A and E to be paired with another connective, CTL* allows them to be used by themselves.

Example 3.4.5: The following is valid CTL* formula:

$$\mathrm{A}[(p\,\mathrm{U}\,r) \vee (q\,\mathrm{U}\,r)] \tag{3.4.1}$$

This means: on all paths, either p is true until r; or q is true until r.

Equation (3.4.1) is *not* the same thing as saying

$$\mathrm{A}[(p \vee q)\,\mathrm{U}\,r] \tag{3.4.2}$$

Equation (3.4.2) says that p or q is true until r is true. This allows oscillation of p and q. (3.4.1) does not permit this oscillation.

Example 3.4.6: The equation

$$\mathrm{A}[\mathrm{X}\,p \vee \mathrm{X}\,\mathrm{X}\,p]$$

is valid in CTL*, but not CTL. This equation means: on all paths, p is true in the next state, or p is true in the state after the next state.

Example 3.4.7: Consider the equations

$$\mathrm{E}[\mathrm{G}\,\mathrm{F}\,p] \tag{3.4.3}$$
$$\mathrm{EG}\,\mathrm{EF}\,p \tag{3.4.4}$$

(3.4.3) says that there exists a path where p is true infinitely often. (3.4.4) says that there exists a path where p is true sometime in the future. (p might be true once; not infinitely often). Figure 3.4.7

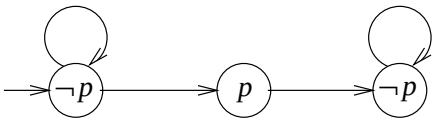

Figure 3.5: Illustration of the difference between (3.4.3) and (3.4.4)

illustrates the difference. (3.4.4) is true in this system. (3.4.3) is not.

3.4.7 Syntax of CTL*

The syntax of CTL* involves two kinds of formulas: state formulas and path formulas. We denote state formulas with ϕ and path formulas with α.

State formulas:

$$\phi ::= \top \mid p \mid (\neg\phi) \mid (\phi \wedge \phi) \mid \mathrm{A}[\alpha] \mid \mathrm{E}[\alpha]$$

Path formulas:

$$\alpha ::= \phi \mid (\neg\alpha) \mid (\alpha \wedge \alpha) \mid (\alpha \,\mathrm{U}\, \alpha) \mid (\mathrm{G}\, \alpha) \mid (\mathrm{F}\, \alpha) \mid (\mathrm{X}\, \alpha)$$

These definitions are mutually recursive, with $\top$ and p as the base cases.

3.4.8 Semantics of CTL*

CTL* has two types of formulas; each has its own semantics.

$\mathcal{M}, s \vDash \phi$	state formula semantics
$\mathcal{M}, \pi \vDash \alpha$	path formula semantics

$\mathcal{M}, s \vDash \mathrm{A}[\alpha]$ is true IFF for all paths p, $\mathcal{M}, \pi \vDash \alpha$ starting from state s. (*check this*). (**Note**: see also section 4.3.1, page 110).

$\mathcal{M}, \pi \vDash \alpha$ is true IFF where s is the first state of the path π, we have $\mathcal{M}, s \vDash \phi$.

3.4.9 LTL and CTL are Subsets of CTL*

$$\text{LTL} \subseteq \text{CTL*}$$
$$\text{CTL} \subseteq \text{CTL*}$$

To restrict CTL* to CTL, we need only restrict the form of path formulas to

$$\alpha ::= (\phi \,\mathrm{U}\, \psi) \mid (\mathrm{G}\, \phi) \mid (\mathrm{F}\, \phi) \mid (\mathrm{X}\, \phi)$$

Figure 3.6 depicts the relationship between CTL, LTL, and CTL*.

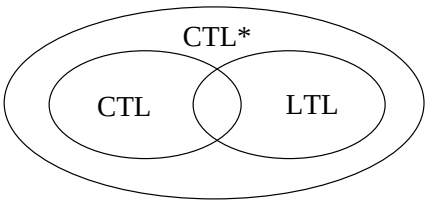

Figure 3.6: Relationship of CTL, LTL, and CTL*

Example 3.4.8 (In LTL and CTL)**:** The atomic formula p is valid in both LTL and CTL

Example 3.4.9 (CTL, Not LTL)**:** The formula $\mathrm{AG\,EF}\,p$ can be expressed in CTL, but cannot be expressed in LTL. (We'll see why shortly, when we examine the sub-model property).

Example 3.4.10 (LTL, Not CTL)**:** The formula $\mathrm{A}[\mathrm{G\,F}\,p \rightarrow \mathrm{F}\,q]$ can be expressed in LTL, but not CTL.

Example 3.4.11 (CTL* only)**:** The formula $\mathrm{E}[\mathrm{G\,F}\,p]$ cannot be expressed in LTL, and it cannot be expressed in CTL. It's only expressible in CTL*.

3.4.10 The Sub-model Property of LTL

Definition 3.4.12 (submodel)**:** Let $\mathcal{M} = (S, \rightarrow, L)$ be a model. $\mathcal{M}' = (S', \rightarrow', L')$ is a *sub-model* of $\mathcal{M}$ if

$$S' \subseteq S$$
$$\rightarrow' \subseteq \rightarrow$$
$$L' = L \restriction S'$$

$L \restriction S'$ means "L restricted to S'".

Theorem 3.4.13 (Submodel Property)**:** If $\mathrm{A}[\alpha]$ is an LTL formula, and $\mathcal{M}'$ is a sub-model of $\mathcal{M}$, and $s \in S'$, then

$$\mathcal{M}, s \vDash \mathrm{A}[\alpha] \rightarrow \mathcal{M}', s \vDash \mathrm{A}[\alpha]$$

If an LTL formula ϕ is valid for a model $\mathcal{M}$, then ϕ is also valid for any sub-model $\mathcal{M}'$ of $\mathcal{M}$.

Theorem 3.4.13 gives us a way to prove that a given ϕ is not LTL. If you can find a model $\mathcal{M}$ where ϕ holds, and a sub-model $\mathcal{M}'$ where ϕ does not hold, then ϕ is not an LTL formula.

Example 3.4.14: Consider the $\mathcal{M}$ and $\mathcal{M}'$ shown in Figure 3.7.

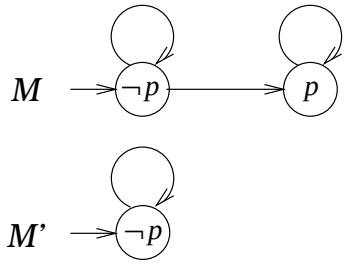

Figure 3.7: Model $\mathcal{M}$ and sub-model $\mathcal{M}'$

The formula $\phi = \mathrm{AG\,EF}\,p$ holds for $\mathcal{M}$. At any point, there is a future state where p is true.

$\mathrm{AG\,EF}\,p$ does not hold for $\mathcal{M}'$ (p is never true). Because ϕ does not hold for $\mathcal{M}'$, ϕ is not an LTL formula.

3.5 Lecture – 4/2/2007

3.5.1 CTL vs CTL*

There are two differences between CTL and CTL*:

1. CTL does not allow the boolean combination of path formulas. For example, you can't say $\mathrm{E}[(p \,\mathrm{U}\, q) \wedge (p \,\mathrm{R}\, r)]$.
2. CTL does not allow one path operator to be applied to another path operator. For example, you can't say $\mathrm{G}\,\mathrm{F}\, p$.

Difference (1) can be gotten around. Boolean combinations of path formulas can usually be expressed using CTL equivalences.

Example 3.5.1: Using equivalences to achieve the effect of boolean combinations of path formulas in CTL.

$$\mathrm{E}[\mathrm{F}\, p \vee \mathrm{F}\, q] \equiv \mathrm{EF}\, p \vee \mathrm{EF}\, q$$
$$\mathrm{E}[\mathrm{F}\, p \wedge \mathrm{F}\, q] \equiv \mathrm{E}[\mathrm{F}(p \wedge \mathrm{EF}\, q)] \vee \mathrm{E}[\mathrm{F}(q \wedge \mathrm{EF}\, p)]$$
$$\mathrm{A}[\mathrm{F}\, p \vee \mathrm{F}\, q] \equiv \mathrm{AF}(p \vee q)$$
$$\mathrm{A}[\mathrm{G}\, p \vee \mathrm{G}\, q] \equiv ?$$

Difference (2) we're stuck with. There's no way to work around it in CTL.

3.5.2 Past Connectives

The path operators we've seen refer to future states. We could consider augmenting them with operators that refer to path states. For example

Y	Yesterday	Past version of X
S	Since	Past version of U
O	Once	Past version of X
H	Historically	Past version of G

Example 3.5.2 (Past Connectives)**:**

$$\mathrm{G}(p \rightarrow \mathrm{O}p)$$

Globally, if q is true then p is true now, or p was true at some point in the past. This is equivalent to the formulas

$$\neg p \,\mathrm{W}\, q$$
$$\neg(\neg q \,\mathrm{U}(p \wedge \neg q))$$

Past operators *do not* add any expressive power to LTL. LTL considers single paths. Past operators allow us to travel backward along that path, but only to points reachable by traveling forward from the start state.

Past operators *do* add expressive power to CTL. CTL gives us no way to say things about states that are not forward reachable.

3.5.3 Model Checking

The general question for temporal logic checking is as follows: does the initial state of a transition system satisfy the given LTL or CTL formula? Or, does $\mathcal{M}, s_0 \vDash \phi$ for the starting state s_0.

Usually it's easier to find a list of states that satisfy ϕ. If the starting state satisfies ϕ, then $\mathcal{M}, s \vDash \phi$ holds. In other words, find states

$$\{s \in S \mid \mathcal{M}, s \vDash \phi\}$$

3.5.4 CTL Labeling Algorithms

We first rewrite ϕ as follows:

- Use only the propositional connectives $\bot$, $\wedge$, and $\neg$.
- Use only the CTL connectives AF, E U, EX. Limiting ourselves to a single adequate set of CTL connectives simplifies the algorithm.

Next, we label each model state with all sub-formulas that are satisfied at that state. This is an iterative process.

- $\bot$. No state is labeled with $\bot$
- p. Label s with p if $p \in L(s)$.
- $\psi_1 \wedge \psi_2$. If s is labeled with both ψ_1 and ψ_2, then label s with $\psi_1 \wedge \psi_2$.
- $\mathrm{AF}\,\psi$.
  - If any state s is labeled with ψ, then label s with $\mathrm{AF}\,\psi$.
  - Repeat until no change: Label any state s' with $\mathrm{AF}\,\psi$ if all successors of s' are labeled with $\mathrm{AF}\,\psi$.
- $\mathrm{E}[\psi_1\,\mathrm{U}\,\psi_2]$.
  - If any state s is labeled with ψ_2, label s with $\mathrm{E}[\psi_1\,\mathrm{U}\,\psi_2]$.
  - Repeat until no change: label s' with $\mathrm{E}[\psi_1\,\mathrm{U}\,\psi_2]$ if s' is labeled with ψ_1 and at least one successor of s' is labeled with $\mathrm{E}[\psi_1\,\mathrm{U}\,\psi_2]$.
- $\mathrm{EX}\,\psi$. Label s with $\mathrm{EX}\,\psi$ if at least one successor of s is labeled with ψ.

3.5.5 CTL Satisfiability Algorithm

Below, ϕ is assumed to be a well-formed CTL formula.

procedure SAT(ϕ)
  if ϕ is $\top$ **then**
    return S ▷ S = all states
  else if ϕ is $\bot$ **then**
    return $\emptyset$
  else if ϕ is atomic **then**
    return $\{s \in S \mid \phi \in L(s)\}$
  else if ϕ is $\neg\phi_1$ **then**
    return $S - \mathtt{SAT}(\phi_1)$
  else if ϕ is $\phi_1 \wedge \phi_2$ **then**
    return $\mathtt{SAT}(\phi_1) \cap \mathtt{SAT}(\phi_2)$
  else if ϕ is $\phi_1 \vee \phi_2$ **then**
    return $\mathtt{SAT}(\phi_1) \cup \mathtt{SAT}(\phi_2)$

else if ϕ is $\phi_1 \rightarrow \phi_2$ **then**
return $\texttt{SAT}(\neg\phi_1 \vee \phi_2)$
else if ϕ is $\mathrm{AX}\,\phi_1$ **then**
return $\texttt{SAT}(\neg\,\mathrm{EX}\,\neg\phi_1)$
else if ϕ is $\mathrm{EX}\,\phi_1$ **then**
return $\texttt{SAT}_{\texttt{EX}}(\phi_1)$
else if ϕ is $\mathrm{A}[\phi_1\,\mathrm{U}\,\phi_2]$ **then**
return $\texttt{SAT}(\neg(\mathrm{E}[\neg\phi_2\,\mathrm{U}(\neg\phi_1 \wedge \neg\phi_2)] \vee \mathrm{EG}\,\neg\phi_2))$
else if ϕ is $\mathrm{E}[\phi_1\,\mathrm{U}\,\phi_2]$ **then**
return $\texttt{SAT}_{\texttt{EU}}(\phi_1, \phi_2)$
else if ϕ is $\mathrm{EF}\,\phi_1$ **then**
return $\texttt{SAT}(\mathrm{E}[\top\,\mathrm{U}\,\phi_1])$
else if ϕ is $\mathrm{EG}\,\phi_1$ **then**
return $\texttt{SAT}(\neg\,\mathrm{AF}\,\neg\phi_1)$
else if ϕ is $\mathrm{AF}\,\phi_1$ **then**
return $\texttt{SAT}_{\texttt{AF}}(\phi_1)$
else if ϕ is $\mathrm{AG}\,\phi_1$ **then**
return $\texttt{SAT}(\neg\,\mathrm{EF}\,\neg\phi_1)$
end if
end procedure

The function $\texttt{SAT}_{\texttt{EX}}$, $\texttt{SAT}_{\texttt{EU}}$, $\texttt{SAT}_{\texttt{AF}}$ are helpers. These are detailed later.

The helpers rely on these two auxiliary functions:

$$\mathsf{pre}_\exists(Y) = \{s \in S \mid \text{exists } s', (s \rightarrow s' \text{ and } s' \in Y)\} \tag{3.5.1}$$
$$\mathsf{pre}_\forall(Y) = \{s \in S \mid \text{for all } s', (s \rightarrow s' \text{ implies } s' \in Y)\} \tag{3.5.2}$$

Both (3.5.1) and (3.5.2) compute pre-images of states.

$\mathsf{pre}_\exists$ returns a set of states that can make a transition into Y.

$\mathsf{pre}_\forall$ returns a set of states that only make transitions into Y.

In an ideal world, we'd want SAT to have a running time that is proportional to (a) the size of the formula and (b) the size of the model. In other words, $\Theta(|\phi| \cdot (|V| + |E|))$

Procedure $\texttt{SAT}_{\texttt{AF}}$

The function $\texttt{SAT}_{\texttt{AF}}$ works as described in Section 3.5.4 on Page 82.

procedure $\texttt{SAT}_{\texttt{AF}}(\phi)$
$X = S$ ▷ S is the set of all model states
$Y = \texttt{SAT}(\phi)$
repeat
$X = Y$
$Y = Y \cup \mathsf{pre}_\forall(Y)$
until $X = Y$ ▷ Until no change
return Y
end procedure

The disadvantage with $\texttt{SAT}_{\texttt{AF}}$ is its running time: proportional to $\Theta(V \cdot (V + E))$. Later, we'll look at a better algorithm.

Procedure $\texttt{SAT}_{\texttt{EU}}$

We'll look at two forms of $\texttt{SAT}_{\texttt{EU}}$. The first form is better for understanding how the algorithm works. The second form is better for understanding the running time.

procedure $\texttt{SAT}_{\texttt{EU}}(\phi, \psi)$ ▷ First Form
  $W = \texttt{SAT}(\phi)$
  $X = S$
  $Y = \texttt{SAT}(\psi)$ ▷ Start with states satisfying ψ
  repeat
    $X = Y$
    $Y = Y \cup (W \cap \mathsf{pre}_{\exists}(Y))$ ▷ Restrict to states satisfying ϕ
  until $X = Y$
  return Y
end procedure

Now, the second form of $\texttt{SAT}_{\texttt{EU}}$:

procedure $\texttt{SAT}_{\texttt{EU}}(\phi, \psi)$ ▷ Second Form
  $W = \texttt{SAT}(\phi)$
  $Y = \texttt{SAT}(\psi)$
  $T = Y$
  while $T \neq \emptyset$ **do**
    chose $s \in T$
    $T = T - \{s\}$
    for all t such that $t \rightarrow s$ **do**
      if $t \notin Y$ and $t \in W$ **then**
        $Y = Y \cup \{t\}$
        $T = T \cup \{t\}$
      end if
    end for
  end while
  return Y
end procedure

Some Points to note on the second form of $\texttt{SAT}_{\texttt{EU}}$

1. If an element is in T, then that element is already in Y.
2. If an element is added to T, then that element is also added to Y.
3. No element is ever removed from Y.
4. If t goes into Y, then $\mathcal{M}, t \vDash \mathrm{E}[\phi \,\mathrm{U}\, \psi]$
5. If $\mathcal{M}, t \vDash \mathrm{E}[\phi \,\mathrm{U}\, \psi]$, then t goes into Y.

The running time of of $\texttt{SAT}_{\texttt{EU}}$ is proportional to $\Theta(V + E)$. Each transition system is edge is examined at most once.

Procedure $\texttt{SAT}_{\texttt{EX}}$

The procedure to determine satisfaction of $\mathrm{EX}\,\phi$ is pretty straightforward:

procedure $\texttt{SAT}_{\texttt{EX}}(\phi)$
  $X = \texttt{SAT}(\phi)$
  $Y = \mathsf{pre}_{\exists}(X)$
  return Y
end procedure

3.5.6 A more efficient way to handle $\mathrm{EG}\,\psi$

There is a more efficient algorithm to handle $\mathrm{EG}\,\psi$.

