Logic via Foundational Algorithms

James Hook and Tim Sheard

March 1, 2011

Contents

1 Overview; Introduction to Propositional Logic

1.1 Course Goals and Outline

It is difficult to overstate the foundational role that logic plays in computer science. The notion of computation was formalized in an attempt to clarify the foundations of logic. Intuitionistic proof theory, once considered obscure, has emerged as an organizing model for programming language type systems. The satisfiability problem, SAT, has become the paradigmatic NP-complete problem. SAT solvers, and their close relatives SMT-solvers, have become powerful tools, used by many to solve problems that are occasionally intractable, but frequently manageable.

In this course we will approach some of the classical results in elementary mathematical logic from a decidedly computer science perspective. We will write programs that mechanize proof systems. We will write programs that reduce problems to instances of SAT as a means to getting efficient solutions. We will look at how logic is automated in various families of theorem provers.

The course is experimental. It will partially be driven by student interests.

Course Goals:

1. Introduce the basic definitions and results in mathematical logic. Specifically propositional logic, predicate logic, and modal logics. Understand how logic expresses the differences between syntax, models, and proofs.

2. Explore the foundational algorithms. We will look at proof checkers, the interactive proof development problem, tautology checkers, and SAT solvers. We may look at more sophisticated theorem provers if time permits.

3. Lay the foundation for future study in mathematical logic, theorem proving, type theory, and problem solving with SAT and SMT solvers.

1.2 Propositional logic

The study of logic in mathematics generally starts from two simple systems. Propositional logic reasons about the truth of very simple declarative sentences. It allows us to focus on the essence of logical connectives. Predicate logic reasons about predicates over a collection of things, sometimes called the universe or domain of discourse. In predicate logic we add key complications: how do we name the things (terms are introduced to name things), what basic properties of things can we reason about (the atomic predicates), and how can we reason about general properties of things stated with quantifiers.

We start with the simplest: Propositional logic.

1.2.1 How to understand a logic?

The general program of mathematical logic reduces a logic to three fundamental aspects: syntax, semantics, and proofs.

- Syntax

  The syntax is generally given by an inductive definition. Huth and Ryan present one (somewhat informally) on page 4. We will give a Haskell datatype below. For propositional logic we will have a set of atomic propositions (proposition letters) that can be combined with logical connectives.

- Semantics or interpretation

  The semantics of a logic is given by mapping the syntax to a mathematical structure in a systematic way. For propositional logic we will show how to take a truth assignment for the atomic propositions and extend that to calculate the truth of any well-formed-formula.

- Proof

  Informally proofs capture the structure of arguments. Formally we will have a set of objects called proofs and an effective mechanism (computer program) to decide if a proof proves a statement.

Philosophical aside. Which is primary, the semantics or the proof system? In classical logic, the semantics is taken as primary. Truth is defined using the mechanisms of mathematics. Proof is secondary. Proof systems are measured against how well they describe the mathematics. For the intuitionists, however, proof is primary. Proof is an abstraction of pure thought that transcends all language. We have proof before language, and before mathematics. The fact that we can use mathematics to codify proof and reflect it as a formal system is just anecdotal evidence for the utility of mathematics.

1.2.2 Syntax of Propositional Logic

[See HR Definition 1.27]

There is a countable set of *proposition letters*, $p, q, r, p_1, p_2, \ldots$.

The *atomic formulas* of propositional logic are proposition letters, the symbol for truth $\top$, and the symbol for absurdity (false) $\bot$.

The set of *propositional formulas* is the smallest set P such that:

1. If ϕ is an atomic formula then $\phi \in P$.
2. If $\phi \in P$ then $\neg\phi \in P$.
3. If $\phi, \psi \in P$ then $\phi \wedge \psi$, $\phi \vee \psi$, and $\phi \rightarrow \psi$ are all in P.

Note the convention used in the book: small Roman letters are proposition letters, small Greek letters are formulas, large Greek letters are sets of Formulas.

In the Haskell file Prop.hs we have an approximation of this definition.

```
data Prop a =
     LetterP a
   | AndP (Prop a) (Prop a)
   | OrP (Prop a) (Prop a)
   | ImpliesP (Prop a) (Prop a)
   | NotP (Prop a)
   | AbsurdP
   | TruthP
```

Note that since Haskell allows infinite values, this datatype contains some values that are not propositional formulas. In particular, the circularly defined value below is not a proposition by the inductive definition given above:

```
andKnot = AndP andKnot andKnot
```

Our code, and most further discussion, will assume that all values of type Prop are finite. Inductive definitions, such as this function that computes the set of proposition letters occurring in a propositional formula, are assumed to be complete and safe (terminating):

```
letters:: Eq a => Prop a -> [a]
letters (LetterP a) = [a]
letters (AndP x y) = nub(letters x ++ letters y)
letters (OrP x y) = nub(letters x ++ letters y)
letters (ImpliesP x y) = nub(letters x ++ letters y)
letters (NotP x) = letters x
letters _ = []
```

The function nub removes duplicate occurrences of an element in a list. Note that the wildcard match in the last case matches only $\top$ or $\bot$.

1.2.3 Natural Deduction Proofs for Propositional Logic

The first proof system we will study is natural deduction. We will initially follow the presentation in Huth and Ryan.

Gentzen introduced natural deduction as a style of proof. In the first formulation of it that we will consider, a proof is a tree. The conclusion of the proof is the formula at the root of the tree. The tree can have leaves, which we call premises and (discharged) assumptions. The official reading of the tree is that the set of premises entails the conclusion.

For example, suppose we wanted to prove $p \wedge q$ entails $q \wedge p$. To do this we will need to construct a tree that has $p \wedge q$ on all of its leaves, and $q \wedge p$ at its root. The first rule for $\wedge$ we have is called $\wedge$ introduction:

$$\frac{\phi \quad \psi}{\phi \wedge \psi}$$

This says that we can combine a proof of ϕ and a proof of ψ to form a proof of $\phi \wedge \psi$. It is called an introduction rule because the connective is introduced by this step in the proof.

The next two rules for $\wedge$ allow us to take a conjunction apart, that is, to eliminate it. They are:

$$\frac{\phi \wedge \psi}{\phi} \qquad \frac{\phi \wedge \psi}{\psi}$$

Note that these rules very straightforwardly capture exactly the intuition we have about conjunction. To prove a conjunction we prove both parts independently. If we know that a conjunction holds, we know that each conjunct holds.

We can now use all three of these rules together for our proof:

$$\frac{\dfrac{p \wedge q}{q} \qquad \dfrac{p \wedge q}{p}}{q \wedge p}$$

Each step in this proof is an instance of one of the three rules. The two premises (leaves of the tree) are the identical propositional formula $p \wedge q$. The root of the tree is the propositional formula $q \wedge p$. Hence it is an official proof of the property we set out to prove.

While these proofs as trees are beautiful and intuitively compelling, they are a bit difficult to read and write. The text gives a more linear way to write these proofs. In that style a proof is a numbered list of formulas and justifications. Every justification names the rule justifying the formula and the line numbers of the antecedent formulas. This linear representation also allows some sharing.

In that style we write the proof:

1	$p \wedge q$	premise
2	q	$\wedge$ elimination 2, 1
3	p	$\wedge$ elimination 1, 1
4	$q \wedge p$	$\wedge$ introduction, 2 and 3

One of the most important features of natural deduction is its ability to express a hypothetical argument. For example, in rhetoric, to show an implication $\phi \to \psi$, it is natural to assume ϕ and then derive ψ as a consequence. Once you pass out of the hypothetical context you no longer are burdened with the assumption of ϕ, and you can freely use the conclusion that $\phi \to \psi$.

Gentzen captures this structure in a rule by indicating that certain arguments have assumptions that can be discharged once the argument is complete. By convention, assumptions to be discharged are written in brackets. The rule for $\to$ introduction, which follows the above intuition given above, is written:

$$\dfrac{\begin{array}{c}[\phi]\\ \vdots \\ \psi\end{array}}{\phi \to \psi}$$

With this rule, we can now turn our proof that from $p \wedge q$ we can prove $q \wedge p$ into a proof of the implication $p \wedge q \to q \wedge p$ with no premises. In pictures that appears as:

$$\dfrac{\dfrac{\dfrac{[p \wedge q]}{q} \quad \dfrac{[p \wedge q]}{p}}{q \wedge p}}{p \wedge q \to q \wedge p}$$

To linearize the hypothetical reasoning of natural deduction, Huth and Ryan draw a box around the hypothetical portion of the proof. I will not draw the boxes in these notes; see the text.

The elimination rule for implication is the familiar *modus ponens* rule. It is written:

$$\frac{\phi \quad \phi \to \psi}{\psi}$$

Rules for disjunction The introduction rules for disjunction (or, a.k.a $\vee$) are straightforward. If we know ϕ then we can conclude either $\phi \vee \psi$ or $\psi \vee \phi$:

$$\frac{\phi}{\phi \vee \psi} \qquad \frac{\psi}{\phi \vee \psi}$$

How do we eliminate a disjunction? I find it easiest to approach this rule from our experience in programming. Think of disjunction as corresponding to the Haskell type Either, which has constructors Left and Right:

```
data  Either a b  =  Left a | Right b
```

In this case the types of the constructors Left and Right correspond to the proof rules above.

In Haskell we eliminate an Either type with a case statement (or pattern matching function definition) such as:

```
orElim e f g =
   case e of
      Left x -> f x
      Right x -> g x
```

This has type:

```
orElim :: Either a b -> (a -> c) -> (b -> c) -> c
```

The or-elimination rule in Huth and Ryan corresponds exactly to this Haskell type. To prove χ by or-elimination on $\phi \vee \psi$ you must give (1) a direct proof of $\phi \vee \psi$, (2) a hypothetical proof of χ from ϕ, and (3) a hypothetical proof χ from ψ. This is written:

$$\frac{\phi \vee \psi \quad \begin{array}{c}[\phi]\\ \vdots \\ \chi\end{array} \quad \begin{array}{c}[\psi]\\ \vdots \\ \chi\end{array}}{\chi}$$

Aside: Propositions as types The correspondence between Haskell's Either type and disjunction mentioned above is an example of the principle of Propositions as types, which was identified by Howard[**?**]. All of the rules discussed to this point have a direct computational interpretation via propositions as types. The fragment introduced to this point is called *minimal logic*. See `http://en.wikipedia.org/wiki/Minimal_logic`. In preparing this lecture I came across a very interesting and detailed discussion of propositions as types, minimal logic, and classical logic by Finn Lawler: `http://www.scss.tcd.ie/publications/tech-reports/reports.08/TCD-CS-2008-55.pdf`. He also has a relatively accessible slide presentation on some of these points: `http://www.scss.tcd.ie/~flawler/talks/classical-slides.pdf`. Lawler appears to be a PhD student at Trinity College Dublin. These are not peer reviewed, but on a quick scan they looked very solid. The bibliography of the tech report is excellent.

Rules for negation, absurdity, and double negation One of the best known rules of classical logic allows us to conclude anything as a consequence of absurdity. In Latin: *ex falso quodlibet*. This rule is easily expressed in natural deduction:

$$\frac{\bot}{\phi}$$

In our naming convention this is $\bot$-elimination.

Another familiar concept from classical logic is that if we have shown a direct contradiction, that is both ϕ and $\neg\phi$, then we have proven absurdity. This is expressed:

$$\frac{\phi \quad \neg\phi}{\bot}$$

This rule is called $\neg$-elimination. (It also seems to function as $\bot$-introduction.)

The $\neg$-introduction rule is similar to a proof by contradiction. If from ϕ we can conclude absurdity, then we can conclude $\neg\phi$. It is written:

$$\frac{\begin{array}{c}[\phi]\\ \vdots \\ \bot\end{array}}{\neg\phi}$$

Note the similarity with implication introduction. In some settings negation is modeled as the implication of absurdity. Note that both the $\neg$ introduction and elimination rules above are derived from the rules of implication in that setting.

In classical logic double negations can be eliminated. That is if $\neg\neg\phi$ holds, then we can conclude that ϕ holds. This is expressed:

$$\frac{\neg\neg\phi}{\phi}$$

Some examples Work some examples on the board in both tree and linear style.

Sequents In the natural deduction proofs we show that the conclusion ϕ (root, or last line of an HR style proof) is a logical consequence of the premises $\psi_1, \psi_2, \ldots, \psi_k$. We write this formally as:

$$\psi_1, \psi_2, \ldots, \psi_k \vdash \phi$$

This is called a *sequent*. A sequent that can be proven is called *valid* (or *provable*).

HR's natural deduction system can be explicitly translated to sequents. The file `HR.hs` has a prover based on this translation. This will be discussed in Lecture 3.

In the treatment of natural deduction in the book all sequents are of this form. As you read elsewhere you may see sequents defined with sets of formulas on both sides, such as $\Gamma \vdash \Delta$. In the general form the interpretation is that the conjunction of the formulas in Γ entail the disjunction of the formulas in Δ. If Δ is the empty set then Γ entails absurdity.

In `Prop.hs` the type `Sequent` represents the type of sequents found in HR. The type `SequentM` represents the more general sequent type.

1.3 Tim's Prover and exercise

This course concerns the role of logic in computer science. It is natural then that we try and write computer programs that elucidate the ideas of logic as we cover them. This is the natural structure of the course as Jim and I have imagined it.

In the file `http://web.cecs.pdx.edu/~hook/logicw11/Exercises/Ex1ForwardProof.html` we have designed an exercise that should help solidify the ideas of Natural deduction. We encode the notion of a proposition as a Haskell datatype exactly as we have introduced in these notes above, and have constructed a proof checker for forward style natural deduction proofs. The exercise allows you to explore some existing proofs, and to allow you to build several proofs of your own.

2 Propositional logic: truth, validity, satisfiability, tautology, and soundness

Last time: Introduced the syntax of propositional logic. Presented natural deduction proofs for propositional logic.

Exercise: Discuss experience doing the exercise using the programs provided.

Today: Semantics of propositional logic. Soundness of proof systems.

2.1 Semantics

2.1.1 Valuation

To give propositional logic a semantics, we will map propositional formulas to the set of truth values, T (True) and F (False). To interpret a formula we will need a truth assignment for our atomic formulas (called a *valuation* or *model*). (See HR Definition 1.28) We will extend this to all formulas by interpreting the connectives as truth valued functions according to their truth tables. Figure 1.5 in the text is an example of a truth table.

The valuation of a formula ϕ in an assignment v, written ϕ^v, is defined inductively as follows:

$$\begin{array}{lcl}
(\phi \wedge \psi)^v & = & \phi^v \& \psi^v \\
(\phi \vee \psi)^v & = & \phi^v | \psi^v \\
(\phi \rightarrow \psi)^v & = & \phi^v \Rightarrow \psi^v \\
(\neg \phi)^v & = & \overline{\phi^v} \\
p^v & = & v(p) \\
\bot^v & = & F \\
\top^v & = & T
\end{array}$$

Where &, |, and $\Rightarrow$ are the standard Boolean functions corresponding to the truth tables in the text. In `Assignment1.hs` you will write a similar evaluator in Haskell.

So given any particular truth assignment (valuation, model) we can calculate the truth of a propositional formula.

Object Language and Meta Language The *object language* is the language being studied or defined. The *meta language* is the language we are using to define or study the object language.

In this discussion $\wedge, \vee, \rightarrow$, and $\neg$, are symbols in the object language, while $\&, |, \Rightarrow$, and $\bar{\ }$ are in the meta language, which in this case is ordinary mathematics or set theory.

2.1.2 Logical equivalence

Given this definition of meaning, we can now explore how the meaning of two formulas are related. For example, we can ask how $p \rightarrow q$ relates to $\neg p \vee q$. Two formulas that are given the same meaning in all truth assignments are *logically equivalent.* That is, if for all v, $\phi^v = \psi^v$ then ϕ and ψ are logically equivalent.

2.1.3 Tautology

Some formulas are true under all assignments. These formulas are very important. They are called *tautologies.* A *tautology* is a propositional formula that is true in all truth assignments.

In Assignment 1 you will write a tautology checker for propositional logic. It is very significant that being a propositional tautology is decidable.

2.1.4 Satisfiable

A set S of formulas is *satisfiable* if some valuation maps all elements of S to true. Such a valuation is called a *satisfying assignment.*

We will use satisfiability in several ways. Sometimes, we will code up problems as sets of formulas where a satisfying assignment is viewed as a solution to the problem.

Another important use of satisfiability relates to tautologies.

Fact 2.1 *ϕ is a tautology if and only if $\{\neg\phi\}$ is not satisfiable.*

A whole family of automated theorem provers, called refutation provers, use this fact as the basis of a proof strategy. To prove ϕ a refutation prover will try to find a satisfying assignment for $\neg\phi$. If it finds an assignment, then that will be a counter example to ϕ being true. If it can conclude that no assignment exists it can conclude that ϕ is true in all assignments. We will study a refutation prover in detail in Section 3.1.

2.2 Soundness

Soundness of a logic says that every provable formula is true. In our particular case, we need to show that every provable sequent $\psi_1, \psi_2, \ldots, \psi_k \vdash \phi$ is true. We formalize this by saying that every valuation that makes the premises true must make the consequent true. This relation is called *semantic entailment*. It is represented by the symbol $\models$. The semantic entailment corresponding to the sequent above is:

$$\psi_1, \psi_2, \ldots, \psi_k \models \phi$$

The test of entailment can be programmed in the style of Assignment 1.

With this definition, we are ready to state the Soundness theorem:

Theorem 2.2 (Soundness, HR 1.35) *Let $\phi_1, \phi_2, \ldots, \phi_n$, and ψ be propositional formulas. If $\phi_1, \phi_2, \ldots, \phi_n \vdash \psi$ is valid then $\phi_1, \phi_2, \ldots, \phi_n \models \psi$ holds.*

The proof of soundness is by induction on the structure of proofs. HR includes a nice discussion of how this can be viewed as a course of values induction on the length of the linearized proofs.

Here I will do an alternative proof on the translation of their proof system into sequent form. The key difference is that in explicit sequent form the implies introduction rule is written:

$$\frac{\Gamma, \phi \vdash \psi}{\Gamma \vdash \phi \rightarrow \psi}$$

This captures the management of the scoping of hypothetical assumptions in the sequent. It is not necessary to have boxes to provide scoping. It is also not necessary to transform the proof when unwrapping a box, as they do in the text.

Figure 1 presents the sequentized rules from the file `HR.hs`. These rules are Haskell functions that implement a rule a function from a consequent to a list of antecedents. The rules use the Haskell Maybe type to manage failure. The code fragment corresponding to the implies introduction rule is:

```
impliesI :: Rule a
impliesI (Sequent gamma (ImpliesP p q)) = Just [Sequent (p:gamma) q]
impliesI _                              = Nothing
```

Comparing this to the rule, you see that `gamma` is Γ, `p` is ϕ, `q` is ψ, and the type constructor `Sequent` is $\vdash$. Note that in a complete proof there will be no failed rule applications, so the Just and Nothing constructors can be disregarded.

Proof: The proof of soundness is by structural induction on the proof tree. It proceeds by cases on the last rule used.

