(Temporal) Logic Tutorial

Edsko de Vries

October 22, 2006

1 Propositional Logic

1.1 Syntax

The syntax of a formula ϕ in propositional logic is given by

$$\phi \ ::= \ p \ | \ \neg\phi \ | \ \phi \wedge \phi \ | \ \phi \vee \phi \ | \ \phi \rightarrow \phi$$

denoting an arbitrary variable p, negation, conjunction, disjunction, and implication respectively. An example is

$$(p \wedge \neg q) \rightarrow r \qquad (\alpha)$$

1.2 Semantic Entailment

Is α *true*? Well, that depends on the values of p, q and r. If we list all possible **valuations** of p, q and r, we obtain the **truth table** for ϕ:

p	q	r	ϕ
-	-	-	+
-	-	+	+
-	+	-	+
-	+	+	+
+	-	-	-
+	-	+	+
+	+	-	+
+	+	+	+

Thus, α is true (+) *unless* $p \wedge \neg q \wedge \neg r$. Conversely, if we are given α, we can conclude that any valuation of p, q and r is possible, *except* $p \wedge \neg q \wedge \neg r$. Suppose we are given α, p and $\neg r$. From the truth table, we can conclude q. To paraphrase, from α, p and $\neg r$, we can conclude q, based on the *truth* of α, p and r. This is called **semantic entailment** and is denoted

$$\alpha, p, \neg r \vDash q$$

1.3 Inference

The question arises whether we can also **prove** q from α, p and $\neg r$, reasoning only **symbolically**. It turns out that we can. One proof is given by:

(1)	$(p \wedge \neg q) \rightarrow r$	given
(2)	p	given
(3)	$\neg r$	given
(4)	$\neg q$	assumption
(5)	$p \wedge \neg q$	from (2) and (4)
(6)	r	from (1) and (5)
(7)	$\neg r \wedge r$	from (3) and (6)
(8)	contradiction	law of the excluded middle
(9)	q	conclusion (proof by contradiction)

This is called **inference** and is denoted by

$$\alpha, p, \neg r \vdash q$$

Inference works by applying proof rules. For example, the two proof rules that were used in lines (7) and (8) of the proof above are

$$\frac{\phi \quad \psi}{\phi \wedge \psi} \qquad \frac{\phi \quad \neg\phi}{\text{contradiction}}$$

I.e., if you know ϕ, and you know ψ, you can infer $\phi \wedge \psi$. Similarly, if you know ϕ and you know $\neg\phi$, you have a contradiction on your hands.

1.4 Soundness and Completeness

A logic is **complete** if we can proof a formula ψ from a formula ϕ whenever ψ follows semantically from ϕ, i.e. $\phi \vdash \psi$ whenever $\phi \vDash \psi$. A logic is **sound** if a formula ψ follows semantically from a a formula ϕ whenever we can proof ψ from ϕ, i.e. $\phi \vDash \psi$ whenever $\phi \vdash \psi$.

In plain English, completeness means that we can prove everything that we should be able to prove, and soundness means that we cannot prove anything that we should not be able to prove. Propositional logic is both sound and complete.

1.5 Some Definitions

A rule $\phi_1, \phi_2, \cdots, \phi_n \vdash \psi$ is called a **sequent**. $\phi_1, \phi_2, \cdots, \phi_n$ are the **premises** of the sequent, and ψ is the **conclusion** of the sequent. If a formula ψ can be proven independent of any premises, i.e. if $\vdash \psi$, then ψ is called a **theorem**. For example, $p \vee \neg p$ is a theorem (this particular theorem is called the *law of the excluded middle*).

If a formula ψ is true independent of the valuation of the variables in the formula (i.e., its truth table lists "true" in

every row), the formula is called a **tautology** (denoted $\vDash \psi$). If a formula is a tautology, the formula is said to be **valid**. When a logic is both sound and complete, the theorems are the tautologies.

Similar to the notion of validity, a formula is **satisfiable** if it is true for *at least one* valuation of its variables (i.e., its truth table lists "true" in at least one row). These two notions are closely linked: if a formula ϕ is valid, then $\neg\phi$ is not satisfiable: if ϕ is valid, it must always be true; hence $\neg\phi$ is *never* true and thus not satisfiable. Conversely, if a formula ϕ is satisfiable, then $\neg\phi$ is not valid.

