Basics of formal verification - DRAFT *

Ganesh Gopalakrishnan
50 S, Central Campus Dr, Rm, 3190
Salt Lake City, UT 84112-9205
ganesh@cs.utah.edu

University of Utah

1 Introduction

This course will attempt to provide a sampling of some of the ideas underlying modern formal methods as well as formal verification tools. Given that this is my very first attempt to put together such a course, given the amount of time I had to actually get ready for this course (a week), and given the actual duration of the course (two days), I have to set very modest, but nevertheless well articulated, goals. Feel free to provide feedback at every juncture as I hope to improve through this experience.

We all pretty much understand how Boolean (combinational) circuits work; we all have designed dozens, if not hundreds, of finite-state machines in our lives; and we all have seen computers in various shapes and sizes. We all pretty much know how test-benches are written, what’s good and bad about simulators, and what’s the promise offered by formal methods tools. So what does this course strive to do?

Well, as explained to me (and which I agree 100% with), one does not become an effective user of these modern tools unless one has a good intuitive understanding of the principles behind formal tools. I shall therefore focus on distilling out some of the principles that have stood the test of time, and will continue to be the basis of tools yet to come. Given the lack of time, I shall attempt to characterize the general nature of these principles, and illustrate them on concrete examples. I would say that this course strives to be the contemplative phase of your learning formal methods. I am told that there will be a hands-on tools oriented course that is to follow.

2 So what underlies formal methods?

In a nutshell, formal methods strive to provide for digital design what engineering mathematics has provided in other walks of engineering: the ability to express specs

* Supported in part by NSF Grant No. CCR-9800928
precisely, the ability to clearly define when an imp meets the spec, and the ability to understand the spec and the imp better through “calculations”.

To sum up, formal methods are based on

- precise expression of ideas
- sound and complete calculational principles
- efficient implementations of these calculations

Precise expression is achieved through mathematical logic. Soundness and completeness are provided by the deductive machinery underlying logic. Efficient implementations are supported by procedures (that may not terminate) or algorithms (that will terminate given enough time and memory resources).

Let’s consider one example: propositional calculus allows us to specify “Boolean relationships” amongst objects. Many sound and complete axiomatizations of propositional calculus exist. Modern efficient implementations involve either Binary Decision Diagrams or SAT methods.

This course will be divided into four modules:

**Module 1**: Understanding propositional and first-order predicate calculus
**Module 2**: Understanding program logics
**Module 3**: Induction and Recursion
**Module 4**: Temporal reasoning

While this division is largely based on my background and the format of the course, it also includes issues that I believe you will encounter in your use of formal tools. In my presentation, I will often blur boundaries between these modules. Last but not least, while I’ll try to give credits wherever possible, apologies for omissions, especially in a working draft similar to this.

### 3 Module 1: Propositional and First-order Logics

#### 3.1 Propositional Logic

Formal logics are designed to express the notion of truth in various domains of discourse. Formal logic systems are set up by defining a finite (usually small) collection of *axiom schemes* and *rules of inference*. For instance, for propositional logic, the following axiom schemes suffice (from Church’s 1956 book):

**Axiom scheme 1**: \( p \Rightarrow (q \Rightarrow p) \)

**Axiom scheme 2**: \( (s \Rightarrow (p \Rightarrow q)) \Rightarrow (s \Rightarrow p) \Rightarrow (s \Rightarrow p) \)
Axiom scheme 3: \((\neg q \Rightarrow \neg p) \Rightarrow (p \Rightarrow q)\)

Let’s get to know the \(\Rightarrow\) symbol well. Basically \(true \Rightarrow false\) has truth-value false, and all other combinations have truth-value true. Thus, the following sentence is true: \((2 > 3) \Rightarrow (4 > 5)\).

These are axiom schemes because any instance of these schemes is an axiom. For example if you have Boolean variables \(x\) and \(y\), you can treat the following as axioms:

\[
x \Rightarrow (y \Rightarrow x) \\
x \Rightarrow ((y \Rightarrow x) \Rightarrow x) \\
(y \Rightarrow y) \Rightarrow (y \Rightarrow (y \Rightarrow y))
\]

etc.

The only rule of inference we need corresponding to the above axioms (to obtain a complete logical system where all truths can be proved) is modus ponens:

- Given \(A\) and \(A \Rightarrow B\) as theorems, infer \(B\) as a theorem.
- The basis case for theorems is provided by the axioms,

Thus, if you find that a sentence \((a \Rightarrow a)\) is true, the complete nature of the above axioms + rules guarantees the existence of a proof for \((a \Rightarrow a)\) (try it if you have some time now, or later after the course is over—it is somewhat non-trivial! See at the end of this writeup). A proof is a sequence of theorems each of it is either an axiom or is obtained from previous theorems using only the rule(s) of inference.

Here are some more definitions (some are derived from the book; see the book for more precise/detailed versions):

- A collection of axioms are independent if none can be derived from the others using the rule(s) of inference.
- A logical system is sound if all theorems are true.
- A sentence is valid (or true) if for all free-variable “settings”, it emerges true. Note: In \(\forall x P(x, y)\), only \(y\) is free.
- A well-formed formula (wff) is a syntactically correct formula.
- A sentence is a well-formed formula that has no free variables.
- A model is a pair <Universe, Meanings of relational symbols>.
- A theory of a model is the set of true sentences that use only the relational symbols defined by the model.

A proof of \(\vdash p \Rightarrow p\) from the above axiomatization

- \(\vdash (p \Rightarrow ((p \Rightarrow p) \Rightarrow p))\) because this is an instance of Axiom scheme 1, above. Read “\(p\)” in the axiom scheme as \(p\) and “\(q\)” as \((p \Rightarrow p)\). Also read \(\vdash\) as “is a theorem”.

\[ \vdash (p \Rightarrow ((p \Rightarrow p) \Rightarrow (p \Rightarrow p))) \Rightarrow (p \Rightarrow (p \Rightarrow p)) \Rightarrow (p \Rightarrow p), \] because this is an instance of Axiom scheme 2, above. Read

- "s" as \( p \)
- "p" as \((p \Rightarrow p)\), and
- "q" as \((p \Rightarrow p)\)

- Apply modus ponens of the above two, to infer \( \vdash (p \Rightarrow (p \Rightarrow p)) \Rightarrow (p \Rightarrow p) \)
- \( \vdash (p \Rightarrow (p \Rightarrow p)) \), because it is an instance of axiom scheme 1
- Apply modus ponens of the above two, to infer \( \vdash (p \Rightarrow p) \). End of proof!

Clearly this method of arriving at a proof through purely syntactic means, while “100% kosher” and “aesthetically pleasing,” is not computationally very efficient. The classical view that truth only be obtained through syntactic manipulations is being rapidly shed. In fact, it is considered quite okay to make gigantic derivational steps using auxiliary trusted means, such as computational procedures using decision diagrams.

**An Aside: On Soundness and Completeness** Soundness and completeness are recurring themes in verification. These notions are not confined to formal logics alone. For instance, consider the following context-free grammar and the associated claim, and what soundness and completeness means:

\[
S \rightarrow \varepsilon | 0 \ S \ 1 | 1 \ S \ 0 | S \ S
\]

Claim: \( S \) generates ALL and ONLY those strings over \( \{0,1\}^* \) which have equal numbers of 0’s and 1’s

**Soundness:** the ‘‘ONLY’’ part (via case analysis)

**Completeness:** the ‘‘ALL’’ part (via induction)

By induction hyg, assume all strings of length \(< n\) have derivations from \( S \).

Induction step: consider a string of length \( \geq n \). Case analysis:

If \( 0 \cdot .1 \) or \( 1 \cdot .0 \), by ind-hyg, \( S \Rightarrow \ldots \) and so \( S \Rightarrow \cdot .1 \) or \( S \Rightarrow \cdot .0 \).

If \( x = \cdot .1 \), we have \( \ldots \) having exactly two more 0’s than 1’s. That means \( x = s1 \) where \( s1 \) and \( s2 \) have equal 0 and 1 (why?). Further, \( \text{neither } s1 \text{ nor } s2 \) are empty (why?). This can be obtained via \( S \Rightarrow s1 \ s2 \).

Q: Can you write a better proof?

Clearly, the above proof is done in the “logic of math”, i.e. the proof is rigorous but not formal. The latter needs that we axiomatize the theory of Parse-trees and derivations, and perform a structural induction based proof of what we proved. That would then make the proof formal. In a formal proof, each step is justified in terms of the previous steps - even arguments done in the theory of strings and integers have to be explicated. Modern verification tools often bridge these “gaps” by invoking decision procedures and not actually writing out every step in the syntax of some formal deductive system.
3.2 The complexity of propositional reasoning

Propositional reasoning (to be specific; determining the satisfiability of Boolean formulae expressed in conjunctive normal form) is not computationally easy. Usually this is attributed to it being “NP complete.” Here’s the deal behind NP-completeness. People in the ’60s noticed that certain problems defied non-exponential exact solutions. So they said, “well if you can’t beat them, club them so that one arrow can kill ’em all.” In other words, they found tricks to simulate 1000s of problems upon each other so that if one has a polynomial exact algorithm, all the others will. 3SAT (satisfiability of CNF Boolean formulae that have at least three literals in each clause) was the first problem induced into this “club” (the NP-completeness club). Since 1000s of scientists can’t be independently and equally stupid in not finding poly algorithms for 1000s of these problems, it is widely held that this “club” of problems have no poly algorithms.

3.3 Advanced Propositional Stuff

While propositional logic may appear quite impoverished to adequately describe many situations, in fact it is sufficiently expressive to model situations with unusual degrees of detail, The best example I can think of to illustrate this is Prof. C.A.R. Hoare’s model for CMOS transistors. In that formalism, Prof. Hoare models an N transistor as follows

\[ N_{trans}(g, dg, s, ds, d, dd) = (g \text{ implies } (s=d)) \land (g \land \neg \nexists \neg d \text{ implies } (ds=dd)) \]

The idea is that a pair of propositional variables are introduced to model each terminal of a transistor. The signal g models the logic level of the gate, and dg models whether the gate is being “driven strongly,” i.e., is not suffering from a threshold drop.

A P transistor is similarly,

\[ P_{trans}(g, dg, s, ds, d, dd) = (g \text{ implies } (s=d)) \land (g \land s \land d \text{ implies } (ds=dd)) \]

Thus, to describe a CMOS inverter, we write the circuit as

\[ inv((i, di), (o, do)) = N_{trans}((i, di), (0, 1), (o, do)) \land P_{trans}((i, di), (1, 1), (o, do)). \]

From this, we should be able to prove do.

An attempt to construct a (non-inverting) buffer as

\[ N_{trans}((i, di), (1, 1), (o, do)) \land P_{trans}((i, di), (0, 1), (o, do)) \]

gives the correct logic functionality but not the drive on the output for further composition.

Hoare goes on to further show the following details in his inimitably lucid papers:
- Circuit connections can be modeled using $\exists$. Thus, two inverters connected in cascade become

$$\exists(t, dt). inv((i, di), (t, dt)) \land inv((t, dt), (o, do)).$$

- Each circuit can be described as a triple of attributes $$(C, D, N)$$ where $C$ expresses the Boolean operating conditions (the behavior of the $g, s, d$ signals), $D$ expresses the drive constraints (the $dd, ds, dg$ variables), and $N$ expresses the need for drive.

- Connection between two circuits $$(C_1, D_1, N_1)$$ and $$(C_2, D_2, N_2)$$ can be expressed as a circuit obtained by component-wise conjunctions.

- Equivalence between two circuits can be expressed as equivalence under the legal operative regions. Thus,

$$(C_1, D_1, N_1) \equiv (C_2, D_2, N_2) =$$

$$C_1 = C_2 \land C_1 \land D_1 = C_2 \land D_2 \land C_1 \land D_1 \land N_1 = C_2 \land D_2 \land N_2.$$ 

- The fact that one circuit is better than another circuit (is a better replacement than the original) can be expressed as

$$C_2 \implies C_1 \land D_2 \implies D_1 \land N_1 \implies N_2.$$

This is shown to be a preorder on circuits. Further this is shown to be a fully compositional (substitutive) property in that the preorder is preserved through all circuit operations such as parallel composition (conjunction) and hiding (existential quantification).

- The conjunction of a preorder and its inverse is an equivalence relation. We can show that the circuit preorder above reduces to $\equiv$ if we conjoin it and its inverse.

The ability to study these notions in the setting of propositional calculus is extremely pleasing. For instance, one can see that preorders allow circuits to “maintain their own identity” while “diplomatcally agreeing” to maintain a notion of equivalence also. This is not possible in partial orders due to the antisymmetry rule; the moment we conjoin a partial order and its inverse, the two objects are forced to be the very same. This is quite irksome in many contexts where, for instance, concurrency and nondeterminism are involved where there truly may be observable differences and yet equivalence under the conjunction of a simulation preorder and its inverse.
3.4 QBF: Calculating $\exists$ and $\forall$

The above Boolean formulae with quantifications are called quantified Boolean formulae. One can simplify such formulae by using the meaning of $\exists$ as disjunction over the Boolean domain, and $\forall$ as conjunction over the Boolean domain. (Show how.)

3.5 First-order Logic

While propositional logic allows one to quantify over true and false, in a domain of discourse where there are many (as many as Nat or Reals, or other domains) items, we will need to quantify in other ways; examples; “for all natural numbers,” “there exists a natural number,” etc. In these cases, we need something more than propositional logic.

In first order logic, one defines $\forall$ as an “infinite conjunction”—over all the individuals, and $\exists$ as “infinite disjunction”. While that is the main idea, we need other concepts also. For instance, we need to describe individuals obtainable from others through computational processes. We use function constants for this purpose. For instance we can use $\text{succ}$ to denote the successor of natural numbers, $\text{succ}(x)$ then denotes (or is meant to denote) $x+1$. However we may prefer to bind the meaning of these operators as late as possible; after all, if we assume that these operators have no known properties and still prove something, we would have won. These and other features are described below.

Late binding, or unintrepreted functions Suppose we introduce two operators $\text{succ}$ and $\text{pred}$. Suppose we assume nothing about them—i.e., we don’t even assume that $\text{pred}(\text{succ}(x))=x$ for a natural number $x$. All we assume is that if two things “look the same” then (and only then) they are the same. We can still prove many useful properties. For instance,

$$\forall x. \text{pred}(\text{succ}(x)) \iff \exists x. \text{pred}(\text{succ}(x)).$$

Avoiding interpretations can sometimes help accomplish many proofs by merely “matching” equals for equals. For instance, in verifying processors, one often compares a detailed implementation model against a more abstract reference model. Before a direct comparison can be made, verification conditions expressing the equality of corresponding “observable entities” are first generated (these observable entities could be registers or memories). In these verification conditions, terms from the specification and implementation are often compared for equality. If such terms are syntactically the same, the proof is accomplished essentially by “matching up the function invocation patterns”—avoiding matching the “results of evaluating the expressions.”
However, as you have guessed, keeping things uninterpreted also prevents us from proving many properties. The act of describing the meanings of function symbols is known as providing interpretations or interpreting.

Predicate calculus (or first-order predicate calculus), or first-order logic are all names used to denote a formal mathematical language which is used to describe properties. The aspect “first order” connotes the fact that quantification is done only over individuals (countably infinite domains such as Naturals, Strings, etc.). When quantification is done over functions and predicates, we get “higher order” logics—or simply predicate calculus.

Zohar Manna’s book “Mathematical Theory of Computation,” McGraw-Hill, 1973, describes many basic notions through examples that are summarized below. The notions are well formed formulas (wff) that are the sentences in the logic, and statements that are sentences together with an interpretation. Basically an interpretation chooses domains for constants, predicates and function symbols to range over and assigns to them. Examples below will clarify.