Case hyp: The basis is the rule `hyp i`. It is the only rule that does not introduce any subgoals; it will be at every leaf of the proof tree. We must show that if the proof consists of exactly one instance of this rule that validity implies entailment. That is, if the proof is exactly:

$$\frac{}{\psi_1, \ldots, \psi_i, \ldots, \psi_n \vdash \psi_i}$$

```
type Rule a = Sequent a -> Maybe [Sequent a]

andI :: Rule a
andI (Sequent gamma (AndP p q)) = Just [Sequent gamma  p,
                                        Sequent gamma  q]
andI _ = Nothing

andE1 :: Prop a -> Rule a
andE1 psi (Sequent gamma  phi) = Just [Sequent gamma (AndP phi psi)]

andE2 :: Prop a -> Rule a
andE2 psi (Sequent gamma  phi) = Just [Sequent gamma (AndP psi phi)]

orI1 :: Rule a
orI1 (Sequent gamma (OrP p _)) = Just [Sequent gamma p]
orI1 _ = Nothing

orI2 :: Rule a
orI2 (Sequent gamma (OrP _ q)) = Just [Sequent gamma q]
orI2 _ = Nothing

orE :: Prop a -> Rule a
orE r@(OrP p q) (Sequent gamma x) = Just [Sequent gamma r,
                                          Sequent (p:gamma) x,
                                          Sequent (q:gamma) x]
orE _ _ = Nothing

impliesI :: Rule a
impliesI (Sequent gamma (ImpliesP p q)) = Just [Sequent (p:gamma) q]
impliesI _                              = Nothing

impliesE :: Prop a -> Rule a
impliesE psi (Sequent gamma phi) = return [Sequent gamma (ImpliesP psi phi),
                                           Sequent gamma  psi]

notI :: Rule a
notI (Sequent gamma (NotP p)) = Just [Sequent (p:gamma) AbsurdP]
notI _ = Nothing

notE :: Prop a -> Rule a
notE psi (Sequent gamma AbsurdP) = Just [Sequent gamma  psi,
                                         Sequent gamma  (NotP psi)]
notE _ _ = Nothing

notNotE :: Rule a
notNotE (Sequent gamma p) = Just [(Sequent gamma  (NotP (NotP p)))]
```

Figure 1: Rules from HR.hs (part 1)

```
absurdE :: Rule a
absurdE (Sequent gamma phi) = Just [Sequent gamma  AbsurdP]

truthI :: Rule a
truthI (Sequent gamma TruthP) = Just []
truthI _ = Nothing

hyp :: (Eq a) => Int -> Rule a
hyp n (Sequent gamma phi) = if phi == gamma !! (n-1)
                            then Just []
                            else Nothing
```

Figure 2: Rules from HR.hs (part 2)

Then:

$$\frac{}{\psi_1, \ldots, \psi_i, \ldots, \psi_n \models \psi_i}$$

But since the definition of $\models$ requires that the conjunction of the premises implies the consequent, and the consequent is one of the premises, this must always hold.

The proof proceeds inductively considering all other possible last rules. We will only include a few of the cases here.

Case andI: If the last rule is and introduction, then the proof ends with

$$\frac{\Gamma \vdash \phi \quad \Gamma \vdash \psi}{\Gamma \vdash \phi \wedge \psi}$$

By induction we know $\Gamma \models \phi$ and $\Gamma \models \psi$. By the definition of truth in a valuation, we know that any assignment v that makes $\phi^v = \psi^v = T$ also makes $(\phi \wedge \psi)^v = T$ true. Hence $\Gamma \models \phi \wedge \psi$, as required.

Case impliesI: If the last rule is implies introduction, then the proof ends with

$$\frac{\Gamma, \phi \vdash \psi}{\Gamma \vdash \phi \rightarrow \psi}$$

By induction we know that $\Gamma, \phi \models \psi$. From the definition of $\models$ and properties of conjunction we know that since this holds, any valuation that makes Γ true must either make ϕ false or ψ true. But these are exactly the conditions to make $\phi \rightarrow \psi$ true. Hence $\Gamma \models \phi \rightarrow \psi$, as required.

Case notI: If the last rule is not introduction then the proof ends:

$$\frac{\Gamma, \phi \vdash \bot}{\Gamma \vdash \neg\phi}$$

By induction, $\Gamma, \phi \models \bot$. Since no valuation makes $\bot$ true, we know that the conjunction of Γ, ϕ is false. Thus we can conclude that every valuation that makes Γ true makes ϕ false. By definition of $(\neg\phi)^v$ this establishes $\Gamma \models \neg\phi$, as required.

The remaining cases are omitted, but you are encouraged to consider each rule in turn and convince yourself that it preserves validity.

2.3 Exercise 2: Backward prover

In exercise 2 you are asked to work with the prover in `HR.hs` to do "backward style" proofs. These are proofs where you start from a goal, stated as a sequent, and you grow a tree of subgoals by applying rules.

Normally backward provers are done in languages with side effects and your proof state evolves as you apply rules. Our first backward prover is purely functional, so it starts from the beginning every time.

This backward prover has a tactic facility modeled after LCF. It does not include LCF's validation functions. It is not as high-assurance as LCF. Feel free to modify the prover to have a more sophisticated notion of rule.

Exercise 2 is located here: `http://web.cecs.pdx.edu/~hook/logicw11/Exercises/Ex2BackwardProof.html`.

3 Tableau Proof

Last time: Semantics of Propositional logic. Definitions of tautology, satisfiability, semantic entailment. Proof of soundness of a proof theory for Propositional logic.

Exercise: Discuss!

Today: Introduce tableau proof. Say a few words about Gentzen's proof systems.

3.1 Signed Tableau prover

Smullyan writes about this origins of Analytic Tableaux:

> We now describe an extremely elegant and efficient proof procedure for propositional logic which we will subsequently extend to first order logic, and which shall be basic to our entire study. This method, which we term *analytic tableaux,* is a variant of the "semantic tableaux" of Beth[?], or of methods of Hintikka [?]. (Cf. also Anderson and Belnap [?].) Our present formulation is virtually that which we introduced in [?]. Ultimately, the whole idea derives from Gentzen [?], and we shall subsequently study the relation of analytic tableaux to the original methods of Gentzen.

These notes draw heavily on:

1. Nerode and Shore's *Logic for Applications.*
2. Fitting's text, *First-Order Logic and Automated Theorem Proving.*

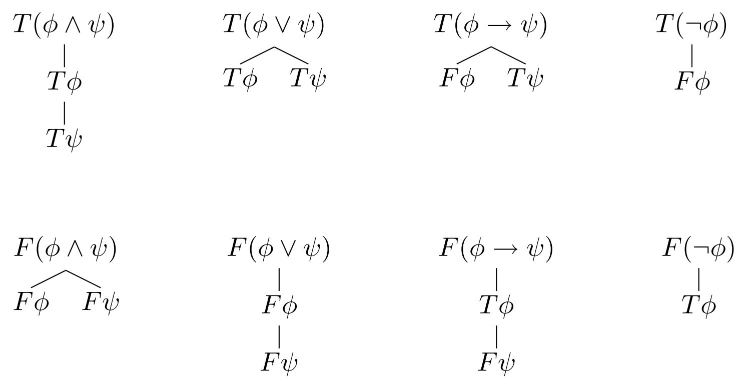


Figure 3: Signed Tableau rules

3. Smullyen's book *First-Order Logic*, 1968. Reprinted by Dover in 1995 (available on Amazon for about \$10.).

The basic idea is that a proof will be a tree of signed formulas. The sign is a Boolean, indicating of the formula is intended to be True or False. Each branch of the tree is an instance of one of the atomic tableaux, listed in Figure 3.

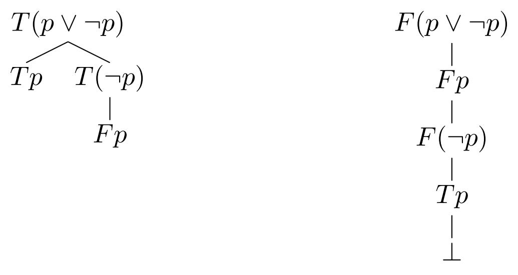
The branches of the path explore all possible truth assignments that might make the root satisfiable. This is an example of a refutation-based method. The root is labeled with the negation of the formula we wish to proof. If we systematically fail to find a satisfying assignment then we conclude that the formula is a tautology by Fact 2.1.

A branch (path) in the tree is *contradictory* or *closed* if both $T\phi$ and $F\phi$ appear on it. It is atomically closed if ϕ is an atomic formula.

A tableau is *contradictory* or *closed* if every path is closed.

A *tableau proof* for ϕ is a closed tableau with root $F\phi$.

We begin with two examples: a tableau rooted with $T(p \vee \neg p)$ and one rooted with $F(p \vee \neg p)$.



In the example on the left we have two open (not closed) paths. All formulas on both paths are satisfied by the assignment generated by the signed atomic formulas. In this case the set $\{Tp, T(p \vee \neg p)\}$ is satisfied by making p true, while the set $\{Fp, T(\neg p), T(p \vee \neg p)\}$ is satisfied by making p false.

In the example on the right we have an atomic contradiction on the only path, which is labeled both Tp and Fp. The fact that the path is closed is shown here with a terminal $\perp$.

The example on the right is a tableau proof of the tautology $p \vee \neg p$.

3.1.1 A systematic example

Next we try a larger example: Prove $((p \to q) \to p) \to p$.

1. Write the negation of the formula as the root of the tree:

$$F(((p \to q) \to p) \to p)$$

2. Find the atomic tableau in Figure 3 that matches making an implication false, and extend the tree according to the atomic tableau:

$$\begin{tikzcd} F(((p \to q) \to p) \to p) \arrow[d, dash] \\ T((p \to q) \to p) \arrow[d, dash] \\ Fp \end{tikzcd}$$

3. The next formula to be reduced is $T((p \to q) \to p)$. We find the appropriate atomic tableau, and instantiate it:

$$\begin{tikzcd} & T((p \to q) \to p) \arrow[dl, dash] \arrow[dr, dash] & \\ F(p \to q) & & Tp \end{tikzcd}$$

We extend the path on which the formula occurs with the children, yielding:

$$\begin{tikzcd} & F(((p \to q) \to p) \to p) \arrow[d, dash] & \\ & T((p \to q) \to p) \arrow[d, dash] & \\ & Fp \arrow[dl, dash] \arrow[dr, dash] & \\ F(p \to q) & & Tp \end{tikzcd}$$

Note that we did not copy the term we reduced.

In a more complex example, the term being reduced might occur on another non-contradictory path. We can make independent choices about the order of terms that we reduce on a path.

4. The rightmost branch is now atomically closed, so we mark it with $\bot$. The leftmost branch has an unreduced non-atomic formula, so again we instantiate the appropriate rule, yielding:

$$\begin{tikzcd}
 & F(((p \to q) \to p) \to p) \arrow[d, dash] & \\
 & T((p \to q) \to p) \arrow[d, dash] & \\
 & Fp \arrow[dl, dash] \arrow[dr, dash] & \\
F(p \to q) \arrow[d, dash] & & Tp \arrow[d, dash] \\
Tp \arrow[d, dash] & & \bot \\
Fq & &
\end{tikzcd}$$

5. This produces an atomic contradiction on the leftmost path (p is both true and false). We mark that path closed, completing the proof:

$$\begin{tikzcd}
 & F(((p \to q) \to p) \to p) \arrow[d, dash] & \\
 & T((p \to q) \to p) \arrow[d, dash] & \\
 & Fp \arrow[dl, dash] \arrow[dr, dash] & \\
F(p \to q) \arrow[d, dash] & & Tp \arrow[d, dash] \\
Tp \arrow[d, dash] & & \bot \\
Fq \arrow[d, dash] & & \\
\bot & &
\end{tikzcd}$$

3.1.2 Soundness

Why does this work? The key invariant is that we preserve satisfiability at every step of the process. Each step of the construction of the tree extends a path Θ to a set of paths S. The path Θ is satisfiable iff there is a path in S that is satisfiable. Closed paths are not satisfiable. If we can extend the root of the tree to a set of closed paths the root is not satisfiable.

What do we need to do to sharpen this argument?

1. Extend the definition of satisfiable to sets of signed formulas.

2. Clearly articulate the rules for constructing a tableau.

3. Prove the invariant identified above.
4. Conclude soundness

Satisfiable signed formulas A set of signed formulas Θ is satisfied by a valuation v if for every sign formula pair s, ϕ the valuation maps ϕ to s.

Formal definition of Tableau Tableaux are constructed as follows:

1. A signed formula is a tableau.
2. A tableau T may be *extended* to a tableau T' by selecting a signed term t on a path Θ, instantiating an atomic tableau with head term t, and extending Θ with the descendents of the instantiated atomic tableau.

A branch Θ is *closed* if both $T\phi$ and $F\phi$ occur on Θ, or if either $T\bot$ or $F\top$ occur on Θ.

A branch Θ is *atomically closed* if both Tp and Fp, for proposition letter p, occur on Θ, or if either $T\bot$ or $F\top$ occur on Θ.

A *tableau proof* of ϕ is a closed tableau with root $F\phi$.

A tableau is *strict* if no formula is used by the same rule twice on the same branch.

Key invariant

Claim 3.1 *Let T be a tableau, let T' be the tableau obtained from T by reducing a signed formula t on path Θ. Let τ be the set of paths in T' that extend Θ. A validation v satisfies Θ iff it satisfies some $\Theta' \in \tau$.*

Proof: By cases on the atomic tableau used to extend Θ.

Case $t = T(\phi \wedge \psi)$: In this case $\tau = \{\Theta'\}$ and $\Theta' = \Theta, T\phi, T\psi$. Let v be a valuation satisfying Θ. Then v makes $\phi \wedge \psi$ true. Hence v makes ϕ and ψ true. Hence v satisfies Θ'. Conversely, if v satisfies Θ' it will also satisfy Θ.

Case $t = F(\phi \wedge \psi)$: In this case $\tau = \{\Theta_1, \Theta_2\}$ where $\Theta_1 = \Theta, F\phi$ and $\Theta_2 = \Theta, T\phi$. Let v satisfy Θ. Then v makes $\phi \wedge \psi$ false. Hence v makes either ϕ or ψ false. Hence v satisfies either Θ_1 or Θ_2. Conversely, if v satisfies Θ_1 or Θ_2 then v satisfies Θ, as Θ is a component of both Θ_1 and Θ_2.

The other cases are similar.

Claim 3.2 *If T is a closed tableau, the root is not satisfiable.*

Proof: By Claim 3.1, every extension of a tableau exactly preserves satisfiability. Thus, if T is a closed tableau, its root is equisatisfiable with the empty set. That is, it is not satisfiable.

Soundness

Theorem 3.3 *If ϕ is tableau provable then ϕ is a tautology.*

Proof: If ϕ is tableau provable then there is a closed tableau with root $F\phi$. By Claim 3.2 $\neg\phi$ is not satisfiable. By Fact 2.1 ϕ is a tautology.

3.1.3 A Simple Implementation

The goal of this section is to write a propositional tautology checker based on tableau proofs. To do this we will:

1. Observe that we can describe a systematic algorithm for constructing tableaux.
2. Note that the systematic algorithm terminates.
3. Identify what to represent in an implementation.
4. Look at a simple, direct implementation of this technique in Haskell (Figure 4).

Systematic tableau In the definition of a tableau we tried to give as much freedom as possible in the selection of formulas to be reduced. Did we give enough freedom to fail to terminate? Yes! We can do the same reduction repeatedly.

When we were defining terms above, we included the definition of a strict tableau. That is one where we never redo work.

Convince yourself that if there is a tableau proof of ϕ then there is a strict tableau proof of ϕ.

If we restrict ourselves to the consideration of strict tableaux can we fail to terminate? I don't think so. Prove it. It may be helpful to recall König's Lemma: A tree that is finitely branching but infinite must have an infinite branch.

There is an algorithm called the complete systematic tableau (CST) that essentially does a breadth first search of the tree for a node to reduce, and then reduces that node. That algorithm is manifestly terminating.

Representation When we write these proofs on the board, the tree structure is very natural and suggestive. However, the most important concept in the algorithm is the set of paths, and the most important elements on the paths are those signed formulas that have not been reduced.

With implementation in mind, review the proof of Claim 3.1. Note that we could have proved a stronger claim. We could have removed the node being reduced from the new paths created! That is, in the case $t = T(\phi \wedge \psi)$, if $\Theta = t, \Sigma$, we could have made $\Theta' = \Sigma, T\phi, T\psi$. While this makes the trees harder to draw on the board, it still satisfies the invariant. It also makes it easier to write the algorithm, since if we can discard the formulas we reduce then we won't accidentally use them again.

The algorithm in Figure 4 represents a tableau as a list of paths. Each path is a list of signed formulas. The function `tableauExtend` maps a path to a set of paths. The fragment dealing with $\wedge$ is copied below:

```haskell
module SimpleTableau where
import Prop

type Path a = [(Bool, Prop a)]

tableauExtend :: Path a -> [Path a]
tableauExtend ((True, AndP a b):ts)      = [ts ++ [(True, a),(True, b)]]
tableauExtend ((False, AndP a b):ts)     = [ts ++ [(False, a)], ts ++ [(False,b)]]
tableauExtend ((True, OrP a b):ts)       = [ts ++ [(True,  a)], ts ++ [(True, b)]]
tableauExtend ((False, OrP a b):ts)      = [ts ++ [(False, a),(False, b)]]
tableauExtend ((True, ImpliesP a b):ts)  = [ts ++ [(False, a)], ts ++ [(True, b)]]
tableauExtend ((False, ImpliesP a b):ts) = [ts ++ [(True, a),(False, b)]]
tableauExtend ((sign, NotP a):ts)        = [ts ++ [(not sign, a)]]

tableauCheck :: (Eq a) => Path a -> Bool
tableauCheck ts = check [] [] ts where
   check pos neg []                 = True
   check pos neg ((True,AbsurdP):_) = False
   check pos neg ((False,TruthP):_) = False
   check pos neg ((True,p):ts)      = if elem p neg then False
                                                else check (p:pos) neg ts
   check pos neg ((False,p):ts)     = if elem p pos then False
                                                else check pos (p:neg) ts

firstReducible :: Path a -> Maybe (Path a)
firstReducible ts = reducible [] ts where
  reducible revAtoms []                        = Nothing
  reducible revAtoms (t@(sign, LetterP x):ts) = reducible (t:revAtoms) ts
  reducible revAtoms (t@(sign, TruthP):ts)    = reducible (t:revAtoms) ts
  reducible revAtoms (t@(sign, AbsurdP):ts)   = reducible (t:revAtoms) ts
  reducible revAtoms ts@((sign, p):_)         = Just (ts ++ (reverse revAtoms))

proveOne :: (Eq a) => Path a -> Maybe [Path a]
proveOne ts =
    do ts' <- firstReducible ts
       return (filter tableauCheck (tableauExtend ts'))

data Result a = Succeed | Fail a deriving Show

proveIt :: (Eq a) => [Path a] -> Result [Path a]
proveIt tab =
  case fmap concat (mapM proveOne tab) of
         Just [] -> Succeed
         Just ts' -> proveIt ts'
         Nothing -> Fail tab

prove :: (Eq a) => Prop a -> Result [Path a]
prove phi = proveIt (filter tableauCheck [[(False,phi)]])
```

Figure 4: A simple propositional tableau prover in Haskell (comments omitted to fit page)

```
tableauExtend :: Path a -> [Path a]
tableauExtend ((True, AndP a b):ts)  = [ts ++ [(True, a),(True, b)]]
tableauExtend ((False, AndP a b):ts) = [ts ++ [(False, a)], ts ++ [(False,b)]]
```

Note that this corresponds exactly to the optimization above. Haskell variable `ts` corresponds to Σ.

Exercise: Prove that `tableauExtend` preserves satisfiability. That is prove the analog of Claim 3.1 for this Haskell function.