Finally, two formulae ϕ and ψ are **provably equivalent**, denoted $\phi \dashv\vdash \psi$, iff $\phi \vdash \psi$ and $\psi \vdash \phi$. Two formulae ϕ and ψ are (semantically) **equivalent**, denoted $\phi \equiv \psi$, iff $\phi \vDash \psi$ and $\psi \vDash \phi$.

2 Predicate (First-Order) Logic

2.1 Syntax

A formula ϕ in predicate logic is built up of **predicates**. A predicate P consists of a number of **terms**. A term is either a variable x, a constant c, or a function $\mathtt{f}(\mathtt{t}_1,\mathtt{t}_2,\cdots,\mathtt{t}_n)$ of a number of terms. The syntax for formulae then closely resembles the syntax for formulae in propositional logic, except that we introduce two **quantifiers**: the **universal quantifier** ("for all x", $\forall x$) and the **existential quantifier** ("there exists an x such that", $\exists x$). The syntax of predicate logic is given by

$$\mathtt{t} ::= x \mid \mathtt{c} \mid \mathtt{f}(\mathtt{t},\cdots,\mathtt{t})$$
$$\phi ::= \mathtt{P}(\mathtt{t}_1,\mathtt{t}_2,\cdots,\mathtt{t}_n)$$
$$\phi ::= \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \phi \rightarrow \phi$$
$$\phi ::= \forall x \cdot \phi \mid \exists x \cdot \phi$$

An example is

$$\forall x\, \exists y \cdot D(x,y) \qquad (\beta)$$

2.2 Models

Like before, we would like to ask the question, is β true? For the propositional case we answered this question by giving a truth table. Unfortunately, it is not so easy to give a truth table for β. The problem is that x and y could potentially range over an infinite set. Moreover, what exactly do we mean by $D(x,y)$?

Suppose that x and y range over the natural numbers, i.e. $x,y \in \mathbb{N}$, and suppose that $D(x,y)$ means $y|x$ (y divides x). Clearly, in that **interpretation**, β is true. But if in another interpretation x and y range over the "set of humans", and $D(x,y)$ means "y is a daughter of x", β is false because not everybody has a daughter.

Formally, an interpretation is a **model** $\mathcal{M}$. For each variable x, the model must specify a set X where x ranges over X. For every function $f(x_1,x_2,\cdots,x_n)$, the model must specify a concrete function $f' \in X_1 X_2 \cdots X_n \rightarrow X_{n+1}$, and finally, for every predicate $P(x_1,x_2,\cdots,x_n)$, the model must specify a concrete relation $P' \in X_1 \times X_2 \times \cdots \times X_n$.

If a function ϕ is true in a model $\mathcal{M}$, we write $\mathcal{M} \vDash \phi$. To check whether $\mathcal{M} \vDash \phi$ is known as **model checking**[1].

2.3 Semantic Entailment

If in some model $I(x)$ means "x is Irish", and $E(x)$ means "x speaks English", the following sequent holds.

$$\forall x \cdot I(x) \vDash_{\mathcal{M}} \forall x \cdot E(x)$$

(Note the use of $\vDash_{\mathcal{M}}$ to indicate semantic entailment with respect the model $\mathcal{M}$.) Now consider the sequent

$$\forall x \cdot P(x) \rightarrow Q(x), \forall x \cdot P(x) \vDash \forall x \cdot Q(x)$$

This sequent is true *no matter what model we assume*, written $\phi \vDash \psi$ (semantic entailment).

2.4 Definitions

A formula ϕ is **satisfiable** iff there is some model $\mathcal{M}$ such that $\mathcal{M} \vDash \phi$. The set of premises Γ of a sequent is **consistent** (or sometimes **satisfiable**) iff there is a model $\mathcal{M}$ such that $\mathcal{M} \vDash \phi$ for all $\phi \in \Gamma$. A formula ϕ is **valid** iff $M \vDash \phi$ for all appropriate models $\mathcal{M}$.