Example 1.

\[ \exists F. F(a) = b \]
\[ \land (\forall x). [p(x) \Rightarrow F(x) = g(x, F(f(x)))] \]

Interpretation 1.

\[ D = \text{Nat} \]
\[ a = 0 \]
\[ b = 1 \]
\[ f = \lambda x. (x = 0 \rightarrow 0, x - 1) \]
\[ g = * \]
\[ p = \lambda x. x > 0 \]

Interpretation 2.

\[ D = \Sigma^* \]
\[ a = \varepsilon \]
\[ b = \varepsilon \]
\[ f = \lambda x. (\text{tail}(x)) \]
\[ g(x,y) = \text{concat}(y, \text{head}(x)) \]
\[ p = \lambda x. x \neq \varepsilon \]

Interpretation 3.

\[ D = \text{Nat} \]
\begin{align*}
a &= 0 \\
b &= 1 \\
f &= \lambda x.x \\
g(x,y) &= y+1 \\
p &= \lambda x.x > 0
\end{align*}

Example 2.
\[
\forall P. P(a) \\
\land (\forall x). [(x \neq a) \land P(f(x)) \Rightarrow P(x)] \\
\Rightarrow (\forall x. P(x))
\]

We can interpret the above formula such that we get the principle of induction over \text{Nat} and over strings.

**Valid formulas.**
Valid formulas are those that are true for all interpretations. For example,
\[
\forall x. f(x) = g(x) \Rightarrow \exists a. f(a) = g(a).
\]

Valid formulas are those where we can assign to the free variables as well as interpret the function and predicate symbols in every possible way, always resulting in the truth-value true.

**Satisfiability versus validity** The notions of satisfiability and validity in first-order logic are good to get straight. Here are a collection of facts:

- When a first-order logic formula is satisfiable, it has (at least) one interpretation that yields true. For instance \( p(x, g(y)) \) is true if \( p \) is set to \(<\) and \( g \) is set to the successor function, assuming the domain to be that of natural numbers. The opposite of satisfiable is unsatisfiable (is a contradiction).
- The notions of universal and existential closures are important, if \( \phi \) is a FOL formula, \( \forall(\phi) \) is the formula obtained by explicitly universally quantifying the free variables in \( \phi \). Similarly we have the notion of existential closure, written \( \exists(\phi) \).
- If \( \text{fsat}(\phi) \) then that means for no assignment to the free variables of \( \phi \) and interpretation of predicate and function symbols can we make the formula true. In other words, \( \text{fsat}(\phi) \leftrightarrow \text{valid}(\forall(\phi)) \).
- Recall that \( a \leftrightarrow b \) can be written \( a = b \) also—meaning \( a \) and \( b \) have the same truth values. So by that token, we have \( (a \leftrightarrow b) \leftrightarrow (\neg(a) \leftrightarrow \neg(b)) \).
- So \( \text{sat}(\phi) \leftrightarrow \text{not}(\text{valid}(\exists(\phi)))) \).
AN ASIDE (for now): the validity problem of FOL is not decidable For $\sigma \in \{0, 1\}$, use the abbreviation

$$f_{\sigma, 1, \sigma_2, \ldots, \sigma_n}(a) = f_{\sigma_n}(f_{\sigma_{n-1}}(\ldots f_{\sigma_1}(a)) \ldots).$$

Given a Post system $S = \{(\alpha_1, \beta_1), (\alpha_2, \beta_2), \ldots, (\alpha_n, \beta_n)\}$, $n \geq 1$ over $\Sigma = \{0, 1\}$, construct the wff $W_S$ as follows:

\begin{align*}
\bigwedge_{i=1}^n p(f_{\sigma_i}(a), f_{\beta_i}(a)) & \quad (2-1) \\
\forall x \forall y [p(x, y) \Rightarrow \bigwedge_{i=1}^n p(f_{\sigma_i}(x), f_{\beta_i}(y))] & \quad (2-2) \\
\Rightarrow \exists z p(z, z) & \quad (2-3)
\end{align*}

We now prove that $S$ has a solution iff $W_S$ is valid.

**Part 1.** $W_S$ valid $\Rightarrow S$ has a solution.

If valid, it is true for all interpretations. Pick the following interpretation:

- $a = \epsilon$
- $f_0(x) = x0$
- $f_1(x) = x1$

Under this interpretation, parts (2-1) and (2-2) of $W_S$ are true. In other words, $W_S$ is not being made vacuously true by this interpretation. Thus, the consequent, namely formula (2-3) is true. This now means that $S$ has a solution, namely some string $z$ that lends itself to being interpreted as some sequence $\alpha_{i_1} \alpha_{i_2} \ldots \alpha_{i_n}$ as well as $\beta_{i_1} \beta_{i_2} \ldots \beta_{i_n}$. That is,

$$\alpha_{i_1} \alpha_{i_2} \ldots \alpha_{i_n} = z = \beta_{i_1} \beta_{i_2} \ldots \beta_{i_n}.$$

**Part 2.** $W_S$ valid $\Leftarrow S$ has a solution.

If $S$ has a solution, let it be the sequence $\alpha_{i_1} \ldots \alpha_{i_n}$. In other words, $\alpha_{i_1} \alpha_{i_2} \ldots \alpha_{i_n} = \beta_{i_1} \beta_{i_2} \ldots \beta_{i_n}$. Now we show that for every interpretation, if the antecedents of $W_S$ are true, the consequent is true.

If the antecedent is true, we can conclude from (2-1) that $p(f_{\sigma_i}(a), f_{\beta_i}(a))$ is true. Now using (2-2) as a rule of inference, we can conclude that $p(f_{\sigma_{i_1}}(a), f_{\beta_{i_1}}(a))$ is true. In other words, $p(f_{\sigma_{i_1} \alpha_{i_2}}(a), f_{\beta_{i_1} \beta_{i_2}}(a))$ is true. Continuing this way,

$$p(f_{\sigma_{i_1} \alpha_{i_2} \ldots \alpha_{i_n}}(a), f_{\beta_{i_1} \beta_{i_2} \ldots \beta_{i_n}}(a))$$

is true. This means that there exists a $z$ such that $p(z, z)$—that $z$ being $f_S(a)$.

Now, given the definition of $sat(\phi)$ earlier, the satisfiability of FOL is also undecidable.

But, both validity and satisfiability are semi-decidable. This means that one can write a procedure $P$ that behaves as follows: whenever one gives $P$ a FOL formula $\phi$ that is indeed valid (say an oracle told you that $\phi$ is valid), $P$ will determine this effectively (by means of the method described via a program—and not by asking the oracle) and halt. If, however, one gives a $\phi$ that is not valid, $P$ may loop.

### 3.6 Decidable subsets of FOL

We will illustrate how to decide a small fragment of FOL formulas - using DFA! This material is from Sipser's book.
A sentence is a well-formed formula that has no free variables. A model is a pair <Universe, Meanings of relational symbols> that is used to assign truth values to formulas. A theory of a model is the set of true sentences that use only the relational symbols defined by the model.

Here are a few examples, $Th(N, +)$ consists of sentences of the form $\forall x \exists y x = y$, $\exists y y = 0$, and so on. These are in the theory because they are sentences (no free variables) and are true. It does not contain the formulas $\exists x \forall y x = y$ and $y = 0$: the first is a false sentence, and the second is not a sentence.

Sipser’s book presents a nice DFA construction to show that the theory $Th(N, +)$ of Naturals with addition is decidable. I will illustrate the construction on a few examples.

$\forall x \exists y x = y$

1. First build a DFA that corresponds to $x = y$. Our convention is that the DFA corresponding to $x = y$ gets the values of $x$ and $y$ in bit-serial fashion, LSB first. This DFA must accept runs such as [0 0] [0 0] [1 1] [1 1] [0 0], i.e., when we take the first components of the above pairs and string them together to get 00110, we must get the same number as doing the same operation with the second components. In this example, this is true. However, the run [0 0] [0 0] [1 0] [1 1] [0 0] must not be an accepting one because the number 00110 is not the same as 00010.

Notice that this DFA accept all those strings that represent all those natural numbers for which the given formula is valid (true). We get:

![DFA Diagram]

2. Next, consider formula $\exists y (x = y)$ and build a corresponding automaton for it. This automaton can accept only $x$ from the external world ($y$ is “hidden”—the automaton non-deterministically guesses all possible $y$s). Existential quantification disjuncts all possible choices of $y$. This can be achieved by considering each choice for bit $y$ and adding an NFA transition to the automaton being built.
We get:

3. This DFA is simplified to obtain the following:

4. Next, to apply \( \forall x \), we convert it to \( \neg \exists x \ \neg \). To apply this, first negate (complement) the automaton in step 3, hide \( x \) from it, then negate the automaton again. This results in:

5. The above automaton accepts the empty tuple \([\,]\). As we had maintained correspondence between runs of DFAs and satisfying assignments of formulas all along, the final result, representing the entire formula, tells us that the formula is true.
6. If we change our example to $\exists x \forall y \ x = y$, we will obtain the automaton

$$
\begin{array}{c}
\circ \\
\downarrow \\
[1]
\end{array}
$$

telling us that the given formula is not true.

3.7 Some rigorous proofs in FOL

Example 1 Prove that

$$[(\forall u \text{Cube}(u) \Rightarrow \forall u \text{Small}(u)) \land \neg \forall u \text{Small}(u)] \Rightarrow \neg \forall u \text{Cube}(u)$$

Proof. The contrapositive of $[(\forall u \text{Cube}(u) \Rightarrow \forall u \text{Small}(u))]$ is $[(\exists u \neg \text{Small}(u) \Rightarrow \neg \forall u \text{Cube}(u))]$. One application of modus ponens gives the desired result.

Example 2 Suppose $p$ and $q$ are predicate constants. Let $Nat$ be the set of natural numbers (numbers $\geq 0$). Suppose the following are true:

1. $\forall n \in Nat \ (p(n) \Rightarrow p(n + 1))$
2. $\forall n \in Nat \ (p(n) \Rightarrow q(n))$
3. $\exists k \in Nat \ (k > 0 \land \neg q(k))$

1. Prove that $\forall i \in Nat \ (i \leq k \Rightarrow \neg p(i))$
2. Argue that $\forall i \in Nat \ (i < k \Rightarrow \neg q(i))$ is not necessarily true.

From $\exists k \in Nat \ (k > 0 \land \neg q(k))$, we conclude $p(k)$. Then using the first rule, we build a contrapositive chain. But this implies nothing about $q(i)$ for $i < k$.

3.8 The Sequent Calculus

Proofs are generally presented in terms of sequent formulas. Rules pertaining to sequents are well explained in Dr. Perry Alexander’s notes available at http://www.ececs.uc.edu/~alex/teach/ece793/sequent.html.

4 Module 2: Program Logics

4.1 Hoare Logic

In this section, we will present an overview of Floyd/Hoare logic. Floyd/Hoare is a topic of considerable importance for all Computer Science students (even undergraduates)
to know. I would say that Floyd/Hoare logic for the first time introduced Computer Science to the idea that programs can be reasoned about formally using the techniques of classical mathematical logic. All the developments we “take for granted” nowadays (e.g. Temporal Logic, Pre- and Post assertions for programs, etc.) have their genesis in Floyd/Hoare logic.

A brief historical note: it was Robert Floyd who first proposed the idea of proving properties about flow-charts by attaching first-order logic assertions to “program-points”, and deriving verification conditions from a flow-chart. The problem of proving something about a flow-chart program was reduced to that of mechanically generating the verification conditions about the program and then showing that all the verification conditions are true (logically valid).

To motivate Floyd/Hoare logic, consider a proof of a simple routine to swap two bit-vectors A and B, as follows:

\[
\{ \text{A = a and B = b} \} \\
\text{begin} \\
\quad \text{A := A xor B ;} \\
\quad \text{B := A xor B ;} \\
\quad \text{A := A xor B ;} \\
\text{end} \\
\{ \text{A = b and B = a} \}
\]

Here, a command consisting of three sequential assignments is annotated with a pre-condition that says “let A be a and B be b before the command is executed”. Then, the post-condition reads “if and when the execution of the command terminates, we have A = b and B = a”. The assertions before and after the command are called the pre and post conditions, respectively, and together they are called partial correctness assertions. The word partial connotes the fact that termination of the program is not part of the assertion—only the nature of the result if the program terminates.

As opposed to partial correctness, there is a notion of total correctness also. Total correctness = partial correctness + termination. For example, no one knew the status of the following total correctness assertion about the “Fermat program” till Andrew Wiles proved that the total correctness assertion is indeed true.

\[
\text{true} \quad \text{if} \ (\exists \ x, y, z : \text{nat} \ . \ \exists n : \text{nat} \ . \ (n > 2) \land x^n + y^n = z^n) \ \text{then loop else halt} \quad \text{true}
\]
On the other hand, no one knows whether the following total correctness assertion is true or not:\footnote{If you have the quixotic instincts of “Gonzo the great”, here’s your chance to hole-up in your attic room for 8 years and ...}

\[
[\text{true}]
\]

\[
\text{while } (x > 1) \text{ do}
\]
\[
\text{if } \text{odd}(x) \text{ then } x := 3x + 1 \text{ else } x := x/2 \text{ endif}
\]

\[
[\text{true}]
\]

On the other hand, the following total correctness assertion is known to be true:

\[
[\text{false}]
\]

\[
\text{while } (x > 1) \text{ do}
\]
\[
\text{if } \text{odd}(x) \text{ then } x := 3x + 1 \text{ else } x := x/2 \text{ endif}
\]

\[
[\text{anything}]
\]

whereas, it is known that the following total correctness assertion is false

\[
[\text{true}]
\]

\[
\text{while } (x > 1) \text{ do}
\]
\[
\text{if } \text{odd}(x) \text{ then } x := 3x + 1 \text{ else } x := x/2 \text{ endif}
\]

\[
[\text{false}]
\]

while it wasn’t known until Wiles’s proof whether the following was true or not!

\[
[\text{true}] \quad \text{if } (\exists x, y, z : \text{nat} . \exists n : \text{nat} . (n > 2) \land x^n + y^n = z^n) \text{ then loop else halt } [\text{false}]
\]

The Swapping Program Returning to the program that swaps using XORs, here is how one may prove it. I show assertions in-between the statements, obtained in the order shown by the numbers in parantheses:
\{ A = a \land B = b \}

\| \| \| \| \backslash / \backslash

(5) \{ ((A \text{xor} B) \text{xor} ((A \text{xor} B) \text{xor} B)) = b \land ((A \text{xor} B) \text{xor} B) = a \}

\text{begin}

(4) \{ ((A \text{xor} B) \text{xor} ((A \text{xor} B) \text{xor} B)) = b \land ((A \text{xor} B) \text{xor} B) = a \}

A := A \text{xor} B ;

(3) \{ (A \text{xor} (A \text{xor} B)) = b \land (A \text{xor} B) = a \}

B := A \text{xor} B ;

(2) \{ (A \text{xor} B) = b \land B = a \}

A := A \text{xor} B ;

(1) \{ A = b \land B = a \}

\text{end}

\{ A = b \land B = a \}

The rule we used in the above derivation is summarized in Hoare logic as
\{ P \ [E / x] \} x := E \{ P \}

which says that “given P is the post-condition after x := E, the \textit{weakest pre-condition} of x := E with respect to post-condition P is P \ [E / x] \}, i.e., the assertion P, except where x is mentioned, one must use E. This rule “works” because we realize that the assertion P is being made with respect to the \textit{new} value of x; therefore, to say something that relates to P, we must use E in place of x because it is E’s value in the old state that x assumes in the new state.