- Label all states with $\mathrm{EG}\,\psi$.
- If any state is not labeled with ψ, then delete $\mathrm{EG}\,\psi$.
- Repeat until no change: delete $\mathrm{EG}\,\psi$ from s if no successor of s is labeled with $\mathrm{EG}\,\psi$.

This algorithm is given a more thorough treatment on page 86.

3.6 Lecture – 4/4/2007

3.6.1 CTL Model Checking

We've looked at three helper functions for checking CTL models. These functions, and their running times are

- $\texttt{SAT}_{\texttt{AF}}$– $\Theta(|\phi| \cdot |V| \cdot (|V| + |E|))$
- $\texttt{SAT}_{\texttt{EU}}$– $\Theta(|\phi| \cdot (|V| + |E|))$
- $\texttt{SAT}_{\texttt{EX}}$– $\Theta(|\phi| \cdot (|V| + |E|))$

These Θ values are slightly misleading. We're talking about transition systems; every node must have at least one outgoing edge. Therefore, the number of edges will be $\geq$ the number of nodes. $|E|$ dominates $|V|$ in the complexity values above.

3.6.2 $\texttt{SAT}_{\texttt{EG}}$: A Better Version of $\texttt{SAT}_{\texttt{AF}}$

The equivalence $\text{EG}\,\phi \equiv \neg\,\text{AF}\,\neg\phi$ allows us to write a more efficient $\texttt{SAT}_{\texttt{AF}}$, one whose running time is a linear function of the model size.

We'll call this function $\texttt{SAT}_{\texttt{EG}}$.

Definition 3.6.1 (Strongly Connected Component)**:** A *strongly connected component* (SCC) in a directed graph is a maximal subset such that each vertex is connected to each other component vertex by a path.

Example 3.6.2: Figure 3.8 shows a directed graph with two strongly connected components. One component consists of vertices $\{a, b, c\}$. The second component consists of the vertex $\{d\}$.

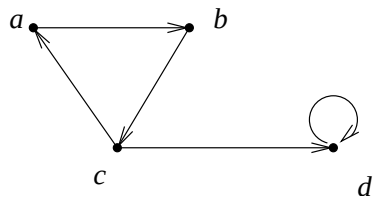

Figure 3.8: Two Strongly Connected Components

A single vertex is always strongly connected (it has a path of length zero to itself).

Definition 3.6.3 (Non-Trivial SCC)**:** A Strongly connected component is *non-trivial* if (a) it contains more than one node or (b) it contains one node with a loop to itself (like d in Figure 3.8).

Let $\mathcal{M}'$ be obtained by $\mathcal{M}$ by deleting states where ϕ is false. $\mathcal{M}'$ may not be a transition system, but it will still be a directed graph.

Claim 3.6.4: $\mathcal{M}, s \vDash \text{EG}\,\phi$ IFF

1. $s \in S'$. ϕ must be true in state s.
2. There is a path in $\mathcal{M}'$ from s to a non-trivial strongly connected component of $\mathcal{M}'$.

Proof: Case 1 (if):

- Let π be a path starting at s, with ϕ true everywhere on π.

- π is a path of $\mathcal{M}'$, because $\mathcal{M}'$ consists only of states where ϕ is true.
- Let us divide π into two paths: π_0 (a finite path) and π_1 (an infinite path). In π_1 every state will occur infinitely often.
- Every state in π_1 is connected to every other state in π_1.
- $\therefore$ states in π_1 are contained in a strongly connected component of $\mathcal{M}'$.

Case 2 (only if):

- Suppose $s \in S'$, and suppose that there is a path in $\mathcal{M}'$ from s to a strongly connected component of $\mathcal{M}'$
- This gives an infinite path from s with ϕ true everywhere along that path.

□

There is a linear-time algorithm to compute strongly connected components. The algorithm is due to Tarjen; the CLR algorithms book should have it.

procedure $\texttt{SAT}_{\texttt{EG}}(\phi)$
  $W = \texttt{SAT}(\phi)$
  $Y = \bigcup\{c \mid c \text{ is an SCC of } W\}$
  $T = Y$
  while $T \neq \emptyset$ **do**
    chose $s \in T$
    $T = T - \{s\}$
    for all t such that $t \rightarrow s$ **do**
      if $t \notin Y$ and $t \in W$ **then**
        $Y = Y \cup \{t\}$
        $T = T \cup \{t\}$
      end if
    end for
  end while
  return Y
end procedure

Because $\texttt{SAT}_{\texttt{EG}}$ is a linear-time algorithm, the entire `SAT` procedure has running time linear in the size of the model.

3.6.3 The State Explosion Problem

`SAT` is linear in the size of the model. Unfortunately, the size of the model can grow at a non-linear rate with respect to the formula. For example, adding one variable could double the number of model states. Although the algorithm is linear, the size of the problem is not.

To effectively deal with large models, we'd really like more than just a linear-time algorithm.

There are techniques that address the state explosion problem. One of these is called an *OBDD*, or *ordered binary decision tree*. An OBDD is a data structure. We'll look at them later in the semester.

3.6.4 CTL Model Checking With Fairness

Consider a fragment from our mutual exclusion problem:

Figure 3.9: A Fragment of the Mutual Exclusion Problem

Figure 3.9 shows two states: (a) process 1 in a critical section; process 2 waiting to enter a critical section and (b) process one outside the critical section, process two in the critical section.

Our model assumes that all state transitions happen in a single clock tick. But what if process 1 needed to stay in its critical section for longer than one tick? We'd have to add another edge, like the one in Figure 3.10

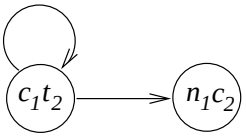

Figure 3.10: Allowing process 1 to stay in its critical section

The new edge in Figure 3.10 creates a problem: process one may stay in its critical section indefinitely, violating liveness. Process two has asked to enter its critical section – when will it be able to do so?

We can augment CTL with *fairness constraints*. The model checker must consider only those paths which satisfy fairness constraints infinitely often.

Example 3.6.5: Fairness constraints for mutual exclusion:

$\mathrm{G\,F}\,\neg c_1$	Process 1 must leave its critical section infinitely often
$\mathrm{G\,F}\,\neg c_2$	Likewise for process 2

Definition 3.6.6 (Fairness)**:** Let $C = \{\psi_1, \ldots, \psi_n\}$ be a set of fairness constraints. (They'll actually be CTL formulas). A path $\pi = s_1 \rightarrow s_2 \rightarrow \ldots$ is fair with respect to C if, for all i there are infinitely many j such that

$$\mathcal{M}, s_j \vDash \psi_i$$

Every fairness constraint must occur infinitely often.

We can augment the CTL quantifiers:

$\mathrm{A_C}$	Every path is fair
$\mathrm{E_C}$	There exists a fair path

$\mathrm{A_C}$ and $\mathrm{E_C}$ are like A and E, but they're restricted to fair paths.

Some equivalences for fair quantifiers:

$$\mathrm{E_C}[\phi\,\mathrm{U}\,\psi] \equiv \mathrm{E}[\phi\,\mathrm{U}(\psi \wedge \mathrm{E_C G}\,\top)]$$
$$\mathrm{E_C X}\,\phi \equiv \mathrm{EX}(\phi \wedge \mathrm{E_C G}\,\top)$$

Claim 3.6.7: A computation path is fair IFF any suffix of the path is fair.

Given these equivalences, we need only supply an algorithm for $\mathrm{E_C G}$.

Algorithm for $\mathrm{E_C G}$

The algorithm for $\mathrm{E_C G}$ is similar to $\mathtt{SAT_{EG}}$.

- Restrict the graph to states satisfying ϕ. Call this $\mathcal{M}'$. Of $\mathcal{M}'$, we want to know which states comprise fair paths.
- Find the maximal strongly connected components of $\mathcal{M}'$.
- Remove an SCC if, for some fairness constraint ψ_i the SCC does not contain a state satisfying ψ_i. The resulting SCCs are the fair SCCs.
- Use backwards breadth-first search to find the states of $\mathcal{M}'$ that can reach a fair SCC.

Definition 3.6.8 (Fair SCC)**:** A strongly connected component of $\mathcal{M}'$ is *fair* if for all $1 \leq i \leq n$, the SCC contains a state in $\mathtt{SAT}(\psi_i)$. (Where each ψ_i is a fairness constraint, expressed as a CTL formula).

The running time of our $\mathrm{E_C G}$ algorithm is $\Theta(|C| \cdot (|V| + |E|))$.

3.6.5 Fairness Constraints in LTL

Fairness constraints require no special handling in LTL. We express the constraints as LTL formulas and insist that they occur infinitely often. Like this:

$$(\mathrm{G\,F}\,\psi_1 \wedge \mathrm{G\,F}\,\psi_2 \wedge \ldots \wedge \mathrm{G\,F}\,\psi_n) \rightarrow \phi \tag{3.6.1}$$

Translation: if the fairness constraints ψ_i are met, then ϕ has to be true.

3.6.6 LTL Model Checking

There are algorithms for model checking LTL formulas, but they tend to be more complicated than CTL. LTL model checkers tend to be based on (a) tableaux or (b) Büchi automata.

Büchi automata are specialized automata, able to handle strings of infinite length. Why the issue of infinite-length strings? A path through a model can be infinitely long. In the context of an automata, when can you accept a string of infinite length?

3.7 Lecture – 4/9/2007

3.7.1 LTL Model Checking

It's sufficient to give an algorithm that checks whether $\mathcal{M}, s \vDash \mathrm{E}\,\phi$.

Note that $\mathcal{M}, s \vDash \mathrm{A}\,\phi$ IFF $\mathcal{M}, s \nvDash \mathrm{E}\,\neg\phi$. (According to H&R, given ϕ, we'll check $\mathcal{M}, s \vDash \mathrm{E}\,\neg\phi$. If ϕ holds for all paths, $\neg\phi$ won't hold for any. But, if $\neg\phi$ holds for some path, then we'll have a counterexample).

We'll assume that ϕ uses only the following connectives $\top$, $\neg$, $\vee$, X, and U.

Definition 3.7.1 (Closure)**:** The *closure* of ϕ, written $\mathcal{C}(\phi)$ is the set of all positive sub-formulas of ϕ and their negations.

Example 3.7.2: Given $\phi = p\,\mathrm{U}\,q \vee \neg p\,\mathrm{U}\,r$

$$\mathcal{C}(p\,\mathrm{U}\,q \vee \neg p\,\mathrm{U}\,r) = \{p, q, r, p\,\mathrm{U}\,q, \neg p\,\mathrm{U}\,r, \neg p, \neg q, \neg r, \neg(p\,\mathrm{U}\,q), \neg(\neg p\,\mathrm{U}\,r)\}$$

Definition 3.7.3 (automaton)**:** We define an *automaton* A_ϕ as an automaton for ϕ (really a directed graph). A_ϕ accepts traces of propositional atoms such that ϕ is true along the path of the trace.

$$A_\phi = (T, \delta)$$

where δ is a transition relation and T is the set of all (consistent) subsets q of $\mathcal{C}(\phi)$.

Formulas in T must be locally consistent.

- For all positive sub-formulas $\psi \in \mathcal{C}(\phi)$, either $\psi \in q$ or $\neg\psi \in q$, but not both.
- For all $\psi_1 \vee \psi_2 \in \mathcal{C}(\phi)$, $\psi_1 \vee \psi_2 \in q$ IFF $\psi_1 \in q$ or $\psi_2 \in q$.
- For all $\psi_1\,\mathrm{U}\,\psi_2 \in \mathcal{C}(\phi)$, $\psi_1\,\mathrm{U}\,\psi_2 \in q$ IFF $\psi_2 \in q$ or $\psi_1 \in q$.
- For all $\neg(\psi_1\,\mathrm{U}\,\psi_2) \in \mathcal{C}(\phi)$, if $\neg(\psi_1\,\mathrm{U}\,\psi_2) \in q$, then "$\neg\psi_2 \in q$". More precisely, if ψ_2 is positive, then $\neg\psi_2 \in q$.

  On the other hand, if $\psi_2 = \neg\psi_2'$, then $\psi_2' \in q$.

We'll also need a *transition function*, δ, where $(q, q') \in \delta$.

- If $\mathrm{X}\,\psi \in q$, then $\psi \in q'$.
- if $\neg\,\mathrm{X}\,\psi \in q$, then $\neg\psi \in q'$. (This assumes that ψ is positive).
- If $\neg\,\mathrm{X}\,\neg\psi \in q$ then $\psi \in q'$.
- If $\psi_1\,\mathrm{U}\,\psi_2 \in q$ and $\psi_2 \notin q$, then $\psi_1\,\mathrm{U}\,\psi_2 \in q'$.

  Note that this has the effect of "pushing" $\psi_1\,\mathrm{U}\,\psi_2$ along the path.
- Suppose $\neg(\psi_1\,\mathrm{U}\,\psi_2) \in q$ and $\psi_1 \in q$. Then $\neg(\psi_1\,\mathrm{U}\,\psi_2) \in q'$.

These rules describe the transition system $A_\phi = (T, \delta)$. This is an abstract system. It doesn't describe any particular model $\mathcal{M}$.

Our next step is to attach a model to the abstract automata, forming $\mathcal{M} \times A_\phi$.

$$\mathcal{M} \times A_\phi = (U, \delta')$$

where

$$U = \{(s, q) \in S \times T \mid \text{for all atoms } p \in \mathcal{C}(\phi), p \in q \text{ IFF } p \in L(s)\}$$
$$\delta' = \{(s, q), (s', q') \mid s \to s' \in \mathcal{M} \text{ and } q \to q' \in \delta\}$$

Given $\mathcal{M} \times A_\phi$, is ϕ true from a given state s?

Definition 3.7.4 (Eventuality Sequence)**:** An *eventuality sequence* is a path $(s_0, q_0), (s_1, q_1) \in \mathcal{M} \times A_\phi$ such that if $\psi_1 \,\mathrm{U}\, \psi_2 \in q_i$ for some i, then $\psi_2 \in q_j$ for some $j \geq i$.

(It seems like an eventuality sequence is a path, where 'until' holds along that path).

Theorem 3.7.5: $\mathcal{M}, s \vDash \mathrm{E}\,\phi$ IFF there is an eventuality sequence in $\mathcal{M} \times A_\phi$ starting with (s, q) where $\phi \in p$.

The proof of theorem 3.7.5 is pretty long. We'll take it in pieces.

Claim 3.7.6: For all $\psi \in \mathcal{C}(\phi)$ and for all $i \geq 0$, $\pi^i \vDash \psi$ IFF $\psi \in q_i$.

Proof: The proof is by induction on ψ. There are five cases to consider.

1. If $\psi = p$ (an atom), then $\pi^i \vDash p$ IFF $p \in L(s_i)$. This happens IFF $p \in q_i$.
2. $\psi = \neg\psi_1$ (Left as an exercise for the reader).
3. $\psi = \psi_1 \vee \psi_2$ (left as an exercise for the reader).
4. $\psi = \mathrm{X}\,\psi_1$. (assume ψ_1 positive)

   Suppose π^i satisfies ψ. Then $\pi^{i+1} \vDash \psi_1$.

   $\pi^{i+1} \vDash \psi_1$ IFF $\psi_1 \in q_{i+1}$ and IFF $\psi \in q_i$.

   If $\mathrm{X}\,\psi_1 \in q_i$, then $\psi_1 \in q_{i+1}$.

   If $\mathrm{X}\,\psi_1 \notin q_i$, then $\neg\,\mathrm{X}\,\psi_1 \in q_i$ and $\neg\psi_1 \in q_{i+1}$ and $\psi_1 \notin q_{i+1}$.
5. $\psi = \psi_1 \,\mathrm{U}\, \psi_2$. (assume ψ_2 positive). There are two cases to consider.
   (a) Suppose $\pi^i = \psi$. Then there is a $j \geq i$ such that $\pi^j \vDash \psi_2$ and for all $1 \leq i \leq j$, $\pi^i \vDash \psi_1$.

   By the inductive hypothesis, if $\pi^j \vDash \psi_2$, then $\psi_2 \in q_j$, and for all $1 \leq i \leq j$, $\pi_1 \in q_i$.

   Let us choose a minimal j. For $i < j$, we have $\neg\psi_2$ (because j was chosen to be minimal).

   $\psi_1 \,\mathrm{U}\, \psi_j \in q_j$, because $\psi_2 \in q_j$. (If this were not the case, we'd have $\neg(\psi_1 \,\mathrm{U}\, \psi_2) \in q_j$, and $\neg\psi_2 \in q_j$ – a contradiction).

   Suppose $\neg(\psi_1 \,\mathrm{U}\, \psi_2) \in q_i$ and $\psi_1 \in q_i$. By the transition system rules, we'd have $\neg(\psi_1 \,\mathrm{U}\, \psi_2) \in q_{i+1}$ and $\psi_i \in q_{i+1}$. This would mean that $\neg(\psi_1 \,\mathrm{U}\, \psi_2) \in q_j$ – another contradiction.

   $\therefore (\psi_1 \,\mathrm{U}\, \psi_2) \in q_i$.

   (b) Suppose $\psi = \psi_1 \,\mathrm{U}\, \psi_2 \in q_i$. We must show that $\pi^i \vDash \psi$.

   By definition of eventuality sequence, there exists a $j \leq i$ such that $\psi_2 \in q_j$.

   Take j to be minimal. We show by induction on k that for $1 \leq i \leq k \leq j$, that $\psi_1 \,\mathrm{U}\, \psi_2 \in q_k$.

   The Basis case is given: $\psi_1 \,\mathrm{U}\, \psi_2 \in q_i$.

   Inductive Case: Suppose $i \leq k < j$, and $\psi_1 \,\mathrm{U}\, \psi_2 \in q_k$. By our definition of j (minimal), $\psi_2 \notin q_k$. So, $\neg\psi_2 \in q_k$, and $\psi_1 \,\mathrm{U}\, \psi_2 \in q_{k+1}$.

   For $i \leq k < j$, we have $\psi_1 \,\mathrm{U}\, \psi_2 \in q_k$. But we also know that $\psi_2 \notin q_k$ (j was chosen to be minimal). So $\psi_1 \in q_k$.