Note that we have overloaded the $\vDash$ operator: $\mathcal{M} \vDash \phi$ denotes model checking, whereas $\phi \vDash \psi$ denotes semantic entailment.

Model checking is decidable only for finite models; semantic entailment (checking whether $\phi \vDash \psi$ holds for any model) is even harder and **undecidable**.

There is also an inference procedure for predicate logic, such that $\phi \vdash \psi$ if and only if $\phi \vDash \psi$, i.e., predicate logic is both sound and complete.

2.5 Higher Order Logics

Suppose in a model $\mathcal{M}$ the variables range over the nodes of a graph, and we have a single predicate $E(x,y)$ indicating an edge from x to y. Suppose we want to define a formula ρ that evaluates to true if where is a path from a node n_1 to another node n_2 (i.e., if n_2 is reachable from n_1). We could try to define ρ as

$$E(n_1,n_2) \vee \exists n_3 \cdot E(n_1,n_3), E(n_3,n_2) \vee \cdots \qquad (\rho)$$

Obviously, ρ would be infinitely long—no use. It turns out this formula cannot actually be written in predicate logic (for a proof refer to [1]). We have to use a **higher order** logic. In second-order logic, the existential and universal quantifiers can range over predicate symbols as well as over variables. We will not discuss higher order logics any further.

[1]We are ignoring free variables here for ease of discussion

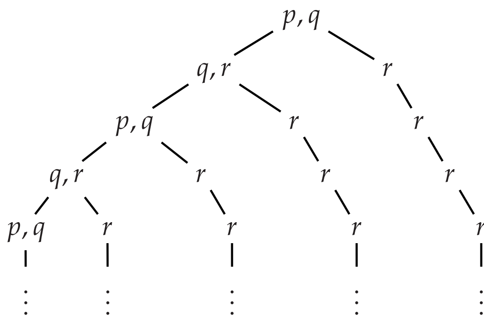
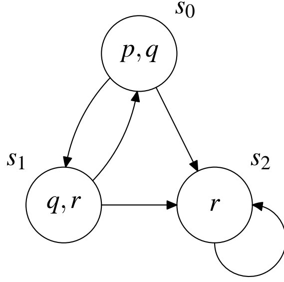


Figure 1: Computation tree

3 Temporal Logic

3.1 Transition Systems

Where in predicate logic the choice of model was left completely open, in temporal logics only a single type of model is considered, which is a transition system (essentially a finite-state machine). In a temporal logic, an atom is no longer always true or always false. Rather, is true for certain states of the transition system. A transition system can be described graphically, for example:



Formally, a transition system $\mathcal{M}$ is a triple $(S, \rightarrow, L)$, with S is a set of states, $\rightarrow$ a binary relation on S such that for all $s \in S$ there exists an s', $s \rightarrow s'$, and finally a function $L : S \rightarrow \wp(\texttt{Atoms})$ that maps states to sets of atoms (for each state it lists the atoms that are true in that state). In the example, $S = \{s_0, s_1, s_2\}$, $\rightarrow = \{(s_0, s_1), (s_0, s_2), (s_1, s_0), (s_1, s_2), (s_2, s_2)\}$ and L maps s_0 to $\{p, q\}$, s_1 to $\{q, r\}$ and s_2 to $\{r\}$.

Given such transition system, we can derive a computation tree starting from a particular state. For the example above, starting from state s_0, we obtain the (infinite) tree shown in figure 1.

3.2 Linear-Time Temporal Logic (LTL)

From the tree in figure 1, we can follow the individual computation paths (amongst others):

$$(p, q), r, r, r, \cdots$$
$$(p, q), (q, r), r, r, r, \cdots$$
$$(p, q), (q, r), (p, q), r, r, r, \cdots$$
$$(p, q), (q, r), (p, q), (q, r), (p, q), \cdots$$

Formulae in LTL range over these paths. The syntax for LTL is given by

ϕ ::= $\top$ | $\bot$
ϕ ::= p | $\neg\phi$ | $\phi \wedge \phi$ | $\phi \vee \phi$ | $\phi \rightarrow \phi$
ϕ ::= X ϕ | F ϕ | G ϕ
ϕ ::= ϕ U ϕ | ϕ W ϕ | ϕ R ϕ

