Logic and Applications

Jussi Rintanen
Department of Computer Science
Aalto University
Helsinki, Finland

March 16, 2024

Contents

Table of Contents . . . i

1 Introduction **1**

2 Propositional Logic **2**
2.1 Formulas . . . 2
2.2 Valuations and Truth-Values . . . 3
2.3 Equivalences . . . 4
2.4 Truth-tables . . . 4
2.5 Logical Consequence, Satisfiability, Validity . . . 6

3 Reasoning in the Propositional Logic **9**
3.1 Truth-Tables . . . 9
3.2 Tree-Search Algorithms . . . 10
3.2.1 Unit Resolution . . . 11
3.2.2 Subsumption . . . 12
3.2.3 Unit Propagation . . . 12
3.2.4 The Davis-Putnam Procedure . . . 12

4 Normal Forms and Logic Data Structures **14**
4.1 Complexity of Validity and Satisfiability for DNF and CNF . . . 16
4.2 Formulas as a Data Structure . . . 16
4.3 Binary Decision Diagrams . . . 17
4.3.1 Properties of OBDDs . . . 21
4.3.2 Model Counting . . . 22
4.3.3 Clausal consequences . . . 23
4.3.4 Restriction . . . 23
4.3.5 Abstraction . . . 23
4.3.6 Variable Ordering . . . 24
4.4 Normal Forms DNNF and d-DNNF . . . 24
4.5 Normal Forms Summary . . . 27

5 Sets and Relations in the Propositional Logic **28**
5.1 Representation of Sets . . . 28
5.1.1 Set Operations as Logical Operations . . . 28
5.2 Relations as Formulas . . . 29
5.2.1 Relational Operations in Logic . . . 31

6 Transition Systems **34**
6.1 Basic Models . . . 34
6.2 Procedural Representations . . . 35
6.2.1 Deterministic Unconditional Transitions . . . 35
6.2.2 General Definition . . . 36
6.3 Higher Order Representations . . . 37

7 Symbolic Reachability Testing **39**
7.1 OBDD-Based Symbolic Reachability . . . 39

7.2 SAT-Based Symbolic Reachability . . . 41
7.2.1 Parallel Actions . . . 42
7.2.2 Alternative Meanings of Parallelism . . . 43

8 Modal Logics **46**
8.1 Modal Logics with One Modality . . . 46
8.1.1 Validity in classes of frames . . . 48
8.1.2 Properties of the accessibility relation . . . 48
8.1.3 Different uni-modal logics . . . 49
8.1.4 Undefinable properties . . . 50
8.1.5 Logical consequence in modal logics . . . 51
8.2 Multi-Modal Logics . . . 51
8.3 Linear Temporal Logic LTL . . . 52
8.4 Branching Time Temporal Logics CTL and CTL$*$ . . . 53
8.4.1 The language of CTL$*$ . . . 53
8.4.2 The semantics of CTL$*$ . . . 53
8.4.3 Computation tree logic CTL . . . 54
8.4.4 Relations between CTL, LTL and CTL$*$ . . . 54
8.5 Dynamic Logic . . . 55
8.5.1 Propositional dynamic logic PDL . . . 55
8.5.2 The semantics of PDL . . . 55
8.5.3 Axiom system for PDL . . . 56
8.5.4 Relations to other modal logics and Hoare logic . . . 56

9 Model-Checking **58**
9.1 Model-Checking Algorithms . . . 58
9.1.1 CTL Model-Checking . . . 59
9.2 A model-checking algorithm for LTL . . . 60
9.2.1 Fairness . . . 65
9.3 Symbolic Model-Checking . . . 66
9.3.1 Symbolic CTL Model-Checking . . . 66
9.3.2 Symbolic LTL Model-Checking . . . 67

10 Predicate Logic **69**
10.1 Introduction . . . 69
10.2 Syntax . . . 71
10.3 Semantics . . . 72
10.4 Decision Problems . . . 74
10.5 Equivalences . . . 74
10.6 Application: Natural Language . . . 75
10.7 Application: Databases . . . 77

Index **81**

A The λ-Calculus **87**
A.1 Basics . . . 87
A.2 Types . . . 89
A.2.1 Type Inference . . . 90
A.3 Expressive Power of Pure λ-Calculus . . . 91

B Relational Algebra **92**

Chapter 1

Introduction

The classical propositional logic is the most basic and most widely used logic. It is a notation for Boolean functions, together with several proof and reasoning methods.

The use of the propositional logic as a computational tool has dramatically increased since the development of powerful search algorithms and implementation methods since the late 1990s. Today it enjoys extensive use in several areas of computer science, especially in Computer-Aided Verification and Artificial Intelligence. Its uses in AI include planning, problem-solving, intelligent control, and diagnosis, and in Computer-Aided Verification it is used in Model-Checking, Reachability Analysis, Deadlock Detection, and many other problems.

The reason why logics are used is their ability to precisely express data and information, in particular when it is *partial* or *incomplete*, and some of the *implicit* consequences of the information must be *inferred* to make them *explicit*.

The propositional logic, as the first known NP-complete problem [Coo71], is used for representing many types of co-NP-complete and NP-complete *combinatorial* search problems. Such problems are prevalent in artificial intelligence as a part of *decision-making*, *problem-solving*, and *planning*.

For many applications natural choices would be various more expressive logics, including the *predicate logic* or various *modal logics*. These logics, however, lack the kind of efficient and scalable algorithms that are available for the classical propositional logic. The existence of high performance algorithms for reasoning with propositional logic is the main reason for its wide use.

Chapter 2

Propositional Logic

2.1 Formulas

Propositional logic is about Boolean functions, which are mappings from *truth-values* 0 and 1 (*false* and *true*) to truth-values. Arbitrary Boolean functions can be represented as *formulas* formed with *connectives* such as $\wedge$, $\vee$ and $\neg$, and it can be shown that any Boolean function can be represented by such a formula.

The informal readings of some of the most standard connectives are as follows.

- The *conjunction* denoted by $\wedge$ is for *and*, as in “It is Monday *and* it rains”.
- The *disjunction* denoted by $\vee$ is for *or*, as in “It is Monday *or* it is Tuesday”.
- The *negation* denoted by $\neg$ is for *not*, as in “It is *not* the case that it is Monday”.
- The *implication* denoted by $\rightarrow$ is for conditional truth, as in “*If* it is Monday *then* it is not not week-end”.
- The *equivalence* denoted by $\leftrightarrow$ is for expressing that truth values are the same, as in “It is week-end *if and only if* it is Saturday or Sunday.”

Boolean functions and formulas have important applications in several areas of Computer Science.

- Digital electronics and the construction of digital computation devices such as microprocessors, is based on Boolean functions.
- The theory of computation and complexity uses Boolean functions for representing abstract models of computation, for example in terms of Boolean circuits, and investigating their properties, for example the theory of Computational Complexity, where some of the fundamental results like NP-completeness [Coo71, GJ79] were first established with Boolean functions.
- In software engineering and related disciplines, formal logics and automated reasoning methods are used for modeling and analyzing software programs and other complex systems, in order to create software systems and validate them, for example.
- Parts of Artificial Intelligence use Boolean functions and formulas for representing models of the world and the reasoning processes of intelligent systems.

Some of the best known and most primitive Boolean functions are represented by the connectives $\vee$, $\wedge$, $\neg$, $\rightarrow$, $\leftrightarrow$, and $\oplus$ as follows, with the columns marked with α and β giving the input values and the third column the value of the indicated Boolean function with these inputs.

α	β	$\alpha \vee \beta$
0	0	0
0	1	1
1	0	1
1	1	1

α	β	$\alpha \wedge \beta$
0	0	0
0	1	0
1	0	0
1	1	1

α	$\neg\alpha$
0	1
1	0

α	β	$\alpha \rightarrow \beta$
0	0	1
0	1	1
1	0	0
1	1	1

α	β	$\alpha \leftrightarrow \beta$
0	0	1
0	1	0
1	0	0
1	1	1

α	β	$\alpha \oplus \beta$
0	0	0
0	1	1
1	0	1
1	1	0

The connectives are used for forming complex Boolean functions from primitive ones.

Often only some of the connectives, typically $\neg$ and one or both of $\wedge$ and $\vee$, are viewed as *primitive connectives*, and other connectives are defined in terms of the primitive connectives. For example the *implication* $\rightarrow$ and *equivalence* $\leftrightarrow$ connectives can be defined as follows in terms of $\neg$, $\wedge$ and $\vee$: $\alpha \rightarrow \beta$ as $\neg\alpha \vee \beta$, and $\alpha \leftrightarrow \beta$ as $(\alpha \rightarrow \beta) \wedge (\beta \rightarrow \alpha)$.

There is a close connection between the Boolean connectives $\vee$, $\wedge$ and $\neg$ and the operations $\cup$, $\cap$ and complementation in Set Theory. Observe the similarity between the truth-tables for the three connectives and the analogous tables for the set-theoretic operations.

α	β	$\alpha \cup \beta$
$\emptyset$	$\emptyset$	$\emptyset$
$\emptyset$	$\{1\}$	$\{1\}$
$\{1\}$	$\emptyset$	$\{1\}$
$\{1\}$	$\{1\}$	$\{1\}$

α	β	$\alpha \cap \beta$
$\emptyset$	$\emptyset$	$\emptyset$
$\emptyset$	$\{1\}$	$\emptyset$
$\{1\}$	$\emptyset$	$\emptyset$
$\{1\}$	$\{1\}$	$\{1\}$

α	$\overline{\alpha}$
$\emptyset$	$\{1\}$
$\{1\}$	$\emptyset$

Formulas in logic are formed analogously to arithmetic expressions in mathematics, with parentheses, variables (atomic propositions) and binary connectives analogous to the binary arithmetic operations like addition and multiplication.

Definition 2.1 (Propositional Formulas) *Formulas are recursively defined as follows.*

- *Atomic propositions $x \in X$ are formulas.*
- *Symbols $\top$ and $\bot$ are formulas (respectively representing the constant* true *and* false*).*
- *If α and β are formulas, then so are*

  1. $\neg\alpha$,
  2. $(\alpha \wedge \beta)$,
  3. $(\alpha \vee \beta)$,
  4. $(\alpha \rightarrow \beta)$, *and*
  5. $(\alpha \leftrightarrow \beta)$.

Nothing else is a formula.

Parentheses (and) do not always need to be written if precedences between the connectives are observed. The highest precedence is with $\neg$, followed by $\wedge$ and $\vee$, then $\rightarrow$, and finally $\leftrightarrow$. So, $\neg a \vee b$ means the same as $(\neg a) \vee b$.

Associativity rules for $\vee$ and $\wedge$ are usually not paid much attention to, as these connectives are associative: for example, $a \vee (b \vee c)$ is equivalent to $(a \vee b) \vee c$. For implication, one can define $\phi_1 \rightarrow \phi_2 \rightarrow \phi_3$ to mean $\phi_1 \rightarrow (\phi_2 \rightarrow \phi_3)$, but this expression would practically always be parenthesized anyway, so adopting this precedence rule is not important.

2.2 Valuations and Truth-Values

Definition 2.2 (Valuation of atomic propositions) *A* valuation $v : X \rightarrow \{0, 1\}$ *is a mapping from atomic propositions $X = \{x_1, \ldots, x_n\}$ to* truth-values *0 and 1.*

Valuations can be extended to *formulas* $\phi \in \mathcal{L}(X)$, i.e. $v : \mathcal{L}(X) \rightarrow \{0, 1\}$.

Often the term *assignment* or *truth-assignment* is used instead of valuation, but they refer to the same thing.

Definition 2.3 (Valuation of propositional formulas) *A given valuation $v : X \rightarrow \{0, 1\}$ of atomic propositions can be extended to a valuation of arbitrary propositional formulas over X.*

$v(\neg\alpha) = 1$ *iff* $v(\alpha) = 0$
$v(\top) = 1$
$v(\bot) = 0$
$v(\alpha \wedge \beta) = 1$ *iff* $v(\alpha) = 1$ *and* $v(\beta) = 1$
$v(\alpha \vee \beta) = 1$ *iff* $v(\alpha) = 1$ *or* $v(\beta) = 1$
$v(\alpha \rightarrow \beta) = 1$ *iff* $v(\alpha) = 0$ *or* $v(\beta) = 1$ *(or, equivalently,* $v(\alpha) \leq v(\beta)$*)*
$v(\alpha \leftrightarrow \beta) = 1$ *iff* $v(\alpha) = v(\beta)$

Example 2.4 Let $v(a) = 1, v(b) = 1, v(c) = 1, v(d) = 1$. Then

$$\begin{aligned} v(a \vee b \vee c) &= 1, \\ v(\neg a \rightarrow b) &= 1, \\ v(a \rightarrow \neg b) &= 0. \end{aligned}$$

■

We also introduce a commonly used notation for the truth of a formula under a given valuation. We will write $v \models \phi$ if $v(\phi) = 1$, and $v \not\models \phi$ if $v(\phi) = 0$.

Definition 2.5 (Subformulas and immediate subformulas) *We recursively define* immediate subformulas *isfma*(ϕ) *of a formula* ϕ *as follows.*

1. *If* $\phi = x$ *where* x *is an atomic proposition, then isfma*$(\phi) = \emptyset$.
2. *isfma*$(\alpha \wedge \beta) = \{\alpha, \beta\}$
3. *isfma*$(\alpha \vee \beta) = \{\alpha, \beta\}$
4. *isfma*$(\alpha \rightarrow \beta) = \{\alpha, \beta\}$
5. *isfma*$(\alpha \leftrightarrow \beta) = \{\alpha, \beta\}$
6. *isfma*$(\neg\alpha) = \{\alpha\}$

Subformulas *of a formula* ϕ *are defined recursively by sfma*$(\phi) = \{\phi\} \cup \bigcup_{\psi \in isfma(\phi)} sfma(\psi)$.

2.3 Equivalences

Table 2.1 lists equivalences that hold in the propositional logic. Replacing one side of any of these equivalences by the other - in any formula - does not change the Boolean function represented by the formula.

These equivalences have several applications, including translating formulas into *normal forms* (Section 4), and simplifying formulas. For example, all occurrences $\top$ and $\bot$ of the constant symbols (except one, if the whole formula reduces to $\top$ or $\bot$) can be eliminated with the equivalences containing these symbols.

In some applications, and especially in implementations as computer programs, it may be useful to define conjunctions and disjunctions to have more than 2 sub-formulas. These are *chain disjunction* and *chain conjunction*. These can be written as

$$\bigwedge\{\phi_1, \ldots, \phi_n\}$$

and

$$\bigvee\{\phi_1, \ldots, \phi_n\}$$

with $n \geq 0$ having arbitrarily high value. Special cases are $n = 1$, when $\bigwedge\{\phi_1\}$ and $\bigvee\{\phi_1\}$ are both the same as ϕ_1, and $n = 0$, when $\bigwedge\emptyset$ is equivalent to the constant $\top$ (because *all* conjuncts are true, and there are none of them), and $\bigvee\emptyset$ is equivalent to the constant $\bot$ (because for disjunction to be true there should be at least one true disjunct, and in this case there are none.)

2.4 Truth-tables

All valuations relevant to a formula are often tabulated as *truth-tables* so that each row corresponds to a valuation. Truth-tables are used for analyzing the most basic properties of formulas.

double negation	$\neg\neg\alpha$	$\equiv$	α
associativity $\vee$	$\alpha \vee (\beta \vee \gamma)$	$\equiv$	$(\alpha \vee \beta) \vee \gamma$
associativity $\wedge$	$\alpha \wedge (\beta \wedge \gamma)$	$\equiv$	$(\alpha \wedge \beta) \wedge \gamma$
commutativity $\vee$	$\alpha \vee \beta$	$\equiv$	$\beta \vee \alpha$
commutativity $\wedge$	$\alpha \wedge \beta$	$\equiv$	$\beta \wedge \alpha$
distributivity $\wedge$ $\vee$	$\alpha \wedge (\beta \vee \gamma)$	$\equiv$	$(\alpha \wedge \beta) \vee (\alpha \wedge \gamma)$
distributivity $\vee$ $\wedge$	$\alpha \vee (\beta \wedge \gamma)$	$\equiv$	$(\alpha \vee \beta) \wedge (\alpha \vee \gamma)$
idempotence $\vee$	$\alpha \vee \alpha$	$\equiv$	α
idempotence $\wedge$	$\alpha \wedge \alpha$	$\equiv$	α
absorption 1	$\alpha \wedge (\alpha \vee \beta)$	$\equiv$	α
absorption 2	$\alpha \vee (\alpha \wedge \beta)$	$\equiv$	α
De Morgan's law 1	$\neg(\alpha \vee \beta)$	$\equiv$	$(\neg\alpha) \wedge (\neg\beta)$
De Morgan's law 2	$\neg(\alpha \wedge \beta)$	$\equiv$	$(\neg\alpha) \vee (\neg\beta)$
contraposition	$\alpha \to \beta$	$\equiv$	$\neg\beta \to \neg\alpha$
negation $\top$	$\neg\top$	$\equiv$	$\bot$
negation $\bot$	$\neg\bot$	$\equiv$	$\top$
constant $\bot$	$\alpha \wedge \neg\alpha$	$\equiv$	$\bot$
constant $\top$	$\alpha \vee \neg\alpha$	$\equiv$	$\top$
elimination $\top$ $\vee$	$\top \vee \alpha$	$\equiv$	$\top$
elimination $\top$ $\wedge$	$\top \wedge \alpha$	$\equiv$	α
elimination $\bot$ $\vee$	$\bot \vee \alpha$	$\equiv$	α
elimination $\bot$ $\wedge$	$\bot \wedge \alpha$	$\equiv$	$\bot$
elimination $\bot$ $\to$	$\bot \to \alpha$	$\equiv$	$\top$
elimination $\bot$ $\to$	$\alpha \to \bot$	$\equiv$	$\neg\alpha$
elimination $\top$ $\to$	$\top \to \alpha$	$\equiv$	α
elimination $\top$ $\to$	$\alpha \to \top$	$\equiv$	$\top$
commutativity $\leftrightarrow$	$\alpha \leftrightarrow \beta$	$\equiv$	$\beta \leftrightarrow \alpha$
elimination $\top$ $\leftrightarrow$	$\top \leftrightarrow \alpha$	$\equiv$	α
elimination $\bot$ $\leftrightarrow$	$\bot \leftrightarrow \alpha$	$\equiv$	$\neg\alpha$

Table 2.1: Propositional Equivalences

Valuations for formulas containing exactly one connective are as follows.

α	$\neg\alpha$
0	1
1	0

α	β	$\alpha \wedge \beta$
0	0	0
0	1	0
1	0	0
1	1	1

α	β	$\alpha \vee \beta$
0	0	0
0	1	1
1	0	1
1	1	1

α	β	$\alpha \rightarrow \beta$
0	0	1
0	1	1
1	0	0
1	1	1

α	β	$\alpha \leftrightarrow \beta$
0	0	1
0	1	0
1	0	0
1	1	1

Truth-tables for more complex formulas ϕ are constructed as follows.

1. Create columns for all subformulas sfma(ϕ), with columns for shorter formulas left of the bigger formulas. The shortest subformulas are the n atomic propositions occurring in ϕ.
2. Create 2^n rows corresponding to all the valuations of the atomic propositions.
3. Fill in truth-values in the remaining cells from left to right: every row in the column for a formula ψ is filled in by looking at the contents of the immediate subformulas isfma(ψ) in the same row (residing to the left), based on the truth table for the outermost connective in ψ.

Example 2.6 The truth-table for $\neg B \wedge (A \rightarrow B)$ is the following.

A	B	$\neg B$	$(A \rightarrow B)$	$(\neg B \wedge (A \rightarrow B))$
0	0	1	1	1
0	1	0	1	0
1	0	1	0	0
1	1	0	1	0

■

2.5 Logical Consequence, Satisfiability, Validity

Once we have represented some facts of whatever object or system we are analyzing as a formula, we are often interested in understanding its properties.

A formula is *satisfiable* if it is *true* under some valuation of its atomic propositions. For example, if our formula described the properties of some system or some scenario, the formula being satisfiable would mean that the scenario is possible.

A formula is a *logical consequence* of another formula, if it is necessarily true when the other formula is true. In English we would say that some statement *follows from* another statement. For example, from "It is Tuesday" it follows that "It is Monday or it is Tuesday". If the first sentence is true, then necessarily also the second sentence is true. Note that there does not have to be any real connection between the two statements: "It is Monday or it is not Monday" is a logical consequence of any statement. We similarly defined logical consequence from a *set* of formulas, viewing the set as the conjunction of all the formulas in it.

A formula is *valid* (a *tautology*) if it is true no matter what the values of the atomic propositions are, similarly to basic mathematical equalities everybody is familiar with, such as $2x = x + x$ which holds no matter which values we choose for x.

Next we formally define these concepts.

Definition 2.7 (Validity) *A formula ϕ is* valid *if and only if $v \models \phi$ for all valuations v.*

Example 2.8 $x \vee \neg x$ is valid.

$x \rightarrow (x \vee y)$ is valid.

$x \vee y$ is not valid. ■

Definition 2.9 (Satisfiability) *A formula ϕ is* satisfiable *if and only if there is at least one valuation v such that $v \models \phi$. A set $\{\phi_1, \ldots, \phi_n\}$ is* satisfiable *if there is at least one valuation v such that $v \models \phi_i$ for all $i \in \{1, \ldots, n\}$.*

	$\alpha \wedge \beta$	$\models$	α
	α	$\models$	$\alpha \vee \beta$
	α	$\models$	$\beta \rightarrow \alpha$
	$\neg\alpha$	$\models$	$\alpha \rightarrow \beta$
Modus Ponens	$\alpha \wedge (\alpha \rightarrow \beta)$	$\models$	β
	$\neg\beta \wedge (\alpha \rightarrow \beta)$	$\models$	$\neg\alpha$

Table 2.2: Useful Logical Consequences

The best known result in computational complexity theory is that the propositional satisfiability problem SAT is NP-complete. SAT is defined as decision problem of deciding whether a given formula ϕ is satisfiable, often stated as $\phi \in$ SAT. The closely related validity problem (TAUT) is similarly hard (co-NP-complete.)

Example 2.10 The following formulas are satisfiable: x, $x \wedge y$, and $x \vee \neg x$. ■

Example 2.11 The following formulas are not satisfiable: $x \wedge \neg x$, $\bot$, $a \wedge (a \rightarrow b) \wedge \neg b$. ■

A satisfiable formula is also said to be *consistent*. A formula that is not satisfiable is *unsatisfiable*, *inconsistent*, or *contradictory*.

Satisfiability means *logically possible*: it is possible that the formula is true (if the truth-values of the atomic propositions in the formula are chosen right.)

Definition 2.12 (Logical Consequence) *A formula ϕ is a* logical consequence *of $\Sigma = \{\phi_1, \ldots, \phi_n\}$, denoted by $\Sigma \models \phi$, if and only if for all valuations v, if $v \models \Sigma$ then $v \models \phi$.*

Notice that we are here using the symbol $\models$ for a completely different purpose than the one we first introduced it for in Section 2.2. With logical consequence $\Sigma \models \phi$ we have a set of formulas on the left-hand side, whereas when talking about the truth of a formula ϕ in valuation v we have a valuation on the left-hand side of $v \models \phi$.

If we talk about logical consequence from a single formula so that $\Sigma = \{\psi\}$ for some formula ψ, then we often write this as $\psi \models \phi$.

Logical consequences ϕ of Σ are what can be inferred with certainty from Σ: if Σ holds, then ϕ is guaranteed to hold too.

Example 2.13 From "if it is a weekday today then most people are working today", and "it is a weekday today" it follows that "most people are working today". This can be expressed as $\{w \rightarrow p, w\} \models p$. ■

Example 2.14 $\{a, a \rightarrow b, b \rightarrow c\} \models c$ ■

Example 2.15 $\{m \vee t, m \rightarrow w, t \rightarrow w\} \models w$ ■

Some useful logical consequences are given in Table 2.2.

Notice two special cases of logical consequence $\phi_1 \models \phi_2$ which may be unintuitive. If ϕ_2 is valid, then $\phi_1 \models \phi_2$ holds for any formula ϕ_1. For example, $a \models b \vee \neg b$. Similarly, if ϕ_1 is unsatisfiable, then $\phi_1 \models \phi_2$ holds for any formula ϕ_2. One sometimes says that "from a contradiction anything follows". This is the formal counterpart of that. For example, $a \wedge \neg a \models b$. The unintuitive thing in these cases may be that the formulas on the different sides of $\models$ do not share any atomic formulas, and hence there isn't any real connection between them.

Definition 2.16 (Logical Equivalence) *A formula ϕ_1 is a* logically equivalent *to formula ϕ_2, denoted by $\phi_1 \equiv \phi_2$, if and only if for all valuations v, $v(\phi_1) = v(\phi_2)$.*

Lemma 2.17 *$\phi_1 \equiv \phi_2$ if and only if both $\phi_1 \models \phi_2$ and $\phi_2 \models \phi_1$.*

Importantly, there are close connections between satisfiability, validity, and logical consequence. These connections allow *reducing* questions concerning these concepts to each other, which allows choosing one of these concepts as the most basic one (for example implemented in algorithms for reasoning in the propositional logic), and reducing the other concepts to the basic one.

Theorem 2.18 (Validity vs. Logical Consequence 1) *The formula ϕ is valid if and only if $\emptyset \models \phi$.*

Theorem 2.19 (Validity vs. Logical Consequence 2) $\{\phi_1, \ldots, \phi_n\} \models \phi$ *if and only if* $(\phi_1 \wedge \cdots \wedge \phi_n) \rightarrow \phi$ *is valid.*

Theorem 2.20 (Validity vs. Satisfiability) ϕ *is* valid *if and only if* $\neg\phi$ *is not satisfiable.*

Theorem 2.21 (Logical Consequence vs. Satisfiability) $\Sigma \models \phi$ *if and only if* $\Sigma \cup \{\neg\phi\}$ *is not satisfiable.*

In practice, algorithms for reasoning in the propositional logic implement *satisfiability*. Other reasoning tasks are reduced to satisfiability testing as shown by Theorems 2.20 and 2.21.

Chapter 3

Reasoning in the Propositional Logic

In this chapter we discuss the main methods for automating inference in the propositional logic.

The simplest method is based on truth-tables (Section 3.1), which enumerate all valuations of the relevant atomic propositions, and questions about satisfiability and logical consequence can be answered by evaluating the truth-values of the relevant formulas for every valuation.

Truth-tables are easy to implement as a program, but are impractical when the number of atomic propositions is higher than 20 or 30. The algorithms used in practice (Section 3.2) are based on searching a tree in which each path starting from the root node represents a (partial) valuation of the atomic propositions. The size of this tree is in practice orders of magnitudes smaller than corresponding truth-tables, and for finding one satisfying valuation not the whole tree needs to be traversed. Finally, only one path of the tree, from the root node to the current leaf node, needs to be kept in the memory at a time, which reduces the memory requirements substantially. This enables the use of these algorithms for formulas of the size encountered in solving important problems in AI and computer science in general, with up to hundreds of thousands of atomic propositions and millions of clauses.

3.1 Truth-Tables

The most basic method for testing satisfiability of a formula ϕ is to construct the truth-table for ϕ, representing all valuations of atomic propositions occurring in ϕ, and then check whether the column for ϕ contains at least one 1: ϕ is satisfiable if and only if the column for ϕ contains at least one 1. Obviously, this is because all possible valuations are represented in the truth-table, and a formula is satisfiable if and only if it is true in at least one valuation.

Example 3.1 The truth-table for $\neg B \wedge (A \rightarrow B)$:

A	B	$\neg B$	$(A \rightarrow B)$	$(\neg B \wedge (A \rightarrow B))$
0	0	1	1	**1**
0	1	0	1	0
1	0	1	0	0
1	1	0	1	0

$\neg B \wedge (A \rightarrow B)$ is satisfiable because it is true when both A and B are false, corresponding to the first row. ■

The second important problem is testing for logical consequence.

Algorithm 3.2 *To test whether ϕ a logical consequence of Σ do the following.*

1. *Construct the truth-table for ϕ and Σ.*
2. *Mark every row in which the columns for all members of Σ contain 1.*
3. *Check that there is 1 in the column for ϕ for every marked row.*

This algorithm tests whether ϕ is true in every valuation in which Σ is true.

If there is marked row with ϕ false, then we have *a counterexample* to $\Sigma \models \phi$: Σ is true but ϕ is false.

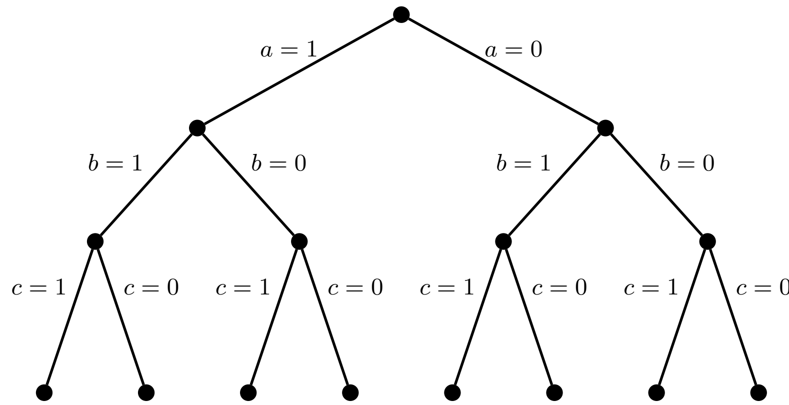


Figure 3.1: All valuations as a binary tree

Example 3.3 Test $\{a \to b, b \to c\} \models a \to c$:

a	b	c	$a \to b$	$b \to c$	$a \to c$
0	0	0	**1**	**1**	1
0	0	1	**1**	**1**	1
0	1	0	1	0	1
0	1	1	**1**	**1**	1
1	0	0	0	1	0
1	0	1	0	1	1
1	1	0	1	0	0
1	1	1	**1**	**1**	1

$\{a \to b, b \to c\}$ true on 4 rows, and we need to confirm that $a \to c$ is true on those rows. ■

Truth-tables are a practical method to be used by hand only up to a couple of atomic propositions. With 6 atomic propositions there are 64 rows, which barely fits on a sheet of paper. With computer it is possible to use the truth-table method for up to 20 or 25 variables. After that memory consumption and computation times will be impractically high. In many applications the number of atomic propositions is in the hundreds, thousands, or even hundreds of thousands, and completely different types of algorithms are necessary.

The exponentiality in the truth-table is inherent in all algorithms for testing satisfiability or logical consequence, similarly to many of challenging problems in AI. For testing satisfiability, and many other problems, this exponentiality is an indication of NP-completeness [Coo71, GJ79]. The exponentiality is known as the *combinatorial explosion*, and is caused by the exponential number 2^n of combinations of values of n atomic propositions. NP-completeness of the satisfiability problem suggests that all algorithms for the problem necessarily need to do an exponential amount of computation, in the worst case, and the problem is *inherently hard* in this sense. Work on practically useful algorithms for the satisfiability problem try to avoid the exponential worst-case by using methods for pruning the search tree, and by using heuristics for choosing the traversal order and a tree structure that minimizes the time it takes to find a satisfying valuation or to determine that none exist.

3.2 Tree-Search Algorithms

More practical algorithms for solving the satisfiability problem do not explicitly go through all possible valuations of the atomic propositions, and do not store all of them in a big table that has to be kept in the memory.

The simplest algorithm for testing satisfiability that does not construct a truth-table does a depth-first traversal of a binary tree that represents all valuations. This is illustrated in Figure 3.1 for atomic propositions $X = \{a, b, c\}$.

During the traversal, this algorithm only needs to store in the memory the (partial) valuation on the path from the root node to the current node, and hence its memory consumption is linear in the number of atomic propositions.

The runtime of the algorithm, however, is still exponential in the number of atomic propositions for all unsatisfiable formulas, because of the size of the tree. For satisfiable formulas the whole tree does not need to be traversed, but since the first satisfying valuation may be encountered late in the traversal, in practice this algorithm is also usually exponential.

The improvements to this basic tree-search algorithm are based on the observation that many of the (partial) valuations in the tree make the formula false, and this is easy to detect during the traversal. Hence large parts of the search tree can be pruned.

In the following, we assume that the formula has been translated into the conjunctive normal form (Section 4). Practically all leading algorithms assume the input to be in CNF.

The fundamental observation in pruning the search tree is the following. Let v be a valuation and $l \vee l_1 \vee l_2 \vee \cdots \vee l_n$ a clause such that $v(l_1) = 0, \ldots, v(l_n) = 0$,

Since we are trying to find a valuation that makes our clause set *true*, this clause has to be *true* (similarly to all other clauses), and it can only be *true* if the literal l is *true*.

Hence if the partial valuation corresponding to the current node in the search tree makes, in some clause, all literals *false* except one (call it l) that still does not have a truth-value, then l has to be made true. If l is an atomic proposition x, then the current valuation has to assign $v(x) = 1$, and if $l = \neg x$ for some atomic proposition x, then we have to assign $v(x) = 0$. We call these *forced assignments*.

Consider the search tree in Figure 3.1 and the clauses $\neg a \vee b$ and $\neg b \vee c$. The leftmost child node of the root node, which is reached by following the arc $a = 1$, corresponds to the partial valuation $v = \{(a, 1)\}$ that assigns $v(a) = 1$ and leaves the truth-values of the remaining atomic propositions open. Now with clause $\neg a \vee b$ we have the forced assignment $v(b) = 1$. Now, with clause $\neg b \vee c$ also $v(c) = 1$ is a forced assignment.

Hence assigning $a = 1$ forces the assignments of both of the remaining variables. This means that the leftmost subtree of the root node essentially only consists of one path that leads to the leftmost leaf node. The other leaf nodes in the subtree, corresponding to valuations $\{a = 1, b = 1, c = 0\}$, $\{a = 1, b = 0, c = 0\}$, and $\{a = 1, b = 0, c = 1\}$, would not be visited, because it is obvious that they falsify at least one of the clauses. Since we are doing the tree search in order to determine the satisfiability of our clause set, the computation in the subtree with $a = 1$ can be stopped here, continuing from the root node with $a = 0$ to the rightmost subtree.

3.2.1 Unit Resolution

What we described as forced assignments is often formalized as the inference rule Unit Resolution.

Theorem 3.4 (Unit Resolution) *In*

$$\frac{l \quad \bar{l} \vee l_1 \vee l_2 \vee \cdots \vee l_n}{l_1 \vee l_2 \vee \cdots \vee l_n}$$

the formula below the line is a logical consequence of the formulas above the line.

Here the *complementation* $\bar{l}$ of a literal l is defined by $\overline{x} = \neg x$ and $\overline{\neg x} = x$.

Example 3.5 From $\{A \vee B \vee C, \neg B, \neg C\}$ one can derive A by two applications of the Unit Resolution rule. ■

A special case of the Unit Resolution rule is when both clauses are unit clauses (consisting of one literal only.) Since the $n = 0$ in this case, the formula below the line has 0 disjuncts. We identify a chain-disjunction with 0 literals with the constant false $\bot$, corresponding to the *empty clause*.

$$\frac{l \quad \bar{l}}{\bot}$$

Any clause set with the empty clause is *unsatisfiable*.

The Unit Resolution rule is a special case of the Resolution rule.

Theorem 3.6 (Resolution) *In*

$$\frac{l \vee l'_1 \vee \cdots \vee l'_m \quad \bar{l} \vee l_1 \vee \cdots \vee l_n}{l'_1 \vee \cdots \vee l'_m \vee l_1 \vee l_2 \vee \cdots \vee l_n}$$

1: **procedure** DPLL(S)
2: $S := \mathit{UP}(S)$;
3: **if** $\bot \in S$ **then return** false;
4: $x :=$ any atomic proposition such that $\{x, \neg x\} \cap S = \emptyset$;
5: **if** no such x exists **then return** true;
6: **if** DPLL($S \cup \{x\}$) **then return** true;
7: **return** DPLL($S \cup \{\neg x\}$);

Figure 3.2: The DPLL procedure

the formula below the line is a logical consequence of the formulas above the line.

The resolution rule can be use as a part of *complete* decision procedures for the satisfiability problem of the propositional logic: apply the resolution rule exhaustively in all possible ways until no more new clauses are produced. If the empty clause $\emptyset$ was produced, then the clause set is unsatisfiable, and otherwise it is satisfiable.

The resolution rule used in this way is, however, in general not a practical decision method, because the number of clauses generated is often exponential in the size of the original clause set, quickly leading to the exhaustion of the available memory for any slightly bigger clause set.

Instead of the resolution rule in its full generality, practical decision procedures for the SAT problem use unit resolution (see Section 3.2.4 for the DPLL procedure), and use the resolution rule in a more restricted way that does not blindly generate huge numbers of clauses, as in the Conflict-Drive Clause Learning (CDCL) procedure [MSS96].

3.2.2 Subsumption

If a clause set contains two clauses with literals $\{l_1, \ldots, l_n\}$ and $\{l_1, \ldots, l_n, l_{n+1}, \ldots, l_m\}$, respectively, then the latter clause can be removed, as it is a logical consequence of the former. The two clause sets, with and without the second clause, are *true* in exactly the same valuations.

Example 3.7 Applying the Subsumption rule to $\{A \vee B \vee C, A \vee C\}$ yields the equivalent set $\{A \vee C\}$. ■

If the shorter clause is a unit clause, then this is called Unit Subsumption.

3.2.3 Unit Propagation

Performing all unit resolutions exhaustively leads to the Unit Propagation procedure, also known as Boolean Constraint Propagation (BCP).

In practice, when a clause $l_1 \vee \cdots \vee l_n$ with $n > 2$ can be resolved with a unit clause, the shorter clause of length $n - 1 > 1$ would not be useful, so, instead of producing these $n - 1$ clauses, of lengths $1, 2, \ldots, n - 1$, unit propagation attempts to do the same useful inferences of unit clauses only, without producing the longer intermediate clauses: only produce a new clause when the complements of all but 1 of the n literals of the clause have been inferred. This can be formalized as the following rule, with $n \geq 2$.

$$\frac{\overline{l_1} \;\; \overline{l_2} \;\; \overline{l_3} \;\; \ldots \;\; \overline{l_{n-1}} \quad l_1 \vee \cdots \vee l_n}{l_n}$$

We denote by $\mathit{UP}(S)$ the clause set obtained by performing all possible unit propagations to a clause set S, followed by applying the subsumption rule.