   By the inductive hypothesis, for $1 \leq k < j$, $\pi^k \vDash \psi_1$ and $\pi^j \vDash \psi_2$, so $\pi^i \vDash \psi_1 \,\mathrm{U}\, \psi_2 = \psi$.

□

See Section 3.9.2 (page 94) for the second half of the proof.

3.8 H&R Notes on LTL Model Checking – 4/11/2007

3.8.1 States of A_ϕ

The states of A_ϕ (denoted as q are the *maximal* subsets of $\mathcal{C}(\phi)$ that satisfy the following conditions:

- For all (non-negated) $\psi \in \mathcal{C}(\phi)$, either $\psi \in q$ or $\neg\psi \in q$, but not both.
- $\psi_1 \vee \psi_2 \in q$ holds IFF $\psi_1 \in q$ or $\psi_2 \in q$ whenever $\psi_1 \vee \psi_2 \in \mathcal{C}(\phi)$.
- Conditions for other boolean combinations are similar.
- If $\psi_1 \,\mathrm{U}\, \psi_2 \in q$, then $\psi_2 \in q$ or $\psi_1 \in q$.
- If $\neg(\psi_1 \,\mathrm{U}\, \psi_2) \in q$, then $\neg\psi_2 \in q$.

Intuitively, these conditions imply that the state of A_ϕ are capable of saying which sub-formulas of ϕ are true.

The initial states of A_ϕ are those states containing ϕ.

3.8.2 Transitions of $A\phi$

δ is the transition relation of A_ϕ. Two states $(q, q') \in \delta$ IFF all of the following conditions hold:

- If $\mathrm{X}\,\psi \in q$, then $\psi \in q'$
- If $\neg\,\mathrm{X}\,\psi \in q$, then $\neg\psi \in q'$
- If $\psi_1 \,\mathrm{U}\, \psi_2 \in q$ and $\psi_2 \notin q$, then $\psi_1 \,\mathrm{U}\, \psi_2 \in q'$. (Note: this has the effect of "pushing" the until along the path).
- If $\neg(\psi_1 \,\mathrm{U}\, \psi_2) \in q$ and $\psi_1 \in q$, then $\neg(\psi_1 \,\mathrm{U}\, \psi_2) \in q'$. (Again, pushing the until).

These rules are based on the recursion laws:

$$\psi_1 \,\mathrm{U}\, \psi_2 \equiv \psi_2 \vee (\psi_1 \wedge \mathrm{X}(\psi_1 \,\mathrm{U}\, \psi_2)) \tag{3.8.1}$$

$$\neg(\psi_1 \,\mathrm{U}\, \psi_2) \equiv \neg\psi_2 \wedge (\neg\psi_1 \vee \mathrm{X}\,\neg(\psi_1 \,\mathrm{U}\, \psi_2)) \tag{3.8.2}$$

3.9 Lecture – 4/11/2007

3.9.1 LTL Model Checking

Last class, we began to look at proofs for LTL model checking (See theorem 3.7.5, page 91). We were trying to prove

> $\mathcal{M}, s \vDash \mathrm{E}\,\phi$ IFF there is an eventuality sequence in $\mathcal{M} \times A_\phi$ that starts from some (s, q) with $\phi \in q$

Last class we showed that there is an eventuality sequence if $\mathcal{M}, s \vDash \mathrm{E}\,\phi$. In this class, we'll prove the opposite direction.

States in A_ϕ are consistent sets of sub-formulas of ϕ (and negations of sub-formulas of ϕ).

When forming $\mathcal{M} \times A_\phi$, we must maintain a consistent structure. If $p \in q$ for an atomic formula p and $q \in A_\phi$, then p must be in $L(s)$ for the corresponding state s in $\mathcal{M}$.

Recall that an *eventuality sequence* is a path in $\mathcal{M} \times A_\phi$ that must eventually reach a condition. Given $\psi_1 \,\mathrm{U}\, \psi_2$, ψ_2 must eventually become true.

If ϕ has no U connectives, then every sequence will be an eventuality sequence.

3.9.2 Proof of Theorem 3.7.5, Continued

Here, we show that $\mathcal{M}, s \vDash \mathrm{E}\,\phi$ if there is an eventuality sequence.

Proof: Suppose that $\mathcal{M}, s \vDash \mathrm{E}\,\phi$.

- If $\mathcal{M}, s \vDash \mathrm{E}\,\phi$, then there is a path $\pi = s_0, s_1, \ldots$ in $\mathcal{M}$ with $\pi \vDash \phi$. Let

  $$q_i = \{\psi \in \mathcal{C}(\phi) \mid \pi^i \vDash \psi\}$$

  q_i is the set of sub-formulas that are true starting at state π^i.
- Is $q_i \in T$? (T is the transition system states for A_ϕ).
  - if $\psi_1 \,\mathrm{U}\, \psi_2 \in q_i$, then $\psi_1 \in q_i$ or $\psi_2 \in q_i$.
  - If $\pi^i \vDash \psi_1 \,\mathrm{U}\, \psi_2$, then (1) $\pi^1 \vDash \psi_1$ or (2) $\pi^i \vDash \psi_2$.
  - Therefore, $\psi_1 \in q_i$ or $\psi_2 \in q_i$.
  - Checking the other requirements of T is done in a similar manner.
- Is $(s_i, q_i) \in \mathcal{M} \times A_\phi$?
  - $p \in q_i$ IFF $p \in L(s_i)$ (For an atomic variable p).
  - $p \in q_i$ IFF $\pi^i \vDash p$, IFF $p \in L(s_i)$.
  - This covers states, next we handle transitions.
- Is $(s_i, q_i) \in \mathcal{M} \times A_\phi$?
  - $\phi \in q_0$ such that $\pi^0 = \pi \vDash \phi$. (Where $q_0 \in \mathcal{C}(\phi)$).

    This covers the starting state.
- Is $((s_i, q_i), (s_{i+1}, q_{i+1})) \in \delta'$?
  - The transition $s_i \to s_{i+1}$ comes from $\mathcal{M}$.

- If $\mathrm{X}\,\psi \in q_i$ then $\psi \in q_{i+1}$.

  Because $\mathrm{X}\,\psi \in q_i$, we also have $\pi^i \vDash \mathrm{X}\,\psi$. Therefore, $\pi^{i+1} \vDash \psi$.

- If $\psi_1 \,\mathrm{U}\, \psi_2 \in q_i$ and $\psi_2 \notin q_i$, then $(\psi_1 \,\mathrm{U}\, \psi_2) \in q_{i+1}$.

  This is the "pass the buck" part of δ's definition. Because we're dealing with an eventuality sequence, if ψ_2 is not true now, we know that ψ_2 will be true sometime later in the future.

  We have $\pi^i \vDash \psi_1 \,\mathrm{U}\, \psi_2$, and $\pi^i \nvDash \psi_2$.

  This implies that $\psi_1 \,\mathrm{U}\, \psi_2 \in q_{i+1}$ and $\pi^{i+1} \vDash \psi_1 \,\mathrm{U}\, \psi_2$.

- Why do we have an eventuality sequence?
  - $((s_0, q_0), (s_1, q_1), \ldots)$ is an eventuality sequence.
  - Suppose $\psi_1 \,\mathrm{U}\, \psi_2 \in q_i$. Then $\pi^i \vDash \psi_1 \,\mathrm{U}\, \psi_2$.

    This implies that there is a $j \geq i$ such that $\pi^j \vDash \psi_2$, which implies that $\psi_2 \in q_j$.

□

3.9.3 An Algorithm For LTL Model Checking

The algorithm is based on Theorem 3.7.5.

$\mathcal{M}, s \vDash \mathrm{E}\,\phi$ IFF there is an eventuality sequence in $\mathcal{M} \times A_\phi$ starting from (s, q) with $\phi \in q$.

The algorithm looks a lot like $\texttt{SAT}_{\mathrm{EG}}$

1. Find all strongly connected components of $\mathcal{M} \times A_\phi$.
2. Retain those strongly connected components that satisfy the eventuality sequence. (This is similar to $\mathrm{E_C G}$, where we kept only paths that satisfied fairness constraints).

   If the strongly connected components contains $\psi_1 \,\mathrm{U}\, \psi_2$, then that strongly connected component must contain ψ_2 in order to satisfy the eventuality sequence.

Definition 3.9.1 (Self-fulfilling SCC)**:** A non-trivial strongly connected component C of $\mathcal{M} \times A_\phi$ is *self-fulfilling* if, for every $(s, q) \in C$ with $\psi_1 \,\mathrm{U}\, \psi_2 \in q$, there is an $(s', q') \in C$ with $\psi_2 \in q'$.

Theorem 3.9.2: There is an eventuality sequence starting at (s, q) in $\mathcal{M} \times A_\phi$ IFF there is a path from (s, q) to a self-fulfilling strongly connected component of $\mathcal{M} \times A_\phi$.

Proof: Suppose there is an eventuality sequence π starting at (s, q).

- Along π, some states occur finitely often; some states occur infinitely often.
- π can be written $\pi = \pi_0\pi_1$, where π_0 contains only those states that occur finitely often and π_1 contains only those states that occur infinitely often. (Each π_i can have multiple states).

  In π_1, every state occurs infinitely often.
- Let C' be the set of states (s, q) that occur infinitely often in π.
- Every state in C' is reachable from every other state (because all states in C' occur infinitely often).
- C' is strongly connected. Furthermore, C' is contained in some strongly connected component C of $\mathcal{M} \times A_\phi$.

$$C' \subseteq C \subseteq \mathcal{M} \times A_\phi \tag{3.9.1}$$

- π_0 is a path from (s, q) to C. (π_0 gets us into the strongly connected component C).
- If $\psi_1 \,\mathrm{U}\, \psi_2 \in C'$, then by the definition of eventuality sequence, $\psi_2 \in C'$.
- If $(s, q) \in C'$ and $\psi_1 \,\mathrm{U}\, \psi_2 \in q_i$, then by the definition of eventuality sequence and construction of C', there is an $(s', q') \in C'$ with $\psi_2 \in q'$. (Because $C' \subseteq C$).
- If $(s, q) \in C - C'$ and $\psi_1 \,\mathrm{U}\, \psi_2 \in q$, then there is a path $\pi_2 \in C$ from (s, q) to C'.
  - Case 1: ψ_2 occurs on π_2. Then $\pi_2 \in C$.
  - Case 2: ψ_2 does not occur on π_2 Then $\psi_1 \,\mathrm{U}\, \psi_2 \in C'$ and $\psi_2 \in C'$. Therefore $\psi_2 \in C$.

  It's possible to have $\neg\psi_1 \in C'$ – the transition relation simply won't let us go there. ($\neg\psi_1$ would make $\psi_1 \,\mathrm{U}\, \psi_2$ false).

Suppose there is a path in $\mathcal{M} \times A_\phi$ from (s, q) to a self-fulfilling strongly connected component C.

- Let π_0 be the path from (s, q) to C. Let π_1 be a finite path in C that includes all nodes of C, and let π_1 start and end in the same place.
- The path is $\pi_0\pi_1\pi_1 \ldots$. (This is our earlier definition of $\pi = \pi_0\pi_1$, where elements in π_1 occur infinitely often).
- If $\psi_1 \,\mathrm{U}\, \psi_2 \in \pi_1$, then $\psi_2 \in \pi_1$ because C is a self fulfilling strongly connected component.
- If $\psi_1 \,\mathrm{U}\, \psi_2 \in \pi_0$, then (1) $\psi_1 \,\mathrm{U}\, \psi_2$ is true in π_0 or (2) $\psi_1 \,\mathrm{U}\, \psi_2$ is true in π_1. In either case, we have ψ_2 true in π_1.

□

3.9.4 LTL Model Checking Pseudocode

This is the pseudocode for the LTL model-checking algorithm. The input ϕ is an LTL formula.

```
procedure SAT_E(φ)
    Y = ⋃{c | c is a self-fulfilling SCC of M × A_φ}
    T = Y
    while T ≠ ∅ do
        choose an s ∈ T
        T = T − {s}
        for all t such that t → s do
            if t ∉ Y then
                T = T ∪ {t}
                Y = Y ∪ {t}
            end if
        end for
    end while
    return {s | (s, q) ∈ Y and φ ∈ q}
end procedure
```

Running time of $\mathtt{SAT}_\mathtt{E}$

- $\mathcal{M} \times A_\phi$ grows exponentially with the size of the formula. The size is based on $|V| \cdot 2^{|\phi|}$.
- We have an algorithm to find strongly connected components it linear time (proportional to $\mathcal{M} \times A_\phi$). The algorithm is linear – the size of the problem is exponential.

- $\texttt{SAT}_\texttt{E}$ is practical for small models. In reality, we'll model check small formulas – small enough for a human to understand.

  The state explosion problem is more a function of the size of $\mathcal{M}$. Adding one variable can double the number of model state.

- $\therefore$ in practical situations, $|V|$ has a more significant size contribution than ϕ.

3.10 Lecture - 4/18/2007

3.10.1 CTL Model Checking and Fixed Points

Given a model $\mathcal{M} = (S, \rightarrow, L)$ and a formula ϕ, we are interested in finding $\texttt{SAT}(\phi)$.

We're going to introduce a new notation:

$$\texttt{SAT}(\phi) \equiv [\![\phi]\!] \tag{3.10.1}$$

The notation $[\![\phi]\!]$ comes from the field of denotational semantics. Here, $[\![\phi]\!]$ denotes the set of states that satisfy ϕ.

We'd like to rewrite $\texttt{SAT}_{\texttt{EG}}$ is a more straightforward (albeit less efficient) way.

The goal of $\texttt{SAT}_{\texttt{EG}}$ is to find paths where ϕ is true globally. We can do this by (1) finding states where $\texttt{SAT}(\phi)$ holds, and (2) finding predecessors to these states where $\texttt{SAT}(\phi)$ holds.

Our algorithm:

procedure $\texttt{SAT}_{\texttt{EG}}(\phi)$
  $Y = \texttt{SAT}(\phi)$
  $X = \emptyset$
  while $X \neq Y$ **do**
    $X = Y$
    $Y = Y \cap \mathsf{pre}_\exists(Y)$
  end while
  return Y
end procedure

The function $\mathsf{pre}_\exists$ was defined on page 83 (equation (3.5.1)).

To analyze this algorithm, let's review a few equivalences:

$$\begin{aligned} \mathrm{AF}\,\phi &\equiv \phi \vee \mathrm{AX}\,\mathrm{AF}\,\phi \\ \mathrm{E}[\phi\,\mathrm{U}\,\psi] &\equiv \psi \vee (\phi \wedge \mathrm{EX}\,\mathrm{E}[\phi\,\mathrm{U}\,\psi]) \\ \mathrm{EG} &\equiv \phi \wedge \mathrm{EX}\,\mathrm{EG}\,\phi \end{aligned}$$

Some more equivalences, using our new set notation

$$\begin{aligned} [\![\mathrm{AX}\,\phi]\!] &\equiv \mathsf{pre}_\forall([\![\phi]\!]) && \text{all predecessors to states that satisfy } \phi \\ [\![\mathrm{EX}\,\phi]\!] &\equiv \mathsf{pre}_\exists([\![\phi]\!]) && \text{some predecessor to states that satisfy } \phi \\ [\![\mathrm{AF}\,\phi]\!] &\equiv [\![\phi]\!] \cup \mathsf{pre}_\forall([\![\mathrm{AF}\,\phi]\!]) \\ [\![\mathrm{E}[\phi\,\mathrm{U}\,\psi]]\!] &\equiv [\![\psi]\!] \cup ([\![\phi]\!] \cap \mathsf{pre}_\exists([\![\mathrm{E}[\phi\,\mathrm{U}\,\psi]]\!])) \\ [\![\mathrm{EG}\,\phi]\!] &\equiv [\![\phi]\!] \cap \mathsf{pre}_\exists([\![\mathrm{EG}\,\phi]\!]) \end{aligned}$$

These are very similar to equivalences we've seen before. The difference is that we're using set notation instead of logic formulas.

Let us define a function

$$F\colon \mathcal{P}(S) \rightarrow \mathcal{P}(S) \tag{3.10.2}$$

Definition 3.10.1 (Fixed Point)**:** We way that X is a *fixed point* if $F(X) = X$.

Example 3.10.2: Suppose we define F as

$$F(X) = [\![\phi]\!] \cup \mathsf{pre}_\forall(X)$$

Notice the similarity to our definition of $[\![\mathrm{AF}\,\phi]\!]$. Indeed, we can substitute

$$\begin{aligned} F([\![\mathrm{AF}\,\phi]\!]) &= [\![\phi]\!] \cup \mathsf{pre}_\forall([\![\mathrm{AF}\,\phi]\!]) \\ &= [\![\mathrm{AF}\,\phi]\!] \end{aligned}$$

Thus, $[\![\mathrm{AF}\,\phi]\!]$ is a fixed point of F. □

Definition 3.10.3 (Monotone)**:** We say that $F\colon \mathcal{P}(S) \to \mathcal{P}(S)$ is *monotone* when the following condition holds

$$X \subseteq Y \text{ implies } F(X) \subseteq F(Y) \tag{3.10.3}$$

For all subsets X and Y of S.

Using our earlier definition: $F(X) = [\![\phi]\!] \cup \mathsf{pre}_\forall(X)$,

- If $X \subseteq Y$, then $\mathsf{pre}_\forall(X) \subseteq \mathsf{pre}_\forall(Y)$
- So, $[\![\phi]\!] \cup \mathsf{pre}_\forall(X) \subseteq [\![\phi]\!] \cup \mathsf{pre}_\forall(Y)$.
- So, $F(X) \subseteq F(Y)$
- $\therefore F$ is monotone.

Theorem 3.10.4: All monotone functions have fixed points.

Example 3.10.5: Let $S = \{s_0, s_1\}$ and let $F(X) = X \cup \{s_0\}$

$$\begin{aligned} F(\emptyset) &= \{s_0\} \\ F(\{s_0\}) &= \{s_0\} \\ F(\{s_1\}) &= \{s_0, s_1\} \\ F(\{s_0, s_1\}) &= \{s_0, s_1\} \end{aligned}$$

F is a monotone function. F also has a least fixed point and a greatest fixed point.