Let a path $\pi = \{s_1, s_2, s_3, \cdots\}$, and let π^i be the path starting from the i'th state, i.e. $\pi^2 = \{s_2, s_3, \cdots\}$. Then:

- $\pi \vDash p$ iff $p \in L(s_1)$. E.g. $\pi \vDash p$ if p is true in the first state of the path
- $\pi \vDash \neg\phi$ iff $\pi \nvDash \phi$ (similarly for $\wedge$, $\vee$ and $\rightarrow$)
- $\pi \vDash \mathrm{X}\ \phi$ iff $\pi^2 \vDash \phi$. E.g., $\pi \vDash \mathrm{X}\ p$ if p is true in the second state of the path
- $\pi \vDash \mathrm{F}\ \phi$ iff $\pi^i \vDash \phi$ for some $i \geq 1$
- $\pi \vDash \mathrm{G}\ \phi$ iff $\pi^i \vDash \phi$ for all $i \geq 1$
- $\pi \vDash \phi\ \mathrm{U}\ \psi$ iff $\pi^i \vDash \psi$ for some i and for all $1 \leq j < i$, $\pi^j \vDash \phi$, i.e. ϕ holds until ψ holds (note that ϕ might continue to hold even after ψ holds). W and R are variants on U.

As before, we are interested in model satisfaction: $\mathcal{M}, s_0 \vDash \phi$ if ϕ holds for all paths in the model starting from state s_0. For the example given above, we have

- $\mathcal{M}, s_0 \vDash p, q$ because $\{p, q\} \subseteq L(s_0)$
- $\mathcal{M}, s_0 \vDash \mathrm{X}\ r$ because in all paths starting from s_0, r holds in the second state of each path
- $\mathcal{M}, s_0 \vDash \mathrm{F}\ (\neg q \wedge r) \rightarrow \mathrm{F\ G}\ r$. For each path, if the path has a state where $\neg q \wedge r$, then on that path, there is a state i such that $\pi^i \vDash \mathrm{G}\ r$, i.e., such that r holds for all states from state i onwards.

3.3 Computation Tree Logic (CTL)

There is many things that we cannot express in LTL. For example, "from a state where p holds, it is always possible to get to a state where q holds". The LTL formula G $(p \rightarrow \mathrm{F}\ q)$ says that from a state where p holds we will *always* go to a state where q holds (remember that LTL formulae quantify over all possible paths).

In CTL, we add quantifiers to the syntax to say whether we want to specify that a formula holds over all paths (universal quantifier) or over at least one path (existential quantifier). The syntax of CTL is given by

ϕ ::= $\top$ | $\bot$
ϕ ::= p | $\neg\phi$ | $\phi \wedge \phi$ | $\phi \vee \phi$ | $\phi \rightarrow \phi$
ϕ ::= AX ϕ | EX ϕ
ϕ ::= AF ϕ | EF ϕ
ϕ ::= AG ϕ | EG ϕ
ϕ ::= A[ϕ U ϕ] | E[ϕ U ϕ]

Because the semantics of these connectives is fairly intuitive (up to a point), we won't specify them formally here. The formula that we were looking for earlier, "from a state where p holds, it is always possible to get to a state where q holds", can now be expressed in CTL: AG$(p \rightarrow \mathrm{EF}\ q)$. In words: for

all paths, if p holds in some state s, then q will hold on *at least one* path starting from s.

3.4 CTL*

As we have seen, there are things that we can express in CTL that we cannot express in LTL. Unfortunately, the reverse holds as well. For example, "all paths which have a p along them also have a q along them", is described by the LTL formula F $p \rightarrow$ F q. The CTL formula AF p $\rightarrow$ AF q means something different: it says, if all paths have a p along them, then all paths have a q along them. The CTL formula AG $(p \rightarrow \text{EF}\ q)$ does not describe quite the same thing either, because it says (as we have seen in the last section) that all paths that have a p along them, will eventually meet a q; but it does not allow for this q to occur *before* the p.