3.2.4 The Davis-Putnam Procedure

The Davis-Putnam-Logemann-Loveland procedure (DPLL) [DLL62], often known simply as the Davis-Putnam procedure, is given in Figure 3.2.

We have sketched the algorithm so that the first branch assigns the chosen atomic proposition x true in the first subtree (line 6) and false in the second (line 7). However, these two branches can be traversed in either order (exchanging x and $\neg x$ on lines 6 and 7), and efficient implementations use this freedom by using powerful heuristics for choosing first the atomic proposition (line 4) and then choosing its truth-value.

Theorem 3.8 *Let S be a set of clauses. Then DPLL(S) returns* true *if and only if S is satisfiable.*

References

Automation of reasoning in the classical propositional logic started in the works of Davis and Putnam and their co-workers, resulting in the Davis-Putnam-Logemann-Loveland procedure [DLL62] (earlier known as the Davis-Putnam procedure) and other procedures [DP60].

Efforts to implement the Davis-Putnam procedure efficiently lead to fast progress from mid-1990s on.

The current generation of solvers for solving the SAT problem are mostly based on the Conflict-Driven Clause Learning (CDCL) procedure which emerged from the work of Marques-Silva and Sakallah [MSS96]. Many of the current leading implementation techniques and heuristics for selecting the decision variables come from the work on the Chaff solver [MMZ$^+$01]. Several other recent developments have strongly contributed to current solvers [PD07, AS09].

Unit propagation can be implemented to run in linear time in the size of the clause set, which in the simplest variants involves updating a counter every time a literal in a clause gets a truth-value [DG84]. Modern SAT solvers attempt to minimize memory accesses (cache misses), and even counter-based linear time procedures are considered too expensive, and unit propagation schemes that do not use counters are currently used [MMZ$^+$01]. The above developments for the CDCL procedure have made it possible to solve very large satisfiability problems efficiently, with up to millions of atomic propositions and tens or even hundreds of millions of clauses.

Also stochastic local search algorithms have been used for solving the satisfiability problem [SLM92], but they are not used in many applications because of their inability to detect unsatisfiability.

The algorithms in this chapter assumed the formulas to be in CNF. As discussed in Section 4, the translation into CNF may in some cases exponentially increase the size of the formula. However, for the purposes of satisfiability testing, there are transformations to CNF that are of linear size and preserve satisfiability (but not logical equivalence.) These transformations are used for formulas that are not already in CNF [Tse68, CMV09].

Chapter 4

Normal Forms and Logic Data Structures

Arbitrary propositional formulas can be translated into syntactically restricted forms which are still capable of expressing all Boolean functions. Use of such normal forms can serve two main purposes. First, it may be more straightforward to define algorithms and inference methods for formulas of a simple form. The *resolution rule* (Section 3.2.1) is an example of this. Second, the process of translating a formula into certain normal form does much of the work in solving important computational problems related to propositional formulas. For example, an answer to the question of whether a formula is valid is obtained as a by-product of translating the formula normal forms such as DNF or OBDD. A number of other operations on propositional formulas are in practice much more efficient when the formulas are in certain normal forms rather than unrestricted propositional formulas.

In this chapter we present some of the best known normal forms, including the disjunctive and conjunctive normal forms, the negation normal form, as well as binary decision diagrams.

Definition 4.1 (Literals) *If x is an atomic proposition, then x and $\neg x$ are* literals.

Definition 4.2 (Clauses) *If $l_1, \dots, l_n$ are literals, then $l_1 \vee \cdots \vee l_n$ is a* clause.

Definition 4.3 (Terms) *If $l_1, \dots, l_n$ are literals, then $l_1 \wedge \cdots \wedge l_n$ is a* term.

Definition 4.4 (Conjunctive Normal Form) *If $\phi_1, \dots, \phi_m$ are clauses, then the formula $\phi_1 \wedge \cdots \wedge \phi_m$ is in* conjunctive normal form.

Example 4.5 The following formulas are in conjunctive normal form.

$$\begin{array}{l} \neg x \\ \neg x_1 \vee \neg x_2 \\ x_3 \wedge x_4 \\ (\neg x_1 \vee \neg x_2) \wedge \neg x_3 \wedge (x_4 \vee \neg x_5) \end{array}$$

■

Table 4.1 lists rules that transform any formula into CNF. These rules are instances of the equivalence listed in Table 2.1

Algorithm 4.6 *Any formula can be translated into conjunctive normal form as follows.*

1. *Eliminate connective $\leftrightarrow$ (Rule 4.1).*
2. *Eliminate connective $\rightarrow$ (Rule 4.2).*
3. *Move $\neg$ inside $\vee$ and $\wedge$ (Rules 4.3, 4.4 and 4.5), also eliminating double negations.*
4. *Move $\wedge$ outside $\vee$ (Rules 4.6 and 4.7).*

Notice that the last step of the transformation multiplies the number of copies of subformulas α and γ. For some classes of formulas this transformation therefore leads to exponentially big normal forms.

$$\alpha \leftrightarrow \beta \rightsquigarrow (\neg\alpha \vee \beta) \wedge (\neg\beta \vee \alpha) \tag{4.1}$$
$$\alpha \rightarrow \beta \rightsquigarrow \neg\alpha \vee \beta \tag{4.2}$$
$$\neg(\alpha \vee \beta) \rightsquigarrow \neg\alpha \wedge \neg\beta \tag{4.3}$$
$$\neg(\alpha \wedge \beta) \rightsquigarrow \neg\alpha \vee \neg\beta \tag{4.4}$$
$$\neg\neg\alpha \rightsquigarrow \alpha \tag{4.5}$$
$$\alpha \vee (\beta \wedge \gamma) \rightsquigarrow (\alpha \vee \beta) \wedge (\alpha \vee \gamma) \tag{4.6}$$
$$(\alpha \wedge \beta) \vee \gamma \rightsquigarrow (\alpha \vee \gamma) \wedge (\beta \vee \gamma) \tag{4.7}$$

Table 4.1: Rewriting Rules for Translating Formulas into Conjunctive Normal Form

Formulas in CNF are often represented as sets of clauses (rather than a conjunction of clauses), with clauses represented as sets of literals. The placement of connectives $\vee$ and $\wedge$ can be left implicit because of the simple structure of CNF. The CNF $(A \vee B \vee C) \wedge (\neg A \vee D) \wedge E$ can be represented as $\{A \vee B \vee C,\ \ \neg A \vee D,\ \ E\}$ or, equivalently, as $\{\{A, B, C\}, \{\neg A, D\}, \{E\}\}$. Any clause set $\{\emptyset, \ldots\}$ that contains the empty clause $\emptyset$ is logically equivalent to $\bot$, and the empty clause set $\emptyset$ is equivalent to $\top$.

Example 4.7 Translate $A \vee B \rightarrow (B \leftrightarrow C)$ into CNF.

$\rightsquigarrow\ \ A \vee B \rightarrow (B \rightarrow C) \wedge (C \rightarrow B)$
$\rightsquigarrow\ \ \neg(A \vee B) \vee ((\neg B \vee C) \wedge (\neg C \vee B))$
$\rightsquigarrow\ \ (\neg A \wedge \neg B) \vee ((\neg B \vee C) \wedge (\neg C \vee B))$
$\rightsquigarrow\ \ (\neg A \vee ((\neg B \vee C) \wedge (\neg C \vee B))) \wedge (\neg B \vee ((\neg B \vee C) \wedge (\neg C \vee B)))$
$\rightsquigarrow\ \ (\neg A \vee \neg B \vee C) \wedge (\neg A \vee \neg C \vee B) \wedge (\neg B \vee ((\neg B \vee C) \wedge (\neg C \vee B)))$
$\rightsquigarrow\ \ (\neg A \vee \neg B \vee C) \wedge (\neg A \vee \neg C \vee B) \wedge (\neg B \vee \neg B \vee C) \wedge (\neg B \vee \neg C \vee B)$

■

Another often used normal form is the Disjunctive Normal Form (DNF).

Definition 4.8 (Disjunctive Normal Form) *If $\phi_1, \ldots, \phi_m$ are terms, then the formula $\phi_1 \vee \cdots \vee \phi_m$ is in* disjunctive normal form.

Disjunctive normal forms are found with a procedure similar to that for CNF. The only difference is that a different set of distributivity rules is applied, for moving $\wedge$ inside $\vee$ instead of the other way round.

A less restrictive normal form that includes both CNF and DNF is the Negation Normal Form (NNF). It is useful in many types of automated processing of formulas because of its simplicity and the fact that translation into NNF only minimally affects the size of a formula, unlike the translations into DNF and CNF which may increase the size exponentially.

Definition 4.9 (Negation Normal Form) *Let X be a set of atomic propositions.*

1. *$\bot$ and $\top$ are in negation normal form.*
2. *x and $\neg x$ for any $x \in X$ are in negation normal form.*
3. *$\phi_1 \wedge \phi_2$ is in negation normal form if ϕ_1 and ϕ_2 both are.*
4. *$\phi_1 \vee \phi_2$ is in negation normal form if ϕ_1 and ϕ_2 both are.*

No other formula is in negation normal form.

In other words, ϕ is in NNF if and only if it only contains the connectives $\neg$, $\vee$ and $\wedge$, and $\neg$ only occurs right in front of atomic propositions.

After performing the first three steps in the translation into CNF, a formula is in NNF. Notice that only the application of the distributivity equivalences increases the number of atomic propositions in the CNF translation. Hence the translation into NNF does not affect the number of occurrences of atomic propositions or the total number of connectives $\vee$ and $\wedge$. Therefore the size of the resulting formula can be bigger only because of the

higher number of negation symbols $\neg$, and there cannot be more of $\neg$ symbols than there are occurrences of atomic propositions.

4.1 Complexity of Validity and Satisfiability for DNF and CNF

Lemma 4.10 *A formula in CNF is valid if and only if for every clause in the formula, there is some atomic proposition x such that both x and $\neg x$ are in the clause.*

Lemma 4.11 *A formula in DNF is satisfiable if and only if at least one of its terms does not contain both x and $\neg x$ for any atomic proposition x.*

So, for formulas in CNF, testing for validity is a simple syntactic operation that can be done in polynomial time, and for formulas in DNF, satisfiability testing is similarly a polynomial time operation. These operations for arbitrary propositional formulas are far harder, co-NP-complete and NP-complete, respectively. For NNF these operations are similarly hard.

While validity for CNF is easy, testing satisfiability is still NP-complete. Similarly, validity for DNF is co-NP-complete.

Constructing a CNF or a DNF for an unrestricted propositional formula may take exponential time (due to the doubling of size of subformulas when applying the distributivity rules), but as a result, some of the computational problems are exponentially easier.

Validity and satisfiability for NNF are respectively co-NP-complete and NP-complete, because DNF and CNF are special cases of NNF.

4.2 Formulas as a Data Structure

Traditional application of logic is as a language of representing data and knowledge and for deductively making inferences from it.

A different use of logic emerged in the 1980s [Bry86, CBM90, Bry92], when logical formulas were used as a data structure for representing sets (of valuations) and relations (on valuations). As will be discussed in Section 5.1, many logical operations have set-theoretic counterparts, for example union can be identified with disjunction. It turned out that logic-based data structures could scale up to much bigger sets and relations than conventional (enumerative) data structures (trees, lists, arrays, and so on.)

To use formulas as a data structure, we need operations for constructing formulas, for manipulating formulas, and for asking questions about formulas.

Constructing formulas:	
Negation:	Given ϕ, form $\neg\phi$
Conjunction:	Given ϕ_1 and ϕ_2, form $\phi_1 \wedge \phi_2$
Disjunction:	Given ϕ_1 and ϕ_2, form $\phi_1 \vee \phi_2$

Manipulating formulas:	
Substitution:	Given ϕ, construct $\phi[\psi/x]$ by replacing occurrences of x by ϕ
Existential Abstraction:	Given ϕ, construct $\exists x.\phi = \phi[\top/x] \vee \phi[\bot/x]$
Universal Abstraction:	Given ϕ, construct $\forall x.\phi = \phi[\top/x] \wedge \phi[\bot/x]$

A commonly occurring special case of Substitution is Renaming, replacing occurrences of atomic propositions by other atomic propositions.

The abstraction operations are central in implementing relational projection operations (Section 5.2) and multiplication of Boolean matrices.

Analyzing formulas:	
Satisfiability:	Given ϕ, is ϕ satisfiable?
Validity:	Given ϕ, is ϕ valid?
Logical Consequence:	Given ϕ_1 and ϕ_2, does $\phi_1 \models \phi_2$ hold?
Model Counting:	Given ϕ over X, for how many valuations $v : X \to \{0, 1\}$ does $v \models \phi$ hold?

We have already seen that with some normal forms, some of these operations are easier than with arbitrary propositional formulas. For unlimited formulas, the construction operations are efficient, but many of the other operations in turn are not. Similarly, for limited normal forms of propositional formulas, some of the construction operations can be expensive, but as soon as the normal form has been constructed, some of the other operations are far easier than with general propositional formulas.

Interestingly, for some normal forms, although their construction can be expensive (exponential in the worst case), the cheaper other operations have made them far preferable to general propositional logic.

A case in point is the abstraction operations, which are central in relational projections (Section 5.2), and for which no good algorithms exist for general propositional formulas, but which can often be very effectively computed for example with Binary Decision Diagrams (see next section).

4.3 Binary Decision Diagrams

Any propositional formula can be turned to a case analysis on a single atomic proposition by using the following equivalence-preserving transformation (also known as *Shannon expansion*.)

Theorem 4.12 (Boole's Expansion) *For any propositional formula ϕ over X and any atomic proposition $x \in X$, the formula $(x \wedge \phi[\top/x]) \vee (\neg x \wedge \phi[\bot/x])$ is logically equivalent to ϕ.*

One can also express this in terms of the ternary Boolean connective ITE (or IF-THEN-ELSE)

$$\text{ITE}(x, \phi_1, \phi_2) = (x \wedge \phi_1) \vee (\neg x \wedge \phi_2)$$

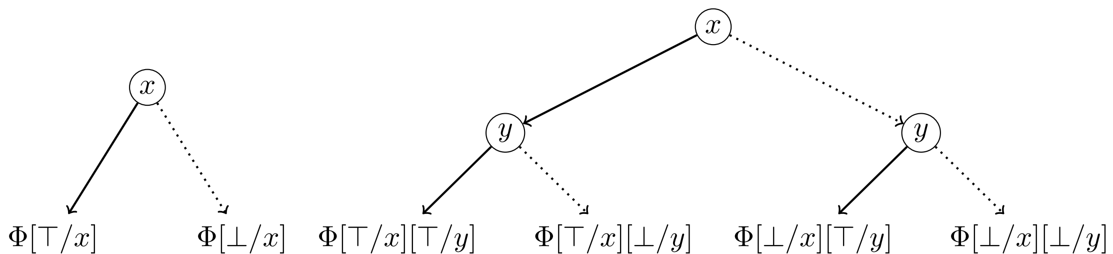
as

$$\phi \equiv \text{ITE}(x, \phi[\top/x], \phi[\bot/x]).$$

The special form of the formulas resulting from the expansion operation gives rise to a graphical representation of any propositional formula: at the topmost level, the formula represents a case analysis on the value of one atomic proposition – either x is true, or x is false – and depending on the value of x, either $\phi[\top/x]$ or $\phi[\bot/x]$ equals the value of the original formula. And, neither of these latter formulas contain occurrences of x.

The binary decision diagram construction proceeds by eliminating x first, and the eliminating further variables from the subformulas $\phi[\top/x]$ and $\phi[\bot/x]$.

The idea is that after translating ϕ to $(x \wedge \phi[\top/x]) \vee (\neg x \wedge \phi[\bot/x])$, we continue applying Boole's expansion to the subformulas $\phi[\top/x]$ and $\phi[\bot/x]$ for other atomic propositions.

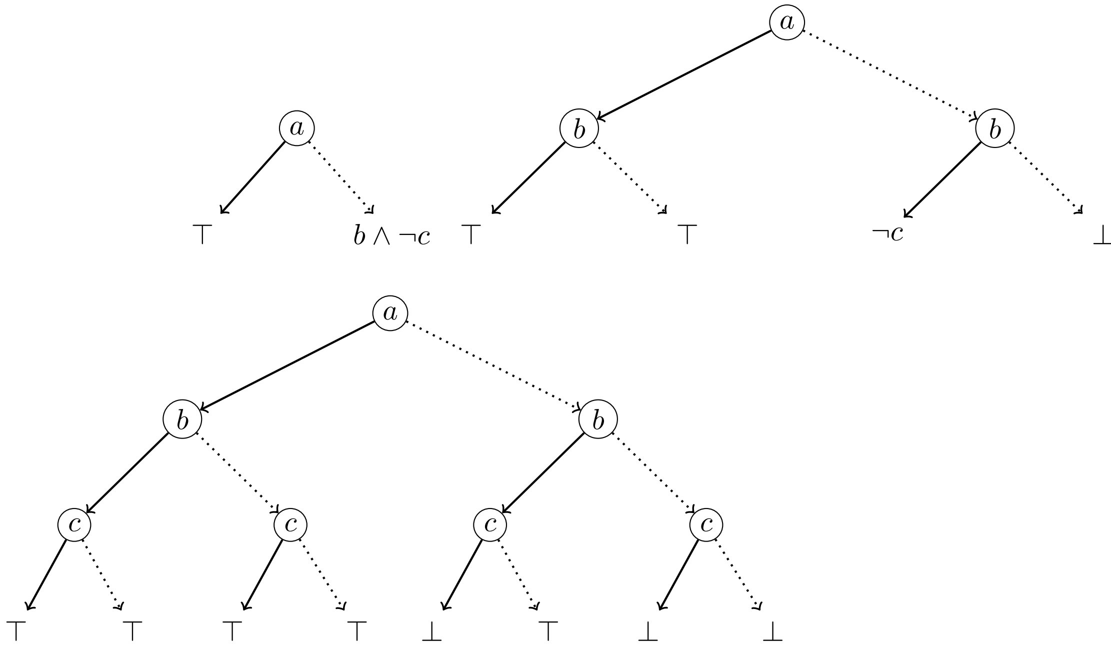


The dashed line is followed when the atomic proposition in the node is *false*, and the non-dashed one when it is *true*.

Doing this systematically for any formula with n atomic propositions yields a binary tree with 2^n leaf nodes labelled with formulas that consist of the constants $\top$ and $\bot$ and connectives only, without any atomic propositions.

Our first observation is that the formulas in the leaves can always be simplified to $\top$ or $\bot$. At this point, we essentially have a graphical representation of the truth table constructed for the n variables: for any valuation of the atomic propositions we can determine the corresponding truth-value of the formula by following the path from from root that correspond to the valuation, and reading the truth-value from the resulting leaf node.

Example 4.13 If we do this for the formula $a \vee (b \wedge \neg c)$, we get the following binary trees.



■

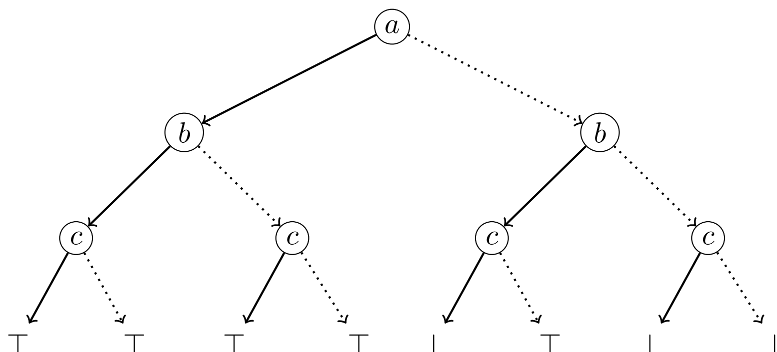
Our second observation is that this tree can often be represented far more compactly as a directed acyclic graph, in which two isomorphic sub-graphs are identified. First, there will be only two leaf nodes $\top$ and $\bot$ (in the special cases for valid and unsatisfiable formulas there is only one leaf node.) After this, higher up in the graph, if two nodes have the same child nodes, the nodes can be merged to one.

A third observation is that if both outgoing arcs of a node point to the same subgraph, then this node can be eliminated by directing the arc from its parents to the node's child node.

After these operations, we have a *reduced ordered binary decision diagram* (ROBDD, OBDD). The *reduced* refers to the merging of nodes that are logically equivalent. The *ordered* refers to the diagram being constructed with a fixed total ordering of the atomic propositions. Sometimes more relaxed binary decision diagram are used, especially without the ordering condition.

Observe that for a given variable ordering, the OBDD is unique up to an isomorphism, that is, all OBDDs generated for a given formula and a given variable ordering are isomorphic to each other, represented by essentially the same graph.

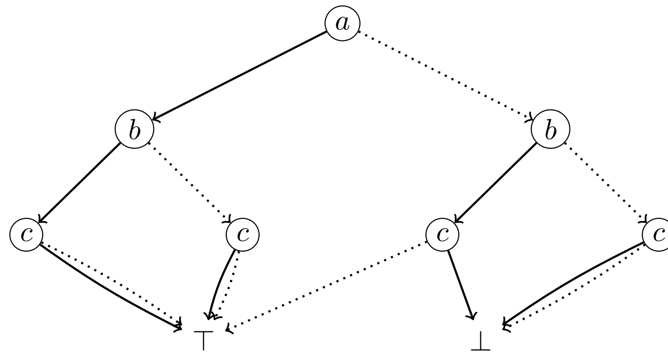
Example 4.14 Consider the binary tree we constructed for the formula $a \vee (b \wedge \neg c)$.



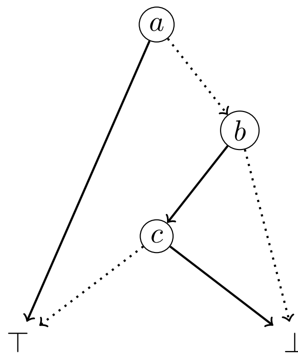
The correspondence with the truth table is exact. (Imagine rotating the truth table 90 degrees clockwise.)

a	b	c	$a \vee (b \wedge \neg c)$
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

Example 4.15 Merge all $\top$ nodes and all $\bot$ nodes.

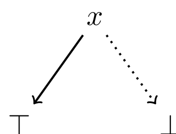


Then eliminate all nodes for which both arcs point to the same node. Above this means first eliminating three of the c nodes, after which the leftmost b node also needs to be eliminated. The resulting diagram is an OBDD.



The above construction progressed top-down starting from an arbitrary formula, generating an exponential size tree as an intermediate stage, before arriving at an OBDD. This OBDD could be constructed bottom up from the original formula, by first generating the OBDD for atomic propositions, and then constructing the OBDDs for $\phi \wedge \phi'$ and $\phi \vee \phi'$ directly from the OBDDs for ϕ and ϕ'.

The OBDD for an atomic proposition x is the following.

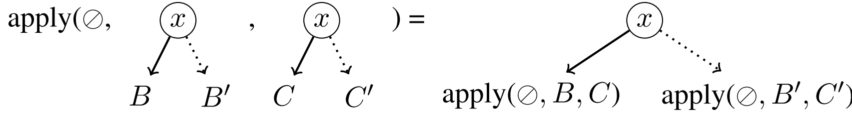


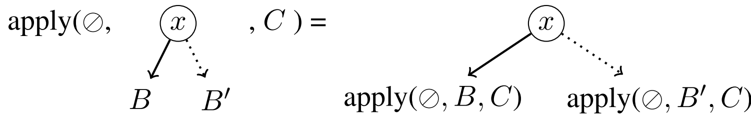
An OBDD can be negated by swapping the leaf nodes $\top$ and $\bot$. However, the typical setting in which an OBDD is negated is inside an OBDD package which maintains a high number of different Boolean functions in a large

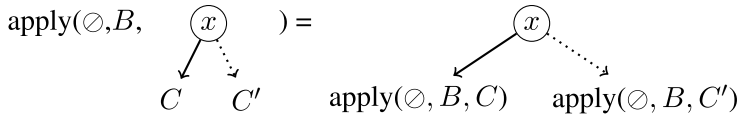
directed graph shared by all these Boolean functions. Swapping the two leaf nodes would negate all of the Boolean functions currently of interest, which is of course not wanted. For this reason, constructing the negation $\neg\phi$ of some BDD for ϕ is implemented indirectly, for example by constructing the BDD for $\phi \leftrightarrow \bot$ instead. This can be done similarly to the other binary connectives, which are described next.

By Boole's expansion theorem, $\phi_1 \oslash \phi_2$ with any connective $\oslash$ is logically equivalent to $(x \wedge \phi_1[\top/x] \oslash \phi_2[\top/x]) \vee (\neg x \wedge \phi_1[\bot/x] \oslash \phi_2[\bot/x])$. This yields as a way of constructing an OBDD for any formula $\phi_1 \oslash \phi_2$, given the OBDDs for ϕ_1 and ϕ_2.

This is what is known as the *apply* procedure.

1. If the root nodes of both constituent OBDDs have the same atomic propositions, then we do as follows.
   

2. If the atomic proposition in the root node of the OBDD C comes later in the variable ordering than x, or if C is one of the terminal nodes $\top$ or $\bot$, then we do the following.
   

3. Analogously to the previous case, if the atomic proposition in the root node of the OBDD B comes later in the variable ordering than x, or if B is $\top$ or $\bot$, then we do the following.
   

4. The remaining case is when both OBDDs are leaf nodes $\top$ or $\bot$. Then we apply the truth table for the connective $\oslash$ to the truth values.
   apply($\oslash$,b,b') = $b \oslash b'$

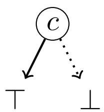
By using *apply*, an OBDD can be constructed from any propositional formula with the unary connective $\neg$ and binary Boolean connectives $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$, and others.

The above description of *apply* does not include the reduction part of OBDD construction. Including this is critical for efficiency. On the first three cases, the two recursive calls are first made, and if the result from both calls is the same OBDD, the node for x is not constructed, and the OBDD from the two recursive calls is returned instead. Further, memoization is used: before constructing an OBDD node for x and two constituent OBDDS B_1 and B_2, it is checked whether a node (x, B_1, B_2) has already been constructed. If so, that already-existing OBDD node is returned instead.

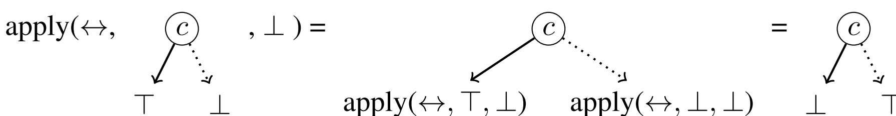
Although the runtime of one call to *apply* is polynomial in the size of the two OBDDs, the construction of an OBDD can be exponential time in the size of the original formula, as we call *apply* repeatedly, and OBDDs produced by *apply* can be bigger than the inputs to *apply*. This is as it should be, since constructing an OBDD solves the NP-complete satisfiability problem: a formula ϕ is satisfiable if and only if the its OBDD is not $\bot$.

Example 4.16 We illustrate *apply* with the OBDD for $a \vee (b \wedge \neg c)$ and the variable ordering a, b, c that is constructed by apply($\vee$,A,apply($\wedge$,B,apply($\leftrightarrow$,C,$\bot$))), where A, B and C are the OBDDs for the atomic propositions a, b and c. We show the first steps of this computation next.

The OBDD for the atomic proposition c is as follows.

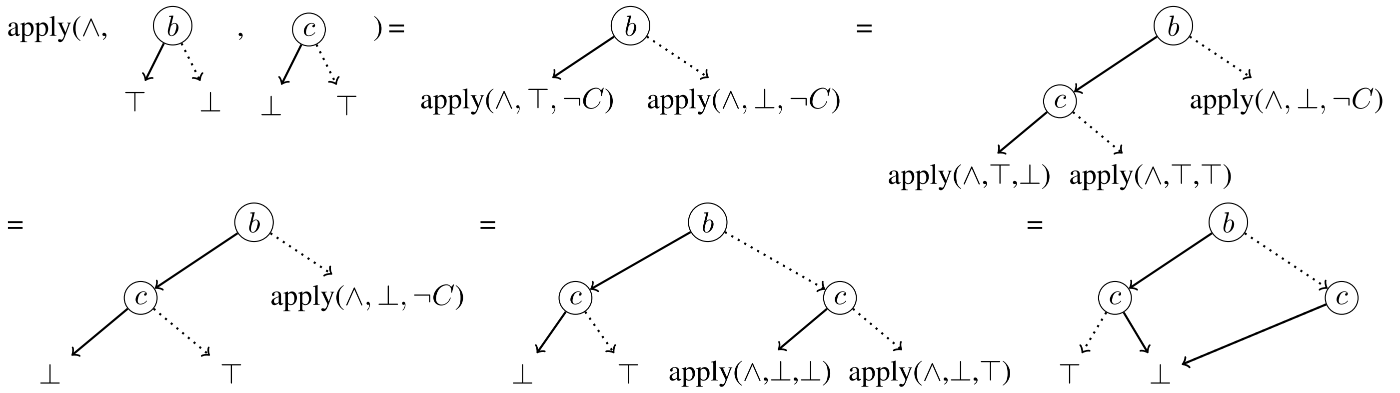


For apply($\leftrightarrow$,C,$\bot$) we use case 2 from above.

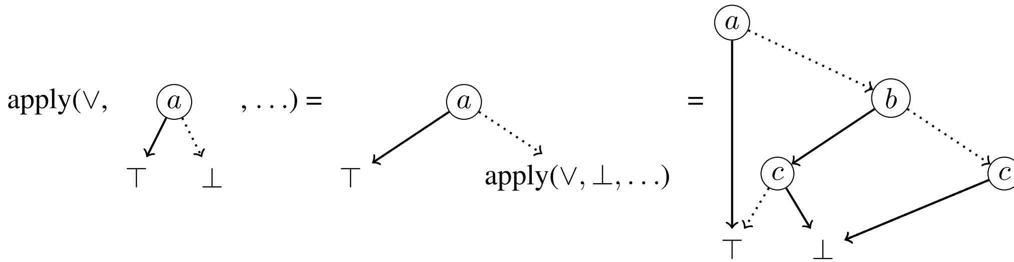


Next we do apply($\wedge$,B,...) with the above OBDD as the second conjunct. Since b precedes c in the ordering, we

again use case 2.



The final step is apply(∨,A,...) with the above BDD as the second argument.



OBDD for ϕ trivially answers the validity and satisfiability questions by equality with $\top$ and inequality with $\bot$, respectively. Logical consequence of $\Phi \models \phi$ is by the validity of $\Phi \rightarrow \phi$ by satisfiability of $\Phi \wedge \neg\phi$. The OBDD for one of these two formulas (which are negations of each other) needs to be constructed. For logical consequence tests $\Phi \models c$ where c is a single literal or a disjunction of literals, simpler graph-theoretic tests, which do not require constructing a new OBDD, exist. See Section 4.3.3.

4.3.1 Properties of OBDDs

Lemma 4.17 *The only valid OBDD is the one consisting of the single node* $\top$.

The only unsatisfiable OBDD is the one consisting of the single node $\bot$.

An OBDD can be exponentially larger than an equivalent propositional formula.

Example 4.18 Consider the formula $(x_1 \leftrightarrow x_1') \wedge \cdots \wedge (x_n \leftrightarrow x_n')$. The OBDD with the variable ordering $x_1, \ldots, x_n, x_1', \ldots, x_n$ has an exponential size. This OBDD is shown in Figure 4.2a. The number of nodes in the middle is 2^n. Hence the OBDD size is exponential in n and the size of the formula we started is linear in n. ■

An OBDD can be exponentially smaller than an equivalent propositional formula.

Example 4.19 The OBDD in Figure 4.1 is true if and only if the number of true atomic propositions is even.

This OBDD is true iff the number of true atomic propositions is even. An equivalent propositional formula *over the atomic propositions* $x_0, \ldots, x_n$ has an exponential size. Representing the OBDD compactly is possible, but requires auxiliary variables as in the Tseitin transformation.

Any path after an even number of true atomic propositions ends up on the right, and after an odd number it ends up on the left.

An equivalent propositional formula *over the atomic propositions* $x_0, \ldots, x_n$ has an exponential size. The OBDD can, of course, be represented compactly as a propositional formula by using auxiliary variables as in the Tseitin transformation [Tse68]. ■

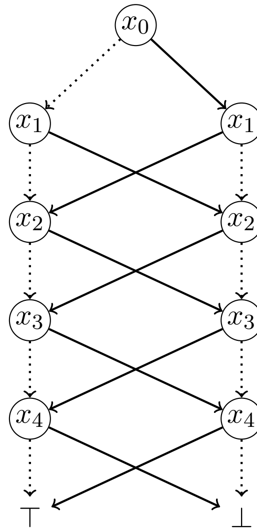


Figure 4.1: OBDD for determining the parity of an assignment

4.3.2 Model Counting

For any node in the OBDD (representing some formula Φ), the two subtrees rooted at that node represent formulas that contradict each other: $(x \wedge \Phi[\top/x])$ and $(\neg x \wedge \Phi[\bot/x])$, because of x and $\neg x$. Hence the number of models of Φ is the sum of the numbers of models of the formulas for the child nodes.

This property makes it possible to count the number of satisfying assignments of an OBDD in polynomial time. This is by traversing the OBDD once. Since the number of paths to a given node can be exponential (see Example 4.19), it is critical that the traversal counts the number of partial assignments for an internal node only once. For this it is enough that the model count for a node is recorded the first time it is needed, and subsequent uses of the count use the recorded number rather than doing the same computation multiple times.

In this algorithm, we assume that the variables in the BDD have integer index $0, \ldots,$ MAX. For OBDDs B, by index(B) we denote the index of the variable in the root node of an OBDD. If B is a terminal node, then the index is MAX $+ 1$.

To avoid traversing a possibly exponential number of paths in an OBDD, we use the array *memo* to record already computed model-counts for subgraphs. All elements of this array are initialized to 0 before calling *MC*.

1: **procedure** MC(B : OBDD)
2: **begin**
3: **if** $B = \top$ **then return** 1;
4: **if** $B = \bot$ **then return** 0;
5: **if** memo$[B] \geq 0$ **then return** memo$[B]$;
6: i := index(B.nodeVar);
7: i_1 := index(B.posChild);
8: i_0 := index(B.negChild);
9: memo$[B]$:= $2^{i_1-i-1}\times$ MC(B.posChild) + $2^{i_0-i-1}\times$ MC(B.negChild);
10: **return** memo$[B]$;
11: **end**

The correction terms 2^{i_1-i-1} and 2^{i_0-i-1} on line 4.3.2 are needed because they may be gaps in the variable ordering on some of the paths: if in a given path the variable x_i is not followed by x_{i+1} but some later variable x_k with $k > i+1$, then the variables $x_{i+1}, x_{i+2}, \ldots, x_{k-1}$ are "don't cares", and all valuations of these variables are possible on this path. Since there are $k-i-1$ of such "don't care" variables, we get the model count for $x_{i+1}, \ldots, x_n$ by multiplying the number of assignments for $x_k, \ldots, x_n$ by 2^{k-i-1}.

When determining the model-count for an OBDD B, we have to multiply the result of MC(B) by $2^{\mathrm{index}(B)}$, because the variable in the root node of B does not in general have the index 0.

4.3.3 Clausal consequences

Testing whether a clause $l_1 \vee \cdots \vee l_n$ is a logical consequence of an OBDD can be implemented as a graph-theoretic test: Do all paths from the root node of the OBDD to the leaf node $\top$ satisfy at least one of the literals in the clause, that is, for every path, is there at least literal l_i such that the atomic proposition x_i in l_i does not appear on the path, or the arc on the path from x_i matches the polarity of x in l_i (the outgoing arc from the node is dashed if $l_i = x_i$ and solid if $l_i = x_i$.)

The test is simplest done as follows: remove all such outgoing arcs for nodes with $x_1, \ldots, x_n$, as well as all arcs between nodes y and y' such that one of the x_i is between y and y' in the variable ordering. Then the clause is logical consequence of the OBDD if there is still at least one path from the root node to the leaf node $\top$: this means that the OBDD is true in at least one valuation in which all of $l_1, \ldots, l_n$ are false.

4.3.4 Restriction

Given an OBDD representing a propositional formula ϕ, the function *restrict* returns an OBDD that represents ϕ with every occurrences of a given atomic proposition replaced by the constant *true* or *false*. This operation is useful in implementing the *existential* and *universal abstraction* operations in Section 4.3.5.

```
1: procedure restrict(B : OBDD, x : variable, b : bool)
2: begin
3:    if index(B)> index(x) then return B;
4:    if index(B)< index(x) then return mkNode(index(B),restrict(B.posChild,x,b),restrict(B.negChild,x,b));
5:    if b = 0 then return B.negChild;
6:    return B.posChild;
7: end
```

Here *mkNode* creates a new OBDD node, or if a node with the same children exists already, returns that one, or, if both recursive calls to *restrict* return the same node, directly returns that node without creating any new nodes.

4.3.5 Abstraction

A key operation, needed in relational operations such as the join, is existential abstraction. This is also the operation that seems to be the reason why OBDDs are in many applications preferable to unrestricted propositional formulas: repeated existential abstraction unavoidably explodes the size of propositional formulas, whereas the size of OBDDs can remain in reasonable bounds.

For unrestricted propositional formulas there are no very effective methods for simplifying $\phi[\top/x] \vee \phi[\bot/x]$ as resulting from abstraction $\exists x.\phi$. Other operations, such as conjunction, renaming, and emptiness/satisfiability testing, would often not be an obstacle for using unrestricted propositional formulas in the applications where OBDDs have been very successful. NP-completeness of SAT is often mentioned as a reason why OBDDs are preferable, but the numbers of variables in many applications are relatively low (hundreds or at most thousands), and modern SAT solvers perform these tests very efficiently for smallish formulas.

