Program Verification & Testing; Review of Propositional Logic

CS 536: Science of Programming, Fall 2022

Mon 2023-01-10: p.5

A. Why

- Course guidelines are important.
- Understanding what Science of Programming is is important.
- Reviewing/overviewing logic is necessary because we'll be using it in the course.

B. Outcomes

At the end of this class, you should

- Know how the course will be structured and graded.
- Have practiced some of the techniques we'll be using in class.
- Know what Science of Programming is about and how it differs from and is related to program testing.
- Understand what a propositional formula is, how to write them, how to tell whether one is a tautology or contradiction using truth tables, and see a basic set of logical rules for transforming propositions.

C. Introduction and Welcome

- The course webpages for are at http://cs.iit.edu/~cs536/ . You're responsible for the information there, even if you don't read it.
- We'll use myIIT → Blackboard for submitting homework and viewing grades.
- The lectures are automatically recorded and posted to Blackboard under Panopto.

D. Course Prerequisite: Basic Logic

- The course prerequisite formally is CS 401. In reality, what you need is some background in boolean logic (propositions and predicates) and some comfort in syntactic operations and formal languages.
- For a very rough assessment of your preparedness for this course, study the following questions: If you get all five correct, you have more than enough background for this course;. If you get five correct, you're probably okay but will need to brush up. If you get none correct, consider quickly dropping this course.

Prerequisite Quiz

1. Are $2+2$ and 4 syntactically equal and why?
2. If AND has higher precedence than OR and OR is left associative, how do you parenthesize V AND W OR X OR Y?
3. If p and q are propositions then what are the contrapositive, converse, and inverse of the implication $p \to q$ and how are they related?
4. How do you pronounce $(\neg \forall x \in \mathbb{Z}. \exists y \in \mathbb{Z}. y^2 < x)$ in English, and is it true?
5. What's the difference between saying that a predicate p is valid versus saying that you have a formal proof of p?

For answers, see the footnote[1]

- We'll quickly review basic logic in class. If you want other references to study, try the references linked to the home page.

E. So What Is Science of Programming Anyway?

- Science of Programming is about ***program verification***.
- Program verification aims to get reliable programs by discerning properties about programs.
  - It's harder to do this by writing programs and then proving them correct.
  - In practice, it's better to reason about programs as we write them.
- For this class, we'll look at a simple programming language. The syntax will be simple (that's not the important part).
  - What's important is formally (= mathematically, logically) specifying the semantics of programs and connecting them to the semantics of logical statements.
  - Put another way, if we want to be very sure about whether a program works or not, we have to be sure what we want the program to do (hence logical statements), and we have to be sure how programs execute (hence formal semantics), and we have to be able to connect the two (which will lead to studying formal rules for logical reasoning about programs).

F. Neither Reasoning or Testing is Completely Sufficient

- Let's contrast program verification and program testing.
  - In testing, we run a program and verify that it behaves correctly.
  - In verification, we reason about a program to predict that it will behave correctly.
- We need both testing of programs and reasoning about programs; neither is always better than the other.

---

[1] Answers: (1) No, because operator expressions aren't constants. (2) ((V AND W) OR X) OR Y. (3) Contrapositive: $\neg q \to \neg p$; Converse: $q \to p$; Inverse: $\neg p \to \neg q$. An implication and its contrapositive are semantically equivalent, as are the converse and inverse, but an implication and its converse are not. (4) "It's not the case that for every integer x, there exists an integer y such that y squared is less than x." It's true (try $x = 0$). (5) Validity is a semantic claim; having a formal proof of it is a syntactic claim.

  - When we reason about a program, we can make mistakes or overlook cases. We need testing as a reality check to show that our reasoning is sound.
  - In the other direction, complete testing of a program might involve a too-large set of test cases (infinite or close enough to infinite) to be practical. So we reason about our programs to identify a practical number of test cases that should represent all the possible test cases.
- As an example, say our specification is "If $z \geq c$ before the program, then $z > c$ after it", where the program is just "add x to z, but only if x is nonnegative."
  - In C, we can write `/* z >= c */ if (x >= 0) z = z+x; else ++z; /* z > c */`
  - To figure out which test cases are good, we reason about how the statements and properties interact.
  - E.g., take $x \geq 0$ (and its negation $x < 0$) and break up ≥ into separate > and = cases ($x > 0$, $x = 0$), to get $x < 0$, $x = 0$, and $x > 0$ as the general set of cases. If we think $x = -1$ and $x = 1$ are good enough generalizations of $x < 0$ and $x > 0$, then we're done: Our test cases are $x = -1$, $x = 0$, $x = 1$.
  - If we decide we want to be more thorough and treat $x = -1$ and $x < -1$ as different cases (and $x > 1$ similarly), we can turn them into $x = -2$ and $x = 2$, and end up with five test cases, namely $x = -2$, $x = -1$, $x = 0$, $x = 1$, $x = 2$.
  - Of course, if we keep breaking edge cases off of the < and > tests, we could get $x = -3$ and $x = 3$, then $x = -4$ and $x = 4$, and so on to infinity. A big part of testing is figuring when to stop doing all this.

G. Type-Checking as a Kind of Program Verification

- ***Static*** (i.e., compile-time) ***type-checking*** is an example of program verification: We analyze a program textually to reason about how it uses types, to check for type-correctness.
- The reasoning is symbolic / textual because we aren't actually running the program, so a type-checker is a mechanical theorem prover for judgements of the form "this variable or expression has type …" and "This operation is type-correct."
  - E.g., if variables x and y are of type integer, then $x + 1$ and x/y are integers, so $x + 1 = x/y$ is type-correct, etc. (Note x/y might still cause a runtime error, but it wouldn't be a type error.)
- A ***strong type-checker*** produces proofs that provide complete evidence for type safety. A ***weak type-checker*** produces proofs that provide only partial evidence for type safety.
  - E.g., type-checkers for Haskell or Standard ML are very strong; they guarantee type safety. (Note: You might still get runtime errors, but not for type-incorrect operations.)
  - However, type-checkers for C are weak; they have to assume you know what you're doing when you cast pointers.

H. Reasoning About One State of Memory vs Many States of Memory

- In testing, we have a finite number of specific values we use for our variables. We can verify that our program works with those specific values.
- In program verification we aim to say that our programs work in all possible cases. Typically, we have an infinite number of cases[2]. (Actually, it's a finite number, since memory is finite, but who wants to deal with, e.g., 2^{32} separate individual tests for an integer variable x?)
- In program verification, we use ***predicates*** like $x > 0$ to stand for a possibly infinite number of values. (A predicate is a syntactic object that has a truth value once you plug in specific values for its variables.)
- Using predicates, we can talk about an infinite number of possible execution paths simultaneously. Instead of actually executing a program, we simulate its execution symbolically, using rules of logic to manipulate our predicates. "If $x > 0$, then after adding 1 to x, we have $x > 1$" stands for an infinite number of execution paths.
- One way to describe program verification is that instead of actually executing a program on one set of inputs to get one set of outputs, we simulate execution on sets of states using reasoning on predicates. We describe a set of input states using a logical predicate and reason about the possible output states using rules of logic plus rules for program execution.
- So to do program verification we need predicates to describe sets of memory states, rules of logic to reason about predicates, plus rules for how our programs execute (i.e., how they take and modify memory states).

I. Logic Review/Overview, Part 1: Propositional Logic

- If you weren't a CS major as an undergrad and haven't seen propositional and predicate logic before, you should study up on it (see ***Course Prerequisites: Basic Logic*** on page 2.)
- ***Propositional logic*** is logic over ***proposition variables***, which are just variables that can have the values true or false. In propositional logic we study the logical connectives and ($\wedge$), or ($\vee$), not ($\neg$), implication ($\rightarrow$), and biconditional ($\leftrightarrow$) operating over variables that have true or false as their values. In computer science terms, propositional logic is the logic used for boolean expressions: True and false are boolean constants, and the connectives are boolean operators (in C, $\wedge$, $\vee$, $\neg$, and $\leftrightarrow$ are written *&&*, *||*, *!*, and *==*).
- ***Notation***: Typically we'll use p, q, ... for preposition variables or propositions and T, F for true and false.

Terminology

- $p \wedge q$ is the ***conjunction*** or ***logical and*** of p and q. We say that p and q are ***conjuncts*** of $p \wedge q$.
- $p \vee q$ is the ***disjunction*** or ***logical or*** of p and q. We say that p and q are ***disjuncts*** of $p \vee q$.

---

[2] Actually, it's probably a finite number of cases but still so many that "infinity" is a decent generalization.

- $p \to q$ is the ***implication*** or ***conditional*** of p and q. We say that p is the ***antecedent*** or ***hypothesis*** and q is the ***consequent*** or ***conclusion***.

Other Ways to Phrase Implications

- Other phrasings of $p \to q$: "***if*** p ***then*** q"; "p is ***sufficient for*** q"; "p ***only if*** q", "q ***if*** p"
- Other phrasings of $q \to p$: $p \leftarrow q$, "p is ***necessary for*** q"; "p ***if*** q"; "***if*** q ***then*** p", "q ***only if*** p".

Biconditional

- $p \leftrightarrow q$ is the ***equivalence*** or ***biconditional*** of p and q; they are both true or both false. p is the ***antecedent*** or ***hypothesis*** and q is the ***consequent*** or ***conclusion***.
- Note $\leftrightarrow$ is not the same as "equivalence". Equivalence is transitive: "If p is equivalent to q, and q is equivalent to r, then p is equivalent to r".
  - For two items, $p \leftrightarrow q$ is true exactly when p and q are both true or both false, so e.g., $F \leftrightarrow F$ evaluates to T. But for three items, $\leftrightarrow$ doesn't behave as you might expect: Since $F \leftrightarrow F$ evaluates to T, we can substitute it for the first T in $T \leftrightarrow T$ and get that $(F \leftrightarrow F) \leftrightarrow T$ evaluates to T.
- The kind of equivalence we want is called "logical equivalence" and it's related to $\leftrightarrow$ but not exactly the same. We'll look at it in a bit.

Precedences and Associativities for Propositional Operators

- **Precedences**: For the precedences of propositional operators, let's use $\neg$, $\wedge$, $\vee$, $\to$, $\leftrightarrow$ (going from strong to weak). E.g., $\neg p \wedge q \vee r \to s \leftrightarrow t$ means $((((\neg p) \wedge q) \vee r) \to s) \leftrightarrow t)$ [2023-01-10]. E.g., we'll take $p \to q \leftrightarrow p \vee \neg q$ to mean $(p \to q) \leftrightarrow (p \vee \neg q)$.
  - Sometimes people take $\to$ and $\leftrightarrow$ as having the same precedence, but in general, it doesn't much matter.
- ***Associativity***: $\wedge$ and $\vee$ are associative, so $((p \wedge q) \wedge r)$ and $(p \wedge (q \wedge r))$ have the same logical value. Let's make these operators left associative, so the fully parenthesization of $p \wedge q \wedge r$ is $((p \wedge q) \wedge r)$.
- Implication is right associative: $p \to q \to r$ means $(p \to (q \to r))$.
  - Unlike $\wedge$ and or, $\to$ is not associative: $((F \to T) \to F)$ and $(F \to (T \to F))$ don't ~~always~~ [2023-01-10] evaluate to the same result
- For the biconditional ($\leftrightarrow$), we'll use right associativity: $p \leftrightarrow q \leftrightarrow r$ means $(p \leftrightarrow (q \leftrightarrow r))$.
  - The biconditional is indeed logical equivalence on two values: $T \leftrightarrow T$ and $F \leftrightarrow F$ both evaluate to T, but $T \leftrightarrow F$ and $F \leftrightarrow T$ both evaluate to F.
  - But it's not logical equivalence on three values. If p and q are both F and r is T, then p, q, and r are not equivalent, but $(p \leftrightarrow (q \leftrightarrow r))$ equals $(F \leftrightarrow (F \leftrightarrow T))$ equals $F \leftrightarrow F$ equals T.

Semantic Equality

- ***Semantic equality*** is equality of meanings or results. This is usually what we mean when we write "=": $2 + 2 = 4$, $a + b = b + a$.

- For propositions, we'll use $\Leftrightarrow$ to indicate semantic equality, which turns out to be what we want for logical equivalence.
  - ***Example***: You can distribute $\vee$ over $\wedge$: $(p \wedge q) \vee r \Leftrightarrow (p \vee r) \wedge (q \vee r)$.
- For propositions, where we have only the values T and F (and only boolean variables), semantic equality can be mechanically determined (though for propositions, it can take time exponential in the number of basic variables).
- Later, when we add other kinds of values (like integers) to get predicates, semantic equality can be impractical or even impossible to determine, so we usually fall back on a property that's easier to determine, namely, syntactic equality.

Syntactic Equality

- ***Syntactic equality*** (written ≡) means equality as structured text: Two expressions or propositions are syntactically equal if they are textually identical — with one exception: We'll ignore redundant parentheses. E.g., $(1*2)+3 \equiv 1*2+3$. We'll use $\not\equiv$ for syntactic inequality. E.g., $2+2 \not\equiv 4$.
- We'll consider three kinds of redundancy for parentheses:
  - ***Precedence***: For example, $(p \wedge q) \vee r \equiv p \wedge q \vee r$ because the conjunction operator has higher precedence than the disjunction operator. If we want $p \wedge (q \vee r)$, the parentheses are necessary.
  - ***Left or Right Associativity***: For example, $p \to q \to r \equiv p \to (q \to r)$ because the implication operator is right associative, so if we want $(p \to q) \to r$, then the parentheses are necessary.
  - ***Associative Operators***: For an associative operator, all parenthesizations will be syntactically equal. E.g., $(p \wedge q) \wedge r \equiv p \wedge (q \wedge r)$. But since the subtraction operator is not associative, we have $(a-b)-c \not\equiv a-(b-c)$.
- ***Commutativity Not Included:*** We aren't going to take commutativity of operators into account, so $p \wedge q \not\equiv q \wedge p$.
  - Leaving out commutativity gives us two helpful properties: First, it makes "≡" easy to calculate. (It takes $O(n)$ time.) Second, the non-parenthesis symbols of two ≡ items have to appear in the same order. E.g., no parenthesizations of $1+3+2$ and $1+2+3$ make them ≡. Since they both equal (evaluate to) 6, we see that syntactically unequal items can stand for the same value.
- What about operator pairs like * and / or + and – where the members of the pair have equal precedence and one of the pair (but not both) are associative. For example, we know $(x+(y-z))$ and $((x+y)-z)$ are semantically equivalent[3]; do we want them to be syntactically equivalent?
- The answer will be no, but to justify this, we need to look a bit more at how syntactic and semantic equality are related, so we'll put off the explanation for a bit.

---

[3] Technically, on floating point numbers, they might be different.

Syntactic Equality Versus Semantic Equality

- ***Why Use Syntactic Equality?*** We often want to know whether two items are semantically equal, but depending on the kind of item, semantic equality can be hard or even impossible to calculate. Syntactic equality is easy to calculate, and if we define ≡ carefully, then we can guarantee that if two items are ≡, then they're semantically =. In other words, we use syntactic equality to be a rough approximation of semantic equality — "rough' because two items can be syntactically unequal but semantically equal.
- ***Syntactic Equality Implies Semantic Equality***: Keeping this property in mind makes it easy to see why we can ignore redundant parentheses when determining ≡ because preserving = is what makes parentheses redundant. E.g., $1+2*3 \equiv (1+(2*3))$, so they stand for the same value. Since "≡ implies =" is true, the contrapositive "≠ implies ≢" is also true. E.g., $2+2 \neq 5$, so $2+2 \not\equiv 5$.
- ***Semantic Equality Does Not Imply Syntactic Equality***: A separate question from "≡ implies =" is the converse: Does = imply ≡? It's easy to find examples that tell us "No": $2+2 \not\equiv 4$, $a+0 \not\equiv a$, and $p \wedge q \not\equiv q \wedge p$.
- Back to mixing * and / (or + and –): We'd like syntactic equality to depend only on syntactic properties, but the equality of $(x+(y-z))$ and $((x+y)-z)$ is a property of the semantics of + and –, so we'll take the answer to be "No".

Parenthesizations

- The ***minimal parenthesization*** of a syntactic item is the one with the fewest parentheses that preserves ≡. (I.e., it is still ≡ to the original.)
- For associative operators, let's omit parentheses inside sequences like $p \wedge q \wedge r$ or $p_1 \wedge (q_1 \vee q_2 \vee q_3)$.
- The ***full parenthesization*** of an item is the one that preserves ≡ and also includes parentheses around each operator expression (i.e., $($ operator $p)$ for a unary operator or $(p$ operator $q)$ for a binary operator).
  - We'll omit parentheses around constants, variables, and already-parenthesized expressions; we don't want to be writing things like $(((1)+(((2)*(3)))))$.
  - Technically, the outer parentheses of $(1+(2*3))$ are required, but let's take omitting them to be an ignorable small mistake[4].
  - For associative operators like + and *, let's write using left associativity, just to avoid having multiple correct results. So we'll take $(((1+2)+3)+4)$ to be the full parenthesization of $1+2+3+4$.
  - Note: A fully parenthesized item has the same number of pairs of parentheses as it has number of operators. E.g., $(1+(2*3))$ has two pairs of parentheses: one for the + and one for the *.

[4] This is just a hack tossed in to save me if I forget to write the outermost parentheses sometimes.

J. Semantics of Propositional Logic

- The typical semantics for propositional logic uses truth tables as below.

p	q	$p \wedge q$	$p \vee q$	$p \to q$	$p \leftrightarrow q$
F	F	F	F	T	T
F	T	F	T	T	F
T	F	F	T	F	F
T	T	T	T	T	T

p	$\neg p$
F	T
T	F

- Implication sometimes bothers people ("Why does $F \to T$?")
  - Basically, $p \to q$ means "p is less true than or equal to q".
  - If you treat T, F, and $\to$ as being like 1, 0, and $\leq$, then $F \to T$ is like $0 \leq 1$.

States and Satisfaction *[2023-01-11 starts here]*

[2023-01-10: various changes start]

- To talk about the truth of a proposition, we'll use ***states***. In CS terms, our states just model computer memory states: They connect variables and values, and we'll talk about a proposition as being ***satisfied*** by a state if it's true given the state's variables and values.
- In terms of propositions, a state represents one truth table row of possible values for a set of proposition variables. Satisfaction means that a proposition is true for that particular truth table row. E.g., the table above has p and q both false for its first row's state; $p \to q$ is satisfied in that state but $p \wedge q$ isn't.
- ***Definition***: A (***well-formed***) ***state*** σ is a finite function from proposition letters to truth values. We'll normally represent them as a finite set of pairs (a.k.a. ***bindings***) of a proposition letter and a truth value. Since a state is a function, there can only be one binding for any given variable.
- ***Notation***: Instead of write a binding as a pair (p, T), we'll write it as $p = T$, just for readability. E.g., the state where p and q are both false is $\{p = F, q = F\}$.
- ***Definition***: A set of bindings is ***ill-formed*** as state if includes more than one binding for some variable, or it includes things that aren't bindings of proposition letters to truth values. E.g., $\{p = T, \ p = F\}$ is ill-formed as a state, since it has two bindings for p.
- A couple of remarks: The smallest state is the empty state $\varnothing$ (it has the empty set of bindings). Second, since states are sets of bindings, the order of presentation of bindings doesn't matter: $\{p = F, \ q = T\}$ and $\{q = T, \ p = F\}$ are the same state.
- ***Not***e: It's important for the bindings to involve only proposition letters, not more complicated propositions. E.g., $\{p \vee q = F\}$ is ill-formed. On the other hand, though we might write just T or F for the value, you can use any mathematical description of a value. E.g., "$\{p = T\}$" and "$\{p = \alpha\}$ where α is T if two plus two equals four" describe the same state.

[end of 2023-01-10 changes]

- ***Definition***: A proposition is ***satisfied in*** (or ***by***) a state if it evaluates to true in that state. E.g., $p \vee \neg q$ is satisfied in $\{p=T,\ q=F\}$. The proposition is ***not satisfied*** (or ***unsatisfied***) in a state if it evaluates to false in that state[5].
- ***Notation***: If σ is a state and p is a proposition, then $\sigma \models p$ means that σ satisfies p, and $\sigma \not\models p$ means σ does not satisfy p. (The $\models$ symbol is a "double turnstile", if you haven't run across it before.) If σ is not a well-formed state, then we can't even ask $\sigma \models$ or $\not\models p$.
- ***Examples***: $\{p=T\} \models p$, $\{p=F\} \models \neg p$, and $\{p=T,\ q=F\} \models p \vee \neg q$. For nonsatisfaction, $\{p=T\}$ $\neg p$, $\{p=F\} \not\models p$, and $\{p=T,\ q=F\} \models p \wedge \neg q$. A less-obvious case is $\varnothing \models T \wedge (F \rightarrow T)$. Here, the state is empty, but that's okay because the proposition doesn't contain any variables, just the constants T and F.
- ***Note***: For satisfaction purposes, it's okay for a state to have unused bindings (bindings of variables that don't appear in the proposition). So if σ and τ are two states, then if $\sigma \models p$ and $\sigma \subseteq \tau$, then $\tau \models p$. Since every state extends the empty set and the empty set satisfies $T \wedge (F \rightarrow T)$, we get that every state satisfies $T \wedge (F \rightarrow T)$.
- Though extra bindings are okay, not having enough bindings can produce an unresolvable situation. For example, we can't say $\varnothing \models p \wedge q$ because we can't evaluate $p \wedge q$ in $\varnothing$ and get true. But we also can't say $\varnothing \not\models p \wedge q$, since we can't evaluate p ∧ q and get false. Does this prevent us from making statements like p ∨ ¬ p is satisfied by every state"?
- So ill-formedness prevents us from using something as a state. There's another kind of error to look at too. Say σ is well-formed, so that questions like $\sigma \models 2+2=4$ have an answer. If σ maps a variable to something other than T or F, then even though σ is well-formed, it's not useable for propositions that include that variable.
- ***Definition***: A state is ***proper*** for a proposition p if it includes bindings for all the variables of p and the state binds each variable to a value of the correct type (T or F for now; it gets more complicated once we include data values). Note σ could be proper for p_1 but not for p_2; it depends on what variables σ, p_1, and p_2 use.
- ***Asking About All/Some/etc. States***: When we say things like "p is satisfied in all σ" or "Let σ be a state that doesn't satisfy p", we'll quietly assume that we're only talking about the states proper for p. So we can say "$\sigma \models p \vee \neg p$ for all σ" even though $p \vee \neg p$ certainly isn't satisfied in $\varnothing$ or in $\{q=T\}$, as two of the infinitely many examples.
- To sum up, for any arbitrary set of bindings σ and a proposition p, we can have four situations
  - σ is ill-formed (not a state at all).
  - σ is (well-formed but) improper for p.
  - σ is well-formed for p, p evaluates to T under σ, so $\sigma \models p$ and $\sigma \not\models \neg p$.
  - σ is well-formed for p, p evaluates to F under σ, so $\sigma \not\models p$ and $\sigma \models \neg p$.

[5] When we get to runtime errors, there will be a third possibility. E.g., asking "Does $\{x=0\}$ satisfy $x/x=1$?" yields an error.

- Note since we don't have to worry about runtime errors, if σ is well-formed for p, then the statements $\sigma \models \neg p$ and $\sigma \not\models p$ have the same answer. Similarly, $\sigma \models p$ and $\sigma \not\models \neg p$ have the same answer.

Validity, Tautologies, and Logical Equivalence

- Now that we have the notion of a proposition being true in a given truth table column, we can go further and talk about a proposition being true in every truth table column.
- ***Definition***: p is ***valid*** (notation: $\models p$) if $\sigma \models p$ for every σ.
- ***Examples***: $\models T$, $\models \neg F$ (the two simplest examples), $\models p \vee \neg p$, $\models (p \to q) \leftrightarrow (\neg p \vee q)$.
- ***Definition:*** p is ***invalid*** (i.e., not valid), written $\not\models p$, if $\sigma \not\models p$ for some σ. (And recall that $\sigma \not\models p$ and $\sigma \models \neg p$ mean the same thing (until we get to runtime errors).
- ***Examples***: $\not\models p \vee q$, since $\{p = F,\ q = F\} \not\models p \vee q$. Another example: $\not\models p \wedge \neg p$, since $p \wedge \neg p$ is false for $\{p = T\}$ and $\{p = F\}$ both. Note that no state satisfies $p \wedge \neg p$ but some states do satisfy $p \vee q$. (E.g., $\{p = T,\ q = F\} \models p \vee q$.)
- One way to categorize propositions is through their validity.
- ***Definition***: p is a ***tautology*** if $\models p$, a ***contradiction*** if $\models \neg p$, and a ***contingency*** if $\not\models p$ and $\not\models \neg p$ (simultaneously). Another way to say this is that a tautology has a truth table column with only T values, a contradiction has a truth table column with only F values, and a contingency has a column that includes at least one T and at least one F.
- Some properties:
  - If p is a tautology, then $\neg p$ is a contradiction and vice versa.
  - If p is a contingency, then so is $\neg p$ and vice versa.
  - If p is not a tautology, then it is a contingency or a contradiction.
  - If p is not a contradiction, then it is a contingency or a tautology.
  - If p is not a contingency, then it is a tautology or a contradiction.

- ***Definition***: Two propositions p and q are ***logically equivalent*** (written $p \Leftrightarrow q$) if $\models p \leftrightarrow q$. I.e., in a truth table, the column for p matches the one for q. It's easy to show that $\Leftrightarrow$ is transitive: If $p_1 \Leftrightarrow p_2$ and $p_2 \Leftrightarrow p_3$, then the columns for p_1 and p_2 match, the columns for p_2 and p_3 match, so the columns for so p_1 and p_3 match.
- ***Notation***: "$\Leftrightarrow$" is often pronounced "if and only if", so people often write "p iff q" for what we're writing as $p \Leftrightarrow q$.
- We most often use $\Leftrightarrow$ to talk about how a sequence of step-by-step transitions produces propositions that are all logically equivalent. E.g., $p \to q \Leftrightarrow \neg p \vee q \Leftrightarrow q \vee \neg p \Leftrightarrow \neg\neg q \vee \neg p \Leftrightarrow \neg q \to p$.
- It turns out that $\Leftrightarrow$ on propositions is like = on arithmetic expressions.
  - "$(x+1)^2 = (x^2+2x+1)$" means we can substitute one for the other in any semantic context.

- Similarly $p \Leftrightarrow q$, then in any semantic context, we can always substitute p for q or vice versa.
- Note in both cases, the context has to be semantic. E.g., the length of the expression $(x+1)^2$ is 6, but we can't replace $(x+1)^2$ with (x^2+2x+1) in that statement.

- Remember: $\Leftrightarrow$ and $\leftrightarrow$ are similar but not identical.
  - The $\leftrightarrow$ symbol is a syntactic operator and can appear in propositions. (More generally, we can use it in boolean expressions: In C, $\leftrightarrow$ is written ==.) On the other hand, $\Leftrightarrow$ is a semantic operator: It doesn't appear in propositions because it describes a semantic property.
  - The $\Leftrightarrow$ operation is transitive: if $T \Leftrightarrow p_1 \Leftrightarrow p_2 \Leftrightarrow \ldots$ etc., then all the p's evaluate to T. The $\leftrightarrow$ operation, however, is not transitive. For example, $(T \leftrightarrow (F \leftrightarrow F))$ evaluates to T but T, F, and F are certainly not all logically equivalent.
  - "$(x+1)^2+3 = (x^2+2x+1)+3 = x^2+2x+4$"? Here, "=" is being used on numbers the same way we use $\Leftrightarrow$ on propositions.
  - So $p \Leftrightarrow q \Leftrightarrow r \Leftrightarrow s$ means ($p \Leftrightarrow q$ and $q \Leftrightarrow r$ and $r \Leftrightarrow s$); i.e., all four are true or all four are false.
  - Compare this to $F \leftrightarrow F \leftrightarrow T \leftrightarrow T$, which $\equiv (F \leftrightarrow (F \leftrightarrow (T \leftrightarrow T)))$, which evaluates to true.

Relations Between Implications

- The ***contrapositive*** of $p \rightarrow q$ is $\neg q \rightarrow \neg p$; its ***converse*** is $q \rightarrow p$; its ***inverse*** is $\neg p \rightarrow \neg q$. An implication is equivalent to its contrapositive; similarly, the converse of an implication is equivalent to its inverse: $(p \rightarrow q) \Leftrightarrow (\neg q \rightarrow \neg p)$ and $(q \rightarrow p) \Leftrightarrow (\neg p \rightarrow \neg q)$.

More Equivalences

- ***Definition of implication***: $p \rightarrow q \Leftrightarrow \neg p \vee q$.
- ***Negation of implication***: $\neg(p \rightarrow q) \Leftrightarrow p \wedge \neg q$.
  - Note the negation and the inverse of $p \rightarrow q$ are different. $\neg(p \rightarrow q) \Leftrightarrow (p \wedge \neg q)$ but $(\neg p \rightarrow \neg q) \Leftrightarrow (p \vee \neg q)$.
- ***Definition of biconditional***: $(p \leftrightarrow q) \Leftrightarrow (p \rightarrow q) \wedge (q \rightarrow p)$.
- ***Exclusive or of p and q***: The exclusive or of p and q is true when one of them is true and the other one is false.
  - The exclusive or is the negation of the biconditional; momentarily writing $p \oplus q$ for the exclusive or of p and q,
    - I.e., $p \oplus q \Leftrightarrow (p \wedge \neg q) \vee (\neg p \wedge q) \Leftrightarrow \neg(p \leftrightarrow q)$
  - The usual "inclusive" or allows one or both of p and q to be true:
    - I.e., $(p \vee q) \Leftrightarrow (\neg p \wedge q) \vee (p \wedge \neg q) \vee (p \wedge q)$.
    - Also, $p \oplus q \Leftrightarrow (p \vee q) \wedge \neg(p \wedge q)$.