CTL* is a logic that is strictly more powerful than both LTL and CTL (i.e., it incorporates both CTL and LTL). The syntax of CTL* is given by

$$\phi ::= \top \mid \bot$$
$$\phi ::= \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi$$
$$\phi ::= \mathtt{A}[\alpha] \mid \mathtt{E}[\alpha]$$

$$\alpha ::= \phi \mid \neg\alpha \mid \alpha \wedge \alpha \mid \alpha \vee \alpha$$
$$\alpha ::= \alpha\ \mathtt{U}\ \alpha \mid \mathtt{G}\ \alpha \mid \mathtt{F}\ \alpha \mid \mathtt{X}\ \alpha$$

The important difference between this syntax and the syntax of CTL is that the quantifiers are no longer always matched to a temporal connective. For example, "all paths which have a p along them also have a q along them" is specified by A[F $p \rightarrow$ F q].

Model checking for LTL and CTL* has an exponential complexity, whereas model checking for CTL has a complexity linear in the size of the formula and quadratic in the size of the model (unfortunately, the size of the model itself often grows exponentially with the complexity of the system being modeled.)

4 Relevance to Compiler Design

Many properties used in program analysis in compiler design can be described by temporal logic formulae. Lacey [2] uses a version of computation tree logic (CTL), limited to the AX, EX, AU and EU operators, but introducing a "backwards" version of each of these operators ($\mathtt{AX}^{\triangle}$, $\mathtt{EX}^{\triangle}$, $\mathtt{AU}^{\triangle}$ and $\mathtt{EU}^{\triangle}$), defined the way you would expect (for example, $\mathcal{M}, s_0 \models \mathtt{EX}^{\triangle}\phi$ if for some *previous* state s_{-i}, we have $\mathcal{M}, s_{-i} \models \phi$).

Lacey describes a compiler framework in which a compiler is specified in terms of conditional rewrite rules; the conditions can be specified using temporal logic. One example, describing constant propagation, is given by

$$\begin{aligned} & n : (x := v) \implies x := c \\ \textbf{if } & n \models \mathtt{A}^{\triangle}\big(\neg \mathit{def}(v)\ \mathrm{U}\ \mathit{def}(v) \wedge \mathit{stmt}(v := c)\big) \\ & \text{and } \mathit{conlit}(c) \end{aligned}$$

This rule should be read as: a statement $x := v$ can be replaced by another statement $x := c$, if c is a constant literal (*conlit*), and if on all paths to the node (i.e., backwards), v is not defined until it is defined and given the value c; i.e., if on all paths to the node, the last assignment to v on every path is an assignment $v := c$ (note that we use the control flow graph of the program as a model).

As another example, (a simple approach to) dead code elimination is described by

$$\begin{aligned} n : (x := e) & \implies \epsilon \\ \textbf{if } n & \models \mathtt{AX}(\neg\mathtt{E}(\top\ \mathrm{U}\ \mathit{use}(x))) \end{aligned}$$

The strange construction $E(\top\ \mathrm{U}\ \phi)$ is used because of the absence of the EF operator; the same rule would be described more simply by

$$\begin{aligned} n : (x := e) & \implies \epsilon \\ \textbf{if } n & \models \mathtt{AX}(\neg\mathtt{EF}(\mathit{use}(x))) \end{aligned}$$

In words, an assignment $x := e$ can be eliminated if x is not used on any path from the assignment. Note that the following simplification of the rule, although more intuitive, is incorrect:

$$\begin{aligned} n : (x := e) & \implies \epsilon \\ \textbf{if } n & \models \neg\mathtt{EF}(\mathit{use}(x)) \end{aligned}$$

This rule is incorrect because the "future" in standard temporal logic includes the "present" (see the formal semantics of the temporal connectives in LTL, section 3.2). So, this last rule would fail to eliminate a rule such as $x := x + 1$, even when x is never used again, because x is used in the rule itself.

The important thing to realise is that these rules are really definitions of when we can apply constant propagation or do dead code elimination, but, equally important, they are also executable: the compiler can be written in a declarative manner.

References

[1] HUTH, M., AND RYAN, M. *Logic in Computer Science*. Cambridge University Press, 2004.

[2] LACEY, D., AND DE MOOR, O. Imperative program transformation by rewriting. In *CC '01: Proceedings of the 10th International Conference on Compiler Construction* (2001), Springer-Verlag, pp. 52–68.