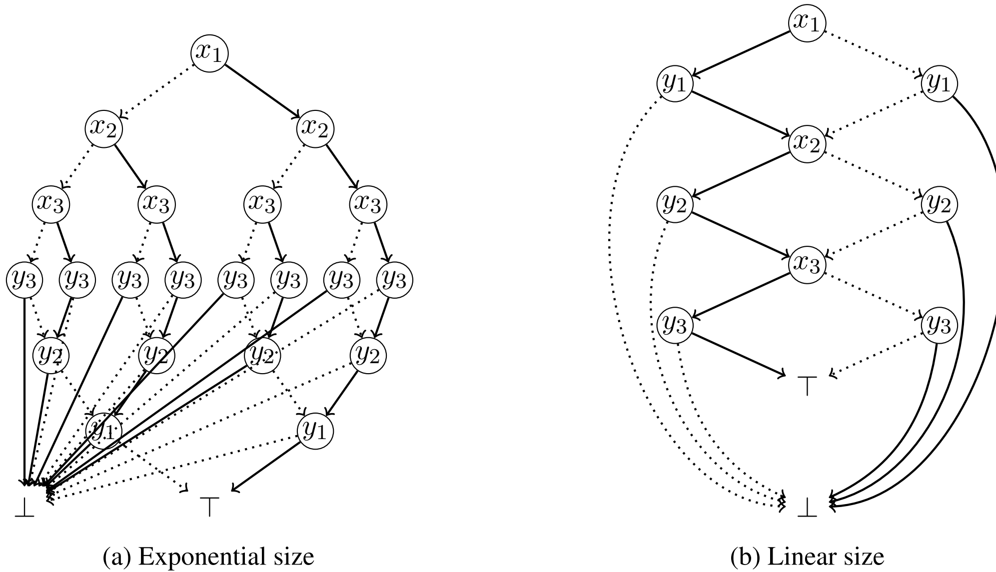
Some properties of the abstraction operations:

$$\exists x.(\phi_1 \vee \phi_2) \equiv (\exists x.\phi_1) \vee (\exists x.\phi_2) \tag{4.8}$$

$$\forall x.(\phi_1 \wedge \phi_2) \equiv (\forall x.\phi_1) \wedge (\forall x.\phi_2) \tag{4.9}$$

$$\exists x.\neg\phi \equiv \neg\forall x.\phi \tag{4.10}$$

$$\forall x.\neg\phi \equiv \neg\exists x.\phi \tag{4.11}$$



(a) Exponential size (b) Linear size

Figure 4.2: OBDDs for $(x_1 \leftrightarrow y_1) \wedge (x_2 \leftrightarrow y_2) \wedge (x_3 \leftrightarrow y_3)$

The abstraction operations can be implemented with the *restriction* operation of Section 4.3.4. Existential abstraction $\exists x.B$ simply as apply($\vee$,restrict(B,x,1),restrict(B,x,0)), and universal abstraction $\exists x.B$ as apply($\wedge$,restrict(B,x,1),restrict

4.3.6 Variable Ordering

For a given Boolean function, OBDDs with different variable ordering may look completely different, and the differences in their sizes may be dramatic. Finding a good variable ordering is one of the critical issues in the efficient use of OBDDs. A bad variable ordering may lead to an astronomically large OBDD, and a good variable ordering may lead to a small one.

Example 4.20 Figures 4.2a and 4.2b contain two OBDDs for the formula $(x_1 \leftrightarrow y_1) \wedge (x_2 \leftrightarrow y_2) \wedge (x_3 \leftrightarrow y_3)$.

The first one has an exponential size, and it has the variable ordering $x_1, \ldots, x_n, y_n, \ldots, y_1$. Intuitively, each of the y_3 nodes in the middle of the OBDD encodes one of the valuations of $x_1, \ldots, x_n$, corresponding to the path from the root node. Since there are 2^n valuations, there are 2^n nodes for y_n. The lower part of the OBDD reads the valuation of y_1, y_2, y_3, and if it does not match the valuation of x_1, x_2, x_3, an arc to the $\perp$-node is followed.

The second OBDD for the same formula has only $3n$ non-leaf nodes, and hence it is linear in n. The OBDD reads the x_i variables one by one, and immediately makes a comparison to the corresponding y_i variable. Hence the OBDD does not need to "remember" previous values, only the last x_i value.

Clearly the ordering $x_1, y_1, x_2, y_2, \ldots, x_n, y_n$ (or any ordering in which the x_i and y_i for the same i are next to each other) is better than $x_1, \ldots, x_n, y_n, \ldots, y_1$ (and other similar ones.) ■

Variable ordering can be designed manually, or they can be found by the automated variable ordering methods provided by OBDD packages. The automated methods perform a form of local search, trying out small modifications in the ordering, and keeping a change if the size of the OBDD got smaller. These methods have not guarantee of finding a good ordering even if an obvious one existed.

4.4 Normal Forms DNNF and d-DNNF

Darwiche et al. [Dar01, DM02] have carried out a thorough study on features of propositional formulas, to identify normal forms of practical importance.

The first new normal form identified by Darwiche was Decomposable NNF [Dar01]. This is NNF with the additional property of *decomposability*.

Definition 4.21 (Decomposability) *A formula ϕ in NNF is* decomposable *if for every subformula $\psi_1 \wedge \psi_2$ of ϕ, the subformulas ψ_1 and ψ_2 don't share atomic propositions.*

The normal form DNNF (Decomposable Negation Normal Form) is defined as formulas in NNF that additionally satisfy the decomposability condition.

Notice that OBDDs are subclass of DNNF: In OBDDs conjunctions are of the form $x \wedge \phi[\top/x]$ and $\neg x \wedge \phi[\bot/x]$, where x is an atomic proposition and $\phi[\top/x]$ and $\phi[\bot/x]$ trivially have no occurrences of x.

Unlike for general NNF formulas, formulas in DNNF have the interesting property that satisfiability testing can be done in polynomial time. The algorithm for doing this is as follows.

$$
\begin{aligned}
dcSAT(x) &= \text{true} \\
dcSAT(\neg x) &= \text{true} \\
dcSAT(\top) &= \text{true} \\
dcSAT(\bot) &= \text{false} \\
dcSAT(\phi_1 \wedge \phi_2) &= dcSAT(\phi_1) \text{ and } dcSAT(\phi_2) \\
dcSAT(\phi_1 \vee \phi_2) &= dcSAT(\phi_1) \text{ or } dcSAT(\phi_2)
\end{aligned}
$$

This algorithm does not work correctly for general NNF. For example, dcSAT($x \wedge \neg x$) will incorrectly return *true*. But the decomposability property guarantees that the satisfiability of ϕ_1 and the satisfiability of ϕ_2 entail the satisfiability of $\phi_1 \wedge \phi_2$. Since the two formulas share no atomic propositions, if they are both satisfiable, we can always easily construct a satisfying valuation from the satisfying valuations of the two formulas.

Translation from general propositional formulas into NNF is polynomial time, but translation from NNF into DNNF takes exponential time in the worst case. This is to be expected, as SAT is NP-hard but for DNNF satisfiability testing is polynomial time. The most effective known methods for translating formulas into DNNF always translate the formulas into the normal form d-DNNF too [Dar02, HD05]. We define d-DNNF next.

A further condition on NNF is *determinism*.

Definition 4.22 (Determinism) *A formula ϕ in NNF is* deterministic *if for every subformula $\psi_1 \vee \psi_2$ of ϕ, the subformulas ψ_1 and ψ_2 are mutually contradictory.*

The normal form d-DNNF (Deterministic DNNF) is defined as formulas in DNNF that additionally satisfy the determinism condition.

Notice that OBDDs are subclass of d-DNNF: In OBDDs disjunctions are of the form $(x \wedge \phi[\top/x]) \vee (\neg x \wedge [\bot/x])$, and the two disjuncts contradict because of x and $\neg x$.

In addition to polynomial-time satisfiability testing, d-DNNF additionally have polynomial-time model-counting. This is because the model counts, that is the number of satisfying assignments, for a subformula $\psi_1 \vee \psi_2$ can be obtained as the sum of the model-counts of ψ_1 and ψ_2.

There are examples of Boolean functions that can be expressed more compactly in DNNF than in d-DNNF, and more compactly in d-DNNF than in OBDD. A property separating OBDD from d-DNNF and DNNF is *canonicity*: the DNNF and d-DNNF for a given Boolean function are not unique. This seems to be the main reason why OBDDs, and not some other normal form, has been extensively used in practice. Canonicity of OBDDs relies on the ordering condition, whereas for DNNF and d-DNNF no obvious ordering condition exists. A different type of normal form, called Sentential Decision Diagram (SDD) [Dar11], located between OBDD and DNNF, has an ordering condition and canonicity.

For n atomic propositions, the truth-table has 2^n rows, and the last row (indicating the value of a Boolean function for one valuation of the atomic propositions) can be assigned a value in 2^{2^n} different ways. Hence there are 2^{2^n} different Boolean functions of n variables (atomic propositions). One can show by a simple cardinality argument that "most" Boolean functions do not have "compact" representation (e.g. size that is polynomial in the number of propositional variables.)

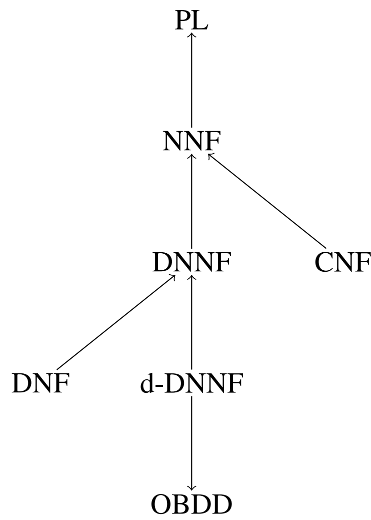


Figure 4.3: Inclusion Relations of Normal Forms

operation	PL	NNF	CNF	DNF	DNNF	d-DNNF	OBDD
ϕ ∈SAT	NP	NP	NP	P	P	P	P
ϕ ∈TAUT	co-NP	co-NP	P	co-NP	co-NP	P	P
$\phi \models c$	co-NP	co-NP	co-NP	P	P	P	P
model counting	NP	NP	NP	NP	NP	P	P
$n \times \wedge$	P	P	P	exp	exp	exp	exp
$n \times \vee$	P	P	exp	P	P	exp	exp
$\neg$	P	P	exp	exp	exp	?	P

Table 4.2: Complexity of Logical Operations for Different Normal Forms

4.5 Normal Forms Summary

Figure 4.3 depicts the inclusion relations between some of the best known normal forms, and Table 4.2 summarizes the complexities of some important operations on logical formulas for these normal forms. In the table, NP denotes NP-hard, co-NP denotes co-NP-hard, and P denotes "polynomial time". The complexity of negating a d-DNNF formula is not known.

For the construction operations $\vee$, $\wedge$ and $\neg$ the result has to be in the same normal form. For one $\wedge$ or $\vee$, construction is polynomial size and time (at most $n_1 \times n_2$ when the constituent formulas have sizes n_1 and n_2, respectively), but repeating the operation n times is exponential in n, as the size can increase exponentially $(((n_1 \times n_2) \times n_3) \times n_4) \times \cdots \times n_k$, as the product $n_1 n_2 \cdots n_k$ is exponential in k.

The *clausal logical consequence* $\phi \models c$ tests whether a clause c is a logical consequence of the formula ϕ. Only the formula ϕ here is expected to be in the normal form in question. (A clause, a disjunction of literals, is not a DNNF, d-DNNF, or OBDD.)

Chapter 5

Sets and Relations in the Propositional Logic

Formulas can be viewed as a data structure for representing sets and relations. The advantage of logic in this application is *succinctness*: a formula can represent a set that has a size that is exponential in the size of the formula. Logic representation is therefore succinct. This is in strong contrast to conventional data structures which would explicitly enumerate all elements of a set, and therefore have a size that is linear in the size of the set.

5.1 Representation of Sets

Formulas can be considered as a representation of those sets of valuations that make the formula true. We can identify a valuation $v : X \to \{0, 1\}$ with a vector of length $|X|$, with each element of the vector corresponding to one of the atomic propositions in X.

Example 5.1 Let $X = \{A, B, C, D\}$. Now a valuation that assigns 1 to A and C and 0 to B and D corresponds to the bit-vector $\overset{ABCD}{1\,0\,1\,0}$, and the valuation assigning 1 only to B corresponds to the bit-vector $\overset{ABCD}{0\,1\,0\,0}$. ■

Any propositional formula ϕ can be understood as a representation of those valuations v such that $v(\phi) = 1$. Since we have identified valuations and bit-vectors, a formula naturally represents a set of bit-vectors.

Example 5.2 Formula B represents all bit-vectors of the form ?1??, and the formula A represents all bit-vectors 1???. Formula B therefore represents the *set*

$$\{0\mathbf{1}00, 0\mathbf{1}01, 0\mathbf{1}10, 0\mathbf{1}11, 1\mathbf{1}00, 1\mathbf{1}01, 1\mathbf{1}10, 1\mathbf{1}11\}$$

and formula A represents the set

$$\{\mathbf{1}000, \mathbf{1}001, \mathbf{1}010, \mathbf{1}011, \mathbf{1}100, \mathbf{1}101, \mathbf{1}110, \mathbf{1}111\}.$$

■

Similarly, $\neg B$ represents all bit-vectors of the form ?0??, which is the set

$$\{0\mathbf{0}00, 0\mathbf{0}01, 0\mathbf{0}10, 0\mathbf{0}11, 1\mathbf{0}00, 1\mathbf{0}01, 1\mathbf{0}10, 1\mathbf{0}11\}.$$

This is the *complement* of the set represented by B.

5.1.1 Set Operations as Logical Operations

There is a close connection between the Boolean connectives $\vee$, $\wedge$, $\neg$ and the set-theoretical operations of *union*, *intersection* and *complementation*, which is also historically the origin of Boole's work on Boolean functions.

If ϕ_1 and ϕ_2 represent bit-vector sets S_1 and S_2, then

sets	**formulas over X**
those $\frac{2^{\lvert X\rvert}}{2}$ bit-vectors where x is true	$x \in X$
$\overline{E}$ (complement)	$\neg E$
$E \cup F$	$E \vee F$
$E \cap F$	$E \wedge F$
$E \backslash F$ (set difference)	$E \wedge \neg F$
the empty set $\emptyset$	$\perp$ (constant *false*)
the universal set	$\top$ (constant *true*)
question about sets	**question about formulas**
$E \subseteq F$?	$E \models F$?
$E \subset F$?	$E \models F$ and $F \not\models E$?
$E = F$?	$E \models F$ and $F \models E$?

Table 5.1: Connections between Set-Theory and Propositional Logic

1. $\phi_1 \wedge \phi_2$ represents set $S_1 \cap S_2$,
2. $\phi_1 \vee \phi_2$ represents set $S_1 \cup S_2$, and
3. $\neg\phi_1$ represents set $\overline{S_1}$.

Example 5.3 $A \wedge B$ represents the set $\{1100, 1101, 1110, 1111\}$ and $A \vee B$ represents the set $\{0100, 0101, 0110, 0111, 1000, 1001, 1010, 1011, 1100, 1101, 1110, 1111\}$. ■

Questions about the relations between sets represented as formulas can be reduced to the basic logical concepts we already know, namely logical consequence, satisfiability, and validity.

1. "Is ϕ satisfiable?" corresponds to "Is the set represented by ϕ *non-empty*?"
2. $\phi \models \alpha$ corresponds to "Is the set represented by ϕ *a subset* of the set represented by α?".
3. "Is ϕ valid?" corresponds to "Is the set represented by ϕ *the universal set*?"

These connections allow using propositional formulas as a *data structure* in some applications in which conventional enumerative data structures for sets are not suitable because of the astronomic number of elements in the sets. For example, if there are 100 atomic propositions, then any formula consisting of just one atomic proposition represents a set of $2^{99} = 633825300114114700748351602688$ bit-vectors, which would require 7493989779944505344 TB in an explicit enumerative representation if each of the 100-bit vectors was represented with 13 bytes (wasting only 4 bits in the 13th byte.)

5.2 Relations as Formulas

Similarly to finite sets of atomic objects, *relations* on any finite set of objects can be represented as propositional formulas. One of the main applications of this is the representation of *transition relations* as formulas in order to solve state-space search problems by automated reasoning methods.

A binary relation $R \subseteq X \times X$ is a set of pairs $(a, b) \in X \times X$, and the representation of this relation is similar to representing sets before, except that the elements are pairs.

As before, we assume that the atomic objects are bit-vectors. A pair of bit-vectors of lengths n and m can of course be represented as a bit-vector of length $n + m$, simply by attaching the two bit-vectors together.

Example 5.4 To represent the *pair* $(0001, 1100)$ of bit-vectors, both expressed as valuations of atomic propositions $X = \{A, B, C, D\}$, instead use the atomic propositions $X_{01} = \{A_0, B_0, C_0, D_0, A_1, B_1, C_1, D_1\}$ as the index variables.

The pair $(0001, 1100)$ is hence represented as 00011100, a valuation of X_{01}. ■

Pair $(\overset{A_0}{0}\overset{B_0}{0}\overset{C_0}{0}\overset{D_0}{1}, \overset{A_1}{1}\overset{B_1}{1}\overset{C_1}{0}\overset{D_1}{0})$ therefore corresponds to the valuation that assigns 1 to D_0, A_1 and B_1 and 0 to all other variables.

Example 5.5 $(A_0 \leftrightarrow A_1) \wedge (B_0 \leftrightarrow B_1) \wedge (C_0 \leftrightarrow C_1) \wedge (D_0 \leftrightarrow D_1)$ represents the identity relation of 4-bit bit-vectors. ■

Example 5.6 The formula

$$\begin{aligned} inc_{01} = & (\neg c_0 \wedge c_1 \wedge (b_0 \leftrightarrow b_1) \wedge (a_0 \leftrightarrow a_1)) \\ & \vee(\neg b_0 \wedge c_0 \wedge b_1 \wedge \neg c_1 \wedge (a_0 \leftrightarrow a_1)) \\ & \vee(\neg a_0 \wedge b_0 \wedge c_0 \wedge a_1 \wedge \neg b_1 \wedge \neg c_1) \\ & \vee(a_0 \wedge b_0 \wedge c_0 \wedge \neg a_1 \wedge \neg b_1 \wedge \neg c_1) \end{aligned}$$

represents the successor relation of 3-bit integers

$$\{(000, 001), (001, 010), (010, 011), (011, 100), (100, 101), (101, 110), (110, 111), (111, 000)\},$$

which can also be depicted in a tabular form as follows, in order to make the variables for the columns explicit.

$a_0b_0c_0$	$a_1b_1c_1$
000	001
001	010
010	011
011	100
100	101
101	110
110	111
111	000

■

Notice in this example that the tabular representation only has 8 rows, whereas the corresponding truth-table for the formula $inc_0 1$ has $2^6 = 64$ rows. The tabular representation of the relation only enumerates those rows for which the formula evaluates to *true*.

Example 5.7 Consider the relation $\{(001, 111), (010, 110), (011, 010), (111, 110)\}$. The relation can be represented as the following truth-table (listing only those of the $2^6 = 64$ lines that have 1 in the column for ϕ),

a_0	b_0	c_0	a_1	b_1	c_1	ϕ
⋮						
0	0	1	1	1	1	1
⋮						
0	1	0	1	1	0	1
⋮						
0	1	1	0	1	0	1
⋮						
1	1	1	1	1	0	1
⋮						

which is equivalent to the following formula.

$$\begin{aligned} & (\neg a_0 \wedge \neg b_0 \wedge c_0 \wedge a_1 \wedge b_1 \wedge c_1)\vee \\ & (\neg a_0 \wedge b_0 \wedge \neg c_0 \wedge a_1 \wedge b_1 \wedge \neg c_1)\vee \\ & (\neg a_0 \wedge b_0 \wedge c_0 \wedge \neg a_1 \wedge b_1 \wedge \neg c_1)\vee \\ & (a_0 \wedge b_0 \wedge c_0 \wedge a_1 \wedge b_1 \wedge \neg c_1) \end{aligned}$$

■

Any binary relation over a finite set can be represented as a propositional formula in this way. These formulas can be used as components of formulas that represent *paths* in the graphs corresponding to the binary relation.

5.2.1 Relational Operations in Logic

As relations as sets of pairs (or, for n-ary relations, sets of n-tuples), the set operations that we already defined ($\cup$, $\cap$, and so on) immediately apply to relations as well. Of interest in many of our applications are also well-known relational algebra operations such as join operations (natural join $\bowtie$), projections, and so on. In this section, we will show how the most important relational operations can be implemented by formula manipulation when the relations in question are represented as formulas. We focus on the core operations of natural join, selection, and projection, as most of the other operations, for example many types of join operations, can be reduced to these basic operations.

The join and projection operations are the basis operations when computing the successors of a set of states with respect to a binary transition relation, when both the set and the relation are represented as formulas.

We first consider the natural join operation. The natural join operation takes two relations, and constructs a new table with columns from the two constituent tables, and each row in the new table consists of rows from the constituent tables that match on the shared columns. There may be zero, one, or more shared columns.

Example 5.8 We form the natural join $R_1 \bowtie R_2$ of two relations R_1 and R_2.

0	1
01	10
10	01
11	11

$\bowtie$

1	2
00	01
01	10
10	11

$=$

0	1	2
01	10	11
10	01	10

The relation R_1 could be called *non-zero swap*, and R_2 could be called *increment*. Formulas to represent R_1 and R_2 are respectively

$$\phi_1 = (a_0 \vee b_0) \wedge (a_1 \leftrightarrow b_0) \wedge (b_1 \leftrightarrow a_0)$$

and

$$\phi_2 = (b_2 \leftrightarrow \neg a_1) \wedge (a_2 \leftrightarrow (a_1 \wedge \neg b_1)) \wedge \neg(a_1 \wedge b_1).$$

(The same Boolean functions can of course be represented by many other logically equivalent formulas.)

Although ϕ_1 only contains occurrences of a_0, b_0, a_1, b_1, one should thing about its truth-table that also includes columns for a_2 and b_2. Essentially, a_2 and b_2 for ϕ_1 are *don't cares*: they are not limited by ϕ_1, and they can obtain any truth-values.

Similarly, ϕ_2 does not explicitly refer to a_0 and b_0.

Now, the join $R_1 \bowtie R_2$ represents all those valuations of $a_0, b_0, a_1, b_1, a_2, b_2$ that satisfy both ϕ_1 and ϕ_2.

The process of matching column 1 values, that is central in computing the join of relations represented as tables, corresponds to the requirement that valuations that satisfy the formula for $R_1 \bowtie R_2$ have to satisfy both ϕ_1 and ϕ_2.

Hence the natural join operation simply corresponds to the logical conjunction $\wedge$, and $R_1 \bowtie R_2$ is represented by $\phi_1 \wedge \phi_2$, as can be easily verified. ■

The *selection* operation $\sigma_\beta(R)$ has a straightforward representation when the relation R is represented as a formula ϕ. This is again simply a conjunction $\phi \wedge \beta$, when β has been expressed in terms of propositional variables occurring in ϕ.

Finally, we consider the important *projection* operation $\pi_c(R)$. This operation select some subset c of the columns of a relation.

Example 5.9 Consider the relation R

0	1
00	00
01	00
10	11
11	11

represented by the formula $\phi_1 = (a_1 \leftrightarrow a_0) \wedge (b_1 \leftrightarrow a_0)$ (over variables a_0, b_0, a_1, b_1), which we would like to project to the column 1 by $\pi_1(R)$ to obtain

$$\pi_1\left(\begin{array}{c|c} 0 & 1 \\ \hline 00 & 00 \\ 01 & 00 \\ 10 & 11 \\ 11 & 11 \end{array}\right) = \begin{array}{c} 1 \\ \hline 00 \\ 11 \end{array}$$

The result of the projection is the atomic formula $a_1 \leftrightarrow b_1$.

Consider the truth-tables of the original relation and its projection to 1:

a_0	b_0	a_1	b_1	$(a_1 \leftrightarrow a_0) \wedge (b_1 \leftrightarrow a_0)$
0	0	**0**	**0**	**1**
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	**0**	**0**	**1**
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	**0**	**0**	**0**
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	**0**	**0**	**0**
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

a_1	b_1	$a_1 \leftrightarrow b_1$
0	0	1
0	1	0
1	0	0
1	1	1

In the second table exactly those rows (valuations of a_1, b_1) are mapped to 1, for which there is at least one row in the first table with matching values for a_1, b_1 and 1 in the last column. We have highlighted the rows in the first table that correspond to $a_1 = 0, b_1 = 0$. This valuation is mapped to 1 in the second table. Similarly, for valuation $a_1 = 0, b_1 = 1$ we get 0 in the second table, because the first table always maps this valuation to 0. ■

What the above example illustrates turns out to exactly match what Existential Abstraction does: eliminating variables a_0 and b_0 from the truth table for ϕ corresponds to existentially abstracting a_0 and b_0 in ϕ, that is, $\exists a_0 \exists b_0 . \phi$.

Definition 5.10 (Existential abstraction) *Existential abstraction of ϕ with respect to x is defined by*

$$\exists x.\phi = \phi[\top/x] \vee \phi[\bot/x].$$

This operation allows eliminating atomic proposition x from any formula. Make two copies of the formula, with x replaced by the constant true $\top$ in one copy and with constant false $\bot$ in the other, and form their disjunction.

Analogously we can define *universal abstraction*, with conjunction instead of disjunction.

Definition 5.11 (Universal abstraction) *Universal abstraction of ϕ with respect to x is defined by*

$$\forall x.\phi = \phi[\top/x] \wedge \phi[\bot/x].$$

Example 5.12

$$\begin{aligned}
&\exists B.((A \to B) \wedge (B \to C)) \\
&= ((A \to \top) \wedge (\top \to C)) \vee ((A \to \bot) \wedge (\bot \to C)) \\
&\equiv C \vee \neg A \\
&\equiv A \to C
\end{aligned}$$

$$\begin{aligned}
&\exists AB.(A \vee B) = \exists B.(\top \vee B) \vee (\bot \vee B) \\
&= ((\top \vee \top) \vee (\bot \vee \top)) \vee ((\top \vee \bot) \vee (\bot \vee \bot)) \\
&\equiv (\top \vee \top) \vee (\top \vee \bot) \equiv \top
\end{aligned}$$

■

Both universal $\forall c$ and existential $\exists c$ abstraction can be viewed as eliminating a column for c in a truth-table of a formula ϕ by combining lines with the same valuation for variables other than c. There are always two lines that agree on the valuation of variables other than c, one in which $c = 0$ and the other with $c = 1$.

For universal abstraction these lines are combined by the Boolean function *and*, and for existential abstraction these lines are combined by the Boolean function *or*. We will illustrate this in the next example.

Example 5.13 We will abstract $a \vee (b \wedge c)$ both existentially and universally, obtaining $\exists c.(a \vee (b \wedge c)) \equiv a \vee b$ and $\forall c.(a \vee (b \wedge c)) \equiv a$. The corresponding truth tables for the original formula and the results of the two abstractions are as follows.

$a\ b\ c$	$a \vee (b \wedge c)$
0 0 0	0
0 0 1	0
0 1 0	0
0 1 1	1
1 0 0	1
1 0 1	1
1 1 0	1
1 1 1	1

$a\ b$	$\exists c.(a \vee (b \wedge c))$
0 0	0
0 1	1
1 0	1
1 1	1

$a\ b$	$\forall c.(a \vee (b \wedge c))$
0 0	0
0 1	0
1 0	1
1 1	1

Here we have colored the pairs of rows in the first truth-table which will be merged to the corresponding rows in the truth-tables for the abstracted formulas. The difference between the abstractions is that the existential abstraction yields 0 if and only if both of the two rows in the original table are 0, and for universal abstraction we get 0 if and only if at least one of the rows is 0. ■

Example

From $\neg A_0 \wedge \neg A_1 \wedge ((\neg B_0 \wedge \neg C_0 \wedge B_1 \wedge C_1) \vee (B_0 \wedge \neg C_0 \wedge \neg B_1 \wedge C_1)$ produce $(\neg A_1 \wedge B_1 \wedge C_1) \vee (\neg A_1 \wedge \neg B_1 \wedge C_1)$.

$\Phi = \neg A_0 \wedge \neg A_1 \wedge ((\neg B_0 \wedge \neg C_0 \wedge B_1 \wedge C_1) \vee (B_0 \wedge \neg C_0 \wedge \neg B_1 \wedge C_1)$

$$\begin{aligned}
&\exists A_0 B_0 C_0.\Phi \\
&= \exists B_0 C_0.(\Phi[0/A_0] \vee \Phi[1/A_0]) \\
&= \exists B_0 C_0.(\neg A_1 \wedge ((\neg B_0 \wedge \neg C_0 \wedge B_1 \wedge C_1) \vee (B_0 \wedge \neg C_0 \wedge \neg B_1 \wedge C_1)) \\
&= \exists C_0.((\neg A_1 \wedge (\neg C_0 \wedge B_1 \wedge C_1)) \vee (\neg A_1 \wedge ((\neg C_0 \wedge \neg B_1 \wedge C_1)))) \\
&= (\neg A_1 \wedge B_1 \wedge C_1) \vee (\neg A_1 \wedge \neg B_1 \wedge C_1)
\end{aligned}$$

Chapter 6

Transition Systems

6.1 Basic Models

Definition 6.1 (Transition System) *A* transition system $\langle S, E, R \rangle$ *consists of*

- *a set S of* states*,*
- *a set E of* events*, and*
- *an assignment $R : E \to 2^{S \times S}$ of* transition relations $R(e) \subseteq S \times S$ *to all events $e \in E$.*

Transition systems model the dynamics of system: the state of the system is changed by events. Zero or more events may be possible in a state, as indicated by the transition relation associated with the event: if s is related by $R(e)$ to some state, then the event e is possible in s. If s is related to s' by $R(e)$ then it is possible that the event e changes the state from s to s'. The transition system does not determine which events take place: if there are more than one event possible in a state, there may be one or more possible next states for a state.

Definition 6.2 (Execution) *In a transition system $\langle S, E, R \rangle$, an* execution $s_0, e_0, s_1, e_1, s_2, e_2, \ldots, e_{n-1}, s_n$ *with* $\{s_0, \ldots, s_n\} \subseteq S$ *and* $e_0, \ldots, e_{n-1} \subseteq E$ *is possible if* $(s_i, s_{i+1} \in R(e_i)$ *for all i such that* $0 \leq i < n$.

A sequence s consisting of a single state $s \in S$ is a trivial execution.

Definition 6.3 (Transition System with State Variables) *A* transition system with state variables $\langle S, E, R, X, v \rangle$ *consists of a transition system $\langle S, E, R \rangle$ and*

- *a set X of* state variables*,*
- *a valuation $v : S \times X \to \{0,1\}$ of state variables in states.*

For convenience, we will write $v_s(x)$ for $v(s, x)$.

In a transition system with state variables we can talk about the truth of a propositional formula in a state of the transition system. It is trivial to adapt Definition 2.3 to transition systems with state variables to define $v_s(\phi)$ for any state $s \in S$ and any propositional formula ϕ over X.

Definition 6.4 (Succinct Transition System) *A* succinct transition system variables $\langle X, E, R, I \rangle$ *consists of*

- *a set X of* state variables*,*
- *a set E of* events
- *a succinctly represented transition relation $R(e) \in \mathcal{L}(X \cup X')$ for every $e \in E$, and*
- *a succinctly represented set of initial states $I \in \mathcal{L}(X)$.*

If the formula I is the constant $\top$, then any valuation of X is a possible initial state of the transition system. Typically I expresses at least dependencies between state variables, if not specifying a unique initial state (by a conjunction of literals for every $x \in X$.)

The relations are represented as formulas $R(e)$ with occurrences of atomic propositions for values of state variables $X = \{a, b, c, \ldots\}$ and values of state variables in *the next state* $X' = \{a', b', c', \ldots\}$.

Representation of transition relations in Section 5.2.

6.2 Procedural Representations

While we have defined succinct transition systems in terms of transitions relations represented as logical formulas, this is not how large transition systems are modelled in practise.

Practical modelling languages represent states are assignments of values to state variables, exactly as with the logic representation, but transitions are typically defined *procedurally*, in terms of *changes* they induce on the state variables. This, in turn, is similar to programming languages: changes are assignments of values to state variables, unconditionally or conditionally on some facts holding.

In this section we describe how such procedural representations can be mapped to the logic representation. The logic representation is important for many of the state space search methods that are scalable to very large state spaces. If it was not for the logic based representations, we could have defined succinct transition systems (Definition 6.4) in terms of functions $f : S \rightarrow 2^S$, where S is the set of all valuations of the state variables, that map individual states to sets of states (zero or one states for a deterministic transitions, and 2 or more states for nondeterministic transitions.) Clearly, both types of definitions are equivalent, because a function $f : S \rightarrow 2^S$ can be understood as a many-to-many binary relation on S, exactly as we have defined transitions in Definition 6.4.

The plan next is to give a simplest possible procedural transition representation and show how it is mapped to the propositional logic (Section 6.2.1), and then give a more expressive definition that covers a complex modelling language (Section 6.2.2)

6.2.1 Deterministic Unconditional Transitions

We first consider a transition representation in which a transition is characterized by a pair (p, e) where p is an arbitrary propositional formula (the precondition), and e is a set of assignments x := 0 or x := 1. That is, every change to the state variables is fixed and independent of the state the transition takes place. The set e of effects can be identified with literals x and $\neg x$, corresponding to assignments to 1 and 0.

This language cannot compactly represent many types of changes. For example, it cannot compactly represent a transition that complements the value of n state variables $x_1, \ldots, x_n$. One would need 2^n separate transitions for representing this type of complementation.

Nevertheless, many types of complex systems can still be represented, with conditional behavior split to multiple transitions. Incrementing an n-bit binary number represented by $x_{n-1}, \ldots, x_0$ can be done with n unconditional transitions.

Example 6.5 Incrementing a 5-bit binary number is achieved by the following transitions, exactly one of which can be applicable in any given state.

$$
\begin{array}{l}
(\neg x_0, \{x_0\}) \\
(\neg x_1 \wedge x_0, \{x_1, \neg x_0\}) \\
(\neg x_2 \wedge x_1 \wedge x_0, \{x_2, \neg x_1, \neg x_0\}) \\
(\neg x_3 \wedge x_2 \wedge x_1 \wedge x_0, \{x_3, \neg x_2, \neg x_1, \neg x_0\}) \\
(\neg x_4 \wedge x_3 \wedge x_2 \wedge x_1 \wedge x_0, \{x_4, \neg x_3, \neg x_2, \neg x_1, \neg x_0\})
\end{array}
$$

■

Definition 6.6 (Unconditional deterministic transition) *Given an unconditional transition* (p, e)*, its translation into the propositional logic is*

$$p \wedge \bigwedge \{x' | x \in e, x \in X\} \wedge \{\neg x' | \neg x \in e, x \in X\} \wedge \{x \leftrightarrow x' | x \in X, x \notin e, \neg x \notin e\}$$

In addition to the precondition, the formula essentially gives, for every state variable $x \in X$, an equivalence $x' \leftrightarrow \phi_x$, where $\phi_x = \top$ if $x \in p$, $\phi_x = \bot$ if $\neg x \in p$, and $\phi_x = x$ if $\{x, \neg x\} \cap p = \emptyset$. These equivalences describe how the new value of every state variable depends on the values of the old values of state variables. In the simple case covered here, the dependencies are simple, as either a state variable retains its old value, or becomes unconditional true or false. In the next section we cover far more complex dependencies.

Exercise 6.1 *Given a state in which a, b and c are all true, and the transition $(a \wedge b, \{\neg b, \neg c\})$,*

1. *express both the state and the transition relation as propositional formulas,*
2. *form their conjunction,*
3. *existentially abstract away all atomic propositions $x \in X$, leaving only atomic propositions $x \in X'$ left,*
4. *simplify the formula by eliminating all occurrences of $\top$ and $\bot$.*

6.2.2 General Definition

The simple procedural representation for unconditional transitions covered in the previous section can be restrictive, as the changes taking place in a state cannot be dependent on the state, which may require the use of a large number of unconditional transitions. For representing more complex systems it is customary to use a transition model with more complex features, which allow representing many problem far more compactly.

Definition 6.7 (Succinct transition procedure) *Given a set X of state variables, a transition is described by a pair (p, e) where p is a propositional formula over X, and e is an* effect. *The set of all possible effects over X is recursively defined by*

1. *ϵ (the NO-OP effect) is an effect,*
2. *x and $\neg x$ for any $x \in X$ are (atomic) effects,*
3. *eITE(ϕ, e_1, e_2) is an effect if ϕ is a formula over X and e_1 and e_2 are effects over X, and*
4. *$e_1; e_2$ is an effect if e_1 and e_2 are effects over X.*

The empty effect ϵ does not do anything. The conditional effect $eITE\phi e_1 e_2$ is the procedural IF-THEN-ELSE statement: if the condition ϕ holds, then e_1 is executed, and otherwise e_2 is executed. We could define IF-THEN *eIT*(ϕ, e) as a special case *eITE*(ϕ, e, ϵ). The sequential composition $e_1; e_2$ of two effects means that e_1 is executed first, and then e_2 after that.

Next we present a systematic derivation of a translation from our action definition to the propositional logic. We could give the definition directly (and you can skip to the table given on the next page), but we can alternatively use the weakest precondition predicate familiar from programming logics [Dij75] to do the derivation systematically not only for our limited language for actions, but also for a substantially more general one.

The weakest precondition predicate $wp_\pi(\phi)$ gives for a program π (effect of an action) and a formula ϕ the weakest (most general) formula $\chi = wp_\pi(\phi)$ so that if a state s satisfies χ, that is $s \models \chi$, then the state obtained from s by executing π will satisfy ϕ.

Example 6.8 If a condition is not affected by the program, then the weakest precondition is the condition itself.

$$wp_{x:=1}(y = 1) = y = 1$$

If the program achieves the condition unconditionally, then the weakest precondition is the constant $\top$.

$$wp_{x:=1}(x = 1) = \top$$

■

Definition 6.9 (Expressions) *Only the constant symbols 0 and 1 are expressions.*

Definition 6.10 (Programs) *1. The empty program ϵ is a program.*

2. *If x is a state variable and f is an expression, then $x := f$ is a program*
3. *If ϕ is a formula over equalities $x = f$ where x is a state variable and f is an expression, and π_1 and π_2 are programs, then eITE(ϕ, π_1, π_2) is a program.*
4. *If π_1 and π_2 are programs, then their sequential composition $\pi_1; \pi_2$ is a program. This program first executes π_2, and then π_2.*
5. *If π_1 and π_2 are programs, then their parallel composition $\pi_1|\pi_2$ is a program. This program first executes π_2, and then π_2.*

Nothing else is a program.