Propositional and Predicate Logic

CS 536: Science of Programming, Fall 2022

ver Tue 2023-01-10, 14:35

A. Why

- Reviewing/overviewing logic is necessary because we'll be using it in the course.
- We'll be using predicates to write specifications for programs.
- Predicates and programs have meaning relative to states.

B. Outcomes

At the end of this class, you should

- Be able to prove simple logical equivalences of propositions from a basic set of rules.
- Know the syntax for predicates (including primitive tests and the quantifiers $\forall$ and $\exists$).
- Understand how states for predicates differ from states for propositions.
- Understand how $\models$ works for non-quantified predicates (for quantified ones we'll need state updates, which we'll see next time).

C. In Case You Missed The First Class

- The course webpages are at http://cs.iit.edu/~cs536/. Read it carefully for policies and refer to it often to download class notes and homework assignments. Check myIIT→Blackboard for class videos.

D. Formal Proofs of Truth

- For propositions, in addition to semantic truth based on truth tables, there is also a notion of ***provable truth*** based on syntactic manipulation of propositions. E.g., "if $p \wedge q$ is provable then $q \wedge p$ is provable" or "$q \wedge p$ follows from $p \wedge q$".
- ***Notation***: if $\vdash p \wedge q$ then $\vdash q \wedge p$. The $\vdash$ symbol is a "turnstile" (compare to $\models$ for semantic truth) and it's pronounced "can prove" or something similar.
- ***Definition***. Given a set of proof rules, two propositions are ***provably equivalent*** if each follows from the other according to those rules. E.g., $p \wedge q$ and $q \wedge p$ are provably equivalent.

Propositional Logic Rules

- You don't need to memorize these rules by name, but you should be able to give the name of a rule. For example, "$(p \rightarrow q) \wedge (p \rightarrow r) \Rightarrow (p \rightarrow r)$ is _____". (Answer: transitivity)
- The rules use the $\Leftrightarrow$ symbol to indicate that each side can be used to prove the other: $lhs \Leftrightarrow rhs$ means that if you can prove the *lhs*, then you can prove the *rhs* and vice versa.

- Using $\vdash$, we can write this as $\vdash lhs \Leftrightarrow rhs$ if and only if both $\vdash lhs$ implies $\vdash rhs$ and $\vdash rhs$ implies $\vdash lhs$.
- ***Logical implication vs logical equivalence***: Analogously to how $(p \leftrightarrow q) \Leftrightarrow T$ lets us know $p \Leftrightarrow q$, if we know $(p \rightarrow q) \Leftrightarrow T$ then we say that $p \Rightarrow q$ (p *logically implies* q).[1] Within the basic proof rules below, $\Rightarrow$ appears in the transitivity rule, and it's critical there: $(p \rightarrow q) \wedge (q \rightarrow r)$ implies $(p \rightarrow r)$ but $(p \rightarrow r)$ doesn't imply $(p \rightarrow q) \wedge (q \rightarrow r)$.

- The set of rules below isn't unique — there are other sets of rules that are equivalent, in the sense of what you can prove using them.

Commutativity $p \vee q \Leftrightarrow q \vee p$

$p \wedge q \Leftrightarrow q \wedge p$ $(p \leftrightarrow q) \Leftrightarrow (q \leftrightarrow p)$

Associativity $(p \vee q) \vee r \Leftrightarrow p \vee (q \vee r)$

$(p \wedge q) \wedge r \Leftrightarrow p \wedge (q \wedge r)$

Distributivity/Factoring

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

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

Transitivity [Note: $\Rightarrow$, not $\Leftrightarrow$ here]

$(p \rightarrow q) \wedge (q \rightarrow r) \Rightarrow (p \rightarrow r)$

$(p \leftrightarrow q) \wedge (q \leftrightarrow r) \Rightarrow (p \leftrightarrow r)$

Identity: $p \wedge T \Leftrightarrow p$ and $p \vee F \Leftrightarrow p$

Idempotentcy: $p \vee p \Leftrightarrow p$ *and* $p \wedge p \Leftrightarrow p$

Domination: $p \vee T \Leftrightarrow T$ and $p \wedge F \Leftrightarrow F$

Absurdity: $(F \rightarrow p) \Leftrightarrow T$

Contradiction: $p \wedge \neg p \Leftrightarrow F$

Excluded middle: $p \vee \neg p \Leftrightarrow T$

Double negation: $\neg \neg p \Leftrightarrow p$

DeMorgan's Laws:

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

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

Definition of $\rightarrow$ $(p \rightarrow q) \Leftrightarrow (\neg p \vee q)$

Definition of $\leftrightarrow$ $(p \leftrightarrow q) \Leftrightarrow (p \rightarrow q) \wedge (q \rightarrow p)$

Negation of Comparisons (in predicate logic)

$\neg (e_1 \leq e_2) \Leftrightarrow e_1 > e_2$ (similar for $<, >, \geq, =, \neq$)

Substitution: Given p, q, and r, let r' be the result of substituting a q for one or more occurrences of p inside r. If $p \Leftrightarrow q$ then $r \Leftrightarrow r'$. ***Example*** (of substitution): Using $(p \rightarrow q) \Leftrightarrow (\neg p \vee q)$ for r, p for p, and $\neg \neg p$ for q, by substitution, we know $(p \rightarrow \neg \neg p) \Leftrightarrow (\neg p \vee \neg \neg p)$.

E. Sample Proofs

- Proofs in predicate logic give step-by-step reasoning for why we think the truth of one proposition is related to another.
- Here is a proof of $\neg (p \rightarrow q) \Leftrightarrow (p \wedge \neg q)$ (also known as "negation of $\rightarrow$").

$\neg (p \rightarrow q)$	
$\Leftrightarrow \neg (\neg p \vee q)$	Defn $\rightarrow$
$\Leftrightarrow \neg \neg p \wedge \neg q$	DeMorgan's Law
$\Leftrightarrow p \wedge \neg q$	Double negation

[1] Unfortunately, we're running out of word phrases, so "logically implies" in English can mean $\rightarrow$ or $\Rightarrow$. But usually you can figure it out from the context (or just write the symbol, which isn't ambiguous).

- The proof above can be read top-down or bottom-up.
  - Top-down: if $\neg(p \to q)$ is provable, then $(p \wedge \neg q)$ is provable, using the indicated rule.
  - Bottom-up: if $(p \wedge \neg q)$ is provable, then $\neg(p \to q)$ is provable, using the indicated rule.
- If you reverse a $lhs \Leftrightarrow rhs$ proof, you get an equivalent $rhs \Leftrightarrow lhs$ proof
- So this proof of $p \wedge \neg q \Leftrightarrow \neg(p \to q)$ is also negation of $\neg$.

$\neg(p \to q)$	
$\Leftrightarrow \neg(\neg p \vee q)$	Defn $\to$
$\Leftrightarrow \neg\neg p \wedge \neg q$	DeMorgan's Law
$p \wedge \neg q$	Double negation

- For another sample proof, here is $((r \to s) \wedge r) \to s \Leftrightarrow T$. Its name, "modus ponens" is Latin, but observations of it were known to the ancient Greeks.

$(r \to s) \wedge r \to s$	
$\Leftrightarrow \neg((r \to s) \wedge r) \vee s$	Defn of $\to$
$\Leftrightarrow (\neg(r \to s) \vee \neg r) \vee s$	DeMorgan's Law
$\Leftrightarrow ((r \wedge \neg s) \vee \neg r) \vee s$	Negation of$\to$ [see above]
$\Leftrightarrow ((r \vee \neg r) \wedge (\neg s \vee \neg r)) \vee s$	Distribute$\vee$over $\wedge$
$\Leftrightarrow (T \wedge (\neg s \vee \neg r)) \vee s$	Excluded middle
$\Leftrightarrow (\neg s \vee \neg r) \vee s$	Identity
$\Leftrightarrow T \vee \neg r$	Excluded middle (see below)
$\Leftrightarrow T$	Domination

- In contrast, a proof of $lhs \Rightarrow rhs$ can only be read correctly from the top down.

Avoid Unpleasant Levels of Detail

- In the proof above, if we're being picky with details, then the use of excluded middle to go from $(\neg s \vee \neg r) \vee s$ to $T \vee \neg r$ is

$(\neg s \vee \neg r) \vee s$	
$\Leftrightarrow \neg s \vee (\neg r \vee s)$	Associativity of $\vee$
$\Leftrightarrow \neg s \vee (s \vee \neg r)$	Commutativity of $\vee$
$\Leftrightarrow (\neg s \vee s) \vee \neg r$	Associativity of $\vee$
$\Leftrightarrow T \vee \neg r$	Excluded middle

- And if we're being even pickier, we should go from $(\neg s \vee s) \vee \neg r$ to $(s \vee \neg s) \vee \neg s$, since the rule for excluded middle says "$p \vee \neg p \Leftrightarrow T$", not "$\neg p \vee p \Leftrightarrow T$". To avoid this level of detail, let's agree that associativity and commutativity can be used without mentioning them specifically.
- ***Notation***: In propositional logic proofs (and later, predicate logic proofs), we can omit uses of associativity and commutativity rules and treat them as being implicit. (It's still okay to spell them out, of course.)

F. Derived Rules

- If $p \leftrightarrow q$ is a tautology (i.e., $(p \leftrightarrow q) \Leftrightarrow T$), then we can use $p \Leftrightarrow q$ as a ***derived rule***. E.g., to prove modus ponens, we showed $(r \rightarrow s) \wedge r \rightarrow s \Leftrightarrow T$. Here's an example of using modus ponens. (For r we substitute $p \wedge q$; for s we substitute r.)

$((p \wedge q) \rightarrow r) \wedge (p \wedge q) \rightarrow r$	Modus ponens
$\Leftrightarrow T$	

- Here we see $p \wedge q \rightarrow p$ is a tautology, often called "*(left) and-elimination*". (There's also "right and-elimination, which is similar.)

$p \wedge q \rightarrow p$	
$\Leftrightarrow \neg(p \wedge q) \vee p$	Defn $\rightarrow$
$\Leftrightarrow (\neg p \vee \neg q) \vee p$	DeMorgan's Law
$\Leftrightarrow T \vee \neg q$	Excluded middle
$\Leftrightarrow T$	Domination

- Some other common derived rules: [you don't have to memorize these]. Note that *or-introduction* uses $\Rightarrow$, since although p implies $p \vee q$, you can have $p \vee q$ true but with p false. Similarly or-elimination uses $\Rightarrow$ because you can have r true but $p \vee q$ false.
  - *contraposition*: $(p \rightarrow q) \Leftrightarrow (\neg q \rightarrow \neg p)$
  - *and-introduction*: $p \rightarrow (q \rightarrow r) \Leftrightarrow p \wedge q \rightarrow r$
  - *or-introduction*: $p \Rightarrow p \vee q$
  - *or-elimination*: $(p \vee q) \wedge (p \rightarrow r) \wedge (q \rightarrow r) \Rightarrow r$
  - *not-introduction*: $(p \rightarrow F) \Leftrightarrow \neg p$

G. Predicate Logic

- In propositional logic, we assert truths about boolean values; in predicate logic, we assert truths about values from one or more "***domains of discourse***" like the integers.
- We extend propositional logic with *domains* (sets of values), plus variables whose values range over these domains, and operations on values (e.g. addition). E.g., for the integers we add the set $\mathbb{Z}$, operations +, –, *, /, % (*mod*), and relations =, ≠, <, >, ≤, and ≥. We'll also add arrays of integers and array indexing, as in $b[0]$.
- A ***predicate*** is a logical assertion that describes some property of values.

- To describe properties involving values, we add basic relations on values (e.g., less-than). We also have rules over these relations, like $x * 0 = 0$ being a rule of arithmetic.

States for Predicates; Satisfaction and Validity of Predicates (Omitting quantifiers)

- We'll see quantifiers in a bit, but in the meantime, we can still look at the satisfaction relation for predicates.
- ***States with bindings for Domain variables***. With propositions, for states, we've been writing sets of bindings like $\{p = T,\ q = F\}$, where proposition variables are bound to boolean values [2]. With predicates, we can also have bindings like $x = 0$, which bind a domain variable to a domain value. E.g., $\{p = T,\ q = F,\ x = 0\}$ is a state now. We might have to give the value a name like α if we don't know it precisely, for example $\{p = T,\ q = F,\ x = \alpha\}$. [I'll generally use Greek letters for semantic values.]
- ***States as functions***: Technically, a state is a function from variables (i.e., symbols) to values (proposition variables to boolean values, domain variables to domain values).
- ***Notation***: $\sigma(x)$ is the value of the state function σ on variable x. (I.e., it's the value α where the binding $x = \alpha$ appears in σ).
- ***Notation***: $\sigma(e)$ is the value of the expression e given that its variables take their values from σ. Eg., if $\sigma = \{x = 1,\ y = 3\}$, then $\sigma(x) = 1$, $\sigma(y) = 3$, and (picking a random expression), $\sigma(x + y * y) = 10$. We'll define the $\sigma(expression)$ idea more formally later, but for now your intuition should be fine.
- ***Notation***: p, q, r, etc. can stand for propositions or predicates. Propositions are also predicates, so technically we don't need to say "propositions or ...", but its good to emphasize the distinction.
- ***Satisfaction of Unquantified Predicates***. State σ ***satisfies*** predicate p, written $\sigma \models p$ is defined as follows.
  - $\sigma \models p$ (a proposition variable) if $\sigma(p) = T$.
  - $\sigma \models e_1 = e_2$ holds if the values $\sigma(e_1)$ and $\sigma(e_2)$ are equal. Tests $<$, $\leq$, $>$, $\geq$, and $\neq$ are similar.
  - $\sigma \models \neg p$ holds if $\sigma \not\models p$ holds.
  - $\sigma \models p \wedge q$ holds if $\sigma \models p$ and $\sigma \models q$ both hold.
  - $\sigma \models p \vee q$ holds if $\sigma \models p$ or $\sigma \models q$ or both hold.
  - $\sigma \models p \rightarrow q$ holds if $\sigma \models \neg p \vee q$ holds, or equivalently if $\sigma \models \neg p$ or $\sigma \models q$ or both hold.
  - $\sigma \models p \leftrightarrow q$ holds if $\sigma \models (p \rightarrow q) \wedge (q \rightarrow p)$ holds.
- Some points about this definition. First, it seems weird — all we're doing is substituting English words like "and" for logic symbols like "$\wedge$". And that's true. All this means is that the proposition connectives behave as you expect them to if you were to write out things in English.

---

[2] Remember, $\{p = T,\ q = F\}$ is just more-readable shorthand for $\{(p, T),\ (q, F)\}$.

Second, the last two rules (for implication and biconditional) don't break down the p and q into separate parts, they substitute some larger predicate for $p \to q$ and $p \leftrightarrow q$. This is because these operations can be described using other connectives.

H. Well-Formed and Proper States

- We have seen well-formedness and properness for states for propositional expressions. States for predicates follow the same concepts, the only difference being the kinds of values that variables map to: In addition to boolean values, there are also values from whatever domain our predicates range over (such as $\mathbb{Z}$ for integers).
  - ***Examples:*** $\{x=1, y=1\}$ is well-formed but $\{x=1, y=x\}$ and $\{x=1, y=x*1\}$ are ill-formed, since they map y to an identifier and an expression respectively. $\{x=1, x=2\}$ is ill-formed because it's not a function (no unique value for x).
  - ***Examples:*** $\{b=3\}$ is proper for $b+2$ but not proper for $b[2]$ (assuming $b[2]$ is an array lookup). It's also improper for $b+x*0$ because it is missing a binding for x (even though the value of x is irrelevant).
- And recall that being well-formed is an intrinsic property of a state; being proper is a relationship between a state and expression or proposition or predicate. Also, when we say "every state" or "no state", we only look at states proper for the context.
  - ***Examples***: We can say "$\sigma(x+0)=\sigma(x)$ in every state" because it's satisfied in every proper state (includes at least a mapping of x to an integer). Ill-formed states like $\{x=y, y=5\}$ or improper states like $\{y=4\}$ are ignored.
- One more point: A state might be proper but that doesn't preclude runtime errors.
  - ***Examples:*** States $\{x=8,\ y=2\}$ and $\{x=8,\ y=0\}$ are both proper for expression x/y, but evaluating the expression gets a runtime error when y is zero.

I. Quantifiers: Syntax

- When a predicate includes a variable, we have to ask for what values of the variable we think the predicate might be true: Some current value? Every value? Some value?
- We use quantifiers to specify all values, some value, and exactly one value.

Universal Quantification

- A ***universally quantified predicate*** (or just "***universal***" for short) has the form $(\forall x \in S.\, p)$ where S is a set and p (the body of the universal) is a predicate involving x. E.g., every integer greater than 1 is less than its own square: $(\forall x \in \mathbb{Z}.\, x>1 \to x<x^2)$ [3].
- Often we leave out the set if it is understood. E.g., $(\forall x.\, x>1 \to x<x^2)$.

[3] The standard definition of the natural numbers $\mathbb{N}$={0, 1, 2, ...}. Avoid any source that says $0 \notin \mathbb{N}$.

Existential Quantification

- An ***existentially quantified predicate*** (or just "***existential***" for short) has the form $(\exists x \in S \,.\, p)$ where S is a set and p (the body of the existential) is a predicate involving x. E.g., there is a nonzero integer that equals its own square: $(\exists x \in \mathbb{Z} \,.\, x \neq 0 \wedge x = x^2)$.
- Usually the set is understood to be $\mathbb{Z}$ and we leave it out. E.g., $(\exists x \,.\, x \neq 0 \wedge x = x^2)$.

Syntactic Equality of Quantified Predicates

- For syntactic equality, which variable you quantify over will make a difference for us, so we'll treat: $(\forall x \,.\, x > x\text{-}1) \not\equiv (\forall y \,.\, y > y\text{-}1)$. But they'll be logically equivalent: $(\forall x \,.\, x > x\text{-}1) \Leftrightarrow (\forall y \,.\, y > y\text{-}1)$.

Bounded Quantifiers

- With bounded quantifiers, we abbreviate a quantifier with a condition in the body by moving the condition to the quantifier. We'll take a predicate with bounded quantifier to be ≡ the one without.
- ***Example 1:*** $(\forall x > 1 \,.\, x < x^2) \equiv (\forall x \,.\, x > 1 \rightarrow x < x^2)$. ("For every $x > 1$, $x < x^2$".)
- ***Example 2:*** $(\exists x \neq 0 \,.\, x = x^2) \equiv (\exists x \,.\, x \neq 0 \wedge x = x^2)$. ("There's some nonzero x such that $x = x^3$".)
- ***Definition***: A ***bounded quantifier*** takes the form $Q\, p \,.\, q$ as an ≡ abbreviation for $Q\, x \;op\; r$ (where Q is $\forall$ and op is $\rightarrow$ or Q is $\exists$ and op is $\wedge$). More specifically,
- ***Definition***: $\forall p \,.\, q$ means $\forall x \,.\, p \rightarrow q$ where x appears in p and x is understood to be the variable we are quantifying over.
- ***Definition***: $\exists p \,.\, q$ means $\exists x \,.\, p \wedge q$ where z appears in p and x is understood to be the variable we are quantifying over. (Note: it's $p \wedge q$ here; compare with $p \rightarrow q$ for bounded universals.)
- It's important to expand bounded $\forall$ and $\exists$ to the correct connectives. For example, take
  - $\exists x \in \mathbb{Z} \,.\, x > 1 \wedge x = x^2$ which is false
  - $\exists x \in \mathbb{Z} \,.\, x > 1 \rightarrow x = x^2$ which is true, e.g. when $x = -1$.

Parentheses for Quantified Predicates

- We'll treat $\forall$ and $\exists$ as having low precedence. (Note: Some people use high precedence). So the body of a quantified predicate is as long as possible.
- ***Example 3***. $\forall x \in \mathbb{Z} \,.\, x > 1 \rightarrow x < x^2$ means $(\forall x \in \mathbb{Z} \,.\, ((x > 1) \rightarrow (x < x^2))$.
- ***Example 4***: $\forall x \in \mathbb{Z} \,.\, \exists y \in \mathbb{Z} \,.\, y \leq x^2$ means $(\forall x \in \mathbb{Z} \,.\, (\exists y \in \mathbb{Z} \,.\, (y \leq x^2)))$.
- ***Notation***: Q means $\forall$ or $\exists$.
- If we have (... $Q\, x$ ...) where the two parentheses shown match, then the body can't extend past the right parenthesis, and we get (... $Q\, x \,.$ (...)).
- ***Example 5***: $(\exists y \in \mathbb{Z} \,.\, y > 0 \wedge x > y) \rightarrow x \geq 1 \equiv ((\exists y \in \mathbb{Z} \,.\, ((y > 0) \wedge (x > y))) \rightarrow (x \geq 1))$
- For full parenthesizations, we add parentheses around basic tests, but we still omit them around variables and constants. Let's also omit them around array indexes, so we'll write $(b[x+1] > y)$, not $(b[(x+1)] > y)$.

- **Example 6**: $x > 0 \wedge y \leq 0$ expands to $((x > 0) \wedge (y \leq 0))$.

J. Quantifiers: Semantics

DeMorgan's Laws For Quantified Predicates

- For quantified predicates, there are two more ***DeMorgan's Laws***:
  - $(\neg \forall x . p) \Leftrightarrow (\exists x . \neg p)$ and $(\neg \exists x . p) \Leftrightarrow (\forall x . \neg p)$
- With bounded quantifiers, because of how $\rightarrow$, $\neg$, and $\wedge$ are related,
  - $(\neg \forall p . q) \Leftrightarrow (\exists p . \neg q)$. I.e., $(\neg \forall x . p \rightarrow q) \Leftrightarrow (\exists x . \neg (p \rightarrow q)) \Leftrightarrow (\exists x . p \wedge \neg q) \Leftrightarrow (\exists p . \neg q)$.
  - $(\neg \exists p . q) \Leftrightarrow (\forall p . \neg q)$. I.e., $(\neg \exists x . p \wedge q) \Leftrightarrow (\forall x . \neg (p \wedge q))$
    $\Leftrightarrow (\forall x . \neg p \vee \neg q) \Leftrightarrow (\forall x . p \rightarrow \neg q) \Leftrightarrow (\forall p . \neg q)$.
- ***Example 7***: $\neg(\forall x . x > 0) \Leftrightarrow (\exists x . \neg (x > 0)) \Leftrightarrow (\exists x . x \leq 0)$
- ***Example 8***: $\neg(\forall x > 0 . x^2 = x) \Leftrightarrow (\exists x > 0 . x^2 \neq x)$
- ***Example 9***: $\neg(\exists x . x \leq 0 \wedge x > 0) \Leftrightarrow (\forall x . \neg (x \leq 0 \wedge x > 0)) \Leftrightarrow (\forall x . x > 0 \vee x \leq 0)$.

Proofs of Quantified Predicates

- Formal systems for proving predicates are pretty complicated; rather than study one of them, let's rely on an informal idea of how to prove universally and existentially quantified predicates.
- In general, to prove $\forall x . p$, you prove p but without imposing any restrictions on x. If you need to restrict x, then this needs to be part of the body of the quantified predicate.
- ***Example 10***: To prove $\forall x \in \mathbb{Z} . x \neq 0 x \leq x^2$, we can say "Let x be an integer. Assume that x isn't zero. In that case, $x \leq x^2$."
- To prove $\exists x \in S . p$, you name a ***witness value*** for x and prove p holds if x has that value.
- ***Example 11***: To prove $\exists x \in \mathbb{Z} . x \neq 0 \wedge x \geq x^2$, the only value that works as a witness is 1. More generally, there may be multiple witness values that work; we just need to name one.
- If a predicate includes unquantified variables, then for it to be a tautology, it has to hold for all possible values of those quantified variables. It's a contradiction if it fails for all values, and it's a contingency if it holds for some values but not some others. (I.e., unquantified variables are "implicitly universally quantified".)
- ***Example 12***: $x > 0 \rightarrow \exists y . y^2 < x$ is a tautology because $\forall x . (x > 0 \rightarrow \exists y . y^2 < x)$ holds.
- ***Example 13***: $x > 0 \rightarrow y^2 < x$ is a contingency because it holds for some x and y (like $x = 2$ and $y x = 1$) but fails for others (like $x = y = 1$).
- ***Example 14***: $\exists y . (y < 0 \wedge y > x^2)$ is a contradiction because it fails for every value of x.

K. Predicate Functions

- Often, we'll give names to predicates and parameterize them. In programming languages, these predicate functions are written as functions that yield a boolean result.

- ***Example 15***: we might define $even(x) \equiv (x \% 2) = 0$, where % is the remainder operator. E.g., $Even(3) \equiv (3 \% 2) = 0 \Leftrightarrow 1 = 0 \Leftrightarrow F$.
- In a programming language, the body of a predicate function can be a general program — one that uses loops and decisions. We want our predicates to be simpler than that: We're going to use predicates to augment our programs with specifications, and it won't help if debugging a predicate function body is exactly as hard as debugging a general program.
- So we'll restrict ourselves to predicate functions that take one or more parameter variables and evaluate a predicate on those variables. ***Example***: $p(x) \equiv x > y \wedge x < z$.
- The body of the predicate function is (surprise) a predicate that evaluates to true or false, not an expression that yields a number.
- Function: $sqrt(x) \equiv$ (in pseudocode) r where $r = 0$ if $x \le 0$ and where $r * r = x$ if $x \ge 0$.
- Predicate function: $isSqrt(x, r) \equiv (x \le 0 \wedge r = 0) \vee r * r = x$.
- The body of a predicate function can use the parameter variables and the built-in relations for our datatypes (for integers, <, ≤, etc.) along with the propositional connectives (∧, ∨, etc.)
- ***Example 16***: Let's define $IsZero(b, m)$ to be true if the first m elements of b are all zero. To help, let's assume $size(b)$ gives the number of elements in b.
- Rewriting, $IsZero(b, m)$ means that $b[0], b[1], ..., b[m-1]$ all equal 0.
- It might be tempting to write $IsZero(b, m) \equiv b[0] = 0 \wedge b[1] = 0 \wedge \ldots \wedge b[m-1] = 0$
- But the right hand side is not a predicate; a predicate needs a fixed number of conjuncts being and'ed together.
- To write this as a predicate, we look for a pattern in our informal description: "$b[0], b[1], ..., b[m-1]$ all $= 0$" is equivalent to "$b[i] = 0$ for (every) $i = 0, 1, ..., m-1$". The implied "every" i tells us we need a universal quantifier ∀.
- So we can get $IsZero(b, m) \equiv \forall i.\, 0 \le i < m \rightarrow b[i] = 0$. With bounded quantifiers, we can write $IsZero(b, m) \equiv 0 \le i < m.\, b[i] = 0$.
- Another way to look at a description and find an equivalent predicate is to imagine writing a loop to calculate whether the property is true or false.
- E.g., with "*b[0], b[1], ..., b[m-1]* all $= 0$" we might imagine a loop

```
for i = 0 to m-1
    if b[i] ≠ 0 then return false
return true
```

- The "*for* $i = 0$ *to* $m-1$" tells us we need to search for i in the range $0 \le i < m$.
- The loop returns true only if ***all*** the $b[i]$ pass the $= 0$ test; this tells us we need $\forall i$.

The general translation for a universal is ∀ *loop var* . ((*var* in search range)→(test on *var*)). For this example, the search range is $0 \le i < m$ and the test on i is $b[i] = 0$. This gives us $\forall i.\, 0 \le i < m \rightarrow b[i] = 0$.

- We need a ∃ search if our loop needs to return true as soon as it finds a $b[i]$ that passes the test. E.g., if the property had been "At least one of $b[0]$, $b[1]$, ..., $b[m-1]=0$", we might imagine a loop

```
for i = 0 to m-1
    if b[i] = 0 then return true
return false
```

- For an existential we translate the loop to ∃ *loop var* . ((*var* in search range) ∧ (test on *var*)). For this example, we get $\exists i.\, 0 \le i < m \wedge b[i]=0$.
- ***Example 17***: Define $SortedUp(b,m,n)$ so that it is true when array b is sorted ≤ on the segment $m..n$. As an example, if $b[0..3]$ is $[1,3,5,2]$, then $SortedUp(b,0,2)$ is true because $1 \le 3$ and $3 \le 5$ but $SortedUp(b,0,3)$ is false because we don't have ($1 \le 3$ and $3 \le 5$ and $5 \le 2$).
  - Another way to describe $SortedUp(b,m,n)$ is that each element in the list $b[m]$, $b[m+1]$, ..., $b[n-2]$, $b[n-1]$ is ≤ the element to its right.
  - Or expanding further, $b[m] \le b[m+1]$, $b[m+1] \le b[m+2]$, ..., $b[n-1] \le b[n]$. We can generalize this to $b[i] \le b[i+1]$ for $i = m, m+1, m+2, ..., n-1$. To get a formal predicate, we need a ∀ over i:
  - $SortedUp(b,m,n) \equiv \forall i.\, m \le i < n \rightarrow b[i] \le b[i+1]$. We can hoist the parts of this that don't depend on the quantified variable i:
    - $SortedUp(b,m,n) \equiv \forall\, m \le i < n.\, b[i] \le b[i+1]$.
  - If we want to make sure that the indexes are legal, instead of $m \le i < n$ we can make sure $0 \le m < n < size(b)$ and write $0 \le m \le i < n < size(b)$
  - Note: Different generalizations of a property can lead us to different predicates.
  - If we generalize

    $b[m] \le b[m+1], b[m+1] \le b[m+2]$, ..., and $b[n-1] \le b[n]$

    to $b[m+j] \le b[m+j+1]$ for $j = 0, 1, ..., n-1-m$

    we get $\forall\, 0 \le j < n-m\ .\, b[m+j] \le b[m+j+1]$ (and $0 \le m \le n < size(b)$).
- ***Example 18***: Let's find a definition for $Extends(b,b')$ so that it's true if b' is an extension of b. I.e., $b[0]=b'[0]$, $b[1]=b'[1]$, ... for all elements of b.
  - Note b' can be the same length as b or can be longer.
  - E.g., if b is $[1,6,2]$ and b' is $[1,6,2,8]$, then $Extends(b,b')$ is true and $Extends(b',b)$ is false.
  - Here's one solution: $Extends(b,b') \equiv size(b) \le size(b') \wedge \forall\, 0 \le k < size(b).\, b[k]=b'[k]$.