The extension function `tableauExtend` assumes it is given a path with a reducible head. The function `firstReducible` rotates a path until it finds a reducible element. If it does not find a reducible element that path is *finished.* The function returns a Maybe type; if the path it is testing is finished, it signals this with the value Nothing. If the path is not finished it applies the Just constructor to the rotated path. If a finished path is not closed then it encodes a satisfying assignment for the tableau root.

The function `tableauCheck` tests a path to see if it is open. It returns true on open paths.

Exercise: Prove that if `ts` is closed that `tableauCheck ts` is false.

Exercise: Prove that `filter tableauCheck ts` is equisatisfiable with `ts`.

The function `proveOne` starts putting things together. It takes a path, which is assumed to be open. It returns either Just of a list of open paths obtained by extending the first reducible formula on the path or it returns Nothing if the path is finished.

The function `proveIt` takes a tableau with no closed paths. It operates on every path in the tableau, replacing it with the set of open paths to which it reduces. If it reduces to an empty tableau it halts with success. If it finds a finished tableau it halts with failure. Otherwise it iterates on the newly reduced tableau.

Finally, the function `prove` takes a proposition and builds a tableau proof. It indicates success if the proposition is a tautology, otherwise it returns a finished path.

3.1.4 Completeness

A proof system is complete if everything that is true is provable.

For the tableau prover, completeness follows from the same invariant that was used to establish soundness. The set of open paths in a tableau proof is always equisatisfiable with the root. Given that, plus the termination of the algorithm implemented above, we have that every tautology is tableau provable.

4 Prop logic: completeness, SAT solvers

Last time: Tableau proof technique.

Exercise: Discuss!

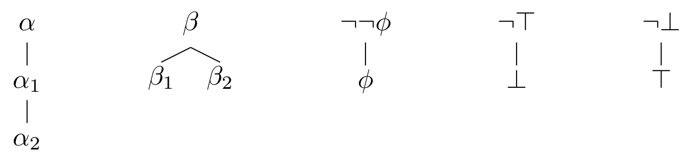


Figure 5: Tableau rules using uniform notation.

Today: Normal forms. Techniques for completeness arguments.

4.1 Propositional Tableau Proofs, Continued

4.1.1 Uniform notation

After introducing the signed tableaux presented here, Smullyan notes that certain patterns repeat. To simplify the proofs and make the uniformity more explicit, he introduces *uniform notation.* The primary differences are:

1. Top level negation is used instead of the F sign to indicate that a formula should be false.

2. Rules for binary connectives (and their negations) are unified into the two rules in Figure 5, capturing the two shapes of the signed rules.

The patterns α and β are defined as follows:

α	α_1	α_2	β	β_1	β_2
$\phi \wedge \psi$	ϕ	ψ	$\neg(\phi \wedge \psi)$	$\neg\phi$	$\neg\psi$
$\neg(\phi \vee \psi)$	$\neg\phi$	$\neg\psi$	$\phi \vee \psi$	ϕ	ψ
$\neg(\phi \to \psi)$	ϕ	$\neg\psi$	$\phi \to \psi$	$\neg\phi$	ψ

Following this convention, the set of atomic tableaux can be reduced to those in Figure 5.

Uniform notation makes the math less tedious in the proofs, but I find signed tableaux pedagogically easier to present. I can hear Nerode enthusiastically shouting "This is so easy we teach it to high school students!" Both versions appear in Smullyen. Although Fitting includes both, he focuses primarily on the uniform presentation.

4.1.2 Improving the Prover

There are several opportunities for improving the prover.

The test for closed paths may be expensive. If we changed the test for closure to atomic closure is the algorithm still correct? What will this trade off in the implementation?

The list representation of paths is suggestive of the pencil and paper method, but what other alternatives are there? If we adopt atomic closure then we could keep atoms in a separate structure. We can either use a map of proposition letters to their assignment, or two sets, one of true letters and one of false.

Our algorithm did not use any strategy in selecting the formula to reduce on a path. What strategies might it employ? Could it postpone duplicating path information by preferring α pattern formulas to β pattern formulas (in the sense of uniform notation, Figure 5)? (For example, experiment with proofs of $(a \to b \to c) \to (a \to b) \to a \to c$. How big can you make this without doing useless work? How small?) Should it have a preference for big or small formulas?

4.2 Gentzen L-style prover

Gentzen formulated several different proof theories. They are systematically named. The first letter is either N for the natural deduction style, or L for the sequent style. The second letter identifies the logical system encoded: M for minimal logic, J for intuitionistic logic, and K for classical logic.

The system presented in the text is NK. The fragment of NK I described as minimal logic in Section 1.2.3 is called NM.

The prover developed in HR.hs forces this NK formulation into a sequent style.

You may have noticed when doing the backwards proofs that you had to frequently supply terms to guide the rules. This is very different from the entirely “turn the crank” style of the tableau prover. This is in part because the tableau prover has the subterm property. Every reduction in the tableau introduces proper subterms of the term reduced. Gentzen’s L systems have a core of logical rules that have this same subterm property.

In the file L.hs I put together a simple prover for the system LK. I did not strictly follow every detail of the structural rules. I was looking at a presentation of LK by Kreiz and Constable, Lecture 9 of the course. These course notes are available at `http://www.cs.cornell.edu/courses/cs4860/2009sp/lec-09.pdf`.

I have since noted several minor discrepancies and one major difference between this presentation and other presentations. I addressed the major issue by adding the cut rule. I invite students to explore other differences that they may find and to improve the code base. Lawler’s tech report, cited above, includes LK in Figure 2.3.1 on page 27.

Rules acting on the left and right In LK, a typical sequent is $\Gamma \vdash \Delta$, where Γ and Δ are sets of formulas. The intended interpretation is that the conjunction of the formulas in Γ entail the disjunction of the formulas in Δ. If Δ is empty it is equivalent to absurd. If Γ is empty it is equivalent to truth.

Instead of being organized around introduction and elimination rules, rules in LK apply on the left and right side of the turn style. For example, the rules for $\wedge$ and $\rightarrow$ are:

$$\frac{\Gamma, \phi, \psi \vdash \Delta}{\Gamma, \phi \wedge \psi \vdash \Delta} \qquad \frac{\Gamma \vdash \phi, \Delta \quad \Gamma \vdash \psi, \Delta}{\Gamma \vdash \phi \wedge \psi, \Delta}$$

$$\frac{\Gamma \vdash \phi, \Delta \quad \Gamma, \psi \vdash \Delta}{\Gamma, \phi \rightarrow \psi \vdash \Delta} \qquad \frac{\Gamma, \phi \vdash \psi, \Delta}{\Gamma \vdash \phi \rightarrow \psi, \Delta}$$

In addition there are structural and logical rules. I am eliding those details. I am including the generalization of the axiom rule:

$$\frac{}{\Gamma, \phi \vdash \phi, \Delta}$$

(If you see other sources (such as Lawler) you may see two rules for $\wedge$ on the left. One is called multiplicative, the other additive.)

Here is a partial transcript of a proof involving implication:

```
*GentzenEgs> goal
|- (a => b => c) => (a => b) => a => c
*GentzenEgs> prove goal []
Remaining Subgoals:
   |- (a => b => c) => (a => b) => a => c
*GentzenEgs> prove goal [iR]
Remaining Subgoals:
   a => b => c |- (a => b) => a => c
*GentzenEgs> prove goal [iR,iR]
Remaining Subgoals:
   a => b, a => b => c |- a => c
*GentzenEgs> prove goal [iR,iR,iR]
Remaining Subgoals:
   a, a => b, a => b => c |- c
```

At this point we have exhausted our attack on the right hand side, and must shift to the left. The rule iL operates on the first arrow on the left that it finds. In the source you will see that a precise position can be specified in an alternate rule interface.

```
*GentzenEgs> prove goal [iR,iR,iR,iL]
Remaining Subgoals:
   a, a => b => c |- a, c,
   a, b, a => b => c |- c
*GentzenEgs> prove goal [iR,iR,iR,iL,ax]
Remaining Subgoals:
   a, b, a => b => c |- c
*GentzenEgs> prove goal [iR,iR,iR,iL,ax,iL]
Remaining Subgoals:
   a, b |- a, c,
   a, b, b => c |- c
```

```
*GentzenEgs> prove goal [iR,iR,iR,iL,ax,iL,ax]
Remaining Subgoals:
   a, b, b => c |- c
*GentzenEgs> prove goal [iR,iR,iR,iL,ax,iL,ax,iL]
Remaining Subgoals:
   a, b |- b, c,
   a, b, c |- c
*GentzenEgs> prove goal [iR,iR,iR,iL,ax,iL,ax,iL,ax]
Remaining Subgoals:
   a, b, c |- c
*GentzenEgs> prove goal [iR,iR,iR,iL,ax,iL,ax,iL,ax, ax]
QED
```

As you can see this is less tedious than the NK system, but still highly repetitive. You may want to experiment with the tactic mechanism. Here is a very simple mechanization of this proof:

```
*GentzenEgs> prove goal [repeatT iR,repeatT (iL `thenAll` (try ax))]
QED
```

- Do you see similarities between the tableau rules and the LK rules?
- Can you develop a complete automatic prover for LK?

4.3 Normal Forms

1. Review Tableau
2. Dual Clause form (disjunctive normal form)
3. Clausal form (conjunctive normal form)

How do we read a tableau proof? What do the paths mean? What does it mean if we calculate a finished path that is not closed?

As we saw when proving soundness, as we build a tableau we are trying to find a valuation that satisfies the whole path. If we do find such a valuation, it will make the conjunction of all formulas on the path true, including, of course, the root.

If we look at all the paths in a tableau, we can think of them as different alternative ways to make the root true.

In this way, we can view the tableau method as analyzing the root by formulating it as a disjunction of conjunctions of formulas. Furthermore, as we noted in discussing the implementation, we can throw out formulas as we reduce them and still have logically equivalent paths. In this way, we can see a tableau implementation as an algorithm to build an equivalent formula that is the disjunction of conjunctions of signed proposition letters. This is the essence of the definition of *disjunctive normal form (DNF)*, sometimes called *dual clause form.*

To articulate the definitions of the normal forms, we first introduce another standard definition. A *literal* is either a proposition letter or the negation of a proposition letter. We can convert from a set of signed proposition letters to literals by mapping Tp to p and Fp to $\neg p$.

A formula is in *disjunctive normal form* if it is the disjunction of conjunctions of literals.

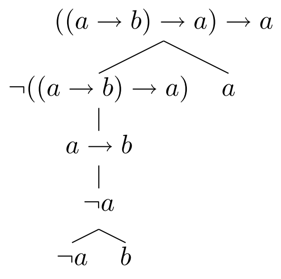
A formula is in *conjunctive normal form (CNF)* if it is the conjunction of disjunctions of literals. CNF is sometimes called *clausal form.*

The tableau proof method can be seen as an algorithmic conversion of the root to disjunctive normal form.

How hard is it to test a formula in DNF for satisfiability?

An Example Consider the formula $((a \to b) \to a) \to a$. (This formula is the tautology known as Pierce's law.)

We can convert it to disjunctive normal form by applying the uniform notation version of the tableau proof rules, without stopping on closed paths. Specifically we get:



From which we can read the following DNF formula:

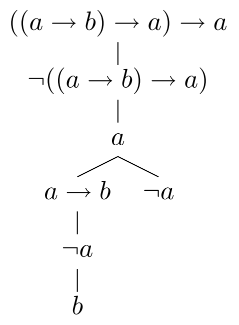
$$(\neg a \wedge \neg a) \vee (\neg a \wedge b) \vee a$$

Since the connectives are dictated by the normal form, formulas in DNF or CNF are often written as lists of lists of literals. Some authors use angle brackets for conjunctions and square braces for disjunction. In that convention this would be:

$$[\langle \neg a, \neg a \rangle, \langle \neg a, b \rangle, \langle a \rangle]$$

In Haskell, we generally just use lists of lists of propositions.

We can calculate the dual normal form, conjunctive normal form, by dualizing the rules. If we write this as a tree we get:



Note: This tree is not a tableau! It is dual!

This is equivalent to the CNF:

$$\langle [a, \neg a, b], [a, \neg a] \rangle$$

Note how easy it is to recognize that this formula is tautologically true when presented in this form!

Conjunctive normal form is also called clausal form. In that terminology the disjunctions are called *clauses*. That is, a clause is a list (or set) of literals. A formula in clausal form is a list (or set) of clauses.

Note: Huth and Ryan develop an algorithm for conversion to CNF that does not refer to uniform notation.

Many algorithms assume propositional formulas are given in DNF or CNF.

When using the list notations for disjuncts and conjuncts, how do we interpret the empty list? The algebraic properties we want are:

$$\begin{array}{rcl} \langle l_1 \rangle \wedge \langle l_2 \rangle & = & \langle l_1 + + l_2 \rangle \\ [l_1] \vee [l_2] & = & [l_1 + + l_2] \end{array}$$

These laws suggest that $[] = \bot$ and $\langle\rangle = \top$. In clausal form the empty clause is false, while the empty formula is true.

The resolution proof technique is associated with clausal form.

4.4 A Framework for Completeness

1. Hintikka Set
2. Propositional Consistency Property
3. Model Existence Theorem
4. Tableau Complete
5. LK Complete
6. NK Complete

4.4.1 Propositional Hintikka Sets and the Model Existence Theorem

Basis of completeness argument: An open finished path is satisfiable.

Note: α, β refer to uniform notation.

Details follow Fitting.

Definition 4.1 (Hintikka Set) *A set H is a* propositional Hintikka set *provided*

1. *For every proposition letter p at most one of p or $\neg p$ is in H.*
2. $\bot \notin H$, $\neg\top \notin H$.
3. *If $\neg\neg\phi \in H$ then $\phi \in H$.*
4. *If $\alpha \in H$ then $\alpha_1 \in H$ and $\alpha_2 \in H$.*
5. *If $\beta \in H$ then $\beta_1 \in H$ or $\beta_2 \in H$.*

Note: All open finished paths are Hintikka.

Lemma 4.2 (Hintikka's Lemma) *Every propositional Hintikka set is satisfiable.*

We now generalize from a Hintikka set to collections of related sets that relate to Hintikka sets. We will build a framework that can be applied to yield completeness results for multiple proof systems. The techniques introduced here will generalize to first-order logic.

One way to view this definition is that we will see proof procedures as moving in a state space where the start state is rather unstructured, but if the initial state is in any way satisfiable, the final state will be a Hintikka set. This state space is characterized by the definition of Propositional Consistency Property. This will let us conclude that if we can build a model for the final state of the proof procedure, we can build a model for the initial state.

Definition 4.3 *Let $\mathcal{C}$ be a collection of sets of propositional formulas. We call $\mathcal{C}$ a propositional consistency property if it meets the following conditions for each $S \in \mathcal{C}$:*

1. *For every proposition letter p at most one of p or $\neg p$ is in S.*
2. $\bot \notin S$, $\neg\top \notin S$.
3. *If $\neg\neg\phi \in S$ then $S \cup \{\phi\} \in \mathcal{C}$.*
4. *If $\alpha \in S$ then $S \cup \{\alpha_1, \alpha_2\} \in \mathcal{C}$.*
5. *If $\beta \in S$ then $S \cup \{\beta_1\} \in \mathcal{C}$ or $S \cup \{\beta_2\} \in \mathcal{C}$.*

To build a concrete example, take a satisfiable formula ϕ. Build a finished tableau with an open finished path Θ. Write down all of the intermediate paths that were extended to build Θ (do not perform the optimization to remove reduced formulas). Name these paths $\Theta_1, \Theta_2, \ldots, \Theta_k$, where $\Theta_1 = \{\phi\}$ and $\Theta_k = \Theta$. Let S_i be the set of formulas in Θ_i. The collection of sets $\{S_1, \ldots, S_k\}$ is a propositional consistency property.

While this example is finite, note that the definition is significantly more general. We will reuse this framework to get models for first-order logics.

Theorem 4.4 (Propositional Model Existence) *If $\mathcal{C}$ is a propositional consistency property, and $S \in \mathcal{C}$, then S is satisfiable.*

Proof strategy: Basic idea, any $S \in \mathcal{C}$ can be expanded into a propositional Hintikka set that is also in $\mathcal{C}$. Challenge, what if S is infinite?

Claim 4.5 *A propositional consistency property is* subset closed *if it contains, with each member, all subsets of that member. Every propositional consistency property can be extended to one that is subset closed.*

Claim 4.6 *A Propositional consistency Property $\mathcal{C}$ is of* finite character *provided $S \in \mathcal{C}$ if and only if every finite subset of S belongs to $\mathcal{C}$. Every propositional consistency property of finite character is subset closed.*

Claim 4.7 *A propositional consistency property that is subset closed can be extended to one of finite character.*

Claim 4.8 *If $\mathcal{C}$ is a propositional consistency property of finite character, and $S_1, S_2, S_3, \ldots$ is a sequence of members of $\mathcal{C}$ such that $S_1 \subseteq S_2 \subseteq S_3 \ldots$. Then $\bigcup_i S_i$ is a member of $\mathcal{C}$.*

Proof of 4.8: Since $\mathcal{C}$ is of finite character, to show $\bigcup_i S_i \in \mathcal{C}$, it is sufficient to show for every finite subset of $\bigcup_i S_i$ is in $\mathcal{C}$. Suppose $\{\phi_1, \phi_2, \ldots, \phi_k\} \subseteq \bigcup_i S_i$. We must show $\{\phi_1, \phi_2, \ldots, \phi_k\} \in \mathcal{C}$. For each ϕ_i there is a smallest n_i such that $\phi_i \in S_{n_i}$. Let $n = \max\{n_1, \ldots, n_k\}$. Clearly $\phi_i \in S_n$. But $S_n \in \mathcal{C}$ and $\mathcal{C}$ is subset closed. Hence $\{\phi_1, \phi_2, \ldots, \phi_k\} \in \mathcal{C}$.

Proof of 4.4 Suppose $S \in \mathcal{C}$. We must show that S is satisfiable. To do this we will expand S to a maximal set $H \in \mathcal{C}$ that is a Hintikka set containing S.

Based on the earlier claims, we can assume that $\mathcal{C}$ is of finite character.

We assume that we have a countable set of proposition letters, and that there is some standard countable enumeration of all propositional formulas: $\psi_1, \psi_2, \ldots$. We build a chain of sets $S_1, S_2, \ldots$ as follows:

$$\begin{aligned} S_1 &= S \\ S_{n+1} &= \begin{cases} S_n \cup \{\psi_n\} & \text{if } S_n \cup \{\psi_n\} \in \mathcal{C} \\ S_n & \text{otherwise} \end{cases} \end{aligned}$$

Let $H = S_1 \cup S_2 \cup S_3 \cup \ldots$. Clearly H extends S. Since $\mathcal{C}$ is of finite character it is closed under chains. Hence $H \in \mathcal{C}$.

H is maximal in $\mathcal{C}$. That is, if $H \subseteq K$ for $K \in \mathcal{C}$ then $H = K$.

H is a Hintikka set. Suppose $\alpha \in H$, then $\alpha_1, \alpha_2 \in H$, because $H \cup \{\alpha_1, \alpha_2\} \in \mathcal{C}$, a set which extends H in $\mathcal{C}$. Since H is maximal $H = H \cup \{\alpha_1, \alpha_2\}$. The other conditions are similar.

By Hintikka's lemma (4.2) H is satisfiable. Hence S is satisfiable since $S \subseteq H$.