Example 3.10.6: Let $G(X)$ be

$$G(X) = \begin{cases} \{s_1\} & \text{if } x \neq \{s_1\} \\ \{s_0\} & \text{if } x = \{s_1\} \end{cases}$$

We can show that G is not monotone:

$$\begin{aligned} \emptyset &\subseteq \{s_1\} \\ G(\emptyset) &= \{s_1\} \\ G(\{s_1\}) &= \{s_0\} \end{aligned} \qquad \text{Not monotone: } G(\emptyset) \not\subseteq G(\{s_1\})$$

G has no fixed points.

If $F\colon \mathcal{P}(S) \to \mathcal{P}(S)$ is monotone, then F has both least and greatest fixed points (the least and greatest fixed points may be the same).

Notation:

$$F^n(X) = F(F(\ldots(F(X))\ldots)) \qquad F \text{ applied to } X \ n \text{ times} \tag{3.10.4}$$

Theorem 3.10.7 (H&R Theorem 3.24)**:** Let S be a set with $n+1$ elements: $S = \{s_0, s_1, \ldots, s_n\}$. If $F\colon \mathcal{P}(S) \to \mathcal{P}(S)$ is a monotone function, then $F^{n+1}(\emptyset)$ is the least fixed point of F, and $F^{n+1}(S)$ is the greatest fixed point of F.

Through $n+1$ applications of F, we have the following relation:

$$\emptyset \subseteq F(\emptyset) \subseteq F^2(\emptyset) \subseteq \ldots \subseteq F^n(\emptyset) \subseteq F^{n+1}(\emptyset) \tag{3.10.5}$$

Because the number of elements in S is finite, we will eventually reach a point where $F^i(\emptyset) \subseteq F^{i+1}(\emptyset)$.

Since $|S| = n+1$, there is a $k \leq n+1$ with $F^k(\emptyset) = F^{k+1}(\emptyset) = F(F^k(\emptyset))$. This is a fixed point. But why is it a *least* fixed point?

Suppose $F(X) = X$ is any fixed point. We have

$$\begin{aligned}
\emptyset &\subseteq X \\
F(\emptyset) &\subseteq F(X) = X && \text{by monotonicity} \\
F^2(X) &\subseteq F(X) = X \\
&\vdots \\
F^n(\emptyset) &\subseteq F(X) = X
\end{aligned}$$

Therefore $F^m(\emptyset)$ is a least fixed point.

The proof for greatest fixed point is similar (but the 'direction' is reversed).

Claim 3.10.8: $[\![\mathrm{EG}\,\phi]\!]$ is the greatest fixed point of F.

Proof: Let X be any fixed point of F. We must show that $X \subseteq [\![\mathrm{EG}\,\phi]\!]$.

Suppose $s \in X$, and $X \in F(X)$, and $F(X) = [\![\phi]\!] \cap \mathsf{pre}_\exists(X)$.

ϕ is true at s and there is an s' with $s \to s'$ and $s' \in X$.

Therefore ϕ is true at s', and there is an $s'' \in X$ with $s' \to s''$.

This gives us a path starting with s, where ϕ is true in every state along that path. Therefore $s \in [\![\mathrm{EG}\,\phi]\!]$. □

Claim 3.10.9: The algorithm $\mathtt{SAT}_{\mathtt{EG}}$ (page 98) computes a greatest fixed point.

Let $X = S$. We have

$$Y = S \cap \mathsf{pre}_\exists(S) = \mathtt{SAT}(\phi) = Y$$

Claim 3.10.10: If we replace $Y = Y \cap \mathsf{pre}_\exists(Y)$ with

$$Y = \mathtt{SAT}(\phi) \cap \mathsf{pre}_\exists(Y)$$

in $\mathtt{SAT}_{\mathtt{EG}}$, then we will get the same result.

Proof: Let Y_i be Y after i iterations.

$$\begin{aligned}
Y_0 &= \mathtt{SAT}(\phi) \\
Y_{i+1} &= Y_i \cap \mathsf{pre}_\exists(Y_i)
\end{aligned}$$

For $i \geq 0$, $Y_{i+1} = \mathtt{SAT}(\phi) \cap \mathsf{pre}_\exists(Y_i)$.

Basis:
For $i = 0$, $Y = Y_0 = \mathtt{SAT}(\phi)$.
For $i = 1$, $Y = Y_1 = Y_1 \cap \mathsf{pre}_\exists(Y_0)$.

Suppose this holds for i iterations.

$$\begin{aligned} Y_{i+2} &= Y_{i+1} \cap \mathsf{pre}_\exists(Y_{i+1}) \\ &= \mathtt{SAT}(\phi) \cap \mathsf{pre}_\exists(Y_i \cap \mathsf{pre}_\exists(Y_{i+1}) \\ &= \mathtt{SAT}(\phi) \cap \mathsf{pre}_\exists(Y_{i+1}) \\ Y_{i+1} &\subseteq Y_i \end{aligned}$$

□

The subset relation for greatest fixed points:

$$F(S) \supseteq F^2(S) \supseteq F^3(S) \supseteq \ldots \supseteq F^{n+1}(S) \tag{3.10.6}$$

3.10.2 $\mathrm{E}[\phi \,\mathrm{U}\, \psi]$ as a Fixed Point Computation

We have the equivalence

$$[\![\mathrm{E}[\phi \,\mathrm{U}\, \psi]]\!] \equiv [\![\psi]\!] \cup ([\![\phi]\!] \cap \mathsf{pre}_\exists([\![\mathrm{E}[\phi \,\mathrm{U}\, \psi]]\!]))$$

So,

$$F(X) = \mathtt{SAT}(\psi) \cup ([\![\phi]\!] \cap \mathsf{pre}_\exists(X))$$

$[\![\mathrm{E}[\phi \,\mathrm{U}\, \psi]]\!]$ is a fixed point of F, but it is a *least* fixed point.

Let X be a fixed point of F. We want to determine the meaning of $\mathrm{E}[\phi \,\mathrm{U}\, \psi] \subseteq X$.

$$X = F(X) = [\![\psi]\!] \cup ([\![\phi]\!] \cap \mathsf{pre}_\exists(X))$$

Suppose that $s \in [\![\mathrm{E}[\phi \,\mathrm{U}\, \psi]]\!]$. Then there is a path where ψ is true and ϕ is true up until that point. (We start by computing $F(\emptyset) = \mathtt{SAT}(\phi)$?)

3.10.3 For Next Class

We'll start chapter 6 next. Read over the first few pages.

3.11 Nice Presentation on CTL/LTL

I came across this one day

`http://www-ti.informatik.uni-tuebingen.de/~weissr/doc/FDL04-final.pdf`

It's a nice presentation on CTL/LTL.

Part 4

Binary Decision Diagrams

This material is covered in Chapter 6 of H&R

4.1 Lecture – 4/23/2007

4.1.1 Introduction To Binary Decision Diagrams

Binary Decision Diagrams (BDDs) are data structures that allowed model checking to go from theoretical concepts to working implementations. BDDs don't solve the state explosion problem completely, but they do make its effects less pronounced.

BDDs can be thought of as representing sets of states in a transition system. They can also be used to represent truth functions over a set of variables V.

Definition 4.1.1 (Truth Formula)**:** Let V be a set of variables. A *truth function* over V is a function

$$f\colon (V \to \{0,1\}) \to \{0,1\} \tag{4.1.1}$$

$(V \to \{0,1\})$ represents the assignment of truth values to variables. f is the valuation of the truth function under that variable assignment.

Operators in truth functions are similar to those of propositional logic.

- $u \cdot v$. True if $u = v = 1$ (like $\wedge$)
- $u + v$. True if $u = 1$ or $v = 1$ (like $\vee$)
- $u \oplus v$. True if u and v have different values (XOR).
- $\overline{0} = 1$ and $\overline{1} = 0$. This is negation. (Above, v represents a tree node).

Definition 4.1.2 (Binary Decision Tree (BDT))**:** A *binary decision tree* (BDT) over the set of variables V is a finite tree T such that

1. each leaf is labeled with 0 or 1,
2. each interior node is labeled with a variable $\mathsf{var}(v) \in V$, and
3. each interior node has exactly two children: $\mathsf{lo}(v)$ and $\mathsf{hi}(v)$.

Figures 4.1 and 4.2 show two examples of Binary Decision Trees. The lo edges are represented by dashed lines while the hi edges are represented by solid lines.

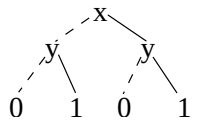

Figure 4.1: Binary Decision Tree Example 1

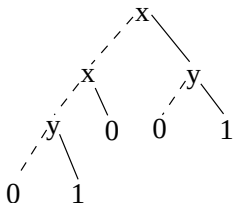

Figure 4.2: Binary Decision Tree Example 2

To evaluate Figure 4.1 with the assignment $\{x = 0, y = 1\}$ we start with x, take the lo edge to the left y, then take the hi edge to the leaf 1.

If we evaluate Figure 4.2 with the same assignment, we take the lo edge from x at the root, the lo edge from x beneath the root, and the hi edge to the leaf 1.

Figure 4.2 shows a Binary Decision Tree with unreachable nodes. For example, the x below the root is only reached if $x = 0$; there is no way to reach the 0 that is $\mathsf{hi}(x)$.

4.1.2 Evaluating Binary Decision Trees (More Formally)

Given a BDT over V, each node v of T determines a truth function f_v over V.

- If v is a leaf labelled b (for $b \in \{0, 1\}$), then $f_v(\tau) = b$.
- If v is not a leaf then

$$f_v(\tau) = \begin{cases} f_{\mathsf{lo}(v)}(\tau) & \text{if } \tau(\mathsf{var}(v)) = 0 \\ f_{\mathsf{hi}(v)}(\tau) & \text{if } \tau(\mathsf{var}(v)) = 1 \end{cases}$$

4.1.3 Other Ways to Represent Truth Functions

We have several ways to represent truth functions (each with their advantages and disadvantages):

- Truth tables. (Conceptually simple, but they're exponentially large)
- Formulas (with no restriction on form)
- DNF. Formulas in Disjunctive normal form have the form $(p \wedge q \wedge r) \vee (q \wedge r) \vee \ldots$. (A disjunction of clauses, where each clause is a conjunction of literals).
- CNF
- BDTs
- BDDs

Constructed naively, a BDT can be as large as a truth table, but there are techniques for reducing their size (we'll reduce them, forming BDDs).

Table 4.1 compares several methods of representing truth formulas. A few notes on Table 4.1:

(1) Only one clause needs to be satisfiable in order for a DNF formula to be satisfiable. A single clause is satisfiable if it does not have the form $p \wedge \neg p$.

	compact	satisifiability	validity (2)	equivalence	$\cdot$	$+$	$^-$
Truth Table	no	never	never	never	never	never	never
Formula	often	hard	hard	hard	easy	easy	easy
DNF	sometimes	easy (1)	hard	hard	hard (3)	easy	hard (3)
CNF	sometimes	hard	easy	hard?	easy	hard	hard
BDD	often (4)	easy	easy	easy	easy	easy	easy
BDT	sometimes						

Table 4.1: Comparison of Formula Representations

(2) Note that validity can be reduced to equivalence. Is $\phi \equiv \top$?

(3) Potentially, these operations could require a lot of distributivity.

(4) BDDs are often compact, but not always.

4.1.4 BDTs to BDDs

Binary Decision Diagrams are a generalization of Binary Decision Trees. The tree structure is relaxed – binary decision diagrams are DAGs rather than trees.

For example, where a BDT gives each leaf explicitly, a BDD will have two "leaves": $\{0, 1\}$.

A BDT can be transformed into a BDD by applying the following optimizations:

(C1) Remove duplicate leaves. If there are two nodes labeled 0 (or 1), then combine them.

(C2) If $\mathsf{lo}(n) = \mathsf{hi}(n)$ for an internal node n, then remove n (it doesn't affect the evaluation). All edges leading into n now go to $\mathsf{lo}(n)$.

(C3) If $\mathsf{var}(n) = \mathsf{var}(m)$ for two nodes m, n; and $\mathsf{lo}(n) = \mathsf{lo}(m)$; and $\mathsf{hi}(n) = \mathsf{hi}(m)$, then combine m and n. Remove m. All edges that led to m will now lead to n.

Acyclicity is essential for a BDD. The acyclicity is guarantees that any evaluation will reach a leaf – it won't get caught in a loop.

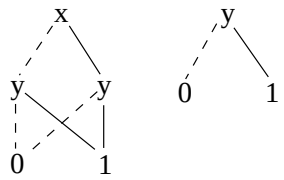

Figure 4.3: Two (Equivalent) Binary Decision Diagrams

Figure 4.3 shows two equivalent binary decision diagrams. The diagram on the left is a BDT after applying optimization (C1). The right diagram combines the two y nodes (optimization C3), and removes the ineffective x node (optimization C2).

Next, we'll formalize BDDs a little more.

Definition 4.1.3 (Initial Node)**:** An *initial node* of a DAG is a node with no inward edges.

Definition 4.1.4 (Terminal Node)**:** An *terminal node* of a DAG is a node with no outward edges.

Definition 4.1.5 (Binary Decision Diagram)**:** A binary decision diagram is a finite DAG where

1. The DAG has one initial node

2. Each terminal node is labeled with $b \in \{0, 1\}$.
3. Each non-terminal node v is labeled with a variable $\mathsf{var}(v)$.
4. Each non-terminal node v has exactly two outgoing edges. One edge leads to $\mathsf{lo}(v)$, the other leads to $\mathsf{hi}(v)$.

Definition 4.1.6 (Reduced BDD)**:** A binary decision diagram is said to be *reduced* if none of the optimizations C1, C2, C3 can be applied.

If T is a binary decision diagram over V, then each node v determines a truth function over V, f_v. This is the same thing we saw with binary decision trees.

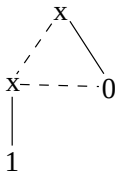

Figure 4.4: A reduced, but unsatisfiable, BDD

Figure 4.4 gives an example of a reduced BDD that is not satisfiable.

4.1.5 Some Operations on BDDs

Satisifiability of a BDD is computed from the bottom to the top. Is there a path from a 1 leaf to the root such that variable assignment is consistent along that path?

Validity of a BDD is computed from the top down. Do all paths representing consistent assignments bring us to a 1 leaf?

Let B_f and B_g be two binary decision diagrams.

- To compute $B_f \cdot B_g$, we replace the 1 leaves of B_f with the initial node of B_g.
- To compute $B_f + B_g$, we replace the 0 leaves of B_f the initial node of B_g.
- To compute $\overline{B_f}$, we swap the leaves $\{0, 1\}$.

If B_f and B_g are reduced BDDs, their combination $B_f \cdot B_g$ (or $B_f + B_g$) may not be reduced. At the very least, there will be duplicate terminal nodes.

It is possible for two different reduced BDDs to represent the same truth function.

Definition 4.1.7 (Ordered Binary Decision Diagram (OBDD))**:** Let $x = \{x_1, \ldots, x_n\}$ be an ordered listing of the set of variables in V. A binary decision diagram T is an *ordered binary decision diagram* if, for all non-terminal nodes v

1. $\mathsf{var}(v) < \mathsf{var}(\mathsf{lo}(v))$ if $\mathsf{lo}(v)$ is not a terminal node, and
2. $\mathsf{var}(v) < \mathsf{var}(\mathsf{hi}(v))$ if $\mathsf{hi}(v)$ is not a terminal node.

We say that two orderings x, x' are *compatible* if no variables occur in one order in x, and occur in the opposite order in x'.

Theorem 4.1.8: If T, T' are two binary decision diagrams with compatible orderings x, x' for the same truth function f, then $T = T'$. In other words, T and T' are *isomorphic*.

4.2 Lecture – 4/25/2007

4.2.1 Binary Decision Diagrams

In this section, we'll frequently use the phrase "Reduced Ordered Binary Decision Diagram". A reduced OBDD is one that cannot be changed by performing optimizations C1–C3 (see page 104). *Ordered* refers to definition 4.1.7.

Theorem 4.2.1: If B_1, B_2 are reduced OBDDs that compute the same truth function, then B_1, B_2 are isomorphic.

Definition 4.2.2 (Isomorphic)**:** B_1, B_2 are *isomorphic* if there is a bijection from the nodes of B_1 to the nodes of B_2 such that:

1. If n is a terminal node of B_1, then $h(n)$ is a terminal node of B_2 and $\mathsf{value}(n) = \mathsf{value}(h(n))$. ($\mathsf{value}(n)$ is a binary value in $\{0, 1\}$).
2. If n is a non-terminal node, then $h(n)$ is a non-terminal node and

$$\begin{aligned} \mathsf{var}(n) &= \mathsf{var}(h(n)) \\ \mathsf{lo}(h(n)) &= h(\mathsf{lo}(n)) \\ \mathsf{hi}(h(n)) &= h(\mathsf{hi}(n)) \end{aligned}$$

Claim 4.2.3: For reduced OBDDs B_1 and B_2, if B_1, B_2 represent the same formula, then B_1, B_2 have identical structures.

When reducing an OBDD, the same structure will be produced regardless of what order the reductions are applied. (order doesn't matter).

In an OBDD, two different nodes represent two different (sub) formulas.

Theorem 4.2.4 (Validity Test for Reduced OBDDs)**:** f is valid IFF the reduced OBDD for f is $\boxed{1}$.

Theorem 4.2.5 (Satisfiability Test for Reduced OBDDs)**:** f is satisfiable IFF the reduced OBDD for f is not $\boxed{0}$.

In a reduced OBDD, there are no unreachable paths. This is not necessarily the case for Unordered BDDs.

Every BDD for a function can be ordered. Let f be a function. We can always write a truth table for f. Because we can write a truth table for f, we can write a BDT for f (where the variables are ordered in the same way that the truth table is ordered). Given an ordered BDT for f, we can reduce it to form an OBDD.

4.2.2 Ordering Matters for an OBDD

In this section, we'll consider how ordering can affect the size of an OBDD.