One can prove that if \{ Q \} x := E \{ P \}, then Q \Rightarrow P[E/x] .

In a similar vein, the rule for if-then-else is
The forward rule for assignment is

\{ P \} x := E \{ \text{exists oldx} . P [\text{oldx} / x] \text{ and } (x = E [\text{oldx} / x]) \}

As the name suggests, \text{oldx} stands for the old value of \text{x} that got destroyed upon assignment.

**The Binary Search Example** We will now consider a more involved program—the Binary search. A flow-chart for this program appears in Figure 1. Applying Floyd's method involves the following steps:

1. Annotate each entry-point and exit-point with assertions (hopefully there is only one of each, following the structured programming paradigm—note however that we could be dealing with finite-state machines that are not very often structured programs.)
2. Cut each loop with a cut-point and attach a loop invariant there.
3. Argue about each linear execution path that results from the above using the assignment wp (weakest precondition) rule and the if-then-else wp rule.

To arrive at a loop invariant for Binary search, consider a “general situation” that arises in its execution, as shown in Figure 2. We now present a proof outline for the Binary search routine.

The first step is to show that the purported loop invariant LI = (f -> A[m]=i) /
\( i \not\in \text{A[L..l-1, h+1..H]} \) is indeed a loop invariant. To prove this, try the following paths:

1. Path 4,1,6,7,4:
   - At 1:
     \( A[m]=i \land (i \not\in A[L..l-1, h+1..H]) \)
   - At 6:
     \( A[m]=i \rightarrow (A[m]=i \land (i \not\in A[L..l-1, h+1..H])) \)
     which simplifies to
     \( A[m]=i \rightarrow \land (i \not\in A[L..l-1, h+1..H]) \)
   - At 7:
     \( A[(l+h)/2]=i \rightarrow (i \not\in A[L..l-1, h+1..H]) \)
found := false ;
low := L ;
high := H ;

P = "A is sorted and H >= L"

found \lor
low > high

Q = "if found then item = A[mid] else item not in A"

mid := (low + high) \div 2

1
A[mid] == item

2
A[mid] < item

3
low := mid

4

5
found := true

6

7

Fig. 1. The Binary Search Routine
found => A [ mid ] == item
\/
item not-in A[ L .. l-1]
\/
item not-in A[h+1 .. H]

Fig. 2. Illustration of the Binary-Search Loop Invariant

- At 4:
  \( f \lor l<h ) => \)
  \( (A[(1+h)/2]=i => (i notin A[L..l-1, h+1..H])) \)
- To show that
  LI =>
  \( ( f \lor l<h ) => \)
  \( (A[(1+h)/2]=i => (i notin A[L..l-1, h+1..H])) \)
Do a case analysis on f.
  - Case f=false:
    \( (i notin A[L..l-1, h+1..H]) => \)
    \( (l<h ) => \)
    \( (A[(1+h)/2]=i => (i notin A[L..l-1, h+1..H])) \)
    which is true
  - Case f=true:
    LI =>
    false =>
    \( (A[(1+h)/2]=i => (i notin A[L..l-1, h+1..H])) \)
    which is also true.

2. Path 4,2,5,6,7,4:
- At 2:
  \( (f => A[m]=i) \lor (i notin A[L..m, h+1..H]) \)
- At 5:
  \[ A[m] < i \Rightarrow \]
  \((f \Rightarrow A[m]=i) \land (i \not\in A[L..m, h+1..H]) \)
- At 6: same as above because \(A[m] < i \Rightarrow\) means \(A[m] \not= i\).
- At 7:
  \[ A[(l+h)/2] < i \Rightarrow \]
  \((f \Rightarrow A[(l+h)/2]=i) \land (i \not\in A[L..(l+h)/2, h+1..H]) \)
- At 4:
  \[ f\backslash l < h \]
  \[ A[(l+h)/2] < i \Rightarrow \]
  \((f \Rightarrow A[(l+h)/2]=i) \land (i \not\in A[L..(l+h)/2, h+1..H]) \)

To show that LI \(\Rightarrow\) above.

Do a case analysis on f.

- Case f=false:
  \(i \not\in A[L..l-1, h+1..H] \Rightarrow\)
  \((l < h) \Rightarrow\)
  \[ A[(l+h)/2] < i \Rightarrow \]
  \((i \not\in A[L..(l+h)/2, h+1..H]) \)

The fact that \(i \not\in A[L..(l+h)/2]\) follows from
\(A[(l+h)/2] < i \Rightarrow\) and that A is sorted.

The fact that \(i \not\in A[h+1..H]\) follows from the antecedent.

3. In the same fashion, establish that LI is an invariant on path 4,3,5,6,7,4.
4. LI is true upon initial entry because wp(LI, (f:=false;1:=L;h:=H)) reduces to
   \((i \not\in A[L..L-1, H+1..H])\), which is true because the ranges are empty.
5. The output assertion Q is established upon exit because
   wp(Q,f\backslash l>h) is

   \[ f\backslash l>h \Rightarrow\]
   \((f \Rightarrow A[m]=i, i \not\in A)\).

LI must imply the above. i.e.,

\[ f \Rightarrow A[m]=i \land (i \not\in A[L..l-1, h+1..H]) \Rightarrow \]
\[ f\backslash l>h \Rightarrow\]
\((f \Rightarrow A[m]=i, i \not\in A)\).

Case analysis on f:
Case f false:

\[(i \text{ notin } A[L..1-1, h+1..H]) \Rightarrow l > h) \Rightarrow i \text{ notin } A\]

which is true because if \( l > h \), then \( 1-1 \geq h \), and hence the whole range is covered by the antecedent.

Case f true:

\[(A[m]=i) \land (i \text{ notin } A[L..1-1, h+1..H]) \Rightarrow \text{true} \Rightarrow (A[m]=i)\]

which is true.

6. Hence the binary search routine is partially correct.

7. To argue that the binary search routine terminates, notice that \( h-1 \) diminishes in each iteration, and so the loop exit condition eventually holds. This is usually termed the well-founded measure for the loop.

8. Hence the binary search routine is totally correct (partial correctness plus termination equals total correctness).

**A Simple Floyd-Hoare logic Prover** Floyd-Hoare logic is used for reasoning about sequential programs. There is, in fact, one Floyd-Hoare logic for every sequential programming language. However, we shall consider features common to most sequential programming languages, and thus define one common Floyd-Hoare logic. The BNF for such a language is described in Gordon’s book, page 7. It includes the following constructs: assignment, sequencing, blocks, if-then, if-then-else, while, and for.

The well-formed formulae (wff) of Floyd-Hoare are annotated programs. Every wff of Floyd-Hoare logic looks like

\[
\{P\} S \{Q\}
\]

where \( P \) and \( Q \) are formulae, and \( S \) is a statement of the programming language under consideration. The above wff reads: “if \( S \) is executed in a state in \( P \) (recall that formulae denote sets of states), the resulting state will be in \( Q \) if the execution terminates.

Some of these well-formed formulae are theorems of Floyd-Hoare logic. New theorems of Floyd-Hoare logic are created from existing theorems by using the rules of inference of Floyd-Hoare logic. To start this process off, we need some theorems, which can be obtained by instantiating the axioms of Floyd-Hoare logic. This describes the forward
proving method. In practice, however, the backward proving method is employed in which we start from the given wff of the form \( \{P\}S\{Q\} \) where \( S \) is the given program to be proved, \( P \) characterizes \( S \)'s inputs and \( Q \) characterizes \( S \)'s answer(s). We then try to establish that the above wff is a theorem.

In directory `/home/handin/cs611/oldstuff/gordon-prover`, there is a toy verifier and a README file that explains how to use this verifier. Note: this “verifier” has a complete verification condition generator. Its “prover” is entirely trivial. It should be possible to substitute other provers (e.g. PVS).

Try this prover on simple examples. If you get a ERROR-PROOF-ABORTED on the screen, look at what gets printed next; that’s the offending “theorem”. You may wish to hand-prove that theorem and insert it into the “logic” database in “mantha-verif.lisp” (look for `(setq logic '(...))`). Once upon a time, this verifier did verify many simple programs. One of its nice features is that it prints a proof-tree. You may also print out the VCs generated to see how the prover actually operates.

4.2 Iteration versus Recursion

The topic of program- and recursion schema used to be widely studied in the 70’s, and fell out of fashion. But like most other things in Computer Science, these topics are an essential part of computing and should not be lost to posterity. And, like most other things in Computer Science, fashions do recur, as Dr. Pnueli mentioned in his recent talk at SRI on processor verification!

Here, I review some of the results from studying program-schema that are of relevance to “verification”. Consider the problem of program transformation. Our motivation for program transformation stems from the fact that we are quite often interested in “getting rid of recursion”. For instance, we all know that the usual fibonacci recursion

\[
\text{fib}(n) = \begin{cases} 
1 & \text{if } n \leq 1 \\
\text{fib}(n-1) + \text{fib}(n-2) & \text{otherwise}
\end{cases}
\]

is very inefficient. We also know that using accumulating parameters, one can transform this into

\[
\text{fib}(n) = \text{fib1}(n, 0, 1)
\]

\[
\text{fib1}(n, a, b) = \begin{cases} 
1 & \text{if } n \leq 1 \\
b & \text{if } n \leq 2 \\
\text{fib1}(n-1, b, a+b) & \text{otherwise}
\end{cases}
\]

However, in general, when can we make such transformations?

The result obtained from studying program schema in the 70’s by Paterson, Hewitt et.al, says that in general such non-linear schema cannot be transformed into linear or iterative schema. A non-linear schema is one where the function being defined appears more than once on the same “arm” of a conditional. A linear schema is one where it appears once, and an iterative schema is a linear schema where the function being defined appears outermost. \text{fib1} is an iterative schema that is computationally equivalent to \text{fib} expressed in a non-linear fashion.
What about our ability to convert linear to iterative form? One may recall from past experience that the linear schema

\[
\text{fac}(n) = \text{if } n = 0 \text{ then } 1 \text{ else } n \times \text{fac}(n-1)
\]

can be transformed into the iterative schema

\[
\text{fac1}(n) = \text{fac1}(n,1)
\]

\[
\text{fac1}(n,\text{ans}) = \text{if } n=0 \text{ then } \text{ans} \text{ else } \text{fac1}(n-1, n\times\text{ans})
\]

However, given "foo" as below:

\[
\text{foo } n = \text{if } n=0 \text{ then } 1 \text{ else } n - \text{foo}(n-1)
\]

we cannot write it as

\[
\text{foo1 } n \text{ ans } = \text{if } n=0 \text{ then } \text{ans} \text{ else } \text{foo1 } (n-1) \text{ (n-ans)}
\]

with the top-level call being "foo1 n 1".

For example, we will get:

\[
\text{foo } 2 \Rightarrow 2 \quad \text{versus} \quad \text{foo1 } 2 \text{ 1 } \Rightarrow 0
\]

\[
\text{foo } 4 \Rightarrow 3 \quad \text{versus} \quad \text{foo1 } 4 \text{ 1 } \Rightarrow -1
\]

\[
\text{foo } 19 \Rightarrow 9 \quad \text{versus} \quad \text{foo1 } 19 \text{ 1 } \Rightarrow 9 \quad \text{(here they agree :)}
\]

What went wrong? In converting \(\text{fac}\) to \(\text{fac1}\) we assumed that \(\times\) is commutative and associative; this is not allowed for \(-\). So, in designing algorithms (as well as proving properties thereof), one often takes advantage of known semantic properties of operators.

To see that such semantic properties were used in the above transformation, read pages 220 onwards in Paulson’s book. There, it is proved that \(\text{facti}(n,1) = n!\) for all \(n \geq 0\). This proof is accomplished by first showing \(\text{facti}(n,p) = n!*p\) by inducting on \(n\).

\[
\text{facti}(n+1, p) = \text{facti}(n, (n+1)*p) \quad [\text{defn of facti}]
\]

\[
= n! * ((n+1)*p) \quad [\text{ind hyp}]
\]

\[
= (n! * (n+1)) * p \quad [\text{assoc of *}]\]
\[(n+1)! \times p\] [arithmetic]

There is a general method for converting linear schema to iterative schema using Patterson and Hewitt’s theorem, quoted from Johnson, ”Synthesis of Digital Designs from Recursion Equations”, MIT Press, 1983, page 36:

\[
\begin{align*}
F(a) & \text{ where } \\
F(x) & \leq p(x) \rightarrow f(x), h(x, F(g(x))) \\
\end{align*}
\]

Can always be translated to

\[
\begin{align*}
G(a,\_,\_,\_,\_,\_) & \\
\end{align*}
\]

where

\[
\begin{align*}
G(u, v, x, y, z) & \leq p(x) \rightarrow f(x), \ \\
& \quad L(u, \_, u, \_, f(x)), \\
& \quad G(u, \_, g(x), \_, \_) \\
L(u, v, x, y, z) & \leq p(x) \rightarrow z, \\
& \quad M(u, g(x), g(x), u, z) \\
M(u, v, x, y, z) & \leq p(x) \rightarrow h(y, z), \\
& \quad M(u, v, g(x), g(y), z) \\
\end{align*}
\]

I’ve stated the above theorem using ”\_” that stands for ”don’t care” to be faithful to Steve Johnson’s notation. In reality (in the SML code below) I don’t need these ”\_”.

Since this translation doesn’t use any semantic property of ”\*” (that it is commutative or associative), we can automate tail-recursion optimization using this trick in a compiler that doesn’t, in general, know semantic properties of operators.

For example, consider the factorial recursion:

\[
F(x) \leq p(x) \rightarrow f(x), h(x, F(g(x)))
\]

In SML, it is:
fun fac x = if x=0 then 1 else x * fac(x-1);
and is invoked as fac 12; Using the above theorem, we should be able to write fac as

\[ G(u, x) \leftarrow p(x) \rightarrow L(u, u, f(x)), G(u, g(x)) \]
\[ L(u, x, z) \leftarrow p(x) \rightarrow z, M(u, g(x), g(x), u, z) \]
\[ M(u, v, x, y, z) \leftarrow p(x) \rightarrow L(u, v, h(y,z)), M(u, v, g(x), g(y), z) \]

or, in SML

fun G(u, x) = if x=0 then L(u, u, 1) else G(u, x-1)
and

fun L(u, x, z) = if x=0 then z else M(u, x-1, x-1, u, z)
and

fun M(u, v, x, y, z) = if x=0 then L(u, v, y * z) else M(u, v, x-1, y-1, z)
;

This can be optimized to

fun G(x) = L(x, x, 1)
and

fun L(u, x, z) = if x=0 then z
else M(u, x-1, x-1, u, z)
and
M(u,v,x,y,z) = if x=0
then L(u, v, y * z)
else M(u, v, x-1, y-1, z)
;

fac 3 becomes G(3) which evaluates as:

G(3)
L(3,3,1)
M(3,2,2,3,1)
M(3,2,1,2,1)
M(3,2,0,1,1)
L(3,2,1*1)
M(3,1,1,3,1*1)
M(3,1,0,2,1*1)
L(3,1, 2*(1*1) )
M(3,0,0,3, 2*(1*1) )
L(3,0, 3*(2*(1*1)) )
3*2*(1*1)

Notice how the "order" of multiplication is preserved.
Similarly, function "foo" becomes

fun foo(x) = L(x, x, 1)

and

L(u,x,z) = if x=0 then z
else M(u, x-1, x-1, u, z)
and

M(u, v, x, y, z) = if x=0 then L(u, v, y - z) 
else M(u, v, x-1, y-1, z)
;

and works correctly!