Types, Expressions, and Arrays

CS 536: Science of Programming, Spring 2023

ver. Sun 2023-01-15, 18:00

A. Why?

- Expressions represent values relative to a state.
- Types describe common properties of sets of values.
- The value of an array is a function value from index values to array values.

B. Outcomes

At the end of this class, you should

- Know what expressions and their values we'll be using in our language
- Know how states are expanded to include values of arrays

C. Types and Expressions

- Let's start looking at programming language we'll be using.
- The ***datatypes*** will be pretty simple (no records or function types, for example).
  - Primitive types: *int* (integers) and *bool* (boolean). We can add other types like characters, strings, and floating-point numbers, but for what we're doing, integers and booleans are enough.
  - Composite types: Multi-dimensional arrays of primitive types of values, with integer indexes.
- ***Expressions*** are built from
  - ***Constants***: Integers (*0*, *1*, *-1*, ...) and boolean constants (*T*, *F*).
  - ***Simple variables*** of primitive types.
  - ***Operations***
    - On integers: Binary +, -, *, /, *min*, *max*, %, =, ≠, <, ≤, >, ≥, *divides*, Unary –, *sqrt*.
      - / and *sqrt* truncate toward zero, to an integer. E.g., *13 / 3 = 4*, *13 / –3 = –4*, and *sqrt*(*17*) = *4*. Division and mod (%) by zero and *sqrt* of negative values generate runtime errors.
    - On booleans: ¬, ∧, ∨, →, ↔, =, ≠ (note = and ↔ mean the same thing).
    - On arrays: *size* and array element selection.
  - ***Conditional expressions***
    - ***if*** *B* ***then*** e_1 ***else*** e_2 ***fi***. Semantically, if *B* evaluates to true, then evaluate e_1; if *B* evaluates to false, then evaluate e_2. The C / Java syntax (*B ?* e_1 *:* e_2) is also okay.

    - Restrictions: To ensure that the entire conditional expression has a consistent type, e_1 and e_2 must have the same type. (This is sometimes called "balancing".) The type must also be simple (not an array type or function type.
- ***Arrays***
    - As usual, *b[e]* is array element selection. *size*(*b*) gives the length of *b*. For multi-dimensional arrays, we have $b[e_1][e_2]...[e_n]$ and *size1*(*b*), *size2*(*b*), etc. Arrays are zero-origin and fixed-size.
    - You can have array parameters with functions and predicates (as in *size*(*b*)) .
    - ***Restrictions:*** No array assignments, no expressions of type array; this includes array slices ($b[e_1]$ of a two-dimensional array, for example). To support these, we'd need identifiers to map to memory locations, with a separate function mapping locations to values. (This is also why we don't have pointers.)
- ***General restrictions***
    - No expressions with functional or array values. (So they all have primitive types.)
        - ***Example***: ***if*** *B* ***then*** *f(x)* ***else*** *g(x)* ***fi*** is legal; ***if*** *B* ***then*** *f* ***else*** *g* ***fi*** *(x)* is not.
    - We don't have assignment expressions (we'll see later how to simulate them).
    - We don't have records (adding them isn't that hard, but they don't really add much. theoretically speaking).
- We won't explicitly declare variables; we will assume we can infer the types. The default type is integer.

- ***Notation***: *c* and *d* are constants; *e* and *s* are general expressions; *B* and *C* are boolean expressions, *a* and *b* are array names, and *u*, *v*, etc. are variables. Greek letters like α and β stand for semantic values.

D. Examples of Expressions

- ***Example 1***: ***if*** *x < 0* ***then*** *0* ***else*** *sqrt(x)* ***fi*** yields 0 if *i* is negative, otherwise it yields the square root of *x*.
- ***Example 2:*** ***if*** *x < 0* ***then*** *x+y* ***else*** *x*y)+z* ***fi*** means "If *x < 0* evaluates to true, then we evaluate *x+y* and add the result to *z*, otherwise evaluate *x*y* and add the result to *z*." (*x*, *y*, and *z* must all be integers.)
- ***Example 3***: ***if*** *i < 0* ***then*** *b[0]* ***else*** *i* ≥ *size*(*b*) ***then*** *b[size*(*b*)*-1]* ***else*** *b[i]* ***fi*** yields *b[i]* if *i* is in range; if *i* is negative, it yields *b[0]*; if *i* is too large, it yields the last element of *b*.
- ***Example 4***: *b[* ***if*** *i < 0* ***then*** *0* ***else*** *i* ≥ *size*(*b*) ***then*** *size*(*b*)*-1* ***else*** *i* ***fi*** *]* yields the same value as Example 3, but it does this by calculating the index first.
- ***Example 5:*** A (conditional) expression can't yield a function, so ***if*** *B* ***then*** *f(x)* ***else*** *g(x)* ***fi*** is legal; ***if*** *B* ***then*** *f* ***else*** *g* ***fi*** *(x)* is not.
- ***Example 6***: We can't have array-valued expressions, so (assuming *a* and *b* are 1-dimensional arrays), ***if*** *x* ***then*** *a[0]* ***else*** *b[0]*) ***fi*** is legal, ***if*** *x* ***then*** *a* ***else*** *b* ***fi****[0]* is not.

E. Syntactic Values and Semantic Values

- When we discuss the meanings of programs, some of the items are syntactic (like expressions) and some items are semantic (values, states). So there's a problem with symbols like “2” or “+”. Sometimes we use them in our programs; this is a syntactic use. But sometimes we mean a mathematical value, the thing denoted by “2” or “two” or "plus" or so on.
- In general, the context tells you whether something is syntactic or semantic. E.g.,
  - ***Example 7:*** In “Does x occur in the predicate p?” since p is a predicate, it is syntactic, so for x to occur in it, x must be syntactic also.
  - ***Example 8:*** In $z \equiv 2+2$, the $\equiv$ symbol is for syntactic equality, so both z and 2+2 are syntactic.
  - ***Example 9:*** In “$\sigma(2+2) = 2+2 = 4$, the σ is semantic (a state) and the first 2+2 is syntactic, since we're looking for its value in σ. The second 2+2 is semantic because σ takes expressions and returns semantic values. (Hence the second + sign is semantic). The result 4 is also semantic. Also, the two equal signs are semantic equality.
  - ***Example 10:*** "The value in σ of 2+2 is two plus two, which is four" is the same as Example 9 but it uses English to write out the semantic values and operations.

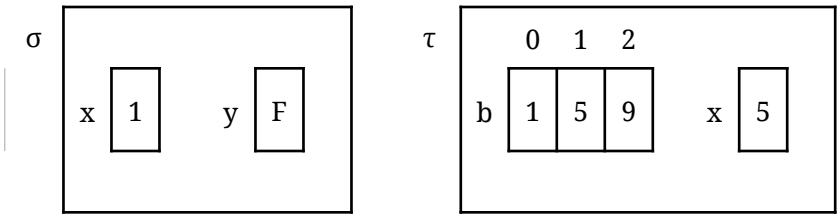
F. Semantic Values and Values of Expressions

- ***Notation:*** In this section, if I really want to emphasize that something is semantic, I'll underline it. So just 2 is syntactic (i.e., the keystroke), but $\underline{2}$ is semantic (i.e., the number in $\mathbb{N}$). The same semantic value often can be described in different ways: $\underline{2}$, $\underline{\text{two}}$, $\underline{1+1}$, and $\underline{\text{one plus one}}$, for example.
- ***Example 11:*** Rewriting Examples 9 and 10: $\sigma(2+2) = \underline{2+2} = \underline{4}$ or: the value in σ of 2+2 is $\underline{\text{two plus two}}$, which is $\underline{\text{four}}$. Technically, the equality tests could be underlined, but $\sigma(2+2) \underline{=} \underline{2+2 = 4}$ really seems like more trouble than it's worth. Furthermore, $\underline{=}$ (underlined equal) looks a lot like $\equiv$ (syntactic equality).
- ***Example 12:*** If $\underline{\sigma}$ is the state that maps x to $\underline{5}$, we could rewrite "$\sigma = \{x = 5\}$” as “$\underline{\sigma} = \{x = \underline{5}\} = \{(x, \underline{5})\}$”.
- In general, expressions have values relative to a state. E.g., relative to $\{x = \underline{1}, y = \underline{2}\}$, the expression $x+y$ has the value $\underline{3}$. Recall that we write $\sigma(x)$ for the value of the variable x and extend this to $\sigma(e)$ for the value of the expression e.
- The value of $\sigma(e)$ depends on what kind of expression e is, so we use recursion on the structure of e (the base cases are variables and constants and we recursively evaluate subexpressions).
  - $\sigma(x)$ = the value that σ binds variable x to
  - $\sigma(c)$ = the value of the constant c. E.g., $\sigma(2) = \underline{2}$. (Note σ is irrelevant here.)
  - $\sigma(e_1 + e_2) = \sigma(e_1)$ plus $\sigma(e_2)$ [and similar for –, *, etc.]
  - $\sigma(e_1 < e_2) = \underline{T}$ iff $\sigma(e_1)$ is less than $\sigma(e_2)$ [similar for $\leq$, =, etc].

- $\sigma(e_1 \wedge e_2) = \underline{T}$ iff $\sigma(e_1)$ and $\sigma(e_2)$ are both = $\underline{T}$ [similar for $\vee$, etc].
- $\sigma($***if*** *B* ***then*** e_1 ***else*** e_2 ***fi***$) = \sigma(e_1)$ if $\sigma(B) = \underline{T}$. It = $\sigma(e_2)$ if $\sigma(B) = \underline{F}$.
- We'll put off the $\sigma(b[e])$ case, the value of the array indexing expression *b[e]*, for just a bit until we look at the value of an array variable.

- ***Example 13***: Let $\sigma = \{x = 1\}$, let $\tau = \sigma \cup \{y = 1\}$, and let $e \equiv (x =$ ***if*** $y > 0$ ***then*** 17 ***else*** y ***fi***$)$.
  - To calculate *e*, first we look up $\tau(x)$ and get $\underline{1}$. (Since τ extends σ with a binding for *y*, τ behaves like σ except on *y*.)
  - Now we need $\tau($ ***if*** $y > 0$ ***then*** 17 ***else*** y ***fi*** $)$.
    - $\tau(y > 0)$ means "Is $\tau(y)$ greater than zero?" Since $\tau(y) = \underline{1}$, the answer is $\underline{T}$.
    - $\tau(y > 0) = \underline{T}$, *so* $\tau($ ***if*** $y > 0$ ***then*** 17 ***else*** y ***fi*** $) = \tau(17)$. I.e., since the test evaluates to $\underline{T}$, the value of the conditional is the value of *17*.
    - $\tau(17) = \underline{17}$, of course.
  - So $\tau($ ***if*** $y > 0$ ***then*** 17 ***else*** y ***fi*** $) = \underline{17}$.
  - For the overall expression, we're comparing $\tau(x)$ and $\tau($ ***if*** $y > 0$ ***then*** 17 ***else*** y ***fi*** $)$ for equality. I.e., we test $\underline{1} = \underline{17}$ and we get $\underline{F}$.
  - So $\tau(e) = \underline{F}$.
- ***The empty state***: Since a state is a set of bindings, the empty set $\varnothing$ is a state (the empty state). It's proper for any expression or predicate that doesn't include variables. E.g., In state $\varnothing$, the expression *2+2* evaluates to four. (In fact, since we don't care about bindings for variables that don't appear in an expression, we can say that in any state σ, *2+2* evaluates to 4.
- ***Example 14***: Let $\sigma = \varnothing$ (the empty state) then
  - $\sigma(2+2 = 4) = \sigma(2+2)$ equals $\sigma(4) = \ldots = \underline{4}$ equals $\underline{4} = \underline{T}$.
- With operators, you have to distinguish the syntactic symbol from the semantic symbol. So $\sigma(v+w) = \sigma(v) \underline{+} \sigma(w)$ is correct: The second plus is the semantic meaning of the syntactic symbol +. You could also write $\sigma(v+w) = \sigma(v)$ *plus* $\sigma(w)$; here, *plus* has a semantic meaning. (If the language under discussion includes an infix binary plus operator, then σ(v plus w) would be legal.)

G. Arrays and Their Values



- Compare the usual way we write states on the blackboard. Below, the left state is $\sigma = \{x = 1, y = F\} = \{(x, 1), (y, F)\}$. The right one, τ, defines an array variable *b* and an integer *x*.

- We'll take the value of an array to be a function from index values to stored values, so τ(*b[0]*) = 3, τ(*b[1]*) = 5, and τ(*b[2]*) = 9. We could write τ = {*b[0]* = 3, *b[1]* = 5, *b[2]* = 9, *x* = 5} = {(*b[0]*, 3), (*b[1]*, 5), (*b[2]*, 9), (*x*, 5)}, but a more convenient notation would be nice.
- ***Notation***: Let β be the function with β(0) = 3, β(1) = 5, β(2) = 9, then we can say τ = {*b* = β, *x* = 5} = {(*b*, β), (*x*, 5)}. (I'm using a greek letter β because the function is semantic, taking index values to memory values.). Since a function is a set of ordered pairs, we can also write β = {(0, 3), (1, 5), (2, 9)}. Since β is actually a sequence, let's allow ourselves to abbreviate this to β = (3, 5, 9). (Note this last notation looks like the graphical picture of τ.)
- We we have a number of ways to express τ, all valid. Going from shortest to longest we have
  - τ = {*b* = β, *x* = 5} where β = (3, 5, 9) — A sequence
  - τ = {*b[0]* = 3, *b[1]* = 5, *b[2]* = 9, *x* = 5} — A set of individual bindings
  - τ = {*b* = β, *x* = 5} where β = {(0, 3), (1, 5), (2, 9)} — A set of ordered pairs
  - τ = {*b* = β, *x* = 5} where β(0) = 3, β(1) = 5, β(2) = 9 — A list of individual bindings

H. Value of An Array Indexing Expression

- Going back to the definition of the value of an expression in a state, here's the array case:
- σ(*b[e]*) = β(α) where β = σ(*b*) and α = σ(*e*). The variable *b* is an array name, so σ(*b*) = a function we're calling β. We call β on the ***value*** of the index expression *e*, hence α = σ(*e*), and the value β(α) is the meaning of *b[e]*.
- You can also write σ(*b[e]*) = (σ(*b*))(σ(*e*)) if you don't want to define α and β. Function application is left-associative, so σ(*b*)(σ(*e*)) = (σ(*b*))(σ(*e*)). I.e., σ(*b*) is a function we're applying to σ(*e*).
- So another way to write the definition is σ(*b[e]*) = σ(*b*)(σ(*e*)) = β(α) where β = σ(*b*) and α = σ(*e*).
- With our earlier example then, σ(*b[x-4]*) = σ(*b*)(σ(*x-4*)) = β (σ(*x*) minus four) = β(5 minus four) = β(1) = 5, where β is as described earlier, β = (3, 5, 9).
- ***Example 15***: Let σ = {*x* = 1, *b* = α} where α = (2, 0, 4). Then
  - σ(*x*) = 1
  - σ(*x+1*) = σ(*x*) + σ(*1*) = 1+1 = 2
  - σ(*b*) = α
  - σ(*b[x+1]*) = (σ(*b*))(σ(*x+1*)) = α(2) = 4
  - If we don't want to write out the intermediate steps first, we could write
    - σ(*b[x+1]*) = (σ(*b*))(σ(*x+1*)) = α(σ(*x*)+1) = α(1+1) = α(2) = 4.
- ***Example 16***: Let σ = {*x* = 1, *b* = α} where α = (2, 0, 4), then
  - σ(*b[x+1]- 2*) = σ(*b[x+1]*) – σ(*2*) = (σ(*b*))(σ(*x+1*)) – 2
    = (σ(*b*))(σ(*x*)+ *1*)) – 2

= α(1+1) – 2

= α(2)–2 = 4 – 2 = 2.

State Updates, Satisfaction of Quantified Predicates

CS 536: Science of Programming, Spring 2023

2023-01-24 pp.4,5

A. Why?

- A predicate is satisfied relative to a state; it is valid if it is satisfied in all states.
- State updates occur when we introduce new variables or change the values of existing variables.

B. Outcomes

At the end of this class, you should

- Know what it means to update a state.
- Know what it means for a quantified predicate to be valid of be satisfied in a state.

C. "Updating" States

- To check quantified predicates for satisfaction, we need to look at different states that are related to, but not identical to, our starting state.
- ***Example 1***: For $\{ y = 1 \} \models \forall x \in \mathbb{Z} . x^2 \geq y - 1$, we need to know that $\{ y = 1, x = \alpha \} \models x^2 \geq y - 1$ for every $\alpha \in \mathbb{Z}$. I.e., we need to know that
    - ....
    - $\{ y = 1, x = -1 \} \models x^2 \geq y - 1$
    - $\{ y = 1, x = 0 \} \models x^2 \geq y - 1$
    - $\{ y = 1, x = 1 \} \models x^2 \geq y - 1$
    - $\{ y = 1, x = 2 \} \models x^2 \geq y - 1$
    - ....
    - Similarly, for $\{ z = 4 \} \models \exists x \in \mathbb{Z} . x \geq z$, we need $\{ z = 4, x = \alpha \} \models x \geq z$ for some particular integer α (α = 5 works nicely).
- There is a complicating factor. If the quantified variable already appears in the state, then we need to ***replace*** its binding with one that gives the value we're interested in checking.
- ***Example 2***: We already know $\{ z = 4 \} \models \exists x \in \mathbb{Z} . x \geq z$ because $\{ z = 4, x = 5 \} \models x \geq z$. If we start with the state $\{ z = 4, x = -15 \}$, which already has a binding for x, we ignore it because at the time we test for satisfaction of $x \geq z$, we're using $z = 4$. In other words, we test for $\{ z = 4, x = 5 \} \models x \geq z$ regardless of whether we started with a value for x or not (i.e., $\{ z = 4, x = -15 \}$ or $\{ z = 4 \}$).

- In $\{ z = 4 , x = 5 \} \models \exists x \in \mathbb{Z} . x \geq z$, the x in $x = 5$ is not the same as the x in $x \geq z$. The problem is that we have two variables, both spelled "x". If we give the x's different names, the difference becomes clear. Let xo be the "outer" x and xi be the "inner" x, then
$$\{ z = 4 , xo = -15 \} \models \exists xi \in \mathbb{Z} . xi \geq z$$
because
$$\{ z = 4 , xo = -15 , xi = 5 \} \models xi \geq z$$
- But there really isn't any use for us to keep xo because there's no way to access it. We would need it more complicated languages where you have code that can take the then-current xo and save it for later use.

- ***Definition***: For any state σ, variable x, and value α, the ***update***[1] ***of*** σ ***at*** x ***with*** α, written $\sigma[x \mapsto \alpha]$, is the state that is a copy of σ except that it binds variable x to value α.
  - Let $\tau = \sigma[x \mapsto \alpha]$, then $\tau(x) = \alpha$; if variable $y \not\equiv x$, then $\tau(y) = \sigma(y)$.
  - Note $\tau(x) = \alpha$ regardless of whether $\sigma(x)$ is defined or not. If $\sigma(x)$ is defined, its type and exact value are irrelevant.
- Set theoretically,
  - If x has no binding in σ, then $\sigma[x \mapsto \alpha]$ is $\sigma \cup \{ x = \alpha \}$: It's like σ but has been extended with $x = \alpha$.
  - If x has a binding in σ, say $\sigma = \{ x = \beta \} \cup \sigma_0$ where σ_0 is the rest of σ, then $\sigma[x \mapsto \alpha]$ is $\sigma_0 \cup \{ x = \alpha \}$. It's like σ but has the binding $x = \alpha$, not $x = \beta$. (Having two bindings for x would be illegal.)
- ***Important***: Calling it the "update" of σ is kind of misleading because we're not modifying σ.
  - Taking $\sigma[x \mapsto \alpha]$ ***does not do*** an update in place; if we define $\tau = \sigma[x \mapsto \alpha]$, then σ is still σ.
  - Conceptually, we aren't modifying σ, we're looking at a state much like it.
  - But "update" is the traditional name, and me personally, I can't find any word that's exactly right. We're not always *extending* σ, we're not always *superseding* σ, ...
- Note though we can give $\sigma[x \mapsto \alpha]$ a new name, it's not required; we just have to write it out explicitly when we use it.
  - If v stands for a variable (not literally the variable v) then if $v \equiv x$, then $\sigma[x \mapsto \alpha](v) = \sigma[x \mapsto \alpha](x) = \alpha$, otherwise (if $x \not\equiv v$), then $\sigma[x \mapsto \alpha](v) = \sigma(v)$.
  - (You have to read $\sigma[x \mapsto \alpha](v)$ left-to-right — we're taking the function $\sigma[x \mapsto \alpha]$ and applying it to v. I.e., $\sigma[x \mapsto \alpha](v) = (\sigma[x \mapsto \alpha])(v)$, where the left pair of parentheses are for grouping and the ones around v are for the function call.)
- ***Example 3***: If $\sigma = \{ x = 2 , y = 6 \}$, then $\sigma[x \mapsto 0] = \{ x = 0 , y = 6 \}$, so

---

[1] Unfortunately, "update" is the traditional name, and for myself, I can't find any word that's exactly right. We're not always *extending* σ, we're not always *superseding* σ, ...

- $\sigma[x \mapsto 0](x) = 0$ (Even though $\sigma(x) = 2$)
- $\sigma[x \mapsto 0](y) = \sigma(y) = 6$ (Since we didn't update y)
- $\sigma[x \mapsto 0](x + y) = 0 + 6 = 6$ (Since the x in $x + y$ gets evaluated to 0)
- $\sigma[x \mapsto 0] \models x^2 \leq 0$ (Even though our starting $\sigma \not\models x^2 \leq 0$)

- The value part of an update has to be a semantic value, not a syntactic one, so if you wanted to add one to x, you can't use "$\sigma[x \mapsto x + 1]$" because it isn't well-formed (the x on the left side of $\mapsto$ must be syntactic, the x on the right side of $\mapsto$ has to be semantic, and the conflict can only be resolved by making one of the x's something else..
  - On the other hand, "$\sigma[x \mapsto \sigma(x + 1)]$" or "$\sigma[x \mapsto \alpha + 1]$ where $\alpha = \sigma(x)$" do make sense.

Multiple Updates

- We can do a sequence of updates on a state. E.g., $\sigma[x \mapsto 0][y \mapsto 8]$ is a doubly updated state. Sequences of updates are read left-to-right, so this is $(\sigma[x \mapsto 0])[y \mapsto 8]$.
- ***Example 4***: If $\sigma = \{x = 2, y = 6\}$, then $\sigma[x \mapsto 0][y \mapsto 8] = \{x = 0, y = 6\}[y \mapsto 8] = \{x = 0, y = 8\}$.
- ***Example 5***: $\sigma[x \mapsto 0][y \mapsto 8] = \sigma[y \mapsto 8][x \mapsto 0]$ because he order of update doesn't matter if you have two different variables.
- ***Example 6***: $\sigma[x \mapsto 0][x \mapsto 17] = \sigma[x \mapsto 17] \neq \sigma[x \mapsto 17][x \mapsto 0] = \sigma[x \mapsto 0]$: If you update the same variable twice, the second update supersedes the first.
- Of course, if the second update is identical to the first, nothing happens: $\sigma[x \mapsto \alpha][x \mapsto \alpha] = \sigma[x \mapsto \alpha]$
- If you have to evaluate an expression, be sure to do it in the correct state.
  - Let $\sigma(x) = 1$ and let $\tau = \sigma[x \mapsto 2]$, then $\tau[z \mapsto \sigma(x) + 10]$ maps z to $\sigma(x) + 10 = 1 + 10 = 11$. We can omit τ and also write $\sigma[x \mapsto 2][z \mapsto \sigma(x) + 10]$, which gives the same state as τ.
  - On the other hand, look at $\tau[z \mapsto \tau(x) + 10]$. Since $\tau = \sigma[x \mapsto 2]$, the value of $\tau(x) + 10 = 12$, so $\tau[z \mapsto \tau(x) + 10] = \tau[z \mapsto 12]$.
  - If we hadn't given the name $\tau = \sigma[x \mapsto 2]$, then we would had to write $\sigma[x \mapsto 2][z \mapsto \sigma[x \mapsto 2](x) + 10]$. This is pretty ugly, so giving $\sigma[x \mapsto 2]$ a name like τ makes things more readable.

D. Updating Array Values

- Updating array elements like $b[0]$ is a bit more complicated than updating simple variables like x and y. First, let's extend our notion of updating states to updating general functions.
- ***Definition***: If δ is a function on one argument and α and β are valid members of the domain and range of δ respectively, then the ***update of*** δ ***at*** α ***with*** β, written $\delta[\alpha \mapsto \beta]$, is the function defined by $\delta[\alpha \mapsto \beta](\gamma) = \beta$ if $\gamma = \alpha$ and $\delta[\alpha \mapsto \beta](\gamma) = \delta(\gamma)$ if $\gamma \neq \alpha$. The name α should be a semantic constant (like 0 or zero).

- [2023-01-24] ***Definition***: Say σ is a (proper) state for an array b, with η = the function $\sigma(b)$. If α is a valid index value for b, then $\sigma[b[\alpha]\mapsto\beta]$ means $\sigma[b\mapsto\eta[\alpha\mapsto\beta]]$. So updating σ at $b[\alpha]$ with β involves updating σ with an updated version of η, namely $\eta[\alpha\mapsto\beta]$, as the value of b.
- ***Example 7***: Say $\sigma=\{x=3,\ b=(2,4,6)\}$, then $\sigma[b[0]\mapsto 8]=\{x=3, b=(8,4,6)\}$. Here, $\sigma(b)$ is $(2,4,6)$ as a function (which can also be written $\{(0,2),(1,4),(2,6)\}$, so $\sigma(b)[0\mapsto 8]$ (the update of function $\sigma(b)$) is the function $(2,4,6)[0\mapsto 8]=(8,4,6)$.
- The notation $\sigma[b[\alpha]\mapsto\beta]$ is a bit of a hack: The name b is syntactic but α is semantic. The restriction that α be a constant like 0 or zero avoids the complications that result if you allow α to be the name for a complicated semantic expression like $\tau(e)$. The intuition is that α models the memory offset from $b[0]$ that we need to find in order to do the update.

E. Satisfaction of Quantified Predicates

- One use of updated states is for describing how assignment works. (We'll see this later.) The other use for updated states is for defining when quantified predicates are satisfied.
- ***Definition***: $\sigma\models\exists x\in S.\,p$ if for one or more ***witness*** values $\alpha\in S$, it's the case that $\sigma[x\mapsto\alpha]\models p$. Note we're asking a hypothetical question: "If we were to calculate $\sigma[x\mapsto\alpha]$, would we find that it satisfies p?"
  - ***Example 8a***: For any state σ, we can show $\sigma\models\exists x.\,x^2\le 0$ using 0 as the witness: $\sigma[x\mapsto 0]\models x^2\le 0$, since $\sigma[x\mapsto 0](x^2\le 0)=\sigma[x\mapsto 0](x^2)\le\sigma[x\mapsto 0](0)=(0^2\le 0)=T$.
- Remember, $\sigma(x)$ is irrelevant, since $\sigma[x\mapsto\alpha]$ overrides any value for $\sigma(x)$.
  - ***Example 8b***: If $\sigma(x)$ is, say 5, it's still the case that $\sigma\models\exists x.\,x^2\le 0$ using 0 as the witness because we $\sigma[x\mapsto 0]\models x^2\le 0$, regardless of $\sigma(x)=5$.
- If there are many successful witness values, we don't have to specify all of them; we just need one.
  - ***Example 9***: If $\sigma(y)=3$, then $\sigma\models\exists x.\,x^2\le y$ with $x=0$ or 1 (or -1) as possible witness values.
- ***Definition***: $\sigma\models\forall x\in S.p$ if for every value $\alpha\in S$, we have $\sigma[x\mapsto\alpha]\models p$. (Again, this is hypothetical: "If for every α, we were to calculate $\sigma[x\mapsto\alpha]$, would we find that it satisfies p?"
  - ***Example 10***: To know $\sigma\models\forall x\in\mathbb{Z}.\,x^2\ge x$, we need to know $\sigma[x\mapsto\alpha]\models x^2\ge x$ for every $\alpha\in\mathbb{Z}$. Since for every integer α, indeed α^2 is $\ge\alpha$, this does hold. Recall that it doesn't matter what $\sigma(x)$ is, since we're interested in $\sigma[x\mapsto\alpha]$.
- When asking if σ satisfies $\forall x\in S.\,q$ or $\exists x\in S.\,q$, we don't care about $\sigma(x)$. For a predicate p in general, for the question "Does $\sigma\models p$?" only depends on how σ operates on the non-quantified variables of p.
  - ***Example 11***: Since the body of $\forall x\in\mathbb{Z}.\,x^2\ge x$ uses only the quantified variable x, it doesn't matter what bindings σ has when checking $\sigma\models\forall x\in\mathbb{Z}.\,x^2\ge x$. Even $\sigma=\varnothing$ works: $\varnothing\models\forall x\in\mathbb{Z}.\,x^2\ge x$.
- Note with nested quantifiers, the notation does get more complicated.

- ***Example 12***: Intuitively, $\sigma \models \forall x.(x > y^2 \rightarrow (\exists z.(z > y^4)))$ means
  For every $\alpha \in \mathbb{Z}$, if $\alpha > \sigma(y)^2$, then there is some $\beta \in \mathbb{Z}$ such that $\beta > \sigma(y)^4$.
  We can justify this observation by going through the definitions.