4.4.2 Application to Tableau completeness

Definition 4.9 (Fitting 3.7.1) *A finite set S of propositional formulas is* tableau consistent *if there is no closed tableau for S.*

Lemma 4.10 (Fitting 3.7.2) *The collection of all tableau consistent sets is a Propositional Consistency Property.*

Consider a tableau consistent set S. To establish a Propositional Consistency Property we must show that related sets from clauses 3, 4 and 5 of the definition are satisfied, that is that

3. If $\neg\neg\phi \in S$ then $S \cup \{\phi\} \in \mathcal{C}$.
4. If $\alpha \in S$ then $S \cup \{\alpha_1, \alpha_2\} \in \mathcal{C}$.
5. If $\beta \in S$ then $S \cup \{\beta_1\} \in \mathcal{C}$ or $S \cup \{\beta_2\} \in \mathcal{C}$.

We establish each of these by showing the contrapositive.

We present case (4), the others are similar. We need to show that if $S \cup \{\alpha_1, \alpha_2\}$ is not tableau consistent then S is not tableau consistent.

Suppose $S \cup \{\alpha_1, \alpha_2\}$ is not tableau consistent. Then there is a closed tableau from $S \cup \{\alpha_1, \alpha_2\}$. Furthermore, we know that $S = \{\alpha, \phi_1, \ldots, \phi_n\}$. We can obtain a closed tableau for S by applying the α rule on α, extending it to $S \cup \{\alpha_1, \alpha_2\}$, and then proceeding to the closed tableau as before.

Theorem 4.11 (Completeness for Propositional Tableaux, Fitting 3.7.3) *If X is a tautology, X has a tableau proof.*

Proof: We show the contrapositive: if ϕ does not have a tableau proof, then ϕ is not a tautology. If ϕ does not have a tableau proof, then $\{\neg\phi\}$ is tableau consistent. By the model existence theorem $\neg\phi$ is satisfiable. Hence it is not a tautology.

Comment This presentation does not deal with strictness—the fact that we only reduce each formula once. For the propositional case the strict algorithm we have implemented is complete, but this proof does not show it. We don't strengthen the proof because these techniques do not generalize to the first-order case.

4.4.3 Application to other systems

Fitting shows the model existence theorem can be used to prove other proof theories consistent as well. The key is to pick the right property.

For natural deduction (NK)

Definition 4.12 (Fitting 4.2.4) *Let ψ be a propositional formula. Call a set S ϕ-*natural deduction inconsistent *if $S \vdash \phi$; otherwise call S ϕ-*natural deduction consistent*.*

Lemma 4.13 (Fitting 4.2.5) *For each formula ϕ, the collection of all ϕ-natural deduction consistent sets is a propositional consistency property.*

For sequent calculus (LK) Negation is extended to sets element wise.

Definition 4.14 (Fitting 4.3.6) *Let S be a set of formulas. An* associated sequent *for S is a sequent $\Gamma \vdash \neg\Delta$, where Γ, Δ partition S.*

Lemma 4.15 (Fitting 4.3.7) *If any associated sequent for S has a proof, every associated sequent does.*

Definition 4.16 (Fitting 4.3.8) *A finite set S of formulas is* sequent inconsistent *if any associated sequent has a proof. S is* sequent consistent *if it is not sequent inconsistent.*

Claim 4.17 (Fitting 4.3.9) *The collection of sequent consistent sets is a propositional consistency property.*

Theorem 4.18 (Sequent Completeness, Fitting 4.3.10) *If ϕ is a tautology, then ϕ is a theorem of the sequent calculus.*

5 Applications of SAT solvers

Insert Tim's stuff here.

6 Ideas in SAT Solving

When we first looked at satisfiability, we looked at it from a mathematical perspective. We immediately focused on complete algorithms, that is, algorithms that can solve any instance of the satisfiability problem and proof procedures that can prove any semantically valid tautology.

Satisfiability is a famously difficult problem. Stephen Cook was studying satisfiability when he invented the concept of NP-completeness. Specifically, he showed that any problem that can be computed by a non-deterministic Turing machine in polynomial time could be reduced to an instance of SAT. Solving

```
anNF (p@(LetterP _)) = p
anNF (NotP p) = NotP (anNF p)
anNF (AndP p q) = AndP (anNF p) (anNF q)
anNF (OrP p q) = NotP (AndP (NotP (anNF p)) (NotP (anNF q)))
anNF (ImpliesP p q) = NotP (AndP (anNF p) (NotP (anNF q)))
anNF _ = error "anNF"
```

Figure 6: And-Negation-Normal Form

that SAT problem would solve the acceptance problem for the non-deterministic Turing machine. This theorem, now known as the Cook-Levin theorem, is a very direct reduction showing that SAT is complete for NP. That means that if you can solve SAT in P, then $P = NP$.

After 40 years of intense effort we have not found a polynomial time algorithm for SAT (although this week one has been claimed to exist). We have also not been able to prove a non-polynomial lower bound.

With an efficient complete solver highly unlikely, we must look at incomplete solvers if we want to be fast. The past decade has seen tremendous progress on SAT solving algorithms that work well in practice. All of these algorithms are either incomplete (meaning they don't solve all instances of the problem) or are very expensive in some cases.

6.1 Simple, Incomplete SAT solvers

Huth and Ryan sketch a couple of simple algorithms that are inspired by ideas in Stalmarck's method, first published in 1990. The algorithms are called linear and cubic; the names describe achievable complexity of the implementations. We will discuss Haskell implementations of these algorithms. The Haskell implementation may not achieve these bounds exactly, but the bounds are achievable.

6.1.1 Conceptual Description

Both algorithms use the same representation of propositions as a directed acyclic graph (dag). The dag maximizes sharing, achieving a compact representation and giving an efficient structure for propagating information. It has the very nice property that every any consistent labeling of the graph with True and False will correspond to a satisfying assignment.

The dag representation is based on the adequate set of operators $\wedge$ and $\neg$, so the first step is rewriting the formula into and-negation-normal-form. In this normal form we only have proposition letters, and and not. Simple code to do this conversion is given in Figure 6. This code does not deal with constants for true and false.

For example, consider the formula $p \wedge \neg(q \vee \neg p)$, which is Huth and Ryan example 1.48. We can calculate its normal form as follows:

```
*SimpleSetExamples> ex1_48
```

```
p /\ ~(q \/ ~p)
*SimpleSetExamples> anNF it
p /\ ~~(~q /\ ~~p)
```

The conversion to a dag will yield a graph with the full term represented by the dag's unique source (in-degree 0) and the proposition letters exactly the set of sinks (out-degree 0). The dag for the example is:

```
8     0 /\ 7
7     ~ 6         8
6     ~ 5         7
5     2 /\ 4      6
4     ~ 3         5
3     ~ 0         4
2     ~ 1         5
1     q           2
0     p           8,3
```

In this representation the left hand column is node number. The center column shows the operator labeling the node and the node numbers of its children. The right hand column is the set of parents of the node.

The algorithm solves the sat problem by building a truth assignment for every node in the graph. This truth assignment is called a *marking.* If a complete, consistent marking can be constructed the algorithm terminates with success, declaring the formula satisfiable. The linear algorithm makes only logically necessary markings "forced" by marking the root as True. If while making a necessary marking the algorithm attempts to mark a True node as False or a False node as True the algorithm terminates declaring the formula unsatisfiable. If the algorithm fails to construct a complete assignment then it terminates declaring the formula undetermined. It is because the third outcome is a possibility that this is called an incomplete method. It does not solve all instances of SAT.

The marking algorithm begins at the root, which it marks True. When an $\wedge$ node is marked True, its children will be marked True as well. In the example, the root (8) is marked True. This propagates to 0, which is the proposition letter p, and to 7, which is a negation. When a negation node is marked, the complement of the mark is passed to its child. This marks 6 as False. Since 6 is a negation, its child is marked True. Thus 5 is marked True. Since 5 is an $\wedge$ node, its children are marked True. The remaining nodes are all negations, which force their children. The final assignment and an animation of its construction is given here:

```
State = Satisfiable
8     0 /\ 7                    TT         +
7     ~ 6         8             T  T       +
6     ~ 5         7             F   F      +
5     2 /\ 4      6             T    T     +
4     ~ 3         5             T       T  +
3     ~ 0         4               F        F e+
2     ~ 1         5               T     T     +
1     q           2               F      F    +
0     p           8,3             T T       = +
```

In this presentation of the algorithm, the first three major columns present the dag, and the fourth presents the animation. Within the animation, the first column is the final assignment constructed. After that, moving left to right, we see how the algorithm proceeded. In the tenth step the `=` indicates that a redundant marking of node 0 was called for. The eleventh step, indicated with an `e`, revisited node 3 because its child was marked. Finally the `+` mark indicates that a verification of the complete marking completed successfully.

In this example we have basically used three rules:

$$\begin{array}{c} T(\phi \wedge \psi) \\ T\phi \quad T\psi \end{array} \qquad\qquad \begin{array}{c} T\neg\phi \\ | \\ F\phi \end{array} \qquad\qquad \begin{array}{c} F\neg\phi \\ | \\ T\phi \end{array}$$

In the example we applied these rules "top to bottom". This kind of propagation is sometimes called forcing, since marking the upper node forces (functionally determines) the mark of the lower node.

The rules can also be read "bottom to top." That is, if the children of an $\wedge$ node are both marked true that $\wedge$ node can be marked True. Similarly markings can propagate up through a $\neg$ node.

Other relationships that can be exploited include the following:

$$\begin{array}{c} F^2(\phi \wedge \psi) \\ F^1\phi \quad ?\psi \end{array} \qquad \begin{array}{c} F^2(\phi \wedge \psi) \\ ?\phi \quad F^1\psi \end{array} \qquad \begin{array}{c} F^1(\phi \wedge \psi) \\ T^1\phi \quad F^2\psi \end{array} \qquad \begin{array}{c} F^1(\phi \wedge \psi) \\ F^2\phi \quad T^1\psi \end{array}$$

Here markings superscripted with 1 force those markings superscripted with 2. The algorithm attempts to propagate information by these rules whenever a child is marked.

For example, consider the formula $p \wedge \neg(p \wedge q)$.

```
State = Satisfiable
4     0 /\ 3                      TT       +
3     ~ 2         4               T  T     +
2     0 /\ 1      3               F   Fe e +
1     q           2               F     F =+
0     p           4,2             T T      +
```

The "top down" forcing marks nodes 4, 3, 2 and 0. But since node 2 is an $\wedge$ marked with False it does not force node 1. However, since node 0 was marked False, nodes 0 and 2 together force node 1 to False, completing the marking of the graph. In the animations the `e` symbol indicates that a node is being explored to see if is participating in a non-top-down forcing relationship. In this case the first exploration marks 1 False. The second exploration, which was triggered because node 2 is parent of node 0, which was marked in step 2, makes a redundant marking of node 1.

Of course, one possible outcome of the algorithm is to discover that a proposition is not satisfiable. The simplest such proposition is $p \wedge \neg p$. It's animation is:

```
State = Unsatisfiable Exploring 1
2     0 /\ 1                    TT
1     ~ 0         2             T  T
0     p           2,1           T T X
```

This is marked unsatisfiable. The symbol `X` in the animation shows the step at which the contradiction was discovered. In particular, after marking node 1 True, it was attempting to mark node 0 False, but it was previously marked True.

While the linear algorithm does very well with $\wedge$ nodes marked True, it does very little with $\wedge$ nodes marked False. In particular, it fails on the very simple example $\neg(p \wedge q)$, which is satisfiable by any valuation in which one of p or q is marked False:

```
State = Unknown
3     ~ 2                       TT  -
2     0 /\ 1      3             F Fe-
1     q           2                 -
0     p           2                 -
```

Note here the outcome is unknown. The symbol `-` in the animation indicates that the test of the completeness of the markings failed. (The test actually tests both completeness and consistency of the markings; in this case the marking is incomplete.)

One reason the linear algorithm has problems with this example, which essentially encodes an $\vee$ operation, is that it only assigns markings that are required; it never guesses or backtracks. Contrast this with the complete tautology checker you implemented in earlier assignments. That did complete search of all possible assignments. Also consider the tableau proof technique which would branch the path to explore disjunctions. Both of these complete algorithms are potentially exponential, partially because they can layer guess upon guess and may explore an exponential number of alternatives.

The cubic algorithm extends the linear algorithm by allowing some speculation, but it only speculates on one graph label at a time. It never explores multiple dependent guesses. Speculation is managed by distinguishing between *permanent marks*, to which the algorithm is fully committed, and *temporary marks*, that are dependent upon the speculative mark (the guess). In the $\neg(p \wedge q)$ example, the cubic algorithm explores what happens if node 0 is marked True. In that case, since node 2 is marked False, this forces node 1 to be marked False. It then tests this marking, and discovers that this satisfies all nodes, confirming that the graph is satisfiable. This is animated:

```
State = Satisfiable
3     ˜ 2                         TT  -     +
2     0 /\ 1      3               F Fe-   e +
1     q           2               F   -w   F+
0     p           2               T   -wtT  +
```

In this animation we see that after the failed test, the two unmarked nodes, 1 and 0, are placed on a worklist (note symbol `w`). Node 0 is then speculatively marked True (note symbol `t`). Since its child has been marked, node 2 is explored, causing node 1 to be marked. This marking is complete and consistent, indicated by the symbol `+`. Whenever a complete and consistent marking is created, even if it involves speculation, the algorithm terminates with the outcome satisfiable.

If marking node 0 True had not yielded a satisfying assignment the algorithm would have explored marking node 0 False instead. To do this it would roll-back the temporary marks and start again with a new temporary marking. If both speculative marks yield non-contradictory and inconclusive markings, then the intersection of the two markings are adopted as permanent marks. For example, consider the animation of the partial run of the algorithm on $\neg((q \wedge \neg q) \wedge r \wedge (p \wedge \neg p)$:

```
State = SpeculateTrue
9     ˜ 8                         TT  -         -        -
8     2 /\ 7      9               F Fe-        e-    e   -
7     3 /\ 6      8                   -w        -        -
6     4 /\ 5      7                   -w        -        -
5     ˜ 4         6                   -w        -        -
4     p           6,5                 -w        -        -
3     r           7                   -w        -        -
2     0 /\ 1      8               F   -w  e  eF -  eF   e-!F
1     ˜ 0         2                   -w   eF   -     eT -
0     q           2,1                 -wtT      -fF      -m
```

Here we see that setting node 0 to be True forced nodes 1 and 2 to be False. Similarly, setting node 0 to be False forced node 2 to be False and node 1 to be true. Going forward, the algorithm does not commit node 0 to have any mark, but it does commit node 2 to be marked False. The symbol `m` marks the node forcing the merge assignment. The symbol `!` flags the nodes marked by merging the two markings.

Later on in the execution of this example, the cubic algorithm explores the consequences of setting node 4 to be True and False. Again both outcomes are inconclusive, but they agree on the markings of nodes 6 and 7:

```
9     ˜ 8                         T          -          -
8     2 /\ 7      9               F         e-      e   -
7     3 /\ 6      8               F       eF -    eF    -!F
6     4 /\ 5      7               F  e  eF   -  eF     e-!F
5     ˜ 4         6                   eF     -       eT -
```

```
4     p           6,5              tT        -fF        -m
3     r           7                          -          -
2     0 /\ 1      8               F          -          -
1     ~ 0         2                          -          -
0     q           2,1                        -          -
```

Ultimately the cubic algorithm fails to find a marking for this graph, even though it is true under all truth assignments.

If one of the speculative markings yields a contradiction then its complement mark is added to the permanent marking. This can be seen in the animation of Huth and Ryan's unsatisfiable equation (1.11):

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

The initial phase of the linear algorithm leaves several nodes unmarked:

```
28    10 /\ 27                    TT                  -
27    13 /\ 26    28              T   T               -
26    16 /\ 25    27              T      T            -
25    19 /\ 24    26              T         T         -
24    ~ 23        25              T            T      -
23    17 /\ 22    24              F             Fe    -
22    ~ 21        23                                  -w
21    ~ 20        22                                  -w
20    11 /\ 14    21                                  -w
19    ~ 18        25              T          T        -
18    5 /\ 17     19              F           F   e   -
17    ~ 1         23,18                               -w
16    ~ 15        26              T       T           -
15    3 /\ 14     16              F        F       e  -
14    ~ 5         20,15                               -w
13    ~ 12        27              T    T              -
12    1 /\ 11     13              F     F           e -
11    ~ 3         20,12                               -w
10    ~ 9         28              T T                 -
9     1 /\ 8      10              F  F               e-
8     ~ 7         9                                   -w
7     ~ 6         8                                   -w
6     3 /\ 5      7                                   -w
5     ~ 4         18,14,6                             -w
4     r           5                                   -w
3     ~ 2         15,11,6                             -w
2     q           3                                   -w
1     ~ 0         17,12,9                             -w
0     p           1                                   -w
```

The cubic algorithm proceeds, speculating that p (node 0) may be True. This eventually yields a contradiction at node 14. The algorithm then concludes that p must be False:

```
28    10 /\ 27                T-
27    13 /\ 26    28          T-
26    16 /\ 25    27          T-
25    19 /\ 24    26          T-
24    ~ 23        25          T-
23    17 /\ 22    24          F-       e
22    ~ 21        23           -w       F
21    ~ 20        22           -w        T
20    11 /\ 14    21           -w         Fe     e
19    ~ 18        25          T-
18    5 /\ 17     19          F-            e
17    ~ 1         23,18        -w    eT
16    ~ 15        26          T-
15    3 /\ 14     16          F-                       e
14    ~ 5         20,15        -w               eT     X
13    ~ 12        27          T-
12    1 /\ 11     13          F-
11    ~ 3         20,12        -w                  F
10    ~ 9         28          T-
9     1 /\ 8      10          F-
8     ~ 7         9            -w
7     ~ 6         8            -w
6     3 /\ 5      7            -w
5     ~ 4         18,14,6      -w            F
4     r           5            -w             T
3     ~ 2         15,11,6      -w                  T
2     q           3            -w                   F
1     ~ 0         17,12,9      -w  eF
0     p           1           F-wtT                    0F
```

Unfortunately, this also leads to a contradiction. It forces a contradictory marking of node 8:

```
28    10 /\ 27                T
27    13 /\ 26    28          T
26    16 /\ 25    27          T
25    19 /\ 24    26          T
24    ~ 23        25          T
23    17 /\ 22    24          F      e
22    ~ 21        23
21    ~ 20        22
20    11 /\ 14    21
19    ~ 18        25          T
18    5 /\ 17     19          F       e        e
17    ~ 1         23,18       F    eF            =
16    ~ 15        26          T
15    3 /\ 14     16          F            e
```

```
14    ~ 5         20,15           F             F
13    ~ 12        27              T
12    1 /\ 11     13              F        e
11    ~ 3         20,12           F         F
10    ~ 9         28              T
9     1 /\ 8      10              F                        e
8     ~ 7         9               T                      eT X
7     ~ 6         8               F                    eF
6     3 /\ 5      7               T                  eT
5     ~ 4         18,14,6         T              T
4     r           5               F               F
3     ~ 2         15,11,6         T          T
2     q           3               F           F
1     ~ 0         17,12,9         T  eT
0     p           1               F0F
```

Having obtained contradictions for both possible markings of node 0, the algorithm concludes the proposition is unsatisfiable.

6.1.2 Discussion of complexity

Why is the linear algorithm linear? The following facts all contribute:

1. A mark can be assigned in constant time.
2. A propagating exploration operation can be performed in constant time.
3. Since the maximum out-degree of the dag is 2, the edge set of the graph is linear in the number of nodes.
4. Each edge will be traversed at most once to perform a "top-down" marking.
5. Each edge will be traversed at most once to trigger a propagating exploration.
6. The test of consistency and completeness of the marking can be performed in linear time and is performed exactly once.