A few general facts to know:

- Any system of recursive equations (involving perhaps mutual recursion) can always be cast as an instance of the universal non-linear schema

\[ F(x) \leftarrow p(x) \rightarrow f(x), \ h( F(g1(x)), F(g2(x)), \ldots, F(gN(x)) ) \]

Such non-linear schema have, in general, no equivalent linear schema (under all interpretations of the primitive operations, that is).

- The universal linear schema

\[ F(x) \leftarrow p(x) \rightarrow f(x), \ h( F(g(x)) ) \]

has an equivalent iterative schema that can be implemented without a run-time stack for any interpretation of the basic operations.

5 Module 3: Induction and Recursion

5.1 Principles of Induction

5.2 Reasoning about Recursive Programs

Example 1:

Consider the program

\[ E \ x = \ if \ (x=0) \ then \ x \ else (E(x-1)) + 2 * x - 1 \]

Prove that

\[ E = (fn \ x \Rightarrow x \ * \ x) \].
In the subgoal induction method, for every function defined, we invent a "psi" relation that relates its input to its output. We denote outputs by "z".

So we want to prove that

\[ \text{psi}_E \ x \ z = (z = x \ast x) \]

Read this as: "the relation psi_E is true of program E exactly when program E when run with input x produces output z such that \( z = x \ast x \)".

Subgoal induction proceeds by generating one VC for every recursive- or basis-case taken by a recursive program.

VC1 for the basis case:

\( (x = 0) \Rightarrow (\text{psi}_E \ x \ x) \)

VC2 for the recursive call:

\( (x \neq 0) \land (\text{psi}_E \ (x-1) \ w) \Rightarrow (\text{psi}_E \ x \ (w+2*x - 1)) \)

VC1 simplifies to:

\( (x=0) \Rightarrow (0 = 0*0). \text{ True.} \)

VC2 simplifies to:

\( (x>0 \land (w = (x-1)^2)) \Rightarrow (x^2 = w + 2*x - 1) \)

i.e.

\( x > 0 \Rightarrow (x^2 = (x-1)^2 + 2*x - 1). \text{ True.} \)

**Example 2:**
Read \( x \rightarrow t \) as "t divides x"

Define

\[ \text{psi}_{gcd}(x,y;z) = \]

\[ (z = \max \{ t : t \mid x \land t \mid y \}) \]

Consider the while program:

\[ \text{INPUT} \ x \text{ and } y \]

\[ \text{while } x \neq y \text{ do} \]

\[ \text{begin} \]

\[ \text{if } x < y \text{ then } y := y \ast x \]
else \[ x := x - y \]
end

**ANSWER x**

Show that the final value of \( x \) is the GCD of \( x \) and \( y \).

Generate the VCS:

VC1: \( \text{psi\textunderscore gcd}(x, y-x; z) \land (x < y) \land x < y \Rightarrow \text{psi\textunderscore gcd}(x, y; z) \)

VC2: \( \text{psi\textunderscore gcd}(x-y, y; z) \land (x > y) \land x < y \Rightarrow \text{psi\textunderscore gcd}(x, y; z) \)

Take VC1; It says:

\[ x | z \text{ and } (y-x) | z \text{ and } z \text{ is the largest such number} \]

\[ \land \]

\[ x < y \]

\[ \Rightarrow x | z \text{ and } y | z \text{ and } z \text{ is the largest such number} \]

It is easy to see that \( x-z \) and \( y-z \).

If there exists a ‘p’ such that \( x-p \) and \( y-p \) and \( p \mid z \)
then we also have

\[ x | p \text{ and } (y-x) | p \text{ and } p > z \]

contradicts the antecedent.

So VC1 is true.

Similarly VC2 is true.

**Example 3:**

Consider an example involving nested recursive calls. These functions \( G \) and \( F \) both “flatten” a given list. For example, given \((\text{a (b 2)) (3 4)})\), the result will be \((\text{a b 2 3 4})\). \( F \) needs to be called with “nil” for \( y \) to be equivalent to \( G \).

To show that the call \( G(x) \)

where

\[ G(x) \leftarrow \begin{cases} \text{if atom}(x) \\
\text{then cons}(x, \text{nil}) \\
\text{else append(} \ G(\text{car}(x)), \ G(\text{cdr}(x)) \ )
\end{cases} \]
yields the same result as the call F(x,nil)
where
\[
F(x,y) \leftarrow \text{if } \text{atom}(x) \text{ then } \text{cons}(x,y) \text{ else } F(\text{car}(x), F(\text{cdr}(x), y))
\]

proceed as follows.

Method:
1. Treat "G" as the "desired behavior" of "F".
2. Also, prove a slightly more general result:

\[
\text{psi}_F (x,y; z) = (z = \text{append}(G(x), \text{y}))
\]

If this is proven, then specializing y to nil gives us

\[
\text{psi}_F (x,\text{nil}; z) = (z = G(x)).
\]

This idea of generalizing the proof will come later when we study the pattern matcher. This generalization is needed because "internally" the function does use the "y" argument with non-nil values.

Generate the verification conditions for F:

VC1: atom(x) \rightarrow \text{psi}_F(x,y; \text{cons}(x,y))

i.e.

atom(x) \rightarrow ( \text{cons}(x,y) = \text{append}(G(x),y) )

from the definition of G, we get G(x) = \text{cons}(x,\text{nil}) when x is an atom.

Thus, atom(x) \rightarrow ( \text{cons}(x,y) = \text{append}(\text{cons}(x,\text{nil}), y) )

This is true, using the standard definition of append.

VC2: \neg atom(x) \vee \text{psi}_F(\text{cdr}(x),y; z1) \vee \text{psi}_F(\text{car}(x),z1; z)

\rightarrow

\text{psi}_F(x,y; z)

Expanding, we have to prove:
\( \text{atom}(x) \land z = \text{append}(G(\text{car}(x)), \text{append}(G(\text{cdr}(x)), y)) \)

\[ \Rightarrow z = \text{append}(G(x), y) \]

using \( \text{atom}(x) \), simplify \( G(x) \) of consequent. This yields

\( \text{atom}(x) \land z = \text{append}(G(\text{car}(x)), \text{append}(G(\text{cdr}(x)), y)) \)

\[ \Rightarrow z = \text{append}(\text{append}(G(\text{car}(x)), G(\text{cdr}(x))), y) \]

which follows from the associativity of append - A LEMMA TO BE GUESSED AND PROVEN BY THE USER, perhaps through a separate induction carried out on the definition of append.

**Example 4:**
In general, the VCs need to be a bit stronger. Example that illustrates this fact:

Consider

\[ H(x) \leftarrow \text{if } x=0 \]
\[ \phantom{H(x)} \text{then 1} \]
\[ \phantom{H(x)} \text{else } H(x-1) - H(x-1) \]

To show that

\[ H(x) \not\leftrightarrow 2. \text{ (Just for example!)} \]

\[ \text{psi}_H(x; z) = (z \not\leftrightarrow 2) \]

**VC1:** \( (x=0) \Rightarrow \text{psi}_H(0; 1) \), which is true because \( 1 \not\leftrightarrow 2. \)

**VC2:** \( (x \not\leftrightarrow 0) \land \text{psi}_H(x-1; z1) \land \text{psi}_H(x-1; z2) \Rightarrow \text{psi}_H(x; z1-z2) \)

i.e. show

\( (x \not\leftrightarrow 0) \land z1\not\leftrightarrow 2 \land z2\not\leftrightarrow 2 \Rightarrow (z1-z2)\not\leftrightarrow 2 \)

which is not true in general!

Why doesn’t it work?

Well, we have to state one more fact:

Whenever the various recursive calls on the RHS have the same argument, these calls yield the same result
In other words, we have to state that the "psi" relations are functions!
Stating this gives us

\[(x-1) = (x-1) \Rightarrow (z1 = z2)\]

and so the proof can be completed, using \[z1 = z2\].

**Example 5:**
The proof of the "match function" from Gordon’s book follows next. Here, we will prove that the pattern-matching function from Gordon’s prover, `matchfn`, works correctly. The function is reproduced below:

```lisp
;; matchfn is a function that matches a pattern against an expression in the context of a substitution. For example
;;
;; (matchfn '(x 2 z) '(1 2 3) nil)
;; yields ((z . 3)(x . 1))
;;
;; and
;;
;; (matchfn '((x + y) * (z * y)) '((3 + 1) * (3 * 1)) '((z . 3)))
;; yields ((y . 1) (x . 3) (z . 3))
;;
(defun matchfn (pat exp sub)
  (if (atom pat)
      (if (is-variable pat)
          (if (assoc pat sub)
              (if (equal (cdr (assoc pat sub)) exp)
                  sub
                  (throw 'fail 'FAIL))
              (cons (assoc pat exp) sub))
          (if (eq pat exp) sub (throw 'fail 'FAIL)))
      (if (atom exp)
          (throw 'fail 'FAIL)
          (matchfn (rest pat) (rest exp) (matchfn (first pat) (first exp) sub))))
```
Let us extend all our functions to be strict, so that \((\text{fn FAIL}) = \text{FAIL}\) for any function \(\text{fn}\).

The definition of \texttt{matchfn} is displayed by the following tree:

Here, "lu" stands for "lookup" which is a short-form for \((\text{cdr (assoc pat sub)})\)
"mf" stands for "matchfn", "fst" for "first", "rst" for "rest", and "\(=\)" for "equal".

Fig. 3. The Execution-tree of the Pattern-Matching Function (Gordon text, p.183)

I’ll now type-set the proof in the \texttt{teletype} font, using \texttt{psi} to denote the I/O relation that \texttt{matchfn} supports.

Let us guess a \texttt{psi} relation:

\[
\text{Define } (\text{psi } p \ e \ s; a) = (a \leftrightarrow \text{FAIL}) \land (s = \text{nil}) \Rightarrow (\text{sub } a \ p = e )
\]

This says if we take the results of the \texttt{psi} call (which is \texttt{a}) and apply it back on the pattern \texttt{p}, we get back the original expression \texttt{e}. Also, \texttt{matchfn} is expected to be called with \texttt{(s = nil)} by its users; hence the proviso \texttt{(s = nil)} is used in the definition of \texttt{psi}. 
Let us number the paths by PATH DESCRIPTORS which are strings of zeros and ones with a zero representing a left-turn and a one representing a right-turn. For example, the path leading up to (p.e.).s is 110 (two right-turns and one left-turn).

We will generate the VCs for path 00 and prove that these VCs are valid.

SHOW

\[
\begin{align*}
& (\text{psi} \ (\text{fst} \ p) \ (\text{fst} \ e) \ s; \ a) \\
& \quad \backslash \ (\text{psi} \ (\text{rst} \ p) \ (\text{rst} \ e) \ a; \ b) \\
& \quad \backslash \ (\text{atom} \ p) \\
& \quad \backslash \ (\text{atom} \ e) \\
& \quad \backslash \ [ \ (\text{fst} \ p) = (\text{rst} \ p) \ ] \ \backslash \ [ \ (\text{fst} \ e) = (\text{rst} \ e) \ ] \ \backslash \ (s = a) \\
& \quad \Rightarrow \ (a = b) \\
& \Rightarrow \ (\text{psi} \ p \ e \ s; \ b)
\end{align*}
\]

EXPANDING USING THE DEFINITION OF PSI, WE HAVE

\[
\begin{align*}
& (a <> \text{FAIL}) \ \backslash \ (s = \text{nil}) \Rightarrow \ (\text{sub} \ a \ (\text{fst} \ p)) = (\text{fst} \ e) \\
& \quad \backslash \ (b <> \text{fail}) \ \backslash \ (a = \text{nil}) \Rightarrow \ (\text{sub} \ b \ (\text{rst} \ p)) = (\text{rst} \ e) \\
& \quad \backslash \ \ldots \text{other conjuncts} \ldots \\
& \Rightarrow \ (b <> \text{fail}) \ \backslash \ (s = \text{nil}) \Rightarrow \ (\text{sub} \ b \ p) = e
\end{align*}
\]

THIS PROOF WON'T GO THRU (try it!). The reason is that we will have to account for a <> nil also, in our reasoning. Currently, psi doesn't tell us anything about the behavior of matchfn when its last argument is non-nil.
REALIZING that matchfn is INTERNALLY used with a non-nil last argument, CHANGE the definition of psi to

\[ \text{psi p e s; a) = (a <> FAIL) => (sub a p) = e } \]

THIS is true because ‘‘a’’ incorporates whatever we feed into the last argument ‘‘s’’ of matchfn. By INCORPORATES, we mean that matchfn will check that ‘‘s’’ is indeed consistent with the bindings generated by matchfn before it extends ‘‘s’’ to obtain ‘‘a’’.

Now, the VC becomes

\[ (a <> FAIL) => (sub a (fst p)) = (fst e) \]
\[ /\ (b <> fail) => (sub b (rst p)) = (rst e) \]
\[ /\ ~\text{atom p} \]
\[ /\ ~\text{atom e} \]
\[ /\ [ (fst p) = (rst p) ) /\ (fst e) = (rst e) ) /\ (s = a) ] \]
\[ => (a = b) \]
\[ => (b <> fail) => (sub b p) = e ) \]

DEFINE CT = [ (fst p) = (rst p) ) /\ (fst e) = (rst e) ) /\ (s = a) ]
\[ => (a = b) \]

Now, if a = FAIL or b = FAIL, the final answer will be FAIL; this is not the interesting case. So then, here’s how I shall try to prove the above VC:

ASSUME

\[ (a <> FAIL) /\ (b <> fail) \]
\( (\text{sub}\ a\ (\text{fst}\ p)) = (\text{fst}\ e) \)
\( (\text{sub}\ b\ (\text{rst}\ p)) = (\text{rst}\ e) \)
\(~(\text{atom}\ p)\)
\(~(\text{atom}\ e)\)
\text{CT}

**PROVE**
\( (\text{sub}\ b\ p) = e \)

**METHOD**

- Unfold the consequent of the implication
- Simplify using the antecedent

SO then, using ‘‘\( \Rightarrow \)’’ to read ‘‘is the same as asking whether’’, we have

\((\text{sub}\ b\ p) \neq e\)

\(\Rightarrow (\text{sub}\ b\ ((\text{fst}\ p).\ (\text{rst}\ p))) \neq e\)

\(\Rightarrow \text{by distributing ‘‘sub’’ over ‘‘cons’’}\)

\( (\text{sub}\ b\ (\text{fst}\ p)).\ (\text{sub}\ b\ (\text{rst}\ p)) \neq e\)

\(\Rightarrow (\text{sub}\ b\ (\text{fst}\ p)).\ (\text{rst}\ e) \neq e\)

WE ARE DONE if we can show that \((\text{sub}\ b\ (\text{fst}\ p))\) is \((\text{fst}\ e)\)...

HOWEVER, this is impossible under the assumed psi !!

ACTUALLY we came close. We had \((\text{sub}\ a\ (\text{fst}\ p)) = (\text{fst}\ e)\) in our assumptions. Wish we had \((\text{sub}\ b\ (\text{fst}\ p)) = (\text{fst}\ e)\)...

IDEA! (suggested by the above failed proof): Does ‘‘\( b \)’’ contain an ‘‘\( a \)’’ WITHIN IT? If so, we are done!

YES, it does! Notice that ‘‘\( a \)’’ is an intermediate answer, and ‘‘\( b \)’’
does incorporate ‘‘a’’ into it.

We didn’t specify this in our psi.

We could have said

\[(\text{psi p e s; a}) = (a \leftrightarrow \text{FAIL}) \Rightarrow ((\text{sub a p}) = e) \land (s = a)\]

where \([=\) is defined in the sense that ‘‘a’’ is an alist that contains all the substitutions in ‘‘s’’, plus some more.