  $\sigma \models \forall x.\, x > y^2 \rightarrow \exists z.\, z > y^4$

  iff for every $\alpha \in \mathbb{Z}$, $\sigma[x \mapsto \alpha] \models x > y^2 \rightarrow \exists z.\, z > y^4$ defn $\models \forall$

  iff for every $\alpha \in \mathbb{Z}$, if $\sigma[x \mapsto \alpha] \models x > y^2$, then $\sigma[x \mapsto \alpha] \models \exists z.\, z > y^4$ defn $\rightarrow$

  iff for every $\alpha \in \mathbb{Z}$, if $\alpha > \sigma(y)^2$, then $\sigma[x \mapsto \alpha] \models \exists z.\, z > y^4$ defn ... $\models x > y^2$

  iff for every $\alpha \in \mathbb{Z}$, if $\alpha > \sigma(y)^2$, then there is some $\beta \in \mathbb{Z}$ such that $\sigma[x \mapsto \alpha][z \mapsto \beta] \models z > y^4$
  [2023-01-23] defn $\models \exists$

  iff for every $\alpha \in \mathbb{Z}$, if $\alpha > \sigma(y)^2$, then there is some $\beta \in \mathbb{Z}$ such that $\beta > \sigma(y)^4$
  defn ...$\models z > y^4$

  - Note defining intermediate names like "let $\tau = \sigma[x \mapsto \alpha][z \mapsto \beta]$" is allowed, if you wish.

Justifying DeMorgan's Laws for Quantified Predicates

- In general, we want our systems of reasoning to be ***sound***: We want the textual transformations that make up logical equivalence to reflect truths about how our semantics work.
- ***Example 15***: Here is a check of DeMorgan's law for existentials, which says $\neg \exists x.\, p \Leftrightarrow \forall x.\, \neg p$. Semantically, we want each of these to be valid if and only if the other is. So we need $\sigma \models \neg \exists x.\, p$ if and only if $\sigma \models \forall x.\, \neg p$.

  $\sigma \models \neg \exists x \in S.\, p$

  iff $\sigma \not\models \exists x.\, p$ defn of $\sigma \models \neg$

  iff not (there is an $\alpha \in S$ such that $\sigma[x \mapsto \alpha] \models p$) defn of $\sigma \models \exists$

  iff for no $\alpha \in S$ does $\sigma[x \mapsto \alpha] \models p$ (rephrasing)

  iff for every $\alpha \in S$ we have $\sigma[x \mapsto \alpha] \not\models p$ equiv. of "no $\models$" vs "every $\not\models$"

  iff for every $\alpha \in S$ we have $\sigma[x \mapsto \alpha] \models \neg p$ defn of $\sigma \models \neg$ predicate

  iff $\sigma \models \forall x.\, \neg p$ defn of $\sigma \models \forall$

- Showing the semantic property that $\models \neg \exists x.\, p \leftrightarrow \forall x.\, \neg p$ gives us a justification for adding $\neg \exists x.\, p \Leftrightarrow \forall x.\, \neg p$ as a proof rule.

- Quick review of terms:
  - ***Validity***: $\models p$ means $\sigma \models p$ for all σ. I.e., "p is valid". Example: $\models x + 1 > x$
  - ***Not valid*** $\not\models p$ means for some σ, $\sigma \not\models p$. (σ is the counterexample state.)
  - ***Example***: $\not\models x^2 > 0$ iff for some σ, $\sigma \not\models x^2 > 0$ iff for some σ, $\neg(\sigma(x)^2 > 0)$ iff for some σ, $\sigma(x)^2 \leq 0$. If $\sigma(x) = 0$, then σ satisfies this requirement. ([2023-01-24] $\sigma[x \mapsto 0]$ is a counterexample state.)
  - ***Example***: $\not\models \exists y.\, x < 0 \vee x^2 < y < (x+1)^2$

iff for some σ, $\sigma \not\models \exists y.\ x > 0 \wedge x^2 < y < (x+1)^2$

iff for some σ, for every α, $\sigma[y \mapsto \alpha] \not\models\ x < 0 \vee x^2 < y < (x+1)^2$.

iff for some σ, for every α, $\sigma[y \mapsto \alpha] \models\ x \geq 0 \wedge \neg(x^2 < y < (x+1)^2)$ [using DeMorgan's]

If we let $\beta = \sigma(x)$, then the line above holds

iff for some β, for every α, $\beta \geq 0 \wedge \neg(\beta^2 < \alpha < (\beta+1)^2)$.

Using $\beta = 0$ satisfies this requirement.

Program Syntax; Operational Semantics

CS 536: Science of Programming, Spring 2023

2023-01-25 pp.3,5,6,7

A. Why

- Our simple programming language is a model for the kind of constructs seen in actual languages.
- Step–by–step program evaluation involves a sequence of program / state snapshots.

B. Outcomes

At the end of today, you should be able to

- Describe the syntax of our simple deterministic programming language.
- Translate programs in our language to and from C / C++ / Java.
- Use operational semantics to describe step–by–step execution of programs in our language.
- ***Note – Now that you're more used to the difference between syntactic and semantic objects, I won't be as strict about using italics to indicate syntactic items.*** For example, writing $\sigma(x+3) = \sigma(x)+3 = 6+3 = 9$ instead of $\underline{\sigma(x+3)} = \underline{\sigma(x) + 3} = \underline{6+3} = \underline{9}$ (assuming $\underline{\sigma(x)} = \underline{6}$).

C. Our Simple Programming Language

- As mentioned before, we're going to use the simplest programming language we can get away with. This is because having fewer language constructs makes it easier to analyze how a language works E.g., we've omitted declarations because we don't need them when studying the semantics of our programs' execution.
- ***Notation:*** We'll typically use e, e' for expressions, B for boolean expressions, S for statements.
- Our initial programming language has five kinds of statements. (We'll add more as we go along.)
- ***No–op*** statement
  - The no–op statement does nothing. Syntax: ***skip***
- ***Assignment*** statement
  - ***Syntax***: $v := e$ or $b[e_1][\ldots][e_n] := e$
  - For an n–dimensional array, all n indexes must appear. (We can't slice arrays as in C.)
- ***Sequence*** statement
  - ***Syntax***: $S ; S'$ (semicolon is a separator)
  - S' can be a sequence, so you can get longer sequences like $S_1 ; S_2 ; S_3$. Longer sequences get read left–to–right.

- ***Conditional*** statement [1]
  - ***Syntax 1***: ***if*** B ***then*** S_1 ***else*** S_2 ***fi***
  - ***Syntax 2***: ***if*** B ***then*** S_1 ***fi*** is an abbreviation for ***if*** B ***then*** S_1 ***else skip fi***.
  - Since we can simulate an ***if–then*** statement (without an ***else*** clause) with ***if … else skip***, we don't need to define a separate semantics for an ***if–then*** statement, which reduces the amount of analysis work we need to do.
- ***Iterative*** statement
  - ***Syntax***: ***while*** B ***do*** S_1 ***od***
  - Similarly to how we defined an ***if–then*** statement to be a particular kind of ***if–else*** statement, we can omit ***for*** loop and ***do–while*** loop statements, because we can simulate them using ***while*** loops. For example,

    for $x := e_1$ ***to*** e_2 ***do*** S

    turns into

    $x := e_1$; ***while*** $x \le e_2$ ***do*** S ; $x := x + 1$ ***od***.
- ***Program***: A program is simply a statement, typically a sequence statement.

Example 1: A Sample Program

- The program below calculates powers of 2. We run it with a value of n and it returns $y = 2^n$, unless $n < 0$, in which case it returns $y = 1$. Formally, on termination, the program establishes $(n \ge 0 \to y = 2^n) \wedge (n < 0 \to y = 1)$. (This is only one way to write the program, of course.)

```
if n < 0 then
    y := 1
else
    x := 0 ;
    y := 1 ;
    while x < n
    do
        x := x + 1 ;
        y := y + y
    od
fi
```

- When we discuss the semantics and correctness of a program, we'll have to look at not only a whole program statement but also all its embedded sub–statements. So the statement, its sub–

[1] (Meant to say write this down with ***if…fi*** for conditional expressions.) The ***fi*** and ***od*** keywords are from Algol 68, which created them to solve the dangling else problem. (in `C` terms, take `if (B1) S1 else if (B2) S2;` — is `S2` the else clause for `if B1` or `if B2`? Nowadays people often use ***end if*** and ***end do*** instead.

statements, its sub–sub–statements, etc.) The program in Example 1 contains 10 embedded sub–statements. In no particular order:

1. $y := 1$ *(the first one; it sets the result if $n < 0$)*
2. $x := 0$ *(part of loop initialization)*
3. $y := 1$ *(part of loop initialization)*
4. $x := 0 ; y := 1$ *(loop initialization)*
5. $x := x + 1$ *(part of loop the body)*
6. $y := y + y$ *(part of loop the body)*
7. $x := x + 1 ; y := y + y$ *(the loop body; let's call this statement S)*
8. ***while*** $x < n$ ***do*** S ***od***
9. $x := 0 ;$ $y := 1 ;$ ***while*** $x < n$ ***do*** S ***od*** *(let's call this statement W)*
10. ***if*** $n < 0$ ***then*** $y := 1$ ***else*** W ***fi***

D. Relationship to Actual Languages

- Converting between a typical language like C or C++ or Java and our programming language is pretty straightforward. The only real issue is that since our language doesn't include assignment expressions, they have to be rewritten as assignment statements.
- In C, C++, and Java,
  - As statements, `z++` and `++z` are identical.
  - As an expression, the value of `++z` is the value of `z` *after* doing `z=z+1`.
  - As an expression, the value of `z++` is the value of `z` *before* doing `z=z+1`. I.e., first do `temp=z; z=z+1;` and then yield the value of `temp` as the value of the `z++` expression.

Example 2

- The C statement: `x=a*++z;` is equivalent to $z := z + 1 ;$ $x := a * z$

Example 3

- The C statement: `x=a*z++;` is equivalent to $temp := z ;$ $z := z + 1 ;$ $x := a * temp$. Another equivalent piece of code is : $x := a * z ;$ $z := z + 1$

Example 4

[2023-01-25]

- The C loop statement: `while (--x >= n) z*=x;` is equivalent to our
$$x := x - 1 ;\ \textbf{\textit{while}}\ x \geq n\ \textbf{\textit{do}}\ z := z * x ; x := x - 1\ \textbf{\textit{od}}$$
- The decrement of x before the loop is for the first execution of $--x >= n$ (it has to be done before the ***while*** test). The decrement of x at the end of the loop body is for all the other executions of `--x >= n`, where we jump to the top of the loop, decrement x, and test vs. n.

Example 5

- The C loop: `while (x-- > 0) z*=x;`
- Our equivalent: ***while*** $x > 0$ ***do*** $x := x - 1$; $z := z * x$ ***od***; $x := x - 1$
- The decrement of x after the ***do*** is for all the `x-- >0` tests that evaluate to true.
- The decrement of x after the ***od*** is for the last `x-- >0` test, which evaluates to false.

Example 6

- The C program below calculates `p=n!`, if `n≥0`.
  - `p=1; for(x=2; x<n; ++x) p=p*x;`
- In C, with a *for* loop, the increment / decrement clause gets done at the end of the loop body, as a statement, before jumping up to do the test, so the program is equivalent to
  - `p=1; x=2; while(x<n) {p=p*x; ++x;}`
- In our language, an equivalent program is
  - $p := 1$; $x := 2$; ***while*** $x < n$ ***do*** $p := p * x$; $x := x + 1$ ***od***
- (There are certainly other ways to translate the `C for` loop.)

Notes on Translations

- There can be more than one possible translation, especially for complicated programs. The important point is that translation is not difficult and doesn't result in grossly different code.
- For the loop translation examples above, if you really have trouble believing the given equivalences, the easiest way to convince yourself that they work is to write them up as C programs and run them. Either trace their execution or toss in a bunch of print statements to show you how the variables change.

E. Operational Semantics of Programs

- To model how our programs work, we'll start with an ***operational*** semantics: We'll model execution as a sequence of "configurations" — snapshots of the program and memory state — over time. The semantics rules describe how step–by–step execution of the program changes memory. When the program is complete, we have the final memory state of execution.
- ***Definition***: A ***configuration*** $\langle S, \sigma \rangle$ is an ordered pair of a program and state.
  - We use angle brackets instead of parentheses just to make it clear that we're talking about a configuration, not an arbitrary pair.
- ***Definition***: The ***operational semantics*** of programs is given by a relation on configurations: $\langle S, \sigma \rangle \rightarrow \langle S_1, \sigma_1 \rangle$ means that ***executing*** S in state σ ***for one step*** yields $\langle S_1, \sigma_1 \rangle$. S_1 is the ***continuation*** of S. The formal definition of $\rightarrow$ will be given in the next section, but we can present some intuitive examples.

- ***Example 1***: $\langle x := x + 1 ; y := x , \{ x = 5 \} \rangle \rightarrow \langle y := x , \{ x = 6 \} \rangle$ because execution of the first assignment changes the value of x and leaves us with one more assignment to execute. (The formal definition of $\langle x := e , \sigma \rangle \rightarrow \ldots$ will be given in the next section.)
- ***Example 2***:
  - $\langle x := x + 1 ; y := x ; z := y + 3 , \{ x = 5 \} \rangle \rightarrow \langle y := x ; z := y + 3 , \{ x = 6 \} \rangle$,
  - $\langle y := x ; z := y + 3 , \{ x = 6 \} \rangle \rightarrow \langle z := y + 3 , \{ x = 6 , y = 6 \} \rangle$, and
  - $\langle z := y + 3 , \{ x = 6 , y = 6 \} \rangle \rightarrow \langle E, \{ x = 6 , y = 6 , z = 9 \} \rangle$.
- ***Notation***: In the line above, the symbol E stands for the empty program. We use it to indicate a program that's finished execution because $\langle E , \tau \rangle$ is more readable than $\langle \ , \tau \rangle$.
- ***Notation***: We can compress Example 2 by writing it as a chain of $\rightarrow$ steps:

$$\begin{aligned} &\langle x := x + 1 ; y := x ; z := y + 3 , \{ x = 5 \} \rangle \\ &\quad \rightarrow \langle y := x ; z := y + 3 , \{ x = 6 \} \rangle \\ &\quad \rightarrow \langle z := y + 3 , \{ x = 6 , y = 6 \} \rangle \\ &\quad \rightarrow \langle E , \{ x = 6 , y = 6 , z = 9 \} \rangle \end{aligned}$$

- ***Definition***: ***Execution of*** S ***starting in state*** σ ***converges to state*** τ if there is a sequence of executions steps $\langle S , \sigma \rangle \rightarrow \ldots \rightarrow \langle E , \tau \rangle$. If we're not interested in τ, we can abbreviate this to say that execution of S ***converges***. Equivalent phrasings are "σ ***terminates in state*** τ" and "σ ***terminates***".
- ***Definition***: The opposite of convergence is ***divergence***. For us, that's infinite loops or an infinitely long sequence of calculations. E.g., executions of $\langle$ ***while*** $x = 0$ ***do*** $x := 0$ ***od***, $\{ x = 0 \} \rangle$ and $\langle$ ***while*** $x \geq 0$ ***do*** $x := x + 1$ ***od***, $\{ x = 0 \} \rangle$ diverge. More generally in languages one can also have infinite recursion.

F. Operational Semantics Rules

- There is an operational semantics rule for each kind of statement. The ***skip*** and assignment statements complete in one step; the sequence and conditional statements require multiple steps; the iterative statement may complete in any number of steps or loop forever.

Skip Statement

- The ***skip*** statement completes execution and does nothing to the state.
  - $\langle$ ***skip***, $\sigma \rangle \rightarrow \langle E , \sigma \rangle$

Simple Assignment Statement

- To execute $v := e$ in state σ, update σ so that the value of v is the value of e in σ.
  - $\langle v := e , \sigma \rangle \rightarrow \langle E , \sigma [v \mapsto \sigma (e)] \rangle$.
- ***Example 3***: $\langle x := x + 1 , \sigma \rangle$ [2023-01-25] $\rightarrow \langle E , \sigma [x \mapsto \sigma (x + 1)] \rangle = \langle E , \sigma [x \mapsto \sigma (x) + 1] \rangle$.

- ***Example 4***: $\langle x := 2 * x * x + 5 * x + 6, \sigma \rangle$ [2023-01-25] $\rightarrow \langle E, \sigma[x \mapsto \alpha] \rangle$ where $\alpha = \sigma(2 * x * x + 5 * x + 6) = 2\beta^2 + 5\beta + 6$ where $\beta = \sigma(x)$. For complicated expressions, it can be helpful to introduce new symbols, like β, but they aren't required: we can also write $2\sigma(x)^2 + 5\sigma(x) + 6$ here.

Array Element Assignment Statements

- To execute $b[e_1] := e$ in σ, evaluate the index e_1 to get some value α; update the function denoted by b at index α with the value of e.
  - $\langle b[e_1] := e, \sigma \rangle \rightarrow \langle E, \sigma[b[\alpha] \mapsto \beta] \rangle$ where $\alpha = \sigma(e_1)$ and $\beta = \sigma(e)$. [2023-01-25]
- ***Example 5***: If $\sigma(x) = 8$, then

  $\langle b[x + 1] := x * 5, \sigma \rangle$
  [2023-01-25] $\rightarrow \langle E, \sigma[b[\beta] \mapsto \delta] \rangle$ where $\beta = \sigma(x + 1) = \sigma(x) + 1 = 8 + 1 = 9$
  $= \langle E, \sigma[b[9] \mapsto 40] \rangle$ and $\delta = \sigma(x * 5) = \sigma(x) * 5 = 8 * 5 = 40$
- The multi–dimensional versions of array assignment are similar; we'll omit them.

Conditional Statements

- For an ***if–then–else*** statement, our one step is to evaluate the test and jump to the beginning of the appropriate branch.
  - If $\sigma(B) = T$ then $\langle \textbf{\textit{if}}\ B\ \textbf{\textit{then}}\ S_1\ \textbf{\textit{else}}\ S_2\ \textbf{\textit{fi}}, \sigma \rangle \rightarrow \langle S_1, \sigma \rangle$
  - If $\sigma(B) = F$ then $\langle \textbf{\textit{if}}\ B\ \textbf{\textit{then}}\ S_1\ \textbf{\textit{else}}\ S_2\ \textbf{\textit{fi}}, \sigma \rangle \rightarrow \langle S_2, \sigma \rangle$
- For an ***if–then*** statement, the missing ***else*** clause defaults to ***else skip***, so our continuation is never empty. (Though, granted, the execution of $\langle \textbf{\textit{skip}}, \sigma \rangle$ is pretty trivial.)
  - If $\sigma(B) = T$ then $\langle \textbf{\textit{if}}\ B\ \textbf{\textit{then}}\ S_1\ \textbf{\textit{fi}}, \sigma \rangle \rightarrow \langle S_1, \sigma \rangle$
  - If $\sigma(B) = F$ then $\langle \textbf{\textit{if}}\ B\ \textbf{\textit{then}}\ S_1\ \textbf{\textit{fi}}, \sigma \rangle = \langle \textbf{\textit{if}}\ B\ \textbf{\textit{then}}\ S_1\ \textbf{\textit{else skip fi}}, \sigma \rangle \rightarrow \langle \textbf{\textit{skip}}, \sigma \rangle$
- ***Example 6***: Let $S \equiv$ ***if*** $x > 0$ ***then*** $y := 0$ ***fi***; we'll evaluate in a state where x is 5.
  - Then $\langle S, \sigma[x \mapsto 5] \rangle \rightarrow \langle y := 0, \sigma[x \mapsto 5] \rangle \rightarrow \langle E, \sigma[x \mapsto 5][y \mapsto 0] \rangle$.
    - (The first arrow is for the ***if–then***; the second is for the assignment to y.)
  - Similarly, $\langle S, \sigma[x \mapsto -1] \rangle \rightarrow \langle \textbf{\textit{skip}}, \sigma[x \mapsto -1] \rangle \rightarrow \langle E, \sigma[x \mapsto -1] \rangle$.
    - (The first arrow is for the ***if–else***; the second is for the ***skip***.)

Iterative Statements

- For a ***while*** statement, our one step is to evaluate the test and jump either to the end of the loop or to the beginning of the loop body. (After the body we'll continue by jumping to the top of the loop.) Let $W \equiv$ ***while*** B ***do*** S ***od***; then
  - If $\sigma(B) = F$, then $\langle W, \sigma \rangle \rightarrow \langle E, \sigma \rangle$
  - If $\sigma(B) = T$, then $\langle W, \sigma \rangle \rightarrow \langle S; W, \sigma \rangle$
  - This last case is the only operational semantics rule that produces a continuation that's textually larger than the starting statement (which is why we can get infinite loops).

- We'll look at a detailed example of a ***while*** loop in a bit.

Sequence Statements

- To execute a sequence $S_1 ; S_2$, for one step, we execute S_1 for one step. This one step may or may not complete all of S_1. If it does, then we continue by executing S_2. If one step of execution takes S_1 to some statement U_1, then we continue by executing $U_1 ; S_2$.
  - If $\langle S_1, \sigma \rangle \to \langle E, \sigma_1 \rangle$ then $\langle S_1 ; S_2, \sigma \rangle \to \langle S_2, \sigma_1 \rangle$
  - If $\langle S_1, \sigma \rangle \to \langle U_1, \sigma_1 \rangle$ then $\langle S_1 ; S_2, \sigma \rangle \to \langle U_1 ; S_2, \sigma_1 \rangle$
- The rule for executing sequences is somewhat different from the other execution rules because it is a ***rule of inference***: We need a recursive call to the definition of "$\to$". Here, we need to know how $\langle S_1, \sigma \rangle$ executes before we can say how $\langle S_1 ; S_2, \sigma \rangle$ executes.
  - All the other execution rules are ***axioms***; they do not need a recursive use of the "$\to$" relation.
- ***Example 7***: Since $\langle x := y, \sigma \rangle \to \langle E, \sigma[x \mapsto \sigma(y)] \rangle$, we know
  - $\langle x := y ; y := 2, \sigma \rangle \to \langle y := 2, \sigma[x \mapsto \sigma(y)] \rangle$ and
  - $\langle y := 2, \sigma[x \mapsto \sigma(y)] \rangle \to \langle E, \sigma[x \mapsto \sigma(y)][y \mapsto 2] \rangle$
  - Say the two statements are the true branch of an ***if*** $x > y \ldots$ ***fi*** where $\sigma(x > y) = T$, then
    - $\langle$ ***if*** $x > y$ ***then*** $x := y ; y := 2$ ***else*** $z := 3$ ***fi***, $\sigma \rangle \to \langle x := y ; y := 2, \sigma \rangle$
      $\to \langle y := 2, \sigma[x \mapsto \sigma(y)] \rangle \to \langle E, \sigma[x \mapsto \sigma(y)][y \mapsto 2] \rangle$
  - On the other hand, if $\sigma(x > y) = F$, then we get [2023-01-25]
    - $\langle$ ***if*** $x > y$ ***then*** $x := y ; y := 2$ ***else*** $z := 3$ ***fi***, $\sigma \rangle \to \langle z := 3, \sigma \rangle \to \langle E, \sigma[z \mapsto 3] \rangle$.

G. Examples of Loops

- Let's look at a couple of examples of loop execution to see how their semantics can be worked out. As part of that, we'll find that it can involve writing an awfully large amount of formal manipulation using the notation. We'll do something about that in the following section.
- ***Example 8***: Let $W \equiv$ ***while*** $x \geq 0$ ***do*** $x := x - 1$ ***od***, then (in maximal detail)

$\langle W, \sigma[x \mapsto 1] \rangle$

$\to \langle x := x - 1 ; W, \sigma[x \mapsto 1] \rangle$	// *Since* $\sigma[x \mapsto 1](x \geq 0) = T$
$\to \langle W, \sigma[x \mapsto 0] \rangle$	// *Since* $\sigma[x \mapsto 1][x \mapsto 0] = \sigma[x \mapsto 0]$
$\to \langle x := x - 1 ; W, \sigma[x \mapsto 0] \rangle$	// *Since* $\sigma[x \mapsto 0](x \geq 0) = T$
$\to \langle W, \sigma[x \mapsto -1] \rangle$	// *Evaluate assignment of* x
$\to \langle E, \sigma[x \mapsto -1] \rangle$	// *Since* $\sigma[x \mapsto -1](x \geq 0) = F$

- ***Example 9***: Let $S \equiv s := 0 ; k := 0 ; W$, where $W \equiv$ ***while*** $k < n$ ***do*** S_1 ***od*** and $S_1 \equiv s := s + k + 1 ; \ k := k + 1$. Let $\sigma(n) = 2$, then (in lots of detail)

$\langle S, \sigma \rangle = \langle s := 0 ; k := 0 ; W, \sigma \rangle$

$\to \langle k := 0 ; W, \sigma[s \mapsto 0] \rangle$	//*Loop initialization*
$\to \langle W, \sigma_0 \rangle$	// *Where* $\sigma_0 = \sigma[s \mapsto 0][k \mapsto 0]$

$\rightarrow \langle S_1 ; W, \sigma_0 \rangle$ // *Since* $\sigma_0(k < n) = T$
$= \langle s := s + k + 1 ;\ k := k + 1 ; W, \sigma_0 \rangle$ // *By defn of* S_1*; note this step is "=", not "*$\rightarrow$*"*
$\rightarrow \langle k := k + 1 ;\ W, \sigma_0[s \mapsto 1] \rangle$ // *Update* s *to* $\sigma_0(s+k+1) = \sigma_0(s) + \sigma_0(k)$
$\rightarrow \langle W, \sigma_1 \rangle$ // *Let* $\sigma_1 = \sigma_0[s \mapsto 1][k \mapsto 1]$ *and evaluate asgt of* k
$\rightarrow \langle S_1 ; W, \sigma_1 \rangle$ // *Since* $\sigma_1(k < n) = T$
$= \langle s := s + k + 1 ;\ k := k + 1 ; W, \sigma_1 \rangle$ // *By defn of* S_1*; note this step is "=", not "*$\rightarrow$*"*
$\rightarrow \langle k := k + 1 ; W, \sigma_1[s \mapsto 3] \rangle$ // *Update* s *to* $\sigma_1(s+k+1) = \sigma_1(s) + \sigma_1(k)$
$\rightarrow \langle W, \sigma_2 \rangle$ // *Where* $\sigma_2 = \sigma_1[s \mapsto 3][k \mapsto 2]$
$\rightarrow \langle E, \sigma_2 \rangle$ // *Since* $\sigma_2(k < n) = F$

H. Using Multi–Step Execution to Abbreviate Executions

- With long executions, we often summarize multiple steps of execution to concentrate on the most interesting configurations.
- ***Definition***: We say $\langle S_0, \sigma_0 \rangle$ ***evaluates to*** $\langle S_n, \sigma_n \rangle$ in ***n steps*** and write $\langle S_0, \sigma_0 \rangle \rightarrow^n \langle S_n, \sigma_n \rangle$ if there are $n + 1$ configurations $\langle S_0, \sigma_0 \rangle, \ldots, \langle S_{n-1}, \sigma_{n-1} \rangle$ such that $\langle S_0, \sigma_0 \rangle \rightarrow \langle S_1, \sigma_1 \rangle \rightarrow \ldots \langle S_{n-1}, \sigma_{n-1} \rangle \rightarrow \langle S_n, \sigma_n \rangle$. For $n = 0$ we define $\langle S_0, \sigma_0 \rangle \rightarrow^0 \langle S_0, \sigma_0 \rangle$.
- ***Definition***: We say $\langle S_0, \sigma_0 \rangle$ ***evaluates to*** $\langle U, \tau \rangle$ and write $\langle S_0, \sigma_0 \rangle \rightarrow^* \langle U, \tau \rangle$ if $\langle S_0, \sigma_0 \rangle \rightarrow^n \langle U, \tau \rangle$ for some n.
  - An equivalent way to say all of this is that $\rightarrow^n$ is the n–fold composition of $\rightarrow$ and $\rightarrow^*$ is the reflexive transitive closure of $\rightarrow$.
- ***Definition***: Execution of S starting in σ ***terminates in*** (= ***converges to***) τ if $\langle S, \sigma \rangle \rightarrow^* \langle E, \tau \rangle$. Execution of S starting in σ ***terminates*** (= ***converges***) if it terminates in some τ.
  - Recall: "terminate" and "converge" are synonyms, "diverge" is the antonym.
- ***Example 10***: Let S be $s := 0 ; k := n + 1 ; W$, where $W \equiv$ ***while*** $k > 0$ ***do*** S_1 ***od*** and $S_1 \equiv k := k - 1 ; s := s + k$. Since we're going to execute the body S multiple times, it will be helpful to look at a generic execution of S in an arbitrary state $\tau[k \mapsto \alpha][s \mapsto \beta]$. We find

  $\langle S_1, \tau[k \mapsto \alpha][s \mapsto \beta] \rangle = \langle k := k - 1 ;\ s := s + k, \tau[k \mapsto \alpha][s \mapsto \beta] \rangle$
  $\rightarrow \langle s := s + k, \tau[s \mapsto \beta][k \mapsto \alpha - 1] \rangle$
  $\rightarrow \langle E, \tau[k \mapsto \alpha - 1][s \mapsto \beta + \alpha - 1]) \rangle$
- Combining the two steps gives us $\langle S_1, \tau[k \mapsto \alpha][s \mapsto \beta] \rangle \rightarrow^2 \langle E, \tau[k \mapsto \alpha - 1][s \mapsto \beta + \alpha - 1] \rangle$. Adding the test at the top of the loop (for when $\alpha > 0$) lets us give behavior of an arbitrary loop iteration, namely $\langle W, \tau[k \mapsto \alpha][s \mapsto \beta] \rangle \rightarrow^3 \langle W, \tau[k \mapsto \alpha - 1][s \mapsto \beta + \alpha - 1] \rangle$.
- Now let's evaluate S when n is 2; if $\sigma(n) = 2$, then

  $\langle S, \sigma \rangle = \langle s := 0 ; k := n + 1 ; W, \sigma \rangle$
  $\rightarrow^2 \langle W, \sigma[s \mapsto 0][k \mapsto 3] \rangle$ // *After loop initialization*
  $\rightarrow^3 \langle W, \sigma[k \mapsto 2][s \mapsto 2] \rangle$ // *After one iteration of the loop*
  $\rightarrow^3 \langle W, \sigma[k \mapsto 1][s \mapsto 3] \rangle$ // *After two iterations*
  $\rightarrow^3 \langle W, \sigma[k \mapsto 0][s \mapsto 3] \rangle$ // *After three iterations*
  $\rightarrow \langle E, \sigma[k \mapsto 0][s \mapsto 3] \rangle$ // *And now we stop, since* $k > 0$ *is false*

Denotational Semantics; Runtime Errors

CS 536: Science of Programming, Spring 2023

2023-01-31: pp. 1,5,6

A. Why

- A program or statement can be viewed as denoting a state transformation.
- Infinite loops and runtime errors cause failure of normal program execution.