Given a program π and formula ϕ, we can compute the *weakest precondition* $wp_\pi(\phi)$, that has the property, that if it is satisfied, then executing π will end in a state that satisfies ϕ, and for any other formula χ that satisfies the same condition, $\chi \models wp_\pi(\phi)$, that is, it is the *weakest* such formula. It was first investigated by Dijkstra [Dij75].

Definition 6.11 (The Weakest Precondition)

$$wp_\epsilon(\phi) = \phi \tag{6.1}$$

$$wp_{x:=b}(\phi) = \phi \textit{ with } x \textit{ replaced by } b \tag{6.2}$$

$$wp_{eITE(\theta,\pi_1,\pi_2)}(\phi) = (\theta \wedge wp_{\pi_1}(\phi)) \vee (\neg\theta \wedge wp_{\pi_2}(\phi)) \tag{6.3}$$

$$wp_{\pi_1;\pi_2}(\phi) = wp_{\pi_1}(wp_{\pi_2}(\phi)) \tag{6.4}$$

$$\tag{6.5}$$

6.3 Higher Order Representations

Practical modelling languages have concepts that are at a higher level than the propositional logic. The set of state variables is not a collection $x_1, \dots, x_n$ of unstructured variables. Instead, these atomic (Boolean) state variables are obtained from a higher level specification language.

We give an example of such a language.

There are different types of objects, each type t characterized by its domain D_t, the set of objects belonging to the type. The objects themselves are represented as their names, an alphanumeric string.

The state variables are formed by combining a *predicate* symbol P with some combination $(o_1, \dots, o_k) \in D_{t_1} \times \cdots \times D_{t_n}$ of objects associated with the type $(t_1, \dots, t_n)$ of the predicate.

Example 6.12 Let there be only one type *location*, with the domain $D_{location} = \{$ Helsinki, Tampere, Jyväskylä, Oulu $\}$.

Let there be only one predicate *arc*, of type (location,location).

All combinations of locations, obtained as the Cartesian product $D_{location} \times D_{location}$, yield the following state variables.

arc(Helsinki,Helsinki)	arc(Helsinki,Tampere)	arc(Helsinki,Jyväskylä)	arc(Helsinki,Oulu)
arc(Tampere,Helsinki)	arc(Tampere,Tampere)	arc(Tampere,Jyväskylä)	arc(Tampere,Oulu)
arc(Jyväskylä,Helsinki)	arc(Jyväskylä,Tampere)	arc(Jyväskylä,Jyväskylä)	arc(Jyväskylä,Oulu)
arc(Oulu,Helsinki)	arc(Oulu,Tampere)	arc(Oulu,Jyväskylä)	arc(Oulu,Oulu)

■

Similarly to creating state variables from a parametric definition (a predicate), also the transitions can be created from a parametric definition.

Our definition of a parametric transition consists of three components (i, p, e), where

- i is a list $(v_1, t_1), \dots, (v_n, t_n)$ of pairs (v,t) where v is a variable name and t is a type,
- p is a propositional formula with schematic variables as the atomic propositions, and
- e is an effect with schematic variables as state variables.

A parametric transition is instantiated by creating all possible bindings $(v_1, \dots, v_n) \in D_{t_1} \times \cdots \times D_{t_n}$, and for each binding producing a succinct transition.

Example 6.13 We have the *arc* predicate of type (location,location) as discussed before, as well as a unary predicate *isAt* of type *location*.

A schematic transition that moves (a single unspecified object) between locations is defined as (i, p, e) where

- $i =$ (o_1,location),(o_2,location)
- $p =$ isAt(o_1) $\wedge$ arc(o_1,o_2)
- $e = \{$ isAt(o_2), $\neg$ isAt(o_1) $\}$

This schematic transtion can be instantiated in 4×4 different ways, as there are two parameters each with 4 objects in its type domain. One of the instances of the transition is (p, e) where

- $p =$ isAt(Helsinki) $\wedge$ arc(Helsinki,Tampere)
- $e = \{$ isAt(Tampere), $\neg$ isAt(Helsinki) $\}$

Chapter 7

Symbolic Reachability Testing

State-space search is the problem of testing whether a state in a *transition system* is reachable from one or more *initial states*. Transition systems in the most basic cases can be identified with *graphs*, and the state-space search problem in this case is the s-t-reachability problem in graphs.

For small graphs the problem can be solved with standard graph search algorithms such as Dijkstra's algorithm. For graphs with an astronomically high number of states, 10^{10} or more, standard graph algorithms are impractical. Many transition systems can be compactly represented, yet their size as graphs is very high. This is because N Boolean (0-1) state variables together with a state-variable based representation of the possible actions or events can induce a state-space with the order of 2^n reachable states. This is often known as the *combinatorial explosion*. It turns out that the s-t-reachability problem for natural compactly represented graphs is PSPACE-complete [GW83, Loz88, LB90, Byl94].

Classical propositional logic has been proposed as one solution to state-space search problems for very large graphs, due to the possibility of representing and reasoning about large numbers of states with (relatively small) formulas.

7.1 OBDD-Based Symbolic Reachability

The logic-based methods are best understood in terms of relational operations, with formulas representing sets and relations. A basic state space traversal algorithm expressed with relation join and projection is as follows.

1. $i := 0$
2. S_0 := set of initial states
3. $i := i + 1$
4. $S_i := S_{i-1} \cup \pi_2(S_{i-1} \bowtie R)$
5. if $S_i \not\subseteq S_{i-1}$, go to 3.

On step 4, set S_i of those states that are reachable from the initial states by i step or less is computed. This consists of the states in S_{i-1} as well as those states that can be reached from a state in S_{i-1} by one step. These latter states are obtained by $\pi_2(S_{i-1} \bowtie R)$, where the natural join $S_{i-1} \bowtie R$ computes those pairs (s, s') of states where $s \in S_{i-1}$ has distance $i - 1$ and where s' can be reached from s by one step, as determined by the 1-step reachability relation R. From $S_{i-1} \bowtie R$ we obtain those states that are reached from S_{i-1} by projection to the second elements of the pairs, dropping the first element.

In Section 5 we have already shown how set and relation operations can be implemented as formula manipulation, when the sets S_i and the relations R are represented as formulas. Next we show how state space reachability problems can be solved with those techniques.

The first idea is that states are valuations of some set of state variables $X = \{x_1, \dots, x_n\}$. Any set of states can be represented as a formula on X, and any formula on X represents a set of states. Next, the arcs in any transition system (with states represented by valuations of X), representing transitions from states to states is a binary relation on valuations of X. As was shown before, such a relation can be represented as a formula on propositional variables $X \cup X'$, where $X' = \{x'_1, \dots, x'_n\}$. Here we have two propositional variables for every

state variable: one indicating its value before a transition takes place, and the other indicating its value after.

We already saw an example of this kind of formula in Example 5.6, here adapted to the x, x' convention on names of propositional variables.

$$\begin{aligned} inc = & (\neg c \wedge c' \wedge (b \leftrightarrow b') \wedge (a \leftrightarrow a')) \\ & \vee(\neg b \wedge c \wedge b' \wedge \neg c' \wedge (a \leftrightarrow a')) \\ & \vee(\neg a \wedge b \wedge c \wedge a' \wedge \neg b' \wedge \neg c') \\ & \vee(a \wedge b \wedge c \wedge \neg a' \wedge \neg b' \wedge \neg c') \end{aligned}$$

Given a set of states, expressed by a formula ϕ on X, and a transition relation, expressed by a formula ρ on $X \cup X'$, we can compute their natural join by simply forming their conjunction $\phi \wedge \rho$. After this, we can project the resulting relation to the successor states of all possible transitions (s, s') from s to s' by existentially abstracting away all variables in X, by $\exists X.(\phi \wedge \rho)$. This formula has occurrences of variables from X' only. After renaming every $x' \in X'$ to the corresponding $x \in X$, the result is a formula over X.

Corresponding to the relation image operation

$$img_R(S) = \pi_2(S \bowtie R)$$

we now have the logical image operation

$$img_\rho(\phi) = \text{subst}_{X/X'}(\exists X.(\phi \wedge \rho)),$$

as well as a *pre-image* operation

$$preimg_\rho(\phi) = \exists X'.((\text{subst}_{X/X'}(\phi)) \wedge \rho)$$

for computing all the possible predecessor states of a set of states with respect to a relation.

Now we can express the relation breadth-first search algorithm in terms of formulas.

1. $i := 0$
2. $\phi_0 :=$ formula that represents all initial states
3. $i := i + 1$
4. $\phi_i := \phi_{i-1} \vee img_\rho(\phi_{i-1})$;
5. if $\phi_i \not\equiv \phi_{i-1}$, go to 4.

If we are not interested in computing all reachable states, but only testing whether some states in a set G are reachable then we can terminate the computation earlier.

1. $i := 0$
2. $\phi_0 :=$ formula that represents all initial states
3. $i := i + 1$
4. $\phi_i := \phi_{i-1} \vee img_\rho(\phi_{i-1})$;
5. if $\phi_i \not\equiv \phi_{i-1}$ and $\phi_i \wedge G \equiv \bot$, go to 4.

Here line 5 has a slightly stronger condition: computation is only continued if no state in G has been reached.

Finally, when interested in the reachability of a specific state or sets of states, it is often useful to explicitly construct a transition sequence that leads from an initial state to those state. If we have the formulas ϕ_i from the previous computation, this can be achieved by the following procedure. Here we assume that the transition relation formula ρ is a disjunction $\rho_1 \vee \rho_2 \vee \cdots \vee \rho_m$, and we are interested in knowing which of the constituent transitions, which we could call either *events* or *actions*, were taken to reach G. The initial value of i below is the final value of i in the previous algorithm.

1. $T := \text{subst}_{X'/X}(\phi_i \wedge G)$
2. if $i = 0$, terminate.
3. for some $t \in \{1, \ldots, m\}$ such that $S_{i-1} \wedge \rho_t \wedge T \not\equiv \bot$ do
   (a) $T := S_{i-1} \wedge preimg_{\rho_t}(T)$
   (b) $t_{i-1} := t$
4. $i := i - 1$

5. go to 2.

Here the sets $B_1, \ldots, B_i$ are states that can be reached when firing a sequence of transitions $t_1, \ldots, t_i$ so that a state in $\phi_i \wedge G$ is reached in the end.

Notice that only if the constituent transitions ρ_i are *deterministic* (it is a (partial) function, rather than a more general relation) will the firing of $t_1, \ldots, t_i$ guarantee that a state in G is reached in the end and that sets $B_1, \ldots, B_{i-1}$ are visited on the way.

7.2 SAT-Based Symbolic Reachability

OBDD-based symbolic state-space traversal find longer and longer transition sequences by interleaved natural joins and projections, when 1-step transitions are represented by the formula ρ over propositional variables X and X', and the initial states are represented by the formula S_0 over X.

$$\begin{aligned}
S_1 &= \text{subst}_{X/X'}(\exists X.(S_0 \wedge \rho))) \\
S_2 &= \text{subst}_{X/X'}(\exists X.(\text{subst}_{X/X'}(\exists X.(S_0 \wedge \rho))) \wedge \rho))) \\
S_3 &= \text{subst}_{X/X'}(\exists X.(\text{subst}_{X/X'}(\exists X.(\text{subst}_{X/X'}(\exists X.(S_0 \wedge \rho))) \wedge \rho))) \wedge \rho))) \\
&\vdots
\end{aligned}$$

Another way of using the same transition relation formulas is to perform only the natural joins, ignoring the projections that are needed for "forgetting" about the past transition sequence. This will lead to formulas of the following form, when testing the reachability of states expressed by a formula G.

$$S_0 \wedge \text{subst}_{X@1/X'}(\rho) \wedge \text{subst}_{X@1/X,X@2/X'}(\rho) \wedge \text{subst}_{X@2/X,X@3/X'}(\rho) \wedge \cdots \wedge \text{subst}_{X@n/X}(G).$$

The sets $X@i$ for integers i consist of all propositional variables $x@i$ where $x \in X$ is a state variable.

This formula is satisfiable if and only if there is a transition sequence of length n from one of the initial states to one of the goal states.

Example 7.1 The successor relation of 3-bit integers $(000, 001), (001, 010), (010, 011), (011, 100), (100, 101), (101, 110), ($ is represented by the formula

$$\begin{aligned}
inc = & (\neg c \wedge c' \wedge (b \leftrightarrow b') \wedge (a \leftrightarrow a')) \\
& \vee(\neg b \wedge c \wedge b' \wedge \neg c' \wedge (a \leftrightarrow a')) \\
& \vee(\neg a \wedge b \wedge c \wedge a' \wedge \neg b' \wedge \neg c') \\
& \vee(a \wedge b \wedge c \wedge \neg a' \wedge \neg b' \wedge \neg c'),
\end{aligned}$$

and the multiplication by 2 for 3-bit integers (left shift, losing the most significant bit) is represented by the formula

$$ml2 = (a' \leftrightarrow b) \wedge (b' \leftrightarrow c) \wedge \neg c'.$$

To test whether the bit-vector 101 is reachable from bit-vector 000 by four steps by using actions inc and $ml2$ test the satisfiability of the formula

$$\begin{aligned}
& \neg a@0 \wedge \neg b@0 \wedge \neg c@0 \\
& \wedge(inc[X@0/X, X@1/X'] \vee ml2[X@0/X, X@1/X']) \\
& \wedge(inc[X@1/X, X@2/X'] \vee ml2[X@1/X, X@2/X']) \\
& \wedge(inc[X@2/X, X@3/X'] \vee ml2[X@2/X, X@3/X']) \\
& \wedge(inc[X@3/X, X@4/X'] \vee ml2[X@3/X, X@4/X']) \\
& \wedge a@4 \wedge \neg b@4 \wedge c@4
\end{aligned}$$

There is exactly one satisfying assignment.

i	$a@i$	$b@i$	$c@i$	action
0	0	0	0	inc
1	0	0	1	inc or $ml2$
2	0	1	0	$ml2$
3	1	0	0	inc
4	1	0	1	

■

The general scheme followed here involves generating formulas Φ_i for i-step reachability, and testing their satisfiability one by one, starting from $\Phi_0, \Phi_1, \ldots$, until a satisfiable formula Φ_j is found. From the satisfying assignment a transition sequence can be extracted: for every two consecutive time points $i, i+1$, the valuation of the propositional variables in $X@i \cup X@(i+1)$ corresponds to (at least) one of the transitions.

7.2.1 Parallel Actions

The scheme we presented involves one transition at a time. Transition systems are typically described by transition rules, which correspond to the different possible actions or events, as discussed in Section 6.2.

To reduce the length of the state sequences to be considered, it is useful to try to pack more than one action/event transition in one step. Hence the transition relation formula does not represent the firing of one transition rule, but several. In this section we demonstrate how this type of partially ordered action sequences can be represented in the propositional logic. Here the partial ordering refers to the fact that at a given step, the ordering of the actions is not total: actions $a_1, \ldots, a_n$ are taken, but no strict ordering between them is imposed (explicitly).

We consider partial ordering schemes in which the simultaneous actions $a_1, \ldots, a_n$, represented by rules $(c_1, e_1), \ldots, (c_n, e_n)$ taken in state s_t of the state sequence $s_0, \ldots, s_N$ satisfy the following two properties.

- The conditions c_i for $i \in \{1, \ldots, n\}$ all satisfy $s_t \models c_i$.
- Each effect e_i of action a_i causes the same changes as it would cause alone, irrespectively of the other actions taken in parallel.

Action $a_1, \ldots, a_n$ occurring "in parallel" means that they occur in some total ordering $a_{i_1}, a_{i_2}, \ldots, a_{i_n}$ with $\{i_1, \ldots, i_n\} = \{1, \ldots, n\}$.

The effects e of actions.

$x := F(x_1, \ldots, x_n)$	assignment of a state variable $x \in X = \{x_1, \ldots, x_n\}$
$eITE(\phi, e_1, e_2)$	IF-THEN-ELSE conditional
$e_1; e_2$	sequential composition: execute e_1 and then execute e_2
$e_1 : e_2$	parallel composition: execute e_1 and e_2 simultaneously

In contrast to the weakest precondition predicate *wp* (Definition 6.11), what we need to determine the conditions under which a state variable changes. This is to distinguish between the cases in which x will be true because it was true already and did not become false, and in which x is changed to true.

The condition under which x becomes true is denoted by $cc_a(x)$, and the condition under which x becomes false is denoted by $cc_a(\neg x)$.

Definition 7.2

$$cc_\epsilon(x) = \bot \tag{7.1}$$
$$cc_{x:=\phi}(x) = \phi \tag{7.2}$$
$$cc_{y:=\phi}(x) = \bot \quad \textit{when } x \neq y \tag{7.3}$$
$$cc_{x:=\phi}(\neg x) = \neg\phi \tag{7.4}$$
$$cc_{y:=\phi}(\neg x) = \bot \quad \textit{when } x \neq y \tag{7.5}$$
$$cc_{eITE(\phi,e_1,e_2)}(l) = (\phi \wedge cc_{e_1}(l)) \vee (\neg\phi \wedge cc_{e_2}(l)) \tag{7.6}$$
$$cc_{e_1;e_2}(l) = (cc_{e_1}(l) \wedge wp_{e_1}(\neg cc_{e_2}(\bar{l}))) \vee wp_{e_1}(cc_{e_2}(l)) \tag{7.7}$$
$$cc_{e_1:e_2}(l) = cc_{e_1}(l) \vee cc_{e_2}(l) \tag{7.8}$$

The propositional variables used to express the transition relation formulas with parallel action are as follows.

x	value of state variable x
x'	value of state variable x at the next time point
a	action a is executed?

Unlike in the formula in Section 6.2, here we have propositional variables a that indicate whether a given action is taken at the current time point.

To express the possibility of taking an action we need to make sure that the condition part c of the action (c, e) is true: $a \rightarrow c$. For given values for the action variables $a_1, \ldots, a_n$ and the propositional variables $x \in X$ that represent the current state, the following formula determines the new value for each state variable $x \in X$ uniquely.

$$x' \leftrightarrow \left(\bigvee_{a \in A} (a \wedge cc_a(x)) \right) \vee \left(x \wedge \neg \bigvee_{a \in A} (a \wedge cc_a(\neg x)) \right) \tag{7.9}$$

This formula is often represented in a different, but logically equivalent form.

First, we view the equivalence as $x' \leftrightarrow (\Phi_1 \vee (x \wedge \neg \Phi_2))$, where Φ_1 is the condition for x becoming true and Φ_2 is the condition for x becoming false. The formula can be equivalently written as follows.

$$\Phi_1 \rightarrow x' \tag{7.10}$$

$$x \wedge \neg \Phi_2 \rightarrow x' \tag{7.11}$$

$$\neg \Phi_1 \wedge \neg (x \wedge \neg \Phi_2) \rightarrow \neg x' \tag{7.12}$$

The formulas 7.10 simply consists of the following for every action.

$$a \wedge cc_a(x) \rightarrow x' \tag{7.13}$$

These formulas are called (positive) *effect axioms*, as they indicate for each action when they will make x true at the next time point.

The formulas 7.11 are known as (positive) *frame axioms* for historical reasons. They indicate the conditions under which x remains true when no action turns it false. They are often written in the form $\Phi_2 \wedge x \rightarrow x'$ or $x' \vee \neg x \vee \neg \Phi_2$.

Formula 7.12 can be rewritten to any of the following implications.

$$\neg \Phi_1 \wedge (\neg x \vee \Phi_2) \rightarrow \neg x' \tag{7.14}$$

$$(\neg \Phi_1 \wedge \neg x) \vee (\neg \Phi_1 \vee \Phi_2) \rightarrow \neg x' \tag{7.15}$$

The last version can be split to two implications.

$$\neg \Phi_1 \wedge \neg x \rightarrow \neg x' \tag{7.16}$$

$$\neg \Phi_1 \wedge \Phi_2 \rightarrow \neg x' \tag{7.17}$$

The implication 7.16 is the (negative) frame axiom, analogous to the (positive) frame axiom 7.11, and indicating the conditions under which x is false and remains false.

The implication 7.17 is the negative counterpart of the effect axiom 7.10. It contains the seemingly superfluous conjunct $\neg \Phi_1$. We have assumed that a state variable cannot simultaneously become true and false. Hence the seemingly superfluous conjunct could be removed. However, our equivalence 7.9 entails that x will be true whenever the conditions for it becoming true hold (by formula 7.10), and x will only become false if conditions for becoming false hold and conditions for becoming true *do not* hold, as is made explicit in formula 7.17.

7.2.2 Alternative Meanings of Parallelism

However, we need something more than the above formulas to guarantee that any set of simultaneous actions can be meaningfully executed.

Example 7.3 Consider actions

- $a_1 = (x, y := 0)$
- $a_2 = (y, x := 0)$

As a formula this is

$$\begin{array}{ll} x' \leftrightarrow (x \wedge \neg a_2) & a_1 \rightarrow x \\ y' \leftrightarrow (y \wedge \neg a_1) & a_2 \rightarrow y \end{array}$$

If both a_1 and a_2 are true, both x' and y' will be false. This does not correspond to any sequential execution of the actions, as the actions disable each other. ■

Only such sets of actions are acceptable that can be executed in some total ordering. To guarantee that such a total ordering exists we need some auxiliary concepts.

Definition 7.4 *Action* (p_1, e_1) *(possibly)* disables (p_2, e_2) *if some* $x \in X$ *occurs in* e_1 *in an assignment* $x := \cdots$ *and in* p_2.

Definition 7.5 *Action* (p_1, e_1) affects (p_2, e_2) *if some* $x \in X$ *occurs in* e_1 *in an assignment* $x := \cdots$ *and in* ϕ *for some eITE*$(\phi, \cdots, \cdots)$ *in* p_2.

Note that these definitions are approximate. E.g. $(\top, (a := 1))$ "disables" $(a, (x := 1))$, although it really doesn't. These definitions could be strengthened e.g. by considering whether the state variables occur positively or negatively.

Example 7.6 $(\top, (x := 0))$ disables $(x \vee y, (a := 1))$ ■

Example 7.7 $(\top, (x := 0))$ affects $(\top, (x := 1))$ ■

Example 7.8 $(\top, (x := 0))$ affects $(\top, \textit{eITE}(x, (a := 1), (b := 1)))$ ■

The simplest condition for simultaneous actions is that they can be executed in any total ordering. This is achieved by the concept of interference that expresses a symmetric dependency relation.

Definition 7.9 (Interference) a_1 *and* a_2 *interfere if*

1. a_1 *disables or affects* a_2*, or*
2. a_2 *disables or affects* a_1

To guarantee that execution in any order is possible and leads to the same state the following suffices. If a_1 and a_2 interfere, then include the following formula.

$$\neg a_1 \vee \neg a_2$$

A weaker, directed definition of dependence, sometimes allowing many more actions in parallel, is the following.

Definition 7.10 (Directed Interference) a_1 *interferes with* a_2 *if* a_1 *disables or affects* a_2.

Now, even a set of actions that have interfere can be taken in parallel, as long as the directed interference relation is acyclic. A particularly simple way of achieving this is based on an arbitrarily total ordering $a_1, \ldots, a_n$ on the set of all actions. If the action a_i interferes with a_j and $j > i$, then we need include the following formula.

$$\neg a_i \vee \neg a_j$$

Now, given a state s in which a set $P \subseteq \{a_1, \ldots, a_n\}$ of actions is applicable (their conditions are true, and their effects are non-conflicting) and the above constraints are satisfied, these actions can be executed in an ordering indicated by the ordering of the set of all actions. This is because each action is applicable in s, and if an action disables (or can disable) another action, then the latter action is earlier in the ordering.

Example 7.11 Consider the nesting of Russian matryoshka dolls, expressed as the following actions.

$$\begin{aligned} a_1 &= (\text{out1} \wedge \text{out2} \wedge \text{empty2}, (\text{1in2} := 1; \text{out1} := 0; \text{empty2} := 0)) \\ a_2 &= (\text{out2} \wedge \text{out3} \wedge \text{empty3}, (\text{2in3} := 1; \text{out2} := 0; \text{empty3} := 0)) \\ a_3 &= (\text{out3} \wedge \text{out4} \wedge \text{empty4}, (\text{3in4} := 1; \text{out3} := 0; \text{empty4} := 0)) \end{aligned}$$

A doll can be put inside another if neither is inside another doll, and the bigger one is empty. As a result, the bigger will not be empty any more, and the smaller is inside the bigger one. Notice that action a_3 disables a_2, and a_2 disables a_1.

Consider the initial state in which the four dolls are initially un-nested. With the stricter definition of parallelism, they can be nested one by one, by taking the actions a_1, a_2, a_3 in this order. But with the more relax definition of parallelism, the nesting can be performed in one step, as all actions are initially applicable, none of the effects conflict, and the directed interference relation is acyclic. The ordering of the actions is still implicitly the same a_1, a_2, a_3, but in terms of the definition of parallelism they can be taken at the same time. ■

References

For the solution of reachability problems in discrete transition systems the use of propositional logic as a representation of sets and relations was first proposed in the works by Coudert et al. [CBM90, CM90] and Burch et al. [BCL$^+$94].

The use of these logic-based techniques was possible with (ordered, reduced) Binary Decision Diagrams [Bry92], generally known as BDDs or OBDDs. BDDs are a restrictive normal form that guarantees a *canonicity* property: any two logically equivalent Boolean formulas are represented by a unique BDD. The canonicity property guarantees the polynomial-time solvability of many central problems about Boolean formulas, including *equivalence*, *satisfiability*, and *model-counting*. Though in some cases necessarily (even exponentially) larger than unlimited Boolean formulas, BDDs also in practice often lead to compact enough representations when unlimited formulas would in practice be too large.

Other similar normal forms, with useful polynomial-time operations, exist, including Decomposable Negation Normal form (DNNF) [Dar01, Dar02]. There are interesting connections between this type of normal forms and the Davis-Putnam procedure [HD05].

Problems with scalability of BDD-based methods was well-understood already in mid-1990s. Another way of using the propositional logic for solving state-space search problems was proposed by Kautz and Selman [KS92] in 1992, and in 1996 they demonstrated it to be often far better scalable than alternative search methods for solving state-space search problems in AI [KS96]. This method was based on mapping one reachability query, whether a state satisfying a given goal formula is reachable in a given number of steps from a specified initial state. The method suffered from the large size of the formulas far less than BDDs, and was soon shown to be in many cases far better scalable than BDD-based methods for example in Computer Aided Verification, for solving the model-checking problem [BCCZ99].

In Sections 5.2 and 6.2 we only presented the most basic translation of action sequences into propositional formulas, with the exactly one action at each step. This is what is known as a *sequential* encoding. This representation suffers from the possibility of sequences of actions $a_1, \ldots, a_n$ that are independent of each other, and which therefore can be permuted to every one of $n!$ different orders. These representations are what is typically used in OBDD-based reachability methods. Having several actions can simultaneous decreases the horizon length. This substantially improves the scalability of SAT-based methods in the main applications including the planning, diagnosis, and model-checking problems. The parallel representations, with multiple simultaneous events or actions, come from Kautz and Selman's works [KS92, KS96]. The more relaxed notion of parallelism, with weaker constraints and potential for more parallelism, by requiring that at least one ordering, rather than all total orderings, are executable, were investigated later [DNK97, RHN06]. The total orderings can be often chosen so that no constraints limiting the parallelism are needed at all [RHN06].

That the SAT problems in testing the existence of transition sequences of increasing lengths are related in a way that can be used for speeding up the SAT solving process, was recognized and utilized by Eén and Sörensson in *incremental SAT solving* [ES03], in which the clauses learned by the CDCL algorithm for the formula for horizon length i could be used also for horizon length $i + 1$, with potentially high speed-ups. Another way of speeding up the search for the satisfiable formulas is to solve the sequence of formulas in parallel, rather than one by one by considering the horizon length i only after the formula for horizon length $i - 1$ has been determined to be unsatisfiable [SS07, Rin04, RHN06].

Chapter 8

Modal Logics

Many concepts are not definable truth-functionally. Consider the following.

- It is possible that ϕ holds.
- At the next time point ϕ holds.
- After executing the program π, the formula ϕ will hold.

In all of these, the truth of ϕ does not in all cases determine the truth of the sentence in question. Modal logics are logics that, in addition to the usual Boolean connectives, also include *modal operators* for defining concepts such as the ones mentioned above. A major difference between modal operators and Boolean connectives is that the former are not truth-functional: whereas the truth of a formula like $\phi_1 \vee \phi_2$ is a function of the truth-values of ϕ_1 and ϕ_2, the truth of a modal formula $\Box\phi$ is *not* a function of the truth of ϕ. If a is true, the formula $\Box a$ could be true or false, depending on the circumstances. This difference becomes obvious from the way the model-theory (semantics) of modal logics is defined.

The interesting and intriguing thing is that the generalization of the propositional logic for the above, and many other, concepts can be achieved in the same kind of formal framework. This framework involves defining the truth of generalized formulas in a *graph*, in which the nodes are propositional valuations, and the nodes are connected to each other by arcs. This framework is related to the notion of transition systems with state variables (Definition 6.3), as we will see later.

In the next sections we first consider basic uni-modal logics with one modality only (the dual modal operators $\Box$ and $\Diamond$). Then we consider multi-modal logics with multiple modalities, temporal logics which talk about time, as well as dynamic logics which allow talking about programs and paths in graphs.

8.1 Modal Logics with One Modality

Modal logics with a single modality were the earliest ones, first used in philosophical investigations in concepts like *necessity* and *possibility*, and later with other concepts. These logics have a modal operator $\Box$, with $\Box\phi$ read as "it is necessary that ϕ", and its dual operator $\Diamond$ with $\Diamond\phi$ read as "it is possible that ϕ". Depending on the application, these operators have multiple readings.

- $\Box\phi$: It is necessary that ϕ (Alethic logic)
- $\Box\phi$: It is obligatory that ϕ (Deontic logic)
- $\Box\phi$: It is known that ϕ (Epistemic Logic)
- $\Box\phi$: It is believed that ϕ (Epistemic Logic)
- $\Box\phi$: Always in the future ϕ (Temporal Logic)
- $[\pi]\phi$: After executing π necessarily ϕ (Dynamic Logic)

As pointed out, the dual concept of *necessity* is *possibility*, with the operators $\Box$ and $\Diamond$ related by $\Diamond\phi \equiv \neg\Box\neg\phi$: something is possible if its negation is not necessary. Similarly, "obligatory" is dual to "permissible", and "always" is dual to "sometimes". There is no unique dual for "is believed" or "is known" in English, but the dual concepts could be expressed as "it is considered possible (as far as knowledge/belief is concerned)". Notice that these two modal operators are related similarly to the universal and existential quantification operators $\forall x.\phi \equiv \neg\exists x.\neg\phi$. It

would be sufficient to use only one of these pairs of operators, but for convenience both are used.

Beliefs, for example, do not in general characterize the actual world exactly, and as beliefs may be false, they might not concern the actual world at all. And, it turns out, beliefs can be characterized by the set of worlds that are considered possible by the person/agent in question:

1. If I consider only such worlds possible in which ϕ is true, then I believe ϕ.
2. If I don't believe ϕ (this is different from believing $\neg\phi$!), then I consider at least one world possible in which A is false.

Clearly, I could not normally believe both A and $\neg A$ (the formula $(\Box A) \wedge (\Box\neg A)$ should in general be false) and none of the possible worlds satisfy both A and $\neg A$, but I might consider both A and $\neg A$ as possible as far as my beliefs are concerned ($(\Diamond A) \wedge (\Diamond\neg A)$ could well be true), and there could well be two possible worlds, one of which satisfying A and the other $\neg A$.

In deontic logics, in which the operator $\Box$ is interpreted as "it is obligatory that", the set of worlds that are referred to when interpreting sentences like "it is obligatory that" are those that satisfy all the laws and norms that we are considering. ϕ is obligatory if and only if ϕ is true in all of those worlds. If ϕ is permitted, then there is at least one world in which ϕ is true. Further, in temporal logics (which are considered in more detail later in Section 8.3 and 8.4), the worlds that are referred to by operators like "always in the future" are those representing the state of the world in later time points.

A simple semantics for a fragment of modal logics could be given by having a valuation w for the actual world and a set of valuations W for all the possible worlds. Then we could say that $\Box\phi$ is true if ϕ is true in all worlds/valuations in W. However, this definition works only when ϕ does not contain modal operators. For the more general case, with the possibility of nesting modal operators, a more sophisticated semantics is needed.

Starting in 1959, Saul Kripke (a 19-year old Harvard undergraduate) developed a semantics for a wide range of modal logics based on possible worlds [Kri63a, Kri63b, Kri65]. The semantics is very intuitive and powerful, and it is the main reason for the wide use of different modal logics in different kinds of applications, varying from philosophical logic to computer science. The idea is to view the set of all possible worlds as a graph. Formally, a *frame* is a pair $F = \langle W, R\rangle$, where W is a set of possible worlds and $R \subseteq W \times W$ (the accessibility relation) is a binary relation on W. A frame just describes how the possible worlds are related to each other, it does not yet say anything about the truth and falsity of formulae.

Let X be a finite set of propositional variables. A *model* M *based on the frame* F is a 4-tuple $M = \langle W, R, X, v\rangle$ where W and R are like in F, and $v : W \times X \to \{0, 1\}$ is a valuation that assigns the truth value *true* or *false* (respectively represented by 1 and 0) to each propositional variable in every possible world.

Definition 8.1 (Uni-modal Kripke model) *A* Kripke model $M = \langle W, R, X, v\rangle$ *consists of*

- *a set* W *of* worlds*,*
- *a* accessibility relation $R \subseteq W \times W$*,*
- *a set* X *of* atomic formulas*,*
- *a valuation* $v : W \times X \to \{0,1\}$ *of propositional variables in worlds.*

For convenience, we will write $v_w(x)$ *for* $v(w, x)$*.*

The truth of a formula is defined with respect to one of the nodes of a Kripke model.

We write $M \models_w \phi$ for the truth of a modal formula ϕ in world w in a Kripke model M.

Definition 8.2 (Truth of uni-modal formulas) *Let* $M = \langle W, R, X, v\rangle$ *be a Kripke model.*

1. $M \models_w x$ *iff* $v_w(x) = 1$
2. $M \models_w \phi \wedge \psi$ *iff* $M \models_w \phi$ *and* $M \models_w \psi$
3. $M \models_w \phi \vee \psi$ *iff* $M \models_w \phi$ *or* $M \models_w \psi$
4. $M \models_w \neg\phi$ *iff not* $M \models_w \phi$
5. $M \models_w \Box\phi$ *iff* $M \models_{w'} \phi$ *for all* $w' \in W$ *such that* wRw'
6. $M \models_w \Diamond\phi$ *iff* $M \models_{w'} \phi$ *for at least one* $w' \in W$ *such that* wRw'

All but the last two lines in the above definition are essentially the same as the truth-definition for the propositional

logic. The second last line defines $\Box\phi$ to be true in the world w if and only if ϕ is true in all worlds w' that are accessible from the current world w according to the relation R.

The necessity operator requires that something holds in *all* accessible states, whereas the possibility requires only that something holds in *at least one* accessible state. So it is not surprising that the relation between $\Box$ and $\Diamond$ is analogous to the *universal* $\forall$ and *existential* $\exists$ quantifications which are related by $\forall x.\phi \equiv \neg\exists x.\neg\phi$ and $\exists x.\phi \equiv \neg\forall x.\neg\phi$.

8.1.1 Validity in classes of frames

The following formulae all valid in *all* frames, meaning that they automatically follow if we are to have Kripke models as the semantics of the logic.

- $\Box\top$
- $\Box(a \rightarrow b) \rightarrow (\Box a \rightarrow \Box b)$ (the formula schema **K**)
- $\Box(a \wedge b) \leftrightarrow (\Box a \wedge \Box b)$
- $\Diamond(a \vee b) \leftrightarrow (\Diamond a \vee \Diamond b)$
- $\Box(a \rightarrow b) \rightarrow (\Diamond a \rightarrow \Diamond b)$

Also, as we will see later when we axiomatically formalize different modal logics, the set of formulae that are valid in a class of frames have the following properties.

- If ϕ is valid in a class C of frames, then also $\Box\phi$ is valid in C.
- If ϕ and $\phi \rightarrow \phi'$ are valid in a class C of frames, then also ϕ' is valid in C.
- If ϕ is valid in a class C of frames, then any substitution instance of ϕ is valid in C.

In the first two cases this is simply because the set of formulae true in one model have the analogous properties.

8.1.2 Properties of the accessibility relation

What makes Kripke models interesting is that there is a close connection between several important axioms schemata and the formal properties of the accessibility relations.

Let us consider the following axiom schemata.

1. **T**: $\Box\varphi \rightarrow \varphi$ (alternatively $\varphi \rightarrow \Diamond\varphi$)
2. **4**: $\Box\varphi \rightarrow \Box\Box\varphi$ (alternatively $\Diamond\Diamond\varphi \rightarrow \Diamond\varphi$)
3. **5**: $\Diamond\varphi \rightarrow \Box\Diamond\varphi$
4. **B**: $\varphi \rightarrow \Box\Diamond\varphi$
5. **D**: $\Box\varphi \rightarrow \Diamond\varphi$ (alternatively $\Box\varphi \rightarrow \neg\Box\neg\varphi$)
6. **L**: $\Box(\varphi \wedge \Box\varphi \rightarrow \chi) \vee (\Box(\chi \wedge \Box\chi \rightarrow \varphi)$

Now, these schemata exactly correspond to the following properties of the accessibility relation. We say that a frame is reflexive, for example, if its accessibility relation is reflexive.

1. reflexivity (wRw for every world w),
2. transitivity ($wRu \wedge uRv$ implies wRv),
3. euclidicity ($wRu \wedge wRv$ implies uRv),
4. symmetry (wRu implies uRw),
5. seriality (for every w there exists v with wRv)
6. weak connectedness ($sRt \wedge sRu$ implies that either tRu, $t = u$ or uRt)

For example, if the frame F if reflexive, then the axiom schema **T** is valid in F. Also the opposite holds. If the axiom schema **T** is valid in a frame F, then F is reflexive. We prove these things and leave the analogous proofs for the other properties as an exercise.