What about the cubic algorithm? In a nutshell, the cubic algorithm as described invokes the linear algorithm twice per node in the graph. That suggests it should be the quadratic algorithm! This quadratic algorithm is sound, but it turns out to be very sensitive to the order of the worklist! Since information accumulates in the markings, it is possible for there to be dependencies between markings. There may be some critical sequence necessary to unlock all of the information necessary to discover a marking. However if we run the quadratic algorithm once for every (unmarked) node, we obtain a cubic algorithm that is insensitive to the order of the worklist.

6.1.3 An implementation in Haskell

We have constructed a Haskell implementation of the linear and cubic algorithms. Our goal is to gain understanding of the algorithms and to develop some visualization tools for the animations included here. We are not trying to write the fastest code possible.

Representation The file DAG.hs contains the code for constructing and-negation-normal-form and building the dag. It imports the library Data.Map.Map to construct the mappings that implement both markings and the dag.

Data.Map.Map gives natural, extensible mappings that are moderately efficient. Basic operations, however, are not unit time. This is the most significant algorithmic choice we make that compromises our ability to meet the linear and cubic time complexity bounds of the algorithms we are studying.

To represent the dag, the key operations we will need are:

1. From a node, find the operator at that location
2. From a node, find its successors (or children)
3. From a node, find its predecessors (or parents)
4. Search for a node representing a particular structure in the graph.

To represent the graph we pick values of type `Int` to name the nodes. We represent edges and the node type by building a type called Struct that mimics the shape of the Propositions we are representing, but where Prop has subterms the Struct type will have `Int` values representing edges to other nodes in the graph. The Struct type is defined:

```
data Struct a =
     LetterS a
   | AndS Int Int
   | NotS Int
 deriving (Eq,Ord,Show)
```

Note that this type is very similar to the `Prop a` type that we used for propositions, but this type has no recursive components, that is, none of the constructors take values of type `Struct a`.

To build the graph representation we begin with a mapping from `Int` to `Struct a`, called `structure`. It naturally supports the first two operations. We combine this with to other maps, parents and key, to support the other two operations. Finally, we add a field indicating the next `Int` value to use when extending the dag with a new node. These three mappings and next node value are collected in a Haskell record of type `Dag a`:

```
data Dag a = Dag { next :: Int
                 , key:: (Map (Struct a) Int)
                 , parents:: (Map Int [Int])
                 , structure:: (Map Int (Struct a)) }
```

Dags are constructed with the `mkDag` function. It assumes a proposition in and-negation-normal form. It's signature is:

```
mkDag :: Ord a => Prop a -> (Dag a, Int)
```

Markings are also implemented by maps form Data.Map. That type is declared:

```
type Marking = Map Int Bool
```

This definition is in the source file SAT.hs, which contains the implementation of the linear and cubic algorithms.

Algorithmic Control The implementation strategy models the imperative nature of the algorithm descriptions by defining a record type to characterize the state of the algorithm and a step function to advance the algorithm by one step. The complete state is represented by the Haskell record type `SAT a`. Its declaration is:

```
data SAT a =  SAT { dag :: Dag a,
                    marking :: Marking,
                    committedMarking :: Marking,
                    alternateMarking :: Marking,
                    toForce :: [(Int, Bool, Maybe Int)],
                    toExplore :: [Int],
                    markSpeculatively :: [Int],
                    currentSpeculation :: Maybe Int,
                    state :: SATState,
                    onContradiction, onIndetermined :: SATState,
                    history :: [Event],
                    annotation :: Doc
                  }
```

The basic elements of the record include the dag being labeled and the current marking being developed. Since the cubic algorithm requires that we track multiple markings, there are additional fields. The committedMarking contains "permanent marks". The alternateMarking contains the most recently abandoned marking generated by speculating that a node might be True. The lists `toForce` and `toExplore` are worklists that are used by the linear algorithm. The list `markSpeculatively` is a worklist used by the cubic algorithm.

The implementation achieves a modular implementation of the two algorithms by using a top-level state machine structure to manage the control flow. The states are specified in the declaration:

```
data SATState
 = Satisfiable | Unsatisfiable  | Unknown  -- Outcome States
 | Working | CubicSetup
 | SpeculateTrue | SpeculateTContra | SpeculateTIndetermined
 | SpeculateFContra | SpeculateFIndetermined
   deriving Show
```

For the linear algorithm the only states that are used are the outcome states and the `Working` state.

The `step` function is defined as nine separate equations. These equations are dispatched on the finite-state machine state and the length of the worklists `toMark` and `toExplore`. In the `Working` state, the highest priority is given to propagating the direct forcings down the tree. The `toMark` worklist has a list of forcings to be carried out. As nodes are marked, a list of nodes that may be able to propagate forcings up or to peer nodes is collected on the worklist `toExplore`. Nodes are added to the `toExplore` list when an $\wedge$ node is marked False and has no direct forcings or when they are the parents of a node that has been marked. If a marking was forced by a parent being marked that parent is not reexplored (it is already marked). If a marked node is assigned a second consistent mark the operation succeeds without scheduling any further work. If a marked node is assigned an inconsistent mark then a logical contradiction has been discovered. In the linear algorithm the algorithm terminates declaring the proposition unsatisfiable. In the cubic algorithm more nuanced actions may be taken using the finite-state control. Details of the mark operation are handled by the `mark`function.

When the `toMark` worklist is empty, the `toExplore` worklist is pursued. Exploration of a node may lead to adding items to either of the two main worklists. The `explore` function performs the actual exploration.

When both primary worklists are exhausted the algorithm tests the mapping. It uses the `verify` function. The verify function returns `Just True` if the marking is complete and consistent. It returns `Just False` if the marking is complete but not consistent. It returns `Nothing` if the marking is incomplete. In both the linear and cubic algorithms the algorithm terminates immediately with success if an assignment is verified. In the linear algorithm any dag that is not Satisfiable is Unknown.

Exercise There will be a companion exercise in which students are given a version of the source code with some of the control information redacted and are invited to reconstitute the program to regain its functionality.

7 First-order Logic

7.1 Motivation

In Propositional logic we studied the essence of the logical connectives. In first-order logic we add a new element: the ability to make logical statements about a collection of things. The collection is often called the *universe* or the *domain of discourse.* The things we can say atomically are called *predicates.*

Examples of things we might want to say in first-order logic include:

- All natural numbers are either zero or the successor of a natural number.

- In the family tree from the Finite Set example no one is a descendent of themselves.
- Addition is commutative.

To talk about things we need a language to describe them. In first-order logic, the set of *terms* describe things in the domain of discourse. Properties of terms are called *predicates*. Predicates take over the role of the atomic formulas in propositional logic. Once we have a collection it becomes natural to talk about all elements of the collection having some property. We do this with *quantification*. In first-order logic we extend the logical language with the quantifiers for all ($\forall$) and there exists ($\exists$). Quantifiers bind the name of a variable. These variables occur within terms, naming elements of the domain of discourse.

In the language of propositional logic the statements above are translated as follows:

- $\forall x.(x = 0) \vee (\exists y.x = S(y))$
- $\neg\exists x.\text{parent-child}(x, x)$
- $\forall x.\forall y.x + y = y + x$

As in the propositional case we will give a meaning to the logical language in mathematics. Informally we expect the three natural language sentences to be true. But that is because we have a great deal of knowledge about the domains of natural numbers and parent-child relationships. In first-order logic we will have to represent this contextual knowledge as well. To represent this we will formulate domain knowledge as a set of sentences we call *axioms*. We will define a semantic notion of truth that makes the formal sentences true in all models that make the axioms true.

We will also develop a proof theory. The proof theory will let us prove sentences that are logical consequences of the axioms. We will find that in the general case truth is not decidable, but that every statement that is semantically true in all models of the axioms is provable. That is, we will prove soundness and completeness for first-order logic.

But what about the incompleteness theorem? How can we have a complete proof system for arithmetic? Don't panic! We do not have a complete proof system for arithmetic. Gödel's incompleteness theorem is safe. It says that we cannot find a complete and effective axiomatization of arithmetic. So even when we can prove every logical consequence of our axioms, we will not be proving every true fact of our intended model because we start from a provably deficient set of axioms.

```
module Term (Term (..), variables, newVar, newFun) where
import Char

-- Note that constants are represented by functions with no arguments
-- Functions can be skolem functions and are then marked as such

data Term f v = Var v
              | Fun Bool f [Term f v] deriving Eq

-- Bind is substitution on Terms
instance Monad (Term f) where
  return v = Var v

  (Var v)      >>= s = s v
  (Fun b n ts) >>= s = Fun b n (map (s =<<) ts)

variables :: Term f v -> [Term f v]
variables (Var v)  = [Var v]
variables (Fun s n ts) = concat (map variables ts)
```

Figure 7: Fragment of Term.hs

```
module Subst(Subst, emptySubst, (|=>), (|->), (|/->)) where

type Subst v m = v -> m v

emptySubst :: Monad m => Subst v m
emptySubst v = return v

-- Substituting the variable v with the term t

(|->) :: (Eq v, Monad m) => v -> m v -> Subst v m
(v |-> t) v' | v == v'   = t
             | otherwise = emptySubst v'

-- Composing two substitutions
(|=>) :: Monad m => Subst v m -> Subst v m -> Subst v m
s1 |=> s2 = (s1 =<<) . s2

-- Removing a variable from a substitution
(|/->) :: (Eq v, Monad m) => v -> Subst v m -> Subst v m
(v |/-> s) v' | v == v'   = return v'
              | otherwise = s v'
```

Figure 8: Fragment of Subst.hs

```
module Formula where

import Term
import Subst
import Unify
import Monad
import Control.Monad (guard)
import List (nub)

data Formula r f v = Rel r [Term f v]
                   | Conn Cs [Formula r f v]
                   | Quant Qs v (Formula r f v)
                   deriving Eq

data Qs = All | Exist deriving Eq

data Cs = And | Or | Imp
        | T | F | Not
        deriving Eq

subst :: Eq v => (v -> Term f v) -> Formula r f v -> Formula r f v
subst s (Rel r ts)    = Rel r (map (s =<<) ts)
subst s (Conn c fs)   = Conn c (map (subst s) fs)
subst s (Quant q v f) = Quant q v (subst (v |/-> s) f)

vars :: (Eq f, Eq v) => Formula r f v -> [Term f v]
vars (Rel r ts)    = nub $ concat $ map variables ts
vars (Conn c fs)   = nub $ concat $ map vars fs
vars (Quant q v f) = nub $ filter (/= (Var v)) $ vars f
```

Figure 9: Fragment of Formula.hs (unification omitted)

7.2 Syntax

7.2.1 Formulas and Terms

First-order logic is a parameterized family of languages. The parameters specify constants ($\mathcal{C}$), function symbols ($\mathcal{F}$), and predicate symbols ($\mathcal{P}$)[1]. We will write $L(\mathcal{P}, \mathcal{F}, \mathcal{C})$ for the first-order language over $\mathcal{P}, \mathcal{F}$, and $\mathcal{C}$. Within a first-order language there are two essential syntactic categories: terms, which name individuals, and formulas, which logical phrases. Terms and formulas interact in two basic ways:

- Quantified variables are bound in formulas, but name individuals used in terms.
- Predicates are atomic elements of formulas, but hold of terms.

Functions and Predicate symbols can be correctly applied to a fixed number of arguments, called the arity. In our implementations it is natural to treat the set of constants as function symbols of arity 0.

Definition 7.1 (HR 2.1) Terms ($\mathcal{F}$) *are defined as follows.*

- *Any variable is a term.*
- *If $c \in \mathcal{F}$ is a nullary function, then c is a term.*
- *If $t_1, t_2, \ldots, t_n$ are terms and $f \in \mathcal{F}$ has arity $n > 0$, then $f(t_1, t_2, \ldots, t_n)$ is a term.*
- *Nothing else is a term.*

Compare this definition with that in the file `Term.hs`, shown in Fig 7. The extra Boolean argument on function symbols should be ignored at this point. It will be set to True for special functions called Skolem functions.

Definition 7.2 *The* atomic formulas *over $(\mathcal{F}, \mathcal{P})$ are defined inductively as follows:*

- *If $P \in \mathcal{P}$ is a predicate symbol of arity $n \geq 1$, and if $t_1, t_2, \ldots, t_n$ are terms over $\mathcal{F}$, then $P(t_1, t_2, \ldots, t_n)$ is an atomic formula.*
- *$\top$ and $\bot$ are atomic formulas.*

Definition 7.3 *The* formulas *over $(\mathcal{F}, \mathcal{P})$ are defined inductively as follows:*

- *If ϕ is an atomic formula over $(\mathcal{F}, \mathcal{P})$ then ϕ is a formula.*
- *If ϕ is a formula then so it $(\neg\phi)$.*
- *If ϕ and ψ are formulas then so are $(\phi \wedge \psi)$, $(\phi \vee \psi)$, and $(\phi \rightarrow \psi)$.*

[1]Fitting calls predicates relations and uses $\mathcal{R}$. I may sometimes accidentally revert to Fitting's notation.

- *If ϕ is a formula and x is a variable, then $(\forall x \phi)$ and $(\exists x \phi)$ are formulas.*
- *Nothing else is a formula.*

Corresponding definitions can be found in the source file `Formula.hs`, shown in Fig 9.

Note that this definition makes official formulas fully parenthesized, and all functions and predicates are written in prefix form. We will of course omit parenthesis whenever we can. We will write familiar functions, such as $+$, in infix. We will write familiar predicates, such as $=$, $<$, as infix. Our assumed precedence and associativity will refine that of HR's convention 2.4 by adopting the order:

- $\neg, \forall y,\ \exists y$
- $\wedge$
- $\vee$
- $\rightarrow$

In addition, we will sometimes use the "Church dot" contention that a dot symbol (.) is an open parenthesis extending as far to the right as possible. (This is similar to the Haskell idiom `$`.)

Some examples:

$$\begin{aligned}
x + y &\equiv +(x, y) \\
x = y &\equiv = (x, y) \\
a \vee b \wedge c &\equiv a \vee (b \wedge c) \\
a \wedge b \rightarrow b \wedge a &\equiv (a \wedge b) \rightarrow (b \wedge a) \\
\forall x\ \exists y\ y = Sx \wedge \phi &\equiv (\forall x(\exists y(= (y, Sx)))) \wedge \phi \\
\forall x\ \exists y . y = Sx \wedge \phi &\equiv (\forall x(\exists y(= (y, Sx) \wedge \phi))) \\
\forall x . (x = 0) \vee (\exists y . x = S(y)) &\equiv \forall x(= (x, 0) \vee (\exists y(= (x, S(y))))) \\
\forall x\ (x = 0) \vee (\exists y . x = S(y)) &\equiv (\forall x(= (x, 0)) \vee (\exists y(= (x, S(y)))))
\end{aligned}$$

Note carefully how different the last two pairs of formulas read!

7.2.2 Free and Bound variables

The introduction of quantifiers and variables brings a new complexity to our formal system. We need to track the association of variables in terms with variables used in quantifiers. When we can associate a variable in a term with an enclosing quantifier we call that a *bound* occurrence of the variable. When we cannot make the association it is a *free* occurrence. These distinctions will become very important when we give a semantics to first-order logic. A formula with no free variables is called a *sentence* or *closed formula.* A formula with free variables is sometimes called an *open formula.*

Ultimately sentences will play a very important role in first-order logic. We will be able to define the truth of sentences. For example, consider the two formulas $x + 1 = y$ and $\forall x\ \exists y\ x + 1 = y$. The first is an open formula, the second a sentence. Our knowledge of number theory says that the sentence should always be true. But when we look at the open formula we have to ask "what are x and y?

Definition 7.4 (Fitting 5.1.6, cf. HR 2.6) *The* free-variable occurrences *in a formula are defined as follows:*

1. *The free-variable occurrences in an atomic formula are all the variable occurrences in that formula.*
2. *The free-variable occurrences in $\neg\phi$ are the free-variable occurrences in ϕ.*
3. *The free-variable occurrences in $\phi \circ \psi$ are the free-variable occurrences of ϕ with the free-variable occurrences of ψ, for any binary connective $\circ$.*
4. *The free-variable occurrences in $\forall x\ \phi$ and $\exists x\ \phi$ are the free-variable occurrences in ϕ except for occurrences of x.*

A variable occurrence that is not free is bound

Compare this definition with the function `vars` defined in `Formula.hs` in Fig 9.

Definition 7.5 (Fitting 5.1.7) *A* sentence *(also called a* closed formula*) is a formula with no free-variable occurrences.*

7.2.3 Substitution

Substitution is the process of replacing variables by terms systematically. It is such a natural part of mathematical reasoning that it almost seems unnatural to talk about it. It is what lets us go from $\forall x\ \exists y.x + 1 = y$ to $\exists y.3 + 1 = y$.

Substitution can be formalized as either the replacement of a single variable by a term or as the replacement of all variables by a set of terms. Huth and Ryan present the former, defining $\phi[t/x]$ as the formula ϕ with term t replacing variable x. In implementations I find it more common to use the more functional view that a substitution σ is a map from variables to terms that is applied to all variables in a formula. These functions are generally written with postfix application, $t\sigma$ is the image of term t under the substitution σ.

For the following definitions we will assume a fixed logical language $L(\mathcal{P}, \mathcal{F}, \mathcal{C})$. The set of variables is V. The set of terms is T.

Definition 7.6 (Fitting 5.2.1) *A* substitution *is a mapping $\sigma : V \to T$ from the set of variables to the set of terms.*

The action of a substitution on a term:

Definition 7.7 (Fitting 5.2.2) *Let σ be a substitution. Define:*

1. $c\sigma = c$ *for any constant (0-airy function) symbol c.*

2. $[f(t_1, \ldots, t_n)]\sigma = f(t_1\sigma, t_2\sigma, \ldots, t_n\sigma)]$ *for an n-place function symbol f.*

Substitutions have a rich algebraic structure. They are an example of a *monad*, a categorical structure common in Haskell programs. In general, a monad in Haskell is defined by a polymorphic type constructor, frequently called `m`. There is a special function, sometimes called the unit, named `return` that has the polymorphic type `a -> m a`. For substitutions, the identity substitution is the unit.

There is a second function, called `>>=`, pronounced "bind", with type:

```
(>>=) :: (Monad m) => m a -> (a -> m b) -> m b
```

For the application of substitution we will focus on an interdefinable function, called the Kleisli star, written `=<<`, with type:

```
(=<<) :: (Monad m) => (a -> m b) -> m a -> m b
```

Kleisli star is exactly the operation that extends the action of a substitution from variables to terms.

In Figure 7 you see the definition of the Haskell type `Term`. Note that `Term` takes two type parameters, `f`, the type to represent function symbols, and `v`, the type to represent variables. For substitution we need to be polymorphic in the representation of variables. So the instance declaration defining the monad is on the partially applied type constructor `(Term f)`. Look at the types above that included a `m`. If you replace `m` by `Term f` you get back familiar, fully-applied Haskell types.

The instance declaration continues by defining `return`. Convince yourself that this is the identity substitution as promised.

The instance declaration completes with the definition of bind, specified as the infix operator `>>=`. Read `t >>= s` as the image of `t` under substitution `s`.

Definition 7.8 (Fitting 5.2.3) *Let σ and τ be substitutions. The composition of the substitutions, written $\sigma\tau$, has the following action on each variable x:*

$$x(\sigma\tau) = (x\sigma)\tau$$

Note in this definition that in the general case τ is lifted to a function from terms to terms according to Def 7.7. Compare this to the composition operator `|=>` in Fig 8. There we see explicitly that the substitution $\sigma\tau$ is the composition of σ with the Kleisli star of τ. Note: the Haskell function for composition of substitutions uses "non-diagrammatic" order. This is $\sigma\tau$ is written `tau |=> sigma`. (Perhaps I should change this!)