B. Outcomes

At the end of today, you should know how to

- Use denotational semantics to describe overall execution of programs in our language.
- Determine that evaluation of an expression or program fails due to a runtime error.

C. Denotational Semantics Definition and Rules

- We've seen the "small" step-by-step operational semantics for our programs. Today we'll look at a "large" step semantics.
- ***Definition***: If in state σ, program S terminates in τ, then τ is the ***denotational semantics*** of S in σ. Symbolically, if $\langle S, \sigma \rangle \rightarrow^* \langle E, \tau \rangle$, then we write $M(S, \sigma) = \{\tau\}$.
  - The reason we have a singleton set containing τ instead of just τ is that later, we'll look at non-deterministic computations, which can have more than one possible final state.
- ***Notation***: We'll often write $M(S, \sigma) = \tau$ instead of $\{\tau\}$. The only (obscure) time the difference is important is when $\tau = \emptyset$; in that case we should write $M(S, \sigma) = \{\emptyset\}$.
- ***Example 1***: Let σ be a state and let $S \equiv x := 1 ; y := 2$. Since $\langle x := 1 ; y := 2, \sigma \rangle \rightarrow \langle y := 2, \sigma[x \mapsto 1] \rangle \rightarrow \langle E, \sigma[x \mapsto 1][y \mapsto 2] \rangle$, we have $M(S, \sigma) = \{\sigma[x \mapsto 1][y \mapsto 2]\}$.

  [2023-01-31]

  $$
  \begin{aligned}
  & M(x := 1 ; y := 2, \sigma) \\
  &= M(y := 2, \sigma[x \mapsto 1]) && \text{asgt } x{:=}1 \\
  &= M(E, \sigma[x \mapsto 1][y \mapsto 2]) && \text{asgt } y{:=}2 \\
  &= \{\sigma[x \mapsto 1][y \mapsto 2]\} && \text{defn } M
  \end{aligned}
  $$

- ***Notation***: In the literature, some people write hollow square brackets around arguments that are syntactic to emphasize that they are indeed syntactic. E.g, our $\sigma(e)$ would be written $\sigma[\![e]\!]$.
- ***Notation:*** Another notation defines $M[\![S]\!]$ or $M(S)$ as a state transformation function, which you can apply to a state σ to get τ. One writes $\tau = M[\![S]\!](\sigma)$ or $M(S)(\sigma)$ or $M[\![S]\!]\sigma$ or $M(S)\sigma$. In yet another notation we pass a set of possible start states to $M(S, \ldots)$ and get a set of possible end states; we'd write $M(S, \{\sigma\}) = \{\tau\}$.

Denotational Semantics Rules

- Since $M(S, \sigma) = \tau$ means $\langle S, \sigma \rangle \rightarrow^* \langle E, \tau \rangle$, we can give specific rules for $M(S, \sigma)$ depending on the kind of S.
- ***Skip and Assignment Statements:*** These statements complete in only one step, so the operational semantics rules give the denotational semantics immediately.
  - $M(\textbf{skip}, \sigma) = \{\sigma\}$.
  - $M(v := e, \sigma) = \{\sigma[v \mapsto \sigma(e)]\}$.
  - $M(b[e_1] := e, \sigma) = \{\sigma[b[\alpha] \mapsto \beta]\}$ where $\alpha = \sigma(e_1)$ and $\beta = \sigma(e)$.
- ***Composition Statements***: $M(S_1; S_2, \sigma) = M(S_2, \tau)$ where $\{\tau\} = M(S_1, \sigma)$. To justify this, say we have $\langle S_1; S_2, \sigma \rangle \rightarrow^* \langle S_2, \tau \rangle \rightarrow^* \langle E, \tau' \rangle$. Since $M(S_1, \sigma) = \{\tau\}$, we run S_2 starting in state τ, so $M(S_1; S_2, \sigma) = M(S_2, \tau) = M(S_2, M(S_1, \sigma))$. Note: In $M(S_2, M(S_1, \sigma))$, the subscripts appear as *2* then *1*, not *1* then *2*.
- ***Notation***: We'll bend the notation a bit and allow $M(S_2, M(S_1, \sigma))$ as short for $M(S_2, \tau)$ when $M(S_1, \sigma) = \{\tau\}$.
- ***Conditional Statements***: The meaning of an ***if-else*** statement is either the meaning of the true branch or the meaning of the false branch.
  - If $\sigma(B) = T$, then $M(\textbf{if } B \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi}, \sigma) = M(S_1, \sigma)$
  - If $\sigma(B) = F$, then $M(\textbf{if } B \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi}, \sigma) = M(S_2, \sigma)$
- ***Example 2***: Let $S \equiv$ ***if*** y ***then*** $x := x + 1$ ***else*** $z := x + 2$ ***fi***, then
  - If $\sigma(y) = T$, then $M(S, \sigma) = \{\sigma[x \mapsto \sigma(x) + 1]\}$
  - If $\sigma(y) = F$, then $M(S, \sigma) = \{\sigma[z \mapsto \sigma(x) + 2]\}$
- ***Iterative Statements***: One way to definite the meaning of $W \equiv$ ***while*** B ***do*** S ***od*** is recursively. The definition is appealing intuitively but is not well-formed if W leads to an infinite loop.
  - If $\sigma(B) = F$, then $M(W, \sigma) = \{\sigma\}$
  - If $\sigma(B) = T$, then $M(W, \sigma) = M(S; W, \sigma) = M(W, M(S, \sigma))$.
- Another way to characterize $M(W, \sigma)$ involves looking at the series of states in which we evaluate the test.
  - Let $\sigma_0 = \sigma$, and for $k = 0, 1, \ldots$, let $\{\sigma_{k+1}\} = M(S, \sigma_k)$. Then $\sigma_0, \sigma_1, \sigma_2, \ldots$ is the sequence of states seen at successive ***while*** loop tests: The k'th time we evaluate the loop test, we use state σ_k.
  - Now we can define $M(W, \sigma) = \{\sigma_k\}$ for the least k in which B is false, if there is one. If there isn't, we have an infinite loop.
  - Then $M(W, \sigma)$ is the (set containing the) first state in this sequence that satisfies $\neg B$, assuming there is such a state. (If there isn't, we have an infinite loop.)
- ***Example 3***: Let $W \equiv$ ***while*** $x < n$ ***do*** S ***od***, where the loop body $S \equiv x := x + 1; y := y + y$, and let's calculate $M(W, \sigma)$ where $\sigma = \{x = 0, n = 3, y = 1\}$.

- The behavior of S in an arbitrary state τ is $M(S, \tau[x \mapsto \alpha][y \mapsto \beta]) = \{\tau[x \mapsto \alpha + 1][y \mapsto 2\beta]\}$. Then our sequence of states is
- $\sigma_0 = \sigma = \{x = 0,\ n = 3,\ y = 1\}$
  - $M(S, \sigma_0) = \{\sigma_1\}$ where $\sigma_1 = \{x = 1,\ n = 3,\ y = 2\}$
  - $M(S, \sigma_1) = \{\sigma_2\}$ where $\sigma_2 = \{x = 2,\ n = 3,\ y = 4\}$, and
  - $M(S, \sigma_2) = \{\sigma_3\}$ where $\sigma_3 = \{x = 3,\ n = 3,\ y = 8\}$.
- Of this sequence, σ_3 is the first state that satisfies $x \geq n$, so $M(W, \sigma) = \{\sigma_3\} = \{\{x = 3, n = 3, y = 8\}\}$.
- Since we stop at σ_3, there's no need to calculate $M(S, \sigma_3) = \{\sigma_4\}$ to find that $\sigma_4 = \{x = 4, n = 3, y = 16\}$. (It's not incorrect, it's just not useful.)

D. Convergence and Divergence of Loops

- Not all loops terminate. Evaluation of an infinite loop yields an unending path of $\rightarrow$ steps: Either an infinite sequence of different configurations or a finite-length cycle of configurations. More generally in computer science we can also also have infinite recursion, which we won't study in detail but is treated similarly to infinite iteration.
- (Recall that S starting in σ ***converges*** to τ / ***terminates*** in τ if $\langle S, \sigma\rangle \rightarrow^* \langle E, \tau\rangle$ (operationally) or $M(S, \sigma) = \{\tau\}$ (denotationally). If S does not converge, it's said to ***diverge***.)
- ***Note***: Divergence is one way in which a program fails to successfully terminate.
- Rather than write $M(S, \sigma) = \varnothing$ for a divergent calculation, we'll use a "pseudo state" $\perp$ (pronounced "bottom"), so $M(S, \sigma) = \{\perp\}$ means that S doesn't terminate successfully in a final state. (Either it doesn't terminate, i.e., diverges, or it gets some sort of runtime error and halts.)
- We'll introduce other flavors of $\perp$ as we look at other ways to not get successful termination.

Divergence: The pseudo-state $\perp_d$ ("bottom sub-d")

- ***Notation***: Denotationally, $M(S, \sigma) = \{\perp_d\}$ means S diverges in σ. Note that although we're writing it in a place where you'd expect a memory state,$\perp_d$ is not an actual memory state; we'll call it a ***pseudo-state*** as apposed to an ***actual*** or ***real*** memory state like σ and τ.
- ***Notation***: Operationally, $\langle S, \sigma\rangle \rightarrow^* \langle E, \perp_d\rangle$ means that S starting in σ diverges. Again, we're not using $\perp_d$ as an actual memory state here, but since $M(S, \sigma) = \{\tau\}$ means $\langle S, \sigma\rangle \rightarrow^* \langle E, \tau\rangle$, if we're going to write $M(S, \sigma) = \{\perp_d\}$ to say that S diverges, writing $\langle S, \sigma\rangle \rightarrow^* \langle E, \perp_d\rangle$ is notationally consistent[1].
- To determine when $M(W, \sigma) = \{\perp_d\}$, recall that in the previous section we looked at the series of states $\sigma_0, \sigma_1, \sigma_2, \ldots$ in which we evaluate the loop test. For this sequence, $\sigma_0 = \sigma$, and $\sigma_{k+1} = M(S, \sigma_k)$ for $k \geq 0$. For terminating loops, $M(W, \sigma)$ is the first state in the sequence that

[1] We should let $\rightarrow^*$ include a countably infinite number of steps, since you can only infer divergence by standing back after watching that may steps.

satisfies $\neg B$. We can now write $M(W, \sigma) = \{ \perp_d \}$ to indicate that no state in the sequence satisfies $\neg B$.

- ***Example 4***: Let $W \equiv$ ***while*** T ***do skip od*** and σ be any state. Then $\langle W, \sigma \rangle \rightarrow \langle$ ***skip*** $; W, \sigma \rangle \rightarrow \langle W, \sigma \rangle$. Hence $M(W, \sigma) = \{ \perp_d \}$. As a directed graph, this is a two-node cycle, $\langle W, \sigma \rangle \rightleftarrows \langle$ ***skip*** $; W, \sigma \rangle$.
- ***Example 5***: Let $W \equiv$ ***while*** $x \neq n$ ***do*** $x := x - 1$ ***od*** and let $\sigma = \{ x = -1, n = 0 \}$.
  - Let $\sigma_0 = \sigma = \{ x = -1, n = 0 \}$
  - Let $\{ \sigma_1 \} = M(x := x - 1, \sigma_0) = \{ \sigma_0 [x \mapsto -2] \} = \{ \{ x = -2, n = 0 \} \}$
  - Let $\{ \sigma_2 \} = M(x := x - 1, \sigma_1) = \{ \sigma_1 [x \mapsto -3] \} = \{ \{ x = -3, n = 0 \} \}$
  - In general, let $\{ \sigma_k \} = M(x := x - 1, \sigma_{k-1}) = \{ \{ x = -k - 1, n = 0 \} \}$
  - Since every $\sigma_k \models x \neq n$, we have $M(W, \sigma) = \{ \perp_d \}$.

E. Expressions With Runtime Errors: The pseudo-state $\perp_e$

- Using $\perp_d$ lets us talk about a program not successfully terminating because it simply doesn't terminate at all.
- Runtime errors cause a program to terminate, but unsuccessfully E.g, in σ, the assignment $z := x/y$ fails if $\sigma(y) = 0$ because evaluation of $\sigma(x/y)$ fails. There are two notions of failure here: The expression fails, and this causes the statement to fail.
- ***Definition***: $\sigma(e) = \perp_e$ means evaluation of expression e in state σ causes a runtime error.
  - Here, $\perp_e$ is used as a pseudo-value of an expression, to indicate an error. It's not a value; we're writing it in place of an actual value.
  - If e can fail at runtime, then instead of $\sigma(e) \in V$ for some set of values V, we now have $\sigma(e) \in V \cup \{ \perp_e \}$. Of course, some expressions never fail: $\sigma(2 + 2) \in \mathbb{Z}$, not just $\sigma(2 + 2) \in \mathbb{Z} \cup \{ \perp_e \}$.
- ***Primary Failure***: The primitive values and operations being supported determine some set of basic runtime errors. For us, let's include:
  - *Array index out of bounds*: $\sigma(b[e]) = \perp_e$ if $\sigma(e) < 0$ or $\geq \sigma(size(b))$; similar for multiple dimensions.
  - *Division by zero*: $\sigma(e_1 / e_2) = \sigma(e_1 \% e_2) = \perp_e$ if $\sigma(e_2) = 0$.
  - *Square root of negative number*: $\sigma(sqrt(e)) = \perp_e$ if $\sigma(e) < 0$.
  - ***Example 6***: $b[-1]$, $n/0$, and $sqrt(-1)$ fail for all σ. $b[k]$ fails in state $\{ b = (2, 3, 5, 8), k = 4 \}$ but not in state $\{ b = (6), k = 0 \}$.
- ***Hereditary Failure***: If evaluating a subexpression fails, then the overall expression fails.
  - If op is a unary operator, then $\sigma(op\ e) = \perp_e$ if $\sigma(e) = \perp_e$.
  - If op is a binary operator, then $\sigma(e_1\ op\ e_2) = \perp_e$ if $\sigma(e_1)$ or $\sigma(e_2) = \perp_e$.
  - For a conditional expression, $\sigma($***if*** B ***then*** e_1 ***else*** e_2 ***fi***$) = \perp_e$ if one of the following three situations occurs: (1) $\sigma(B) = \perp_e$ (2) $\sigma(B) = T$ and $\sigma(e_1) = \perp_e$ or (3) $\sigma(B) = F$ and $\sigma(e_2) = \perp_e$.

  - As usual, ***if*** expressions are executed lazily: We don't worry about a hypothetical failure of the branch we don't evaluate.
  - [2023-01-31] if σ(e) = $\perp_e$ then b[e]=$\perp_e$.
- ***Example 7***: $\sigma(x/y) = \perp_e$ when $\sigma(y) = 0$, but $\sigma($ ***if*** $y = 0$ ***then*** 0 ***else*** x/y ***fi*** $)$ never=$\perp_e$.

F. Statements With Runtime Errors

- An expression that causes a runtime error causes the statement it appears in terminate unsuccessfully. We'll write $\langle S, \sigma\rangle \to \langle E, \perp_e\rangle$ for the operational semantics of such a statement. This use of $\perp_e$ as a (pseudo)-state is different from its use as a pseudo-value in $\sigma(e) = \perp_e$.
- ***Definition: (Statements with expressions with runtime errors)*** If a statement evaluates an expression that causes a runtime error, then the statement terminates unsuccessfully. To the operational semantics, we add:
  - If $\sigma(e) = \perp_e$, then $\langle v := e, \sigma\rangle \to \langle E, \perp_e\rangle$.
  - If $\sigma(b[e_1])$ or $\sigma(e_2) = \perp_e$, then $\langle b[e_1] := e_2, \sigma\rangle \to \langle E, \perp_e\rangle$.
  - If $\sigma(B) = \perp_e$, then $\langle$ ***if*** B ***then*** S_1 ***else*** S_2 ***fi***$, \sigma\rangle \to \langle E, \perp_e\rangle$.
    - If $\sigma(B)$ does not fail, we continue with $\langle S_1, \sigma\rangle$ or $\langle S_2, \sigma\rangle$. Failure of those automatically cause failure of the overall conditional, so there's no need to treat failure of the true or false branch as a separate case.
  - If $\langle S_1, \sigma\rangle \to \langle E, \perp_e\rangle$ then $\langle S_1; S_2, \sigma\rangle \to \langle E, \perp_e\rangle$.
  - If $\sigma(B) = \perp_e$, then $\langle$ ***while*** B ***do*** S ***od***$, \sigma\rangle \to \langle E, \perp_e\rangle$.
    - If $\sigma(B)$ is true, we continue with $\langle S;$ ***while*** B ***do*** S ***od***$, \sigma\rangle$, and failure of this causes failure of the loop, so we don't need to treat failure of the loop body as a separate case.
- The pseudo-states $\perp_d$ and $\perp_e$ share some properties, so it's helpful to have a more general notation for "error".
- ***Notation***: $\perp$ refers generically to $\perp_d$ and/or $\perp_e$. In particular we use $\langle S, \sigma\rangle \to^* \langle E, \perp\rangle$ when it's not important which of $\perp_e$ or $\perp_d$ can occur. Similarly, $\perp \in M(S, \sigma)$ means $\langle S, \sigma\rangle$ leads to $\perp_d$ or $\perp_e$.

Properties and Consequences of $\perp$

- ***Trying to use*** $\perp$: Since we are writing $\perp$ in some of the places where an actual memory state would appear, we should be thorough and look at the other places states appear so we can extend those notions or notations.
  - $\perp$ is not a well-formed state.
  - When we say "for all states..." or "for some state...," we don't include $\perp$.
  - We can't add a binding to $\perp$: $\perp[v \mapsto \beta] = \perp$.
  - We can't bind a variable to $\perp$: $\sigma(v) \neq \perp$ and $\sigma[v \mapsto \perp] = \perp$.
  - We can't take the value of a variable or expression in $\perp$: If $\sigma = \perp$ then $\sigma(v) = \sigma(e) = \perp$. (More succinctly, $\perp(v) = \perp(e) = \perp$.)

  - Operationally, execution halts as soon we generate $\perp$ as a "state": $\langle S, \perp\rangle \to^0 \langle E, \perp\rangle$.
  - Denotationally, we can't run a program in $\perp$: $M(S, \perp) = \{\perp\}$.
- From the properties we have, it follows that we can't evaluate something after generating $\perp$.
  - If $\langle S_1, \sigma\rangle \to \langle E, \perp\rangle$, then $\langle S_1; S_2, \sigma\rangle \to \langle E, \perp\rangle$.
  - If $M(S_1, \sigma) = \{\perp\}$, then $M(S_1; S_2, \sigma) = M(S_2, M(S_1, \sigma)) = M(S_2, \perp) = \{\perp\}$.
  - If $W \equiv$ ***while*** B ***do*** S_1 ***od*** and $\sigma(B) = T$ but $M(S_1, \sigma) = \{\perp\}$, then $M(W, \sigma) = \{\perp\}$.
    - (In detail, $M(W, \sigma) = M(S_1;\ W, \sigma) = M(W, M(S_1, \sigma)) = M(W, \perp) = \{\perp\}$.)
- ***Satisfaction and Validity and*** $\perp$: Note: $\perp$ never satisfies a predicate: $\perp \not\models p$ for all p, even if p is the constant T. In general, we now have three possibilities for a state trying to satisfy a predicate: $\sigma \models p$, $\sigma \models \neg p$, or $\sigma = \perp$. So $\sigma \not\models p$ implies $\sigma \models \neg p$ or $\sigma = \perp$, not just $\sigma \models \neg p$.
  - Note if $\sigma = \perp$, then $\sigma \not\models p$ and $\sigma \not\models \neg p$, both. The converse doesn't hold, though. E.g., if p is the conditional expression ***if*** $x = 0$ ***then*** $x/x = 0$ ***else*** $2 = 2$ ***fi***, then $\{x = 0\} \not\models p$ because $0/0$ causes a runtime error. For the false branch, $\{x = 1\} \models 2 = 2$, so $\{x = 1\} \not\models \neg(2 = 2)$, so $\{x = 1\} \not\models \neg p$.
- ***Logical negation and*** $\perp$: We still have that $\sigma \models \neg p$ implies $\sigma \not\models p$, but the converse no longer holds. It's possible now for the meaning of p to be $\perp$ (we'll look at this more in a moment), so $\sigma \not\models p$ doesn't imply $\sigma \models \neg p$. We need a new answer to the question of what $\sigma \models \neg p$ means. The solution is to treat $\neg p$ as shorthand for $p \to F$ where F is the predicate false.
  - Just a quick note: For the meaning of T and F, we have $\sigma \models T$ and $\sigma \not\models F$ for all $\sigma \neq \perp$. (We can also derive F by defining $F \equiv T \neq T$). For all σ that are not $\perp$, we get $\sigma \models F \to F$, so $\sigma \models \neg F$.
- ***Generating*** $\perp$ ***while testing for satisfaction***: We certainly don't want to say $\{y = 0\} \models y/y = 1$. To handle the situation of $\sigma \models p$ when evaluation of p causes an error, we can add $\perp$ to the semantics of basic operations and tests.
  - For any *relation* (like less than, etc), we have $(\beta$ *relation* $\delta)$ yields $\perp$ if β or δ are $\perp$.
  - For any binary *operation* (like addition, etc), we have $(\alpha$ *operation* $\beta)$ yields $\perp$ if β or $\delta = \perp$.
  - Similarly for a unary operation, we have $($*operation* $\perp)$ yields $\perp$.
  - [2023-01-31] this also applies to logical relations/operations.
- Some of the implications of this are reasonably intuitive: $(\perp$ *plus one*$)$ yields $\perp$. But some implications are less intuitive: Semantic operations and tests like $\perp \neq 2$, $\perp < \perp$, $\perp = \perp$, and $\perp \neq \perp$ all yield $\perp$ (not T or F).
  - Returning to $y/y = 1$, we have $\sigma \models y/y = 1$ iff $\sigma(y/y) = \sigma(1)$ iff $\sigma(y) \div \sigma(y) = 1$, so if $\sigma(y) =$ some $\beta \neq 0$, then $\sigma \models y/y = 1$ iff $\beta \div \beta = 1$ iff $1 = 1$ iff T.
    - But if $\sigma(y) = 0$, then $\sigma \models y/y = 1$ iff $0 \div 0 = 1$ iff $\perp = 1$ iff $\perp$. Hence $\sigma \not\models y/y = 1$.
    - But also, since $\perp \neq 1$ is $\perp$, we also have $\sigma \not\models y/y \neq 1$.
    - So as expected, here σ satisfies neither $y/y = 1$ nor $y/y \neq 1$.=

Sequential Nondeterminism

CS 536: Science of Programming, Spring 2023

A. Why

- Nondeterminism can help us avoid unnecessary determinism.
- Nondeterminism can help us develop programs without worrying about overlapping cases.

B. Objectives

At the end of this class you should know

- The syntax and operational and denotational semantics of nondeterministic statements.

C. Avoiding Unnecessary Design Choices Using Nondeterminism

- When writing programs, it's hard enough concentrating on the decisions we ***must*** make at any given time, so it's helpful to avoid making decisions we don't have to make.
- ***Example 1***: A very simple example is a statement that sets max to the larger of x and y. It doesn't really matter which of the following two statements we use. They're written differently but behave the same:
  - ***if*** $x \geq y$ ***then*** $max := x$ ***else*** $max := y$ ***fi***
  - ***if*** $y \geq x$ ***then*** $max := y$ ***else*** $max := x$ ***fi***
- The difference is when $x = y$, the first statement sets $max := x$; the second sets $max := y$. It doesn't matter which one of these we choose, we just have to pick one.
- Our standard ***if-else*** statement is ***deterministic***: It can only behave one way. A nondeterministic ***if-fi*** will specify that one of $max := x$ and $max := y$ has to be run, but it won't say how we choose which one.
  - We don't plan to execute our programs nondeterministically; we design programs using nondeterminism in order to delay making unnecessary decisions about the order in which our code makes choices.
  - When we make the code more concrete by rewriting it using everyday deterministic code, then we'll decide which way to write it.

D. Nondeterministic if-fi

- ***Syntax***: ***if*** $B_1 \rightarrow S_1 \ \square \ B_2 \rightarrow S_2 \ \square \ \ldots \ \square \ B_n \rightarrow S_n$ ***fi***
  - The box symbols separate the different arms, like commas in an ordered n-tuple.
  - Don't confuse these right arrows with ones in other contexts (implication operator and single-step execution).

- ***Definition***: Each $B_i \rightarrow S_i$ clause is a ***guarded command***. The ***guard*** B_i tells us when it's okay to run S_i.
- ***Informal semantics***
  - If none of the guard tests B_1, B_2, ..., B_n are true, abort with a runtime error.
  - If exactly one guard B_i is true then execute S_i.
  - If more than one guard is true, then select a corresponding statement and execute it.
    - The selection is made nondeterministically (unpredictably); we'll discuss this more soon.
- ***Example 2***: ***if*** $x \geq y \rightarrow max := x \ \square \ y \geq x \rightarrow max := y$ ***fi*** sets max to the larger of x and y.
  - If only one of $x \geq y$ and $y \geq x$ is true, we execute its corresponding assignment.
  - If both are true, we choose one of them and execute its assignment.
- In this example, the two arms set max to the same value when $x = y$, so it doesn't matter which one gets used.
- In more general examples, the different arms might behave differently but as long as each gets us to where we're going, we don't care which one gets chosen.
  - E.g., say we have an ***if-fi*** with two arms; one arm sets a variable $z := 0$; the other arm sets $z := 1$. If, for correctness's sake, we need $z \geq 0$ after the ***if-fi***, then this is fine. (If we needed $even(z)$, for example, we'd have a bug.)
- We can also have ***if-fi*** statements that never have to make a nondeterministic choice.
  - ***Example 3***: Our usual deterministic ***if*** B ***then*** S_1 ***else*** S_2 ***fi*** can be written as ***if*** $B \rightarrow S_1 \ \square \ \neg B \rightarrow S_2$ ***fi***.

E. Nondeterministic Choices are Unpredictable

- For us, "nondeterministic" means "unpredictable".
- Let *flip* ≡ ***if*** $T \rightarrow x := 0 \ \square \ T \rightarrow x := 1$ ***fi***, which sets x to either 0 or 1. I've called it *flip* because it's similar to a coin flip, but it's not identical.
  - With a real coin flip, you expect a 50-50 chance of getting 0 or 1, but since *flip* is nondeterministic, its behavior is completely unpredictable.
  - A thousand calls of *flip* might give us anything: all 0's, all 1's, some pattern, random 500 heads and 500 tails, etc.
- ***Nondeterminism shouldn't affect correctness***: We write nondeterministic code when we don't want to worry about how choices are made: We only want to worry about producing correct results given that a choice has been made.
  - E.g., code written using *flip* should produce a correct final state whether we get heads or tails. Of course, eventually, we'll replace *flip* with a deterministic coin-flipping routine, and at that point we'll have to worry about the fairness of the deterministic routine.

F. Nondeterministic Loop

- Nondeterministic loops are very similar to nondeterministic conditionals, both in syntax and semantics. We can derive nondeterministic loops using nondeterministic if and a ***while*** loop.
- ***Syntax***: ***do*** $B_1 \rightarrow S_1 \;\square\; B_2 \rightarrow S_2 \;\square\; \ldots \;\square\; B_n \rightarrow S_n$ ***od***
- ***Informal semantics***:
  - At the top of the loop, check for any true guards.
  - If no guard is true, the loop terminates.
  - If exactly one guard is true, execute its corresponding statement and jump to the top of the loop.
  - If more than one guard is true, select one of the corresponding guarded statements and execute it. (The choice is nondeterministic.) Once we finish the guarded statement, jump to the top of the loop.
- A nondeterministic do loop is equivalent to a regular ***while*** loop (with a nondeterministic test but) with a nondeterministic if body. Let $BB \equiv (B_1 \vee B_2 \ldots \vee B_n)$ be the disjunction of the guards, then ***do*** $B_1 \rightarrow S_1 \;\square\; \ldots \;\square\; B_n \rightarrow S_n$ ***od*** behaves like ***while*** BB ***do if*** $B_1 \rightarrow S_1 \;\square\; \ldots \;\square\; B_n \rightarrow S_n$ ***fi od***.

G. Operational Semantics of Nondeterministic *if-fi*

- Let $IF \equiv$ ***if*** $B_1 \rightarrow S_1 \;\square\; B_2 \rightarrow S_2 \;\square\; \ldots \;\square\; B_n \rightarrow S_n$ ***fi*** and let $BB \equiv B_1 \vee B_2 \vee \ldots B_n$.
- To evaluate IF,
  - If evaluation of any guard fails $(\sigma(BB) = \perp_e)$, then IF causes an error: $\langle IF, \sigma \rangle \rightarrow \langle E, \perp_e \rangle$.
  - If none of the guards are satisfied $(\sigma(BB) = F)$, then IF causes an error: $\langle IF, \sigma \rangle \rightarrow \langle E, \perp_e \rangle$.
  - If one or more guarded commands $B_i \rightarrow S_i$ have $\sigma(B_i) = T$, then one such i is chosen nondeterministically and we jump to the beginning of S_i: $\langle IF, \sigma \rangle \rightarrow \langle S_i, \sigma \rangle$.