Theorem 8.3 *Axiom schema* **T** *is valid in the class of reflexive frames.*

Proof: Let F be a reflexive frame. Let M be a model based on F and let w be any world in M. If $\Box\varphi$ is not true w, then axiom $\mathbf{T}$ is true in w. If $\Box\varphi$ is true in w, then φ is true in all worlds that are accessible from w. Since the accessibility relation is *reflexive*, w is among the accessible worlds, i.e., φ is true in w. This means that also in this case $\mathbf{T}$ is true w. This means, T is true in all worlds in all models based on reflexive frames, which we wanted to show. □

Theorem 8.4 *If T is valid in a frame F, then F is reflexive.*

Proof: We prove the contraposition of the claim, that is, start from the negation of the conclusion and derive the negation of the antecedent.

Assume that F is not reflexive. We will construct a model based on F that falsifies $\mathbf{T}$. Because F is not reflexive, there is a world w such that not wRw. Construct a model M such that $M \not\models_w p$ and $M \models_v p$ for all v such that wRv. Now $M \models_w \Box p$ and $M \not\models_w p$, and hence $M \not\models_w \Box p \to p$. □

The next table summarizes the correspondences between relational properties and modal axioms.

Name	Property	Axiom schema
K	–	$\Box(\varphi \to \psi) \to (\Box\varphi \to \Box\psi)$
T	reflexivity	$\Box\varphi \to \varphi$
4	transitivity	$\Box\varphi \to \Box\Box\varphi$
5	euclidicity	$\Diamond\varphi \to \Box\Diamond\varphi$
B	symmetry	$\varphi \to \Box\Diamond\varphi$
D	seriality	$\Box\varphi \to \Diamond\varphi$
L	weak connectedness	$\Box(\varphi \land \Box\varphi \to \chi) \lor (\Box(\chi \land \Box\chi \to \varphi)$

8.1.3 Different uni-modal logics

Because the different properties considered above are to some extent mutually independent, there is a possibility to construct many different modal logics corresponding to different classes of frames. Interestingly, the early modal logics S4 and S5, originally developed without reference to Kripke-style semantics by C. I. Lewis in 1910, are some of these logics.

One axiom schema is common to most logics that correspond to a class of frames, namely the axiom schema

$$\mathbf{K} = \Box(a \to b) \to (\Box a \to \Box b).$$

We call the corresponding logic K. In general, we simply give the list of names of axioms schemata to name a logic, but many of the logics of historical importance have their own name that is usually used.

$$\begin{aligned} & K \\ S4 &= KT4 \\ S5 &= KT5 \\ K4.3 &= K4L \\ S4.3 &= KT4L \\ &\vdots \end{aligned}$$

These logics have been applied for different purposes. The choice of axiom schemata depends on how one exactly wants to interpret the modal operators. Some properties often used for formalizations of natural language notions like knowledge, belief, and obligation, among others, are summarized in the following table. In the column for each axiom we have indicated whether it should be included in the given type of logic (marked with “Y”). This is based on reading the axiom with the corresponding meaning (for example “$\Box\phi \to \Box\Box\phi$ for epistemic logics is “If I know ϕ, then I know that I know ϕ.”).

logics	$\Box$	$\Diamond = \neg\Box\neg$	K	T	4	5	B	D
aletic	necessary	possible	Y	Y	?	?	?	Y
epistemic	known	possible	Y	Y	Y	Y	N	Y
doxastic	believed	possible	Y	N	Y	Y	N	Y
deontic	obligatory	permitted	Y	N	N	?	?	Y
temporal	always (in future)	sometimes	Y	Y	Y	N	N	Y

8.1.4 Undefinable properties

Many properties of binary relations exactly correspond to modal logic formulae. However, there are simple properties of binary relations that do not have a corresponding axiom schema. One such property is irreflexivity. For showing that there is no modal axiom schema corresponding to irreflexivity we use p-morphisms.

Definition 8.5 *Let $M_1 = \langle W_1, R_1, X, v_1\rangle$ and $M_2 = \langle W_2, R_2, X, v_2\rangle$ be models over the same propositional variables X. Then a* p-morphism *from M_1 to M_2 is a function $f : W_1 \to W_2$ such that*

- *sR_1t implies $f(s)R_2f(t)$,*
- *$f(s)R_2u$ implies that there is $t \in W_1$ such that sR_1t and $f(t) = u$, and*
- *$M_1 \models_s p$ iff $M_2 \models_{f(s)} p$ for all $p \in X$ and $s \in W_1$.*

So, every world in M_1 is mapped to a world in M_2, and the arrows between world in M_1 are similarly mapped to arrows in M_2. Notice that M_2 may have worlds that do not correspond to any world in M_1, and that there may be arrows in M_2 that do not correspond to any arrow in M_1 as long as the starting world does not correspond to any world in M_1.

A *p-morphism* between the frames $F_1 = \langle W_1, R_1\rangle$ and $F_2 = \langle W_2, R_2\rangle$ is a function $f : W_1 \to W_2$ that satisfies the first two conditions of the definition of p-morphisms between models.

Lemma 8.6 *Let f be a p-morphism from $M_1 = \langle W_1, R_1, X, v_1\rangle$ to $M_2 = \langle W_2, R_2, X, v_2\rangle$. Then for all $s \in W_1$ and all formulae ϕ, $M_1 \models_s \phi$ if and only if $M_2 \models_{f(s)} \phi$.*

Proof: By induction on the structure of the formula.

Base case: ϕ is an atomic formula. For any $s \in W_1$ by definition of p-morphisms, $M_1 \models_s \phi$ if and only if $M_2 \models_{f(s)} \phi$.

Inductive cases:

1. Let $\phi = \psi \to \psi'$. By induction hypothesis $M_1 \models_s \psi$ iff $M_2 \models_s \psi$, and $M_1 \models_s \psi'$ iff $M_2 \models_s \psi'$. From this the result directly follows.
2. Let $\phi = \neg\psi$. By induction hypothesis $M_1 \models_s \psi$ iff $M_2 \models_s \psi$. From this the result directly follows.
3. Let $\phi = \Box\psi$. By induction hypothesis $M_1 \models_{s'} \psi$ iff $M_2 \models_{f(s')} \psi$ for all worlds $s' \in W_1$, including those for which sRs'. From this the result directly follows.

Therefore, $M_1 \models_s \phi$ if and only if $M_2 \models_{f(s)} \phi$, for all $s \in W_1$ and all formulae ϕ. □

Lemma 8.7 *Let there be a surjective p-morphism f from frame F_1 to F_2. Then for any formula ϕ, $F_1 \models \phi$ implies $F_2 \models \phi$.*

Proof: Assume there is a model $M_2 = \langle W_2, R_2, X, v_2\rangle$ based on F_2 such that $M_2 \not\models_t \phi$ for some $t \in W_2$. Define the model M_1 based on F_1 by

$$M_1 \models_s p \text{ iff } M_2 \models_{f(s)} \quad \text{for all } s \in W_1$$

Take any s such that $f(s) = t$ (such an s exists because f is a surjection.) Now by Lemma 8.6 $M_1 \not\models_s \phi$. □

Theorem 8.8 *The class of irreflexive frames is not defined by any axiom schema.*

Proof: We show that there is an irreflexive frame F_1 that has a p-morphic image that is not irreflexive. Consider the frame $\langle N, < \rangle$ of natural numbers ordered by the less than relation. The frame $\langle \{1\}, \{(1,1)\} \rangle$ is its p-morphic image but not irreflexive.

Hence the class of irreflexive frames is not closed under p-morphic images. But the class of frames defined by any axiom schema is closed under p-morphic images by Lemma 8.7. This shows that the class of irreflexive frames is not definable by any axiom schema. □

8.1.5 Logical consequence in modal logics

Logical consequence $\Sigma \models \phi$ in the classical propositional logic is defined as truth of the consequent ϕ in all models in which the antecedents Σ are true.

In modal logic there are several alternative definitions of logical consequence, because the truth of the antecedent can be considered in several different ways: do we require that the antecedents are true in all worlds of the model and then check whether ϕ is true in all worlds of a model, or do we just check that ϕ is true in those worlds of a model in which Σ are true?

These two possibilities lead to two different notions of logical consequence. The first is known as the *global* definition of logical consequence, and the second as the *local* definition of logical consequence. In this lecture we use the local definition.

Definition 8.9 (Logical consequence) *The formula ϕ is a* logical consequence *of a set Σ of formulae in a class C of frames (or models) (written $\Sigma \models_C \phi$) iff for all worlds s in all models M based on $F \in C$ (or all models M in C) $M \models_s \Sigma$ implies $M \models_s \phi$*

Clearly, logical consequence from the empty set $\Sigma = \emptyset$ in a class of frames coincides with validity in that class of frames.

8.2 Multi-Modal Logics

We can easily have multiple modalities by having multiple modal-operators each corresponding to a different accessibility relation.

Definition 8.10 (Multi-modal Kripke model) *A* Kripke model *$M = \langle W, D, R, X, v \rangle$ consists of*

- *a set W of* worlds*,*
- *a set D of* modalities*,*
- *an assignment $R : D \rightarrow 2^{W \times W}$ of accessibility relations to each modality,*
- *a set X of* atomic formulas*,*
- *a valuation $v : W \times X \rightarrow \{0,1\}$ of propositional variables in worlds.*

Now we have multiple modal operators $\Box_d$, one for each $d \in D$.

Definition 8.11 (Truth of multi-modal formulas) *1. $M \models_w x$ iff $v_w(x) = 1$*

2. *$M \models_w \phi \wedge \psi$ iff $M \models_w \phi$ and $M \models_w \psi$*
3. *$M \models_w \phi \vee \psi$ iff $M \models_w \phi$ or $M \models_w \psi$*
4. *$M \models_w \neg\phi$ iff not $M \models_w \phi$*
5. *$M \models_w \Box_d\phi$ iff $M \models_{w'} \phi$ for all $w' \in W$ such that $wR(d)w'$*

We consider multi-modal logics of belief and knowledge of several agents. Each modality $d \in D$ correspond to one of the agents, and the operator $\Box_d$ refers to the beliefs of the agent: $\Box_d\phi$ means that the agent d belief that ϕ, that is, ϕ is true in all worlds that the agent considers possible. What the agent considers possible is represented by the accessibility relation $R(d)$.

The beliefs of an agent depend on the agent's accessibility relation.

Commonly used axioms in epistemic logics.

1. $\Box_i(p \to q) \to (\Box_i p \to \Box_i q)$ (Distribution Axiom for Knowledge)
2. $\Box_i p \to p$ (Knowledge/Truth Axiom)
3. $\Box_i p \to \Box_i \Box_i p$ (Positive Introspection)
4. $\neg\Box_i p \to \Box_i \neg\Box_i p$ (Negative Introspection)

We are often interested in the multi-agent epistemic logic $S5_n$ that satisfies these principles. The accessibility relation of each agent is then of course reflexive, transitive and euclidic.

Knowledge generalization rule:

If $M \models \phi$, then $M \models_s \Box_i \phi$ for every M, ϕ and s.

For many applications it is useful to introduce further modal operators based on the operator K.

Define the operator "everybody knows that ϕ" by

$$M \models_s E\phi \text{ iff } M \models_s \Box_i \phi \text{ for every } i \in \{1, \dots, n\}$$

This of course corresponds to the axioms schema

$$E\phi \leftrightarrow (\Box_1 \phi \wedge \cdots \wedge \Box_n \phi).$$

Define the operator "it is common knowledge that ϕ" by

$$M \models_s C\phi \text{ iff } M \models_s E^i \phi \text{ for every } i \geq 1$$

Here E^i means a sequence $EEEEE \dots E$ of i Es.

Common knowledge has the following property

$$C\phi \leftrightarrow E(\phi \wedge C\phi).$$

There is a simple and very useful graph-theoretic view of these two operators in terms of paths in the Kripke-model.

Definition 8.12 *A world t is reachable from a world in s in k steps if there is a sequence of worlds $s_0, s_1, \dots, s_k$ so that*

1. *$s_0 = s$ and $s_k = t$, and*
2. *$s_i R_j s_{i+1}$ for every $i \in \{0, \dots, k-1\}$ and any $j \in \{1, \dots, n\}$.*

Lemma 8.13 *$M \models_s E^k \phi$ iff $M \models_t \phi$ in every world t that is reachable from s in k steps.*

$M \models_s C\phi$ iff $M \models_t \phi$ in every world t that is reachable from s in any number of steps.

8.3 Linear Temporal Logic LTL

Linear temporal logic is one of the best-known and most used modal logics in computer science. It can be viewed as a sub-logic of the branching time temporal logic CTL$*$ (Section 8.4), omitting the path quantifiers A and E that refer to the branching of time.

Since LTL cannot refer to alternative futures, the truth of its formulas are interpreted over individual paths or state sequences, and in the CTL$*$ terminology, all LTL formulas are *path formulas*, meaning that they talk about truth of facts along a single timeline, often corresponding to a path in a tree or graph of computations.

The semantics of LTL, similarly to the temporal logics CTL and CTL$*$, is not given in the standard framework of Kripke models as for the uni-modal and multi-modal logics.

Definition 8.14 *A linear temporal model $M = (X, \sigma)$ consists of propositional variables X and an infinite sequence of propositional valuations $\sigma = (v_0, v_1, v_2, \dots)$*

- $M \models_i x$ if and only if $v_i(x) = 1$
- $M \models_i \neg\phi$ if and only if $M \not\models_i \phi$
- $M \models_i \phi_1 \vee \phi_2$ if and only if $M \models_i \phi_1$ or $M \models_i \phi_2$
- $M \models_i \phi_1 \wedge \phi_2$ if and only if $M \models_i \phi_1$ and $M \models_i \phi_2$
- $M \models_i G\phi$ if and only if $M \models_j \phi$ for all $j \geq i$
- $M \models_i F\phi$ if and only if $M \models_j \phi$ for some $j \geq i$
- $M \models_i X\phi$ if and only if $M \models_{i+1} \phi$
- $M \models_i \phi U \psi$ if and only if for some $j \geq i$, $M \models_j \psi$ and $M \models_k \phi$ for all $k \in \{i, \ldots, j-1\}$

8.4 Branching Time Temporal Logics CTL and CTL∗

In many applications of temporal logics in computer science the temporal logic models are based on transition systems, best understood as arbitrary Kripke structures, instead of linearly ordered structures like in the linear temporal logic.

An arbitrary Kripke model naturally represents not just one linear ordering of worlds, but a tree, representing a high number of linear orderings corresponding to the paths in the tree. In this setting it is natural to extend the linear logic language further to be able to express properties of such trees. This leads to branching time temporal logics, of which we discuss the computation tree logic CTL∗, and its sublogics LTL and CTL [Eme90].

8.4.1 The language of CTL∗

1. Every propositional variable $x \in X$ is a formula.
2. If φ and ψ are formulae, then so are $\neg\varphi$, $\varphi \vee \psi$, $\varphi \wedge \psi$, $\varphi \rightarrow \psi$, $F\varphi$, $G\varphi$, $X\varphi$, $\varphi U \psi$, $A\varphi$, $E\varphi$.

The semantics of the operators F, G, X and U is known from the linear time temporal logics. The *path quantifiers* quantify over all of the possibly infinite number of computation paths in a transition system. Formulae $E\varphi$ with existential path quantification say that φ must be true on at least one computation path, and formulae $A\varphi$ say that φ must be true on all computation paths through the current state.

Example 8.15 CTL∗ formulae with path quantification are useful in expressing many correctness properties programs, controllers or other systems might have to fulfil.

$AG\neg$failure	A failure state is never reached.
$AG(EF$restart$)$	From every state of the system it is possible to reach the *restart* state. That is, the system cannot get stuck anywhere.
$AG($request $\rightarrow AF$ack$)$	Any request will be eventually acknowledged.

■

8.4.2 The semantics of CTL∗

A model in the computation tree logic is a tuple $M = \langle W, R, X, v \rangle$ where W is a set of states (possible worlds), R is an accessibility relation so that for every $w \in W$ there is $w' \in W$ such that wRw', X is the set of propositional variables, and $v : W \times X \rightarrow \{0, 1\}$ is a valuation that assigns truth values to propositional variables in every state.

The truth of CTL∗ formulae are in general defined over a computation path, an infinite sequence of states in a model. However, for *state formulae* that only have E and A as their outermost modal operators the truth can also be defined with respect to a state.

Definition 8.16 *A* computation path *is an infinite sequence of states* $\lambda = w_0 w_1 w_2 \ldots$ *such that* $w_i R w_{i+1}$ *for each* $i \geq 0$. *For a path* $\lambda = w_0 w_1 w_2 \ldots$, $\lambda[i]$ *denotes the state* w_i. *For a path* λ, λ^i *denotes the* i*th suffix of a path* λ,*that is,* λ^i *is defined by* $\lambda^i[j] := \lambda[i + j]$ *for all* $j \geq 0$.

We often write briefly “path” instead of “computation path”.

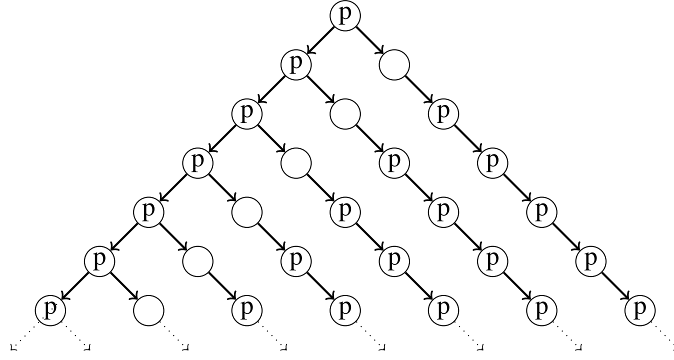


Figure 8.1: Branching temporal model in which $A\mathcal{F}\mathcal{G}p$ is true but $A\mathcal{F}A\mathcal{G}p$ is false

The truth-definition of formulae in states $w \in W$ or over computation paths λ is as follows.

1. $M \models_w x$ iff $v(w, x) = 1$
2. $M \models_w \neg\varphi$ iff $M \not\models_w \varphi$
3. $M \models_w \varphi \vee \psi$ iff $M \models_w \varphi$ or $M \models_w \varphi$
4. $M \models_w A\varphi$ iff $M \models_\lambda \varphi$ for every path λ such that $\lambda[0] = w$
5. $M \models_w E\varphi$ iff $M \models_\lambda \varphi$ for some path λ such that $\lambda[0] = w$
6. $M \models_\lambda \varphi$ iff $M \models_{\lambda[0]} \varphi$ for state formulae φ
7. $M \models_\lambda \neg\varphi$ iff $M \not\models_\lambda \varphi$
8. $M \models_\lambda \varphi \vee \psi$ iff $M \models_\lambda \varphi$ or $M \models_\lambda \varphi$
9. $M \models_\lambda G\varphi$ iff $M \models_{\lambda^i} \varphi$ for every $i \geq 0$
10. $M \models_\lambda F\varphi$ iff $M \models_{\lambda^i} \varphi$ for some $i \geq 0$
11. $M \models_\lambda X\varphi$ iff $M \models_{\lambda^1} \varphi$
12. $M \models_\lambda \varphi U \psi$ iff there is $i \geq 0$ such that $M \models_{\lambda^i} \psi$ and $M \models_{\lambda^j} \varphi$ for every $j \in \{0, \ldots, i-1\}$.

8.4.3 Computation tree logic CTL

CTL is a simpler variant of CTL$*$ that does not allow arbitrary combination of temporal operators. Specifically, every occurrence of operators X, F, G and U is *preceded by the path quantifier E or A*. Essentially, we have 8 modal operators AX AF AG AU EX EF EG EU. Hence every formula is a state formula and can be evaluated with respect to a state without referring to a designated computation path. This restriction on modal operators reduces the class of properties expressible in CTL, but it also makes CTL much easier to handle computationally.

8.4.4 Relations between CTL, LTL and CTL$*$

Every LTL formula and every CTL formulae has an equivalent CTL$*$ formula, but LTL and CTL are not comparable in expressivity, that is, there are formulae in both of these logics that express properties of computation trees not expressible in the other logic.

For the LTL formula $F(p \wedge Xp)$ (interpreted universally with implicit prefix A) there is no equivalent CTL formula. See the tree in Figure 8.1 that satisfies $A\mathcal{F}\mathcal{G}p$ (Every path through it satisfies $\mathcal{F}\mathcal{G}p$), but it doesn't satisfy $A\mathcal{F}A\mathcal{G}p$ (No node on the leftmost path satisfies $A\mathcal{G}p$.)

For the CTL formula $AG(EFp)$ there is no equivalent LTL formula.

The conjunction $AF(p \wedge Xp) \wedge AG(EFp)$ of the above two formulae is a $CTL*$ formula that has no counterpart in LTL nor in CTL. Hence CTL$*$ is strictly more expressive than either of the these two logics.

8.5 Dynamic Logic

Both Floyd and Hoare [Flo67, Hoa69] have used correctness assertions

$$\{\varphi\}\alpha\{\psi\}$$

for stating that executing the program α in a state satisfying φ will always end in a state satisfying ψ. There are proof systems that allow the step-wise derivation of such assertions about programs α from similar assertions on components of α.

The possible steps a program can take can be understood as a Kripke structure, and this idea leads to the definition of dynamic logics [Pra76, FL77, FL79], in which we have an infinite number of modal operators $[\alpha]$ for all possible programs α, and the above correctness assertion corresponds to the dynamic logic formula

$$\varphi \rightarrow [\alpha]\psi.$$

8.5.1 Propositional dynamic logic PDL

The language of the propositional dynamic logic is based on a set X of propositional variables and a set A of atomic programs.

Then formulas and programs based on A and X are defined recursively as follows.

1. The constant true $\top$ is a formula.
2. Every propositional variable $x \in X$ is a formula.
3. If φ and ψ are formulae, then so are $\neg\varphi$, $\varphi \vee \psi$, $\varphi \wedge \psi$, $\varphi \leftrightarrow \psi$ and $\varphi \rightarrow \psi$.
4. Every atomic program $a \in A$ is a program.
5. If α and β are programs and φ is a formula, then $\alpha \cup \beta$, $\alpha;\beta$, $\alpha*$ and $\varphi?$ are programs.
6. If α is a program and φ is a formula, then $[\alpha]\varphi$ and $\langle\alpha\rangle\varphi$ are formulae.

The intuitive meaning of $\alpha \cup \beta$ is nondeterministic choice: execute either α or β. The construct $\alpha;\beta$ is sequential composition: first execute α and then execute β. The $\alpha*$ programs correspond to looping: execute α zero or any finite number of times. Conditional execution is expressed by $\varphi?$: the execution continues only if φ is true.

When considering ordinary programming languages, the atomic programs would most naturally be assignment statements that take us from one possible world to another in which one or more propositional variables have a different truth-value. Of course, the definition of PDL allows more general types of atomic programs, for example ones that are nondeterministic and therefore do not define exactly one unique successor for each possible world.

Familiar constructs from procedural programming languages can be defined in terms of the PDL program constructs as follows.

$$\begin{aligned}
\text{if } \varphi \text{ then } \alpha \text{ else } \beta &\equiv (\varphi?;\alpha) \cup (\neg\varphi?;\beta)\\
\text{while } \varphi \text{ do } \alpha &\equiv (\varphi?;\alpha)*;\neg\varphi?\\
\text{repeat } \alpha \text{ until } \varphi &\equiv \alpha;(\neg\varphi?;\alpha)*;\varphi?
\end{aligned}$$

8.5.2 The semantics of PDL

Models for dynamic logic are Kripke structures with an accessibility relation for every atomic program.

Definition 8.17 *A model in the propositional dynamic logic is a tuple $M = \langle W, R_{a_1}, \ldots, R_{a_n}, A, X, v\rangle$ where*

- *W is a set of possible worlds,*
- *A is a set of n atomic programs,*
- *$R_a \subseteq W \times W$ for every $a \in A$ is an accessibility relation,*
- *X is the set of propositional variables, and*
- *$v : W \times X \rightarrow \{0,1\}$ is a valuation that assigns truth values to propositional variables in every possible world.*

Analogously to modal logics defined earlier we can talk about the frame $F = \langle W, R_{a_1}, \ldots, R_{a_n}\rangle$.

The truth-definition of modal formulae in possible worlds $w \in W$ is as follows (the truth-definition of non-modal connectives is just like earlier.)

- $M \models_w [\alpha]\varphi$ iff $M \models'_w \varphi$ for every possible world w' such that $wR_\alpha w'$.
- $M \models_w \langle\alpha\rangle\varphi$ iff $M \models'_w \varphi$ for some possible world w' such that $wR_\alpha w'$.

Here the difference to earlier modal logics is that there is an infinite number of different accessibility relations for all of the non-atomic programs α. The accessibility relations for these programs are defined as follows.

1. The accessibility relations of atomic programs $a \in A$ are given explicitly by the model M.
2. If α and β have respectively accessibility relations R_α and R_β, then
   (a) $R_{\alpha\cup\beta} = R_\alpha \cup R_\beta$ (union of relations),
   (b) $R_{\alpha;\beta} = \{\langle w, w'\rangle | u \in W, wR_\alpha u, uR_\beta w'\}$ (composition of relations),
   (c) $R_{\alpha*} = (R_\alpha)*$ (the reflexive transitive closure of a relation), and
   (d) $R_{\varphi?} = \{\langle w, w\rangle | w \in W, M \models_w \varphi\}$.

8.5.3 Axiom system for PDL

The propositional dynamic logic includes the propositional logic, and hence its set of theorems includes all the theorems of the classical propositional logic and its (modal) substitution instances, and it is closed under substitution and modus ponens.

To formalize PDL axiomatically, one additionally needs axioms for describing the properties of the modal operators.

Definition 8.18 (Propositional dynamic logic) *PDL is the smallest logic that is closed under the necessitation/generalization rule*

If $\vdash_{PDL} \varphi$, then $\vdash_{PDL} [\alpha]\varphi$ for any program α.

and contains the following axiom schemata for programs α and β and formulae/propositions p and q.

1. $[\alpha](p \rightarrow q) \rightarrow ([\alpha]p \rightarrow [\alpha]q)$
2. $[\alpha;\beta]p \leftrightarrow [\alpha][\beta]p$
3. $[\alpha \cup \beta]p \leftrightarrow [\alpha]p \wedge [\beta]p$
4. $[\alpha*]p \leftrightarrow p \wedge [\alpha][\alpha*]p$
5. $[\alpha*](p \rightarrow [\alpha]p) \rightarrow (p \rightarrow [\alpha*]p)$
6. $[q?]p \leftrightarrow (q \rightarrow p)$

8.5.4 Relations to other modal logics and Hoare logic

Relation to LTL

Variants of the temporal operators F, X, G and U can be translated into the propositional dynamic logic. Essentially, temporal logics can be understood as dynamic logics that do not make the different programs explicit, or alternatively, have only one atomic program a. The following is an embedding of LTL operators in PDL.

operator	translation into PDL
$F\varphi$	$\langle a*\rangle\varphi$
$X\varphi$	$\langle a\rangle\varphi$
$G\varphi$	$[a*]\varphi$
$\varphi U\psi$	$\langle(\varphi?; a)*\rangle\psi$

Relation to unimodal logics

There are simple embeddings of some the unimodal logics with operators $\Box$ and $\Diamond$ in the dynamic logic. The interesting thing here is that what is the properties of the relations in these uni-modal logics, are represented in the PDL modal operator.

logic	translation of $\Box\varphi$ into PDL
K	$[a]\varphi$
S4	$[a*]\varphi$
S5	$[(a \cup a^{-1})*]\varphi$

Here we need the converse connective $^{-1}$ that is given a semantics as follows.

$$R_{\alpha^{-1}} = \{\langle t, s\rangle | \langle s, t\rangle \in \alpha\}.$$

The converse program α^{-1} of α computes the computation of α in reverse from the end states back to the starting states.

With the above translation the satisfiability problems in the logics in question can be directly translated into PDL. Notice that the relations in the PDL frames may be arbitrary (K frames), and the reflexivity and transitivity for S4 and reflexivity, transitivity and symmetry for S5 are properties of the PDL modal operator, and need not be properties of the frames in question.

Relation to Hoare logic

The Hoare logic is a programming logic that may be used for proving the correctness of programs in procedural programming languages [Hoa69]. There are different variants of the logic, depending on the definition of programs and the language in which the correctness assertions are expressed. Here we consider a propositional variant of the logic, in which the assertions are propositional formulae.

We can consider atomic programs a that are associated with assertions $\{\varphi\}a\{\psi\}$ explaining their behavior. Assertions about more complex programs can be proved by using these simplest assertions and inference rules concerning the program constructs.

The Hoare logic has the following inference rules.

$$\frac{\{\varphi\}\alpha\{\sigma\}, \{\sigma\}\beta\{\psi\}}{\{\varphi\}\alpha;\beta\{\psi\}} \tag{8.1}$$

$$\frac{\{\varphi \wedge \sigma\}\alpha\{\psi\}, \{\neg\varphi \wedge \sigma\}\beta\{\psi\}}{\{\sigma\} \text{ if } \varphi \text{ then } \alpha \text{ else } \beta\{\psi\}} \tag{8.2}$$

$$\frac{\{\varphi \wedge \psi\}\alpha\{\psi\}}{\{\psi\} \text{ while } \varphi \text{ do } \alpha\{\neg\varphi \wedge \psi\}} \tag{8.3}$$

$$\frac{\varphi' \rightarrow \varphi, \{\varphi\}\alpha\{\psi\}, \psi \rightarrow \psi'}{\{\varphi'\}\alpha\{\psi'\}} \tag{8.4}$$

The counterparts of these inference rules hold in PDL.

Chapter 9

Model-Checking

Model-checking is the problem of testing whether $M \models \phi$ holds for a given model M and a formula ϕ, that is, evaluating the truth-value of a formula.

Model-checking for the propositional logic is easy, as it only involves looking up the truth-values of the atomic propositions and then computing bottom-up the values of all sub-formulas according to the truth-tables of the different connectives.

For modal logics the model-checking problem becomes more difficult. First, there is a Kripke model instead of single valuation, and second, in some logics there is an exponential number of different paths through the Kripke model which all needs to be accounted for. For this reason the complexity of model-checking for linear temporal logics like LTL and their extensions like CTL$*$ are far harder.

9.1 Model-Checking Algorithms

Uni-modal logics like K can be model-checked with a simple labeling procedure, in which the worlds of the Kripke model are labeled with all subformulas of ϕ that are true in it. This can be done in O($|\phi| \times |W|$) time, considering each subformula and world only once.

1. Label each world w with $L(w) = \{x \in X | M \models_w x\}$
2. Consider all subformulas ϕ' of ϕ in the order of increasing length:
   For every world w, assign $L(w) := L(w) \cup \{\phi'\}$ if
   - $\phi' = \psi_1 \wedge \psi_2$ and $\psi_1 \in L(w)$ and $\psi_2 \in L(w)$, or
   - $\phi' = \psi_1 \vee \psi_2$ and $\psi_1 \in L(w)$ or $\psi_2 \in L(w)$, or
   - $\phi' = \neg\psi_1$ and $\psi_1 \notin L(w)$, or
   - $\phi' = \Box\psi$ and $\psi \in L(w')$ for all w' such that wRw', or
   - $\phi' = \Diamond\psi$ and $\psi \in L(w')$ for some w' such that wRw'.

logic	logical consequence	model-checking
propositional logic	co-NP-complete	P-complete
K	PSPACE-complete	P-complete
S4	PSPACE-complete	P-complete
S5	co-NP-complete	P-complete
PDL	EXP-complete	P-complete
CTL	EXP-complete	P-complete
CTL$*$	2-EXP-complete	PSPACE-complete
LTL	PSPACE-complete	PSPACE-complete

Table 9.1: Complexity of Logical Consequence and Model-Checking

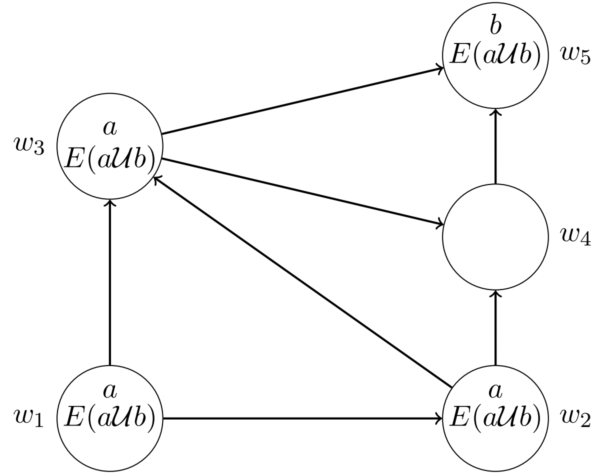


Figure 9.1: Labeling for $E(a\mathcal{U}b)$

1: **procedure** EU(α,β)
2: $T := \{w \in W | \beta \in L(w)\}$;
3: **for each** $w \in T$ **do** $L(w) := L(w) \cup \{E(\alpha\mathcal{U}\beta)\}$;
4: **while** $T \neq \emptyset$ **do**
5: take any $w \in T$;
6: $T := T \backslash \{w\}$;
7: **for each** t such that tRw **do**
8: **if** $\alpha \in L(t)$ and $E(\alpha\mathcal{U}\beta) \notin L(t)$ **then**
9: $L(t) := L(t) \cup \{E(\alpha\mathcal{U}\beta)\}$;
10: $T := T \cup \{t\}$;
11: **end if**
12: **end for**
13: **end while**

Figure 9.2: Procedure for Model-Checking Formulas $E(\phi\mathcal{U}\psi)$

9.1.1 CTL Model-Checking

Temporal logics with state formulas only, which do not require making any path through the temporal model explicit, can be model-checked with a similar labeling procedure. CTL temporal operators like $E\mathcal{G}$ and $E(\phi\mathcal{U}\psi)$ require a more complicated procedure: $E\mathcal{G}$ implicitly refers to an infinite path that exhibits itself as a cycle in the temporal model, and $E(\phi\mathcal{U}\psi)$ refers to a long path. For $E\mathcal{G}$ we need to detect cycles in the graph, whereas a procedure for $E(\phi\mathcal{U}\psi)$ is suggested by observing that the formula is equivalent $\psi \vee (\phi \wedge E\mathcal{X}(E(\phi\mathcal{U}\psi)))$.

All other CTL operators are reducible to $E\mathcal{X}$, $E\mathcal{G}$, $E\mathcal{U}$ by the following equivalences.

$$E\mathcal{F}\varphi \equiv E(\top\mathcal{U}\varphi) \tag{9.1}$$
$$A\mathcal{F}\varphi \equiv \neg E\mathcal{G}\neg\varphi \tag{9.2}$$
$$A\mathcal{G}\varphi \equiv \neg E(\top\mathcal{U}\neg\varphi) \tag{9.3}$$
$$A\mathcal{X}\varphi \equiv \neg E\mathcal{X}\neg\varphi \tag{9.4}$$
$$A(\varphi\mathcal{U}\psi) \equiv \neg E(\neg\psi\mathcal{U}(\neg\varphi \wedge \neg\psi)) \wedge \neg E\mathcal{G}\neg\psi \tag{9.5}$$

Labeling for $E(\phi\mathcal{U}\psi)$ can be done after labeling for ψ and ϕ have been completed. First, label all those worlds with $E(\phi\mathcal{U}\psi)$ that have label ψ. Then, label all those worlds with $E(\phi\mathcal{U}\psi)$ that include label ϕ and that have a successor with label $E(\phi\mathcal{U}\psi)$. This is procedure is illustrated in Figure 9.1 and given as pseudo-code in Figure 9.2.

Labeling for $E\mathcal{G}\phi$ requires checking for infinite paths in the temporal model.

1. Let $W_\phi = \{w \in W | \phi \in L(w)\}$.

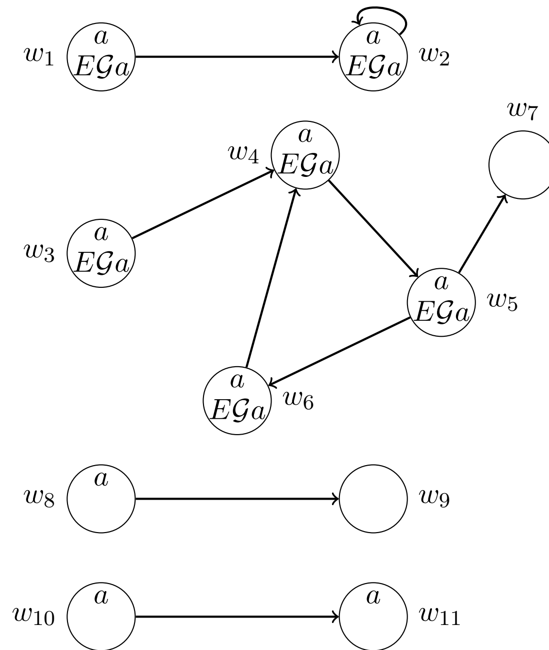


Figure 9.3: Procedure for Model-Checking $E\mathcal{G}\phi$

2. Let $G = \langle W_\phi, R \cap (W_\phi \times W_\phi)\rangle$.
3. Find the strongly connected components of G.
4. For each SCC C such that $|C| > 1$ or wRw for some $w \in C$, do $L(w) := L(w) \cup \{E\mathcal{G}\phi\}$ for all $w \in C$.
5. For each $w \in W_\phi$, if $E\mathcal{G}\phi \in L(w')$ for some w' such that wRw' then $L(w) := L(w) \cup \{E\mathcal{G}\phi\}$.

This procedure is illustrated in Figure 9.3 and pseudo-code for it is given in Figure 9.4.

9.2 A model-checking algorithm for LTL

The model-checking algorithms for the temporal logics LTL and CTL$*$ with pure path formulas not directly prefixed with path quantifiers like in CTL are conceptually more complicated, and also computationally much more expensive than the algorithm we gave for CTL.

The reason for this additional complexity is that we cannot talk about the truth of formulas in a state without

1: **procedure** EG(α)
2: $S' := \{w \in W | \alpha \in L(w)\}$;
3: $SCC := \{C | C$ is a SCC of $\langle S', R \cap (S' \times S')\rangle, |C| \geq 1$ or there is $w \in C$ with $wRw\}$;
4: $T := \{w \in C | C \in SCC\}$;
5: **for each** $w \in T$ **do** $L(w) := L(w) \cup \{E\mathcal{G}\alpha\}$;
6: **while** $T \neq \emptyset$ **do**
7:   take any $w \in T$;
8:   $T := T \backslash \{w\}$;
9:   **for each** $t \in S'$ such that tRw **do**
10:     **if** $E\mathcal{G}\alpha \notin L(t)$ **then**
11:       $L(t) := L(t) \cup \{E\mathcal{G}\alpha\}$;
12:       $T := T \cup \{t\}$;
13:     **end if**
14:   **end for**
15: **end while**

Figure 9.4: Procedure for Model-Checking $E\mathcal{G}\phi$

making a reference to a particular path in the transition system. This causes a problem because the number of different paths starting in a given state is often infinite. The number of finite paths of length n can be exponential in n. At a first look this might seem to imply that we need to consider quantification over an infinite number of paths, but fortunately this turns out not to be the case. It will suffice to consider only an exponential number of different kinds of paths starting from the given state, without uniquely representing these paths in detail.