Let f be the function

$$f(x_1, \ldots, x_{2n}) = (x_1 + x_2) \cdot (x_3 + x_4) \cdot \ldots \cdot (x_{2n-1} + x_{2n}) \tag{4.2.1}$$

If we order the BDD as $(x_1, x_2, \ldots, x_{2n-1}, x_{2n})$ the reduced OBDD will have size $2n + 2$.

If we order the BDD as $(x_1, x_3, \ldots, x_{2n-1}, x_2, x_4, \ldots, x_{2n})$ then the reduced OBDD will have size 2^{n+1} (BIG!)

This shows that picking the right order can result in a very compact representation, while picking the wrong order can result in a very big representation (as big as the truth table).

Some boolean functions do not have any "small" OBDD, no matter what ordering is chosen.

Example 4.2.6: Let f be

$$f(a_n, \ldots, a_0, b_n, \ldots, b_0) = (a_n \ldots a_0)_2 \times (b_n \ldots b_0)_2 \tag{4.2.2}$$

Where the output is the two middle bits. f treats a and b as bit vectors, performing base-2 multiplication.

Any OBDD that represents this function has $2 \cdot (1.09)^n$ nodes.

4.2.3 Algorithms for OBDDs

The Reduce Algorithm

Reduce takes an OBDD as input and applies optimizations C1–C3. The reduction is performed from the bottom up, as follows.

We can think of the OBDD as having "layers", where each layer represents a single variable.

1. At the terminal layer, combine all $\boxed{0}$ and $\boxed{1}$ nodes.
2. For $i = n \ldots 1$, process the x_i layer as follows:
   (a) if $\mathsf{lo}(n) = \mathsf{hi}(n)$, then remove n. All edges into n now go to $\mathsf{lo}(n)$.
   (b) If n' is an earlier x_i node, and $\mathsf{lo}(n') = \mathsf{lo}(n)$ and $\mathsf{hi}(n') = \mathsf{hi}(n)$, then remove n. Edges that formerly went into n now lead into n'.
3. Otherwise, leave n alone.

The Apply algorithm

Let

- op be op: $\{0,1\} \times \{0,1\} \to \{0,1\}$ (i.e. - a binary truth operator).
- Let B_f be a reduced OBDD for the function f
- Let B_g be a reduced OBDD for the function g.

The algorithm $\mathsf{apply}(\text{op}, B_f, B_g)$ describes how to (recursively) build a reduced OBDD for $(f \,\text{op}\, g)$. We assume that apply calls reduce before returning its final result.

Let f be a truth function over the set of variables V. Let x be a variable such that $x \in V$, and let $b = \{0,1\}$.

Finally, let $\tau(x)$ represent the truth value of x.

We write

$$f[b/x](\tau)$$

to mean f with the value of x restricted to b.

$$f[b/x](\tau) = f(\tau[x \to b])$$

$$\tau([x \to b]/y) = \begin{cases} \tau(y) & \text{if } y \neq x \\ b & \text{if } y = x \end{cases}$$

Example 4.2.7: Let f be

$$f = (x + y) \cdot z + xw$$

$$\begin{aligned} f[0/x] &= (0 + y) \cdot z + 0w && \text{replace } x \text{ with } 0 \\ &= y \cdot z + 0 \\ &= y \cdot z \end{aligned}$$

Claim 4.2.8: Let B be an OBDD, and let v be a non-terminal node of B. Then

$$f_{\mathsf{lo}(v)} = f_v[0/\mathsf{var}(v)]$$

Intuitively, $[0/\mathsf{var}(v)]$ forces use to take the "false" path for the variable in node v. Therefore it's the same as $\mathsf{lo}(v)$

PROOF: Let $x = \mathsf{var}(v)$.

$$\begin{aligned} f_v[0/x](\tau) &= f_v(\tau[x \to 0]) \\ &= f_{\mathsf{lo}(v)}(\tau[x \to 0]) \\ &= f_{\mathsf{lo}(v)}(\tau) \end{aligned}$$

Because the OBDD is ordered, if $x = \mathsf{var}(v)$, then x does not appear anywhere beneath v. Since B is ordered, $f_{\mathsf{lo}(v)}$ cannot depend on x.

Let B_f and B_g be OBDDs.
Let r_f be the initial node of B_f.
Let r_g be the initial node of B_g.

Our goal is to construct a new OBDD that represents $B_{f \,\mathrm{op}\, g}$

There are four cases to consider:

1. If r_f and r_g are terminal nodes labeled b_f and b_g, then return $b_f \,\mathrm{op}\, b_g$.

   In the remaining cases, at least one of r_f, r_g is a non-terminal.

2. Suppose r_f and r_g are non-terminal nodes such that $\mathsf{var}(r_f) = \mathsf{var}(r_g) = x$ (they represent the same variable). We create and return a new tree as follows:

   (a) A new node whose variable is x acts as the root.

   (b) Draw a dashed line to $\mathsf{apply}(\mathrm{op}, \mathsf{lo}(r_f), \mathsf{lo}(r_g))$

   (c) Draw a solid line to $\mathsf{apply}(\mathrm{op}, \mathsf{hi}(r_f), \mathsf{hi}(r_g))$

   (i.e. - apply is called recursively on each child)

3. r_f is a non-terminal x node and r_g is (1) a terminal node or (2) a non-terminal node y with $x < y$. x is ordered before y.

   Because B_f, B_g have compatible orderings, there are no x nodes in B_g. This case is handled by

   (a) Creating a new root node, and giving it the label x (x must come before y).

   (b) Draw a dashed line to $\mathsf{apply}(\mathrm{op}, \mathsf{lo}(r_f), r_g)$

   (c) Draw a solid line to $\mathsf{apply}(\mathrm{op}, \mathsf{hi}(r_f), r_g)$

4. r_g is a non-terminal x node and r_f is (1) a terminal, or (2) a non-terminal y node with $x < y$.

   This is handled symmetrically to case 3:

(a) Creating a new root node, and giving it the label x

(b) Draw a dashed line to $\mathsf{apply}(\mathrm{op}, r_f, \mathsf{lo}(r_g))$

(c) Draw a solid line to $\mathsf{apply}(\mathrm{op}, r_f, \mathsf{hi}(r_g))$

4.2.4 The Shannon Expansion

The Shannon expansion is named for Claude Shannon.

Let f be a truth function over V such that $x \in V$. Shannon's expansion is

$$f = \overline{x} \cdot f[0/x] + x \cdot f[1/x] \tag{4.2.3}$$

Here, x is a truth fact: $f(\tau) = \tau(x)$.

Also,

$$(\overline{x} \cdot f[0/x] + x \cdot f[1/x])(\tau) = \overline{\tau(x)} \cdot f(\tau[x \to 0]) + \tau(x) \cdot f(\tau[x \to 1])$$

The function apply is based on the Shannon expansion for $f \,\mathrm{op}\, g$:

$$f \,\mathrm{op}\, g = \overline{x_i} \cdot (f[0/x_i] \,\mathrm{op}\, g[0/x_i]) + x_i \cdot (f[1/x_i] \,\mathrm{op}\, g[1/x_i]) \tag{4.2.4}$$

Also,

$$(f \,\mathrm{op}\, g)[b/x] = f[b/x] \,\mathrm{op}\, g[b/x]$$

$$\begin{aligned}(f \,\mathrm{op}\, g)[b/x](\tau) &= (f \,\mathrm{op}\, g)(\tau[x \to b]) \\ &= f(\tau[x \to b]) \,\mathrm{op}\, g(\tau[x \to b]) \\ &= f[b/x](\tau) \,\mathrm{op}\, g[b/x](\tau) \\ &= (f[b/x] \,\mathrm{op}\, g[b/x])(\tau)\end{aligned}$$

4.3 Lecture – 4/30/2007

4.3.1 Some Notes Regarding hw3 and CTL*

Compare these two CTL formulas:

1. AF G p means "on every path, somewhere on that path, p starts holding forever.
2. AF AG p means "In the future, p is globally true on every path".

(2) is strictly stronger than (1). For (1), G p on a single future path will satisfy it, but that's not the case for (2).

The key to interpreting CTL* correctly is making the distinction between *states formulas* and *path formulas*. (See section 3.4.7, page 79).

Let's discect AF G p:

AF G p	A state formula
F G p	A path formula
G p	A path formula
p	A state formula (and by mututal recursion, also a path formula).

CTL* formulas that begin with E or A are *always* state formulas, and must be evaluated with respect to a particular state.

CTL* formulas that begin with F, G, X, etc (the LTL quantifiers) are *always* path formulas, and must be evaluated with respect to a particular path.

Path formulas always pertain to a specific path, unless explicitly quantified with A or E.

In AF G p, we really have A applied to F G p. Not AF applied to G p.

Operations on OBDDs

4.3.2 The apply operation

Last class, we looked at the operation

$$\mathsf{apply}(\mathrm{op}, B_f, B_g)$$

This operation applies the operation op to the OBDDs B_f and B_g. The result is an OBDD that represents the composite function $f \,\mathrm{op}\, g$.

Apply has many recursive sub-calls. During the recursion, it's possible that apply will need to solve the same sub-problem over and over.

We can make apply more efficient with dynamic programming techniques. Specifically, we'll want to use *memoization*: we compute a sub-formula and save the result. The next time we have to compute that sub-formula again, we re-use the results from the earlier computation.

4.3.3 The restrict operation

Let B_f be on OBDD for f.

$$\mathsf{restrict}(b, x, B_f)$$

computes a reduced OBDD for $f[b/x]$. (Here, $b = \{0, 1\}$, and x is a variable in f).

restrict works as follows:

- For $\mathsf{restrict}(0, x, B_f)$: For each node n labelled with x, remove n. Edges going into n now go to $\mathsf{lo}(n)$.
- For $\mathsf{restrict}(1, x, B_f)$: For each node n labelled with x, remove n. Edges going into n now go to $\mathsf{hi}(n)$.

4.3.4 The exists operation

Let f be a truth function over the set of variables V. Let $x \in V$. exists is

$$\exists x.f = f[0/x] + f[1/x] \tag{4.3.1}$$

$$\forall x.f = f[0/x] \cdot f[1/x] \qquad \text{Analogous } \forall \text{ operation} \tag{4.3.2}$$

$\exists x.f$ represents the relation of a constraint on some set of variables. $\exists x.f$ is true if f can be made true by setting x to 0 or to 1.

exists can be implemented in terms of apply:

$$\mathsf{apply}(+, \mathsf{restrict}(0, x, B_f), \mathsf{restrict}(1, x, B_f))$$

We can use the following trick to minimize the work that exists has to do: The OBDD B_f only changes in the subtree rooted at n labelled with x. Until we reach this node n, we simply copy the input OBDD to the output.

4.3.5 OBDDs and Symbolic Model Checking

Here, we're referring specifically to CTL model checking.

Our software models consist of sets of states (or more abstractly, of finite sets). We need a way to encode the various subsets of a set of states S.

In general, we will use boolean (bit) vectors.

- Each variable $s \in S$ will be assigned a bit v_i.
- A vector is $(v_1, \ldots, v_n)$ where $v_i \in \{0, 1\}$.
- A v_i (bit) is associated with a boolean variable.

Let T be a subset of states ($T \subseteq S$). Let f_T be the function $f_T \colon \{0,1\}^n \to \{0,1\}$

$$f_T = \begin{cases} 1 & \text{if } s \in T \\ 0 & \text{otherwise} \end{cases}$$

f_T is the *characteristic function* of T. It determines membership in T.

We need to pick an ordering of variables for our bitset. For example, if we have boolean variables x_1, x_2, x_3, we can represent these by the bits (v_1, v_2, v_3).

The translation between states and bit vectors is based on the Model labelling function $L \colon S \to \mathcal{P}(atoms)$.

So, we *identify a state by the labelled atoms in that state.* For example, if $x_1, x_2 \in L(s_1)$ and $x_3 \notin L(s_1)$, we represent s_1 by the bit vector `110`.

For these bit vectors

$$v_i = \begin{cases} 1 & \text{if } x_i \in L(s) \\ 0 & \text{otherwise} \end{cases}$$

Again, a state is uniquely identified by its label.

If our model has two or more states with identical labelling, we'll introduce new atomic variables to make the labelling unique. (In the worst case, we'd have to add $|S|$ new variables).

A state is prepresented as the OBDD of the boolean function $l_1 \cdot l_2 \cdot \ldots \cdot l_n$.

The set of states $\{s_1, \ldots, s_m\}$ is represented by

$$(l_{11} \cdot l_{12} \cdot \ldots \cdot l_{1n}) + (l_{21} \cdot l_{22} \cdot \ldots \cdot l_{2n}) + \ldots + (l_{m1} \cdot l_{m2} \cdot \ldots \cdot l_{mn})$$

There n variables, m states.

$l_i = x_i$ if $x_i \in L(S)$. $l_i = \overline{x_i}$ otherwise.

Figure 4.5 shows a model with three states and two variables. We'll use this as the basis for several examples that follow.

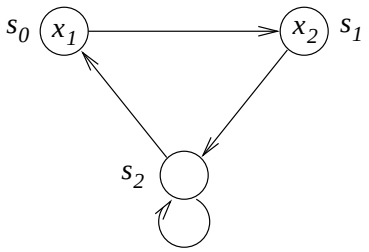

Figure 4.5: Model used for OBDD Binary Representation Examples

Table 4.2 shows how Figure 4.5 is represented.

States	**Boolean Value Representation**	**Boolean Function Representation**
$\emptyset$	0	0
$\{s_0\}$	(1,0)	$x_1 \cdot \overline{x_2}$
$\{s_1\}$	(0,1)	$\overline{x_1} \cdot x_2$
$\{s_2\}$	(0,0)	$\overline{x_1} \cdot \overline{x_2}$
$\{s_0, s_1\}$	(1,0), (0,1)	$x_1 \cdot \overline{x_2} + \overline{x_1} \cdot x_2$
$\{s_0, s_2\}$	(1,0), (0,0)	$x_1 \cdot \overline{x_2} + \overline{x_1} \cdot \overline{x_2}$
$\{s_1, s_2\}$	(0,1), (0,0)	$\overline{x_1} \cdot x_2 + \overline{x_1} \cdot \overline{x_2}$
$\{s_0, s_1, s_2\}$	(1,0), (0,1), (0,0)	$x_1 \cdot \overline{x_2} + \overline{x_1} \cdot x_2 + \overline{x_1} \cdot \overline{x_2}$

Table 4.2: State and atom representation of Figure 4.5

That takes care of state and atomic variables. Next, we need a way to represent transitions.

Transitions are a relation on $S \times S$. We represent an edge $s \rightarrow s'$ by a pair of bit vectors.

$$((v_1, v_2, \ldots, v_n), (v'_1, v'_2, \ldots, v'_n)) \tag{4.3.3}$$

In Equation (4.3.3), the first bit vector represents s, the second bit vector represents s', and the pair of bit vectors represents $s \rightarrow s'$.

In the database world, this is like joining a table to itself. One 'copy' of the table needs to be aliased. Here, the aliasing is done with primes.

There are actually two ways we can order the concatenation:

$((v_1, v_2, \ldots, v_n), (v'_1, v'_2, \ldots, v'_n))$	all v, then all v'
$(v_1, v'_1, v_2, v'_2, \ldots, v_n, v'_n)$	in bit order (alternating primes)

As an OBDD, the edge is represented by the boolean function

$$(l_1 \cdot l_2 \cdot \ldots \cdot l_n) \cdot (l'_1 \cdot l'_2 \cdot \ldots \cdot l'_n)$$

Table 4.3 shows the edge representation for the model in Figure 4.5. It's not the complete representation. The complete representation would have $2^4 = 16$ rows. We've only shown the rows where an edge exists (i.e. - where $\rightarrow = 1$).

x_1	x_2	x'_1	x'_2	$\rightarrow$	edges
1	0	0	1	1	$s_0 \rightarrow s_1$
0	1	0	0	1	$s_1 \rightarrow s_2$
0	0	0	0	1	$s_2 \rightarrow s_2$
0	0	1	0	1	$s_2 \rightarrow s_0$

Table 4.3: Edge transitions for figure 4.5

4.3.6 Implementing $\mathsf{pre}_\forall$ and $\mathsf{pre}_\exists$

To implement $\mathsf{pre}_\forall$ and $\mathsf{pre}_\exists$ with this OBDD representation:

$$\mathsf{pre}_\forall(X) = S - \mathsf{pre}_\exists(X)$$

$\mathsf{pre}_\exists(X)$ is

$$\mathsf{exists}((x'_1, \ldots x'_n), \mathsf{apply}(\cdot, B_\rightarrow, B_{X'}))$$

$B_\rightarrow$ is the transition relation.

4.4 Lecture – 5/2/2007

4.4.1 OBDDs for Transition Relations

Transition relations are represented by the concatenation of a pair of bit vectors:

$$((v_1, \ldots, v_n), (v'_1, \ldots, v'_n))$$

This can be represented as a truth table with 2^{2n} rows, but the truth table representation is too large to be practical.

SMV allows one to specifiy how variables change from one state to the next (ie. - `next(var)`). We can represent this sort of thing with a formula

$$x'_i \leftrightarrow f_i$$

x'_i is the next value of x_i. f_i is a boolean expression of the remaining variables.

$x'_i \leftrightarrow f_i$ handles a single variable. The overall transition relation is

$$\prod_{i=1}^{n} x'_i \leftrightarrow f_i$$

This approach is useful (for example) representing logic circuits.

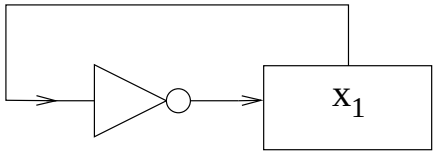

Figure 4.6: Simple logic circuit (an inverter)

As a function, Figure 4.6 is $x'_i = \overline{x_i}$. (or `next(xi) = !x1`).

There are two main categories of circuit simulations:

1. **synchronous**. All circuit elements update in a single clock tick.
2. **asynchronous**. Not all circuit element updates in a single clock tick.