H. Operational Semantics of Nondeterministic *do-od*

- Let $DO \equiv$ ***do*** $B_1 \rightarrow S_1 \;\square\; B_2 \rightarrow S_2 \;\square\; \ldots \;\square\; B_n \rightarrow S_n$ ***od*** and let $BB \equiv B_1 \vee B_2 \vee \ldots B_n$.
- Evaluation of DO is very similar to evaluation if IF:
  - If evaluation of any guard fails $(\sigma(BB) = \perp_e)$, then DO causes an error: $\langle DO, \sigma \rangle \rightarrow \langle E, \perp_e \rangle$.
  - If none of the guards are satisfied $(\sigma(BB) = F)$, then the loop halts: $\langle DO, \sigma \rangle \rightarrow \langle E, \sigma \rangle$.
  - If one or more guarded commands $B_i \rightarrow S_i$ have $\sigma(B_i) = T$, then one such i is chosen nondeterministically and we jump to the beginning of S_i; after it completes, we'll jump back to the top of the loop: $\langle DO, \sigma \rangle \rightarrow \langle S_i; DO, \sigma \rangle$.

I. Denotational Semantics of Nondeterministic Programs

- ***Notation***:
  - Σ is the set of all states (that proper for whatever we happen to be discussing at that time).
  - $\Sigma_\perp = \Sigma \cup \{\text{all flavors of } \perp\} = \Sigma \cup \{\perp_d, \perp_e\}$ right now; other versions can be added later.

- As before, writing $\tau = \perp$ means $\tau \in \{\perp_d, \perp_e\}$, so it refers ambiguously to one or the other.
- For a nondeterministic program, to get its denotational semantics, we have to collect all the possible final states (or the pseudo-state $\perp$): $M(S, \sigma) = \{\tau \in \Sigma_\perp \mid \langle S, \sigma\rangle \rightarrow^* \langle E, \tau\rangle\}$.
- For a deterministic program, there is only one such τ, so this simplifies to our earlier definition: $M(S, \sigma) = \{\tau\}$ where $\langle S, \sigma\rangle \rightarrow^* \langle E, \tau\rangle$ and $\tau \in \Sigma_\perp$.
- ***Example 4***: Let $S \equiv$ ***if*** $T \rightarrow x := 0 \ \square \ T \rightarrow x := 1$ ***fi***. Then $\langle S, \varnothing\rangle \rightarrow^* \langle E, x = 0\rangle$ and $\langle S, \varnothing\rangle \rightarrow^* \langle E, x = 1\rangle$ are both possible, and $M(S, \sigma) = \{\{x = 0\}, \{x = 1\}\}$. (Be careful not to write this as $\{\{x = 0, x = 1\}\}$, which is a set containing a single, ill-formed state.) For any particular execution of S in a σ, we'll get exactly one of these final states.
- ***Notation***: For convenience, most times we can still abbreviate $M(S, \sigma) = \{\tau\}$ to $M(S, \sigma) = \tau$. But let's agree not to shorten $M($***skip***$, \varnothing) = \{\varnothing\}$ to $M($***skip***$, \varnothing) = \varnothing$, since it might look like we're claiming that $M($***skip***$, \varnothing)$ has no final state — it does, the empty state.
- A nondeterministic program doesn't have to have multiple final states.
  - ***Example 5***: The *max* program from Example 2 has only one final state. Let $Max \equiv$ ***if*** $x \geq y \rightarrow max := x \ \square \ y \geq x \rightarrow max := y$ ***fi***, in the nondeterministic case, where $x = y$, both possible execution paths take us to the same state: $M(Max, \{x = \alpha, y = \alpha\}) = \{\{x = \alpha, y = \alpha, max = \alpha\}\}$.
  - ***Note***: To keep from confusing the graders, avoid writing things that look like multisets, such as "$\{\tau, \tau\}$ where $\tau = \{x = \alpha, y = \alpha, max = \alpha\}$".
- For arbitrary S, if $M(S, \sigma)$ has more than one member, then S is nondeterministic. The *Max* program shows us that the converse doesn't hold: If $M(S, \sigma)$ has just one member, S still could be nondeterministic. Note also that the size of $M(S, \sigma)$ can vary depending on σ.
  - ***Example 6***: If $S \equiv$ ***if*** $x \geq 0 \rightarrow x := x * x \ \square \ x \leq 8 \rightarrow x := -x$ ***fi***, then $M(S, \{x = 0\}) = \{\{x = 0\}\}$, but $M(S, \{x = 3\}) = \{\{x = 9\}, \{x = -3\}\}$.

Difference between M(S,σ) = {τ} and τ ∈ M(S,σ)

- There's a difference between $M(S, \sigma) = \{\tau\}$ and $\tau \in M(S, \sigma)$. They both say that τ can be a final state, but $M(S, \sigma) = \{\tau\}$ says there's only one final state, but $\tau \in M(S, \sigma)$ leaves open the possibility that there are other final states.
- In particular, $M(S, \sigma) = \{\perp\}$ says S always causes an error whereas $\perp \in M(S, \sigma)$ says that S might cause an error. Remember that when we write $\perp$, we're being ambiguous as to whether we mean $\perp_d$ or $\perp_e$. If multiple kinds of failure are possible, we should say so, as in $M(S, \sigma) = \{\perp_d, \perp_e\}$ or $\{\perp_d, \perp_e\} \subseteq M(S, \sigma)$.

J. Why Use Nondeterministic programs?

- Without having defined program correctness yet, the discussion here will be informal.

Reason 1: Nondeterminism makes it easer to combine partial solutions

- With nondeterministic code, it's straightforward to combine partial solutions to a problem to form a larger solution. This means we can solve a large problem by solving smaller instances of it and combining them.
- ***Example 7:*** Let's solve the *Max* problem. Say we specify "*Max* takes x and y and (without changing them), sets *max* to the larger of x and y."
  - Since the program has to end with $max = x$ or $max = y$, one way to approach is to ask "When does $max := x$ work?" and "When does $max := y$ work?".
  - Since $max := x$ is correct exactly when $x \geq y$, the program ***if*** $x \geq y \rightarrow max := x$ ***fi*** is (partially) correct. (It's not totally correct, since it fails if $x < y$.)
  - Similarly, since $max := y$ is correct exactly when $x \leq y$, the program ***if*** $x \leq y \rightarrow max := y$ ***fi*** is also (partially) correct.
  - We can combine the two partial solutions and get

    if $x \geq y \rightarrow max := x$
    $\square$ $x \leq y \rightarrow max := y$
    fi
  - This program works when $x \geq y$ or $x \leq y$, and since that covers all possibilities, our program is done.

Reason 2: Nondeterminism makes it easier to find overlapping solutions but delay worrying about them

- ***Find overlapping solutions***: When approaching a problem nondeterministically, you can concentrate on discovering partial solutions but put off decisions about which ones might be better. Here's a vague example: Say we want to process a stream of widgets; the red ones can be processed using techniques A or B; the blue ones can be processed using using techniques B or C. Very roughly, ***do*** $red \rightarrow A \square red \rightarrow B \square blue \rightarrow$ B $\square$ $blue \rightarrow C$ ***od***.
  - Since we're using a nondeterministic approach, we can improve the individual cases separately because we don't have to decide immediately about which process we want to run. For example, we might discover that red widgets can also be handled by process B_1: ***do*** $red \rightarrow A \square red \rightarrow B \square red \rightarrow B_1 \square blue \rightarrow$ B $\square$ $blue \rightarrow C$ ***od***.
- ***Delay worry about overlapping cases***: In nondeterministic ***if***/***do***, the order of the guarded commands makes no difference, so we can it doesn't matter if guards overlap in what states satisfy them. That means we can write the code nondeterministically, with overlapping cases, and not worry about the overlap. We can remove the overlap when we rewrite the code deterministically to run it.
  - Going back to the widget example, we found B_1 for red widgets but don't have to immediately ponder questions like "Should we use B or B_1 for red widgets?" and "Can we improve B for blue widgets (using the insight that gave us B_1 for red widgets?"

- ***Example 8:*** For a simpler example, let's take the *Max* program yet again.
  - Both of these programs are correct:

    if $x \geq y \rightarrow max := x \;\square\; x \leq y \rightarrow max := y$ ***fi***

    if $x \leq y \rightarrow max := y \;\square\; x \geq y \rightarrow max := x$ ***fi***
  - Since the programs behave identically when $x = y$, it doesn't matter if we drop that case from one of the tests, say the second, which yields

    if $x \geq y \rightarrow max := x \;\square\; x < y \rightarrow max := y$ ***fi***

    if $x \leq y \rightarrow max := y \;\square\; x > y \rightarrow max := x$ ***fi***
  - Introducing the asymmetry makes the code correspond to the deterministic statements

    if $x \geq y$ ***then*** $max := x$ ***else*** $max := y$ ***fi***

    if $x \leq y$ ***then*** $max := y$ ***else*** $max := x$ ***fi***
- ***Example 9:*** A similar pair of examples of introducing asymmetry takes

  if $x \geq 0 \rightarrow y := sqrt(x) \;\square\; x \leq 0 \rightarrow y := 0$ ***fi***

  to ***if*** $x \geq 0$ ***then*** $y := sqrt(x)$ ***else*** $y := 0$ ***fi*** or

  if $x > 0$ ***then*** $y := sqrt(x)$ ***else*** $y := 0$ ***fi***

K. Example 10: Array Value Matching

- As an example of how nondeterministic code can help us write programs, let's look at an array-matching problem. We're given three arrays, *b0, b1*, and *b2*, all of length n and all sorted in non-descending order. The goal is to find indexes *k0, k1*, and *k2* such that $b0[k0] = b1[k1] = b2[k2]$ if such values exist.
  - But what if no such *k0*, *k1*, and *k2* exist? One solution is to terminate with $k0 = k1 = k2 = n$. This certainly works, but we need to test each index before testing its value. Assuming "∧" is short-circuiting, we test $k0 < n \wedge k1 < n \wedge b0[k0] < b1[k1]$ to make sure that *k0* and *k1* are in range before using them as indexes.
  - We'll use an alternate approach by having sentinels: We'll assume $b0[n], b1[n],$ and $b2[n]$ all equal +∞ (positive infinity). This lets us write tests like $b0[k0] < b1[k1]$ without having to test for *k0* or $k1 = n$.
- How does the program work? If we set $k0 = k1 = k2 = 0$ initially, then we have to increment *k0* or *k1* or *k2* until we find a match.
- Let's study one pair of indexes, say *k0* and *k1*. There are three cases:
  1. $b0[k0] < b1[k1]$. If this happens, we should increment *k0*. Since the arrays are sorted by ≤, incrementing *k1* can't possibly result in $b0[k0] = b1[k1]$, whereas incrementing *k0* might.
  2. $b0[k0] > b1[k1]$. Symmetrically, if this happens, we should increment *k1*.

3. $b0[k0] = b1[k1]$. If this happens, we don't want to do anything, since we have a possible match. (Of course, we still need $b1[k1] = b2[k2]$ or $b0[k0] = b2[k2]$ — they're equivalent in this case.)

- If we write this up as a nondeterministic ***if-fi***, we get

  if $b0[k0] < b1[k1] \rightarrow k0 := k0 + 1 \;\square\; b0[k0] > b1[k1] \rightarrow k1 := k1 + 1$ ***fi***

- Repeating for the other two pairs of indexes, we get

  if $b1[k1] < b2[k2] \rightarrow k1 := k1 + 1 \;\square\; b1[k1] > b2[k2] \rightarrow k2 := k2 + 1$ ***fi***

  if $b2[k2] < b0[k0] \rightarrow k2 := k2 + 1 \;\square\; b2[k2] > b0[k0] \rightarrow k0 := k0 + 1$ ***fi***

- If we repeat these three ***if-fi*** statements until none of the < or > cases apply, then we're guaranteed that = holds between each pair. We can combine the six cases above into:

  // Program 10(a)
  //
  do $b0[k0] < b1[k1] \rightarrow k0 := k0 + 1$
  $\square\;\; b0[k0] > b1[k1] \rightarrow k1 := k1 + 1$
  $\square\;\; b1[k1] < b2[k2] \rightarrow k1 := k1 + 1$
  $\square\;\; b1[k1] > b2[k2] \rightarrow k2 := k2 + 1$
  $\square\;\; b0[k0] < b2[k2] \rightarrow k0 := k0 + 1$
  $\square\;\; b0[k0] > b2[k2] \rightarrow k2 := k2 + 1$
  od

- If none of the loop guards apply, the ≤ and ≥ combine and ensure $b0[k0] = b1[k1] = b2[k2]$.
- The code can be cleaned up a couple of ways. The obvious one is to combine guards that guard the same command:

  // Program 10(b)
  //
  do $b0[k0] < b1[k1] \vee b0[k0] < b2[k2] \rightarrow k0 := k0 + 1$
  $\square\;\; b1[k1] < b2[k2] \vee b1[k1] < b0[k0] \rightarrow k1 := k1 + 1$
  $\square\;\; b2[k2] < b0[k0] \vee b2[k2] < b1[k1] \rightarrow k2 := k2 + 1$
  od

- Another way is to note that if we don't have $b0[k0] = b1[k1] = b2[k2]$, then there must be a < relation between two of the three values. This gives us

  // Program 11(c)
  //
  do $b0[k0] < b1[k1] \rightarrow k0 := k0 + 1$
  $\square\;\; b1[k1] < b2[k2] \rightarrow k1 := k1 + 1$
  $\square\;\; b2[k2] < b0[k0] \rightarrow k2 := k2 + 1$
  od

- For all three of the guards to be false, we need $b0[k0] \geq b1[k1] \geq b2[k2] \geq b0[k0]$, which only happens when $b0[k0] = b1[k1] = b2[k2]$.

Correctness ("Hoare") Triples

Part 1: Definitions and Basic Properties

CS 536: Science of Programming, Spring 2023

2023-02-07 pp. 3, 4, 8

A. Why

- To specify a program's correctness, we need to know its precondition and postcondition (what should be true before and after executing it).
- The semantics of a verified program joins a program's state-transformation semantics with the state-oriented semantics of the specification predicates.

B. Objectives

At the end of today you should know

- The syntax of correctness triples (a.k.a. Hoare triples).
- What it means for a correctness triples to be satisfied or to be valid.
- That a state in which a correctness triple is not satisfied is a state where the program has a bug.

C. Correctness Triples ("Hoare Triples")

- A ***correctness triple*** (a.k.a. "***Hoare triple***," after C.A.R. Hoare) is a program S plus its specification predicates p and q.
  - The ***precondition*** p describes what we're assuming is true about the state before the program begins.
  - The ***postcondition*** q describes what should be true about the state after the program terminates.
- ***Syntax of correctness triples***: $\{p\}\,S\,\{q\}$ (Think of it as `/* p */ S /* q */`)

  ⇒ ***Note: The braces are not part of the precondition or postcondition*** ⇐
- The precondition of $\{p\}\,S\,\{q\}$ is p, not $\{p\}$. Similarly the postcondition is q, not $\{q\}$.
  - Saying "$\{p\}$" is like saying "In C, the test in '`if (B) x++;`' is '`if (B)`'" instead of just `B`.

D. Satisfaction and Validity of a Correctness Triple

- Informally, for a state to ***satisfy*** $\{p\}\,S\,\{q\}$, it must be that if we run S in a state that satisfies p, then after running S, we should be in a state that satisfies q.
  - There's more than one way to understand "after running S", and this will give us two notions of satisfaction.

- ***Important***: If we start in a state that doesn't satisfy p, we claim nothing about what happens when you run S.
  - In some sense, "the triple is satisfied in σ" means "the triple is not buggy in σ", which seems like a rather weak claim.
  - However, "the triple is not satisfied in σ" means "the triple has a bug in σ", which is a pretty strong statement.
- For example, say you're given the triple $\{ x \geq 0 \}\ S\ \{ y^2 \leq x < (y+1)^2 \}$.
  - The triple claims that running the program when x is nonnegative sets y to the integer square root of x.
  - If you run it when x is negative, all bets are off: S could run and terminate with y = some value, it could diverge, it could produce a runtime error. None of these behaviors are bugs because you ran S on a bad input.
- ***Validity*** for correctness triples is analogous to validity of a predicate: The triple must be satisfied in every (well-formed, proper) state.
  - Say you (as the user) have been told not to run S when $x < 0$ because S calculates $sqrt(x)$.
  - And say the triple is $\{ x \geq 0 \}\ y := sqrt(x)\ \{ y^2 \leq x < (y+1)^2 \}$.
  - You can't say this program has a bug when you start in a state with $x < 0$, even though the program fails, because you ran the program on bad input.
- ***Notation:*** Analogous to our notation for predicates, for triples
  - $\sigma \vDash \{ p \}\ S\ \{ q \}$ means σ satisfies the triple.
  - $\sigma \nvDash \{ p \}\ S\ \{ q \}$ means σ does not satisfy the triple.
  - $\vDash \{ p \}\ S\ \{ q \}$ means the triple is valid.
  - $\nvDash \{ p \}\ S\ \{ q \}$ means the triple is invalid: $\sigma \nvDash \{ p \}\ S\ \{ q \}$ for some σ.

E. Simple Informal Examples of Correctness

- Before going to the formal definitions of partial and total correctness, let's look at some simple examples, informally. (As usual, we'll assume the variables range over $\mathbb{Z}$.)
- ***Example 1***: $\vDash \{ x > 0 \}\ x := x + 1\ \{ x > 0 \}$. The triple is valid: It's satisfied for all states where $x > 0$.
- ***Example 2***:
  - $\{ x = 1 \} \nvDash \{ x > 0 \}\ x := x - 1\ \{ x > 0 \}$: The triple is not satisfied (has a bug) when run with $x = 1$ because it terminates with $x = 0$, not > 0. Thus the triple is not valid: $\nvDash \{ x > 0 \}\ x := x - 1\ \{ x > 0 \}$.
- There are a number of ways to fix the buggy program in Example 2:
  - ***Example 3***: Make the precondition "***stronger***" = "more restrictive". For example, we could use $\vDash \{ x > 1 \}\ x := x - 1\ \{ x > 0 \}$.
  - ***Example 4***: Make the postcondition "***weaker***" = "less restrictive". For example, we could use $\vDash \{ x > 0 \}\ x := x - 1\ \{ x > -1 \}$.

- ***Example 5***: Change the program. One way is $\{x > 0\}$ **if** $x > 1$ **then** $x := x - 1$ **fi** $\{x > 0\}$.
- Let's have some more complicated examples.
- ***Example 6***: $\models \{x \geq 0 \wedge (x = 2 * k \vee x = 2 * k + 1)\}\ x := x/2\ \{x = k \geq 0\}$.
  - If x is nonnegative, then the program halves it with truncation.
- ***Example 7***: Assume $sum(0, k)$ yields the sum of the integers 0 through k, then
  $\models \{s = sum(0, k)\}\ s := s + k + 1;\ k := k + 1\ \{s = sum(0, k)\}$.
  - The triple says if $s = sum(0, k)$ when we start, then $s = sum(0, k)$ when we finish.
  - It's ok that s and k are changed by the program because $s = sum(0, k)$ is true in both places relative to the state at that point in time.
  - (Later, we'll use this program as part of a larger program, and we'll augment the conditions with information about how the ending values of k and s are larger than the starting values.)
  - Note we can write $s = 0 + 1 + 2 + \ldots + k$ as an informal equivalent of $s = sum(0, k)$, but it doesn't strictly have the form of a predicate as $s = sum(0, k)$ does.
- ***Example 8***: $\models \{s = sum(0, k)\}\ k := k + 1;\ s := s + k\ \{s = sum(0, k)\}$
  - This has the same specification as Example 7 but the code is different: It increments k first and then update s by adding k (not $k + 1$) to it.)
- ***Example 9***: [Note the invalidity] $\not\models \{s = sum(0, k)\}\ k := k + 1;\ s := s + k + 1\ \{s = sum(0, k)\}$
  - This is like Example 8 but the program doesn't meet its specification. To get validity, the postcondition should be $s = sum(0, k) + 1$. (Or more likely, the code needs to be fixed.)

F. Connecting Starting and Ending Values of Variables

- There are times when we want the postcondition to be able to refer to values that the variables started with.
- Recall Examples 7 and 8: $\models \{s = sum(0, k)\}\ S\ \{s = sum(0, k)\}$ (where S is different in the two examples). Say we want the postcondition to include "k gets larger by 1" somehow. What we can do is create a new variable (call it k_0) whose job it is to refer to the starting value of k, before we run S.
- We'll make the precondition $k = k_0 \wedge s = sum(0, k)$ ("k has some starting value and s is the sum of 0 through k"). We'll make the postcondition $k = k_0 + 1 \wedge s = sum(0, k)$ ("k is one larger than its starting value and s is the sum of 0 through k (for this new value of k)".
- [2023-02-07] We actually did the same thing in Example 6: $\models \{x \geq 0 \wedge (x = 2 * k \vee x = 2 * k + 1)\}\ x := x/2\ \{x = k \geq 0\}$. The variable k helps describe the value of x before and after execution. One interesting feature of k and k_0 is that they don't appear in the program, only the specifications. So where do variables appear in correctness triples?
- ***Definition:*** For a triple $\{p\}\ S\ \{q\}$,
  - A variable that appears in S is a ***program variable***. E.g., x is a program variable in $x := 1$. We manipulate them to get work done.

- A variable that appears in p or q is a ***condition variable***. E.g., y in $\{y > 0\}\ \ldots\ \{\ldots.\}$. We use condition variables to reason about our program. They may or may not also be program variables. (These are not the same kind of condition variables used in distributed programming.)
  - E.g., in $\{y > 0\}\ y := y + 1\ \{y > 1\}$, y is a program and a condition variable.
  - A ***logical variable*** is a condition variable that is not also a program variable. E.g., c in $\{z \geq c\}\ z := z + 1\ \{z > c\}$. We use them to reason about our program but they don't appear in the program itself. (Note that here, "logical" doesn't mean "Boolean".)
  - A ***logical constant*** is a named constant logical variable. E.g., c in the previous example. Logical constants are great for keeping track of an old value of a variable.
- ***Example 10***: $\models \{x = x_0 \geq 0\}\ x := x/2\ \{x_0 \geq 0 \wedge x = x_0/2\}$. If x is ≥ 0, then after the assignment $x := x/2$, the old value of x (which we're calling x_0) was ≥ 0 and x is its old value divided by 2. Here, x is a program and condition variable and $x_0$is a logical constant.

G. Having a Set of States that Satisfy a Predicate

- Before looking at the definitions of program correctness, it will help if we extend the notion of a single state satisfying a predicate to having a set of states satisfying a predicate.
- ***Notation***: Recall that $\Sigma_\perp = \Sigma \cup \{\perp\}$, where Σ is the set of all (well-formed, proper) states.
  - Then, $\sigma \in \Sigma_\perp$ allows $\sigma = \perp$, but $\sigma \in \Sigma$ implies $\sigma \neq \perp$.
  - Similarly for a set of states Σ_0, if $\Sigma_0 \subseteq \Sigma_\perp$, then we may have $\perp \in \Sigma_0$.
  - On the other hand, if $\Sigma_0 \subseteq \Sigma$, then $\perp \notin \Sigma_0$.
- ***Notation***: $\Sigma_0 - \perp$ means $\Sigma_0 \cap \Sigma$, the subset of Σ_0 containing its non-$\perp$ members.
- ***Definition***: Let $\Sigma_0 \subseteq \Sigma_\perp$. We say Σ_0 ***satisfies*** p if every element of Σ_0 satisfies p.
  - In symbols, $\Sigma_0 \models p$ iff for all $\tau \in \Sigma_0$, $\tau \models p$. It follows that $\Sigma_0 \not\models p$ iff $\tau \not\models p$ for some $\tau \in \Sigma_0$.
  - (Note $\varnothing \not\models p$ is clearly false, which means $\varnothing \models p$ is true.)
- Some consequences of the definition:
  - If $\perp \in \Sigma_0$, then $\Sigma_0 \not\models p$ and $\Sigma_0 \not\models \neg p$.
  - *($\Sigma_0 \models p$* and $\Sigma_0 \models \neg p$*)* iff $\Sigma_0 = \varnothing$.
    - Since $\perp \not\models p$ (and $\not\models \neg p$), we have $\perp \notin \Sigma_0$. If $\tau \neq \perp$ and $\tau \models p$ then $\tau \not\models \neg p$, so $\tau \notin \Sigma_0$. So $\Sigma_0 = \varnothing$.
  - If $\perp \notin \Sigma_0$ and Σ_0 is a singleton set (it has size = 1), then $\Sigma_0 \models p$ iff $\Sigma_0 \not\models \neg p$ (and $\Sigma_0 \models \neg p$ iff $\Sigma_0 \not\models p$). [2023-02-07]
    - Either $\tau \models p$ or $\tau \models \neg p$ but not both, so ($\tau \models p$ and $\tau \not\models \neg p$) or ($\tau \not\models p$ and $\tau \models \neg p$).
  - If $\Sigma_0 - \perp$ is not a singleton set then it is possible that $\Sigma_0 - \perp \not\models$ both p and $\neg p$.
    - Say we have $\sigma_1, \sigma_2 \in \Sigma_0 - \perp$ where $\sigma_1 \models p$ and $\sigma_2 \models \neg p$. For $\Sigma_0 - \perp \models p$, we need all its members to satisfy p, but that's false, so $\Sigma_0 - \perp \not\models p$. Similarly, $\Sigma_0 - \perp \not\models \neg p$ because not all members of $\Sigma_0 - \perp$ satisfy $\neg p$.

H. Total Correctness

- Normally, we want our programs to always terminate[1] in states satisfying their postcondition (assuming we start in a state satisfying the precondition). This property is called ***total correctness***.
- ***Definition***: The triple $\{p\}\,S\,\{q\}$ is ***totally correct in*** σ or σ satisfies the triple under ***total correctness*** iff it's the case that if σ satisfies p, then running S in σ always terminates in a state satisfying q.[2]
- In symbols, $\sigma \models_{\text{tot}} \{p\}\,S\,\{q\}$ iff $\sigma \neq \bot$ and $($if $\sigma \models p$ then $\bot \notin M(S,\sigma)$ and $M(S,\sigma) \models q)$.
  - Note $M(S,\sigma) \models q$ implies $\bot \notin M(S,\sigma)$, so it's redundant to say $\bot \notin M(S,\sigma)$ explicitly, but it's not a bad idea to emphasize it for a while.
  - We require $\sigma \neq \bot$ because we want the implication ($\sigma \models p$ implies $M(S,\sigma) \models q$) to be false when $\sigma = \bot$. Since $M(S,\bot) = \{\bot\} \not\models q$, if we allowed $\bot \models p$ then the implication would become true (since false implies false).
- ***Definition***: The triple $\{p\}\,S\,\{q\}$ is ***totally correct*** (is ***valid*** under ***total correctness***) iff $\sigma \models_{\text{tot}} \{p\}\,S\,\{q\}$ for all $\sigma \in \Sigma$ (Recall Σ is the set of well-formed proper states.) Usually, we'll write $\models_{\text{tot}} \{p\}\,S\,\{q\}$.

I. Partial vs Total Correctness

- It turns out that reasoning about total correctness can be broken up into two steps: Determine "partial" correctness, where we ignore the possibility of divergence or runtime errors, and then show termination -- i.e., that those errors won't occur.
- ***Definition***: The triple $\{p\}\,S\,\{q\}$ is ***partially correct in*** σ or σ ***satisfies the triple under partial correctness*** iff
  - $\sigma \neq \bot$ and
  - If σ satisfies p, then whenever running S in σ terminates (without error), the final state satisfies q.
- In symbols, $\sigma \models \{p\}\,S\,\{q\}$ iff $\sigma \neq \bot$ and $(\sigma \models p$ implies (for every $\tau \in M(S,\sigma)$, if $\tau \in \Sigma$, then $\tau \models q))$.
- Equivalently, $\sigma \models \{p\}\,S\,\{q\}$ iff $\sigma \neq \bot$ and $(\sigma \models p$ implies $M(S,\sigma) - \bot \models q)$.
  - It might help to point out that S not terminating under σ doesn't make partial correctness false.

---

[1] "Terminate" will mean "terminate without error" (Final state $\in \Sigma - \bot$). "Terminate possibly with an error" means we end in $\Sigma_\bot$.

[2] The sense of "implies" or "if... then..." used here is not like $\rightarrow$ (which appears in predicates) or $\Rightarrow$ (which is a relationship between predicates). It's "if...then" at a semantic level: If this triple is satisfied or if this set is nonempty, then ... holds.

- Note we must say explicitly that $\bot \not\models \{p\}S\{q\}$ because otherwise the general case would hold: $\bot \not\models p$ and $M(S,\sigma) - \bot = \{\bot\} - \bot = \varnothing \models q$, so the general case ($\sigma \models p$ implies $M(S,\sigma) - \bot \models q$) would be true (i.e., false implies false).
- ***Definition***: The triple $\{p\}S\{q\}$ is ***partially correct*** (i.e., is ***valid*** under/for ***partial correctness***) iff $\sigma \models \{p\}S\{q\}$ for all states σ. ***Notation***: We usually write $\models \{p\}S\{q\}$ but $\Sigma \models \{p\}S\{q\}$ is also ok.