**USING THE ABOVE psi, we now have the task:**

**ASSUME**

\[
(\text{sub a (fst p)) = (fst e) ) \\
\land s = a
\]

\[
\land (\text{sub b (rst p)) = (rst e) ) \\
\land a = b
\]

\[
\land \sim(\text{atom p)} \\
\land \sim(\text{atom e)} \\
\land (a \leftrightarrow \text{FAIL)} \\
\land (b \leftrightarrow \text{fail)} \\
\land \text{CT}
\]

**PROVE**

\[(\text{sub b p}) = e ) \land s = b
\]

We have \(s = a\) and \(a = b\). So \(s = b\).

So need to prove \((\text{sub b p}) = e )\) only.

**BUT, (sub b p) = (sub b (fst p)) . (rst e), as shown earlier.**

Now we can write

\[(\text{sub b (fst p))}\]
as

(sub (b \ a) (sub a (fst p)))

i.e. as a sequential substitution of two binding lists.

NOW if we can make sure that (b \ a) doesn't provide any more
substitutions into (fst p) than \('a'\) does, then we can reduce
the above to (sub a (fst p)) which we know to be the same as
(fst e), and then we will be able to complete our proof.

THIS IS ONE ASPECT that we missed from psi.
We also need to say that \('a'\) is consistent too - i.e. it doesn't
have any conflicting bindings in it.

\[
(psi p e s; a) = (a <> FAIL) =>
(sub a p) = e
/\ s [= a
/\ for all v in vars(p), there exists w such that
(v . w) in a
/\ consistent(a)
\]

Using this psi, we can simplify

(sub (b \ a) (sub a (fst p))) to (sub a (fst p))
which is (fst e), which ends our proof.

Proof of a topological sorting function
The topsort function from Paulson's book, page 105 is as follows:

fun topsort graph =
  let fun sort ([], visited) = visited
        | sort (x::xs, visited) =
          sort(xs, if x mem visited then visited
               else x :: sort(nexts(x, graph), visited))
  val (starts, _) = ListPair.unzip graph
in
  sort(starts, [])
end;

I now present a hand-proof that this topsort function “works”. In the process, we identify what “works” means, too. In fact, verification is largely about trying to clearly state what “works” means! The first thing to do in attempting any hand-proof is to simplify the notations so that writing tens of pages of proof becomes a tolerable task. For this, I shall abbreviate function sort using \( s \), function topsort using \( \text{ts} \), and so on. In these proofs, one typically sketches many failed proofs, tries to “cheat” by thinking of patches, until one realizes what was forgotten to be stated, so patience is an important pre-requisite.

First, rewrite the function \( s \) as

\[
\begin{align*}
  s([], v) &= v \\
  s(x::xs, v) &= \\
  &\quad \text{if } x \in v \\
  &\quad \text{then } s(xs, v) \\
  &\quad \text{else } s(xs, x :: s(\text{nexts}(x, g), v))
\end{align*}
\]

\( \text{ts } g = s(\text{nonterm}(g), []) \)

Notice how I have pulled out the \if\ to the top level. This is a legal transformation if sort is \emph{strict} in its second argument (being \emph{strict} means it forces the second argument to be evaluated—which is also what the transformation does). The top-level call to \text{ts} also is written with our abbreviations, i.e. as \( s(\text{nonterm}(g), []) \), which says “take the non-terminals of graph \( g \) and apply \( s \) with accumulator \( [] \)”.

After much trial and error (lasting many hours and about 12 pages), I realized the true correctness assertion associated with \( s \).

To give you an idea of an incorrect one that I struggled with a lot, here it is:

\[
\psi_s(L, v \ ; A) = \text{topsort}(v) \Rightarrow \text{topsort}(A) \\
\quad \wedge \\
\quad A \text{ in } \Pi(L \ \text{Union } v)
\]

Here, I mean

For any list \( v \),

\[
\text{topsort}(v) = \wedge \text{ \( v \) is a subset of the nodes of the underlying graph } g \\
\quad \wedge \text{ \( \text{for every } \text{n1, n2 in } v \) }
\]
and 
\[ n_1 \rightarrow^* n_2 \text{ in } g \]
we have 
\[ n_2 \text{ listed after } n_1 \text{ in list } v. \]

That is, I said, “well, \( s \) is taking a list \( L \) and an accumulated list \( v \). We had better ensure that \( v \) is not garbage at any time, or else we cannot guarantee any predictable outcome of \( s \)”. It also is clear by studying the program that there are two uses made of \( v \):

1. It records the visited set of nodes.
2. \( v \) is kept topologically sorted, too.

Thus, I wrote “if \( \text{topsort}(v) \), then the final answer \( A \) is such that \( \text{topsort}(A) \) and it is a permutation (\( P_i \)) of the nodes in \( L \) and \( v \)”. This proof proceed considerably, but ended up getting stuck in an awkward situation always. Here was that awkward situation:

If \( A_1 \) represents the answer produced by 
\[ s(\text{nexts}(x, g), v), \text{ i.e. } A_1 \text{ in } P_i(\text{nexts}(x, g) \cup v) \]

and \( A_2 \) represents the answer produced by 
\[ s(xs, x :: A_1), \text{ i.e. } A_2 \text{ in } P_i( xs \cup x :: A_1) \]

we had to show that 
\[ A_2 \text{ in } P_i( x :: xs \cup v) \]

If you think through this, you will find that we had to prove
\[ \text{nexts}(x,g) \subseteq (v \cup x :: s) \]

This simply would not go thru! The following example was constructed to discover that this proof is futile:

\[ \begin{array}{c}
  a \\
  / \ \\
  / \ \\
  v \quad v \\
  \quad b \quad c
\end{array} \]
Consider \( x \) to be \( b \). Let \( v \) be the empty-set (e.g., when we make the first call to \( s(\text{nexts}(x, g), v) \). Let us also have \( xs \) be \( \{b, c\} \). Then, \( \text{nexts}(x, g) \) is \( \{e, d\} \) while \( v \) Union \( xs \) is \( \{b, c\} \).

Not only that, the top-level call to \( ts \) is also going to land in trouble, because in the call \( ts \ g = s(\text{nonterm}(g), [] \), we pass only the non-terminals of \( g \). Clearly a correct answer can’t merely include the non-terminals of \( g \) alone—it must include the terminals of \( g \) also!

The correct correctness-assertion was then written:

\[
\psi_s(L, v; A) = \text{topsort}(v) \Rightarrow \text{topsort}(A)
\]

\[
\wedge
A \text{ in } \Pi(L \rightarrow* \text{ Union } v)
\]

Here, \( L \rightarrow* \) means all nodes reachable in zero or more steps from \( L \). Let us now do the sub-goal induction proof. Note that in this whole proof, we suppress the “cross-term”, as it was thought to be (and found to be) not useful:

**VC1,** derived from path \( s([], v) = v: \)

\[\text{To show } \psi_s([], v; v).\]

\[\text{I.e.}\]

\[\text{topsort}(v) \Rightarrow \text{topsort}(v) \]

\[\wedge\]

\[v \text{ in } \Pi([], \rightarrow#* \text{ Union } v)\]

which is true.

**VC2,** derived from path \( \text{if } x \text{ in } v \text{ then } s(x::xs, v) = s(xs, v): \)

\[\text{To show}\]
\[ x \text{ in } v \]
\[ \psi_s(xs, v ; A) \]

\[ \Rightarrow \]
\[ \psi_s(x :: xs, v ; A) \]

i.e.

/------- Lamport's notation for conjunction.
\[ x \text{ in } v \]
\[ v \]
\[ \topsort(v) \Rightarrow \topsort(A) \]
\[ \wedge A \text{ in } \Pi(xs \rightarrow \ast \text{ Union } v) \]

=============================================== <-- this is the proof notation 'from things above === show things below ==='. Things above the line are called antecedents while things below the line are called consequents.

\[ \topsort(v) \Rightarrow \topsort(A) \]
\[ \wedge A \text{ in } \Pi(x :: xs \rightarrow \ast \text{ Union } v) \]

The line of reasoning is to assume \( \topsort(v) \) (which only makes the consequent non-vacuously true) - then we argue as follows.

I initially argued...

> Since \( x \) is in \( v \), and hence
>
> (\( x :: xs \)) \rightarrow \ast \text{ Union } v
>
> is the same set as \( xs \rightarrow \ast \text{ Union } v \),
>
> we are done!
but then Milan Ikits pointed out an error, and hence the fixed proof for this piece is as follows (shown against >>)

>> Since x is in v, and hence
>> (x::xs)-->* Union v
>> is the same set as xs-->* Union v,
>> PROVIDED v is ‘‘complete’’ wrt g in the sense that
>> if a node x is in v, then x-->* is also in v.
>> WE need to assert this in our definition of topsort.
>> we are done!

VC3, derived from path if x notin v then
----------
s(xs, x : : s(nexts(x,g), v)) :
----------

\ \ x notin v
\ \ psi_s( nexts(x,g), v ; A1 )
\ \ psi_s( xs, x : : A1 ; A2 )

i.e.
\ \ x notin v
\ \ topsort(v) => \ \ topsort(A1)
\ \ A1 in Pi( nexts(x,g)-->* Union v )
\( \text{topsort}(x::A1) \Rightarrow \text{topsort}(A2) \)
\[ \land \ A2 \text{ in } \Pi(x::xs->* \text{ Union } (x::A1)) \]

=================================

top\text{sort}(v) \Rightarrow \text{topsort}(A2)
\[ \land \ A2 \text{ in } \Pi(x::xs->* \text{ Union } v) \].

Again, proceed in the proof by assuming topsort(v).

Then, we have topsort(A1), from the antecedent.

Then we have topsort(x::A1) true, because A1 is a permutation over the set of states reachable from nexts(x,g) Union v

and since x is not in v,

and since nexts can’t create a cycle back to x

x has not appeared in nexts(x,g) Union v

but appears FIRST in x::A1.

Thus, topsort(A2), thus, finishing off one piece of the consequent.

Now, all we need to prove is

A2 in Pi(x::xs->* Union v), which

follows from the facts that

A1 in Pi( nexts(x,g)->* Union v )

and

A2 in Pi(xs->* Union (x::A1))
(draw a diagram out, and you will be convinced!)

The reasoning is as follows:

A1 in Pi( nexts(x,g)->* Union v )
and
A2 in Pi(xs->* Union (x::A1))

A2 in Pi( (xs->*) Union (x :: ( nexts(x,g)->* Union v )) )

Since nexts(x,g)->* and x together are equivalent to (x::xs)->*,
we have

A2 in Pi( ( (x::xs)->*) Union v ).

5.3 A taste of fixed-points and the Lambda calculus

So far in these notes, we looked at specifying the meaning of imperative and functional programs using auxiliary assertions that are easy to understand, and showed how to prove these assertions to be true (and in what sense we prove them true). Now we begin looking at the implementation and evaluation of programs in general, and functional programs in particular. Our discussion will also involve a few issues pertaining to typing and modularization. Rather than digress, I shall mention these as we go along, and revisit these issues much later.

5.4 “What’s in a Name”, or Why the Lambda Notation?

We denote numbers by their names (numerals). It is crazy to refer to numbers by names such as “Fred” because the numeral sequence spells out what the number denoted is—e.g. “1998”.

In a framework where functions and “other values” are treated alike, we must follow this convention—of not attaching arbitrary names to functions. Yet, we break this convention in most languages, thus:

function Fred (x) = x+1

You might say that this can be easily fixed by “de-Freding” the function definition thus:
function (x) = x+1

Thus was born the Lambda notation—there, we can say \( \lambda x. x + 1 \).
But, how do we treat recursive function definitions?

function fred x = if x=0 then 0 else fred(x-1)

Can we say this?

function(x) = if x=0 then 0 else self(x-1)

But what does “self” mean? (We will see this concept when later we talk about OO programming also...). This example will serve to illustrate the fixed-point combinator, \( Y \).
\( Y \) finds the least fixed-point of its argument.
Here is how we can learn to systematically “de-Fred” the recursive definition. But before we do that, let us look at another function “Bob”:

function Bob(x) = Bob(x+1)

Though this is “nonsense” from the programming point of view, it is good to know in what exact sense it is “nonsense” (if we don’t, we may miss other nonsense that cannot be discerned through casual visual inspection only). This example will serve to illustrate the fact that if-then-else is not strict in its arguments.

5.5 An Aside—the Lambda Calculus

Kindly read pages 372-378 and 384-395 from our book. Mike Gordon’s “Programming Language Theory and its Implementation” is also an excellent reference for Lambda calculus.

The lambda calculus was invented by Alonzo Church as a formal representation of computations. Church’s Thesis tells us that the lambda-based evaluation machinery, Turing machines, as well as other formal models of computation (Post systems, Thue systems, ...) are all formally equivalent. Formal equivalences between these systems have all been worked out by the 1950s.

Basically there are three kinds of Lambda terms:

\[
t ::= \quad x \quad \text{a variable} \\
| \lambda x \, t \quad \text{functional abstraction} \\
| t \, t \quad \text{function application}
\]
There are also two conversions available—the alpha conversion, and the beta conversion. An example of alpha conversion is

\[
\text{lambda } x \cdot x + 1 \rightarrow \text{lambda } y \cdot y + 1
\]

In other words, one can rename the formal parameters of a function together with all occurrences thereof in the function body without changing the meaning of the function. An example of beta conversion is

\[
(\text{lambda } x \cdot x + 1) \ 23 \rightarrow 23 + 1 \rightarrow 24
\]

You may be baffled that I suddenly use “23” and “+” as if they were Lambda terms. Please see our book to know how these can be encoded using Church numerals. Thus anything that you wish to compute is formally definable in the Lambda calculus.

5.6 Back to the main issue—De-Freding Recursive Functions

Let us write our Fred function as an equation:

\[
\text{fred} = \text{lambda } x. \ \text{if } x = 0 \ \text{then } 0 \ \text{else } \text{fred}(x-1)
\]

OK, now we are getting somewhere. We are looking for a solution for the above equation. We can also glean that function Bob(x) = Bob(x+1) has no such solution possible for Bob (for brevity, we say “normally speaking”—wierd cases ruled out. It is easy—but takes more time—to be exact, too).

Now what we have can be expressed as follows:

\[
\text{fred} = (\text{lambda } \text{fred}'. \ \text{lambda } x. \ \text{if}(x=0) \ \text{then } 0 \ \text{else } \text{fred}'(x-1)) \ \text{fred}
\]

Yay, we have almost eliminated the redundant information “fred”. What we have is

\[
\text{fred} = (\text{lambda } y. \ \text{lambda } x. \ \text{if}(x=0) \ \text{then } 0 \ \text{else } y(x-1)) \ \text{fred}
\]

This, in mathematics, describes a situation where \text{fred} is a fixed-point of

\[
(\text{lambda } y. \ \text{lambda } x. \ \text{if}(x=0) \ \text{then } 0 \ \text{else } y(x-1))
\]

A fixed-point of a function \(F\) is a value \(x\) such that \(F \ x = x\). For example, a fixed-point of \(!\), i.e. the mathematical function factorial defined over natural numbers, is 1 (\(1! = 1\)). Another way to see a fixed-point is to type something random on your calculator and keep hitting the \(\sin\) key till the display doesn’t change. (Dana Scott mentions somewhere that I read how his young daughter once thought the calculator
was broken! You can easily see that some functions don’t have fixed-points (type in \texttt{7777} and keep hitting \textit{sin} first; then try with hitting \textit{log}; which finds a fixed-point and which doesn’t)? Of course, multiple fixed-points are also possible: for function \((\texttt{lambda} \, x)\), there are as many fixed-points as the domain (type) of \(x\) has members.