There are two categories of asynchronous circuit modes.

1. **simultaneous model**. Some number of circuit elements update during a single clock tick, but not all of them.
2. **interleaving model**. Exactly 1 element is updated during a clock tick (round robin?)

As boolean equations, these types of transitions can be represented by

$$\prod_{i=1}^{n} (x'_i \leftrightarrow f_i) + (x'_i \leftrightarrow x_i) \qquad \text{simultaneous model} \qquad (4.4.1)$$

$$\sum_{i=1}^{n} \left((x'_i \leftrightarrow f_i) \cdot \prod_{i \neq j} x'_j = x_j \right) \qquad \text{interleaving model} \qquad (4.4.2)$$

In (4.4.1), $(x'_i \leftrightarrow f_i)$ represents a variable that changes; $(x'_i \leftrightarrow x_i)$ represents a variable that does not change.

In (4.4.2), $(x'_i \leftrightarrow f_i)$ is the *single* variable that change. The product expression requires that all other variables remain the same.

4.4.2 Relational Mu-Calculus

(Also known as μ-calculus)

μ-calculus is a way to talk about least and greatest fixed points.

There are two types of variables

$$v ::= x \mid Z$$

μ-calculus as a BNF:

$$f ::= 0 \mid 1 \mid v \mid \overline{f} \mid f_1 + f_2 \mid f_1 \cdot f_2 \mid f_1 \oplus f_2 \mid$$
$$\exists x.f \mid \forall x.f \mid \mu Z.f \mid \nu Z.f \mid f[\hat{x} := \hat{x}']$$

The first line is very similar to propositional logic. The second line will require some explanation.

The precedence rules for μ-calculus are:

$^- \; [\hat{x} := \hat{x}']$	highest precedence
$\exists x \; \exists y$	
$\mu Z \; \nu Z$	
$\cdot$	
$+ \; \oplus$	lowest precedence

$\exists x.f$ and $\forall x.f$ are quantified expressions. (for example, we could represent $\mathsf{pre}_\exists(X)$ as $\exists \hat{x'} B^{\rightarrow} \cdot B_{X'}$.

$\mu Z.f$ and $\nu Z.f$ require that any occurrences of Z in f occur within an even number of negations. The even number of negations makes f monotonic. These functions represent least, and respectively greatest, fixed points.

Definition 4.4.1 (valuation)**:** A *valuation* ρ of f is an assignment of values $\{0, 1\}$ to all variables v in f.

Definition 4.4.2: Let ρ be a valuation and let v be a variable. We write $\rho(v)$ for the value of v assigned by ρ.

We define $\rho[v \mapsto 0]$ to be the updated valuation that assigns 0 to v, while leaving the values of all other variables unchanged.

$\rho[v \mapsto 1]$ assigns 1 to v while leaving the values of all other variables unchanged.

$\rho[\hat{x} := \hat{x}']$ works similarly, but it deals with sets of variables. Each member of $x_i \in \hat{x}$, it assigns the value $\rho(x'_i)$ to x_i.

Example 4.4.3: Suppose $\rho(x'_1) = 0$ and $\rho(x'_2) = 1$.

$$\begin{aligned}
&\rho \vDash (x_1 \cdot \overline{x_2})[\hat{x} := \hat{x}'] \\
=&\rho[x_1 \mapsto 0][x_2 \mapsto 1] \vDash x_1 \cdot \overline{x_2} \\
=&\rho[x_1 \mapsto 0][x_2 \mapsto 1] \vDash 0 \cdot 0 \\
\therefore&\rho[x_1 \mapsto 0][x_2 \mapsto 1] \nvDash x_1 \cdot \overline{x_2}
\end{aligned}$$

In this case $\rho \vDash f$, but we can see how the substitution works.

We write $\rho \vDash f$ to say that the valuation ρ satisifies the formula f.

Definition 4.4.4 (satisfaction relation $\rho \vDash f$)**:** The satisfaction relation $\rho \vDash f$ is defined by structural induction

- $\rho \nvDash 0$
- $\rho \vDash 1$
- $\rho \vDash v$ IFF $\rho(v) = 1$
- $\rho \vDash \overline{f}$ IFF $\rho \nvDash f$
- $\rho \vDash f + g$ IFF $\rho \vDash f$, or $\rho \vDash g$
- $\rho \vDash f \cdot g$ IFF $\rho \vDash f$, and $\rho \vDash g$
- $\rho \vDash f \oplus g$ IFF $\rho \vDash (f \cdot \overline{g} + \overline{f} \cdot g)$
- $\rho \vDash \exists x.f$ iff $\rho[x \mapsto 0] \vDash f$, or $\rho[x \mapsto 1] \vDash f$.

  (i.e. - if either $x = 0$ or $x = 1$ makes f true).
- $\rho \vDash \forall x.f$ iff $\rho[x \mapsto 0] \vDash f$, and $\rho[x \mapsto 1] \vDash f$.

  (i.e. - if both $x = 0$ and $x = 1$ make f true).
- $\rho \vDash f[\hat{x} := \hat{x}']$ iff $\rho[\hat{x} := \hat{x}'] \vDash f$

Least Fixed Points

$\mu Z.f$ is the least fixed point of f. This defines a boolean function on $x_1, \ldots, x_n, Z$. When evaluating this function, Z will be replaced with another valuation $x_1, \ldots, x_n$.

Replacing Z by a function of $x_1, \ldots, x_n$ gives another function $f(g)$.

We want $\mu Z.f$ to be *monotonic*, so that it represents a least fixed point of f.

Suppose $f = x_1 + Z$. $\mu Z.f(x_1 + Z)$ will act like set union.

$$f(t) = t \cup \{\rho \mid \rho(x_i) = 1\} \qquad t \text{ is a set of truth assignments}$$

On page 99, we started least fixed point computations with $\emptyset$. Here, we'll start them with 0, where 0 is the "empty valuation" (all zeros).

Greatest Fixed Points

$\nu Z.f$ represents a greatest fixed point computation.

Suppose $f = x_1 \cdot Z$. $\nu Z.f(x_1 \cdot Z)$ will act like set intersection.

$$f(t) = t \cap \{\rho \mid \rho(x_i) = 1\}$$

When dealing with sets, we started fixed point computations with S. Here, we'll start with '1' (the truth valuation consisting of all 1's). This simplifies f to

$$f(t) = \{\rho \mid \rho(x_i) = 1\}$$

4.4.3 Other Sources of OBDD information

H&R cite this paper as giving a good overview on OBDDs:
`http://doi.acm.org/10.1145/136035.136043`

4.5 Lecture – 5/7/2007

4.5.1 Mu-Calculus

Recall that mu-calculus is

$$
\begin{aligned}
&v := x \mid Z \\
&f := 0 \mid 1 \mid \overline{f} \mid f + f \mid f \cdot f \mid f \oplus f \mid \\
&\qquad \exists x.f \mid \forall x.f \mid \mu Z.f \mid \nu Z.f \mid f[\hat{x} := \hat{x}']
\end{aligned}
$$

mu-calculus was designed to allow easy representation of greatest and least fixed points.

ρ represents a set of satisfying assignments for f: $\rho \vDash f$

Recall that for $\mu Z.f$, $\nu Z.f$ we have Z replaced by a boolean function to obtain a new f. $\mu Z.f$ and $\nu Z.f$ carry the requirement that f must be *formally monotonic* in Z. (Formally monotonic means that Z must contain an even number of negations. We'll see an example of why this is important later on).

Z stands for a boolean function – a set S of truth assignments (i.e. - each $s \in S$ makes Z true).

If, $f = x_1 \cdot Z$, then the assignments that make f true can be given as $\{\rho \in S \mid \rho(x_1) = 1\}$. All members of S already make Z true; the ones that make f true are the ones that make both Z and x_1 true.

$\mu Z.f$ represents a *least* fixed point. We'll start the fixed point computation with $Z = 0$ (or $Z = \emptyset$. Both representations work, one represents Z as a formula, the other as a set).

$\nu Z.f$ represents a *greatest* fixed point. We'll start this fixed point computation with $Z = 1$.

Example 4.5.1: Suppose our formula is $f = x_1 \cdot Z$. We'll have

$$
\begin{aligned}
&\mu Z.f = \emptyset && \text{or } 0 \\
&\nu Z.f = \{\rho \mid \rho(x_1) = 1\}
\end{aligned}
$$

Least Fixed points, More formally

More formally, the least fixed point computation is

$$\mu_0 Z.f = 0 \tag{4.5.1}$$

$$\mu_{m+1} Z.f = f[\mu_m Z.f / Z] \tag{4.5.2}$$

In (4.5.2), the notation $[\mu_m Z.f/Z]$ means roughly the same thing as the notation we used for first order logic. Replace all *free* occurrences of Z in f with $\mu_m Z.f$.

We say that

$$\rho \vDash \mu Z.f \text{ IFF } \rho \vDash \mu_m Z.f \text{ for some } m \geq 0 \tag{4.5.3}$$

Greatest Fixed Points, More Formally

The greatest fixed point computation is

$$\nu_0 Z.f = 1 \tag{4.5.4}$$

$$\nu_{m+1} Z.f = f[\nu_m Z.f / Z] \tag{4.5.5}$$

We say that

$$\rho \vDash \nu Z.f \text{ IFF } \rho \vDash \nu_m Z.f \text{ for ALL } m \geq 0 \tag{4.5.6}$$

Note the difference between (4.5.3) and (4.5.6). The former is satisifed by *some* m. The latter requires *all* m.

Example 4.5.2: Suppose we have the equation $f = x_1 \cdot Z$.

$$\begin{aligned}
&\mu_0 Z.f = 0 \\
&\mu_1 Z.f = x_1 \cdot 0 \\
&\mu_2 Z.f = x_1 \cdot x_1 \cdot 0 \\
&\quad\vdots \\
&\quad = 0
\end{aligned}$$

$$\begin{aligned}
&\nu_0 Z.f = 1 \\
&\nu_1 Z.f = x_1 \cdot 1 \\
&\nu_2 Z.f = x_1 \cdot x_1 \cdot 1 \\
&\quad\vdots \\
&\quad = x_1
\end{aligned}$$

In these examples, note that the formulas do not stabilize, but their *meaning* does.

Example 4.5.3 (The need for Formal Monotonicity)**:** Suppose our formula was $f = \neg Z$. f has an odd number of negations, so f is not monotonic. If we try to compute a fixed point, we have

$$\begin{aligned}
&\mu_0 Z.f = 0 \\
&\mu_1 Z.f = \neg 0 \\
&\mu_2 Z.f = \neg\neg 0
\end{aligned}$$

In this example, the value of f doesn't stabilize – it oscillates. This function has no fixed points.

4.5.2 Model Checking with Mu-Calculus

Suppose we are given $\mathcal{M} = (S, \rightarrow, L)$, and for each CTL formula ϕ, we want a mu-calculus formula f^ϕ that represents $\{s \mid \mathcal{M}, s \vDash \phi\}$.

The states will be represented with our bitstring formula. L will determine the bitstrings for each states $s \in S$.

$f^{\rightarrow}$ will represent $\rightarrow$ as a boolean function.

We're interested in determining whether

$$\mathcal{M}, I \vDash \phi$$

Where I is the set of initial states. For all $s \in I$, does $\mathcal{M}, s \vDash \phi$? This will happen IFF

$$f^I \cdot \overline{f^\phi}$$

is unsatisifiable. (f^I is the boolean formula representation of the initial states).

We can define f^ϕ inductively:

- $f^x = x$

- $f^{\perp} = 0$
- $f^{\neg\phi} = \overline{f^{\phi}}$
- $f^{\phi \wedge \psi} = f^{\phi} \wedge f^{\psi}$
- $f^{\phi \vee \psi} = f^{\phi} \vee f^{\psi}$

These are the simple cases. Formulas with path quantifiers will require a little more explanation.

$$f^{\mathrm{EX}\,\phi} = \exists \hat{x}'.(f^{\rightarrow} \cdot f^{\phi}[\hat{x} := \hat{x}']) \tag{4.5.7}$$

Recall that $\mathcal{M}, s \vDash \mathrm{EX}\,\phi$ IFF there is an s' where $s \rightarrow s'$ and $s' \vDash \phi$. Equation (4.5.7) reflects this: $\hat{x}$ is the current state, $\hat{x}'$ is the successor state, and $f^{\rightarrow}$ means that there is a transition $(\hat{x}, \hat{x}')$.

The clause for EF is based on the equivalence

$$\mathrm{EF}\,\phi \equiv \phi \vee \mathrm{EX}\,\mathrm{EF}\,\phi$$

$f^{\mathrm{EF}\,\phi}$ is derived from

$$\begin{aligned} f^{\mathrm{EF}\,\phi} &= f^{\phi} + f^{\mathrm{EX}\,\mathrm{EF}\,\phi} \\ &= f^{\phi} + \exists \hat{x}'.(f^{\rightarrow} \cdot f^{\mathrm{EF}\,\phi}[\hat{x} := \hat{x}']) \\ &= \mu Z.(f^{\phi} + \exists \hat{x}'.(f^{\rightarrow} \cdot Z[\hat{x} := \hat{x}'])) \end{aligned}$$

The last line comes about because $\mathrm{EF}\,\phi$ requires a least fixed point computation.

$\mathrm{EF}\,\phi \equiv \mathrm{E}[\top\,\mathrm{U}\,\phi]$. We can use this fact to generalize a formula for $f^{\mathrm{E}[\phi\,\mathrm{U}\,\psi]}$. Note that

$$\mathrm{E}[\phi\,\mathrm{U}\,\psi] \equiv \psi \vee (\phi \wedge \mathrm{EX}\,\mathrm{E}[\phi\,\mathrm{U}\,\psi])$$

So

$$\begin{aligned} f^{\mathrm{E}[\phi\,\mathrm{U}\,\psi]} &= f^{\psi} + f^{\phi} \cdot f^{\mathrm{EX}\,\mathrm{E}[\phi\,\mathrm{U}\,\psi]} \\ &= f^{\psi} + f^{\phi} \cdot \exists \hat{x}'.(f^{\rightarrow} \cdot f^{\mathrm{E}[\phi\,\mathrm{U}\,\psi]}[\hat{x} := \hat{x}']) \\ &= \mu Z.(f^{\psi} + f^{\phi} \cdot \exists \hat{x}'.(f^{\rightarrow} \cdot Z[\hat{x} := \hat{x}'])) \end{aligned}$$

The definiton of $f^{\mathrm{AF}\,\phi}$ is similar to the one for $f^{\mathrm{EF}\,\phi}$, except that the existential quantification is replaced by universal quantification.

$$f^{\mathrm{AF}\,\phi} = \mu Z.(f^{\phi} + \forall \hat{x}'.(\overline{f^{\rightarrow}} + Z[\hat{x} := \hat{x}']))$$

Finally, we'll look at $f^{\mathrm{EG}\,\phi}$. Recall that

$$\mathrm{EG}\,\phi \equiv \phi \wedge \mathrm{EX}\,\mathrm{EG}\,\phi$$

So

$$f^{\mathrm{EG}\,\phi} = \nu Z.(f^{\phi} \cdot \exists \hat{x}'.(f^{\rightarrow} \cdot Z[\hat{x} := \hat{x}']))$$

Tip: In a recursive equivalence, $\wedge$ usually means we'll need a greatest fixed point (like $\mathrm{EG}\,\phi$); $\vee$ usually means we'll need a least fixed point (like $\mathrm{EF}\,\phi$).

4.5.3 Symbolic Model Checking with Fairness

In the world of CTL model checking, we want to consider paths where the fairness constraints occur infinitely often.

LTL can express fairness constraints directly, so there's no special handling for fairness constraints in LTL.

Given fairness constraints $C = \{\phi_1, \ldots, \phi_k\}$, we'd like to define mu-calculus formulas $f^{\mathrm{E_C X}}$, $f^{\mathrm{E_C G}}$, and $f^{\mathrm{E_C U}}$. EX, EG, and EU are an adequate set of connectives for CTL, so we can get by with only this set of definitions.

First, we define fairness as a boolean formula

$$\mathsf{fair} = f^{\mathrm{E_C G}\,\top} \tag{4.5.8}$$

The idea: (4.5.8) evaluates to 1 IFF there is a fair path with respect to C that begins in the state s.

The fair version of $\mathrm{E_C X}$ can be derived directly from f^{EX}:

$$f^{\mathrm{E_C X}\,\phi} = \exists \hat{x}'.(f^{\rightarrow} \cdot (f^{\phi} \cdot \mathsf{fair})[\hat{x} := \hat{x}']) \tag{4.5.9}$$

A fair version of EU can be derived similarly:

$$f^{\mathrm{E_C}[\phi_1\,\mathrm{U}\,\phi_2]} = \mu Z.(f^{\phi_2} \cdot \mathsf{fair} + f^{\phi_1} \cdot \exists \hat{x}'.(f^{\rightarrow} \cdot Z[\hat{x} := \hat{x}'])) \tag{4.5.10}$$

Now, we just have to define $f^{\mathrm{E_C G}}$ (which is used to define fair). We employ a pair of helper functions to do this.

$$\mathsf{checkEX}(f) = \exists \hat{x}'.(f^{\rightarrow} \cdot f[\hat{x} := \hat{x}']) \tag{4.5.11}$$

Equation (4.5.11) checks the EX condition. With this definition, we could rewrite $f^{\mathrm{E_C X}\,\phi}$ as

$$f^{\mathrm{E_C X}\,\phi} = \mathsf{checkEX}(f^{\phi} \cdot \mathsf{fair})$$

We'll also define a helper function for EU

$$\mathsf{checkEU}(f, g) = \mu Y.g + (f \cdot \mathsf{checkEX}(Y)) \tag{4.5.12}$$

Finally, we code $f^{\mathrm{E_C G}}$:

$$f^{\mathrm{E_C G}\,\phi} = \nu Z.f^{\phi} \cdot \prod_{i=1}^{k} \mathsf{checkEX}(\mathsf{checkEU}(f^{\phi}, Z \cdot f^{\psi_i}) \cdot \mathsf{fair}) \tag{4.5.13}$$

(ψ_i are the fairness constraints).