Consider a formula φ to be model-checked. For every state of the transition system, we need only consider all the possible values the subformulas of φ, and formulas closely related to these subformulas, may have. For example, when we are interested in the truth of a subformula $\psi\mathcal{U}\chi$ in a state s, we are interested in the truth of ψ and χ in s, and the truth of $\psi\mathcal{U}\chi$ in the successor states of s, but it does not matter why $\psi\mathcal{U}\chi$ is true in the successor states, that is, whether ψ and χ are true there or not.

There is a simple construction that uses the above ideas to provide a relatively efficient model-checking algorithm for LTL. When testing $\mathcal{M} \models_s \varphi$ by the algorithm, the runtime is exponential on the size of φ (because of the possibly exponential number of different kinds of paths that have to be considered), but only polynomial in the size of $\mathcal{M}$. Because the formulas expressing useful properties of transition systems are often small, the exponentiality in the size of the formula is usually not a problem, and hence the algorithm is applicable to very big transition systems.

Consider LTL formulas with only the temporal operators $\mathcal{X}$ and U. The operators $\mathcal{F}$ and $\mathcal{G}$ are reducible to these operators by the following equivalences.

$$\mathcal{F}\varphi \equiv \top\mathcal{U}\varphi \tag{9.6}$$

$$\mathcal{G}\varphi \equiv \neg(\top\mathcal{U}\neg\varphi) \tag{9.7}$$

Only the subformulas of the formulas φ to be model-checked and a small number of formulas closely related to them need to be considered by the model-checking algorithm. This is the set CL(φ).

Definition 9.1 *The* closure *CL(φ) of a formula φ is the smallest set of formulas such that*

1. $\varphi \in CL(\varphi)$,
2. $\top \in CL(\varphi)$,
3. $\psi \in CL(\varphi)$ *iff* $\neg\psi \in CL(\varphi)$,
4. *if* $\psi_1 \vee \psi_2 \in CL(\varphi)$ *then* $\psi_1 \in CL(\varphi)$ *and* $\psi_2 \in CL(\varphi)$,
5. *if* $\mathcal{X}\psi \in CL(\varphi)$ *then* $\psi \in CL(\varphi)$,
6. *if* $\neg\mathcal{X}\psi \in CL(\varphi)$ *then* $\mathcal{X}\neg\psi \in CL(\varphi)$,
7. *if* $\psi_1\mathcal{U}\psi_2 \in CL(\varphi)$ *then* $\{\psi_1, \psi_2, \mathcal{X}(\psi_1\mathcal{U}\psi_2)\} \subseteq CL(\varphi)$.

Above we identify $\neg\neg\psi$ and ψ so that the third rule does not cause the size of the set to become infinite.

The number of formulas in CL(φ) is linear in the size of φ: in addition to subformulas of φ and their negations it contains at most one additional formula for each subformula of form $\psi_1\mathcal{U}\psi_2$ and $\neg\mathcal{X}\psi$.

Next we can proceed with the construction of the graph structure that represents all the different kinds of infinite paths in the Kripke model $\mathcal{M}$. This graph is obtained from $\mathcal{M}$ by taking all of its states and making several copies of them, corresponding to all the possible executions they could be a part of.

Definition 9.2 *Let $\mathcal{M} = \langle S, R, V\rangle$ be a Kripke model on which we are testing $\mathcal{M} \models_s \varphi$ for $s \in S$. Then the* tableau[1] *of $\mathcal{M}$ is the graph $\langle S', R'\rangle$ where S' consists of all pairs $\langle s, K\rangle$ such that $s \in S$ and*

1. $K \subseteq CL(\varphi) \cup \mathcal{P}$,
2. $p \in K$ *iff* $V(s,p) = 1$, *for every* $p \in \mathcal{P}$,
3. $\psi \in K$ *iff* $\neg\psi \notin K$, *for every* $\psi \in CL(\varphi)$,
4. $\psi_1 \vee \psi_2 \in K$ *iff* $\psi_1 \in K$ *or* $\psi_2 \in K$, *for every* $\psi_1 \vee \psi_2 \in CL(\varphi)$,
5. $\neg\mathcal{X}\psi \in K$ *iff* $\mathcal{X}\neg\psi \in K$, *for every* $\neg\mathcal{X}\psi \in CL(\varphi)$,
6. $\psi_1\mathcal{U}\psi_2 \in K$ *iff* $\psi_2 \in K$ *or* $\{\psi_1, \mathcal{X}(\psi_1\mathcal{U}\psi_2)\} \subseteq K$, *for every* $\psi_1\mathcal{U}\psi_2 \in CL(\varphi)$.

[1] This is not related to the proof procedures with analytic/semantic tableaux, and we use the word "tableau" for historical reasons.

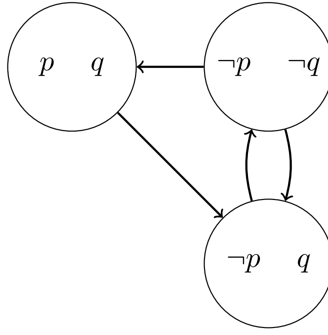


Figure 9.5: A transition system

and $\langle s, K\rangle R' \langle s', K'\rangle$ *whenever*

1. sRs'
2. *for every* $\mathcal{X}\psi \in CL(\varphi)$, $\mathcal{X}\psi \in K$ *iff* $\psi \in K'$.

Essentially, we have taken a model $\mathcal{M}$ and made all the different kinds of paths that could go through a state s in $\mathcal{M}$ explicit in the elements $\langle s, K\rangle$ of the graph $\langle S', R'\rangle$. The formulas in K describe the relevant properties of one particular class of paths through the state s, and the number of these classes is finite even though the number of all paths through s may be infinite.

Example 9.3 Consider the transition system (Kripke model) $\langle S, R, V\rangle$ in Figure 9.5. We will model-check

$$\varphi = E(\neg p\mathcal{U}(p \wedge q)).$$

The unnegated formulas in CL(φ) are

$$\top \quad p \quad q \quad p \wedge q \quad \neg p\mathcal{U}(p \wedge q) \quad \mathcal{X}(\neg p\mathcal{U}(p \wedge q))$$

Of these, only p, q, $\alpha = \neg p\mathcal{U}(p \wedge q)$ and $\mathcal{X}\alpha$, are not truth-functionally determined by simpler formulas.

The potential nodes in the tableau $\langle S', R'\rangle$ therefore correspond to sets of formulas obtained by negating some of $\{p, q, \alpha, \mathcal{X}\alpha\}$. Not all of these sets however occur in a node in the graph: for example, we cannot have $\{p, q, \neg\alpha\} \subseteq K$ for any K, nor $\{\neg p, \neg\alpha, \mathcal{X}\alpha\} \subseteq K$. The nodes in $\langle S', R'\rangle$ correspond to the following sets.

$$\{p, q, \alpha, \mathcal{X}\alpha\} \subseteq K_1$$
$$\{p, q, \alpha, \neg\mathcal{X}\alpha\} \subseteq K_2$$

$$\{\neg p, q, \alpha, \mathcal{X}\alpha\} \subseteq K_3$$
$$\{\neg p, q, \neg\alpha, \neg\mathcal{X}\alpha\} \subseteq K_4$$

$$\{\neg p, \neg q, \alpha, \mathcal{X}\alpha\} \subseteq K_5$$
$$\{\neg p, \neg q, \neg\alpha, \neg\mathcal{X}\alpha\} \subseteq K_6$$

The nodes S' in the graph $\langle S', R'\rangle$ are therefore

$$S' = \{\langle s_1, K_1\rangle, \langle s_1, K_2\rangle, \langle s_2, K_3\rangle, \langle s_2, K_4\rangle, \langle s_3, K_5\rangle, \langle s_3, K_6\rangle\}$$

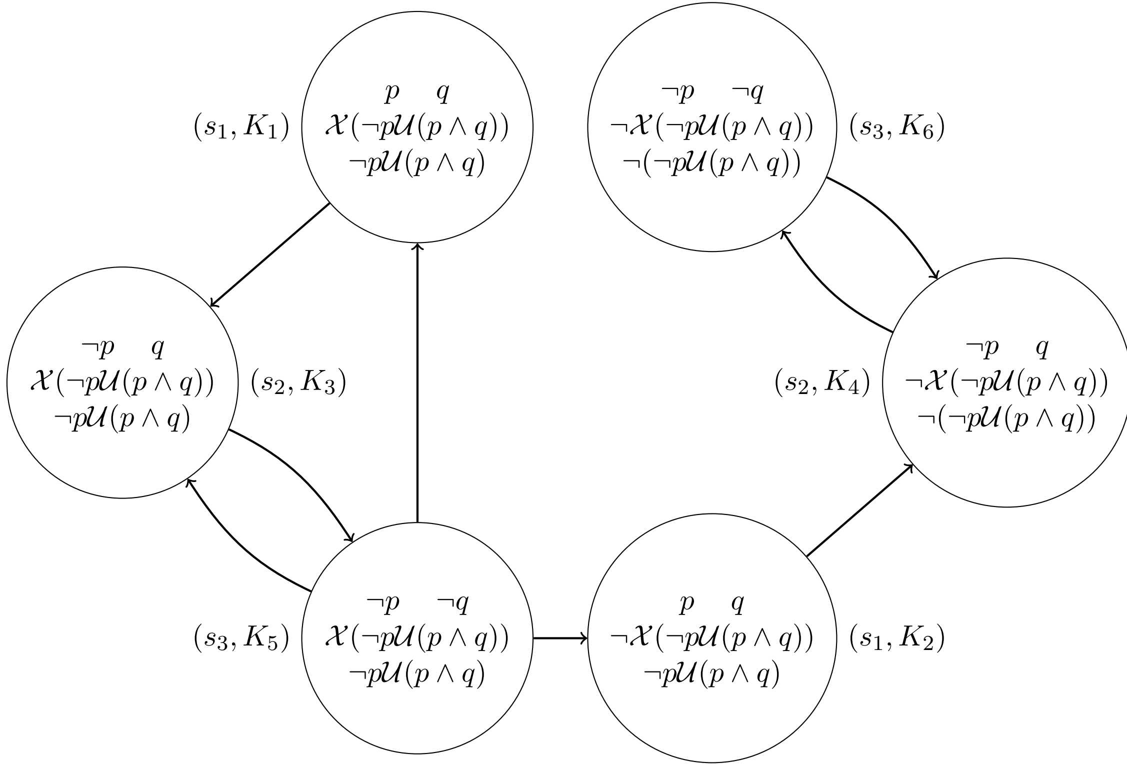


Figure 9.6: The tableau for the transition system

and the arcs described by R' are

$$\langle s_1, K_1\rangle R' \langle s_2, K_3\rangle$$
$$\langle s_1, K_2\rangle R' \langle s_2, K_4\rangle$$

$$\langle s_3, K_5\rangle R' \langle s_1, K_1\rangle$$
$$\langle s_3, K_5\rangle R' \langle s_1, K_2\rangle$$

$$\langle s_2, K_3\rangle R' \langle s_3, K_5\rangle$$
$$\langle s_2, K_4\rangle R' \langle s_3, K_6\rangle$$

$$\langle s_3, K_5\rangle R' \langle s_2, K_3\rangle$$
$$\langle s_3, K_6\rangle R' \langle s_2, K_4\rangle$$

The graph $\langle S', R'\rangle$ is depicted in Figure 9.6.

■

Notice that the sets K are maximally consistent subsets of $\mathrm{CL}(\varphi) \cup \mathcal{P} \cup \{\neg p | p \in \mathcal{P}\}$. For each $\langle s, K\rangle \in S'$ we have $\mathcal{M} \models_s K \cap \mathcal{P}$, that is, the atomic propositions true in s are contained in K.

We also have for any formula $\psi \in \mathrm{CL}(\varphi)$ with occurrences of the temporal operator $\mathcal{X}$ only: $\mathcal{M} \models_\lambda \psi$ for some path λ such that $\lambda[0] = s$ if and only if $\psi \in K$ and there is a path starting from $\langle s, K\rangle$ having length at least the depth of nesting of $\mathcal{X}$ in ψ.

Our goal is to have this generalized to arbitrary formulas, that is, $\mathcal{M} \models_\lambda K$ for some path λ starting from s. What remains to be done is to have this for formulas with occurrences of U.

For formulas $\psi_1 \mathcal{U} \psi_2$ we need to guarantee that eventually a state in which ψ_2 is true is reached. Put differently, we have to identify infinite paths on which ψ_1 is always true and $\psi_1 \mathcal{U} \psi_2$the formula is false because ψ_2 will never be true. For this purpose we introduce the notion of eventuality sequences.

Definition 9.4 *An* eventuality sequence π *is an infinite path in* $\langle S', R'\rangle$ *such that if* $\psi_1 \mathcal{U} \psi_2 \in K$ *for some* $\langle s, K\rangle$ *on* π*, then there is* $\langle s', K'\rangle$ *on* π *not before* $\langle s, K\rangle$ *such that* $\psi_2 \in K'$*.*

Lemma 9.5 *Let $\mathcal{M}$ be a Kripke model, φ a formula, and $\langle S', R' \rangle$ the tableau for $\mathcal{M}$ and φ. Then $\mathcal{M} \models_s E\varphi$ if and only if there is an eventuality sequence starting at $\langle s, K \rangle$ such that $\varphi \in K$.*

Proof: So assume there is an eventuality sequence $\langle s_0, K_0 \rangle, \langle s_1, K_1 \rangle, \ldots$ starting at $\langle s, K \rangle = \langle s_0, K_0 \rangle$ such that $\varphi \in K$. Let $\lambda = s_0, s_1 \ldots$, that is, a computation path in $\mathcal{M}$.

We show that for every $\psi \in \mathrm{CL}(\varphi)$, $\mathcal{M} \models_{\lambda^i} \psi$ iff $\psi \in K_i$.

The proof is by induction on the structure of the formulas in $\mathrm{CL}(\varphi)$.

Base case 1, $\psi = p$ for some $p \in \mathcal{P}$:

By definition of $\langle s_i, K_i \rangle$, $V(s_i, p) = 1$ iff $p \in K_i$.

Inductive case 1, $\psi = \neg\psi'$:

By the induction hypothesis $\mathcal{M} \models_{\lambda^i} \psi'$ iff $\psi' \in K_i$. By definition of $\langle s_i, K_i \rangle$, now $\psi' \in K_i$ iff $\neg\psi' \notin K_i$. Hence $\mathcal{M} \models_{\lambda^i} \neg\psi'$ iff $\neg\psi' \in K_i$.

Inductive case 2, $\psi = \psi_1 \vee \psi_2$:

By the induction hypothesis $\mathcal{M} \models_{\lambda^i} \psi_j$ iff $\psi_j \in K_i$ for $j \in \{1, 2\}$. By definition of $\langle s_i, K_i \rangle$, $\psi \in K_i$ iff $\psi_1 \in K_i$ or $\psi_2 \in K_i$. Hence $\psi \in K_i$ iff $\mathcal{M} \models_{\lambda^i} \psi$.

Inductive case 3, $\psi = \mathcal{X}\psi'$:

By the definition of R', $\langle s_i, K_i \rangle R' \langle s_{i+1}, K_{i+1} \rangle$ holds because $\mathcal{X}\psi' \in K_i$ iff $\psi' \in K_{i+1}$. By the induction hypothesis $\psi' \in K_{i+1}$ iff $\mathcal{M} \models_{\lambda^{i+1}} \psi'$. By truth-definition of $\mathcal{X}$, $\mathcal{M} \models_{\lambda^{i+1}} \psi'$. iff $\mathcal{M} \models_{\lambda^i} \mathcal{X}\psi'$. Hence $\mathcal{X}\psi' \in K_i$ iff $\mathcal{M} \models_{\lambda^i} \mathcal{X}\psi'$.

Inductive case 4, $\psi = \psi_1 \mathcal{U} \psi_2$:

Assume $\psi_1 \mathcal{U} \psi_2 \in K_i$, either $\psi_2 \in K_i$ or $\{\psi_1, \mathcal{X}(\psi_1 \mathcal{U} \psi_2)\} \subseteq K_i$. If $\psi_2 \in K_i$ then by the induction hypothesis $\mathcal{M} \models_{\lambda^i} \psi_2$ and hence $\mathcal{M} \models_{\lambda^i} \psi_1 \mathcal{U} \psi_2$. Otherwise $\mathcal{M} \models_{\lambda^i} \psi_1$ by the induction hypothesis and $\psi_1 \mathcal{U} \psi_2 \in K_{i+1}$ by the definition of R'. By definition of eventuality sequences there is some $\langle s_j, K_j \rangle$ with $j \geq i$ such that $\psi_2 \in K_j$. Let j be the minimal such time point. By the induction hypothesis $\mathcal{M} \models_{\lambda^j} \psi_2$. An induction proof now shows that $\mathcal{M} \models_{\lambda^k} \psi_1$ for all $k \in \{i, \ldots, j-1\}$, and hence $\mathcal{M} \models_{\lambda^i} \psi_1 \mathcal{U} \psi_2$.

Assume $\mathcal{M} \models_{\lambda^i} \psi_1 \mathcal{U} \psi_2$. Therefore $\mathcal{M} \models_{\lambda^j} \psi_2$ for some $j \geq i$ and $\mathcal{M} \models_{\lambda^k} \psi_1$ for all $k \in \{i, \ldots, j-1\}$. By the induction hypothesis $\psi_1 \in K_k$ for all $k \in \{i, \ldots, j-1\}$ and $\psi_2 \in K_j$.

Assume $\psi_1 \mathcal{U} \psi_2 \notin K_i$. Because $\psi_1 \in K_i$, now $\mathcal{X}(\psi_1 \mathcal{U} \psi_2) \notin K_i$, and hence $\mathcal{X}\neg(\psi_1 \mathcal{U} \psi_2) \in K_i$, and hence $\neg(\psi_1 \mathcal{U} \psi_2) \in K_{i+1}$. and $\psi_1 \mathcal{U} \psi_2 \notin K_{i+1}$. Going inductively further we eventually get $\psi_1 \mathcal{U} \psi_2 \notin K_j$, which contradicts $\psi_2 \in K_j$. Therefore it must be that $\psi_1 \mathcal{U} \psi_2 \in K_i$.

This finishes the inductive case 4 and the proof of the *if* direction of the equivalence. The *only if* direction follows. So assume $\mathcal{M} \models_s E\varphi$. Hence there is path $\lambda = s_0, s_1, s_2, \ldots$ starting from $s = s_0$ such that $\mathcal{M} \models_\lambda E\varphi$.

Define $K_i = \{\psi \in \mathrm{CL}(\varphi) | \mathcal{M} \models_{\lambda^i} \psi\}$ for all $i \geq 0$. It is easy to verify that $\langle s_0, K_0 \rangle, \langle s_1, K_1 \rangle, \ldots$ is an eventuality sequence starting from $\langle s_0, K_0 \rangle$. Further, by definition of $\mathrm{CL}(\varphi)$, $\varphi \in K_0$. This concludes the proof. □

Now the algorithmic problem that remains is to find eventuality sequences. Like in the CTL model-checking algorithm, also in this case the strong components are the means of identifying infinite paths having some property.

Definition 9.6 *Let $\mathcal{M}$ be a Kripke model, φ a formula, and $\langle S', R' \rangle$ the tableau for $\mathcal{M}$ and φ.*

Then a non-trivial (either $|C| > 1$ or there is $s \in C$ such that $sR's$) strong component $C \subseteq S'$ of $\langle S', R' \rangle$ is self-fulfilling *iff for every $\langle s, K \rangle \in S'$ and $\psi_1 \mathcal{U} \psi_2 \in K$ there is $\langle s', K' \rangle \in C$ such that $\psi_2 \in K'$.*

A self-fulfilling strong component in the tableau yields infinite execution paths along which *until* formulas are satisfied.

Lemma 9.7 *Let $\mathcal{M}$ be a Kripke model, φ a formula, and $\langle S', R' \rangle$ the tableau for $\mathcal{M}$ and φ.*

There is an eventuality sequence starting at $\langle s, K \rangle \in S'$ if and only if there is a path in $\langle S', R' \rangle$ from $\langle s, K \rangle \in S'$ to a self-fulfilling strong component.

Proof: Assume there is an eventuality sequence σ starting at $\langle s, K\rangle$. Let C' be the set of all $a \in S'$ that occur infinitely many times in σ. This set is a subset of a strongly connected component C of $\langle S', R'\rangle$. Hence there is a path from $\langle s, K\rangle$ to a strongly connected component. It remains to show that it is self-fulfilling.

Consider a subformula $\psi_1 \mathcal{U} \psi_2$ of φ such that $\psi_1 \mathcal{U} \psi_2 \in K'$ for some $\langle s', K'\rangle \in C$. Now there is a finite path from $\langle s', K'\rangle$ to C' because C is a strongly connected component and $C' \subseteq C$. If $\psi_2 \in K''$ for some $\langle s'', K''\rangle$ on that finite path, then $\psi_2 \in K'''$ for some $\langle s''', K'''\rangle \in C$.

If $\psi_2 \notin K''$ for any $\langle s'', K''\rangle$ on that finite path, then $\psi_1 \mathcal{U} \psi_2 \in K''$ for every $\langle s'', K''\rangle$ on that finite path, and in particular, its last element which is in C'. Because C' is part of an eventuality sequence, there is $\langle s'', K''\rangle \in C'$ such that $\psi_2 \in K''$. Hence in both cases there is ψ_2 in one of the nodes in C, and therefore C is a self-fulfilling strong component.

Assume there is a path from $\langle s, K\rangle$ to a self-fulfilling strongly connected component C.

For any $\psi_1 \mathcal{U} \psi_2$ occurring in a node in C we can construct a path from that node to a node with ψ_2, because C is self-fulfilling. Hence we can construct an infinite cycle of nodes in C with the same property. To make this infinite cycle an eventuality sequence, we also have to show that the path from $\langle s, K\rangle$ to C satisfies the required properties, that is, any $\psi_1 \mathcal{U} \psi_2$ on that path is followed by ψ_2. By the definition of R', for every node on that path following $\psi_1 \mathcal{U} \psi_2$ we either have ψ_2, or $\mathcal{X}(\psi_1 \mathcal{U} \psi_2)$ and hence $\psi_1 \mathcal{U} \psi_2$ in the following node. In the first case we have ψ_2 on the path, and in the second we have $\psi_1 \mathcal{U} \psi_2$ in C, and because C is self-fulfilling we have ψ_2 in C. Hence the infinite path with the cycle is an eventuality sequence. □

Theorem 9.8 *Let $\mathcal{M}$ be a Kripke model, φ a formula, and $\langle S', R'\rangle$ the tableau for $\mathcal{M}$ and φ.*

Now $\mathcal{M} \models_s E\varphi$ if and only if there is a path in $\langle S', R'\rangle$ from $\langle s, K\rangle \in S'$ (for some K) to a self-fulfilling strong component.

Proof: By the preceding two lemmata. □

From the above result we can construct an algorithm for LTL model-checking that runs in $\mathcal{O}((|S| + |R|)|2^{\mathcal{O}(|\varphi|)})$ time, which is polynomial in the size of the transition system but exponential in the size of the formula to be model-checked.

9.2.1 Fairness

Assume that we are verifying a specification of a computer system consisting of two processes P_1 and P_2 that perform their computational otherwise independently but communicate once in a while. Because there is only one CPU, there is a scheduler that decides when the computations proceed and for how long. When verifying the correctness of the system we may want not to model the functioning of the scheduler, and assume that the scheduler gives CPU to the both processes in a fair manner. This is the assumption of *fairness* that is used in connection with concurrent systems. A fair scheduler could for example produce computation $P_1, P_2, P_1, P_2, P_1, \ldots$, or possibly $P_1, P_2, P_2, P_1, P_2, P_2, P_1, P_2, \ldots$ if P_2 has a higher overhead, but definitely not $P_1, P_1, P_1, \ldots$ ignoring P_2 completely.

When model-checking such systems the assumption that the scheduler is fair should be taken into account. If it is not, and the correct behavior of the system involves both P_1 and P_2, then it is not difficult to find behaviors $E\varphi$ that violate the correctness properties φ. So, we often want to make the fairness assumption when doing model-checking.

Fairness assumptions can be represented as sets of formulas Φ with the requirement that for every $\phi \in \Phi$, any computation path contains infinitely many states in which ϕ is true.

In LTL handling fairness assumptions is easy. When model-checking $E(\mathcal{F}\varphi)$ (test whether there is a "bad" execution having undesirable property $\mathcal{F}\varphi$), just include the fairness assumptions as $E(\varphi \wedge \mathcal{GF}\phi_1 \wedge \cdots \wedge \mathcal{GF}\phi_n)$ where $\Phi = \{\phi_1, \ldots, \phi_n\}$.

CTL cannot express fairness assumptions in the same way this is possible with LTL. In particular, adjoining the formulas $\mathcal{GF}\phi_i$ as further conjuncts is not possible because of restrictions on path quantification. Instead,

1: **procedure** MC$_\rho(\phi)$
2: **if** $\phi = x$ **then return** BDD for x
3: **if** $\phi = \phi_1 \vee \phi_2$ **then return** MC$_\rho(\phi_1) \vee$ MC$_\rho(\phi_2)$;
4: **if** $\phi = \phi_1 \wedge \phi_2$ **then return** MC$_\rho(\phi_1) \wedge$ MC$_\rho(\phi_2)$;
5: **if** $\phi = \neg\phi_1$ **then return** $\neg$MC$_\rho(\phi_1)$;
6: **if** $\phi = E\mathcal{X}\phi_1$ **then return** modelcheckE$\mathcal{X}_\rho$(MC$_\rho(\phi_1)$);
7: **if** $\phi = E\mathcal{G}\phi_1$ **then return** modelcheckE$\mathcal{G}_\rho$(MC$_\rho(\phi_1)$);
8: **if** $\phi = E(\phi_1\mathcal{U}\phi_2)$ **then return** modelcheckE$\mathcal{U}_\rho$(MC$_\rho(\phi_1)$,MC$_\rho(\phi_2)$);

Figure 9.7: Symbolic Model-Checking Algorithm for CTL

1: **procedure** modelcheck$E\mathcal{X}_\rho(\phi)$
2: **return** $\exists X'.(\rho \wedge \phi[X'/X])$;

Figure 9.8: Symbolic Model-Checking for $E\mathcal{X}\phi$

researchers have defined variants of CTL (Fair CTL) with extra machinery for expressing fairness.

9.3 Symbolic Model-Checking

The scalability of standard model-checking algorithms for modal and temporal logics is limited by the size of the models. The models are often represented succinctly by using logic-based representations, and model-checking algorithms can be adapted to work with these succinct representations.

In this section we introduce the BDD-based model-checking algorithm for CTL, as well as bounded LTL model-checking that reduces the model-checking problem to SAT, the satisfiability problem of the propositional logic.

9.3.1 Symbolic CTL Model-Checking

The BDD-based model-checking algorithm is given in Figure 9.7.

The connectives $\vee$, $\wedge$ and $\neg$ are by the standard BDD operations, implemented with the *apply* procedure.

BDD-based procedures for model-checking $E\mathcal{X}$, $E\mathcal{G}$, and $E\mathcal{U}$ are given in Figures 9.8, 9.9 and 9.10, respectively.

Model-checking $E\mathcal{X}\phi$ in Figure 9.8 is simply by identifying states for which at least one successor state satisfies ϕ. This is similar to the forward step we have in BDD-based reachability in Section 7.1, but we go backwards from states that satisfy ϕ.

The procedure for $E\mathcal{G}\phi$ in Figure 9.9 finds states that are starting states for increasingly long sequences of states that satisfy ϕ: S_0 consists of state that satisfy ϕ, S_1 consists of states that satisfy ϕ and that have a successor state that satisfies ϕ, and so on. In general S_i consists of sequences of length $i+1$ satisfying ϕ. When $S_i = S_{i-1}$ for some i, then it would also be $S_{i+1} = S_i$, and more generally $S_j = S_{j-1}$ for every $j \geq i$. This means that the state sequences satisfying ϕ starting in states in S_i have an infinite length (and it must be cycles in the transition graph.) These sequences clearly satisfy $E\mathcal{G}\phi$.

Last, the procedure for model-checking $E(\phi\mathcal{U}\psi)$ is given in Figure 9.10. We find sequences of states that satisfy

1: **procedure** modelcheck$E\mathcal{G}_\rho(\phi)$
2: $S_0 := \phi$;
3: $i := 0$;
4: **repeat**
5: $\quad S_{i+1} := \phi \wedge \exists X'.(\rho \wedge S_i[X'/X])$;
6: $\quad i := i + 1$;
7: **until** $S_i \equiv S_{i-1}$;
8: **return** S_i;

Figure 9.9: Symbolic Model-Checking for $E\mathcal{G}\phi$

1: **procedure** modelcheck$E\mathcal{U}_\rho(\phi,\psi)$
2: $S_0 := \psi$;
3: $i := 0$;
4: **repeat**
5: $S_{i+1} = \psi \vee (\phi \wedge \exists X'.(\rho \wedge S_i[X'/X]))$;
6: $i := i+1$;
7: **until** $S_i \equiv S_{i-1}$;
8: **return** S_i;

Figure 9.10: Symbolic Model-Checking for $E(\phi\mathcal{U}\psi)$

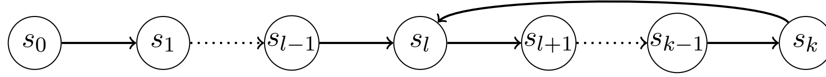


Figure 9.11: Infinite Executions in Bounded LTL Model-Checking

ϕ except for the last state which satisfies ψ. Similarly to $E\mathcal{G}$, we first find sequences of length 1, then 2, 3, and so on, until we have covered arbitrarily long sequences.

Unlike in LTL, witnesses or certificates for the satisfaction (or falsification) of a CTL formula are not state sequences but have a tree structure, and are trickier to construct. We do not discuss this topic here.

9.3.2 Symbolic LTL Model-Checking

The SAT-based LTL model-checking algorithm [BCCZ99], known as *Bounded Model-Checking* (BMC), can be viewed as a generalization of the SAT-based reachability method first introduced in connection with AI planning [KS92].

LTL requires the representation of infinite loops, and to represent them in the propositional logic, they have to be represented finitely. This is possible for paths of the form $s_0, s_1, \ldots, \overbrace{s_l, \ldots, s_k}, \overbrace{s_l, \ldots, s_k}, s_l, \ldots$, where the segment $s_l, \ldots, s_k$ repeats infinitely many times. See Figure 9.11.

$$
\begin{array}{rclrcl}
[\![x]\!]_i^{l,k} & = & x@i & [\![\mathcal{X}\phi]\!]_i^{l,k} & = & [\![\phi]\!]_{succ(i)}^{l,k} \\
[\![\neg x]\!]_i^{l,k} & = & \neg x@i & [\![\mathcal{G}\phi]\!]_i^{l,k} & = & \bigwedge_{j=\min(l,i)}^{k} [\![\phi]\!]_j^{l,k} \\
[\![\phi_1 \vee \phi_2]\!]_i^{l,k} & = & [\![\phi_1]\!]_i^{l,k} \vee [\![\phi_2]\!]_i^{l,k} & [\![\mathcal{F}\phi]\!]_i^{l,k} & = & \bigvee_{j=\min(l,i)}^{k} [\![\phi]\!]_j^{l,k} \\
[\![\phi_1 \wedge \phi_2]\!]_i^{l,k} & = & [\![\phi_1]\!]_i^{l,k} \wedge [\![\phi_2]\!]_i^{l,k} & [\![\phi\mathcal{U}\psi]\!]_i^{l,k} & = & \bigvee \begin{array}{l} \bigvee_{j=i}^{k}([\![\psi]\!]_j^{l,k} \wedge \bigwedge_{z=i}^{j-1}[\![\phi]\!]_z^{l,k}) \\ \bigvee_{j=l}^{i-1}([\![\psi]\!]_j^{l,k} \wedge \bigwedge_{z=l}^{j-1}[\![\phi]\!]_z^{l,k} \wedge \bigwedge_{z=i}^{k}[\![\phi]\!]_z^{l,k}) \end{array}
\end{array}
$$

Above $succ(i) = i+1$ if $i < k$ and otherwise $succ(k) = l$.

ϕ is true on some path (representable as a (k,l)-loop) iff $[\![\phi]\!]_0^{l,k} \wedge I \wedge T[X@1/X'] \wedge \cdots \wedge T[X@(k-1)/X, X@k/X'] \wedge T[X@k/X, X@l/X']$ is satisfiable.

The formula we have given has two parameters, l and k. There is no a priori way of determining their value, and one needs to try different values for them. First, k is increased gradually, e.g. $k = 1, k = 2, k = 3, \ldots$, and for every value of k, different values $l = 0, l = 1, l = 2, \ldots$ with $l < k$ are tried.

Another possibility is to device a slightly more complicated formula that is parameterized with k only, and alternative values of $l < k$ are tried out by the SAT solver. This makes the search for right value of k one dimensional, similarly to the SAT-based reachability method in Section 7.

Example 9.9 Consider the 2,4-loop in Figure 9.12. The reduction of LTL model-checking would give the follow-

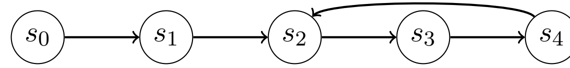


Figure 9.12: Example: 2,4-loop

ing two formulas for this loop.

$$
\begin{aligned}
[\![\mathcal{X}\mathcal{G}a]\!]_0^{2,4} &= a@1 \wedge a@2 \wedge a@3 \wedge a@4 \\
[\![a\mathcal{U}b]\!]_1^{2,4} &= b@1 \\
&\vee(b@2 \wedge a@1) \\
&\vee(b@3 \wedge a@1 \wedge a@2) \\
&\vee(b@4 \wedge a@1 \wedge a@2 \wedge a@3)
\end{aligned}
$$

■

The main strength of the bounded model-checking method for LTL is its scalability to very large systems, as long as the horizon length parameter k remains small enough. This makes the method especially useful as a *bug detector*.

If there is no path satisfying the LTL formula, all the formulas for different values of k are unsatisfiable. Since there is no obvious upper bound on the values of k, it is not clear when to stop. Therefore the bounded model-checking method cannot in general be used for *proving* that certain behaviors are not possible. This is in contrast to the enumerative and BDD-based model-checking methods that are *complete*.

For restricted correctness properties like *safety*, temporal induction [SSS00] is a highly scalable SAT-based method that can provide completeness.

Chapter 10

Predicate Logic

Propositional logics are perfect for applications that can be represented in terms of a fixed number of atomic facts. Many problems, even if superficially requiring more complex datatypes than just Booleans, for example multi-valued variables and bounded integer variables, are often reducible to propositional logic, and can be solved with propositional methods.

However, there are plenty of applications that are more complex in some way, requiring potentially infinitary and unbounded models. These are often not practically reducible to the propositional logic.

The predicate logic is capable of representing relational data, about a finite or infinite universe (which might not be explicitly represented) and relations over the objects in the universe, as well as reasoning about complex statements about properties of relational data. This very much differs from what is (practically) possible in propositional logics.

Predicate logic has been used in research on the foundations of mathematics, for example for expressing basic concepts of arithmetic and set-theory.

Theories of the semantics of natural languages have extensively used predicate logic and its various variants and extensions.

There are lots of applications of the predicate logic in computer science. As an example, much of database theory is expressible in logical terms, and there is a close connection between relational database languages such as SQL which are based on relational algebra, and the predicate logic. Relational database queries can be expressed in the predicate logic equally well (and sometimes far more intuitively) as in SQL.

Outside database theory, the basic concepts of the predicate logic, like quantification, are useful also in many types of high-level specification languages. The logic programming language Prolog is based on the predicate logic and the automated reasoning methods developed for it, especially the resolution rule for automated reasoning.

In addition to usual forms of logical reasoning, the need to support relational data has been recognized in other areas of computer science and artificial intelligence. Relational languages built on the predicate logic have been broadly adopted in areas such as relational machine learning [MR06, NMTG15, DRKNP16] and probabilistic reasoning [RD06].

10.1 Introduction

Predicate logic was originally introduced to analyze the concepts of quantification "for all" and "for some", as they appear in natural languages like English, and to express concepts in mathematics that require the same quantification.

Similarly to the close connection between the Boolean connectives $\wedge$, $\vee$ and $\neg$ and the natural language words "and", "or", and "not", the features of the predicate logic also have a natural language counterpart.

Example 10.1 Many simple sentences in natural languages like English can be straightforwardly encoded in the predicate logic. Intransitive verbs and adjectives can be represented as unary (1-place) predicates, transitive verbs can be represented as binary (2-place) predicates, proper names can be represented by constant symbols, and

John is sad.	sad(John)
John is sleeping.	sleeps(John)
Somebody sleeps.	$\exists x$.sleeps(x)
John admires Mary	admires(John,Mary)
John admires somebody.	$\exists x$.admires(John,x)
John admires a computer scientist.	$\exists x$.(admires(John, x)$\wedge$computerscientist(x))
John is admired by somebody.	$\exists x$.admires(x,John)
Somebody admires himself/herself.	$\exists x$.admires(x,x)
Somebody does not admire anybody.	$\exists x.\forall y.\neg$admires(x,y)
Everybody admires someone.	$\forall x.\exists y$.admires(x, y)
There is somebody everybody admires.	$\exists y.\forall x$.admires(x, y)
Everybody is admired by someone.	$\forall x.\exists y$.admires(y, x)
A bachelor is an unmarried male.	$\forall x$.(bachelor(x) $\leftrightarrow$ (male(x) $\wedge$ $\neg\exists y$.married(x, y)))

Table 10.1: Sentences in English translated into the Predicate Logic

words like "all" and "some" correspond to quantification.