So considering \texttt{fred}, how many fixed-points are there for the following function, and how to find them?

\begin{verbatim}
(lambda y . lambda x . if(x=0) then 0 else y(x-1))
\end{verbatim}

There is a “magic” function \texttt{Y}, usually written as follows, that \textit{finds the least fixed-point} of any function given as its argument:

\begin{verbatim}
(lambda x . (lambda h . x(h h)) (lambda h . x(h h)))
\end{verbatim}

Let us check it out; for an arbitrary \(f\), we want

\[
Y \ f = f(Y \ f).
\]

Here is how the proof goes:

\[
Y \ f = (\lambda x . (\lambda h . x(h h)) (\lambda h . x(h h))) \ f
\]

\[
= (\lambda h . f(h h)) (\lambda h . f(h h)) <----| \]

\[
= (\lambda h . f(h h)) (\lambda h . f(h h)) \ | \]

\[
= f( (\lambda h . f(h h)) (\lambda h . f(h h)) ) ----
\]

\[
= f( Y \ f )
\]

Thus, finally, the “de-Freded” form for our function Fried function is

\[
Y (\lambda y . \lambda x . if(x=0) then 0 else y(x-1))
\]

We will later justify that it is the \textit{least} fixed-point that we want—given the option of multiple fixed-points. Of course there are many many problem-domains where the \textit{greatest} fixed-points are the ones we want. We will look at some of those later.

\section{5.7 An exercise with \texttt{Y}}

OK, if \texttt{Y} can find the fixed-point of “any function”, can it find a fixed-point of \textit{succ}—the successor function defined over integers?
\[ Y \text{ succ} = \text{ succ} (Y \text{ succ}) \\
= \text{ succ} (\text{ succ}(Y \text{ succ})) \\
= \text{ succ} (\text{ succ}\text{ (\text{ succ}(Y \text{ succ}))}) \\
= \ldots \]

Clearly, we get a divergent computation. Such divergent computations are treated as returning a “special” value called “bottom” (⊥). This will be discussed again when we talk about applicative order evaluation, normal order evaluation, head normal-forms and such—in Assignment 3.

5.8 Verification of pipelined systems

We will now consider some of the basic techniques employed for verifying processors in general, and pipelined processors in particular. The concepts involved in these exercises are few in number, and are widely applicable. Hence we will begin our discussion with a few simple examples.

An Overview of Verification Criteria for Datapath Circuits The term “datapath circuits” denotes all those circuits that perform computations similar to a processor. Members of this family include general purpose processors as well as special purpose processors such as priority queues, LRU units, microprogram engines, etc.

There are many different ways to model datapath circuits. One conceptually clear way (developed by many, including myself during my PhD dissertation) is to view them as abstract data types. For example, an LRU unit viewed as an abstract data type (ADT) has the following constructors that create its states: \text{ init} to initialize the LRU, and \text{ use}(\text{ S}, \text{ a}) to record the usage of address \text{ a} with respect to state \text{ S}. An observer for the LRU is \text{ least}(\text{ S}) that observes a given state \text{ S} and returns the LRU value with respect to this state.

Once upon a time, researchers in this area used to advocate a purely algebraic style to specifying datatypes. However, their examples seldom went beyond stacks and queues. The author considers the LRU as an eminent example of a system whose purely algebraic style specification is pretty interesting (and nearly useless in terms of clarity). One interesting fact about LRU is that it resembles a queue except it needs to “know” the oldest item inserted into it and check if that item has been inserted since! (Try writing a purely algebraic specification.)
Suppose we take a more practical approach where we *specify* the LRU unit through one simple (and concrete) implementation. Suppose we also specify a more involved LRU as another concrete implementation (say, realized using a matrix of bits). How do we ensure that the implementation of the LRU is faithful to its specification? One verification criterion commonly used is based on the notion of algebraic homomorphism, expressed with the help of a commuting diagram:

![Commuting Diagram](image)

**Fig. 4.** ADT verification criterion. States $s$ and $S$ “match” or “correspond” if all observers yield the same value when applied to $s$ and $S$. Here, $s$ and $S$ match as do $s'$ and $S'$. The proof technique illustrated in this diagram is based on induction on constructor application.

Captured as a mathematical equation, Figure 4 specifies the following verification condition:

$$o(s, \text{args}) = O(S, \text{args}) \implies o(c(s, \text{args}'), \text{args}) = O(C(S, \text{args}'), \text{args})$$

As a concrete example, suppose we specify the LRU unit using a queue data structure and implement it using a matrix data structure, as shown in Figure 5:

To keep the examples simple, we will return to a familiar (albeit hackneyed) one; a FIFO queue. We will discuss a non pipelined queue in Section 5.8 and a pipelined queue in Section 5.9.
Implementation of "use(s,i)":
move "i" to the tail.

Implementation of "use(S,i)":
reset ith row ; set ith column

Implementation of "least(s)":
front of the queue.

Implementation of "least(S)":
look for row containing all "1"s

---

Observer
"least" returns 1

use(s,3)

Observer
"least" returns 1

use(s,1)

Observer
"least" returns 2

---

This implementation "works" by leaving exactly one bit set
in the i-th row, adding one more
bit to all rows that originally
contained fewer than i bits, and
leaving the # of bits in rows that
contained more than i bits the same.

---

Fig. 5. An LRU specification and its implementation
A Simple Non-pipelined FIFO Queue  Let’s take a simple circulating pointer queue, specify it in the usual way (using two pointers “chasing each other”) and implement it in a VLSI efficient way (using MLS counters for the pointers, as explained later). We will use a textual language for specifying behaviors, whose constructs will be introduced as needed. The basic descriptive style will be that of state transition systems, except we shall specify things in a symbolic fashion. The tacit convention is that we will be working with a universally quantified fragment of first-order logic.

Specification  We will write the FIFO queue specification using the tuple data type <M, F, R, f, e> where M is the memory, F is the front-pointer of the queue, R is the rear-pointer, f is the “full flag”, and e is the “empty flag”. Call this FIFO specification F1, standing for “FIFO version 1”. We will subject M to the operation M[v/a] which means “update M such that its address a contains value v”. The update specification ([v/a]) can be cascaded; thus, M[v/a][v1/a1] means “update M such that its address a has v, and update the resulting memory such that a1 has v1”. Obviously, if a = a1, then the value v “gets over-written” by the value v1. We will subject F and R to the + 1 operation, which is tacitly assumed to occur under the modulus of an unspecified (but fixed) capacity, with the condition that capacity is 2^N − 1 for some N.

We will describe three constructor operations on the queue: init(s) which initializes the queue, ins(s, v) which inserts v at the rear of the queue in state s, and rem(s) which removes the element at the head of the queue. We will describe three observers on the queue: front(s) which returns the front value of the queue, full(s) which returns true if s is in a full state, and empty(s) which returns true if s is in an empty state.

Specification of the operations of FIFO F1:

init(<M, F, R, f, e>) = <M, 0, 0, false, true>

empty(<M, F, R, f, e>) = e

full(<M, F, R, f, e>) = f

ins(<M, F, R, false, e>, v) = <M[v/R], F, R + 1, (F == R + 1), false>

rem(<M, F, R, f, false>) = <M, F + 1, R, false, (F + 1 == R)>

front(<M, F, R, f, e>) = M[F]
Implementation Let us implement this FIFO queue by implementing F and R using highly efficient hardware counters called maximum length sequence (MLS) counters. An N-bit MLS counter is a shift-register with bits numbered, say, 0 to N-1 from left to right, and with the shift register input set to the XOR of bits N-2 and N-1, It counts thus:

starting at 0000 (for N=4)
remain stuck in 0000 forever

starting in 0001, count as follows

0001, 1000, 0100, 0010, 1001, 1100, 0110, 1010, 1101, 1110, 1111, 0111, 0011, 0001
and back to 0001

Thus an N-bit MLS counter cannot be initialized in 0000. Initialized in any other states, it counts “higgledy piggledy” over $2^N - 1$ states.

The implementation of the FIFO is achieved by replacing F and R with MLS counters of the appropriate lengths, initializing them in a non-zero state, and for each F+1 or R+1, replace them by the “shift” command on the MLS counter. Call this implementation of FIFO “F2” standing for FIFO, version 2.

Correctness of Implementation We can now apply the previous verification criterion based on constructor induction. It will be clear that we need to strengthen the former verification condition to the following (taking s to represent the queue specification states that employ “ordinary” counters and S to represent the queue implementation states that employ MLS counters:

\[ o(s, args) = 0(S, args) \land \text{good}(s) \land \text{GOOD}(S) \]

\[ => \]

\[ o(c(s, args'), args) = 0(C(S, args'), args) \land \text{good}(c(s, args')) \land \text{GOOD}(C(S, args')) \]

where \( \text{GOOD}(S, \ldots) \) is true if \( F<>0 \) and \( R<>0 \) and the full and empty bits \( f, e \) are consistent with the counters \( F \) and \( R \), while \( \text{good}(s, \ldots) \) is true if \( f, e \) are consistent with the counters \( F \) and \( R \).

Notice that we still have the same “commuting diagram” as before, except that the transitions are defined only on a subset of the implementation states.

5.9 A Simple Pipelined FIFO Queue

Let us now define a version of F1 called F3 that is implemented in a pipelined fashion. More specifically, F3 will carry out the queue operations following a time schedule...
defined by a clock oscillator. Below, we will use the notation \( c(S_1, S_2, \ldots) \) to denote that a hardware module \( (F) \) is in control state \( c \) and data state \(<S_1, S_2, \ldots>\). We will use the notation

\[ c(S_1, S_2, \ldots) \rightarrow \text{op(args)} \rightarrow c'(S_1', S_2', \ldots) \]

to indicate that the hardware module in question performs operation \( \text{op} \) with arguments \( \text{args} \) in one clock cycle, reaching the “next state” \( c'(S_1', S_2', \ldots) \). A collection of such state transition rules will define the FIFO queue \( F3 \). The control state names used are \( q \) for \textit{quiescent} and \( p \) for \textit{pipelined}.

\[
\begin{align*}
q(M, F, R, f, e) & \quad \text{-- nop} \quad \rightarrow \quad q(M, F, R, f, e) \\
q(M, F, R, f, e) & \quad \text{where } e = \text{false} \\
& \quad \quad \quad \quad \rightarrow q(M, F+1, R, \text{false}, (F+1 == R)) \\
p(M, F, R, f, e) & \quad \text{-- nop} \quad \rightarrow \quad q(M, F, R+1, f, e) \\
p(M, F, R, f, e) & \quad \text{where } f = \text{false} \\
& \quad \quad \quad \quad \rightarrow q(M, F+1, R+1, \text{false}, (F+1 == R+1)) \\
p(M, F, R, f, e) & \quad \text{where } f = \text{false} \\
& \quad \quad \quad \quad \rightarrow p(M[v1/R+1], F, R+1, (F == R+2), \text{false})
\end{align*}
\]

Does \( F3 \) implement \( F1 \) faithfully? For this, we can make an attempt to apply the verification criterion shown in Figure 4. However this attempt fails because for a pipelined processor, there exists no point of clear correspondence. However we can apply the technique proposed by Burch and Dill illustrated in Figure 6. This figure also illustrates a slight modification of the original verification criterion for handling cases where the the specification state cannot be obtained by merely projecting the innards from the implementation state.

Notice from Figure 6 that this verification criterion also is based on induction on the length of instruction traces. In effect, it proves (basis case) that a single instruction followed by an infinity of NOPs is handled correctly by both systems, and assuming that an instruction stream of length \( N \) followed by an infinity of NOPs is handled correctly, then so is an instruction stream of length \( N+1 \) followed by an infinity of NOPs correctly handled.
The "original" Burch/Dill verification criterion.

The above verification criterion may also be generalized for handling Spec states that can't be derived from Imp states by mere projection (e.g. LRU). In this case, the Flushed New Imp state and New Spec state must correspond assuming that the Flushed Old Imp state and Old spec state do so.

Fig. 6. A verification criterion for pipelined processors
Let us apply this verification criterion and generate a few of the VCs. Suppose the specification machine for this verification is a non-pipelined FIFO with identical internal state representation. More specifically, the specification machine doesn’t have the pipelined state $p$ and its $\text{ins}$ operation is as follows:

$$q(M, F, R, f, e) \text{ where } f = \text{false}$$

$$-- \text{ins}(v) \rightarrow q(M[v/R], F, R+1, (F == R+1), \text{false})$$

As one example of generating a VC, consider the old implementation state $p(M, F, R, f, e)$. “Flushing” this implementation state by executing a $\text{nop}$ results in the old specification state $q(M, F, R+1, f, e)$.

Now, we assume that $f$ is false, and execute one instruction (e.g. an $\text{ins}(v1)$) from the old implementation state, resulting in

$$p(M[v1/R+1], F, R+1, (F == R+2), \text{false}).$$

Flushing this state results in

$$p(M[v1/R+1], F, R+2, (F == R+2), \text{false}).$$

If one were to execute $\text{ins}(v1)$ from the old specification state

$$q(M, F, R+1, f, e),$$

one would get

$$q(M[v1/R+1], F, R+2, (F == R+2), e)$$

which agrees with the flushed new implementation state.

In general, doing the above style of reasoning requires a theorem prover. Fortunately, Burch, Dill and most others have been able to use automatic decision procedures for fragments of first-order logic to accomplish much of their reasoning automatically.

### 5.10 Verifying pipelined processors

In this section, we will consider a toy pipelined processor. For the sake of simplicity, we consider exactly one instruction class, namely the $\text{alu}$ class. The instruction set level specification of the processor is
State: RF -- register file

State Transition (called a_step below):

RF
   -- (op s1 s2 d st) -->
   if st
   then RF
   else wr(RF, d, alu(op, rd(RF, s1), rd(RF, s2)))

The implementation uses a 4-stage pipeline (as opposed to previous versions of this example which say that there are 3 stages). A diagram showing this pipelined implementation is in Figure 7.

The pipelined machine state and the stage behaviors are as follows:

State: (RF, all the other registers which are mentioned below)

State Transitions per stage:

Instruction fetch stage:

   opcode := op
   dstn := d
   s1reg := s1
   s2reg := s2
   stall := st

Operand fetch stage:

dstnd := dstn
stalld := stall
opcoded := opcode
opreg1 := if (s1reg = dstnd) and not(stalld)
   then aluout
   else if (s1reg = dstnd) and not(stalld)
   then wb
   else rd(RF, s1reg)
endif
opreg2 := if (s2reg = dstnd) and not(stalld)
c_wb = if stall then RF
    else wr(RF, dstndd, wb)

c_ex = if stall then RF
    else wr(RF, dstnd, alu(op, opreg1, opreg2))

c_of = if stall then RF
    else wr(RF, dstn, alu(op, rd(RF, s1), rd(RF, s2)))

c_if = if stall then RF
    else wr(RF, d, alu(op, rd(RF, s1), rd(RF, s2)))

istep = one step of execution of the diagram on the left
        with a fresh instruction input

stall = one step of execution of the diagram on the left
        with a stall-asserted instruction input

a_step = one step of the "abstract" machine which is exactly c_if

Fig. 7. A simple 4-stage pipeline
then aluout
else if (s2reg = dstndd) and not(stalldd)
  then wb
else rd(RF, s2reg)
endif

Alu op stage:

  dstndd := dstnd
  stalldd := stalld
  wb := alu(opcoked, opreg1, opreg2)

Writeback stage:

  RF := if stalldd then RF
  else wr(RF, dstndd, wb)
  endif

The verification condition using the “standard” (e.g. Burch/Dill) method would look like in Figure 8:

The verification conditions generated are:

imp_state is an arbitrary implementation state meeting the system
invariant for ‘‘legal’’ states (in our example, the invariant is ‘‘true’’):

VC : a_step(stall(stall(stall(imp_state))))) =
stall(stall(stall(istep(imp_state))))

This usually generates in gigantic “if-then-elses” besides inviting other problems
when iterative loops, interlocks, etc. are present in pipeline stages.