Proposition 7.9 (Fitting 5.2.4) *For every term t, $t(\sigma\tau) = (t\sigma)\tau$.*

Proposition 7.10 (Fitting 5.2.5) *Composition of substitutions is associative. That is, $(\sigma_1\sigma_2)\sigma_3 = \sigma_1(\sigma_2\sigma_3)$.*

Definition 7.11 (Fitting 5.2.6) *The* support *of a substitution σ is the set of variables x for which $x\sigma \neq x$. A substitution has* finite support *if the support set is finite.*

Proposition 7.12 (Fitting 5.2.7) *The composition of two substitutions having finite support is a substitution having finite support.*

Definition 7.13 (Fitting 5.2.8) *The notation $\{x_1/t_1, \ldots, x_n/t_n\}$ stands for the substitution σ with finite support that extends the identity substitution by mapping $x_i\sigma = t_i$.*

Definition 7.14 (Fitting 5.2.10)

$$y\sigma_x = \begin{cases} y\sigma & \text{if } y \neq x \\ x & \text{if } y = x \end{cases}$$

Definition 7.15 (Fitting 5.2.12, compare to HR) *A substitution being* free for a formula *is characterized as follows:*

1. *If A is atomic, σ is free for A.*
2. *σ is free for $\neg\phi$ if σ is free for ϕ.*
3. *σ is free for $\phi \circ \psi$ if σ is free for ϕ and σ is free for ψ.*
4. *σ is free for $(\forall x)\phi$ and $(\exists x)\phi$ provided: σ_x is free for ϕ, and if y is a free variable of ϕ other than x, $y\sigma$ does not contain x.*

Theorem 7.16 (Fitting 5.2.13) *Suppose the substitution σ is free for the formula ϕ, and the substitution τ is free for $\phi\sigma$. Then $(\phi\sigma)\tau = \phi(\sigma\tau)$.*

7.3 Semantics

To give meaning to formulas in first-order logic we need a mathematical structure, called a model, to interpret the domain of discourse, the constants, functions, and predicates. Once these elements are established we can define the truth of a sentence mathematically.

Definition 7.17 *A* model *for first-order logic is a structure $M = \langle D, I\rangle$ where:*

- *D is a non-empty set called the* domain *of M.*
- *I is a mapping, called an interpretation, that associates:*
  - *$c^I \in D$ for every $c \in C$*
  - *$f^I \in D^n \to D$ for every k-place function symbol $f \in F$.*
  - *$P^I \subseteq D^n$ for every n-place relation symbol $P \in R$.*

Ultimately models will give meaning to sentences.

To give meanings for formulas, we need an assignment of value to the variables:

Definition 7.18 *An* assignment *is a model $M = \langle D, I \rangle$ is a mapping A from the set of variables to the set D. We denote the image of the variable v under an assignment A by v^A.*

Given a model and an assignment, we can map terms to D.

Definition 7.19 *Association of values with terms in model $M = \langle D, I \rangle$ and assignment A.*

1. $c^{I,A} = c^I$
2. $v^{I,A} = v^A$
3. $[f(t_1, \ldots, t_n)]^{I,A} = f^I(t_1^{I,A}, \ldots, t_n^{I,A})$

Definition 7.20 (Fitting 5.3.4) *Let x be a variable. The assignment B in the model M is an x-*variant *of the assignment A, provided A and B assign the same values to every variable except possibly x.*

Definition 7.21 (Fitting 5.3.5, Cf. HR 2.18) *Let $M = \langle D, I \rangle$ be a model and A an assignment. To each formula ϕ associate a truth value $\phi^{I,A}$:*

1. *Atomic cases:*
   (a) $[P(t_1, \ldots, t_n)]^{I,A} = \mathbf{t} \leftrightarrow \langle t_1^{I,A}, \ldots, t_n^{I,A} \rangle \in P^I$
   (b) $\top^{I,A} = \mathbf{t}$
   (c) $\bot^{I,A} = \mathbf{f}$
2. $[\neg\phi]^{I,A} = \neg[\phi^{I,A}]$
3. $[\phi \circ \psi]^{I,A} = \phi^{I,A} \circ \psi^{I,A}$
4. $[(\forall x)\phi]^{I,A} = \mathbf{t} \leftrightarrow \phi^{I,B} = \mathbf{t}$ *for every assignment B in M hat is an x-variant of A.*
5. $[(\exists x)\phi]^{I,A} = \mathbf{t} \leftrightarrow \phi^{I,B} = \mathbf{t}$ *for some assignment B in M hat is an x-variant of A.*

Definition 7.22 (Fitting 5.3.6)

- *ϕ is* true in the model *M provided $\phi^{I,A} = \mathbf{t}$ for all assignments A.*
- *ϕ is* valid *if ϕ is true in all models for the language.*
- *A set S of formulas is* satisfiable *in M provided there is some assignment A (called a* satisfying assignment*), such that $\phi^{I,A} = \mathbf{t}$ for all $\phi \in S$.*
- *S is* satisfiable *if it is satisfiable in some model.*

7.4 Exercises

1. (a) Work out the logical language and standard model for the integers modulo 3 under addition. Start with some constants, at least the function $+$, and the relation $=$.

   (b) Show that an expected true sentence is true in the model; show that an expected false sentence is false in the model.

   (c) Can you write a sentence that will be true in your intended model but false in any infinite model? False in any model with the wrong cardinality? (The answer is no.)

2. (a) Work out the logical language and standard model for the integers under addition.

   (b) Write a few sentences that you expect to be true and false; verify them in your intended model.

   (c) Which of these sentences are true in the model from the first exercise?

   (d) Can you write a sentence that will be true in your intended model but false in any finite model?

3. Prove Prop 7.10 (verify that substitution is associative).

4. Verify that if ϕ is a sentence then $\phi^{I,A} = \phi^{I,B}$ for any assignments A and B.

5. Let P be a unary relation symbol and c a constant. Demonstrate the validity of $(\forall x P(x)) \rightarrow P(c)$.

8 First-order Logic Continued

Exercise Postponed discussion of the Cubic SAT solver assignment.

1. What worked? What didn't?

2. Can someone describe designing a gadget to get a behavior? (any behavior is fine!)

3. Did anyone try large examples?

4. Did anyone attempt the "advanced" options? Algorithmic ideas to try when cubic is inconclusive? More aggressive termination when cubic learns all it can?

Last Time First-order logic:

- Motivation
- Syntax, including definition of substitution

- Semantics up to definitions of truth in a model, validity, satisfiability
- Please note the exercises added after last lecture. Key concepts include finding sentences that are only true when models have critical properties (such as begin finite or infinite).

This Lecture

- First-order tableau proof, take 1
- Building models out of syntax (Herbrand models)
- Highlights of classic results of model theory

8.1 A little proof theory

Before continuing with the model theory thread, we look at a version of the tableau proof method for first-order logic.

The basic idea of a first-order tableau proof is exactly the same as for propositional logic. Start with the negation of a formula as the root of a tree. Expand the tree according to the tableau proof rules in a manner that the set of branches is equisatisfiable with the root. The changes are, of course, that we now have relations applied to terms as atomic formulas and we have quantifiers.

Tableaux still close when an explicit contradiction is formed. Atomic closure will be closure by a contradiction built from atomic formulas.

Quantifiers What rules do we use to reduce quantifiers? Before introducing uniform notation we will look at the signed case. How do we expand a universally quantified formula is a tableau? If we followed the spirit of the reduction rules for the propositional case, we would replace the occurrence of $\forall x\, \phi(x)$ with every possible instance of ϕ. (The convention $\phi(x)$ indicates that ϕ is a formula in which x is free. The corresponding $\phi(t)$ represents the same term with the free occurrences of the variable replaced by the term t.) This immediately brings up two fundamental problems: (1) this is not a finite expansion in general, and (2) how do we name every possible instance?

For the first issue, while it is morally correct to give every instance, we really only need the set of instances that will lead us to a contradiction and let us close the branch for unsatisfiable branches. Anticipating that we only need finitely many instances in any particular proof we allow $\forall x\, \phi(x)$ to be instantiated any finite number of times. We do this by allowing $\forall x\, \phi(x)$ to be used any number of times, even in a "strict" tableau, and having each use introduce exactly one instance.

For the second problem, how do we name all values, we are forced to only consider values that are named by terms. Since proofs are fundamentally finite, syntactic objects, they will use the syntactic logical language to name the elements of the domain. Our proof theory must quantify over the terms of the object language; it cannot quantify directly over values in the meta-language.

The role of syntax is so important that in the next subsection we will show that we can always construct a model out of syntax.

The tableau rules for universal quantification (and its dual form) are:

$$\begin{array}{c} T\,\forall x\,\phi(x) \\ | \\ T\,\phi(t) \end{array} \qquad\qquad \begin{array}{c} F\,\exists x\,\phi(x) \\ | \\ F\,\phi(t) \end{array} \qquad\qquad \begin{array}{c} \gamma \\ | \\ \gamma(t) \end{array}$$

For any closed term t. (The left two forms are the signed forms, the rightmost is the uniform notation form of the rule.)

The next question is how do we reduce an existential quantifier (signed with true)? In this case it is fine to use the rule only once, as we did for strict expansion. But what term can we use to name the value that is promised to exist? We solve this problem by enriching our language with a countable set of new constants, called parameters. Each time we reduce an existential statement we consume one of these parameters, and assert the property of the parameter. This yields the signed tableau instances:

$$\begin{array}{c} T\,\exists x\,\phi(x) \\ | \\ T\,\phi(p) \end{array} \qquad\qquad \begin{array}{c} F\,\forall x\,\phi(x) \\ | \\ F\,\phi(p) \end{array} \qquad\qquad \begin{array}{c} \delta \\ | \\ \delta(p) \end{array}$$

For p a new parameter not occurring previously on the branch. (Signed and uniform versions are given as above)

An example Assume O is a binary relation symbol. We wish to prove the following sentence with a signed first-order tableau:

$$(\forall x\, O(x,x)) \rightarrow (\forall x \exists y\, O(x,y))$$

The proof begins by marking the goal false and applying the rule for a false implication:

$$\begin{array}{c} F\,(\forall x\, O(x,x)) \rightarrow (\forall x \exists y\, O(x,y)) \\ | \\ T\,\forall x\, O(x,x) \\ | \\ F\,\forall x \exists y\, O(x,y) \end{array}$$

We expand the last formula, introducing a new parameter a:

$$\begin{array}{c} F\,(\forall x\,O(x,x)) \to (\forall x \exists y\,O(x,y)) \\ | \\ T\,\forall x\,O(x,x) \\ | \\ F\,\forall x \exists y\,O(x,y) \\ | \\ F\,\exists y\,O(a,y) \end{array}$$

We again expand the last formula. Since it is an existential quantifier signed with false, we use the generalized universal rule, instantiating the variable y to the term a:

$$\begin{array}{c} F\,(\forall x\,O(x,x)) \to (\forall x \exists y\,O(x,y)) \\ | \\ T\,\forall x\,O(x,x) \\ | \\ F\,\forall x \exists y\,O(x,y) \\ | \\ F\,\exists y\,O(a,y) \\ | \\ F\,O(a,a) \end{array}$$

Finally, we use the second formula, instantiated on a as well to obtain the contradiction:

$$\begin{array}{c} F\,(\forall x\,O(x,x)) \to (\forall x \exists y\,O(x,y)) \\ | \\ T\,\forall x\,O(x,x) \\ | \\ F\,\forall x \exists y\,O(x,y) \\ | \\ F\,\exists y\,O(a,y) \\ | \\ F\,O(a,a) \\ | \\ T\,O(a,a) \\ | \\ \perp \end{array}$$

Uniform notation As in the propositional case there are basically two quantifier rules: generalized universal quantification, which introduces an instance with an arbitrary term, and generalized existential quantification, which introduces a new parameter. Smullyan names these patterns γ and δ.

γ	$\gamma(t)$	δ	$\delta(p)$
$\forall x\, \phi$	$\phi\{x/t\}$	$\exists x\, \phi$	$\phi\{x/p\}$
$\neg\exists x\, \phi$	$\neg\phi\{x/t\}$	$\neg\forall x\, \phi$	$\neg\phi\{x/p\}$

In this example we have seen one example of a finite proof of a true sentence. What happens if we try to prove something that isn't true? that is, what happens if the root is satisfiable? Assume we have a logical language for arithmetic that includes 0, a successor function S, and an equality predicate $=$. Consider a tableau with root $T\,\forall n\, 0 \neq S(n)$ (in official syntax: $T\,\forall n\, \neg = (0, S(n))$). How does this tableau expand? We can apply the root fact to an arbitrary number of terms, each time introducing a new true fact: $0 \neq S(0)$, $0 \neq S(S(0))$, $0 \neq S(S(S(0)))$, ... Given a first-order tableau with any γ nodes it will never "finish open" for a non-trivial domain. In the next section we return to the semantics of first-order logic. We will show if we allow these open paths to run to infinity they will give us models built out of syntax.

Formalizing parameters Fitting introduces notation for the extension of a logical language with parameters:

Definition 8.1 (Fitting 5.7.1) *Let $L(P, F, C)$ be a first-order language. Let par be a countable set of constant symbols disjoint from C. L^{par} is shorthand for $L(P, F, C \cup par)$.*

Key invariants As in the propositional case, the key invariant of tableau proofs is that when a path is expanded, satisfiability is preserved. This is captured in the following property:

Proposition 8.2 (Fitting 5.5.1) *Let S be a set of sentences, and γ and δ be sentences:*

1. *If $S \cup \{\gamma\}$ is satisfiable, so is $S \cup \{\gamma, \gamma(t)\}$ for any closed term t.*
2. *If $S \cup \{\delta\}$ is satisfiable, so is $S \cup \{\delta, \delta(p)\}$ for any constant symbol p that is new to S and δ.*

8.1.1 Exercises

1. Pick a handful of sentences from HR Exercises 2.3. You will need to convert sequents into implications. Complete the proofs in tableau form. I recommend starting with signed tableaux. Once the pattern used in the uniform notation becomes natural you may wish to switch to unsigned tableaux.

8.2 Building Models out of Syntax

We have established the syntax of first-order logic, we have developed enough semantics to define truth (and validity) of sentences, and we have sketched a simple proof system. In the proof system we saw that at some point the values we can name syntactically became more important than the values we intend to describe with our intended models. In this section we show that given any first-order logical language we can always build a model directly out of syntax, called an Herbrand model. Such models are always countable.

Before getting to the definition of Herbrand models, we first revisit two concepts from the last lecture: substitutions and assignments. Recall that a substitution was a map from variables to terms, and an assignment was a map from variables to values in the model. We expect, in general these two maps to relate closely. In particular, sometimes we will want to manipulate terms syntactically, and understand the semantic consequences, and sometimes we will want to lift semantic relationships to syntactic ones.

The first proposition says that if we are substituting a closed term for a variable we can either do the substitution and then calculate the interpretation, or define a new assignment based on the interpretation of the term, and calculate the meaning with that assignment.

Proposition 8.3 (Fitting 5.3.7) *Suppose: L is a first-order language, t is a closed term, ϕ is a formula, and $M = \langle D, I\rangle$ is a model for L. Let x be a variable and A be such that $x^A = t^I$. Then:*

$$[\phi\{x/t\}]^{I,A} = \phi^{I,A}$$

More generally:

$$[\phi\{x/t\}]^{I,B} = \phi^{I,A} \quad \text{for any } x\text{-variant } B \text{ of } A$$

The next proposition generalizes this from a single term to an arbitrary substitution:

Proposition 8.4 (Fitting 5.3.8) *Suppose: L is a first-order language, $M = \langle D, I\rangle$ is a model, ϕ is a formula, A is an assignment, and σ is a substitution free for ϕ. Define B by setting, for each v, $v^B = (v\sigma)^{I,A}$. Then $\phi^{I,B} = (\phi\sigma)^{I,A}$.*

8.2.1 Herbrand models

In the last two propositions we saw that assignments and substitutions are closely related. If we build our model directly out of syntax, we can make them essentially the same.

Definition 8.5 (Fitting 5.4.1) *A model $M = \langle D, I\rangle$ for the language L is an* Herbrand model *if:*

1. *D is exactly the set of closed terms of L.*

2. *For each closed term t, $t^I = t$.*

In an Herbrand model assignments are just a special case of substitutions where variables are mapped to closed terms.

Proposition 8.6 (Fitting 5.4.2, 5.4.3) *Suppose $M = \langle D, I \rangle$ is an Herbrand model for the language L.*

1. *For any term t of L, $t^{I,A} = (tA)^I$. (t is not necessarily closed.)*
2. *For a formula ϕ of L, $\phi^{I,A} = (\phi A)^I$.*

For Herbrand models quantifier truth is a property of the set of all closed terms.

This allows the key invariant of the tableau proofs (Prop 8.2) to be restated more simply:

Proposition 8.7 (Fitting 5.5.2) *Suppose $M = \langle D, I \rangle$ is Herbrand*

1. *γ is true in M if and only if $\gamma(d)$ is true in M for every $d \in D$.*
2. *δ is true in M if and only if $\delta(d)$ is true in M for some $d \in D$.*

8.2.2 Hintikka's Lemma

As in the propositional case we use results due to Hintikka to construct a framework for completeness. In the propositional case these tools were more powerful than what was required. In the first-order case these tools are much more appropriate to the task.

The first step is to revise the definition of a Hintikka set (Definition 4.1).

Definition 8.8 (Hintikka Set, Fitting 5.6.1) *A set H is a* first-order Hintikka set *provided*

1. *For atomic formula ϕ at most one of ϕ or $\neg\phi$ is in H.*
2. *$\bot \notin H$, $\neg\top \notin H$.*
3. *If $\neg\neg\phi \in H$ then $\phi \in H$.*
4. *If $\alpha \in H$ then $\alpha_1 \in H$ and $\alpha_2 \in H$.*
5. *If $\beta \in H$ then $\beta_1 \in H$ or $\beta_2 \in H$.*
6. *If $\gamma \in H$ then $\gamma(t) \in H$ for every closed term t of L.*
7. *If $\delta \in H$ then $\delta(t) \in H$ for some closed term t of L.*

Note that the rule for γ formulas implements the intuition we had when developing tableau proofs. If a γ formula is in the set then we add every instance, which in logics with an infinite set of closed terms (almost all logics) will be infinite.

As in the propositional case, first-order Hintikka sets have enough information in them to construct a model in which every element of the set is satisfiable. This is captured by the following proposition:

Proposition 8.9 (Hintikka's Lemma, Fitting 5.6.2) *Suppose L is a language with a nonempty set of closed terms. If H is a first-order Hintikka set with respect to L, then H is satisfiable in an Herbrand model.*

Hintikka sets will again be the limiting case of proof search. To show a proof system complete we will show any proof process that cannot construct a proof, will, in the limit, construct a Hintikka set. Thus the only unprovable things will be things that can be satisfied in an Herbrand model.

While developing this machinery, we will also obtain several important results that are key to characterizing the limits of first-order logic.