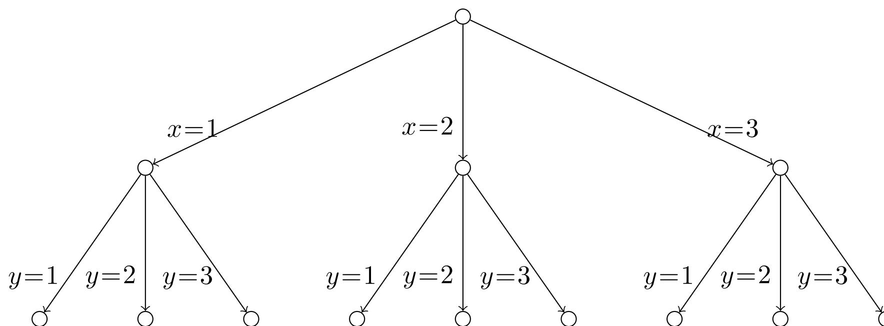
A collection of sentences and their counterparts in the predicate logic are given in Table 10.1. ■

In the above example, notice the difference between "Everybody admires someone" and "There is somebody everybody admires", for which the formula is the same except the reversed ordering of the quantifiers. Two universal quantifiers $\forall x.\forall y$ or two existential quantifiers $\exists x.\exists y$ can be reversed without changing the meaning of the formula, but the ordering of consecutive different quantifiers cannot.

The above formulas could be reduced to the propositional logic if the domain of the quantified variables x and y was a fixed finite set, for example $\{John, Mary, Jack\}$. Elimination of $\forall x$ in $\forall x$.sleeps(x) in favor of the conjunction connective leads to sleeps(John)$\wedge$sleeps(Mary)$\wedge$sleeps(Jack), and similarly $\exists x$.sleeps(x) corresponds to sleeps(John)$\vee$sleeps(Mary)$\vee$sleeps(Jack). But in general, the domain or universe of a predicate logic theory would not be fixed, or it would not be finite, and hence such a syntactic substitution of quantifications by chains of conjunctions or disjunctions would not be possible.

Quantification can be viewed in terms of *and-or* trees.

Example 10.2 Consider quantification $\forall x.\exists y.\Phi$ where x and y vary over $\{1, 2, 3\}$.



This is an *and-or* tree, where the root node is an *and*-node, and the internal nodes following labels $x = i$ are *or*-nodes. For the formula $\forall x.\exists y.\Phi$ to be true, for every choice of $x \in \{1, 2, 3\}$ there must be one (possibly different) choice of $y \in \{1, 2, 3\}$ so that Φ evaluates to *true*.

We could informally also view the quantification in terms of conjunctions and disjunctions as

$$(\Phi(1, 1) \vee \Phi(1, 2) \vee \Phi(1, 3)) \wedge (\Phi(2, 1) \vee \Phi(2, 2) \vee \Phi(2, 3)) \wedge (\Phi(3, 1) \vee \Phi(3, 2) \vee \Phi(3, 3))$$

Where $\Phi(x, y)$ is parameterized with the possible values of x and y. The first or outer choice is conjunctive (*and*) choice between values of x, and the inner choice is disjunctive (*or*) choice between values of y.

Clearly, reordering the quantifiers from $\forall x \exists y \Phi$ to $\exists y \forall x \Phi$ would change the meaning of the formula, as the connectives $\wedge$ and $\vee$ in the formula, and the *and* and *or* in the tree, would change places.

For the formula $\exists y.\forall x..\Phi$ to be true, there should be one choice of $y \in \{1,2,3\}$ so that with every $x \in \{1,2,3\}$ the formula Φ evaluates to true. This is clearly different from the original formula $\forall x.\exists y.\Phi$. ■

One of the early applications of logics and the predicate logic was investigations of the foundations of mathematics.[1]

Example 10.3 (Peano Axioms) The following formulas belong to Peano's formalization of natural numbers with addition and multiplication.

1. $\forall x(Zero \neq Succ(x))$
2. $\forall x \forall y(Succ(x) = Succ(y) \rightarrow x = y)$
3. $\forall x(Plus(x, Zero) = x)$
4. $\forall x \forall y(Plus(x, Succ(y)) = Succ(Plus(x, y)))$
5. $\forall x(Mult(x, Zero) = Zero)$
6. $\forall x \forall y(Mult(x, Succ(y)) = Plus(Mult(x, y), x))$

Additionally, Peano introduced an *induction axiom*, but it is not expressible in the predicate logic because it requires quantification over all formulas ψ.

$$\begin{aligned}\forall\psi.\forall y_1.\forall y_2.\cdots\forall y_k.(&\psi(Zero, y_1, y_2, \ldots, y_k)\\ &\wedge \forall x(\psi(x, y_1, y_2, \ldots, y_k) \rightarrow \psi(Succ(x), y_1, y_2, \ldots, y_k))\\ &\rightarrow \forall x(\psi(x, y_1, y_2, \ldots, y_k)))\end{aligned}$$

■

10.2 Syntax

The differences between the propositional logic and the predicate logic is the much more complex form of the atomic formulas, as structured entities in the predicate logic, and quantification.

The predicate logic talks about the objects of a *universe*, the members of which, however, are not enumerate explicitly. The objects in the universe can be given names by *constant* symbols, or they can be referred to by *terms* that are formed from simpler terms and *function symbols*, which express functions from n-tuples of objects to objects.

Definition 10.4 (Terms) *1. Constant symbols and variables are* terms.

2. If $t_1, \ldots, t_n$ are terms and f is an n-ary function symbol, then $f(t_1, \ldots, t_n)$ is a term.

Definition 10.5 (Atomic Formulas) *Atomic formulas are constructed from terms and predicate symbols.*

1. *If P is an n-ary predicate symbol and $t_1, \ldots, t_n$ are terms, then $P(t_1, \ldots, t_n)$ is an* atomic formula.
2. *If t_1 and t_2 are terms, then $t_1 = t_2$ is an* atomic formula.

Definition 10.6 (Formulas) *Formulas are constructed from atomic formulas by using Boolean connectives and quantifiers.*

1. *Atomic formulas are* formulas.
2. *If ϕ ja ψ are formulas and x is a variable, then $(\neg\phi)$, $(\phi \vee \psi)$, $(\phi \wedge \psi)$, $(\phi \rightarrow \psi)$, $(\phi \leftrightarrow \psi)$, $(\forall x.\phi)$ and $(\exists x.\phi)$ are* formulas.

To understand how quantification, it is important to understand what the variable x in a quantifier $\forall x$ or $\exists x$ refers to.

Definition 10.7 (Scope of quantifiers, free and bound variables) *The* scope *of the quantification $\forall x$ in formula $\forall x.\phi$ (or in any formula in which this formula is a subformula) is ϕ. The variable x is* bound *in its scope. The occurrences of x in ϕ bound by $\forall x$ are* bound occurrences *of x. A variable occurrence that is not bound is* free.

[1] These investigations led to groundbreaking results such as Gödel's incompleteness theory for the theory of arithmetics.

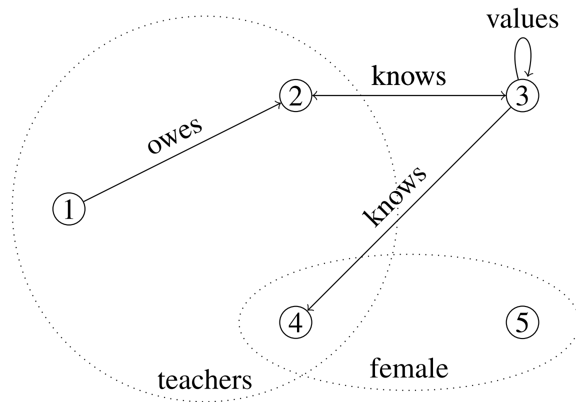


Figure 10.1: Diagrammatic illustration of relational data

Example 10.8 Consider variables in the following formula.

$$\forall \overbrace{x}^{\text{bound}} \left(P(\overbrace{x}^{\text{bound}}, \overbrace{y}^{\text{free}}, \overbrace{z}^{\text{free}}) \vee \exists \overbrace{y}^{\text{bound}} (Q(\overbrace{y}^{\text{bound}}) \rightarrow R(\overbrace{x}^{\text{bound}}, \overbrace{z}^{\text{free}})) \right).$$

All occurrences of the variable z are free, as there is no quantifier $\forall z$ or $\exists z$. All occurrences of x are bound, as the outermost quantifier $\forall x$ binds them all. The first occurrence of y is free, but the rest are bound by the quantifier $\exists y$. ■

Quantification with the *universal quantifier* $\forall$ (read as "for all") and the *existential quantifier* $\exists$ (read as "for some") is a core feature of the predicate logic. A formula $\forall x.\Phi$ denotes "for all objects x the formula Φ holds", and $\exists x.\Phi$ denotes "for some object x the formula Φ holds".

Quantifiers $\forall$ and $\exists$ are dual to each other similarly to the duality between $\wedge$ and $\vee$. Hence the universal quantification $\forall x.\Phi$ is logically equivalent to $\neg\exists x.\neg\Phi$, and $\exists x.\Phi$ is logically equivalent to $\neg\forall x.\neg\Phi$, analogously to the equivalences $\alpha \wedge \beta \equiv \neg(\neg\alpha \vee \neg\beta)$ and $\alpha \vee \beta \equiv \neg(\neg\alpha \wedge \neg\beta)$,

The analogy of $\forall$ and $\wedge$ is more general. The universal quantification $\forall x.\Phi$ could be understood as a (potentially infinitely long) conjunction $\Phi(v_1) \wedge \cdots \wedge \Phi(v_n) \wedge \cdots$ for all possible values v_i for the variable x. Similarly, the existential quantification $existsx$ can be viewed as equivalent to a long disjunction.

10.3 Semantics

First first discuss the semantics of the predicate logic informally in terms of examples, and after that proceed with a formal definition.

Example 10.9 Consider the diagrammatic representation of a five persons, identified by the numbers 1, 2, 3, 4 and 5. We additionally have the following relations on the set of these persons, and the constant symbols (names) a, b, c, d, and e to name the individuals in our universe $\{1, 2, 3, 4, 5\}$.[2]

relation	interpretation	constant	interpretation
knows	$\{(2,3),(3,2),(3,4)\}$	a	1
owes	$\{(1,2)\}$	b	2
values	$\{(3,3)\}$	c	3
female	$\{4,5\}$	d	4
teachers	$\{1,2,4\}$	e	5

This diagram shows for example that person 1 owes something to person 2, that person 3 values himself/herself, and that persons 4 and 5 are female and the rest are not, and person 4 is the only person of these five that is a female teacher.

[2]Notice that in general there is no necessity to have a name for all elements in the universe, and there would not need to be any constant symbols at all.

Now we can view the following formulas in the predicate logic in regard to the relations *knows*, *values* and *owes*, and the sets (unary relations) *teacher* and *female*.

1. $\exists x \exists y(knows(x, y) \wedge knows(y, x))$
2. $\exists x \exists y(teacher(x) \wedge teacher(y) \wedge owes(x, y))$
3. $\exists x(values(x, x) \wedge \exists x(teacher(x) \wedge female(x)))$
4. $\neg \forall x(teacher(x) \rightarrow female(x))$
5. $\forall x(knows(x, c) \rightarrow owes(a, x))$

The first sentence is true (with respect to these relations), because we can find persons x and y so that knows(x, y) and knows(y, x). There two possibilities: either $x = 2, y = 3$, or $x = 3, y = 2$. There are no other possibilities.

The fourth sentence says that "Not all teachers are female". This is also true, because we can choose $x = 1$ or $x = 2$, and the implication teacher(x)$\rightarrow$female(x) is false, because both 1 and 2 are teachers and not female.

The truth of the remaining formulas can be verified similarly. ■

Next we go through the formal semantics of the predicate logic. In the propositional logic it was sufficient to give a mapping from atomic formulas to truth values. In the predicate logic we have to first associate a meaning with all terms, and in addition to the Boolean connectives $\wedge$, $\vee$ and $\neg$, we also have the quantifiers $\forall$ and $\exists$ to give a meaning to.

Definition 10.10 (Interpretation of symbols under a structure) A structure $\mathcal{S}$ *for given constant, function and predicate symbols) consists of a non-empty* universe U*, which can be finite or infinite, and* interpretations $[\![...]\!]^{\mathcal{S}}$ *of those symbols.*

1. *Constant symbols c are mapped to $[\![c]\!]^{\mathcal{S}} \in U$.*
2. *Variable symbols v are mapped to $[\![v]\!]^{\mathcal{S}} \in U$.*
3. *Function symbols f of arity n map to functions $[\![f]\!]^{\mathcal{S}} : U^n \rightarrow U$.*
4. *Predicate symbols P of arity n map to relations $[\![P]\!]^{\mathcal{S}} \subseteq U^n$.*

Note that we typically leave the interpretations of variables symbols open when giving a structure. For example, when evaluating $\forall x.P(x)$ under some structure, that structure would not say anything about the value of x. It is only the truth-definition of quantified formulas in Definition 10.13 that would bind different elements of the universe when testing the truth of $\forall x.P(x)$.

Interpretation of all symbols can be extended to an the interpretation $[\![t]\!]^{\mathcal{S}}$ of all terms as follows.

Definition 10.11 (Interpretation of terms under a structure) *Given the interpretation of all constant symbols, function symbols, and predicate symbols, we define the interpretations of all terms as follows.*

- *Constant symbols c are directly evaluated by $[\![c]\!]^{\mathcal{S}}$.*
- *Variables v are directly evaluated by $[\![v]\!]^{\mathcal{S}}$.*
- *For terms $f(t_1, \dots, t_n)$, we have $[\![f(t_1, \dots, t_n)]\!]^{\mathcal{S}} = [\![f]\!]^{\mathcal{S}}([\![t_1]\!]^{\mathcal{S}}, \dots, [\![t_n]\!]^{\mathcal{S}})$.*

Definition 10.12 (Truth of atomic formulas under a structure) *For atomic formulas $P(t_1, \dots, t_n)$ we define* truth in $\mathcal{S}$*:*

$$\mathcal{S} \models P(t_1, \dots, t_n) \text{ iff } ([\![t_1]\!]^{\mathcal{S}}, \dots, [\![t_n]\!]^{\mathcal{S}}) \in [\![P]\!]^{\mathcal{S}}$$
$$\mathcal{S} \models (t_1 = t_2) \text{ iff } [\![t_1]\!]^{\mathcal{S}} = [\![t_2]\!]^{\mathcal{S}}$$

Definition 10.13 (Truth of formulas under a structure) *Let $\mathcal{S}$ be a structure and U its universe. For all formulas α and β and variables v we define the truth of formulas in the structure as follows.*

1. *For atomic formulas α, $\mathcal{S} \models \alpha$ as defined earlier.*
2. *Define $\mathcal{S} \models \alpha \wedge \beta$, $\mathcal{S} \models \alpha \vee \beta$ and $\mathcal{S} \models \neg\alpha$ as in propositional logic*
3. *$\mathcal{S} \models \exists v.\alpha$ iff $\mathcal{S}[v \mapsto a] \models \alpha$* for some $a \in U$
4. *$\mathcal{S} \models \forall v.\alpha$ iff $\mathcal{S}[v \mapsto a] \models \alpha$* for all $a \in U$

Here $\mathcal{S}[v \mapsto a]$ is a structure exactly like $\mathcal{S}$ except that it maps v to a.

$\forall x.\phi$	$\equiv$	$\neg\exists x.\neg\phi$	
$\exists x.\phi$	$\equiv$	$\neg\forall x.\neg\phi$	
$\forall x.\forall y.\phi$	$\equiv$	$\forall y.\forall x.\phi$	
$\exists x.\exists y.\phi$	$\equiv$	$\exists y.\exists x.\phi$	
$\forall x.\forall x.\phi$	$\equiv$	$\forall x.\phi$	
$\exists x.\exists x.\phi$	$\equiv$	$\exists x.\phi$	
$\forall x.(\phi_1 \wedge \phi_2)$	$\equiv$	$(\forall x.\phi_1) \wedge (\forall x.\phi_2)$	
$\exists x.(\phi_1 \vee \phi_2)$	$\equiv$	$(\exists x.\phi_1) \vee (\exists x.\phi_2)$	
$\forall x.\phi$	$\equiv$	ϕ	if x does not occur free in ϕ
$\exists x.\phi$	$\equiv$	ϕ	if x does not occur free in ϕ
$\forall x.(\phi_1 \vee \phi_2)$	$\equiv$	$(\forall x.\phi_1) \vee \phi_2$	if x does not occur free in ϕ_2
$\exists x.(\phi_1 \wedge \phi_2)$	$\equiv$	$(\exists x.\phi_1) \wedge \phi_2$	if x does not occur free in ϕ_2

Table 10.2: Predicate Logic Equivalences

Notice how the nesting for quantifications of the *same variable* are handled in the definition.

Example 10.14 Given a structure $\mathcal{S}$ with universe $U = \{1, 2\}$ and interprets P as $\{1\}$. The formula $\forall x.\exists x.P(x)$ is syntactically well-formed. The truth-definition of $\forall x$ iterates over all values in U for x, and then recursively tests the truth of $\exists x.P(x)$ in each case, first with the binding $x = 1$ and then with $x = 2$. But when evaluating $\exists x.P(x)$, there is another iteration over U, where it is sufficient that $P(x)$ is true for one value of x. Here the values of x bound by $\forall x$ are overwritten and therefore irrelevant. The truth value of $\forall x.\exists x.P(x)$ under this and any structure is therefore the same as that of $\exists x.P(x)$. Therefore these two formulas are logically equivalent. ■

10.4 Decision Problems

As in the propositional logic, the main decision problems in the predicate logic are satisfiability, validity, and logical consequence.

- ϕ is *satisfiable* iff $\mathcal{S} \models \phi$ for some $\mathcal{S}$
- ϕ is *valid* iff for $\mathcal{S} \models \phi$ for all $\mathcal{S}$
- $\psi \models \phi$ iff $\mathcal{S} \models \phi$ for all $\mathcal{S}$ s.t. $\mathcal{S} \models \psi$

These definitions (implicitly) involve going through all (infinitely many) possible structures (with the given constant, function and predicate symbols): this is the source of the very high complexity of these tasks.

There is no algorithm that is guaranteed to end with the answer "yes" or "no" to the question of satisfiability of a given formula in the predicate logic. Therefore the satisfiability problem for the predicate logic is not *decidable*. There is, however, a semi-decision procedure: one can systematically go through all different kinds of structures, and if the formula is satisfiable, this procedure will eventually find a structure that satisfies the formula. This is not a decision procedure, however, because there is no criterion for stopping the procedure and giving the answer "not satisfiable".

Practical algorithms for testing satisfiability and logical consequence don't exhaustively go through all structures, but do something more clever. One approach to automated reasoning in the predicate logic is based on a resolution rule for the predicate logic. This rule is more complicated than the resolution rule for the propositional logic, as the notion of CNF and clauses for formulas with quantifiers and structured atomic formulas is more complicated that in the propositional logic. We do not discuss this rule here.

10.5 Equivalences

The propositional equivances listed in Table 2.1 hold for the predicate logic as is. Additionally, there are valid equivalences that relate the quantifiers to each other and to the Boolean connectives, as shown in Table 10.2.

rule	example
S : NP VP	$\overbrace{\text{A man}}^{\text{NP}}\ \overbrace{\text{owns a dog that sleeps}}^{\text{VP}}$
NP : DET CN	$\overbrace{\text{a}}^{\text{DET}}\ \overbrace{\text{man}}^{\text{CN}}$
VP : TV NP	$\overbrace{\text{owns}}^{\text{TV}}\ \overbrace{\text{a dog that sleeps}}^{\text{NP}}$
NP : DET RCN	$\overbrace{\text{a}}^{\text{DET}}\ \overbrace{\text{dog that sleeps}}^{\text{RCN}}$
RCN : CN *that* VP	$\overbrace{\text{dog}}^{\text{CN}}\ \textit{that}\ \overbrace{\text{sleeps}}^{\text{VP}}$
VP : IV	$\overbrace{\text{sleeps}}^{\text{IV}}$

S	sentence
NP	noun phrase
VP	verb phrase
CN	common noun
DET	determiner
TV	transitive verb
IV	intransitive verb

Table 10.3: A grammar for a small fragment of English

10.6 Application: Natural Language

There are several semantic theories for natural languages. Here we present one theory that was developed by Richard Montague, now known as *Montague semantics* or *Montague grammar*. It is based on a combination of the λ-calculus (see Section A) and the predicate logic, and its important feature is that it is *compositional*, which means that the meaning of a phrase is obtained as a function of the meanings of its components.

In Table 10.3 a very simple context-free grammar for a narrow fragment of English is given. The *non-terminal* symbols represent different grammatical categories such as verb phrases and noun phrases.

Our goal in this section is to present a simple Montague-style grammar [Mon70, Mon73] that maps simple English sentences to the predicate logic, as in the next table.

English	predicate logic
John sees Mary.	sees(John,Mary)
A man sleeps.	$\exists x.$(man$(x)\wedge$sleeps(x))
A man owns a dog.	$\exists x.$(man$(x) \wedge \exists y.$(dog$(y)\wedge$owns(x, y)))
A man owns a dog that sleeps.	$\exists x.$(man$(x) \wedge \exists y.$(dog$(y)\wedge$owns$(x, y)\wedge$sleeps(y)))

To do this, we need to associate a *type* with each grammar category, as in the typed λ-calculus (see A). And then we need to associate with each grammar rule an expression that forms the meaning of that expression from the meanings of its sub-expressions.

One possibility is the following [VEU10].

- Verb phrase (VP): term $\to$ formula
- Noun phrase (NP): (term $\to$ formula) $\to$ formula
- Common noun (CN): term $\to$ formula
- Transitive verb (TV): term $\to$ (term $\to$ formula)

Here a verb phrase is associated with a function from terms to formulas. For example, if we have the term *John*, and the verb phrase is "loves Mary", expressed by the function $\lambda x.$loves(x,Mary), we would obtain the formula $loves(John, Mary)$ as a result.

One of the less intuitive things about the above choice of types is that of noun phrases. In the preceding example, it would seem sufficient to represent noun phrases like "John" as a term, from which the function associated with a verb phrase like "loves Mary" would then immediately produce the formula for the whole sentence.

However, this is not more generally possible: consider the phrase "A young child loves Mary", in which we cannot represent the phrase "a young child" as a single term. The formula we are after is $\exists x.$(child$(x)\wedge$young$(x)\wedge$loves(x,Mary)), and the meaning of "a young child" has to also include the atomic formula young(x) somehow, and this cannot be expressed as a term.

S	:	NP VP	$(NP\ VP)$
NP	:	name	$\lambda P.(P\ name)$
	:	DET CN	$(DET\ CN)$
	:	DET RCN	$(DET\ RCN)$
DET	:	”some”	$\lambda P.\lambda Q.\exists x((P\ x) \wedge (Q\ x))$
	:	”a”	$\lambda P.\lambda Q.\exists x((P\ x) \wedge (Q\ x))$
	:	”every”	$\lambda P.\lambda Q.\forall x((P\ x) \rightarrow (Q\ x))$
	:	”no”	$\lambda P.\lambda Q.\forall x((P\ x) \rightarrow (\neg(Q\ x)))$
VP	:	IV	$\lambda x.IV(x)$
	:	TV NP	$\lambda x.(NP\ (\lambda y.(TV\ y\ x)))$
TV	:	transverb	$\lambda x.\lambda y.transverb(x,y)$
RCN	:	CN ”that” VP	$\lambda x.((CN\ x) \wedge (VP\ x))$
	:	CN ”that” NP TV	$\lambda x.((CN\ x) \wedge (NP\ (\lambda y.(TV\ y\ x))))$
CN	:	predicate	$\lambda x.predicate(x)$

Table 10.4: A grammar with semantics rules for a fragment of English

expression	meaning
a	$\lambda P.\lambda Q.\exists x((P\ x) \wedge (Q\ x))$
man	$\lambda x.man(x)$
a man	$\lambda Q.\exists x(man(x) \wedge (Q\ x))$
sleeps	$\lambda x.sleeps(x)$
a man sleeps	$\exists x(man(x) \wedge sleeps(x))$
man that sweats	$\lambda x.(man(x) \wedge sweats(x))$
a man that sweats	$\lambda Q.\exists x(man(x) \wedge sweats(x) \wedge (Q\ x))$
A man that sweats sleeps.	$\exists x(man(x) \wedge sweats(x) \wedge sleeps(x))$
Sirpa sees Jarmo.	$sees(sirpa, jarmo)$
Every woman sees a man.	$\forall x(woman(x) \rightarrow (\exists y(man(y) \wedge sees(x,y))))$
Every woman sees a man that sleeps.	$\forall x(woman(x) \rightarrow (\exists y(man(y) \wedge sleeps(y) \wedge sees(x,y))))$
A woman that eats sees a man that sleeps.	$\exists x(woman(x) \wedge eats(x) \wedge \exists y(man(y) \wedge sleeps(y) \wedge sees(x,y)))$

Table 10.5: English sentences and their translations in the predicate logic

Instead, the meaning of a noun phrase will be a function that takes the meaning of the verb phrase (of type term $\rightarrow$ formula), and maps it to a formula. This way the possibly quite complex meaning of the noun phrase can be expressed. The meaning of a verb phrase is still of type $term \rightarrow formula$, so the only part of the noun phrase this function uses is a term (either a constant symbol or a variable.)

A more complex grammar is given in Table 10.4.

Note that all variables x introduced in the DET rules must be *new* in the sense that they occur in their scope only as explicitly stated in the grammar rule, and any possible quantified sentence that represent the meaning of a sub-expression uses other variables.

Some examples of this translation are given in Table 10.5.

Natural language processing with these and related methods is discussed in detail in the literature [VEU10, FL89, Fro06]. The original papers are by Richard Montague from the early 1970s [Mon70, Mon73].

10.7 Application: Databases

Relational algebra [Cod72] is one of the fundaments of relational database systems and query languages such as SQL.[3] In this section we present predicate logic as a query language alternative to SQL and relational algebra. While the relational algebra as a more procedural formalism leads more directly to effective implementations, the predicate logic is arguably in many cases a far more intuitive and readable language for expressing complex queries.

The relational calculus is a term applied to query languages directly based on the predicate logic, and there are several variants presented in the literature [Cod72, Cod83, LP77]. Their commonalities include predicates as in the predicate logic for referring to the relations in a database, and logical operations familiar from the predicate logic, including the Boolean connectives $\wedge$, $\vee$, $\neg$ and the quantifiers $\forall$ and $\exists$. In this section we do not follow any of these faithfully, and instead try to use standard predicate logic syntax and concepts unchanged, to make the connection to the predicate logic clear.

The idea in queries in these languages is to express the desired information as a logical formula, in which the *free* variables indicate the information to be retrieved from the database. Queries without free variables test whether the formula is *true* when interpreting the contents of the database as a *structure* (Section 10.3).

Example 10.15 Consider a relational database consisting of three relations called *parent*, *male* and *female* as follows.

parent

parent	child
Mary	Jack
John	Jack
Jack	Jill
Jack	Bob

female

person
Jill
Mary

male

person
Bob
Jack
John

The query *parent(Mary,x)* with the free variable x, is asking about all pairs in the relation *parent* in which the first element (the column *parent*) has the value "Mary", and returns the values in the column *child*.

A query for asking about grandparents of *Jack* could be formulated as $\exists y.(\text{parent}(x,y) \wedge \text{parent}(y,\text{Jack}))$.

A query for asking about grandmothers of *Jack* could be formulated as $\exists y.(\text{parent}(x,y) \wedge \text{parent}(y,\text{Jack})) \wedge \text{female}(x)$.

■

The standard relational algebra operations are as follows. (See Appendix B for formal definitions.)

- *natural join* $R_1 \bowtie R_2$ of two relations
- *relational union* $R_1 \cup R_2$ of two relations
- *relational intersection* $R_1 \cap R_2$ of two relations
- *relational difference* $R_1 - R_2$ of two relations
- *projection* $\pi_{a_1,\ldots,a_n}(R)$ which eliminates all columns from R except $a_1, \ldots, a_n$.
- *(Cartesian) product* $R_1 \times R_2$ that combines rows pairwise in all possible ways
- *selection* $\sigma_\phi(R)$ which chooses those rows of R that satisfy condition ϕ, and
- *renaming* $\rho_{a/b}(R)$ in which column b is renamed to a.
- *division* operation $R_1 \div R_2$ that satisfies the property $(Q \times P) \div P = Q$

The division operation that is very often ignored, also is not (very unfortunately) part of SQL.

In the SQL query language the *SELECT* statement combines projection, selection and renaming to one construct, and other operations like joins and unions are separate.

We present a translation from a large class of formulas of the predicate logic to the relation algebra. The relational algebra expression can be further relatively easily translated to practical query languages such as SQL.

The general scheme in the translation is as follows.

[3] A main difference between relational databases that use SQL and relational algebra is that the former treat relations as *multi-sets* of tuples, with the possibility of a tuple occurring in a relation multiple times, whereas standard relational algebra uses sets only. Other than that, much of SQL is best understood as syntactic sugar for relational algebra, with various extensions on top of it.

- Conjunction $\wedge$ translates to a *relational join* operation.[4]
- Disjunction $\vee$ translates to a *relational union* operation.
- Negation $\neg$ translates to a *relational difference* operation.
- Existential quantification $\exists$ translates to a *relational projection* operation.
- Universal quantification $\forall$ translates to a *relational division* operation.

Atomic formulas with non-variable terms (constant symbols) translate to the *selection* operation, which we discuss later in detail.

The translation from the predicate logic to the relational algebra is *compositional* in the sense that the translation of a subformula is formed from the translations of the subformulas in a uniform way.

Example 10.16 We show how the query $\exists x$.(parent(Mary,x)$\wedge$parent(x,z)$\wedge$female(z)) is translated to a relational algebra query. The results of this query are the female grandchildren of Mary. Let the column names be as in Example 10.15.

The atomic formula parent(x,z) is translated to

$$\rho_{x/parent,z/child}(parent).$$

Hence the relation on the left is mapped to a relation on the right. Only the names of the columns change.

parent

parent	child
Mary	Jack
John	Jack
Jack	Jill
Jack	Bob

x	z
Mary	Jack
John	Jack
Jack	Jill
Jack	Bob

The column names in this and all the other intermediate results are the names of the variables in the formula.

The atomic formula parent(Mary,x) is translated in three steps. First, those rows from *parent* are selected in which Mary is the parent:

$$\sigma_{parent=\text{Mary}}(parent)$$

Then the *child* column is projected.

$$\pi_{child}(\sigma_{parent=\text{Mary}}(parent)).$$

And finally, the *child* column is renamed to x.

$$\rho_{x/child}(\pi_{child}(\sigma_{parent=\text{Mary}}(parent))).$$

This expression maps the relation on the left to the relation on the right.

parent

parent	child
Mary	Jack
John	Jack
Jack	Jill
Jack	Bob

x
Jack

The atomic formula female(z) has no constant symbols, and it is translated similarly to parent(x,z).

$$\rho_{z/person}(female).$$

This expression maps the relation on the left to the relation on the right.

female

person
Jill
Mary

z
Jill
Mary

[4] The special case in which both conjuncts have the same free variables corresponds to the intersection operation for relations. Essentially, intersection is just a special case of the natural join.

What remains is the conjunction and the existential quantification. The conjunction is done with the natural join operation of the translations of all three conjuncts.

$$\rho_{x/child}(\pi_{child}(\sigma_{parent=\text{Mary}}(parent))) \bowtie \rho_{x/parent,z/child}(parent) \bowtie \rho_{z/person}(female)$$

$$\begin{array}{c} x \\ \hline \text{Jack} \end{array} \bowtie \begin{array}{cc} x & z \\ \hline \text{Mary} & \text{Jack} \\ \text{John} & \text{Jack} \\ \text{Jack} & \text{Jill} \\ \text{Jack} & \text{Bob} \end{array} \bowtie \begin{array}{c} z \\ \hline \text{Jill} \\ \text{Mary} \end{array} = \begin{array}{cc} x & z \\ \hline \text{Jack} & \text{Jill} \end{array}$$

Finally, the outermost existential quantification $\exists x$ means projecting the columns other than x.

$$\pi_z(\rho_{x/child}(\pi_{child}(\sigma_{parent=\text{Mary}}(parent))) \bowtie \rho_{x/parent,z/child}(parent) \bowtie \rho_{z/person}(female))$$

This projection maps the relation on the left to the relation on the right.

$$\begin{array}{cc} x & z \\ \hline \text{Jack} & \text{Jill} \end{array} \quad \begin{array}{c} z \\ \hline \text{Jill} \end{array}$$

The result of this relational algebra expression is a relation with a single column z that contains Mary's all female grandchildren. An automated translation would further produce the following SQL query.

```
SELECT x FROM
      (SELECT child as x
       FROM parent
       WHERE parent='Mary')
   NATURAL JOIN
      (SELECT parent as x, child as z
       FROM parent)
   NATURAL JOIN
      (SELECT person as z
       FROM female)
```

The three operations not covered in this example are disjunction, negation, and universal quantification.

Disjunction corresponds to the relational union operation, but there is an issue when the set of free variables, or, equivalently, the columns, in the two components to be unioned do not match.

Consider the formula $P(x, y) \vee Q(x)$. The disjuncts respectively correspond to a binary relation and a unary relation. The unary relation for $Q(x)$ would have to be made to a binary relation with columns x and y before the union operation. Since $Q(x)$ does not restrict the value of y at all, the y column in that relation should contain all possible values of y from the universe in consideration. This corresponds to there being a unary relation U that contains all elements of the universe, and a corresponding predicate, so that we can rewrite the disjunction to $P(x, y) \vee (Q(x) \wedge U(y))$.

Negation corresponds to relational difference. Examples like $P(x) \wedge \neg Q(x)$ are unproblematic because this directly corresponds to the difference of relations P and Q. But what if negation appears in the query without being a part of a conjunction with some positive elements? The query $\neg P(x)$, similarly to $P(x, y) \vee Q(x)$, would require considering the universe, from which $P(x)$ is to be deducted, to obtain the complement of P. Hence the formula would be rewritten to $U(x) \wedge \neg P(x)$, which can be translated to the relational algebra as the difference of these two relations $U - P$.

Universal quantification can be translated with the *relational division* operation (not present in SQL), or grouping and aggregation (present in SQL). Division can be effectively reduced to grouping and aggregation. Codd [Cod72] points out that division is (inefficiently) reducible to the other standard operations with the (rather expensive) complementation operation. We do not discuss the universal quantification here in more detail.

We now give the translation $[\![\phi]\!]$ from predicate logic to the relational algebra formally.

The notation $\mathrm{Fr}(\phi)$ refers to the variables that are free in ϕ.

Each relation R has a *degree* $\mathrm{Dgr}(R) \geq 1$ that indicates how many columns it has. We assume an ordering on the columns so that we can connect the relations to the predicates. The columns are numbered from 0 to $\mathrm{Dgr}(R) - 1$, and the name of column i is given by $\mathrm{Col}_i(R)$.

Definition 10.17 *The translation $[\![\phi]\!]$ of a predicate logic formula ϕ into the relational algebra is as follows.*

1. $[\![P(t_1,\ldots,t_n)]\!] = \rho_{t_{n_1}/Col_{n_1}(P),\ldots,t_{n_m}/Col_{n_m}(P)}(\pi_{Col_{n_1}(P),\ldots,Col_{n_m}(P)}(\sigma_C(P)))$ *where*
   - $n_1,\ldots,n_m$ *are indices of the subset of those terms $t_1,\ldots,t_n$ that are variables,*
   - *C is $Col_{k_1}(P) = t_{k_1},\ldots,Col_{k_l}(P) = t_{k_l}$ for the subset $\{k_1,\ldots,k_l\}$ of $\{t_1,\ldots,t_n\}$ that are constant symbols,*
   - *and we assume that no variable occurs twice in $t_1,\ldots,t_n$.*
2. $[\![\phi_1 \wedge \phi_2]\!] = [\![\phi_1]\!] \bowtie [\![\phi_2]\!]$
3. $[\![\phi_1 \vee \phi_2]\!] = [\![\phi_1 \wedge \bigwedge_{x\in Fr(\phi_2)\setminus Fr(\phi_1)} U(x)]\!] \cup [\![\phi_2 \wedge \bigwedge_{x\in Fr(\phi_1)\setminus Fr(\phi_2)} U(x)]\!]$
4. $[\![\neg\phi]\!] = [\![\bigwedge_{x\in Fr(\phi)} U(x)]\!] - [\![\phi]\!]$
5. $[\![\exists x\phi]\!] = \pi_{Fr(\phi)\setminus\{x\}}([\![\phi]\!])$
6. $[\![\forall x\phi]\!] = [\![\phi]\!] \div [\![U(x)]\!]$
7. $[\![t_1 = t_2]\!] = \sigma_{t_1=t_2}([\![\bigwedge_{x\in Fr(t_1=t_2)} U(x)]\!])$ *(when $Fr(t_1 = t_2) \neq \emptyset$)*
8. $[\![\phi_1 \wedge \neg\phi_2]\!] = [\![\phi_1 \wedge \bigwedge_{x\in Fr(\phi_2)\setminus Fr(\phi_1)} U(x)]\!] - [\![\phi_2 \wedge \bigwedge_{x\in Fr(\phi_1)\setminus Fr(\phi_2)} U(x)]\!]$

The last case describes the occurrence of the negation connective inside a conjunction. If all free variables inside the negation also occur in the other conjunct, then introducing the $U(x)$ atoms is avoided. Having a long conjunction $U(x_1) \wedge \cdots \wedge U(x_n)$ results in an n-fold natural join over the U relation, which has a very high cost as a database query operation. That is why it should be avoided as much as possible.

A formula can often be reformulated so that the first variant can be used with as small a set of variables as possible for which the predicate $U(x)$ is needed. For example, the formula $P(x) \wedge (Q(y) \wedge (R(z) \wedge \neg V(x)))$ can be equivalently be stated as $((P(x) \wedge Q(y)) \wedge R(z)) \wedge \neg V(x)$ for which the translation does not require any additional conjuncts with the U predicate. The general case, in which the universal predicate is needed for all free variables in $\neg\phi$, is only used as a last resort. Also, the complexity arising from cases 8 and 4 would sometimes be considerably reduced by translating the whole query formula to Negation Normal Form, so that negations would only appear right in front of atomic formulas.