J. More Phrasings of Total and Partial Correctness

- An equivalent way to understand partial and total correctness uses the property that if $\sigma \neq \bot$, then ($\sigma \models \neg p$ iff $\sigma \not\models p$) and ($\sigma \models p$ iff $\sigma \not\models \neg p$).
- For total correctness, just generally, if $\sigma \neq \bot$, then

  $\sigma \models_{\text{tot}} \{p\}S\{q\}$
  iff $\sigma \models p$ implies $M(S,\sigma) \models q$
  iff $\sigma \models \neg p$ or $M(S,\sigma) \models q$
  iff $\sigma \models \neg p$ or $\tau \models q$ for every member $\tau \in M(S,\sigma)$
- Under total correctness, if S is deterministic, then $M(S,\sigma) = \{\tau\}$ for some τ, with $\tau \neq \bot$ and $\tau \models q$. If S is nondeterministic, we can have multiple $\tau \in M(S,\sigma)$ and none of them can be $\bot$ [Mon 2023-02-06, 14:52] and all of them satisfy q.
- For partial correctness, if $\sigma \neq \bot$, then

  $\sigma \models \{p\}S\{q\}$
  iff $\sigma \models p$ implies $M(S,\sigma) - \bot \models q$
  iff $\sigma \models \neg p$ or $M(S,\sigma) - \bot \models q$
  iff $\sigma \models \neg p$ or for every $\tau \in M(S,\sigma)$, either $\tau = \bot$ or $\tau \models q$.
- Under partial correctness, if S is deterministic, then $M(S,\sigma) = \{\tau\}$ for some τ, and either $\tau = \bot$ or $\tau \models q$. If S is nondeterministic, we can have multiple $\tau \in M(S,\sigma)$ and all of them either are some version of $\bot$ or satisfy q.

K. Unsatisfied Correctness Triples

- It's useful to figure out when a state ***doesn't satisfy*** a triple because not satisfying a triple tells you that there's some sort of bug in the program.

Unsatisfied Total Correctness

- For a state $\sigma \neq \bot$ to not satisfy $\{p\}S\{q\}$ under total correctness, it must satisfy p and running S in it can cause an error or one of its final states does not satisfy q.
  - We have $\sigma \models_{\text{tot}} \{p\}S\{q\}$ iff $\sigma \models \neg p$ or $M(S,\sigma) \models q$
  - So $\sigma \not\models_{\text{tot}} \{p\}S\{q\}$ iff $\sigma \models p$ and $M(S,\sigma) \not\models q$
    iff $\sigma \models p$ and ($\bot \in M(S,\sigma)$ or $\tau \not\models q$ for some $\tau \in M(S,\sigma)$).
  - (Recall if $\tau \neq \bot$ then $\tau \not\models q$ iff $\tau \models \neg q$.)

- So breaking down the cases, $\sigma \models_{\text{tot}} \{p\}\, S\, \{q\}$ means
  - If S is deterministic, then $\sigma \models p$ and $M(S,\sigma) = \{\tau\}$ where $\tau = \bot$ or $\tau \models \neg q$.
  - If S is nondeterministic, then $\sigma \models p$ and $(\bot \in M(S,\sigma)$ or $\tau \models \neg q$ for some $\tau \in M(S,\sigma))$.
- Note for nondeterministic S, having $\sigma \not\models_{\text{tot}} \{p\}\, S\, \{q\}$ only says that one $\tau \in M(S,\sigma)$ is $\bot$ or satisfies $\neg q$. This doesn't preclude $M(S,\sigma)$ from having states that satisfy q.

Unsatisfied Partial Correctness

- For a state to not satisfy $\{p\}\, S\, \{q\}$ under partial correctness, either the state is $\bot$ or, it satisfies p and running S in it always terminates in a state satisfying $\neg q$.
  - We have $\sigma \models \{p\}\, S\, \{q\}$ iff $\sigma \models \neg p$ or $M(S,\sigma) - \bot \models q$
  - So $\sigma \not\models \{p\}\, S\, \{q\}$ iff $\sigma \models p$ and $M(S,\sigma) - \bot \not\models q$
    iff $\sigma \models p$ and $\tau \models \neg q$ for some $\tau \neq \bot$ in $M(S,\sigma)$.
  - For deterministic S, there's only one τ in $M(S,\sigma)$ and (it must be $\neq \bot$ and) satisfy $\neg q$.
  - For nondeterministic S, we need one $\tau \in M(S,\sigma)$, ($\tau \neq \bot$ and) $\tau \models \neg q$.
    - The other $\tau \in M(S,\sigma)$ can be $\bot$ or satisfy q.
    - I.e., at least one path $\langle S,\sigma\rangle \to^* \langle E,\tau\rangle$ with $\tau \models \neg q$, but there can be paths $\langle S,\sigma\rangle \to^* \langle E,\bot\rangle$ or $\langle S,\sigma\rangle \to^* \langle E,\tau\rangle$ with $\tau \models q$.

L. Three Extreme (Mostly Trivial) Cases

- There are three edge cases where partial correctness occurs for uninformative reasons.. First recall the definition of partial correctness: $\sigma \models \{p\}\, S\, \{q\}$ means (if $\sigma \models p$, then $M(S,\sigma) - \bot \models q$).
  - ***p is a contradiction*** (i.e., $\models \neg p$). Since $\sigma \models p$ never holds, $M(S,\sigma) - \bot \models q$ is irrelevant and partial correctness of $\{p\}\, S\, \{q\}$ always holds. So for example, $\{F\}\, S\, \{q\}$ is valid under partial correctness, for all S and q. (Even $\{F\}\, S\, \{F\}$ and $\{F\}\, S\, \{T\}$.)
  - ***S always fails to terminate***[3]. If $M(S,\sigma) = \{\bot\}$ then $M(S,\sigma) - \bot = \varnothing$, which satisfies q, so we get partial correctness of $\{p\}\, S\, \{q\}$.
  - ***q is a tautology*** (i.e., $\models q$). Then for any σ, $M(S,\sigma) - \bot \models q$, so ($\sigma \models p$ implies $M(S,\sigma) - \bot \models q$) is true (so p is irrelevant) and we get partial correctness of $\{p\}\, S\, \{q\}$. So for example, $\{p\}\, S\, \{T\}$ is valid under partial correctness for all p and S. (Even $\{F\}\, S\, \{T\}$.)
- For total correctness, recall $\sigma \models_{\text{tot}} \{p\}\, S\, \{q\}$ means (if $\sigma \models p$, then $M(S,\sigma) \models q$). Note $\bot \notin M(S,\sigma)$ because $\bot \notin M(S,\sigma)$ implies $M(S,\sigma) \not\models q$)
  - ***p is a contradiction***. The argument here is the same as for partial correctness, so for all S and q, we have $\models_{\text{tot}} \{F\}\, S\, \{q\}$.
  - ***S always fails to terminate***. Since $M(S,\sigma) = \{\bot\}$, we know $M(S,\sigma) \not\models q$. So total correctness of $\{p\}\, S\, \{q\}$ always fails. I.e., $\sigma \not\models_{\text{tot}} \{T\}\ S\, \{q\}$ for all σ. [2023-02-07]

---

[3] Remember, just "terminate" implicitly includes "without error". "Not terminate" means "Diverges or gets a runtime error".

  - ***q is a tautology***. This case is actually useful. Since $M(S, \sigma) \models T$ implies $\perp \notin M(S, \sigma)$, satisfaction of $\sigma \models_{\text{tot}} \{p\}\, S\, \{T\}$ requires S ***to always terminate*** under σ. So validity of $\models_{\text{tot}} \{p\}\ S\, \{T\}$ happens exactly when S always terminates when started in a state satisfying p.
- ***Lemma:*** $\sigma \models_{\text{tot}} \{p\}\, S\, \{q\}$ iff $\sigma \models \{p\}\, S\, \{q\}$ and $\sigma \models_{\text{tot}} \{p\}\, S\, \{T\}$.
  - This just says that total correctness is partial correctness plus termination.
  - Partial correctness says that $\langle S, \sigma \rangle \rightarrow^*$ to a final state that $\models q$ or is $\perp$). Termination says every $\langle S, \sigma \rangle \rightarrow^*$ to a final state that satisfies true (and thus $\neq \perp$)). So we have total correctness: Every $\langle S, \sigma \rangle \rightarrow^*$ to a final state that $\models q$.

Correctness ("Hoare") Triples

Part 2: Sequencing, Assignment, Strengthening, and Weakening

CS 536: Science of Programming, Spring 2023

2023-02-09 p.3

A. Why

- To specify a program's correctness, we need to know its precondition and postcondition (what should be true before and after executing it).
- The semantics of a verified program combines its program semantics rule with the state-oriented semantics of its specification predicates.
- To connect correctness triples in sequence, we need to weaken and strengthen conditions.

B. Objectives

At the end of today you should know

- Programs may have many different annotations, and we might prefer one annotation over another (or not), depending on the context.
- Under the right conditions, correctness triples can be joined together.
- One general rule for reasoning about assignments goes "backwards" from the postcondition to the precondition.
- What strength is and what weakening and strengthening are.

C. Examples of Partial and Total Correctness With Loops

- For the following examples, let $W \equiv$ ***while*** $k \neq 0$ ***do*** $k := k - 1$ ***od***.
  - ***Example 1:*** $\models_{\text{tot}} \{ k \geq 0 \}\ W\ \{ k = 0 \}$. If we start in a state with $k \geq 0$, the loop is guaranteed to terminate in a state satisfying $k = 0$.
  - ***Example 2:*** $\models \{ k = -1 \}\ W\ \{ k = 0 \}$ but $\not\models_{\text{tot}} \{ k = -1 \}\ W\ \{ k = 0 \}$. The triple is partially correct but not totally correct because it diverges if $k = -1$. I.e., we have $\not\models_{\text{tot}} \{ k = -1 \}\ W\ \{ T \}$. Also note that partial correctness would hold if we substitute any predicate for $k = 0$.
  - ***Example 3***: $\models \{ T \}\ W\ \{ k = 0 \}$ but $\not\models_{\text{tot}} \{ T \}\ W\ \{ k = 0 \}$. The triple is partially correct but not totally correct because it diverges for at least one value of k.
- For the following examples, let $W' \equiv$ ***while*** $k > 0$ ***do*** $k := k - 1$ ***od***. (We're changing the loop test of W so that it terminates immediately when k is negative.)
  - ***Example 4:*** $\models_{\text{tot}} \{ T \}\ W'\ \{ k \leq 0 \}$.

- ***Example 5:*** $\models_{\text{tot}} \{ k = c_0 \}\ W'\ \{ (c_0 \le 0 \to k = c_0) \wedge (c_0 \ge 0 \to k = 0) \}$. This is Example 4 with the "strongest" (most precise) postcondition possible. (In general, it's not always possible to find such a postcondition for loop, but it is here.)

D. More Correctness Triple Examples

Same Code, Different Conditions

- The same piece of code can be annotated with conditions in different ways, and there's not always a "best" annotation. An annotation might be the most general one possible (we'll discuss this concept soon), but depending on the context, we might prefer a different annotation.
- As before, let $sum(x, y)$ = the sum of x, $x+1$, $x+2$, ... y. (If $x > y$, let $sum(x, y) = 0$.) In Examples 9 – 12, we have the same program annotated (with preconditions and postconditions) of various strengths (strength = generality).
- ***Example 9:*** $\{ T \}\ i := 0\ ;\ s := 0\ \{ i = 0 \wedge s = 0 \}$.
  - This is the strongest (most precise) annotation for this program.
- ***Example 10:*** $\{ T \}\ i := 0\ ;\ s := 0\ \{ i = 0 \wedge s = 0 = sum(0, i) \}$.
  - This adds a summation relationship to i and s when they're both zero.
- ***Example 11:*** $\{ n \ge 0 \}\ i := 0\ ;\ s := 0\ \{ 0 = i \le n \wedge s = 0 = sum(0, i) \}$
  - This limits i to a range of values 0, ..., n. We have to include $n \ge 0$ in the precondition if we want to claim $n \ge 0$ in the postcondition.
- ***Example 12:*** $\{ n \ge 0 \}\ i := 0; s := 0\ \{ 0 \le i \le n \wedge s = sum(0, i) \}$
  - The postcondition no longer includes i and s being zero, so this postcondition is weaker (less precise) than the postcondition for Example 11. This might seem like a disadvantage but will turn out to be an advantage later.
- The next two examples relate to calculating the midpoint in binary search. Though the code is the same, whether the midpoint is strictly between the left and right endpoints depends on whether or not the endpoints are nonadjacent.
  - ***Example 13:*** $\{ lt < rt \wedge lt \ne rt - 1 \}\ mid := (lt + rt) / 2\ \{ lt < mid < rt \}$
  - ***Example 14:*** $\{ lt < rt \}\ mid := (lt + rt) / 2\ \{ lt \le mid < rt \}$
- In Examples 13 and 14, the differences in the postcondition have an effect on how to detect that the value being searched for doesn't exist. In Example 13, it's $lt = rt - 1$; in Example 14, it's $lt > rt$.

Use of DeMorgan's Laws

- When a loop terminates, we know that the negation of the loop test holds, so DeMorgan's laws can be useful. Similarly, for a condition, we know that the negation of the test holds just before we execute the false branch.
- ***Example 15:*** Here we search downward for $x \ge 0$ such that $f(x)$ is $\le y$; we stop if we find such an x or if we run out of values to test.

$\{ x \geq 0 \}$
while $x \geq 0 \wedge f(x) > y$ ***do*** $x := x - 1$ ***od***
$\{ x < 0 \vee f(x) \leq y \}$ [2023-02-08 typo] *// Negation of loop test*

- ***Example 16:*** This is Example 15 rephrased as an array search; we search to the left for an index k such that $b[k] \leq y$; we stop if we find one or run out of indexes to test[1].

  $\{ k \geq 0 \}$
  while $k \geq 0 \wedge b[k] > y$ ***do*** $k := k - 1$ ***od***
  $\{ k < 0 \vee b[k] \leq y \}$ *// Negation of loop test*

Joining Two Triples

- To make two statements a sequence, we have to compare the postcondition of the first statement and the precondition of the second. If they're the same, we can make the join.
  - I.e., if we have $\{p\} S_1 \{q\}$and $\{q\} S_2 \{r\}$, then we can form $\{p\} S_1 ; S_2 \{r\}$ because when S_1 finishes executing, it will satisfy the precondition of S_2.
- ***Example 17:*** We can join these two statements because the postcondition of the first statement matches the precondition of the second. (Note though $s = sum(0, k)$ holds before and after the two assignments, it doesn't hold between.)

  Combining $\{ s = sum(0,k) \}\ s := s + k + 1\ \{ s = sum(0, k+1) \}$
  and $\{ s = sum(0,k+1) \}\ k := k + 1\ \{ s = sum(0,k) \}$
  yields $\{ s = sum(0,k) \}\ s := s + k + 1\ ;\ k := k + 1\ \{ s = sum(0,k) \}$
- ***Example 18:*** Alternatively, we can increment k first and then update s.

  Combining $\{ s = sum(0,k) \}\ k := k + 1 \{ s = sum(0, k-1) \}$
  and $\{ s = sum(0,k-1) \}\ s := s + k \{ s = sum(0,k) \}$
  yields $\{ s = sum(0,k) \}\ k := k + 1\ ;\ s := s + k \{ s = sum(0,k) \}$

Reasoning About Assignments (Technique 1: "Backward")

- There are two general rules for reasoning about assignments.
- The first rule is a goal-directed one that works "backwards", from the postcondition to the precondition.
- ***Assignment Rule 1 ("Backward" assignment:*** If $P(x)$ is a predicate function, then $\{P(e)\}\ v := e\ \{P(v)\}$. It turns out that $P(e)$ is the most general (the so-called "weakest") precondition that works with the assignment $v := e$ and postcondition $P(v)$. We'll study this in the next lecture.
- ***Example 19:*** $\{P(m/2)\}\ m := m/2\ \{P(m)\}$
  - If $P(x) \equiv x > 0$, then this triple expands to $\{ m/2 > 0 \}\ m := m/2\ \{ m > 0 \}$.
  - If $P(x) \equiv x^2 > x^3$, then this triple expands to $\{ (m/2)^2 > (m/2)^3 \}\ m := m/2\ \{ m^2 > m^3 \}$.
- ***Example 20:*** $\{Q(k+1)\}\ k := k + 1\ \{Q(k)\}$

---

[1] Don't think I've said before that we can make $p \wedge q$ short-circuiting by using ***if*** p ***then*** T ***else*** q ***fi***.

  - If $Q(x) \equiv s = sum(0, x)$ then this triple expands to
    $\{s = sum(0, k+1)\}\ k := k+1\ \{s = sum(0, k)\}$.
- ***Example 21:*** $\{R(s + k + 1)\}\ s := s + k + 1\ \{R(s)\}$
  - If $R(x) \equiv x = sum(0, k+1)$, then this triple expands to
    $\{s + k + 1 = sum(0, k+1)\}\ s := s + k + 1\ \{s = sum(0, k+1)\}$

- In general, for $\{P(e)\}\ v := e\ \{P(v)\}$ to be valid, we need the following lemma:
- ***Assignment Lemma:*** For all σ, if $\sigma \models P(e)$ then $M(v := e, \sigma) = \sigma[v \mapsto \sigma(e)] \models P(v)$.
  - Intuitively, what this says is that if we want to know that v has property P after binding v to the value of e, we need to know that e has property P beforehand.
  - We won't go into the detailed proof of this lemma, but basically, you work recursively on the structures of $P(v)$ and $P(e)$ simultaneously. The important case is when we encounter an occurrence of v in $P(v)$ and the corresponding occurrence of e in $P(e)$. In $\sigma \models P(e)$, the value of e is $\sigma(e)$. In $\sigma[v \mapsto \sigma(e)] \models P(v)$, the value of v is also $\sigma(e)$.

E. Stronger and Weaker Predicates

- ***Generalizing the Sequence Rule:*** We've already seen that two triples $\{p\}\ S_1\ \{q\}$and $\{q\}\ S_2\ \{r\}$ can be combined to form the sequence $\{p\}\ S_1; S_2\ \{r\}$.
- Say we want to combine two triples that don't have a common middle condition, $\{p\}\ S_1\ \{q\}$ and $\{q'\}\ S_2\ \{r\}$.
  - We can do this iff $q \rightarrow q'$. If S_1 terminates in state τ and $\{p\}\ S_1\ \{q\}$ is valid, then $\tau \models q$. If $\models q \rightarrow q'$, then $\tau \models q'$, so if $\{q'\}\ S_2\ \{r\}$, then if running S_2 in τ terminates, it terminates in a state satisfying r.
  - This reasoning works both for partial and total correctness.
- Our earlier discussion used a special case of the above. When $q \equiv q'$, we get that $\{p\}\ S_1\ \{q\}$ and $\{q\}\ S_2\ \{r\}$ can be joined.
- ***Definition:*** If $p \rightarrow q$ then p is ***stronger than*** q and q is ***weaker than*** p. I.e., the states that satisfy p also satisfy q.
  - (Technically, we should say "stronger than or equal to" and "weaker than or equal to," because if $p \leftrightarrow q$, then p is stronger and weaker than q and vice versa. But it's too much of a mouthful.
- ***Definition***: p is ***strictly stronger*** than q and q is ***strictly weaker*** than p if $(p \rightarrow q) \wedge \neg(q \rightarrow p)$.
- ***Example 22***: $x = 0$ is (strictly) stronger than $x = 0 \vee x = 1$, which is (strictly) stronger than $x \geq 0$.

Predicates and Venn Diagrams

- You can view a predicate as standing for the set of states that satisfy it. In that case, $p \rightarrow q$ means that the set of states for p is $\subseteq$ the set of states for q.[2]
- ***Notation:*** Sometimes we'll abbreviate "*the set of states satisfying p*" to just "*p*".
- Venn diagrams with sets of states can help illustrate comparisons of predicate strength.
- In Figure 1, $p \rightarrow q$ because the set of states for p is $\subseteq$ the set of states for q. It's less obvious, but the contrapositive $\neg q \rightarrow \neg p$ also holds.
  - r has an intersection with q; the part inside q is $q \wedge r$; the part outside q is $\neg q \wedge r$.
  - p has no intersection with r, so neither $p \rightarrow r$ nor $r \rightarrow p$ hold, but all of p is outside r, so $p \rightarrow \neg r$ (and $r \rightarrow \neg p$).

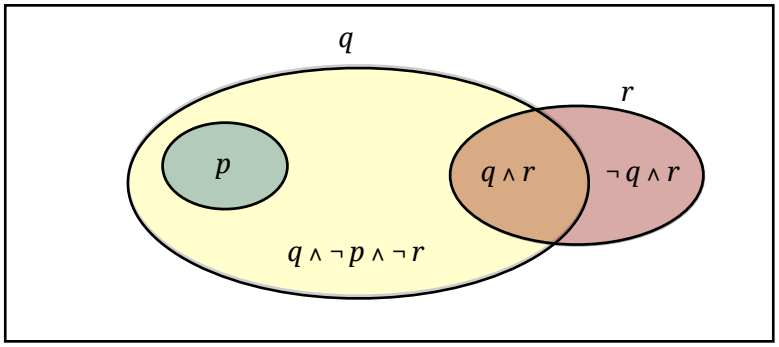


Figure 1: Predicates As Standing for Sets of States

F. Strengthening and Weakening Conditions

- The relationship between stronger and weaker predicates allows us to make certain changes to the conditions of triples "for free" (i.e, we just have to know one implies the other) Both of the following properties are valid for both partial and total correctness of triples (i.e., for $\vDash$ and $\vDash_{\text{tot}}$).
- The "always" below means that given an appropriate correctness triple it's sufficient to know $s \rightarrow b$, (so that s and b have *smaller* and *bigger* sets of satisfying states).
  - ***Preconditions can always be strengthened***: If $s \rightarrow b$ and $\{ b \} S \{ q \}$, then $\{ s \} S \{ q \}$.
  - ***Postconditions can always be weakened***: If $s \rightarrow b$ and $\{ p \} S \{ s \}$, then $\{ p \} S \{ b \}$.
- With p and q in Figure 1, $p \rightarrow q$, so p is stronger than q.
  - So if q were the precondition of a triple, it could be strengthened to p. We can't strengthen q to r, but we can strengthen it to $q \wedge r$.
  - In the other direction, a postcondition of p or $q \wedge r$ can be weakened to q.

---

[2] One very old notation for implication is $p \supset q$; it's important not to read that $\supset$ as a superset symbol, since $p \rightarrow q$ means that the set of states for p is $\subseteq$ the set of states for q.

- ***Example 23***: If $\{ x \geq 0 \}\ S\ \{ y = 0 \}$ is valid, then so are
  - $\{ x \geq 0 \}\ S\ \{ y = 0 \vee y = 1 \}$ *Weakened postcondition*
  - $\{ x = 0 \}\ S\ \{ y = 0 \}$ *Strengthened precondition*
  - $\{ x = 0 \}\ S\ \{ y = 0 \vee y = 1 \}$ *Strengthened precondition and weakened postcondition*

- ***Example 24***: Since $s = sum(0, k) \Leftrightarrow s + k + 1 = sum(0, k+1)$, we can "strengthen" the precondition of $\{ s + k + 1 = sum(0, k+1) \}\ k := k + 1\ \{ q \}$ and get $\{ s = sum(0, k) \}\ k := k + 1\ \{ q \}$. (I put the "strengthen" in quotes here to point out that since the two conditions are $\Leftrightarrow$, we can also do "strengthening" in the other direction, going from $s = sum(0, k)$ to $s + k + 1 = sum(0, k+1)$.)

Limitations of Strengthening and Weakening

- If $p \rightarrow q$ says that the sets of states satisfying p and q respectively are $\subseteq$, then the implications $q \rightarrow q_1$, $q_1 \rightarrow q_2$, $q_2 \rightarrow q_3$, etc. form a sequence of weaker and weaker predicates because the sets of states get larger and larger. There is a limit, namely, Σ, the set of all states. As a set of states, Σ corresponds to T (true) and is the weakest possible state.
- Similarly, going right to left, the implications ...., $p_3 \rightarrow p_2$, $p_2 \rightarrow p_1$, $p_1 \rightarrow p$ form a sequence of stronger and stronger predicates because their sets of states get smaller and smaller. Here, the limit is the empty set $\varnothing$, the set of states that correspond to F (false), making it the strongest possible state.
- Having F be strongest and T be weakest may be counterintuitive. It may help if you think of the strength of a predicate being the amount of constraints it puts on the set of the states that satisfy it. The strongest predicate, F has contradictory constraints on it, hence is satisfied by no state. The weakest predicate, T, has no constraints on it, hence it is satisfied by every state. In sets of states notation, $\{\ \} \models F$ and $\{\Sigma\} \models T$.

Can vs Should

- Just because we ***can*** strengthen preconditions and weaken postconditions doesn't mean we ***should***. Recall our edge cases for satisfying correctness triples:
  - $\sigma \models \{ F \}\ S\ \{ q \}$ and $\sigma \models_{\text{tot}} \{ F \}\ S\ \{ q \}$ have the strongest possible preconditions.
  - $\sigma \models \{ p \}\ S\ \{ T \}$ has the weakest possible postcondition.
  - $\sigma \models_{\text{tot}} \{ p \}\ S\ \{ T \}$ says that S terminates when you start it in p but it says nothing about what the state looks like when it terminates.
- From the programmer's point of view, if $\{ p \}\ S\ \{ q \}$ has a bug, then strengthening p or weakening q can get rid of the bug without changing S.
  - ***Example 25 (Strengthening the precondition)***
    - If $\{ p \}\ S\ \{ q \}$ causes an error if $x = 0$, we can tell the user to use $\{ p \wedge x \neq 0 \}\ S\ \{ q \}$.
  - ***Example 26 (Weakening the postcondition)***
    - If $\{ p \}\ S\ \{ q \}$ causes an error because S terminates satisfying predicate r (where r doesn't imply q) then we can tell the user to use $\{ p \}\ S\ \{ q \vee r \}$.

Weaker Preconditions and Stronger Postconditions

- From the user's point of view, weaker preconditions and stronger postconditions make triples more useful.

Weaker Preconditions

- Weaker preconditions make code more applicable by increasing the set of starting states.
- Say $\{ p \} S \{ r \}$ is valid. If $q \to p$, then weakening the precondition to get $\{ q \} S \{ r \}$ gives the programmer more flexibility in what states to start.
- Unlike strengthening a precondition, however, weakening a precondition requires work because we need to show that the states in q that are not in p also work as precondition states.
  - In symbols, if we show $\{ q \wedge \neg p \} S \{ r \}$, then since we already know $\{ p \} S \{ r \}$, we can $\vee$ the preconditions. We get $p \vee (q \wedge \neg p) \Leftrightarrow (p \vee q) \wedge (p \vee \neg p) \Leftrightarrow (p \vee q) \wedge T \Leftrightarrow p \vee q$.
  - Then we can take precondition $p \vee q$ and strengthen it to just q, so $\{ q \} S \{ r \}$.

Stronger Postconditions

- Stronger postconditions make code more specific by decreasing the set of ending states.
- Say $\{ r \} S \{ q \}$ is valid. if $q \to p$, then we could strengthen postcondition q to get $\{ r \} S \{ p \}$.
- But we can't do this unless we know that running S gets us to the part of q that is inside p, (and never to the part of q outside p).
  - If we show $\{ r \} S \{ \neg (q \wedge \neg p) \}$, then we can $\wedge$ that with postcondition p and get $q \wedge \neg (q \wedge \neg p) \Leftrightarrow q \wedge (\neg q \vee p) \Leftrightarrow (q \wedge \neg q) \vee (q \wedge p) \Leftrightarrow q \wedge p$.
  - Then we can weaken postcondition $p \wedge q$ to just p.

Weakest Preconditions

Part 1: Definitions and Basic Properties

CS 536: Science of Programming, Spring 2023

A. Why

- Weakest liberal preconditions (wlp) and weakest preconditions (wp) are the most general requirements that a program must meet to be correct.

B. Objectives

At the end of today you should understand

- What wlp and wp are and how they are related to preconditions in general.

Part 1: The Deterministic Case

C. Weakening the Precondition of $\models_{tot} \{p\}S\{q\}$

- Let's assume that S is deterministic. Figure 1 illustrates how $\models_{\text{tot}} \{p\}S\{q\}$ works: If you take any state in p and follow the arrow by applying S, you end in a state that satisfies q.
  - (To illustrate partial correctness, we would add arrows from p to $\neg q$ or to $\perp$.)
- The predicate r intersects p, so states within $p \wedge r$ are guaranteed to lead (via S) to states in q.
- States in $\neg p \wedge r$ might lead via S to p or $\neg p$ or to $\perp$, but if all of them lead to p, then we could extend our precondition p and we'd have $\{p \vee \neg p \wedge r\}S\{q\}$, which simplifies to $\{p \vee r\}S\{q\}$.
  - A shorter way to say this is if $\models_{\text{tot}} \{p\}S\{q\}$ and $\models_{\text{tot}} \{\neg p \wedge r\}S\{q\}$, then $\models_{\text{tot}} \{p \vee r\}S\{q\}$.
  - Of course, in general, we don't know $\models_{\text{tot}} \{\neg p \wedge r\}S\{q\}$, but if we can prove it, we can weaken the precondition p to r, which provides the user with more flexibility for running S. But sometimes, we can't weaken the precondition any more than it is already.
- ***Definition: w is the weakest precondition of S and q*** (we write $w = wp(S, q)$) if w is a precondition that can't be weakened. I.e., $\models_{\text{tot}} \{w\}S\{q\}$ and there is no r strictly stronger than w such that $\models_{\text{tot}} \{r\}S\{q\}$.
  - The converse holds, since it's just precondition strengthening: If $\models_{\text{tot}} \{w\}S\{q\}$, then knowing $r \rightarrow w$ lets us conclude (is ***sufficient*** for) $\models_{\text{tot}} \{r\}S\{q\}$.
  - Being the weakest precondition makes $r \rightarrow w$ a ***necessary*** condition for $\models_{\text{tot}} \{r\}S\{q\}$.
  - So if w is the weakest precondition, then $\{r\}S\{q\}$ iff $w \rightarrow r$.
  - In terms of states, $wp(S, q) = \{\sigma \in \Sigma \mid M(S, \sigma) \models q\}$

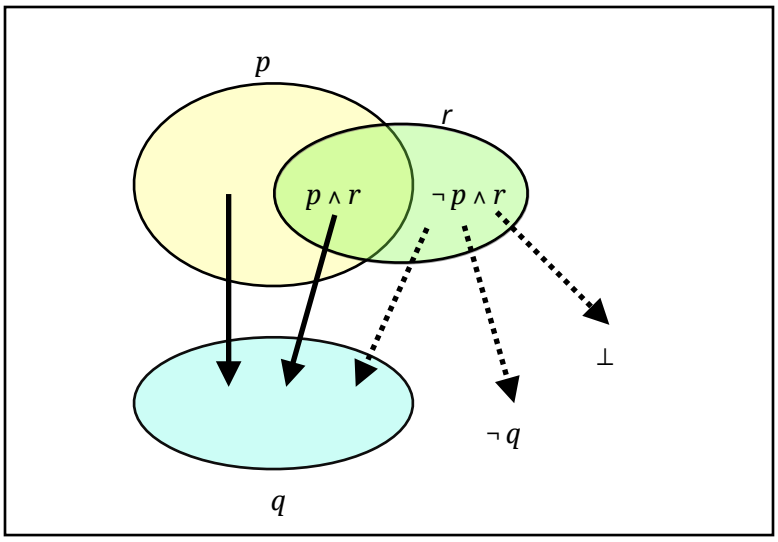


Figure 1: Extending Precondition of {p}S{q}

- Recall that in general, $\models_{\text{tot}} \{ p \} S \{ q \}$ doesn't tell us anything about $M(S, \sigma)$ if $\sigma \not\models p$. But if p is weakest, we know $M(S, \sigma) \not\models q$.
  - For deterministic programs, we can state this using partial correctness: If $w = wp(S, q)$ and S is deterministic then $\models \{ \neg w \} S \{ \neg q \}$. If $\sigma \models \neg w$ then $M(S, \sigma) = \{ \tau \}$ where $\tau = \bot$ or $\tau \models \neg q$.
- Writing $wp(S, q)$ as a predicate is convenient, but technically the weakest precondition of S under q is a set of states (the set of all states that are preconditions of S and q under total correctness). As sets, there are $wp(S, q)$ that don't correspond well to writable predicates, and in those cases we'll have to write predicates that approximate $wp(S, q)$.
- Usually, we talk about "the" $wp(S, q)$, but as a predicate, a wp is unique only "up to logical equivalence": If $u \Leftrightarrow w$, then u is also a wp. For example, if the $wp(S, q)$ is $x > 0$, then $x \geq 1$ and $0 < x$ and so on are also wp's.
- Later we'll see a syntactic algorithm that helps us calculate some wp's; in those cases, we'll write $wp(S, q) \equiv w$ where w is the syntactic representation produced by the algorithm.