Definition 8.10 (Fitting 5.8.1) *Given:*

- *L a first-order language*
- *L^{par} the extension of L with parameters*
- *$\mathcal{C}$ a collection of sets of sentences of L^{par}*

$\mathcal{C}$ is a first-order consistency property *if, for each $S \in \mathcal{C}$:*

1. *For every atomic proposition ϕ at most one of ϕ or $\neg\phi$ is in S.*
2. *$\bot \notin S$, $\neg\top \notin S$.*
3. *If $\neg\neg\phi \in S$ then $S \cup \{\phi\} \in \mathcal{C}$.*
4. *If $\alpha \in S$ then $S \cup \{\alpha_1, \alpha_2\} \in \mathcal{C}$.*
5. *If $\beta \in S$ then $S \cup \{\beta_1\} \in \mathcal{C}$ or $S \cup \{\beta_2\} \in \mathcal{C}$.*
6. *If $\gamma \in S$ then $S \cup \{\gamma(t)\} \in \mathcal{C}$ for every closed term t of L^{par}.*
7. *If $\delta \in S$ then $S \cup \{\delta(p)\} \in \mathcal{C}$ for some parameter p of L^{par}.*

Theorem 8.11 (First-order Model Existence, Fitting 5.8.2) *If $\mathcal{C}$ is a first-order consistency property with respect to L, S is a set of sentences of L, and $S \in \mathcal{C}$, then S is satisfiable; in fact S is satisfiable in an Herbrand model (but Herbrand with respect to L^{par}.*

Fitting sketches a proof of this. The proof requires an alternate version of the consistency property to deal with technical details having to do with parameter management.

8.2.3 Classic results from model theory

The following fundamental results of model theory are all consequences of Fitting's proof of the model existence theorem:

Theorem 8.12 (First-order Compactness, Fitting 5.9.1) *Let S be a set of sentences of the first-order language L. If every finite subset of S is satisfiable, so is S.*

Theorem 8.13 (Löwenheim-Skolem, Fitting 5.9.3) *Let L be a first-order language, and let S be a set of sentences of L. If S is satisfiable, then S is satisfiable in a countable model.*

What does this say about an axiomatization of the real numbers?

Theorem 8.14 (Herbrand Model, Fitting 5.9.4) *A set S of sentences of L is satisfiable if and only if it is satisfiable in a model that is Herbrand with respect to L^{par}. A sentence ϕ of L is valid if and only if ϕ is true in all models that are Herbrand with respect to L^{par}.*

The take home messages This lecture is a bit of a teaser for many of the most exciting results that would be featured in a traditional course on mathematical logic. Given the emphasis on algorithms in this class, we are not assigning the math exercises necessary to build mastery of this material. However, it is important for students to be aware of these results and have confidence that, given time and motivation, mastery is well within reach.

- Read Fitting! It's beautiful!
- Models built out of syntax play a critical role in the meta-theory of first-order logic. Even when the intended model is uncountable, it is sufficient to consider only countable models in determining truth and validity.
- There is a good match between the meta-theory of first-order logic and the fundamental mechanisms of symbolic computation.

9 Proof Systems for First-order Logic

Last time:

- Defined tableau proofs for FOL
- Defined Hebrand models

This time:

1. Finish up topics from last section including Hintikka's lemma, Model existance theorem, and compactness
2. Introduce other proof procedures for first-order logic

9.1 Natural deduction

In Section 8.1 we gave tableau proof rules for First-order logic. In general, tableau rules expose essential aspects of the syntactic construction of models. They don't provide much direct insight into how these proof concepts are manifest in mathematical discourse. For this purpose, natural deduction is the most illuminating system to consider.

The natural deduction framework asks us to identify how to introduce and eliminate each logical concept. Understanding quantifier reasoning then reduces to asking the following questions:

1. How to introduce all?
2. How to eliminate all?
3. How to introduce exists?
4. How to eliminate exists?

We consider each in turn.

All introduction When can we conclude $\forall x\, \phi$? In a simple argument we might assert that "every natural number is either even or odd". We would support this with an argument such as:

> Consider an arbitrary natural number n, if n is divisible by 2 then n is even. If n is not divisible by 2 then $n = 2 * k + 1$ for some k, and we conclude n is odd.

This argument can be safely generalized from an argument about the specific symbol n to an argument about all natural numbers because we assumed no specific properties of n other than those that can be ascribed to all natural numbers, such as $n = 2 * k + r$ for $0 \leq r < 2$.

This pattern of safe generalization is captured in the following rule, and the critical side condition:

$$\frac{\phi}{\forall x\, \phi} \quad \text{provided } x \text{ does not occur free in any premise}$$

This rule allows proof of $\forall x.even(x) \to odd(x{+}1)$ but disallows the (nonsensical) $even(x) \to \forall x.even(x)$.

All elimination The elimination rule is essentially identical to that for tableau, if we know something is true of every element of the domain, we can instantiate it on any term we choose:

$$\frac{\forall x\, \phi}{\phi(t)} \quad \text{where } t \text{ is free for } x \text{ in } \phi$$

Exists introduction The most direct way to show that an existential statement is true is to show that it holds for some element in the domain. This is captured in the rule:

$$\frac{\phi(t)}{\exists x\, \phi(x)} \quad \text{where } t \text{ is free for } x \text{ in } \phi$$

Exists elimination It is natural to view universal quantification as a generalization of conjunction and existential quantification as a generalization of disjunction. This is a good model for understanding existential elimination. Recall that in $\vee$ elimination we had three proof obligations, we had to prove $\phi \vee \psi$, and we had to hypothetically prove χ from either ϕ or ψ. In the end we could conclude χ. Now instead of having two alternatives we have infinitely many: one for each possible term in the logic. Since it is impractical to write down infinitely many hypothetical arguments, we use the same trick we used in all introduction to collapse them into a single argument about a variable that is otherwise new to the argument. This gives the rule:

$$\frac{\exists x\, \phi(x) \qquad \begin{matrix}[\phi(x)]\\ \vdots \\ \chi\end{matrix}}{\chi} \quad \text{provided } x \text{ does not occur free in any premise}$$

Presentation in HR and Fitting Huth and Ryan present essentially these rules. They draw boxes around the hypothetical proofs in all introduction and exists elimination. Within the boxes they indicate the fresh variable that is only to be used within that scope. If you are doing proofs by hand I recommend following their conventions; they help keep you honest.

Fitting points out that the proof system is complete without the introduction rules. He gives only the elimination rules. He gives an alternate formulation that uses parameters. His elimination rules (in uniform notation) are:

$$\frac{\gamma}{\gamma(t)} \qquad \frac{\delta}{\delta(p)}$$

Where t is closed over L^{par} and p is a new parameter.

9.2 Sequent Calculus

In Section 4.2 we introduced Gentzen's sequent calculus for classical propositional logic. That logic is implemented in one of the backwards provers provided the code repository.

In the sequent calculus instead of having the introduction/elimination pairing of rules we have rules acting on the left and right of the sequent.

All Right The all on the right rule corresponds very closely to all introduction in natural deduction. Given an appropriately generic argument, we can generalize it:

$$\frac{\Gamma \vdash \phi[y/x], \Delta}{\Gamma \vdash \forall x\, \phi, \Delta} \quad \text{provided } y \text{ is not free in } \Gamma,\ \phi, \text{ or } \Delta$$

All Left The all on the left rule corresponds to all elimination. If we already have a universal statement as a hypothesis, we can add an instance of that statement to the set of hypotheses.

$$\frac{\Gamma, \phi[t/x] \vdash \Delta}{\Gamma, \forall x\, \phi \vdash \Delta}$$

Exists Again, the duality between all and exists is apparent in the similarity between the right rule for exists and the left rule for all, and v.v.

$$\frac{\Gamma, \phi[y/x] \vdash \Delta}{\Gamma, \exists x\, \phi \vdash \Delta} \quad \frac{\Gamma \vdash \phi[t/x], \Delta}{\Gamma \vdash \exists x\, \phi, \Delta}$$

Side condidtions as for corresponding all rules.

Fitting presents an alternate formulation in uniform notation using closed terms and parameters.

$$\frac{\Gamma, \gamma(t) \vdash \Delta}{\Gamma, \gamma \vdash \Delta} \quad \frac{\Gamma \vdash \gamma(p), \Delta}{\Gamma \vdash \gamma, \Delta}$$

$$\frac{\Gamma \vdash \delta(t), \Delta}{\Gamma \vdash \delta, \Delta} \quad \frac{\Gamma, \delta(p) \vdash \Delta}{\Gamma, \delta \vdash \Delta}$$

9.3 Soundness and Completeness of Tableau

Follow Fitting Sections 6.3 and 6.4.

10 Automating Logic

Exercise:

- How did it go?
- Examples of proofs completed.
- Did anyone write any tactics?
- What felt most awkward?

Last time:

- Hebrand models
- Hintikka's lemma
- Model Existence Theorem
- Basic results in proof theory
-

This time:

- Discuss natural deduction and sequent proof rules.
- Discuss soundness and completeness
- Quantifier elimination and Skolemization
- Unification
- Revisiting proof theory with Skolemization

10.1 Skolemization

In the proof systems that we have presented we have been very explicit about "conditions on variables" in quantifier rules. We have required that parameters be fresh in tableau proofs. We have required that variables not occurr in other terms in the universal generalization rules in sequent calculus.

When we automate these procedures, it is natural to try to use computation to build the terms that are needed to carry out proofs. The primary tool we will use to automate construction is unification. Through unification we will construct substitutions that we will apply globally to proofs. In effect unification will allow the prover to delay the choice of terms as long as possible.

Before we introduce unification, we need to look at a concept that comes out of a process originally designed to eliminate quantifiers in formulas, called Skolemization.

In the tableau proofs, when we had an existential sentence with a positive sign weintroduced a new parameter, p, to witness its truth. In the soundness proof we observed that $\exists x\, \phi(x)$ is equisatisifable with $\phi(p)$ for p an fresh parameter.

Skolemization is conceptually a similar. But it is going to rewrite in a more general context. It is going to try to completely eliminate the existential from the sentence. To do this it replaces the parameter p with a function f from the things on which it depends.

For a motivating example, we will temporarily leave the context of strict FOL logic and recall a familiar result from Automata theory. Recall the quantifier structure of the pumping lemma for finite automata: For every language, there is a pumping length, such that for every string, if the string is long enough there is a decomposition of the string into components satisfying the condidions. Symbolically this is:

$$\forall L \exists p \forall s. |s| \geq p \to \exists u, v, w. \ldots (u, v, w, s, p, L) \ldots$$

If we eliminate the inner $\exists(u, v, w)$ and replace it by a magic function, what arguments does that function intuitively require? The choice depends on the langauge L, the pumping length p, and the string s. So we could imagine a triple of functions $g_u(L, p, s), g_v(L, p, s)$, and $g_w(L, p, s)$ that return the strings u, v and w. Similarly, if we eliminate the $\exists p$ in the above by introducing a magic function, what parameters should that function have? Obviously, the choice of the pumping length morally depends on the language, so we might replace p with $f(L)$.

Using these magic functions we construct an alternative PL:

$$\forall L. \forall s. |s| \geq f(L) \to \ldots . (g_u(L, f(L), s), g_v(L, f(L), s), g_w(L, f(L), s), s, f(L), L) \ldots$$

While linguistically quite different, this sentence is equisatisfiable with the pumping lemma if we give the interpretation the same kind of freedom to pick the new functions as it was given to pick parameters. The big difference is that we can make this change inside a formula. We could only introduce parameters when we had a quantifier in the outermost position of a sentence.

The basic idea is that when we have an existential sentence, like $\exists x\, \phi(x)$ we can view the existential witness as being computed by a new function, called a Skolem function. The Skolem function is given arguments correpsonding to the free variables of ϕ. This transformation replaces $\exists x\, \phi(x)$ with $\phi(f(y_1, \ldots, y_k))$ where $y_1, \ldots, y_k$ include all free variables of ϕ distinct from x.

Quantifier elimination can be continued, by allowing top level universal quantifiers to be eliminated, and interpreting free variables as implicitly universally quantified. Continuing the example, this would give an open formula:

$$|s| \geq f(L) \to \ldots . (g_u(L, f(L), s), g_v(L, f(L), s), g_w(L, f(L), s), s, f(L), L) \ldots$$

This open formula is again equisatisfiable with the original sentence describing the pumping lemma.

Together these mechanims allow a sentence to be translated into an equisatisifable open formula in which we have introduced new function symbols, called Skolem functions, and interpret free variables as universally quantified. Details justifying this development can be found in Fitting Chapter 8.

With these mechanisms, we can revisit the basic tableau rules. An alternative version can maintain a path as a set of implicitly universally quantified open formulas. In that version the γ rule introduces a new unbound variable and the δ rule introduces a new Skolem function parameterized by the free variables of δ:

$$\frac{\gamma}{\gamma(x)} \qquad \frac{\delta}{\delta(f(x_1, \ldots, x_n))}$$

Where x is a fresh variable, $x_1, \ldots, x_n$ contain all free variables of δ, and f is a new Skolem function.

Using this form of the rules we revisit an example:

$$\begin{array}{c} F\,(\forall x\, O(x,x)) \rightarrow (\forall x \exists y\, O(x,y)) \\ | \\ T\, \forall x\, O(x,x) \\ | \\ F\, \forall x \exists y\, O(x,y) \end{array}$$

Since we aren't going to commit to a value for the universal variable, we do not need to postpone its introduction. We can reduce the second line:

$$\begin{array}{c} F\,(\forall x\, O(x,x)) \rightarrow (\forall x \exists y\, O(x,y)) \\ | \\ T\, \forall x\, O(x,x) \\ | \\ F\, \forall x \exists y\, O(x,y) \\ | \\ T\, O(a,a) \end{array}$$

Then we can do the third line. Since there are no free variables in the formula we get a 0-ary Skolem function:

$$\begin{array}{c} F\,(\forall x\, O(x,x)) \rightarrow (\forall x \exists y\, O(x,y)) \\ | \\ T\, \forall x\, O(x,x) \\ | \\ F\, \forall x \exists y\, O(x,y) \\ | \\ T\, O(a,a) \\ | \\ F\, \exists y\, O(f_1(), y) \end{array}$$

We do a γ reduction on the last formula:

$$\begin{tikzcd}
F\,(\forall x\, O(x,x)) \to (\forall x \exists y\, O(x,y)) \arrow[d, dash] \\
T\,\forall x\, O(x,x) \arrow[d, dash] \\
F\,\forall x \exists y\, O(x,y) \arrow[d, dash] \\
T\,O(a,a) \arrow[d, dash] \\
F\,\exists y\, O(f_1(),y) \arrow[d, dash] \\
F\,O(f_1(),b)
\end{tikzcd}$$

Are we there yet? Consider the substitution that maps a and b to $f_1()$. Under that substitution this tableau becomes:

$$\begin{tikzcd}
F\,(\forall x\, O(x,x)) \to (\forall x \exists y\, O(x,y)) \arrow[d, dash] \\
T\,\forall x\, O(x,x) \arrow[d, dash] \\
F\,\forall x \exists y\, O(x,y) \arrow[d, dash] \\
T\,O(f_1(),f_1()) \arrow[d, dash] \\
F\,\exists y\, O(f_1(),y) \arrow[d, dash] \\
F\,O(f_1(),f_1())
\end{tikzcd}$$

Which is contradictory.

Is it ok to declare a branch contradictory if a substitution instance of it is contradictory?

Short answer: yes.

What do we need to set up this answer? When we interpret open formulas as implicitly universally quantified a set of open formulas will be true if it is true under all assignments. In the Herbrand model this will be truth under all substitutions for closed terms (over a logical language extended with Skolem functions).

10.2 Unification

The key to automating the tableau method using open formulas will be calculating substitutions. We need a few more ideas from the algebra of substitutions, then we can give an algorithm that helps us find particularly useful substitutions.

Definition 10.1 (Fitting 7.2.1) *The substitution σ_2 is* more general *than the substitution σ_1 if, for some substitution τ, $\sigma_1 = \sigma_2\tau$.*

Prop: more general is a transitive relation

Definition 10.2

- *Substitution σ is a* unifier *of t_1 and t_2 if $t_1\sigma = t_2\sigma$.*
- *t_1 and t_2 are* unifiable *if a unifer exists.*
- *σ is a* most general unifier *if it is a unifier and is more general than any other unifier.*

Most general unifiers are unique up to variable renaming.
How to calculate unifiers?
Find a disagreement pair.
Pseudocode following Fitting:

```
• Set $\sigma$ to be the identity
• While $t_1\sigma \neq t_2\sigma$ do
    1. choose a disagreement pair, $d_1, d_2$ for $t_1\sigma$, $t_2\sigma$.
    2. If neither $d_1$ nor $d_2$ is a variable then fail.
    3. Let $x$ be whichever of $d_1$ or $d_2$ is a variable (pick one if both are), let
       $t$ be the other one of $d_1$ or $d_2$.
    4. If $x$ occurs in $t$ then fail (occurs check).
    5. Set $\sigma$ to $\sigma\{x/t\}$
```

10.3 Problem Set

1. Implement unification on the first-order logic framework used in the exercise for LK.hs.

   In my implementation I have the following functions (the first two are mutually recursive):

   ```
   unify :: (Eq f, Eq v) => Term f v -> Term f v -> Maybe (Subst v (Term f))
   unifyLists :: (Eq f, Eq v) => [Term f v] -> [Term f v] -> Maybe (Subst v (Term f))
   occurs :: Eq v => v -> Term f v -> Maybe ()
   ```

2. Using unification implement a tableau prover for FOL. If you want your prover to terminate in all cases you will need to provide an argument to control the number of times it will apply a γ rule.

11 Higher-order Logic

11.1 Motivation

In first-order logic quantification is over individuals, which are elements of the universe, and named by terms of the logical language. In second-order logic, quantification and abstraction are extended from individuals to properties of individuals.

For example, consider an inductive definition of lists:

1. The empty list is a list.
2. If l is a lists and a is and element then $a : l$ is the list with head a and tail l.
3. Nothing else is a list.

Such a definition gives an induction principle that lets us prove properties about lists:

> To show that property P holds for all lists show:
> 1. P holds of the empty list.
> 2. For any l and a, $P(l)$ implies $P(l : a)$.

This induction principle can either be seen as a reasoning schema that can be added to a first-order language and instantiated separately for every property, or as a single sentence in a higher-order language. In particular, if the symbol $\mathbf{o}$ is used to represent the type of propositions (or propositional truth values), the type of a predicate over values of type a can be written as a function $a \to \mathbf{o}$. This lets the above induction principle be written as:

$$\forall P : List\, a \to \mathbf{o}.P(nil) \wedge (\forall x : a, l : List\, a.P(l) \to P(a : l)) \to \forall l : List\, a.P(l)$$

Abstraction over properties is natural in mathematics. For example, it is common to talk about properties of an equivalence relation. Using higher-order concepts it is easy to talk about a relation R of type $a \to a \to \mathbf{o}$ that is reflexive:

$$REF \equiv \lambda R : a \to a \to \mathbf{o}.\forall x : a.R\,x\,x$$

Exercise

- Work out the remainder of an equivalence relation.
- Define equivalence classes, partitions, and the quotent construction for arbitrary equivalence relations.

Quantification over properties can be used to define new properties. For example, Liebnitz identity is defined in words as "two objects are identical if no property distinguishes them". In some higher-order logics this can be expressed directly as:

$$Q\,x\,y \equiv \forall P : a \to \mathbf{o}.P(x) \to P(y)$$

Note that clearly Q is reflexive, as any property P of x holds of x. For symmetry the argument is a little more interesting. From $P(x) \to P(y)$ you need to show $Q\,y\,x$. Well, consider the property $\lambda z.Qzx$. Clearly this holds of x by reflexivity. Hence it holds of y by assumption, establishing $Q\,y\,x$ as required.

The proof of transitivity is left as an exercise.