The verification condition using the Hosabettu/Srivas method would look like in
Figure 9:

The verification conditions generated are:

VC1: RF(c_wb(imp_state)) = RF(istep(imp_state))

VC2: RF(c_ex(c_wb(imp_state))) = RF(c_wb(istep(imp_state)))
\textbf{Fig. 8.} The usual verification criterion for pipelined processors

\textbf{Fig. 9.} A better verification criterion for pipelined processors
VC3: RF(c_of(c_ex(c_wb(imp_state)))) = RF(c_ex(c_wb(istep(imp_state))))

VC4: RF(A_step(c_of(c_ex(c_wb(imp_state))))) = RF(c_of(c_ex(c_wb(istep(imp_state)))))

The proof has been found to decompose in a modular fashion, besides circumventing problems due to interlocks, loops, etc. Essentially, for embedded stage loops, the user is forced to state an I/O behavior while writing the “completion” function for that stage.

6 Module 4: Temporal Reasoning

6.1 Reduced Ordered Binary Decision Diagrams (ROBDDs)

Consider the decision-tree representation of a Boolean function in Figure 10(a), taken from Bryant’s ACM Surveys paper. This function’s truth-table is spelt out by the paths from its root to its leaves, with dashed lines representing negation. Let the total ordering \( x_1 < x_2 < x_3 \) be maintained for its variables (this ordering is read in the direction going top to bottom in the figure). After

- removing duplicate terminals,
- removing duplicate non-terminals, and
- removing redundant tests

we obtain a canonical representation for the Boolean function. This makes many Boolean procedures very simple (and elegant):

- Functional equivalence \( \rightarrow \) graph isomorphism (with hashing techniques, it reduces to a much simpler test, as we shall see)
- Satisfiability \( \rightarrow \) existence of a 1 leaf.
- Tautology \( \rightarrow \) the only leaf is a 1.

OBDDs are constructed using procedure APPLY which is much cheaper than the decision-tree method which was used only for illustration (decision-trees can be exponential in size).

6.2 Effect of Variable Orderings

Try constructing ROBDDs (or OBDDs) for the following functions:

\[ a_1 \, b_1 + a_2 \, b_2 + a_3 \, b_3 \] under two orderings:
Fig. 10. An Example Decision Tree
a1, a2, a3, b1, b2, b3
a1, b1, a2, b2, a3, b3
Two-bit vector less-than relation: \( a_1, a_0 < b_1, b_0 \)

Go over
- The Shannon expansion for a Boolean function (originally recognized by Boole himself)
  \[ f = \overline{x} \cdot (f\text{-with-x-set-to-zero} + x \cdot f\text{-with-x-set-to-one}) \]
- Function composition
  \[ f\text{-with-g-for-x} = \overline{g} \cdot (f\text{-with-x-set-to-zero} + g \cdot f\text{-with-x-set-to-one}) \]
- "There exists an x such that f" can be written as
  \[ f\text{-with-x-set-to-zero} + f\text{-with-x-set-to-one} \]
- "For all x f" can be written as
  \[ f\text{-with-x-set-to-zero} \cdot f\text{-with-x-set-to-one} \]

### 6.3 The APPLY Algorithm to Construct ROBDDs

The APPLY algorithm is key to the efficiency of ROBDD manipulation procedures. Its outline is as follows,

**Input:** Two ROBDDs and an operation to be applied

**Output:** One ROBDD which results from applying the operation to the BDDs

**Data structures:**

- Hash-table "recursion-cutoff" to prevent redundant recursive calls
- Hash-table "bdds-so-far" to help maximize sharings

**Initialize bdds-so-far** to contain a zero node at position 0 and a 1 node at position 1

I now provide snapshots of execution of this algorithm wrt the example on page 302 of Bryant’s paper

**Step 0 (initial condition)**
Call A4,B3 comes. Not present in recursion-cutoff. See what value it yields. Here it yields 0, which is at index 0. So update recursion-cutoff to have A4,B3 : 0 thus:

<table>
<thead>
<tr>
<th>Args</th>
<th>Value</th>
<th>Index</th>
<th>v (variable)</th>
<th>hi(v)</th>
<th>low(v)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A4,B3 : 0</td>
<td></td>
<td>0</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Call A5,B4 comes. Not present in recursion-cutoff. Yields value 1 which is at index 1. So update recursion-cutoff thus:

<table>
<thead>
<tr>
<th>Args</th>
<th>Value</th>
<th>Index</th>
<th>v (variable)</th>
<th>hi(v)</th>
<th>low(v)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A4,B3 : 0</td>
<td></td>
<td>0</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>A5,B4 : 1</td>
<td></td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Call A3,B2 comes. Has variable d, and expects hi(d)=1 (whatever the call A5,B4 returns) and low(d) = 0 (whatever A4,B3 returns). Triple (d,1,0) not present in bdds-so-far, so add it there, and return index
where added as result to be entered in recursion-cutoff, thus:

\[ \begin{array}{|c|c|c|c|c|}
\hline
\text{Args} & \text{Value} & \text{Index} & \text{v (variable)} & \text{hi(v)} & \text{low(v)} \\
\hline
A4,B3 : & 0 & 0 : 0 & - & - \\
A5,B4 : & 1 & 1 : 1 & - & - \\
A3,B2 : & 2 & 2 : d & 1 & 0 \\
\hline
\end{array} \]

Call A5,B2 comes. handled as explained before:

\[ \begin{array}{|c|c|c|c|c|}
\hline
\text{Args} & \text{Value} & \text{Index} & \text{v (variable)} & \text{hi(v)} & \text{low(v)} \\
\hline
A4,B3 : & 0 & 0 : 0 & - & - \\
A5,B4 : & 1 & 1 : 1 & - & - \\
A3,B2 : & 2 & 2 : d & 1 & 0 \\
A5,B2 : & 1 & & & & \\
\hline
\end{array} \]

Call A6,B2 comes. Has var c, hi(c)=1, low(c)=2. Triple (c,1,2) not in bdds-so-far, so enter it, and update recursion-cutoff:

\[ \begin{array}{|c|c|c|c|c|}
\hline
\text{Args} & \text{Value} & \text{Index} & \text{v (variable)} & \text{hi(v)} & \text{low(v)} \\
\hline
A4,B3 : & 0 & 0 : 0 & - & - \\
A5,B4 : & 1 & 1 : 1 & - & - \\
A3,B2 : & 2 & 2 : d & 1 & 0 \\
A5,B2 : & 1 & & & & \\
A6,B2 : & 3 & 3 : c & 1 & 2 \\
\hline
\end{array} \]

Call A2,B2 comes with var b, hi(b) = 3, low(b) = 2. This triple
is not present, so add to bdds-so-far, and update recursion-cutoff also:

<table>
<thead>
<tr>
<th>Args</th>
<th>Value</th>
<th>Index</th>
<th>v (variable)</th>
<th>hi(v)</th>
<th>low(v)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A4,B3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>A5,B4</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>A3,B2</td>
<td>2</td>
<td>2</td>
<td>d</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>A5,B2</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A6,B2</td>
<td>3</td>
<td>3</td>
<td>c</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>A2,B2</td>
<td>4</td>
<td>4</td>
<td>b</td>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>

Call A3,B4 comes. Handled as before:

<table>
<thead>
<tr>
<th>Args</th>
<th>Value</th>
<th>Index</th>
<th>v (variable)</th>
<th>hi(v)</th>
<th>low(v)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A4,B3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>A5,B4</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>A3,B2</td>
<td>2</td>
<td>2</td>
<td>d</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>A5,B2</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A6,B2</td>
<td>3</td>
<td>3</td>
<td>c</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>A2,B2</td>
<td>4</td>
<td>4</td>
<td>b</td>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>

Call A6,B5 comes with var c, hi(c)=1 and low(c) = 1. So don’t enter triple (c,1,1); instead, return as result ‘1’ itself:

<table>
<thead>
<tr>
<th>Args</th>
<th>Value</th>
<th>Index</th>
<th>v (variable)</th>
<th>hi(v)</th>
<th>low(v)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A4,B3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>A5,B4</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
A3, B2 : 2
A5, B2 : 1
A6, B2 : 3
A2, B2 : 4
A3, B4 : 1
A6, B5 : 1

Finally, call A1, B1 comes with var a, hi(a)=1, low(a)=4. Enter this triple, and update both HTs. We are done. Bdds-so-far has the resulting BDD, pointed to by the value field of A1, B1 in recursion-cutoff:

<table>
<thead>
<tr>
<th>Args</th>
<th>Value</th>
<th>Index</th>
<th>v (variable)</th>
<th>hi(v)</th>
<th>low(v)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A4, B3 : 0</td>
<td>0</td>
<td>: 0</td>
<td></td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>A5, B4 : 1</td>
<td>1</td>
<td>: 1</td>
<td></td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>A3, B2 : 2</td>
<td>2</td>
<td>: d</td>
<td></td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>A5, B2 : 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A6, B2 : 3</td>
<td>3</td>
<td>: c</td>
<td></td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>A2, B2 : 4</td>
<td>4</td>
<td>: b</td>
<td></td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>A3, B4 : 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A6, B5 : 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A1, B1 : 5</td>
<td>5</td>
<td>: a</td>
<td></td>
<td>1</td>
<td>4</td>
</tr>
</tbody>
</table>

Look into the paper for complexity arguments and how to model other operations in terms of APPLY, the RESTRICT operation’s implementation, etc.

6.4 Computational Tree Logic, and Model-checking

In these notes, I shall try to explain some of the difficult portions of McMillan’s thesis and of the paper by Clarke, Emerson, and Sistla.

Introduction Model checking is the process of checking whether a temporal formula is true of a “model” (a finite-state machine (FSM), in our case). There are basically two approaches: explicit enumeration through graph-traversal, and implicit enumeration
through the use of Reduced Ordered Binary Decision Diagrams (or ROBDDs). The latter is also known as symbolic model checking.

Since historically an explicit enumeration algorithm was proposed for CTL, we shall look at one such first; then we shall look at implicit enumeration.

**LTL versus CTL** We have already seen that in LTL, the “modalities” are □ standing for “henceforth” and ◇ standing for “eventually”. Formulae can then be constructed to capture truths about system executions, Figure 11 shows an example illustrating LTL and CTL. Consider the LTL assertion □(p ⇒ ◇q) for “henceforth” and ◇ standing for “eventually”. This assertion is false for the executions shown in Figure 11 (why?)².

![Diagram](image)

**Fig. 11. Illustrating LTL and CTL**

In CTL, we have the modalities

- □ standing for all paths, and
- ◇ standing for there exists a path.

However, we cannot use the above modalities in isolation; we **must** combine it with the following two modalities:

- G standing for everywhere along a path, and
- E standing for some path.

² The reason being that there is one execution where p happens without q following it. Viewed another way, the language of system executions has an “offending” string that is not contained in the language of the property.
$F$ standing for somewhere along a path.

Thus, \( AG(p \Rightarrow EF \ q) \) is roughly equivalent to “whenever \( p \) happens, it will be followed by a \( q \) in some execution”. Obviously, this property gets violated in the example executions of Figure 11.

How about considering \( AF \) as a translation for \( \Diamond \)? Are they equivalent? To “test” this, consider the following formulae:

\[
(\Diamond p) \Rightarrow (\Diamond q) \\
(AF \ p) \Rightarrow (AF \ q)
\]

We see that the first formula is false of the given system (that has the three executions shown in Figure 11) while the second (CTL) assertion is true of the system (true in a vacuous sense, because the \( AF \ p \) part is false).

Many different temporal logics have already been proposed, They all have different expressive powers as well as different “containment relationships”. Of these, CTL and LTL enjoy the largest following in practice.

**Explicit Enumeration-based Model Checking** We shall explain this procedure with the help of an example. Consider the formula written as an s-expression:

\[
(AG \ (OR \ (NOT \ T1) \ (AF \ C1)))
\]

Let us attempt to check whether this formula is true of state 0 of the FSM shown in Figure 12. This FSM represents the execution of two parallel processes with alternating priorities, taking \( \mathbb{N} \) to stand for “in non-critical region”, \( \mathbb{T} \) for “trying to enter”, \( \mathbb{C} \) for “in critical region”, and the suffixes for the process IDs.

To check the above assertion, number the formula and its sub-formulae as shown below:

<table>
<thead>
<tr>
<th>Formula number</th>
<th>Formula</th>
<th>Subformulas</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( (AG \ (OR \ (NOT \ T1) \ (AF \ C1))) )</td>
<td>(2)</td>
</tr>
<tr>
<td>2</td>
<td>( (OR \ (NOT \ T1) \ (AF \ C1)) )</td>
<td>(3 5)</td>
</tr>
<tr>
<td>3</td>
<td>( (NOT \ T1) )</td>
<td>(4)</td>
</tr>
<tr>
<td>4</td>
<td>T1</td>
<td>nil</td>
</tr>
<tr>
<td>5</td>
<td>( (AF \ C1) )</td>
<td>(6)</td>
</tr>
<tr>
<td>6</td>
<td>C1</td>
<td>nil</td>
</tr>
</tbody>
</table>

Then, execute the following for loop that considers the sub-formulae of the given formula bottom-up, and labels each state of the FSM with those sub-formulae that are true in that state.
for i := length(f) step -1 until 1
  do label_graph(f)

Complexity: \( O(\text{length}(f) \times (\text{card}(S) + \text{card}(R))) \) where the multiplier is the time for one label_graph invocation.

Thus, in our example, we would invoke label_graph on formulae C1, (AF C1), T1, (NOT T1), (OR ...), and (AG ...), in that order. Then generate the following “tableau”. Because of the order in which the sub-formulae are considered, the tableau can be filled using a recursive depth-first traversal.

<table>
<thead>
<tr>
<th>State number</th>
<th>Atomic Propositions</th>
<th>C1</th>
<th>(AF C1)</th>
<th>T1</th>
<th>(NOT T1)</th>
<th>(OR (NOT T1))</th>
<th>(AF C1)</th>
<th>(AG (OR (NOT T1))</th>
<th>(AF C1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>N1,N2</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>T1,N2</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>N1,T2</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>C1,N2</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>T1,T2</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>T1,T2</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>N1,C2</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>C1,T2</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>T1,C2</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The call to label_graph(AF C1) will, for example, work as follows:

```
procedure label_graph(f)
begin
  ...
  % main operator is AF
  begin
```
ST := empty_stack;
for all s in S do marked(s) := false;
L : for all s in S do
   if not marked(s) then AF(f, s, b)
end

procedure AF(f, s, b)
begin
   if marked(s) then
      begin
         if labeled(s, f) then
            begin b := true; return end;
            b := false; return end;
   marked(s) := true;
   if labeled(s, f) then
      begin add_label(s, f); b := true; return end;
   push(s, ST);
   for all s1 in successors(s) do
      begin
         AF(f, s1, b1);
         if not(b1) then
            begin pop(ST); b := false; return end;
         pop(ST); b := true; add_labels(s, f); return
end

6.5 Symbolic Model-checking

We shall illustrate symbolic model checking with respect to a much smaller FSM and
 temporal formula, as it becomes quite tedious for larger ones. I shall present the steps
 of the algorithm in the following subsections.