Note that (4.5.13) has a least fixed point (checkEU), in the body of a greatest fixed point (checkEX). Computationally, it's a little expensive.

We can also express

$$\mathrm{E_C G}\,\phi \equiv \phi \bigwedge_{i=1}^{k} \mathrm{EX}\,\mathrm{E}[\phi\,\mathrm{U}(\psi_i \wedge \mathrm{E_C G}\,\phi)]$$

If $\mathrm{E_C G}\,\phi$ is true, then $s \vDash \phi$, and for each i, there is (a) a fairness constraint that is true and (b) $\mathrm{E_C G}\,\phi$ is true somewhere later on that path.

If $\phi \bigwedge_{i=1}^{k} \mathrm{EX}\,\mathrm{E}[\phi\,\mathrm{U}(\psi_i \wedge \mathrm{E_C G}\,\phi)]$ is true, then there is a path with ϕ true until ψ_1 is true. From the point where ψ_i is true, we can get to ψ_2, etc.

Note that fairness constraints *nest fixpoints.*

4.5.4 Logistics

- In the next few classes, we'll cover material from the beginning of Chapter four.
- Our final will be a take-home exam. It will be distributed later this week, due during finals week.

Part 5

Program Verification

This material is covered in Chapter 4 of H&R

5.1 Lecture – 5/9/2007

Our look at program verification will focus on sequential programs that run on a single processor (no concurrency). These programs can be characterized by having an infinite state space.

The classical work in program verification was done by Floyd and Hoare. Floyd's approach used inductive assertion. Hoare's approach used Hoare Logic. We'll focus on Hoare's approach.

5.1.1 Classification of Program Verification Techniques

We'll study techniques that are

- **Proof-based**. (Not the exhaustive state checking used for model verification).
- **Semi-automatic**.
- **Property Based**. We will verify certain aspects of program behavior, but not the full program behavior.
- **Application Domain**. These techniques will apply to sequential transformational programs (not the reactive systems we studied with model checking).

5.1.2 A Framework For Program Verification

- We can start with an informal specification R. R will be converted into a formal specification in the form of a formula ϕ_R. ϕ_R is represented using some form of symbolic logic.
- We write a program P to implement R.
- We prove that P satisfies ϕ_R.

5.1.3 A Simple Programming Language

To discuss verification techniques, we'll introduce a simple programming language. This language uses (1) integer expressions, (2) boolean expressions, and (3) control structures.

Although simple, it's really sufficient to represent any computable function (although it won't be very convenient to do so).

Integer Expressions

$$E ::= n \mid x \mid (-E) \mid (E + E) \mid (E - E) \mid (E \times E) \tag{5.1.1}$$

Above, n is an integer ($n \in \mathbb{Z}$). x is any program variable.

Boolean Expressions

$$B ::= \texttt{true} \mid \texttt{false} \mid (!B) \mid (B \& B) \mid (B || B) \mid (E < E) \tag{5.1.2}$$

$(E < E)$ is our only native integer test. For notational convenience, we'll use `==` and `!=`, with the knowledge that these tests could be implemented as

$!(E_1 < E_2)$ & $!(E_2 < E_1)$ and
$(E_1 < E_2)$ || $(E_2 < E_1)$

respectively.

Control Structures

Our control structures are

$$C ::= x = E \mid C; C \mid \texttt{if } B \{C_1\} \texttt{ else } \{C_2\} \mid \texttt{while } B \{C\} \tag{5.1.3}$$

`while` is our only looping construct.

Here's an example of a program (which computes $x!$).

Example 5.1.1 (`Fac1`)**:**

```
y = 1;
z = 0;
while (z != x) {
  z = z + 1;
  y = y * z;
}
```

There's no return statement. That's okay. We're really interested in the properties that variables have before and after execution.

5.1.4 Hoare Triples

All programs have a starting state (before execution), and an ending state (after execution). Here, *state* can be thought of as a vector of variables.

We will make assertions using Hoare triples. Hoare triples have the following form:

$$(\!|\phi|\!)\ P\ (\!|\psi|\!) \tag{5.1.4}$$

In (5.1.4),

- ϕ is *precondition* (in first-order logic)

- ψ is a *postcondition* (in first-order logic)
- P is a program.

This means: if program P is run from a starting state that satisfies ϕ, then the state resulting from P's execution will satisfy ψ.

The formulas ϕ and ψ will primarily use $-$ (unary), $-$ (binary), $+$, $\cdot$, $<$, $=$. However, we'll occasionally use other "well-known" mathematical symbols too.

ϕ, ψ are allowed to use quantifiers, provided that quantifiers are not bound to any variable that occurs in P.

Example 5.1.2:

$$(\!|x > 0|\!)\ P\ (\!|y \cdot y < x|\!)$$

Says "if x is positive, then y^2 will be less than x when the program completes".

Our formulas will be evaluated in the context of a model $\mathcal{M}$, whose universe is $\mathbb{Z}$, the set of integers.

We also make use of a *lookup table* l. $l(x)$ gives us the value of the variable x. l also establishes the *state* of the program.

To say "l satisfies ϕ", we write $\mathcal{M} \vDash_l \phi$ (just like we did with first-order logic).

Example 5.1.3:

Using the `Fac1` program in example 5.1.1, we can write a specification:

$$(\!|x \geq 0|\!)\ \texttt{Fac1}\ (\!|y = x!|\!)$$

Here's another example program

Example 5.1.4 (`Prog51a`)**:**

```
# Given x, find a y whose square is not
# greater than x
y = 0;
while y * y <= x {
  y = y + 1;
}
y = y - 1;
```

Example 5.1.4 satisfies the specification

$$(\!|x \geq 0|\!)\ \texttt{Prog51a}\ (\!|y \cdot y \leq x|\!)$$

But this isn't a terribly good specification. For example, the specification is also satisfied by the program:

```
y = 0
```

A better specification for `Prog51a` would be

$$(\!|x \geq 0|\!)\ \texttt{Prog51a}\ (\!|y \cdot y \leq x \wedge (y+1)(y+1) > x|\!)$$

The moral of the story: your postcondition must be strong enough to capture the behavior that you intended.

Types of Correctness

Definition 5.1.5 (Partial Correctness)**:** $(\!|\phi|\!)\, P\, (\!|\psi|\!)$ is correct under *partial correctness* if, whenever P is started in a state where ϕ is true and P halts, then ψ is true in the final state of P.

We write this $\vdash_{\mathsf{par}} (\!|\phi|\!)\, P\, (\!|\psi|\!)$.

Partial Correctness is a weak concept: a program that never terminates satisfies *any* specification under partial correctness. For example:

$(\!|\phi|\!)$ `while true { x = x }` $(\!|\psi|\!)$

is true for any ϕ, ψ.

Definition 5.1.6 (Total Correctness)**:** $(\!|\phi|\!)\, P\, (\!|\psi|\!)$ is correct under *total correctness* if, whenever P is started in a state that makes ϕ true, P terminates, and ψ is true in the final state of P.

We write this $\vdash_{\mathsf{tot}} (\!|\phi|\!)\, P\, (\!|\psi|\!)$

This is a stronger concept. Given our earlier infinite loop program P, P is not totally correct if it starts with ϕ true. However, if P is started with ϕ false, then P *is* totally correct.

The traditional way to prove total correctness is a two-step process

1. Prove that P is partially correct
2. Prove that P terminates.

We can think of total correctness as

total correctness = partial correctness + termination

Example 5.1.7: Some examples of total and partial correctness, using our `Fac1` program (example 5.1.1).

$\vDash_{\mathsf{tot}} (\!|x > 0|\!)$ `Fac1` $(\!|y = x!|\!)$	holds
$\nvDash_{\mathsf{tot}} (\!|\mathtt{true}|\!)$ `Fac1` $(\!|y = x!|\!)$	Doesn't hold when $x < 0$ (wrong answer)
$\vDash_{\mathsf{par}} (\!|\mathtt{true}|\!)$ `Fac1` $(\!|y = x!|\!)$	Holds - the program never halts

5.1.5 Soundness and Completeness

As with first-order logic, Hoare logic has notions of soundness and completeness.

$\vdash_{\mathsf{par}}, \vdash_{\mathsf{tot}}$	Soundness, a syntactic notion
$\vDash_{\mathsf{par}}, \vDash_{\mathsf{tot}}$	Completeness, a semantic notion

Although we won't prove it, Hoare logic is sound and complete:

$\vdash_{\mathsf{par}} \ldots \Rightarrow \vDash_{\mathsf{par}}$
$\vdash_{\mathsf{tot}} \ldots \Rightarrow \vDash_{\mathsf{tot}}$

$\vDash_{\mathsf{par}} \ldots \Rightarrow \vdash_{\mathsf{par}}$
$\vDash_{\mathsf{tot}} \ldots \Rightarrow \vdash_{\mathsf{tot}}$

5.1.6 Logical Variables

Let's consider another program:

Example 5.1.8 (`Fac2`)**:**

```
y = 1;
while x != 0 {
  y = y * x;
  x = x + 1;
}
```

This program also computes factorial. Note that it modifies ("consumes") the input variable x. Because x is consumed, the postcondition we'd want, $(\!|y = x!|\!)$, doesn't hold.

We'll handle this by adding *logical variables*. Logical variables are used to preserve program input, enabling one to use those values when reasoning about the program output. Example:

$$(\!|x \geq 0 \wedge x = x_0|\!)\ \texttt{Fac2}\ (\!|y = x_0!|\!)$$

Above, x_0 is a logical variable. This is *not* an assignment. The clause $x = x_0$ means "we start the program from a state where the logical variable x_0 has the same value as the program variable x".

By using logical variables, we can make the specification work *without* changing the program.

Only logical variables can be quantified.

5.1.7 A Proof Calculus for Partial Correctness

Below are the syntactic rules for Hoare Logic.

$$\frac{(\!|\phi|\!)\ C_1\ (\!|\eta|\!) \qquad (\!|\eta|\!)\ C_2\ (\!|\psi|\!)}{(\!|\phi|\!)\ C_1; C_2\ (\!|\psi|\!)} \quad \text{Composition}$$

$$\frac{}{(\!|\psi[E/x]|\!)\ x = E\ (\!|\psi|\!)} \quad \text{Assignment}$$

$$\frac{(\!|\phi \wedge B|\!)\ C_1\ (\!|\psi|\!) \qquad (\!|\phi \wedge \neg B|\!)\ C_2\ (\!|\psi|\!)}{(\!|\phi|\!)\ \texttt{if}\ B\{C_1\}\ \texttt{else}\ \{C_2\}\ (\!|\psi|\!)} \quad \text{If-statement}$$

$$\frac{(\!|\psi \wedge B|\!)\ C\ (\!|\psi|\!)}{(\!|\psi|\!)\ \texttt{while}\ B\{C\}\ (\!|\psi \wedge \neg B|\!)} \quad \text{Partial-While}$$

$$\frac{\vdash_{\mathsf{AR}} \phi' \rightarrow \phi \quad (\!|\phi|\!)\ C\ (\!|\psi|\!) \quad \vdash_{\mathsf{AR}} \psi \rightarrow \psi'}{(\!|\phi'|\!)\ C\ (\!|\psi'|\!)} \quad \text{Implied}$$

A few of these rules require some explanation.

Composition η acts like an intermediate state, called a *mid-condition*. Think of η as the place where C_1 ends and C_2 picks up.

Assignment Assignment is an axiom in our system. It's written the opposite of what one might expect. Suppose the assignment rule were

$$\frac{}{(\!|\phi|\!)\ x = E\ (\!|\phi[E/x]|\!)}$$

WRONG definition for assignment

If, so - we could derive

$$(\!|x = 2|\!)\ x = y\ (\!|y = 2|\!)$$

That's obviously wrong: if we never assign to y, we can't make any statements about it's value.

Using the correct definition:

$$(\!|x[y/x] = 2|\!)\ x = y\ (\!|y = 2|\!)$$

it works – we change x to y in the postcondition, then $y = 2$ will hold in the output.

If-Statement Notice that we have two cases to consider: those where B holds and those where $\neg B$ holds. Either way, we expect to satisfy the same postconditions.

Partial-While We start by assuming B is true (if not, C never executes). We also require $\neg B$ when the loop terminates.

This construct only holds under partial correctness.

Implies AR stands for "arithmetic". This rule allows us to draw conclusions by mathematically manipulating the precondition and postcondition.

5.2 Lecture – 5/14/2007

5.2.1 While Rules

Our rule for proving while statements:

$$\frac{(\!|\psi \wedge B|\!)\ \mathtt{C}\ (\!|\psi|\!)}{(\!|\psi|\!)\ \mathtt{while}\ \ B\{C\}\ (\!|\psi \wedge \neg B|\!)}$$

$\neg B$ captures the exit condition of the while loop. ψ is known as a *loop invariant*.

5.2.2 Proofs With Hoare Triples

When writing proofs, we deal with one statement at a time, writing them

C_1
C_2
$\vdots$
C_n

Where no C_i is a compound statement.

Suppose we want to prove $\vdash_{\mathsf{par}} (\!|\phi_0|\!)\ P\ (\!|\phi_n|\!)$. Using composition rules, if we find $\phi_1 \ldots \phi_{n-1}$ and prove

$$\vdash_{\mathsf{par}} (\!|\phi_i|\!)\ C_{i+1}\ (\!|\phi_{i+1}|\!) \qquad \text{for } 1 \leq i < n-1$$

then $\vdash_{\mathsf{par}} (\!|\phi_0|\!)\ P\ (\!|\phi_n|\!)$ is true.

The general shape of our proofs will be

	$(\!\|\phi_0\|\!)$	
C_1		
	$(\!\|\phi_1\|\!)$	justification
C_2		
$\vdots$		
	$(\!\|\phi_{n-1}\|\!)$	justification
C_n		
	$(\!\|\phi_n\|\!)$	justification

The assertions will be mixed directly with the program code.

When doing these proofs, it's usually easier to to work from ϕ_n to ϕ_0, deriving the mid-conditions backwards.

By working backwards, we may not end up with ϕ_0 precisely. That's okay, as long as we find a condition that can be implied by ϕ_0.

Definition 5.2.1 (Weakest Precondition)**:** The process of obtaining ϕ_i from C_{i+1} is called computing the *weakest precondition* of C_{i+1}, given the postcondition ϕ_{i+1}. We are looking for the logically weakest formula whose truth at the beginning of the execution of C_{i+1} is enough to guarantee ϕ_{i+1}.

Or,

The weakest precondition of C, ψ, is a formula ϕ such that $\vDash_{\mathsf{par}} (\!|\phi'|\!)\ C\ (\!|\psi|\!)$ IFF $\vdash_{\mathsf{AR}} \phi' \rightarrow \phi$. □

Given $\phi' \to \phi$, ϕ is weaker. We want ϕ to be the weakest formula that ensures $(\!|\phi|\!)\, P\, (\!|\psi|\!)$.

This is really our Implied rule:

$$\frac{\vdash \phi' \to \phi,\ (\!|\phi|\!)\, C\, (\!|\psi|\!),\ \vdash \psi \to \psi'}{(\!|\phi'|\!)\, C\, (\!|\psi'|\!)}$$

5.2.3 The Assignment Rule

Recall the assignment rule (an axiom)

$$(\!|\psi[E/x]|\!)\ x = E\ (\!|\psi|\!)$$

Example 5.2.2: We show that $\vdash_{\mathsf{par}} (\!|y = 5|\!)$ `x = y + 1` $(\!|x = 6|\!)$ is valid.

$(\!\|y = 5\|\!)$	
$(\!\|y + 1 = 6\|\!)$	Implied
`x = y + 1`	
$(\!\|x = 6\|\!)$	Assignment

Note how we used the assignment rule to get from $x = 6$ to $y + 1 = 6$. $(x = 6)[y + 1/x] = y + 1 = 6$.

Also note the use of Implied. $(y = 5) \to (y + 1 = 6)$.

If our proof takes the shape

$$\frac{\phi_1 \to \phi,\ (\!|\phi|\!)\, C\, (\!|\psi|\!)}{(\!|\phi_1|\!)\, C\, (\!|\psi|\!)} \tag{5.2.1}$$

This is still the implied rule. If we wrote the last step explicitly, it would be $\psi \to \psi$.

Example 5.2.3 (Swapping variables)**:** Let us define the program `swap`:

```
t = x;
x = y;
y = t;
```

We want to prove $(\!|x = x_0 \wedge y = y_0|\!)$ `swap` $(\!|x = y_0 \wedge y = x_0|\!)$

$(\!\|x = x_0 \wedge y = y_0\|\!)$	
$(\!\|y = y_0 \wedge x = x_0\|\!)$	Implication
`t = x`	
$(\!\|y = y_0 \wedge t = x_0\|\!)$	Assignment
`x = y`	
$(\!\|x = y_0 \wedge t = x_0\|\!)$	Assignment
`y = t`	
$(\!\|x = y_0 \wedge y = x_0\|\!)$	Assignment

The key thing to notice: with the Assignment rule, the assertions are derived *mechanically*. It's a syntactic derivation – just like natural deduction.

To do: try this with a "broken" swap program, and see why the proof doesn't work.

5.2.4 If Statements

We want to derive the weakest precondition of

$$(\!|\phi|\!)\ \texttt{if}\ B\{C_1\}\ \texttt{else}\ \{C_2\}\ (\!|\psi|\!)$$

Steps:

1. Pull ψ backwards through C_2, to get ϕ_2. This expresses the weakest precondition of C_2, ψ.
2. Pull ψ backwards through C_1, to get ϕ_1. This expresses the weakest precondition of C_1, ψ.
3. Set ϕ to be $(B \to \phi_1) \wedge (\neg B \to \phi_2)$.

We'll call this the If′ Rule.

$$\frac{(\!|\phi_1|\!)\ C_1\ (\!|\psi|\!)\quad (\!|\phi_2|\!)\ C_2\ (\!|\psi|\!)}{(\!|B \to \phi_1 \wedge \neg B \to \phi_2|\!)\ \texttt{if}\ \ B\{C_1\}\ \texttt{else}\ \ \{C_2\}\ (\!|\psi|\!)} \tag{5.2.2}$$

Equation (5.2.2) can be derived from our original If rule.