The symmetry proof illustrates a kind of near circularity that can arrise in higher-order settings. The property of being equal is defined in terms of quantification over *all properties*, including properties defined in terms of the property of being equal. Such definitions are called impredicative. [See Stanford article on type theory, Section 3]

Definition 11.1 *A definition is* impredicative *if it defines a collection of which the defined element is a member. A definition is* predicative *if it is defined in terms of a collection that is defined independtly of the thing being defined.*

Example, attributed to Russell:
Preicative: Napolean was Corsican.
Impredicative: Napolean had all the qualities of a great general.

11.2 The Paradoxes

Gotlieb Frege was the first to characterize the language of mathematics as a formal language. He was the first to identify quantified variables. His program of formalization was profound and revolutionary. It included both clear formulation of the language, a contribution which persists, and an attempt to give an underlying logical system from which all mathematical knowledge was a logical consequence. Unfortunately, his logical foundations were flawed. When his manuscript presenting the logical foundations was in the final statges or preparation for publication he received a letter from Russell showing his system was inconsistent.

One of the principles of Frege's system was extensionality. Two functions are the same if they are the same on all values. The formulation of extensionality was such that for any predicate, P, it was possible to get the extension of the predicate, that is $\{x|P(x)\}$. Russell's paradox exploits extensionality.

11.2.1 Russell's paradox

In words, Russell's paradox is generally characterized something like:

> Let S be the set of all sets that do not contain themselves. Is S an element of S? If S is not in S then it meets the criteria of belonging to S. If S is in S then it fails to meet the criteria of membership. Since both possibilities lead to a contradiction, any

> system admitting the statement of the paradox must be logically inconsistent.

Symbolically it is even easier. The formula $x \notin x$ is easily seen as a predicate on x, so by extensionality, $S = \{x | x \notin x\}$ defines a set S. If $S \in S$ then it follows that $S \notin S$, which contradicts the assumption. If $S \notin S$, then clearly $S \in S$. Having shown that both a formula and its negation lead to a contradiction we have shown the inconsistency of the system.

This paradox is simple and clear. So what went wrong? Is it the impredicative nature of the definition of S? Clearly the pardox is obtained by applying the defining property to the set defined. Is the power of Fege's extensionality letting us pass from a predicate to a set?

Frege's system was not easily repaired. Frege's attempt at repair while the manuscript was in press was fundamentally flawed; it trivialized his theory. [See wikipedia article.]

In an appendix to his monograph *Principles of Mathematics* Russell, first published in 1902, Russell sketches his "Doctrine of Types" as a potential solution to the paradox. In the preface to the second edition (published 34 years after the first) he writes:

> In the "Principles," only three contradiction are mentioned .... What is said as to possible solutions may be ignored, except Appendix B, on the theory of types; and this itself is only a rough sketch. ...
>
> The techncial essence of the theory of types is merely this: Given a propositinoal function "ϕx" of which all values are true, there are expressions which are not legitimate to substitue for "x." For example: All values of "if x is a man x is a mortal" are true, and we can infer "if Socrates is a man, Socrates is a mortal"; but we cannot infer "if the law of contradiction is a man, the law of contradiction is a mortal." The theory of types declares this latter set of words to be nonsense, and gives rules as to permissible values of "x" in "ϕx." In the detail there are difficulties and complications, but the general principle is merely a more precise form of one that has always been recognized. ...Thus, for example I stated above that "classes of things are not things." This will mean: "If 'x is a member of the class α' is a proposition, and 'ϕx' is proposition, then '$\phi\alpha$ is not a proposition, but a meaningless collection of symbols."

In modern set theory (ZF) Russell's paradox is avoided by distinguishing between sets, which can be built from other sets using predicates called comprehensions, and classes, which are collections that are not necessarily sets. There is a class of all sets, but there is not a set of all sets. Given a set of sets, S, we can build the set of sets from S that contain themselves and the set of sets that do not contain themselves, but since this process is with respect to the specific collection S it is predicative and no contradiction arrises.

11.2.2 Burali Forti paradox

The second paradox that we study preceeded Russell's paradox historically. I will give a modern statement of the Burali Forti paradox.

An *ordinal number* can be represented in set theory by the set of numbers that precede it. For example, the least ordinal number is ϕ, the empty set. The next are $\{\phi\}, \{\phi, \{\phi\}\}, \ldots$.

Given an ordinal number X it is possible to build it successor as $X \cup \{X\}$. Starting from ϕ we can construct ordinal numbers corresponding to all the natural numbers using this successor operation.

Note that this definition allows for ordinal numbers that are not built with successor. Consider the union of the ordinal numbers corresponding to natural numbers described above. This is an infinite union, but well defined. The orginal corresponding to every natural number is in this set, which we call ω, but the set itself has no greatest element. Why? if some finite natural number n was the greatest element then there would be a natural number $n + 1$ not included.

What if we apply the successor construction to the ordinal ω? We get a new ordinal number, greater than ω, which we call $\omega + 1$.

One of the key properties of ordinal numbers is that there is no greatest ordinal number. Given any ordinal number we can apply the successor construction and get a greater one.

Consider the set of all ordinal numbers. Since it is a set of ordinal numbers, it is itself an ordinal number. Since it contains all ordinal numbers it must be the greatest ordinal number. But that cannot be, as there is no greatest ordinal number.

This is the paradox.

This paradox is a little more subtle than Russell's, but has some of the same issues. Extensionality plays a role: the paradox requires that we be able to go from a property—being an ordinal number—to a set.

The paradox is avoided in ZF set theory by the same mechanism as Russell's paradox. The collection of all ordinal numbers is not deemed a set. Hence it cannot be an ordinal number.

`http://plato.stanford.edu/entries/paradoxes-contemporary-logic`

The Burali Forti paradox is the inspiration for Girard's paradox. Girard used this paradox to show the inconsistency of a predicative type theory developed by Martin-Löf in the early 1970's.

11.3 Church's type theory

Good references:

- `http://plato.stanford.edu/entries/type-theory/`
- `http://plato.stanford.edu/entries/type-theory-church/`

Church and Curry developed similar type systems based on the lambda calculus. These languages begin with first-order concepts, then extend them in ways that introduce limited higher-order concepts. In later sections we will see subsequent generalizations, some of which are sound and some of which are not. Church's notation was significantly more explicit than Curry's. In Church's notation, every symbol was subscripted by its type. In what follows I typically use Curry's more implicit notation, as it is easier for modern functional programmers to read.

Types

1. $\imath$ is the type of individuals (other domain-specific base types can be added as well, e.g. σ for natural numbers)
2. $\mathbf{o}$ is the type of truth values (used in propositions)
3. $\alpha \to \beta$ is the type symbol denoting functions from type α to type β (in Church's notation $(\beta\alpha)$).

Terms Church's language of terms gives a typed lambda calculus in which variables are indexed explicitly by types. He includes the following:

$$\begin{array}{rcl} \neg & : & \mathbf{o} \to \mathbf{o} \\ \vee & : & \mathbf{o} \to \mathbf{o} \to \mathbf{o} \\ \Pi & : & (\alpha \to \mathbf{o}) \to \mathbf{o} \\ \iota & : & (\alpha \to \mathbf{o}) \to \alpha \end{array}$$

Church defines a set of "well formed formulas of type α", which we would recognize as well-typed terms of type α.

The logical constants Π and ι are very interesting. Π is used to encode universal quantification. What we would have written as $\forall x.\phi(x)$ gets written as $\Pi(\lambda x.\phi(x))$. We now call this Higher-order abstract syntax. It was introduced by Church in 1940.

The operator ι is called a description operator. It selects an element that satisfies a property, if such an element exists. (See the axiom of descriptons in the Stanford article.)

The mechanisms provided are sufficient to define the Liebnitz identity discussed above.

$$Q : \alpha \to \alpha \to \mathbf{o} = \lambda x.\lambda y.\forall f : \alpha \to \mathbf{o}.f\,x \supset f\,y$$

Metatheory The λ-conversion aspects of Church's type system are normalizaing, that is every well formed formula of type α is equivalent to one in β-normal form. Further more, it is strongly normalizing, because every sequence of contractions is finite. In addition, since reduction is Church-Rosser, normal forms are unique up to the renaming of bound variables.

11.4 Adding Universes

Martin-Löf impredicative.

Added a universe, U. All types are in the universe, including the universe.

Girard demonstrated this system was inconsistent by constructing a version of the Burali Forti paradox known as Giard's paradox.

Two solutions:

Girard: System F, remain impredicative, but restrict universes.

Martin-Löf: Go to a predicative system with a hierarchy of universes.

11.5 Martin-Löf's Intuitionistic Type Theory

Hierarchy of universes.

11.6 Girard's System F

Add a universe U for all types, but do not include U in U.

Introduce explicit abstraction and quantification over types.

12 Propositions as Types

12.1 Prawitz Inversion Principle

Recall natural deduction. Rules came in introduction/elimination pairs.

Consider a transformation on proofs that rewrites proofs that have an introduction rule followed immediately by an elimination rule. From such a proof a more direct proof can be calculated, possibly duplicating parts of the proof.

This works very naturally for the minimal and intuitionistic natural deduction systems.

Using this transformation, proofs can be normalized. That is, they can be rewritten by a finite number of applications of these rules into a normal form.

This normalization process is a form of computation. Consider implication. The hypothetical proof constructed in the implies introduction form is treated as a process to transform a proof of the hypothesis into a proof of the conclusion. If we view a proof of a proposition A as the construction for some abstract evidence for A, then a proof of an implication, $A \rightarrow B$, becomes a transformation of evidence for A into evidence for B. That is, proof of an implication is a computational function.

Pushing this further, we find a strong correspondence between proofs of impilcations and functions, proofs of conjunctions and pairing, and proofs of disjunctions and disjoint union. This correspondence leads to a framework where we represent proofs as computational terms in the *lambda*-calculus and propositions as the types of the propositions. This is called the Curry-Howard correspondence.

12.2 Curry Howard

From the point-of-view of propositions as types, we revisit Church's logic. We no longer have separate logical constants for the logical connectives. Instead we have new type constructors corresponding to the connectives.

Work some examples.

What logic does this look the most like? A simple core functional language with ML polymorphism looks like propositional logic.

What are some of the consequences of the correspondence?

1. The existance of normal forms for proofs corresponds to computational normal forms for typed terms.

2. If we use an untyped language to encode proofs and perform ML-style type inference, what does that mean in the proof view?

How do we add predicates? Need sorts corresponding to individuals and sorts corresponding to k-ary predicates. If we bolt on an external term language, much as we did in first-order logic, we can make this work. We end up adding to the language of propositions quantification over k-ary predicates. We then need a language to name individuals, a language to construct k-ary predicates, and a language of proofs and propositions.

In this case, we end up with a simple Pi-type quantifying over k-ary predicates.

12.3 Generalizing Curry-Howard

Can we unify these languages? If we use computational terms to name individuals, construct predicates, and represent proofs, we will need to have types that can express well-formed individuals, well-formed predicates, and the proposition proved by a term.

Suppose we had the ability to say:

```
0 : Nat
S : Nat -> Nat
+ : Nat -> Nat -> Nat

= : Nat -> Nat -> Prop
even : Nat -> Prop
odd : Nat -> Prop

OR  : Prop -> Prop -> Prop
```

What does it take to say that every number is either even or odd?

We need something like the following to be a type:

```
ALL n : Nat . OR (even n) (odd n)
```

How strange is this type? Types depend on terms (even depends on n). Types are quantifying over terms.

Calculus of constructions builds a formal system that attempts to make all these distinctions.

Topics to follow up:

Pure Type Systems Pure type systems charaterize the design space of higher-order logics in terms of the potential kinds of dependencies they exhibit.

Logical Frameworks One particularly interesting point in the design space is the class of type systems called Logical Frameworks. These are higher-order systems that support the encoding of logical systems.

They are sometimes characterized by "judgements are types," that is, the logical judgements of a a logic are encoded into the type system.

13 Model Checking: Motivation and Tools

Last time: Tim is preparing some examples to follow up on dangling issues from the last lecture on Propositions-as-types.

This unit: In the next two lectures we will touch the surface of model checking. The Huth and Ryan text is good for motivation, applications, and sketching some algorithms. The Clarke, Grumberg and Peled book titled *Model Checking* gives a more complete foundational treatment of the underlying mathematics.

13.1 The Motivating Problem

Model checking, at the highest level, takes a specific state transition system (the model) and a property of the state transitions and answers the question does the property hold of the transition system. The transition system is specified as a set of states, a transition relation, and a labeling of the states with properties drawn from a fixed set of atomic formulas. This structure is called a Kripke structure.

The property is specified in a temporal logic. That is, it is specified in a logic that can express properties that vary over time. The design space of temporal logics is large. We will focus on three temporal logics:

LTL Linear-time Temporal Logic (LTL) models a sequence of states extending indefinitely into the future.

CTL Computation Tree Logic (CTL) models a tree of possible futures.

CTL* Is a generalization of both LTL and CTL.

LTL and CTL are both capable of expressing properties that cannot be expressed in the other. They are also both computationally more tractable than CTL*.

13.1.1 Example: Mutual Exclusion

Following Clarke, et al.:

```
P = m : cobegin P0 || P1 coend m'

P0 ::  l0:  while True do
               NC0:  wait (turn = 0);
               CR0:  turn := 1;
            end while;
       l0'

P1 ::  l1:  while True do
               NC1:  wait (turn = 1);
               CR1:  turn := 0;
            end while;
       l1'
```

Observations:

- Every program point is labeled, including exit points.
- The prime character is used by convention to indicate the state of a variable after an operation. This is why the entry and exit points of `P`, `P0`, and `P1` are realted in this manner.
- Concurrent programs communicate via shared variables.
- The wait primitive only proceeds when its guard becomes true.

Sketch diagram similar to Clarke Figure 2.2. Show the reachable states in the Kripke structure.

The intent of the example is to give a program in which processes 0 and 1 are not in their critial region at the same time. That is, at no time do the pair of program counters have the value `(CR0,CR1)`.

13.2 Kripke Structure

HR calls this a transition system or model. See their Definition 3.4.

Definition 13.1 (Clarke Section 2.1) *Let AP be a set of atomic propositions. A* Kripke structure *M over AP is a four tuple $M = (S, S_0, R, L)$ where*

1. *S is a finite set of states*
2. *$S_0 \subseteq S$ is the set of initial states.*
3. *$R \subseteq S \times S$ is a transition relation that must be total, that is for every state $s \in S$ there is a state $s' \in S$ such that $R(s, s')$.*

4. *$L : S \to 2^{AP}$ is a function that labels each state with the set of atomic propositions true in that state.*

Sometimes S_0 is omitted (HR omits S_0).

Definition 13.2 *A* path *in the structure M from a state s is an infinite sequence of states $\pi = s_0 s_1 s_2 \ldots$ such that $s_0 = s$ and $R(s_i, s_{i+1})$ holds for all $i \geq 0$.*

13.3 Linear-time temporal logic

Atomic $\top$, $\bot$, or $p \in AP$

Logical $\neg\phi$, $\phi \wedge \phi$, $\phi \vee \phi$, $\phi \to \phi$

Temporal

$X\phi$ Next (true in the next state)

$F\phi$ Future (true in some future state)

$G\phi$ Globally (true in all future states)

$\phi\, U\, \phi$ Until.

$\phi\, W\, \phi$ Weak until.

$\phi\, R\, \phi$ Release.

LTL is given semantics in terms of paths. We will first define $\pi \models \phi$, a satisfaction relation between an individual path and an LTL formula. Later on we will define satisfaction in a model, $M, s \models \phi$, quantifying over all paths starting from s.

Definition 13.3 (following HR 3.6) *Let $M = (S, R, L)$ be a model and $\pi = s_0 s_1 \ldots$ be a path in M. $\pi \models \phi$ is defined by structural induction on ϕ as follows:*

1. $\pi \models \top$

2. $\pi \not\models \bot$

3. $\pi \models p$ *iff* $p \in L(s)$

4. $\pi \models \neg\phi$ *iff* $\pi \not\models \phi$

5. *and*

6. *or*

7. *implies*

8. $\pi \models X\phi$ *iff* $\pi^1 \models \phi$

9. $\pi \models G\phi$ *iff for all* $i \geq 0$, $\pi^i \models \phi$

10. *$\pi \models F\phi$ iff there is some $i \geq 0$ such that $\pi^i \models \phi$.*

11. *$\pi \models \phi \, U \, \psi$ iff there is some $i \geq 0$ such that $\pi^i \models \psi$ and for all $j = 0, \ldots, i-1$ we have $\pi^j \models \phi$.*

12. *$\pi \models \phi \, W \, \psi$ iff either there is some $i \geq 0$ such that $\pi^i \models \psi$ and for all $j = 0, \ldots, i-1$ we have $\pi^j \models \phi$; or for all $k \geq 0$ we have $\pi^k \models \phi$.*

13. *$\pi \models \phi \, R \, \psi$ iff either there is some $i \geq 0$ such that $\pi^i \models \phi$ and for all $j = 1, \ldots, i$ we have $\pi^j \models \psi$, or for all $k \geq 0$ we have $\pi^k \models \psi$.*

Notation:

$$\begin{aligned} \pi^0 &= \pi \\ (s_0 \pi)^{n+1} &= \pi^n \end{aligned}$$

This convention differs from HR. I couldn't wrap my brain around the algebra of their choice. This choice follows Clarke *et al.*.

Discussion

- Relationship between U and W.
- U and R are duals via $\phi \, R \, \psi = \neg(\neg\phi \, U \, \neg\psi)$
- Release intuition: $\phi \, R \, \psi$ because ψ must be true up to and including the moment when ϕ becomes true.

Definition 13.4 (HR 3.8) *Suppose M is a Kripke structure, $s \in S$ and ϕ an LTL formula. We write $M, s \models \phi$ if, for every execution path π of M starting at s we have $\pi \models \phi$.*

This is often abreviated $s \models \phi$.

Examples following HR Figure 3.3 and discussion pages 182, 183.

Limitations Cannot assert the existance of a path.

For example, cannot assert that it is always possible to return to the "restart" state.

13.4 LTL equivalences

$$\begin{aligned} \neg G\phi &\equiv F\neg\phi && (1) \\ \neg F\phi &\equiv G\neg\phi && (2) \\ \neg X\phi &\equiv X\neg\phi && (3) \\ \neg(\phi \, U \, \psi) &\equiv \neg\phi \, R \, \neg\psi && (4) \end{aligned}$$

$$\begin{aligned} F(\phi \vee \psi) &\equiv F\phi \vee F\psi && (5) \\ G(\phi \wedge \psi) &\equiv G\phi \wedge G\psi && (6) \end{aligned}$$

$$\begin{aligned}
F\phi &\equiv \top\, U\, \phi && (7)\\
G\phi &\equiv \bot\, R\, \phi && (8)\\
\\
\phi\, U\, \psi &\equiv \phi\, W\, \psi \wedge F\psi && (9)\\
\phi\, W\, psi &\equiv \phi\, U\, \psi \vee G\phi && (10)\\
\\
\phi\, W\, \psi &\equiv \psi\, R\, (\phi \vee \psi) && (11)\\
\phi\, R\, \psi &\equiv \psi\, W\, (\phi \wedge \psi) && (12)
\end{aligned}$$

Remark: HR discusses adequate sets. This will be important when automating.

13.5 NuSMV

The "New Symbolic Model Verifier".

Program expressed as modules, with a distinuished module main.

First-order imperative language with shared memeory.

14 Model Checking: Algorithms

14.1 Computational Tree Logic

14.2 CTL*

14.3 Algorithms