The translation of all of the other connectives also benefits from logical simplifications and optimizations. These can be understood in terms of optimizing SQL queries, and some would be performed as a part of optimizations performed by the SQL query optimizer. For example, rewriting by the equivalences in Table 10.2 that reduce the scope of the quantifiers corresponds to moving relational projections deeper inside an SQL query, to reduce the size of the intermediate tables.

The reduction from predicate logic formulas to the relational algebra could obviously be composed with the translation from English to predicate logic in Section 10.6, to map database queries expressed in English to conventional SQL queries.

Index

β-reduction, 88
$\equiv$, 7
$\models$, 4, 7
abstraction, 16, 23
accessibility relation, 47
aletic logic, 49
apply, 20
assignment, 3
atomic formula, 71
atomic formulas, 47, 51
atomic proposition, 3
BDD, 18, 45
binardy decision diagram, 18
binary decision diagrams, 45
Boole's expansion, 17
Boolean constraint propagation, 12
bound occurrence, 71, 88
bounded model-checking, 67
branching time temporal logic, 53
Cartesian product, 93
CDCL, 13
clause, 14
CNF, 14
combinatorial explosion, 39
complement (of a literal), 11
Conflict-driven clause learning, 13
conjunctive normal form, 14
consistency, 7
CTL, 54, 58, 59, 66
CTL$*$, 53, 58
d-DNNF, 25
Davis-Putnam procedure, 13
Davis-Putnam-Logemann-Loveland procedure, 13
decomposability (formula), 24
determinism (formula), 25
disjunctive normal form, 15
division (relational), 93
DNF, 15
DNNF, 25
doxastic logic, 49
dynamic logic, 49, 55
empty clause, 11
epistemic logic, 49
euclidicity, 48
event, 34
eventuality sequence, 63
existential abstraction, 16, 23, 32
existential quantification, 72
fairness, 65
formula, 3, 71
frame, 47
frame axiom, 43
free occurrence, 71, 88
Hoare logic, 57
immediate subformula, 4
inconsistency, 7
join (relational), 31, 93
Kripke model, 47, 51
linear temporal logic, 52
literal, 14
logical consequence, 7, 51, 74
logical equivalence, 7
LTL, 52, 58, 60, 67
modal logic K, 58
modal logic S4, 49, 57, 58
modal logic S5, 49, 57, 58
model counting, 17, 22, 25
model, based on a frame, 47
model-checking, 58, 60
Montague grammar, 75
Montague semantics, 75
natural join, 31, 93
negation normal form, 15, 80
NNF, 15, 80
OBDD, 18, 45
p-morphism, 50
path formula, 52
path quantifier, 53
PDL, 55
pre-image operation, 40
precondition, 35
product (relational), 93
projection (relational), 31, 93
propositional dynamic logic, 55
quantifier, 71
reflexivity, 48
relational division, 93
relational join, 31
relational projection, 31, 93
relational renaming, 93
relational selection, 31
renaming (relational), 93

resolution, 11
restriction, 23
ROBDD, 18
S4, 49, 57, 58
S5, 49, 57, 58
SAT, 7
satisfiability, 6, 74
scope, 71, 88
SDD, 25
selection (relational), 31
Sentential decision diagram (SDD), 25
seriality, 48
Shannon expansion, 17
state, 34
state formula, 53
state variable, 34
structure (predicate logic), 73
subformula, 4
subsumption, 12
succinct transition system, 34
symmetry, 48
tautology, 6
temporal induction, 68
temporal logic, 49
term, 14, 71
transition relation, 29, 34
transition system, 34
transition system with state variables, 34
transitivity, 48
truth table, 4, 9
truth-assignment, 3
truth-definition (predicate logic), 73
unit clause, 11
unit propagation, 12
unit resolution, 11
unit subsumption, 12
universal abstraction, 16, 23, 32
universal quantification, 72
validity, 6, 74
valuation, 3
weak connectedness, 48
weakest precondition, 37

Bibliography

[AS09] Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern SAT solvers. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence*, pages 399–404. Morgan Kaufmann Publishers, 2009.

[Bar84] Hendrik P. Barendregt. *The lambda calculus*. North-Holland, 1984.

[BCCZ99] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In W. R. Cleaveland, editor, *Tools and Algorithms for the Construction and Analysis of Systems, Proceedings of 5th International Conference, TACAS'99*, volume 1579 of *Lecture Notes in Computer Science*, pages 193–207. Springer-Verlag, 1999.

[BCL+94] Jerry R. Burch, Edmund M. Clarke, David E. Long, Kenneth L. MacMillan, and David L. Dill. Symbolic model checking for sequential circuit verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(4):401–424, 1994.

[Bry86] Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, 100(8):677–691, 1986.

[Bry92] R. E. Bryant. Symbolic Boolean manipulation with ordered binary decision diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.

[Byl94] Tom Bylander. The computational complexity of propositional STRIPS planning. *Artificial Intelligence*, 69(1-2):165–204, 1994.

[CBM90] Olivier Coudert, Christian Berthet, and Jean Christophe Madre. Verification of synchronous sequential machines based on symbolic execution. In Joseph Sifakis, editor, *Automatic Verification Methods for Finite State Systems, International Workshop, Grenoble, France, June 12-14, 1989, Proceedings*, volume 407 of *Lecture Notes in Computer Science*, pages 365–373. Springer-Verlag, 1990.

[Chu36] Alonzo Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58(2):345–363, 1936.

[CM90] Olivier Coudert and Jean Christophe Madre. A unified framework for the formal verification of sequential circuits. In *Computer-Aided Design, 1990. ICCAD-90. Digest of Technical Papers., 1990 IEEE International Conference on*, pages 126–129. IEEE Computer Society Press, 1990.

[CMV09] Benjamin Chambers, Panagiotis Manolios, and Daron Vroon. Faster SAT solving with better CNF generation. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 1590–1595, 2009.

[Cod72] Edgar F. Codd. Relational completeness of data base sublanguages. Technical report, IBM Corporation, 1972.

[Cod83] Edgar Frank Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 26(1):64–69, 1983.

[Coo71] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, pages 151–158, 1971.

[Dar01] Adnan Darwiche. Decomposable negation normal form. *Journal of the ACM*, 48(4):608–647, 2001.

[Dar02] Adnan Darwiche. A compiler for deterministic, decomposable negation normal form. In *Proceedings of the 18th National Conference on Artificial Intelligence (AAAI-2002) and the 14th Conference on Innovative Applications of Artificial Intelligence (IAAI-2002)*, pages 627–634, 2002.

[Dar11] Adnan Darwiche. SDD: A new canonical representation of propositional knowledge bases. In *Proceedings of the 22nd International Joint Conference on Artificial Intelligence*, pages 819–826. AAAI Press, 2011.

[DG84] William F. Dowling and Jean H. Gallier. Linear-time algorithms for testing the satisfiability of propositional Horn formulae. *Journal of Logic Programming*, 1(3):267–284, 1984.

[Dij75] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975.

[DLL62] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Communications of the ACM*, 5:394–397, 1962.

[DM02] Adnan Darwiche and Pierre Marquis. A knowledge compilation map. *Journal of Artificial Intelligence Research*, 17:229–264, 2002.

[DNK97] Yannis Dimopoulos, Bernhard Nebel, and Jana Koehler. Encoding planning problems in nonmonotonic logic programs. In *Recent Advances in AI Planning. Fourth European Conference on Planning (ECP'97)*, number 1348 in Lecture Notes in Computer Science, pages 169–181. Springer-Verlag, 1997.

[DP60] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, 1960.

[DRKNP16] Luc De Raedt, Kristian Kersting, Sriraam Natarajan, and David Poole. *Statistical Relational Artificial Intelligence: Logic, Probability, and Computation*. Synthesis Lectures on Artificial Intelligence and Machine Learning. 2016.

[Eme90] E. Allen Emerson. Temporal and modal logic. In J. Van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 995–1072. Elsevier Science Publishers, 1990.

[ES03] Niklas Eén and Niklas Sörensson. Temporal induction by incremental SAT solving. *Electronic Notes in Theoretical Computer Science*, 89(4):543–560, 2003.

[FL77] M. J. Fischer and R. E. Ladner. Propositional modal logic of programs. In *Proceedings of the 9th Symposium on the Theory of Computing*, pages 286–294, 1977.

[FL79] Michael J. Fischer and Richard E. Ladner. Propositional dynamic logic of regular programs. *Journal for Computer and System Sciences*, 18(2):194–211, 1979.

[FL89] Richard Frost and John Launchbury. Constructing natural language interpreters in a lazy functional language. *The Computer Journal*, 32(2):108–121, 1989.

[Flo67] R. W. Floyd. Assigning meanings to programs. In *Proceedings of the Symposium on Applied Mathematics*, volume 19, pages 19–31. American Mathematical Society, 1967.

[Fro06] Richard A. Frost. Realization of natural language interfaces using lazy functional programming. *ACM Computing Surveys (CSUR)*, 38(4):11–es, 2006.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability*. W. H. Freeman and Company, San Francisco, 1979.

[GW83] Hana Galperin and Avi Wigderson. Succinct representations of graphs. *Information and Control*, 56:183–198, 1983. See [Loz88] for a correction.

[HD05] Jinbo Huang and Adnan Darwiche. DPLL with a trace: From SAT to knowledge compilation. In Leslie Pack Kaelbling, editor, *Proceedings of the 19th International Joint Conference on Artificial Intelligence*, pages 156–162. Professional Book Center, 2005.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[Kri63a] Saul Kripke. Semantical analysis of modal logic i, normal propositional calculi. *Zeitschrift für mathematische Logik und Grundlagen der Mathematik*, 9:67–96, 1963.

[Kri63b] Saul Kripke. Semantical considerations on modal logics. *Acta Philosophica Fennica*, pages 83–94, 1963.

[Kri65] Saul Kripke. Semantical analysis of modal logic ii. non-normal modal propositional calculi. In J. W. Addison, L. Henkin, and A. Tarski, editors, *The Theory of Models*, pages 206–220. North-Holland, Amsterdam, 1965.

[KS92] Henry Kautz and Bart Selman. Planning as satisfiability. In Bernd Neumann, editor, *Proceedings of the 10th European Conference on Artificial Intelligence*, pages 359–363. John Wiley & Sons, 1992.

[KS96] Henry Kautz and Bart Selman. Pushing the envelope: planning, propositional logic, and stochastic search. In *Proceedings of the 13th National Conference on Artificial Intelligence and the 8th Innovative Applications of Artificial Intelligence Conference*, pages 1194–1201. AAAI Press, 1996.

[LB90] Antonio Lozano and José L. Balcázar. The complexity of graph problems for succinctly represented graphs. In Manfred Nagl, editor, *Graph-Theoretic Concepts in Computer Science, 15th International Workshop, WG'89*, number 411 in Lecture Notes in Computer Science, pages 277–286. Springer-Verlag, 1990.

[Loz88] Antonio Lozano. NP-hardness of succinct representations of graphs. *Bulletin of the European Association for Theoretical Computer Science*, 35:158–163, June 1988.

[LP77] Michel Lacroix and Alain Pirotte. Domain-oriented relational languages. In *Proceedings of the Third International Conference on Very Large Data Bases*, volume 3, pages 370–378, 1977.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978.

[MMZ$^+$01] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: engineering an efficient SAT solver. In *Proceedings of the 38th ACM/IEEE Design Automation Conference (DAC'01)*, pages 530–535. ACM Press, 2001.

[Mon70] Richard Montague. Universal grammar. *Theoria*, 36:373–398, 1970.

[Mon73] Richard Montague. The proper treatment of quantification in ordinary English. In *Approaches to natural language*, pages 221–242. Springer-Verlag, 1973.

[MR06] Brian Milch and Stuart J. Russell. First-order probabilistic languages: Into the unknown, inductive logic programming. volume 4455 of *Lecture Notes in Computer Science*, pages 10–24. Springer-Verlag, 2006.

[MSS96] João P. Marques-Silva and Karem A. Sakallah. GRASP: A new search algorithm for satisfiability. In *Computer-Aided Design, 1996. ICCAD-96. Digest of Technical Papers., 1996 IEEE/ACM International Conference on*, pages 220–227, 1996.

[NMTG15] Maximilian Nickel, Kevin Murphy, Volker Tresp, and Evgeniy Gabrilovich. A review of relational machine learning for knowledge graphs. *Proceedings of the IEEE*, 104(1):11–33, 2015.

[PD07] Knot Pipatsrisawat and Adnan Darwiche. A lightweight component caching scheme for satisfiability solvers. In Joao Marques-Silva and Karem A. Sakallah, editors, *Proceedings of the 8th International Conference on Theory and Applications of Satisfiability Testing (SAT-2007)*, volume 4501 of *Lecture Notes in Computer Science*, pages 294–299. Springer-Verlag, 2007.

[Pra76] Vaughan R. Pratt. Semantical considerations on Floyd-Hoare logic. In *Proceedings of the 17th IEEE Symposium on Foundations of Computer Science*, pages 109–121, 1976.

[RD06] Matthew Richardson and Pedro Domingos. Markov logic networks. *Machine learning*, 62(1-2):107–136, 2006.

[RHN06] Jussi Rintanen, Keijo Heljanko, and Ilkka Niemelä. Planning as satisfiability: parallel plans and algorithms for plan search. *Artificial Intelligence*, 170(12-13):1031–1080, 2006.

[Rin04] Jussi Rintanen. Evaluation strategies for planning as satisfiability. In *ECAI 2004. Proceedings of the 16th European Conference on Artificial Intelligence*, pages 682–687. IOS Press, 2004.

[SLM92] B. Selman, H. Levesque, and D. Mitchell. A new method for solving hard satisfiability problems. In *Proceedings of the 11th National Conference on Artificial Intelligence*, pages 46–51, 1992.

[SS07] Matthew Streeter and Stephen F. Smith. Using decision procedures efficiently for optimization. In *ICAPS 2007. Proceedings of the Seventeenth International Conference on Automated Planning and Scheduling*, pages 312–319. AAAI Press, 2007.

[SSS00] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. Checking safety properties using induction and a SAT-solver. In W. A. Hunt and S. D. Johnson, editors, *Formal Methods in Computer-Aided Design, Third International Conference, FMCAD 2000, Austin, Texas, USA, November 1-3, 2000, Proceedings*, volume 1954 of *Lecture Notes in Computer Science*, pages 108–125. Springer-Verlag, 2000.

[Tse68] G. S. Tseitin. On the complexity of derivations in propositional calculus. In A. O. Slisenko, editor, *Studies in Constructive Mathematics and Mathematical Logic, Part II*, pages 115–125. Consultants Bureau, 1968.

[VEU10] Jan Van Eijck and Christina Unger. *Computational semantics with functional programming*. Cambridge University Press, 2010.

Appendix A

The λ-Calculus

The standard definition of *functions* in mathematics and in set-theory is as relations $f \subseteq D_1 \times D_2$ that satisfy the condition that if $(x, y) \in f$ and $(x, z) \in f$, then $y = z$, and where D_1 is the *domain* of the the function and D_2 is the *range* of the function. Functions are *sets of pairs* that map inputs to outputs. For example, the successor function of natural numbers is represented by the infinite set $\{(0, 1), (1, 2), (2, 3), (3, 4), \ldots\}$.

This definition does not say anything about how something is computed. Also, this definition is infinite for every infinite domain, no matter how simple the function is.

Many functions can be represented finitely simply as a mathematical expression such as $f(x) = x + 1$. This finite representation of functions is what is used in much of Computer Science. Besides the finite size, it has the advantage of being implementable in a computer, as a sequence of computation steps.

Well-known formalization of computations as sequences of primitive computation steps include conventional real-world CPU instruction sets as well as more theoretical models of computation such as Turing machines. These, however, do not formalize the notion of *function* explicitly in any way.

A major model of computation that does precisely that is λ-calculus [Chu36, Bar84]. In addition to an abstract model of computation, λ-calculus can also be used as a formal model of *programming languages*: many (pure) functional programming languages can be viewed as λ-calculus extended with primitive functions for various datatypes such as numbers and lists.

Here we give a brief introduction to λ-calculus.

A.1 Basics

First, we introduce the λ notation, which is also used in some programming languages for unnamed functions. The intuition behind the λ notation is demonstrated by the following.

Example A.1
- $\lambda x.(x + 1)$ is the unnamed function that increments its input by one. Applying this function to the number 5 results in $(\lambda x.(x + 1))$ 5 = 5+1 = 6.
- $\lambda x.\lambda y.(x + y)$ denotes a function that maps some number x to a function that maps some number y to the sum $x + y$
  - $(\lambda x.\lambda y.(x + y))$ 5 = $\lambda y.(5 + y)$
  - $(\lambda x.\lambda y.(x + y))$ 5 7 = $(\lambda y.(5 + y))$ 7 = $5 + 7 = 12$

Next we proceed with the formal definition of the expressions of the pure λ-calculus.

Definition A.2 (Expression)
1. *x is an expression if x is a variable,*
2. *$(E_1 E_2)$ is an expression if E_1 and E_2 are expressions,*
3. *$(\lambda x.E)$ is an expression if x is a variable and E is an expression.*

Example A.3 (Expressions)

$$x$$
$$(xy)$$
$$(\lambda x.x)$$
$$((\lambda x.x)(\lambda y.y))$$
$$(\lambda x.(\lambda y.(xy)))$$

■

Parentheses can be dropped if it does not cause ambiguity.

expression	shortform	
$(\lambda x.(xy))$	$\lambda x.xy$	
$(\lambda x.x)y$	$(\lambda x.x)y$	
$((xy)z)$	xyz	applicationleft − associative
$(x(yz))$	$x(yz)$	

The notion of bound and free variables is analogous to the definition in the predicate logic.

Definition A.4 (Bound and free) *In sub-expression $\lambda x.E$ of any expression, occurrences of x in E are* bound*. Here E is the* scope *of x. An occurrence of x is* free *if it is not bound.*

Example A.5

expression	freevariables
$\lambda x.x$	$none$
$\lambda x.xy$	y
$(\lambda x.x)(\lambda y.y)$	$none$
$\lambda x.\lambda y.(xy)$	$none$
$\lambda x.((\lambda y.(xy))y)$	y
$(\lambda x.((xy)z))u$	y, z, u

■

The primitive computation step in the λ-calculus is that of a single function application.

Definition A.6 (β-reduction) *One step of β-reduction turns $(\lambda x.E_1)E_2$ to $E_1[E_2/x]$, which is E_1 with every free occurrence of x replaced by E_2.*

Note: When we say "every free occurrence of x" above, we refer to free occurrences of x in E_1, not to free occurrences of x in $\lambda x.E_1$.

Example A.7

expression	resultofβreduction
$(\lambda x.x)y$	y
$(\lambda x.xx)y$	yy
$(\lambda x.x)(\lambda y.y)$	$\lambda y.y$
$(\lambda x.\lambda y.y)z$	$\lambda y.y$
$(\lambda x.xx)(\lambda y.y)$	$(\lambda y.y)(\lambda y.y)$
$(\lambda x.(x(\lambda x.x)))E$	$E(\lambda x.x)$

■

There may be several possible β-reduction steps applicable to a given expression. When evaluating an expression "until the end", different choices for the next β-reduction step lead to different *evaluation strategies*.

Definition A.8 (Normal order evaluation) *In* normal order *evaluation the leftmost β-reduction is performed first.*

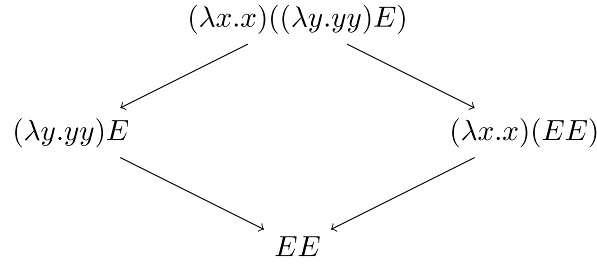


Figure A.1: Example of two evaluation orderings leading to the same result

Definition A.9 (Applicative order evaluation) *In* applicative order *evaluation, in any sub-expression* $(\lambda x.E_1)E_2$*, all* β*-reductions in* E_2 *performed before the outermost/leftmost* β*-reduction that substitutes* E_2 *for* x*.*

Example A.10 With normal order evaluation we get $(\lambda x.xx)((\lambda z.z)E) \leadsto ((\lambda z.z)E)((\lambda z.z)E)$.
With applicative order evaluation we get $(\lambda x.xx)((\lambda z.z)E) \leadsto (\lambda x.xx)E$ ■

Almost all programming languages use applicative order: in $f(E)$, expression E is evaluated before "calling" the function f. *Lazy* functional programming languages use a form of normal order that avoids evaluating an expression multiple times. Normal order evaluation makes it possible for example to create an "infinite list", and any computation that uses only a finite portion of it will still terminate.

A central result in the λ-calculus is its *Church-Rosser* property, which means that no matter which evaluation strategy is used, all terminating evaluation sequences will lead to the same result. This is illustrated in Figure A.1.

Theorem A.11 (Church-Rosser) *Any two terminating maximal* β*-reduction sequences end in the same expression.*

A.2 Types

Expression in the λ-calculus, similarly to programming languages, can be assigned types.

We first consider functions that have a type that uniquely determines the types of its inputs and outputs.

Example A.12 The Successor function $S(n) = n + 1$ has type $\mathbb{N} \to \mathbb{N}$. The addition function $A(x, y) = x + y$ has type $\mathbb{N} \times \mathbb{N} \to \mathbb{N}$. ■

Next we formally define types.

Definition A.13 (Types) *1. Atomic types are types*
2. If t_0 *and* t_1 *are types, then so is* $t_0 \to t_1$

Example A.14

$$\begin{array}{l} t \\ t \to t \\ t \to (t \to t) \\ (t \to t) \to t \\ (t \to t) \to (t \to t) \\ t \to ((t \to t) \to (t \to t)) \end{array}$$

■

Example A.15 Assume that the type $\mathbb{N}$ for natural numbers is the primitive type. Then we can assign types to the following expressions.

expression	type
$\lambda x.(1+x)$	$\mathbb{N} \to \mathbb{N}$
$\lambda x.\lambda y.(x+y)$	$\mathbb{N} \to (\mathbb{N} \to \mathbb{N})$
$\lambda x.\lambda y.\lambda z.(x+y+z)$	$\mathbb{N} \to (\mathbb{N} \to (\mathbb{N} \to \mathbb{N}))$
$\lambda f.\lambda x.(1+f(x+1))$	$(\mathbb{N} \to \mathbb{N}) \to (\mathbb{N} \to \mathbb{N})$
$\lambda i.\lambda f.\lambda x.(1+f(x+1+i))$	$\mathbb{N} \to ((\mathbb{N} \to \mathbb{N}) \to (\mathbb{N} \to \mathbb{N}))$

■

Remark A.16 *$E_1 : t_0 \to t_1$ can be applied to $E_2 : t_2$ only if $t_0 = t_2$. Then E_1E_2 has type t_1.*

Example A.17 $E_1 = \lambda i.\lambda f.\lambda x.(f(x+1+i))$ has type $\mathbb{N} \to ((\mathbb{N} \to \mathbb{N}) \to (\mathbb{N} \to \mathbb{N}))$
Application E_1E_2 is possible only if E_2 has type $\mathbb{N}$.
$(\lambda i.\lambda f.\lambda x.(f(x+1+i)))7$ results in $\lambda f.\lambda x.(f(x+1+7))$. ■

Clearly, a function $\lambda x.x$ is applicable to objects of any type, so we do not need to fix the type of its input. We could use *type variables* t instead, and say that the type of $\lambda x.x$ is $t \to t$. We do not discuss this topic here, because it would require more complex concepts such as *type unification* to be able to define when application E_1E_2 is possible. The possibility to have type variables in the types of a function is extensively used in programming languages with a *polymorphic type system* [Mil78]. Examples of such languages are the ML family (Standard ML, OCAML) and lazy functional languages such as Haskell and Miranda. The advantage of such type systems it that any type errors are detected during the compilation process, and hence type errors are impossible when the program is run.

A.2.1 Type Inference

There are only two principles in typing λ-expressions:

P1 For EF, type of E is $T_1 \to T_2$, type of F is T_1, and type of EF is T_2, for some types T_1 and T_2.

P2 For $\lambda x.E$, type of X is T_1, type of E is T_2, and type of $\lambda x.E$ is $T_1 \to T_2$, for some types T_1 and T_2.

In the examples in the course material, in *non-pure* λ-expressions, we additionally can type some variables based on the types of "built-in" functions, such as arithmetic operators, such as x as INT if x occurs e.g. in $x+1$. These are always specific to the functions and operators in question, and we do not use them below.

Now, the problem is how to figure out what these T_1 and T_2 are in each case. The general algorithms for doing this recursively go through a λ-expression, assign each subexpression some types $T_1, T_2, T_3, \ldots$, and then do *type unification* to establish connections between the types. For example, if in EF we got type T_1 for E and type T_2 for F, then from the principle $P1$ we have that it must be $T_1 = T_3 \to T_4$ for some types T_3 and T_4, and T_2 must be T_3.

Example A.18 In $\lambda x.x$, x is of some unknown type T_1, so $\lambda x.x$ has type $T_1 \to T_2$ by $P1$. Obviously, it must be that $T_2 = T_1$, because the body is just x. So the type of $\lambda x.x$ is $T_1 \to T_1$. ■

Example A.19 Next we find the typing for $\lambda x.\lambda y.xy$ by systematically assigning a type to each sub-expression, and in the end unifying the types.

1. By Principle $P2$, from $\lambda x.\lambda y.xy$ we get
   (a) x has type T_1
   (b) $\lambda x.\lambda y.xy$ has type $T_1 \to T_2$
   (c) $\lambda y.xy$ has type T_2
2. By Principle P2, from $\lambda y.xy$ we get
   (a) y has type T_3
   (b) $\lambda y.xy$ has type $T_3 \to T_4$
   (c) xy has type T_4
3. By Principle $P1$, from xy we get

(a) x has type $T_5 \to T_6$
(b) y has type T_5
(c) xy has type T_6

Now we *unify* these, as some expressions have different types assigned to them in (1), (2) and (3).

- For x we get: $T_1 = (T_5 \to T_6)$
- For y we get: $T_5 = T_3$
- For xy we get: $T_6 = T_4$
- For $\lambda y.xy$ we get: $T_2 = (T_3 \to T_4)$
- For $\lambda x.\lambda y.xy$ we get: $(T_1 \to T_2)$

And then replace every simpler type with the equal more complex type: T_1 replaced by $T_5 \to T_6$ and T_2 replaced by $T_3 \to T_4$:

- For x we get: $T_1 = (T_5 \to T_6)$
- For y we get: $T_5 = T_3$
- For xy we get: $T_6 = T_4$
- For $\lambda y.xy$ we get: $T_2 = (T_3 \to T_4)$
- For $\lambda x.\lambda y.xy$ we get: $((T_5 \to T_6) \to (T_3 \to T_4))$

Additionally, $T_3 = T_5$ and $T_4 = T_6$, so eliminate one of these type variables in each pair (we arbitrarily choose to eliminate the second one):

- $x : T_3 \to T_4$
- $y : T_3$
- $xy : T_4$
- $\lambda y.xy : T_2 = (T_3 \to T_4)$
- $\lambda x.\lambda y.xy : ((T_3 \to T_4) \to (T_3 \to T_4))$

■

One could read the same typing more directly from $\lambda x.\lambda y.xy$: x is a function with input of the type y, so we have

- $x : T_1 \to T_2$
- $y : T_1$
- $xy : T_2$
- $\lambda y.xy : T_1 \to T_2$
- $\lambda x.\lambda y.xy : (T_1 \to T_2) \to (T_1 \to T_2)$

A.3 Expressive Power of Pure λ-Calculus

Pure λ-calculus is Turing-equivalent, and it can express all computations expressible as Turing machines.

There are several schemes for encoding natural numbers with addition and arithmetic operations in λ-calculus, and this is one route to reducing Turing-machine computations to λ-calculus.

Well known *combinators* expressible in λ-calculus include the following.

- $I = \lambda x.x$
- $K = \lambda x.(\lambda y.x)$
- $S = \lambda x.\lambda y.\lambda z.xz(yz)$
- $Y = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$
- $\Theta = (\lambda xy.y(xxy))(\lambda xy.y(xxy))$

Combinators S and K are enough to express all computations, and they have been used as primitives in implementing functional programming languages. The identity function I can be expressed as $I = SKK$.

The lack of names of functions in λ-calculus prevents the direct expression of recursion, as reference to the function itself inside its body is not possible. Instead, the *fixpoint combinators* Y and Θ can be used for expressing recursion.

Appendix B

Relational Algebra

Relational algebra is one of the fundamental formalisms used in database research. The implementation of relational databases, which is the most powerful class of databases in wide use, is based on relation algebra operations. SQL is a query language for relational bases, and much of the core functionality of SQL uses operations from relational algebra. The SELECT construct of SQL combines projection, renaming and selection in one syntax, and inside SELECT there can be join, union and other relational operations.

A relation in mathematics is a set of fixed-length sequences of some values. A binary relation is a set of pairs (e_1, e_2), a ternary relation is a set of triples (e_1, e_2, e_3), and so on. In each element of a relation the component values can be referred to based on their position in the element.

In contrast, in relational algebra, the positions are named. For example, the following relation could represent the relation "to be the parent of".

parent	child
Jack	Mary
Jack	John
Jill	Mary

This can be represented as a conventional mathematical relation as the set $\{(\text{Jack}, \text{Mary}), (\text{Jack}, \text{John}), (\text{Jill}, \text{Mary})\}$.

In relational algebra, we refer to *parent* and *child* as *columns*. For many of the central operations in the relational algebra, including *natural join*, it is critical that the columns have names.

To represent the relational algebra relations faithfully with set-theoretic concepts, we view the elements of a relation as mappings from columns to values. This leads to – following the standard definition of mappings or functions as sets – representing the above relation as the following set sets.

$$\begin{aligned}\{\ &\{(\text{“parent”}, \text{“Jack”}), (\text{“child”}, \text{“Mary”})\},\\ &\{(\text{“parent”}, \text{“Jack”}), (\text{“child”}, \text{“John”})\},\\ &\{(\text{“parent”}, \text{“Jill”}), (\text{“child”}, \text{“Mary”})\}\}\end{aligned}$$

Notice that unlike in the standard mathematical definition of relations where there is an ordering on every tuple in the relation, in this definition the columns are not ordered, and the following is set-theoretically exactly the same entity as the previous one, with both columns and rows written in a different order.

$$\begin{aligned}\{\ &\{(\text{“child”}, \text{“Mary”}), (\text{“parent”}, \text{“Jack”})\},\\ &\{(\text{“child”}, \text{“Mary”}), (\text{“parent”}, \text{“Jill”})\},\\ &\{(\text{“parent”}, \text{“Jack”}), (\text{“child”}, \text{“John”})\}\}\end{aligned}$$

Now we can formally define the operations of the relational algebra.

Some operations of the relations algebra are trivial, including *union*, *intersection*, and *difference*, because they are the standard set-theoretic operations.

name	symbol	definition
union	$\cup$	$R_1 \cup R_2 = \{e \mid e \in R_1 \text{ or } e \in R_2\}$
intersection	$\cap$	$R_1 \cap R_2 = \{e \mid e \in R_1 \text{ and } e \in R_2\}$
difference	$-$	$R_1 - R_2 = \{e \mid e \in R_1 \text{ and } e \notin R_2\}$

Relational algebra allows these three set-theoretic operations on relations only when the columns for the two relations are the same.

One of the core operations in the relational algebra is that of *natural join*. This operations combines two relations to a new relation that has all the columns of the two components, and each element is a combination of two elements from the two components so that the joint columns contain the same value.

The *natural join* $\bowtie$ is defined by

$$R_1 \bowtie R_2 = \{r_1 \cup r_2 | r_1 \in R_1, r_2 \in R_2, \forall c \in \text{Cols}(R_1) \cap \text{Cols}(R_2).r_1(c) = r_2(c)\}.$$

Here Cols(R) denotes the column names in relation R, defined by Cols$(R) = \{n | r \in R, (n,v) \in r \text{ for some } v\}$. Notice that the empty relation always has no columns, and this is not problematic for the definition of the natural join operation.

The *intersection* $\cap$ of two relations is defined directly by set-theoretic intersection, whenever the columns of the two relations are the same. Note that the intersection is a special case of natural join, and that's why it is often not introduced as a separate operation.

The *Cartesian product* of two relations that do not share columns can be defined as follows.

$$R_1 \times R_2 = \{r_1 \cup r_2 | r_1 \in R_1, r_2 \in R_2\}$$

So here we require that Cols$(R_1) \cap$ Cols$(R_2) = \emptyset$. Note that this is a special case of natural join: if the columns are disjoint, there are no values that have to coincide, and all pairs of elements from the two relations end up in the resulting relation. For this reason the Cartesian product operation is used in place of the join operation mostly just to emphasize that there are no common columns in the two relations.

The *projection* operation $\pi_{n_1,\ldots,n_k}(R)$ eliminates some of the columns, keeping only those listed in $n_1, \ldots, n_k$. It is defined as follows.

$$\pi_{n_1,\ldots,n_k}(R) = \{\{(n,v) \in r | n \in \{n_1, \ldots, n_k\}\}, r \in R\}$$

The *renaming* operation changes the names of one column of a relation.

$$\rho_{n_1/n_0}(R) = \{\{(n,v) \in r | n \neq n_0\} \cup \{(n_1,v) | (n,v) \in r, n = n_0\}\} | r \in R\}$$

The *selection* operation selects the subset of those elements of a relation that satisfy a given condition. The condition can refer to the values in the different columns in one element. In general it could be any Boolean combination of comparisons between the values in the different columns and any constants.

$$\sigma_\phi(R) = \{r \in R | \phi\}$$

We could allow all Boolean connectives in ϕ ($\vee, \wedge, \neg, \leftrightarrow, \rightarrow, \ldots$) and use equalities $t_1 = t_2$ as the atomic conditions in ϕ, where t_1 and t_2 are either column names, evaluating to the value in that column in a given element, or constants of the type the column is (e.g. strings, integer, dates, times, or whatever value is stored in that column.) In addition to equality $=$, for numeric values of course comparisons $<$, $\leq$ and so on could be used.

Finally, the *relational division* operation is defined by

$$R_1 \div R_2 = \{\text{cut}(r_1, R_1, R_2) | r_1 \in R_1, \forall r_2 \in R_2.((\text{cut}(r_1, R_1, R_2) \cup r_2) \in R_1)\}$$

where we define cut$(r_1, R_1, R_2) = \{(n,v) \in r_1 | n \in \text{Cols}(R_1) - \text{Cols}(R_2)\}$. This is the row r_1 of relation R_1 limited to the columns that only occur in R_1 and not in R_2. For the division to be well-defined, we require that Cols$(R_2) \subseteq$ Cols(R_1).

The division operation can be reduced to the other operations by

$$R_1 \div R_2 = \pi_{C_1}(R_1) - \pi_{C_1}(\pi_{C_1}(R_1) \times R_2 - R_1) \tag{B.1}$$

where $C_1 = \text{Cols}(R_1) - \text{Cols}(R_2)$. The computation of the right-hand side of the difference is expensive and generally impractical, and that is the reason why this reduction is rarely used, and why it is important to keep division as one of the primitive operations.

Doing what is (implicitly) universal quantification is also one of the main motivations for the aggregation operations in SQL. Many complex database queries used in practice are essentially about universal quantification, and SQL makes it possible to express them by using aggregation and related features.

Example B.1 We have two relations, *TeamLeader* (column "person"), which lists all company employees that are team leaders, and *Skill* (columns "person" and "skill"), which indicates which skills different people have.

Consider the following two queries.

1. "Find all team leaders with at least one of the skills Python, SQL, Javascript"
2. "Find all team leaders with all of the skills Python, SQL, Javascript"

Let these skills be in the relation *BasicSkills*, with only one column "skill". So the three relations needed in these queries could for example be as follows.

TeamLeader

person
Derek
Jack
Jill
Mary
Rebecca

Skill

person	*skill*
Derek	Fortran
Hamish	Python
Hamish	SQL
Hamish	Javascript
Jack	Python
Jack	MUMPS
Jill	Python
Mary	Python
Mary	SQL
Mary	Javascript

BasicSkills

skill
Python
SQL
Javascript

The first query is done as a join followed by projection as

$$\pi_{\text{person}}(\text{Employee} \bowtie \text{Skill} \bowtie \text{BasicSkill}).$$

This is trivial to express in SQL as follows.

```
SELECT person FROM Employee NATURAL JOIN Skill NATURAL JOIN BasicSkill
```

The second query has universal quantification in the word "all", and it is done most naturally by using division (but could be reduced to the other operation, as shown in Equation (B.1), but very inefficiently.)

$$\pi_{\text{person}}((\text{TeamLeader} \bowtie \text{Skill}) \div \text{BasicSkills}).$$

This is more complicated, and we go through it step by step. First compute TeamLeader $\bowtie$ Skill as follows.

TeamLeader

person
Derek
Jack
Jill
Mary
Rebecca

$\bowtie$

Skill

person	*skill*
Derek	Fortran
Hamish	Python
Hamish	SQL
Hamish	Javascript
Jack	Python
Jack	MUMPS
Jill	Python
Mary	Python
Mary	SQL
Mary	Javascript

$=$

person	*skill*
Derek	Fortran
Jack	Python
Jack	MUMPS
Jill	Python
Mary	Python
Mary	SQL
Mary	Javascript

Then we divide the resulting relation by *BasicSkills*.

person	*skill*
Derek	Fortran
Jack	Python
Jack	MUMPS
Jill	Python
Mary	Python
Mary	SQL
Mary	Javascript

$\div$

BasicSkills

skill
Python
SQL
Javascript

$=$

person
Mary

The result is obtained by picking all those *person*s in the leftmost table for which the *skill* column includes all *skill*s in **BasicSkills**. The corresponding SQL query is quite a bit more complicated, involving a sub-query, COUNT, and GROUP BY.

```
SELECT person
FROM
TeamLeader NATURAL JOIN Skill
GROUP BY person
HAVING COUNT(*) = (SELECT COUNT(*)
                   FROM BasicSkills)
```

This could be done with more basic SQL by using nested DIFFERENCE/EXCEPT operations, as shown in Equation (B.1), but that would be very inefficient.