Example 5.2.4 (Absolute Value)**:** Let's define the program `ABS`

```
if (x >= 0) {
  y = x;
} else {
  y = -x;
}
```

We want to prove $(\!|\top|\!)$ `ABS` $(\!|y = |x||\!)$.

$(\!|\top|\!)$	
$(\!|(x \geq 0 \to x = \|x\|) \wedge (\neg(x \geq 0) \to -x = \|x\|)|\!)$	
`if (x >= 0) {`	
$(\!|x = \|x\||\!)$	If Statement
`y = x`	
$(\!|y = \|x\||\!)$	Assignment
`} else {`	
$(\!|-x = \|x\||\!)$	If Statement
`y = -x`	
$(\!|y = \|x\||\!)$	Assignment
`}`	
$(\!|y = \|x\||\!)$	If′ Rule.

5.2.5 While Statements

The weakest pre-conditions for Assignment and If can be mechanically generated. This is not the case for while statements.

While rules will usually have the form

$$\frac{(\!|\eta \wedge B|\!)\ C\ (\!|\eta|\!)}{(\!|\eta|\!)\ \texttt{while}\ \ B\{C\}\ (\!|\eta \wedge \neg B|\!)} \tag{5.2.3}$$

But what we really want to show is

$$(\!|\phi|\!)\ \texttt{while}\ \ B\{C\}\ (\!|\psi|\!)$$

Typically, we'll have to use a creative guess to find η. The general procedure is:

1. Guess η
2. Try to show $\vdash_{\mathsf{AR}} \phi \to \eta$, and $\vdash_{\mathsf{AR}} \eta \wedge \neg B \to \psi$. If we can't show this, go back to step one and find a new η.

3. Pull η back through C to get η'
4. Try to find $\vdash (\eta \wedge B) \to \eta'$. If that's not possible, go back to step (1) and pick a new η. Otherwise, the selection of η worked.

Example 5.2.5 (Factorial #1)**:** Let's define the program `Fac1` as

```
y = 1;
x = 0;
while (z != x) {
  z = z + 1;
  y = y * z;
}
```

We want to show $(\!|\top|\!)$ `Fac1` $(\!|y = x! \wedge z \geq 0|\!)$. Let us chose η: $(y = z! \wedge z \geq 0)$.

$(\!\|\top\|\!)$	
$(\!\|1 = 0! \wedge 0 \geq 0\|\!)$	Implication
`y = 1`	
$(\!\|y = 0! \wedge 0 \geq 0\|\!)$	Assignment
`z = 0`	
$(\!\|y = z! \wedge z \geq 0\|\!)$	Assignment
`while (z != x) {`	
$(\!\|y = z! \wedge z \geq 0 \wedge z \neq x\|\!)$	
$(\!\|y \cdot (z+1) = (z+1)! \wedge (z+1) \geq 0\|\!)$	Implication
`z = z + 1`	
$(\!\|y \cdot z = z! \wedge z \geq 0\|\!)$	Assignment
`y = y * z`	
$(\!\|y = z! \wedge z \geq 0\|\!)$	Assignment
`}`	
$(\!\|y = z! \wedge z \geq 0 \wedge \neg\neg(z = x)\|\!)$	While
$(\!\|y = x!\|\!)$	Implication

Example 5.2.6 (Factorial #2)**:** Let us define the program `Fac2`:

```
y = 1;
while (x != 0) {
  y = y * x;
  x = x - 1;
}
```

We'll want to prove

$$(\!|x = x_0|\!)\ \texttt{Fac2}\ (\!|y = x_0!|\!)$$

Our invariant will be

$$\eta\colon x \geq 0 \to y \cdot x! = x_0! \tag{5.2.4}$$

We'll continue this example in our next lecture.

5.3 Lecture – 5/16/2007

5.3.1 Total Correctness

With one exception, the partial correctness rules we've looked at are also sound for total correctness. The exception to this is the while rule.

A total correctness rule must show that (1) the while statement is partially correct and (2) the while statement terminates.

We've looked at a partial correctness proof for `Fac1`:

```
y = 1;
z = 0
while (x != y) {
  z = z + 1;
  y = y * z;
}
```

The partial correctness proof had

$$\vdash_{\mathsf{par}} (\!|\top|\!)\ \texttt{Fac1}\ (\!|y = x!|\!)$$

A total correctness proof needs a stronger precondition (`Fac1` will not terminate if x is negative)

$$\vdash_{\mathsf{tot}} (\!|x \geq 0|\!)\ \texttt{Fac1}\ (\!|y = x!|\!)$$

A total correctness proof will also need to make use of *loop variants*.

5.3.2 Loop Variants

Definition 5.3.1 (Loop Variant)**:** A *loop variant* is a quantity that

1. is ≥ 0 whenever then loop execution finishes
2. Decreases (by an integer value) each time the loop is executed.

For `Fac1`, the loop invariant is $(x - z)$.

Total While Rule

$$\frac{(\!|\eta \wedge B \wedge 0 \leq E = E_0|\!)\ C\ (\!|\eta \wedge 0 \leq E < E_0|\!)}{(\!|\eta \wedge E \geq 0|\!)\ \texttt{while B \{ C \}}\ (\!|\eta \wedge \neg B|\!)} \tag{5.3.1}$$

In (5.3.1), E is an expression (e.g. $x - z$, the variant), and E_0 is a variable.

The precondition in the top half of the rule requires the variant to be positive. E_0 remembers the variant's starting value.

The postcondition in the top half of the rule says that the variant must be smaller after the loop terminates.

E need not play a rule in the while loop's predicate condition.

Example 5.3.2 (`Fac1`: total correctness)**:** A total correctness proof for `Fac1`.

$\{\!|0 \geq 0 \wedge 1 = 0! \wedge x \geq 0 \wedge 0 \leq x - 0|\!\}$
`y = 1`
$\{\!|0 \geq 0 \wedge y = 0! \wedge x \geq 0 \wedge 0 \leq x - 0|\!\}$
`z = 0`
$\{\!|z \geq 0 \wedge y = z! \wedge x \geq 0 \wedge 0 \leq x - z|\!\}$
`while (x` $\neq$ `z) {`
$\{\!|z \geq 0 \wedge y = z! \wedge x \geq 0 \wedge 0 \leq x - z = E_0 \wedge z \neq x|\!\}$ [Note 1]
$\{\!|z + 1 \geq 0 \wedge y \cdot (z + 1) = (z + 1)! \wedge x \geq 0 \wedge 0 \leq x - (z + 1) \leq E_0|\!\}$
`z = z + 1`
$\{\!|z \geq 0 \wedge y \cdot z = z! \wedge x \geq 0 \wedge 0 \leq x - z < E_0|\!\}$
`y = y + z`
$\{\!|z \geq 0 \wedge y = z! \wedge x \geq 0 \wedge 0 \leq x - z < E_0|\!\}$
`}`
$\{\!|z \geq 0 \wedge y = z! \wedge \neg(x \neq z) \wedge x \geq 0|\!\}$
$\{\!|y = x!|\!\}$

Note 1: η is $z \geq 0 \wedge y = z! \wedge x \geq 0$.
$0 \leq x - z = E_0$ captures the loop variant, $(x - z)$.
$z \neq x$ is B.

5.3.3 Working With Arrays

In a formal proof system, arrays can be tricky to work with. For example, given $a[i] = a[j]$, what if $i = j$?

We'll notate arrays as

$$a[0] \ldots a[n-1]$$

Let's define a *section of an array* to be a consecutive series of elements, $a[i] \ldots a[j]$ such that $i \leq j$.

We'd like to find the *minimal section sum* for the array. What is the smallest number that can be derived from summing elements in a section?

For example, given the array $[-1, 2, -3, 5, 4, -3]$, the minimal section sum is -3. Given $[-1, 2, -3, 1, 1, -3]$, the minimal section sum is -4.

The naive approach would be to try all possible sections; a runtime complexity of $\Theta(n^3)$. But we can do better, in $\Theta(n)$ time.

Let

$s =$ the minimal section sum up the current point k
$t =$ the minimal section sum up to and including the current point k.

Given current values s, t, we compute the next values s', t' as follows:

$$s' = \min(s, a[k+1], a[k+1] + t)$$
$$t' = \min(a[k+1], a[k+1] + 1)$$

The algorithm `Min-sum` is

```
k = 1;
s = a[0];
t = a[0];
```

```
while (k != n) {
  t = min(a[k + 1],  a[k + 1] + t);
  s = min(s, t);
  k = k + 1;
}
/* s holds final result */
```

Suppose we wanted to formally verify this algorithm. Let us denote

$$S_{ij} = \sum_{k=i}^{j} a[k]$$

We would use two postconditions:

$$(\!|n \geq 0|\!)\ \texttt{Min-sum}\ (\!|\forall i, j(0 \leq i \leq j < n \rightarrow s \leq S_{ij})|\!)$$
$$(\!|n \geq 0|\!)\ \texttt{Min-sum}\ (\!|\exists i, j(0 \leq i \leq j < n \wedge s = S_{ij})|\!)$$

The first postcondition states that s is $\leq$ any section sum. The second postcondition states that s *is* the sum of some section.

This example illustrates how modularity can be introduced into the verification process. We have a common set of preconditions and a pair of post conditions. The two postconditions can be proven independantly.

To prove the $\forall$ part, we'll define a few shorthand notations:

$$\mathsf{Inv1}(s, k) = \forall i, j(0 \leq i \leq j < k \rightarrow s \leq S_{ij})$$
$$\mathsf{Inv2}(t, k) = \forall i(0 \leq i < k \rightarrow t \leq S_{i,k-1})$$

With these helpers, the loop invariant is

$$(\!|0 \leq k \leq n \wedge \mathsf{Inv1}(s, k) \wedge \mathsf{Inv2}(t, k)|\!)$$

For the $\exists$ part, we'd do a similar thing:

$$\mathsf{Inv1}'(s, k) = \exists i, j(0 \leq i \leq j < k \wedge s = S_{ij})$$

(There'd also be an $\mathsf{Inv2}'$ and another loop invariant).

5.4 Final Exam Review – 5/21/2007

5.4.1 Problem 3 (LTL Model Checking)

Given an LTL formula ϕ, we want to determine whether $\mathcal{M}, s \vDash \mathrm{A}\,\phi$.

Note that $\mathcal{M}, s \vDash \mathrm{A}\,\phi$ holds IFF $\mathcal{M}, s \nvDash \mathrm{E}\,\neg\phi$.

Our goal is to find a set of states such that $\mathcal{M}, s \vDash \mathrm{E}\,\neg\phi$. The compliment of this set will be states such that $\mathcal{M}, s \vDash \mathrm{A}\,\phi$.

We give an algorithm to find $\mathcal{M}, s \vDash \psi$, where $\psi = \neg\phi$. We are interested in the compliment of the set of states satisfying $\mathcal{M}, s \vDash \psi$.

For this problem, $\phi = a\,\mathrm{U}\,\mathrm{X}\,\neg(\neg a \vee b)$, and $\psi = \neg\phi = \neg(a\,\mathrm{U}\,\mathrm{X}\,\neg(\neg a \vee b))$.

We assume that ϕ uses only the CTL connectives $\neg$, X, U.

Our first goal is to form the transition system $A_\psi = (T, \delta)$.

Let $\mathcal{C}(\psi)$ be the set of positive sub-formulas of ψ and their negations.

$$\begin{aligned}\mathcal{C}(\psi) =& a, b, \neg a \vee b, \mathrm{X}\,\neg(\neg a \vee b), a\,\mathrm{U}\,\mathrm{X}\,\neg(\neg a \vee b),\\ & \neg a, \neg b, \neg(\neg a \vee b), \neg\,\mathrm{X}\,\neg(\neg a \vee b), \neg(a\,\mathrm{U}\,\mathrm{X}\,\neg(\neg a \vee b))\end{aligned}$$

Forming T

Let T be the set of all subsets q of $\mathcal{C}(\psi)$ such that for all positive subformulas $\eta \in \psi$, either $\eta \in q$ OR $\neg\eta \in q$, but not both.

Consistency Conditions for T

All member $q \in T$ must meet the following conditions:

1. For all η_1, η_2 in $\mathcal{C}(\psi)$; $\eta_1 \vee \eta_2 \in q$ IFF (a) $\eta_1 \in q$ OR (b) $\eta_2 \in q$.
2. For all $\eta_1\,\mathrm{U}\,\eta_2 \in \mathcal{C}(\psi)$; if $\eta_1\,\mathrm{U}\,\eta_2 \in q$, then (a) $\eta_1 \in q$ OR (b) $\eta_2 \in q$.
3. For all $\eta_1\,\mathrm{U}\,\eta_2 \in \mathcal{C}(\psi)$; if $\neg(\eta_1\,\mathrm{U}\,\eta_2) \in q$, then $\eta_2 \notin q$. (equivalently, $\neg\eta_2 \in q$).

There are no restrictions on X connectives when computing maximal consistent sets of $\mathcal{C}(\psi)$.

δ Rules

A transition $(q, q') \in \delta$ IFF

1. $\mathrm{X}\,\eta \in q \Rightarrow \eta \in q'$
2. $\mathrm{X}\,\eta \notin q \Rightarrow \eta \notin q'$
3. $\eta_1\,\mathrm{U}\,\eta_2 \in q$, $\eta_2 \notin q \Rightarrow \eta_1\,\mathrm{U}\,\eta_2 \in q'$
4. $\eta_1\,\mathrm{U}\,\eta_2 \notin q$, $\eta_1 \in q \Rightarrow \eta_1\,\mathrm{U}\,\eta_2 \notin q'$

$A_\psi = (T, \delta)$ is an abstract picture of ψ. It doesn't tell us anything about our specific model $\mathcal{M}$. Our next step is to attach $\mathcal{M}$ to A_ψ.

Forming $\mathcal{M} \times A_\psi$

We define $\mathcal{M} \times A_\psi = (U, \delta')$, where

$$U = \{(s, q) \in S \times T \mid \text{for all atoms } p \in \mathcal{C}(\psi), p \in q \text{ IFF } p \in L(s)\}$$

Using problem 3 as an example, if our state is $q = \texttt{00100}$, then $\neg a \in q$ and $\neg b \in q$. We must pair q with a state s of $\mathcal{M}$ where s has $a \notin L(s)$ and $b \notin L(s)$.

Whatever $\mathcal{M}$ says about atoms in s, q must say the same things about atoms in $\mathcal{C}(\psi)$.

Next, the transition relation δ'.

$$\delta' = \{(s, q) \to (s', q') \mid s \to s' \in \mathcal{M} \text{ and } q \to q' \in \delta\}$$

In other words, the transition relations of $\mathcal{M}$ and A_ψ must be in agreement.

Strongly Connected Components

Next, we'll find the strongly connected components of $\mathcal{M} \times A_\psi$. There are algorithms that accomplish this task. For our purpose, eyeballing $\mathcal{M} \times A_\psi$ will be sufficient.

A strongly connected component C is a connected component where the following is true: from each node in C, you can get to every other node in C.

The set of strongly connected components forms a partition of a graph. A trivial SCC cannot be part of a non-trivial SCC. Figure 5.1 shows a graph with 1 Trivial SCC and two non-trivial SCCs.

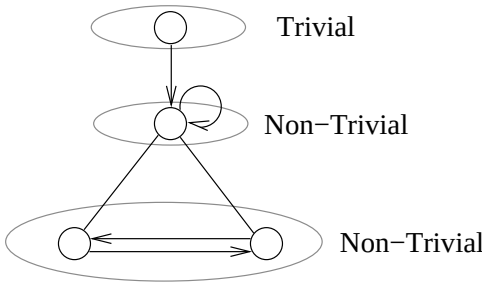

Figure 5.1: Graph with 1 trivial and 2 non-trivial SCCs

Self Fulfilling SCC

A strongly connected component of $\mathcal{M} \times A_\psi$ is self fulfilling if, for all $\eta_1 \,\mathrm{U}\, \eta_2$ that are in some SCC node (s, q), η_2 is in some node (s', q') of the SCC.

If $\eta_1 \,\mathrm{U}\, \eta_2$ does not occur in any SCC component, then this condition is vacuously true (the SCC is self fulfilling).

$\mathcal{M}, s \vDash \mathrm{E}\,\psi$ IFF there is a path from an (s, q) (with $\psi \in q$) in $\mathcal{M} \times A_\psi$, to a non-trivial self-fulfilling SCC of $\mathcal{M} \times A_\psi$.

Additional Notes

Assuming ϕ is positive, A_ϕ and $A_{\neg\phi}$ will produce the same model, so $A_\phi = A_{\neg\phi}$ for a positive ϕ.

In the last step (self-fulfilling SCC), we are looking for $\psi = \neg\phi$.

We find $\mathrm{E}\,\neg\phi$ in order to find $\mathrm{A}\,\phi$.

5.4.2 Problem 4

The definition of "active assumption" will involve boxes. We can assume that the proof has properly nested boxes.

5.4.3 Problem 5

A Greatest fixed point proof involves two things

1. $K([\![\mathrm{E_C G}\,\phi]\!]) = [\![\mathrm{E_C G}\,\phi]\!]$.
2. If $K(X) = X$, then $X \subseteq [\![\mathrm{E_C G}\,\phi]\!]$.

The first step shows that $[\![\mathrm{E_C G}\,\phi]\!]$ is a fixed point of K. The second step shows that it is the greatest fixed point of K.

GNU Free Documentation License

Version 1.2, November 2002

51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA

Preamble

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "**Document**", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "**you**". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "**Modified Version**" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "**Secondary Section**" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "**Invariant Sections**" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "**Cover Texts**" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "**Transparent**" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "**Opaque**".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "**Title Page**" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section "**Entitled XYZ**" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "**Acknowledgements**", "**Dedications**", "**Endorsements**", or "**History**".) To "**Preserve the Title**" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose

the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties–for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.