Represent the FSM through a Boolean Relation Any FSM can be encoded
 through a Boolean formula specifying the transition relation captured by the FSM.
 For instance, an FSM with states 0 and 1, and transitions (0,1), (1,1), and (1,0) with
 variable $b$ being false at 0 and true at 1 can be represented by taking $b'$ to denote the
next state of $b$ and forming the relation

$$R = -bb' + bb' + b\neg b'$$

which says the machine has a move where $b$ is false now and true in the next state, or true in the present and next states, or true in the present state and false in the next state. The above simplifies to

$$R = b \lor b'.$$

To illustrate symbolic model checking, let us try to find out the set of states in which $EX\neg b$ is true. Suppose $s$ were such a state, then, there exists a state $s'$ such that $(s, s')$ is a transition and $s' \models \neg b$. But how do we obtain all states such as $s$ above in one go? To accomplish this, we note that the state transition relation $R$ captures all possible relationships between the value of the $b$ variable in the “present” state and the “next” state. $R$ uses $b'$ to denote the value of $b$ in the “next” state for a given present state. Given all this,

$$\exists b'. R \land \neg b'$$

is a way of asking “give me all the $b$-variable values such that $b$ is false in the next state.” The above expression is also known as the relational inverse of $R$ with respect to $b'$. The above simplifies to

$$\exists b'. b \land \neg b'$$

which in turn simplifies to $b$. So, expression $b$ characterizes all states in which $EXb$ is true. In our example, this includes only state 1.

Let us now ask “in which states is $EF\neg b$ true”? We use the fixed-point approximation for $EF\neg b$. We want the least fixed-point of the function

$$\lambda y. \neg b \lor EXy.$$ 

As usual, we begin iterating with $false$ and generate the series:

false

$$\neg b \lor EXfalse, \text{ i.e.}, \neg b$$

$$\neg b \lor EX\neg b$$

Aha! We know how to do $EX\neg b$! We just did it thru relational inverse, which gave $b$ as the answer;

So, plug in that answer for $EX\neg b$ and continue with the fixed-point iteration,
\(-b \lor b\), \text{i.e., true.}

Having converged to a constant \text{true}, there is no point continuing with the fixed-point iteration.

The answer \text{true} says that \(EF \neg b\) is true in all the states (true in state 0 and 1 in our case)! The SMV does this fixed-point iteration using ROBDDs.

6.6 More on CTL Model Checking

Computational Tree Logic (CTL) is presented in McMillan’s book (based on his dissertation). These notes attempt to reiterate some of the points he has made, I will attempt to address only the difficulty points.

**Fixed-point theory of CTL** Consider the recursive definition of the \(A\cup\) operator:

\[
(A[p \cup q]) = q \lor (p \land AX(A[p \cup q]))
\]

Does this equation define anything at all? How many solutions are there for this equation? We can easily see that \(A[p \cup q]\) is a fixed-point of

\[
\lambda z.(q \lor (p \land (AXz)))
\]

As is customary in fixed-point theory, we can try seeking the fixed point by applying the above function repeatedly on “bottom”. \text{False} as well as \text{true} both seem to be good choices for “bottom”. (The rationale for these choices will be provided later.) The former series develops as

\begin{align*}
\text{false} \\
q \\
q \lor (p \land \text{AX}q) \\
q \lor (p \land \text{AX}(q \lor (p \land \text{AX}q)))) \\
\ldots
\end{align*}

\(q\) must be inevitable for this fixed-point to yield \text{true}.

The second series develops as

\begin{align*}
\text{true} \\
q \lor p \\
q \lor (p \land \text{AX}(q \lor p)) \\
q \lor (p \land \text{AX}(q \lor (p \land \text{AX}q \lor p)))) \\
\ldots
\end{align*}

\(q\) is evitable; an execution with \(p\) true everywhere can satisfy the above fixed-point.
Fig. 12. An Example to Illustrate the CES Procedure
Let us now study the fixed-point theory of CTL in some more detail. Recall that a Kripke structure is a triple \((S, R, L)\) where \(S\) is a set of states, \(R\) is the accessibility relation, and \(L\) is the valuation function for atomic propositions. \(S\) is usually finite—at least in all our examples, it is so. \(R\) defines the \textquotedblleft edges\textquotedblright\ in the execution graph, \(L\) maps each atomic proposition \(pp\) (a positive literal) to the set of states at which \(p\) is true. Since atomic propositions are also formulae, each formula (denoted by \(f, g, h, \ldots\)) of CTL also denotes a subset of \(S\)—those states at which that formula is true. In particular:

- formula \textit{false} denotes the empty set
- \textit{true} denotes \(S\)

Let \(p\) denote \(\mathcal{P}\) and \(q\) denote \(\mathcal{Q}\). Then \(p \land q\) denotes \(\mathcal{P} \cap \mathcal{Q}\)
\[p \Rightarrow q\text{ iff }\mathcal{P} \subseteq \mathcal{Q}\]

From now on, we will treat formulae and the sets they denote virtually interchangeably.

As we have seen, we can convert recursive definitions into fixed-point equations of the form \(x = Fx\). Function \(F\) will be one that maps formulae to formulae. We will sometimes use \(\tau\) to denote these functions.

**Monotonicity:** We can define function \(\tau\) to be monotonic if \(q \Rightarrow p\) implies \(\tau(q) \Rightarrow \tau(p)\), or, treating formulae as sets (without using the calligraphy font, which was a visual \textquotedblleft crutch\textquotedblright\ thus far) \(q \subseteq p\) implies \(\tau(q) \subseteq \tau(p)\). In other words, if formulae are connected by set operators, read the formula as the set it denotes; otherwise if formulae are connected by Boolean operators, read formulae as formulae themselves.

**Examples:** We can show that \(\lambda y.(x \land \neg y)\) is not monotonic, while \(\lambda x.(x \land \neg y)\) is monotonic. **Proof:** Assume \(p \subseteq q\). To show \(x \cap \overline{p} \subseteq x \cap \overline{q}\). We can see this is false if \(x \neq \emptyset\) by, say, taking \(p = \text{false}\) and \(q = \text{true}\).

It is customary to denote the least fixed-point of a function \(\tau\) by \(\mu x.\tau(x)\) and the greatest fixed-point by \(\nu x.\tau(x)\). A monotonic functional \(\tau\) has a least fixed-point given by the intersection of all its fixed-points, or in math notation
\[\mu x.\tau(x) = \bigcap_i f p_i\]
and has a greatest fixed-point given by
\[\nu x.\tau(x) = \bigcup_i f p_i\]

To prove the above, it suffices to show that the union of two fixed-points is a fixed-point and the intersection of two fixed-points is a fixed-point. Clearly, fixed-points in this domain are sets/formulae.

**Proof:** Let \(a\) and \(b\) be fixed-points of \(\tau\) (so \(\tau(a) = a\) and \(\tau(b) = b\)). Then, to show \(\tau(a \land b) = a \land b\). This can be shown through structural induction over the structure
of \( \tau \). If we prove the above for \( \tau = \lambda z.x \land z \), \( \tau = \lambda z.z \land x \), and \( \tau = \lambda z.\neg z \), we would have covered the set of all propositional formulae. Since, in our domain of discourse, formulae can include CTL operators also, we will have to consider those cases also. I’ll just show one example CTL formula.

Consider \( \tau = \lambda z.x \land z \).

Then \( \tau(a \land b) = x \land a \land b \).
\( \tau(a) = x \land a = a \)
\( \tau(b) = x \land b = b \)
\( \tau(a \land b) = x \land a \land b = a \land b \), from the above.

Now consider \( tau = \lambda z.\neg z \). The premises \( tau(a) = a \) and \( tau(b) = b \) turn into \( \neg a = a \) and \( \neg b = b \). From these premises, anything you want can be proved, including \( \tau(a \land b) = \neg(a \land b) \).

As another example, let us consider \( \tau = \lambda y.p \lor EXy \), which is the \( \tau \) extracted from the recursive definition of \( EFp \). Let us show that this \( \tau \) is monotonic.

if \( r \subseteq q \), i.e. \( r \Rightarrow q \)
to show \( (p \lor EXr) \Rightarrow (p \lor EXq) \)
or \( (p \lor EXr) \subseteq (p \lor EXq) \).

Suppose \( s \models (p \lor EXr) \)
then \( s \models p \) OR \( s \models EXr \)
The latter means \( \exists \) path \( s, s_1, s_2, \ldots \) such that \( s_1 \models r \)
But, since \( r \Rightarrow q \), \( s_1 \models q \)
So, \( s \models EXq \)
So, finally, \( s \models p \) OR \( s \models EXq \).

Thus, we have hope that the \( \tau \)s that we “pull out” of recursive definition bodies are monotonic. But of course, we need to prove that every such monotonic function that we end up using for each recursive definition we write is monotonic.

**Homework:** Prove that the \( \tau \) that you pull out of the recursive definition of \( E\!G\!x \) is monotonic.

**Continuity:** A function \( t \) is continuous (in our context) if given a chain \( p_i \) of sets such that for every \( i \) \( p_i \subseteq p_{i+1} \),

\[ \tau(\cup_i p_i) = \cup_i (\tau(p_i)) \]

In the discussions above, \( i \) goes from 0 to infinity. Note: McMillan’s book/thesis defines \( \cap \)-continuity and \( \cup \)-continuity. The above definition of continuity pertains to \( \cup \)-continuity, just because the operator in question is \( \cup \). A definition for \( \cap \)-continuity is similar, and hence omitted.
The fact (proved above) that the intersection of all fixed-points is the least fixed point is not a constructive method of defining the least fixed point (and likewise for the greatest fixed-point). We will show that if $\tau$ is monotonic and continuous,

$$\mu x.\tau(x) = \bigcup_i (\tau^i(false))$$

i.e., show

$$\tau(\bigcup_i (\tau^i(false))) = \bigcup_i (\tau^i(false))$$

and further, if $\tau(g) = g$, then

$$\tau(\bigcup_i (\tau^i(false))) \subseteq g.$$  

**Proof of** $\tau(\bigcup_i (\tau^i(false))) = \bigcup_i (\tau^i(false))$

LHS = $\tau(\bigcup_{i=0}^\infty (\tau^i(false)))$

Because of the continuity of $\tau$, the above equals

$\tau(\bigcup_{i=1}^\infty (\tau^i(false)))$

which is the same as LHS, because, for $i = 0$, we include only the empty set into the iterated union operation.

**Proof of** If $\tau(g) = g$, then $\tau(\bigcup_i (\tau^i(false))) \subseteq g$.

We have $false \subseteq g$

Because $\tau$ is monotonic, $\tau(false) \subseteq \tau(g) = g$

So $\bigcup_i (\tau^i(false)) \subseteq g$

Since the LHS as well as the RHS (above) are fixed-points of $\tau$, $\tau(\bigcup_i (\tau^i(false))) \subseteq g$

**Example illustrating fixed-point iteration:**

Given

$S = \{s_0, s_1, s_2, s_3\}$,

$R = \{(s_0, s_1), (s_1, s_2), (s_2, s_0), (s_2, s_3)\}$, and

$L(p) = \{s_1\}$.

Then, let us compute $EFp$ by iterating from $false$, trying to converge towards the least fixed-point.

$false$ denotes the empty set of states $\emptyset$. So, initially, we are saying $EFp$ is true nowhere. Not true, but we are not done yet!

Then we say $EFp$ is $p \lor (EXfalse)$ or $p$. i.e., we are saying that $EFp$ is true where $p$ is. Still not quite true but we are getting there...
Next time, we say $EFp = (p \lor EXp)$ which is true at $\{s_0, s_1\}$
Next time, we say $EFp = (p \lor EX(p \lor (EXp)))$ which is true at $\{s_0, s_1, s_2\}$.
Continuing further doesn’t add any more new states. Also, we have realized that the right answer has been found!

**Homework:** Show that $\cap_i(\tau^i(true))$ is the greatest fixed-point of $\tau$ which is as above (monotonic and continuous).

**Homework:** If $S$ is finite, show that any monotonic $\tau$ is necessarily $\cap$-continuous and $\cup$-continuous. Hint: If $S$ is finite, every chain $p_i$ has a maximal element, say $p_m$. Now “plug and chug”.

**Proving that in certain cases we need the least fixed-point:** Let us once again consider $\tau = \lambda y.p \lor EXy$, which is the $\tau$ extracted from the recursive definition of $EFp$. Let us, for the sake of illustration, try taking its greatest fixed-point by iterating from true. We will quickly see that the greatest fixed-point is true, which says any sequence of states at all satisfies $EFp$, which is quite untrue—we know in our minds that we want something else.

So we will now obtain the least fixed-point of the above $\tau$ and see if it corresponds to the understanding that we have about $EFp$ (also check it against the definition of $EFp$).

**Claim:** “the $EFp$ that we want” is the least fixed-point of the above $\tau$.

**Proof:** Suppose $y$ is a fixed-point of $\tau$. Then

\[
\tau(y) = y, \text{ i.e., } \\
p \lor (EXy) = y. \text{ Then, } \\
p \Rightarrow y, \text{ and } \\
EXy \Rightarrow y, \text{ or } EXy \subseteq y.
\]

In other words, all the states from which there is a one-step move to a state in $y$ are states in $y$.

Consider a state $s$ from which there is a two-step move to a state in $y$. Then we can go from $s$ to $s'$ where there is a one-step move from $s'$ to a state in $y$. Then $s'$ is in $y$. Then there is a one-step move from $s$ to a state in $y$.

By induction, we can show that if all states from which there is an $m$-step move to a state in $y$ are in $y$, then all states from which there is an $m + 1$-step move to a state in $y$ are in $y$.

So, whenever $EXy \Rightarrow y$, any state from which there is an $n$-step move to a state in $y$ is also a state in $y$, for an arbitrary $n$.

This means $EXy \Rightarrow y$ implies $EFy \Rightarrow y$. 


SO, $EXy \Rightarrow y$ implies $EFy \Rightarrow y$.

Since $p \Rightarrow y$, we have $p \subseteq y$, and so states in $p$ are states in $y$.

So, states from which there are $n$-step moves to states in $p$ are states from which there are $n$-step moves to states in $y$.

States from which there are $n$-step moves to states in $p$ are given by $EFp$.

So, at states in $EFp$, $EFy$ also holds.

So, $EFp \subseteq EFy \subseteq y$.

OR, $EFp$ is smaller than the arbitrarily chosen fixed-point $y$.

OR, $EFp$ is the least fixed-point!

**Homework:** Show that $EGp$ is the greatest fixed-point of $\lambda x.p \land EXx$.

Similarly, for $E[q \cup p]$, the “$\tau$ function” is $\lambda x.(p \lor (q \land (EXx)))$. Let $y$ be a fixed point. Then let us see what all we can glean:

$$(p \lor (q \land (EXx))) = y$$

So, $p \Rightarrow y$

and $q \land EXy \Rightarrow y$.

So, a state $s$ where $q$ is true now and there is a next state $s'$ where $y$ is true are in $y$!

Got the “pump handle”!!

States where $q$ is true now, and there is a next state $s'$ where $q$ is true and there is a next state $s''$ where $y$ is true are also in $y$.

Now, another question. What does $E[q \cup p]$ mean?

$E[q \cup p]$ is true of state $s_0$ iff $\exists j \geq 0$ such that $s_j \models p$ and for all $0 \leq i < j$, $s_i \models q$.

We have

$p \Rightarrow y$

$(q \land EXp) \Rightarrow y$, because states in $p$ are also states in $y$.

$(q \land (EX(q \land (EXp)))) \Rightarrow y$.

But the above is the series generated by the least-fixed point iteration.

So, the least fixed-point gives the desired meaning and (obviously) is contained in any other fixed-point $y$.

**Homework:** Argue that a formula that is true of exactly one execution, namely one where $q \land \neg p$, is a fixed-point, and lies above the least fixed-point. What relation does this formula have to the greatest fixed-point of $\tau$? For this question, you may give intuitive answers, but try to be as rigorous as possible.