D. The Weakest Liberal Precondition, wlp

- The ***weakest liberal precondition*** is analogous to the wp but for partial correctness instead of total correctness.
- ***Definition***: The ***weakest liberal precondition*** for S and q, written $wlp(S, q)$, is a valid precondition for q under partial correctness where no strictly weaker valid precondition exists.
  - In symbols, $w = wlp(S, q)$ iff $\models \{ w \} S \{ q \}$ and for all u, $\models \{ u \} S \{ q \}$ if and only if $\models u \rightarrow w$.
  - In terms of states, $wlp(S, q) = \{ \sigma \in \Sigma \mid M(S, \sigma) - \bot \models q \}$.

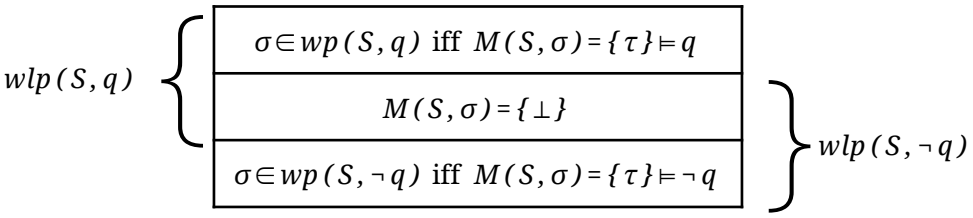


Figure 2: The Weakest Liberal Precondition for Deterministic S

Relationships Between wp and wlp

- Figure 2 illustrates the relationships between *wp* and *wlp* for deterministic programs.
- The top third shows the states in $wp(S, q)$: Those states in $M(S, \sigma)$ satisfy q.
- The bottom third shows the states in $wp(S, \neg q)$: Those states in $M(S, \sigma)$ satisfy $\neg q$.
- The middle third shows that states that cause nontermination.
  - Adding the nonterminating states to $wp(S, q)$ gives $wlp(S, q)$.
  - Adding the nonterminating states to $wp(S, \neg q)$ gives $wlp(S, \neg q)$.
  - Subsequently, $\neg\, wp(S, \neg q) \Leftrightarrow wlp(S, q)$ and $\neg\, wp(S, q) \Leftrightarrow wlp(S, \neg q)$.
- The relationship: $wlp(S, q) \wedge wlp(S, \neg q)$ describes the states that cause nontermination.

Why Are wp and wlp Important?

- The reason *wp* and *wlp* are important is that if you have a precondition and can show that it's the weakest precondition, you have the most general solution to "What states can I start in and successfully end in q?
  - With *wp*, "successfully end" means "terminates satisfying q". With *wlp*, it means "if we terminate, we terminate satisfying q".
- The solution is most general in the sense that any state not satisfying the *wp* or *wlp* is guaranteed to ***not*** successfully end in q.
- Compare with non-weakest preconditions, where starting in a state not satisfying the precondition might end successfully or end not successfully (satisfying $\neg q$) or not terminate.

E. Examples of wp and wlp

- ***Example 1***: The assignment $y := x * x$ always terminates, so *wp* and *wlp* behave identically on it. $wp(y := x * x, x \geq 0 \wedge y \geq 4) \Leftrightarrow wlp(y := x * x, x \geq 0 \wedge y \geq 4) \Leftrightarrow x \geq 2$.
- ***Example 2***: The *wp* and *wlp* of ***if*** $y \leq x$ ***then*** $m := x$ ***else skip fi*** and $m = max(x, y)$ are $(y > x \rightarrow m = y)$.
  - Later, we'll see how to calculate the *wp* in this instance, but for now, let's look at it intuitively. The true branch sets up the postcondition when $y \leq x$. The false branch (the implicit ***else skip***) runs when $y > x$ and doesn't change the state, so we need the postcondition m = y to already be satisfied.

- ***Example 3***: The weakest precondition of ***while*** $x \neq 0$ ***do*** $x := x - 1$ ***od*** and $x = 0$ is $x \geq 0$. Starting with $x \geq 0$ terminates with $x = 0$. Starting with $x < 0$ doesn't terminate.
  - The *wlp* of the loop and postcondition is simply T. Since we're ignoring termination, the body of the loop doesn't affect the fact that for ***while*** $x \neq 0$ ... to exit, x must be zero.
  - Our loop terminates iff run with $x \geq 0$, so if W is our loop, then $wp(W, T) \Leftrightarrow x \geq 0$.
  - We can verify $x \geq 0 \Leftrightarrow wp(W, x = 0) \Leftrightarrow wlp(W, x = 0) \wedge wp(W, T) \Leftrightarrow T \wedge x \geq 0 \Leftrightarrow x \geq 0$.
- ***Example 4***: The weakest precondition of $W \equiv$ ***while*** $x > 0$ ***do*** $x := x - 1$ ***od*** and $x \leq 0$ is T (true). Again, starting with $x \geq 0$ terminates with $x = 0$. We can terminate with any negative value for x simply by running the loop with that value; the loop terminates immediately without changing x.
  - Since $T \Leftrightarrow wp(W, x \leq 0) \Leftrightarrow wlp(W, x \leq 0) \wedge wp(W, T)$, both $wlp(W, x \leq 0)$ and $wp(W, T) \Leftrightarrow T$. Semantically, we can also justify this by arguing that ***while*** $x > 0$ ... terminates immediately iff $x \leq 0$.
- ***Example 5***: For any S and σ, either we terminate (in a state satisfying true) or we don't terminate. Therefore $wlp(S, T) \Leftrightarrow T$. Also, since $wlp(S, T) \Leftrightarrow \neg wp(S, \neg T) \Leftrightarrow T$, we have that $wp(S, F) \Leftrightarrow F$. (In Figure 2 terms, the bottom third of the diagram is empty because running S in σ never terminates in a state satisfying false.)

Part 2: The Nondeterministic Case

- With nondeterministic programs, *wp* and *wlp* are more complicated (of course). The basic definitions are the same:
  - $\sigma \in wp(S, q)$ iff $M(S, \sigma) \models q$ or equivalently $\models_{tot} \{p\} S \{q\}$ iff $\models wp(S, q) \rightarrow p$.
  - $\sigma \in wlp(S, q)$ iff $M(S, \sigma) - \perp \models q$ or equivalently $\models \{p\} S \{q\}$ iff $\models wlp(S, q) \rightarrow p$.
- Let $\Sigma_0 = M(S, \sigma)$ or $M(S, \sigma) - \perp$ depending on whether we're discussing *wp* or *wlp*.
- Since Σ_0 satisfies q iff every individual state in Σ_0 satisfies q, nonsatisfaction only requires one counterexample state:
  - $\sigma \notin wp(S, q)$ iff for some $\tau \in M(S, \sigma)$, we have $\tau = \perp$ or $\tau \not\models q$ (and since τ is a state, $\tau \models \neg q$).
  - $\sigma \notin wlp(S, q)$ iff for some $\tau \in M(S, \sigma)$, we have $\tau \not\models q$ (and since τ is a state, $\tau \models \neg q$).
- But there are no constraints on other members of Σ_0, so $\sigma \notin wp(S, q)$ and $\sigma \notin wlp(S, q)$ are both compatible with having $\tau \in M(S, \sigma)$ with $\tau \models q$.

F. Properties of wp and wlp for Deterministic and Nondeterministic Programs

- There are a number of properties connecting the *wp*, *wlp*, $\neg wp$, and $\neg wlp$ of q and $\neg q$.
- Some properties are common to both deterministic and nondeterministic programs:
  1. $M(S, \sigma) = \{\perp\} \Rightarrow wlp(S, q) \wedge wlp(S, \neg q)$

- $M(S,\sigma) - \bot = \emptyset$, so it $\models q$ and $\models \neg q$, so $\sigma \models wlp(S,q) \wedge wlp(S,\neg q)$.

2. $M(S,\sigma) = \{\bot\} \Rightarrow \neg wp(S,q) \wedge \neg wp(S,\neg q)$
   - $M(S,\sigma) = \{\bot\} \not\models q$ and $\not\models \neg q$, so $\sigma \models \neg wp(S,q) \wedge \neg wp(S,\neg q)$.
3. $wlp(S,q) \wedge wlp(S,\neg q) \Rightarrow M(S,\sigma) = \{\bot\}$
   - For $\sigma \models wlp(S,q) \wedge wlp(S,\neg q)$, we must have $M(S,\sigma) - \bot \models q \wedge \neg q$. Since no actual state satisfies $\models q \wedge \neg q$, that implies that $M(S,\sigma) - \bot = \emptyset$, so $M(S,\sigma) = \{\bot\}$.
4. $wp(S,q) \Rightarrow wlp(S,q)$
   - If $\sigma \models wp(S,q)$, then $M(S,\sigma) \models q$, so $M(S,\sigma) - \bot \models q$, and so $\sigma \models wlp(S,q)$.
5. $wlp(S,q) \Rightarrow \neg wp(S,\neg q)$
   - If $\sigma \models wlp(S,q)$, then $M(S,\sigma) - \bot \models q$, so for all $\tau \in M(S,\sigma) - \bot$, we have $\tau \not\models \neg q$. If $\bot \in M(S,\sigma)$ then $M(S,\sigma) \not\models \neg q$, so $\tau \not\models \neg q$.
6. $wp(S,q) \Rightarrow \neg wlp(S,\neg q)$
   - If σ is in $wp(S,q)$ then $M(S,\sigma) \models q$. For σ to be in $wlp(S,\neg q)$, we need every $\tau \in M(S,\sigma)$ to be either $\bot$ or to satisfy $\neg q$. But every $\tau \in M(S,\sigma)$ satisfies q, so $\tau \neq \bot$ and τ doesn't satisfy $\neg q$. So if σ is in $wp(S,q)$, it's not in $wlp(S,\neg q)$, it's in $\neg wlp(S,\neg q)$.

- There are also properties that hold for deterministic programs but not nondeterministic programs.

7a. If S is deterministic, then $\neg wp(S,q) \wedge \neg wp(S,\neg q) \Rightarrow M(S,\sigma) = \{\bot\}$.
   - For deterministic S, we know $M(S,\sigma) =$ some $\{\tau\}$, where $\tau = \bot$, $\tau \models q$, or $\tau \models \neg q$. But $\sigma \models \neg wp(S,q) \wedge \neg wp(S,\neg q)$ implies that $M(S,\sigma) \not\models q$ and $M(S,\sigma) \not\models \neg q$, which leaves $M(S,\sigma) = \{\bot\}$ as the only possibility.

7b. If S is nondeterministic, then $\neg wp(S,q) \wedge \neg wp(S,\neg q)$ doesn't imply $M(S,\sigma) = \{\bot\}$.
   - For a nondeterministic program, if $M(S,\sigma) \not\models q$ and $M(S,\sigma) \not\models \neg q$, it's still possible for $M(S,\sigma)$ to contain non-$\bot$ states. A simple counterexample is $M(S,\sigma) = \{\tau_1, \tau_2\}$ where $\tau_1 \models q$ and $\tau_2 \models \neg q$. Note it's possible that $\bot \notin M(S,\sigma)$, which definitely makes $M(S,\sigma)$ unequal to $\{\bot\}$.

8a. If S is deterministic, then $\neg wp(S,q) \Rightarrow wlp(S,\neg q)$.
   - $M(S,\sigma) = \{\tau\}$ where $\tau = \bot$, $\tau \models q$, or $\tau \models \neg q$. If $\sigma \models \neg wp(S,q)$, then $\tau \models q$ fails, which leaves $\tau = \bot$ or $\tau \models \neg q$, in which case $M(S,\sigma) - \bot \models \neg q$, so $\sigma \models wlp(S,\neg q)$.

8b. If S is nondeterministic, then $\neg wp(S,q)$ doesn't imply $wlp(S,\neg q)$.
   - Since $M(S,\sigma) \not\models q$ says only that not every value in $M(S,\sigma)$ satisfies q, so there can still be a $\tau_1 \in M(S,\sigma)$ with $\tau_1 \models q$, in which case $\sigma \not\models wlp(S,\neg q)$.

G. Disjunctive Postconditions Under Nondeterminism are Different

- For deterministic and nondeterministic both, the *wp*/*wlp* of a conjunction is the same as the conjunction of the *wp*/*wlp* 's.
   - $wp(S,q_1) \wedge wp(S,q_2) \Leftrightarrow wp(S,q_1 \wedge q_2)$.

  - $wlp(S, q_1) \wedge wlp(S, q_2) \Leftrightarrow wlp(S, q_1 \wedge q_2)$.
- Also, the disjunction of the *wp/wlp* 's is sufficient to imply the *wp/wlp* of the disjunction:
  - $wp(S, q_1) \vee wp(S, q_2) \Rightarrow wp(S, q_1 \vee q_2)$.
  - $wlp(S, q_1) \vee wlp(S, q_2) \Rightarrow wlp(S, q_1 \vee q_2)$.
- Necessity of the *wp/wlp* of the disjunction holds for deterministic programs:
  - $wp(S, q_1 \vee q_2) \Rightarrow wp(S, q_1) \vee wp(S, q_2)$.
  - $wlp(S, q_1 \vee q_2) \Rightarrow wlp(S, q_1) \vee wlp(S, q_2)$.
- But for nondeterministic programs, $wp(S, q_1 \vee q_2) \Rightarrow wp(S, q_1) \vee wp(S, q_2)$ can be invalid. The standard example for this property is the coin-flip program we've seen before.
- ***Example 11***: Let $flip \equiv$ ***if*** $T \rightarrow x := 0 \ \square \ T \rightarrow x := 1$ ***fi***.
  - Let $heads \equiv x = 0$ as and $tails \equiv x = 1$, then $M(flip, \varnothing) = \{ \{ x = 0 \}, \{ x = 1 \} \}$. Though $\{ x = 0 \}$ and $\{ x = 1 \}$ both satisfy $heads \vee tails$, neither of them satisfies $heads$ or $tails$. So $wp(flip, heads \vee tails) = T$ but $wp(flip, heads) = wp(flip, tails) = F$.
- Let's look at the situation in terms of sets of states. Assume $\perp \notin M(S, \sigma)$ and let $M(S, \sigma) = \Sigma_1 \cup \Sigma_2$ where $\Sigma_1 \vDash q_1$ and $\Sigma_2 \vDash q_2$. We have that $\Sigma_1 \cup \Sigma_2 \vDash q_1 \vee q_2$, but if neither Σ_1 nor Σ_2 are $\varnothing$, then $\Sigma_1 \cup \Sigma_2$ includes at least one element satisfies q_1 and one that satisfies q_2, so $\Sigma_1 \cup \Sigma_2$ satisfies neither q_1 nor q_2.

Weakest Preconditions

Part 2: Calculating wp, wlp; Domain Predicates

CS 536: Science of Programming, Spring 2023

2023-02-15: pp. 2-5

A. Why

- Weakest liberal preconditions *(wlp)* and weakest preconditions *(wp)* are the most general requirements that a program must meet to be correct under partial and total correctness.

B. Objectives

At the end of today you should understand

- How to calculate the *wlp* of loop-free programs.
- How to add error domain predicates to the *wlp* of a loop-free program to obtain its *wp*.

C. Calculating wlp for Loop-Free Programs

- Say a program is loop-free. If it is also error-free, then its *wp* and *wlp* are identical. Otherwise we will need to add error-avoiding information to the *wlp* to calculate the *wp*. Either way, calculating the *wlp* is the first step.
- The following algorithm takes S and q and calculates a predicate for $wlp(S, q)$.
- The calculation is syntactic, which is why it's described using $wlp(S, q) \equiv \ldots$ instead of $wp(S, q) \Leftrightarrow \ldots$.
  - $wlp(\textbf{skip}, q) \equiv q$
  - $wlp(v := e, Q(v)) \equiv Q(e)$ where Q is a predicate function over one variable .
    - The operation that takes us from $Q(v)$ to $Q(e)$ is called ***syntactic substitution***; we'll look at it in more detail in the next class, but for the examples here and in earlier classes, we've been using the simplest case, where we inspect the definition of Q and replacing each occurrence of the variable v with the expression e.
  - $wlp(S_1 ; S_2, q) \equiv wlp(S_1, wlp(S_2, q))$
    - The $wlp(S_2, q)$ guarantees that we'll run S_2 in a state that gets us to q. To guarantee that S_1 gets us to one of those states, we use the outer $wlp(S_1, \ldots)$.
  - $wlp(\textbf{if } B \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi}, q) \equiv (B \rightarrow w_1) \wedge (\neg B \rightarrow w_2)$ where $w_1 \equiv wlp(S_1, q)$ and $w_2 \equiv wlp(S_2, q)$.
    - This is $\Leftrightarrow (B \wedge w_1) \vee (\neg B \wedge w_2)$, so it's also acceptable as a result of this *wlp* calculation.
  - $wlp(\textbf{if } B_1 \rightarrow S_1 \ \square \ B_2 \rightarrow S_2 \textbf{ fi}, q) \equiv (B_1 \rightarrow w_1) \wedge (B_2 \rightarrow w_2)$ where $w_1 \equiv wlp(S_1, q)$ and $w_2 \equiv wlp(S_2, q)$.

- For the nondeterministic ***if***, you ***must*** use $(B_1 \to w_1) \wedge (B_2 \to w_2)$, not $(B_1 \wedge w_1) \vee (B_2 \wedge w_2)$, because they're not equivalent (unlike the deterministic ***if*** statement).
- When B_1 and B_2 are both true, either S_1 or S_2 can run, so we need $B_1 \wedge B_2 \to w_1 \wedge w_2$, and this is implied by $(B_1 \to w_1) \wedge (B_2 \to w_2)$.
- Using $(B_1 \wedge w_1) \vee (B_2 \wedge w_2)$ fails because it allows for the possibility that B_1 and B_2 are both true but only one of w_1 and w_2 is true. This isn't a problem when $B_2 \Leftrightarrow \neg B_1$, which is why we can use $(B \wedge w_1) \vee (\neg B \wedge w_2)$ with deterministic ***if*** statements.

D. Some Examples of Calculating wp/wlp:

- The programs in these examples never end in "state" $\perp$, so the *wp* and *wlp* are equivalent.
- These two examples are connected. [2023-02-15]
  - ***Example 2***: $wlp(x := x+1, x \geq 0) \equiv x+1 \geq 0$
  - ***Example 3***: $wlp(y := y+x; x := x+1, x \geq 0)$
    $\equiv wlp(y := y+x, wlp(x := x+1, x \geq 0))$
    $\equiv wlp(y := y+x, x+1 \geq 0) \equiv x+1 \geq 0$ (There's no y in the postcondition.)

- If we change the postcondition to include y, then it will be substituted for.) [2023-02-15]
- ***Example 4***: $wlp(y := y+x; x := x+1, x \geq y)$
  $\equiv wlp(y := y+x, wlp(x := x+1, x \geq y))$
  $\equiv wlp(y := y+x, x+1 \geq y)$
  $\equiv x+1 \geq y+x$
  (If we asked to calculate and logically simplify, not just calculate, the *wlp*, we'd continue)
  $\Leftrightarrow y \leq 1$.

- Changing the order of the assignments changes what gets substituted and when. [2023-02-15]
- ***Example 5***: Swap the two assignments in Example 4:
  $wlp(x := x+1; y := y+x, x \geq y)$
  $\equiv wlp(x := x+1, wlp(y := y+x, x \geq y))$
  $\equiv wlp(x := x+1, x \geq y+x))$
  $\equiv x+1 \geq y+x+1$ [$\Leftrightarrow y \leq 0$ if you want to logically simplify]

- The postcondition of an if-else statement and its two branches are the same. [2023-02-15]
- ***Example 6***: $wlp($***if*** $y \geq 0$ ***then*** $x := y$ ***fi***$, x \geq 0)$
  $\equiv wlp($***if*** $y \geq 0$ ***then*** $x := y$ ***else skip fi***$, x \geq 0)$
  $\equiv (y \geq 0 \to wlp(x := y, x \geq 0)) \wedge (y < 0 \to wlp($***skip***$, x \geq 0))$
  $\equiv (y \geq 0 \to y \geq 0) \wedge (y < 0 \to x \geq 0)$ or $(y \geq 0 \wedge y \geq 0) \vee (y < 0 \wedge x \geq 0)$

(If we were asked to calculate and logically simplify the *wlp*, we'd continue):

$\Leftrightarrow y \geq 0 \vee (y < 0 \wedge x \geq 0)$

$\Leftrightarrow (y \geq 0 \vee y < 0) \wedge (y \geq 0 \vee x \geq 0)$

$\Leftrightarrow (y \geq 0 \vee x \geq 0)$ (A correct answer) [2023-02-15]

$\Leftrightarrow (y < 0 \rightarrow x \geq 0)$ (Also correct, just differs in style)

E. Avoiding Runtime Errors in Expressions with Domain Predicates

- To avoid runtime failure of $\sigma(e)$, we'll take the context in which we're evaluating e and augment it with a predicate that guarantee non-failure of $\sigma(e)$. For example, for $\{P(e)\}$ $v := e \{P(v)\}$, we'll augment the precondition to guarantee that evaluation of e won't fail.
- For each expression e, we will define a ***domain predicate*** $D(e)$ such that $\sigma \vDash D(e)$ implies $\sigma(e) \neq \perp_e$.
  - This predicate has to be defined recursively, since we need to handle complex expressions like $b[b[k]]$. As we'll see, $D(b[b[k]]) \equiv 0 \leq k < size(b) \wedge 0 \leq b[k] < size(b)$.
  - As with *wp*, the domain predicate for an expression is unique only up to logical equivalence. For example, $D(x/y + u/v) \equiv y \neq 0 \wedge v \neq 0 \Leftrightarrow v * y \neq 0$. (Me personally, I prefer $y \neq 0 \wedge v \neq 0$, but it's a taste issue.)
- ***Definition***: ***(Domain predicate D(e) for expression e):*** We must define D for each kind of expression that can cause a runtime error:
  - First, a shortcut: if e contains no operations that can fail, then $D(e) \equiv T$.
    - For example, for a constant c or variable v, we have $D(c) \equiv T$ and $D(v) \equiv T$ because evaluation of a variable or constant doesn't cause failure,
  - The basic requirement is to define domain expressions for operations that can cause errors. For us, that's array lookup, division, modulus, and square root. Adding other operations or datatypes might introduce other cases.
    - $D(b[e]) \equiv D(e) \wedge 0 \leq e < size(b)$.
    - $D(e_1/e_2) \equiv D(e_1 \,\%\, e_2) \Leftrightarrow D(e_1) \wedge D(e_2) \wedge e_2 \neq 0$.
    - $D(sqrt(e)) \equiv D(e) \wedge e \geq 0$.
  - For operations that don't themselves cause errors, we simply check the subexpressions. This includes the arithmetic operators $+$, $-$, $*$, and the relational operators $\leq$, $<$, $=$, $\neq$, $>$, and $\geq$.
    - $D(e_1 \, op \, e_2) \equiv D(e_1) \wedge D(e_2)$, except when op is / or %.
    - $D(op \; e) \equiv D(e)$.
    - $D(f(e_1, e_2, \ldots)) \equiv D(e_1) \wedge D(e_2) \wedge \ldots$, except for $f \equiv sqrt$.
  - For conditional expressions [2023-02-15], we need safety of the tests and safety of the arms / branches.
    - $D(\textbf{\textit{if}}\ B\ \textbf{\textit{then}}\ e_1\ \textbf{\textit{else}}\ e_2\ \textbf{\textit{fi}}) \equiv D(B) \wedge (B \rightarrow D(e_1)) \wedge (\neg B \rightarrow D(e_2))$

[2023-02-15] (Removed a misplaced paragraph)

- ***Example 7***: $D(b[b[k]]) \equiv D(b[k]) \wedge 0 \le b[k] < size(b)$
  $\equiv D(k) \wedge 0 \le k < size(b) \wedge 0 \le b[k] < size(b)$
  $\equiv T \wedge 0 \le k < size(b) \wedge 0 \le b[k] < size(b)$
  $\equiv 0 \le k < size(b) \wedge 0 \le b[k] < size(b)$

- ***Example 8***: $D((-b + sqrt(b*b - 4*a*c))/(2*a))$

$\equiv D(e) \wedge D(2*a) \wedge 2*a \neq 0$	where $e \equiv -b + sqrt(b*b - 4*a*c)$
$\equiv D(-b) \wedge D(sqrt(b*b - 4*a*c)) \wedge D(2*a) \wedge 2*a \neq 0$	
$\equiv D(sqrt(b*b - 4*a*c)) \wedge 2*a \neq 0$	since $D(-b) \equiv D(2*a) \equiv T$
$\equiv D(b*b - 4*a*c) \wedge (b*b - 4*a*c \ge 0) \wedge 2*a \neq 0$	
$\equiv b*b - 4*a*c \ge 0 \wedge 2*a \neq 0$	since $D(b*b - 4*a*c \ge 0) \equiv T$
$\Leftrightarrow b*b - 4*a*c \ge 0 \wedge a \neq 0$	if asked to simplify arithmetically

[2023-02-15 miscellaneous changes below]

- ***Example 9***: $D($ ***if*** $0 \le k < size(b)$ ***then*** $b[k]$ ***else*** 0 ***fi***$)$. Here, the test guarantees that the array lookup won't fail. (The expression ***if*** B_1 ***then*** T ***else*** B_2 ***fi*** is equivalent to B_1 && B_2 in C, etc.)

$\equiv D(B) \wedge (B \rightarrow D(b[k]) \wedge (\neg B \rightarrow D(0))$	where $B \equiv 0 \le k < size(b)$
$\equiv (B \rightarrow D(b[k]) \wedge (\neg B \rightarrow T)$	since $D(B)$ and $D(0) \equiv T$
$\Leftrightarrow B \rightarrow D(b[k])$	since $\neg B \rightarrow T \Leftrightarrow T$
$\equiv B \rightarrow D(k) \wedge 0 \le k < size(b)$	expanding $D(b[k])$
$\equiv 0 \le k < size(b) \rightarrow T \wedge 0 \le k < size(b)$	definition of B
$\Leftrightarrow T$	logical simplification

F. Avoiding Runtime Errors in Statements with Domain Predicates

- Recall that we extended our notion of operational semantics to include $\langle S, \sigma \rangle \rightarrow^* \langle E, \perp_e \rangle$ to indicate that evaluation of S causes a runtime failure.
- We can avoid runtime failure of statements by adding domain predicates to the preconditions of statements. Though we can't in general calculate the *wlp*/*wp* of a loop, we can calculate a domain predicate for it.
- ***Definition***: For statement S, the [2023-02-15] ***domain predicate*** $D(S)$ gives a sufficient condition to avoid runtime errors. For loops, avoiding divergence is a separate problem we'll look at later.
  - $D($***skip***$) \equiv T$
  - $D(v := e) \equiv D(e)$
  - $D(b[e_1] := e_2) \equiv D(b[e_1]) \wedge D(e_2)$

- $D(S_1; S_2) \equiv D(S_1) \wedge wp(S_1, D(S_2))$
  - [Wed 2023-02-15, 18:27] The $D(S_1)$ tells us S_1 won't cause an error when run. The $wp(S_1, D(S_2))$ tells us that S_1 will establish $D(S_2)$, so running S_2 won't cause an error. To see this,
    - If $\sigma \models D(S_1)$ then $\perp_e \notin M(S_1, \sigma)$.
    - If $\sigma \models wp(S_1, D(S_2))$, then $M(S_1, \sigma) \models D(S_2)$, which implies $\perp_e \notin M(S_2, M(S_1, \sigma))$.
    - Combining $\perp_e \notin M(S_1, \sigma)$ and $\perp_e \notin M(S_2, M(S_1, \sigma))$ tells us $\perp_e \notin M(S_1; S_2, \sigma)$.
- $D(\textbf{\textit{if}}\ B\ \textbf{\textit{then}}\ S_1\ \textbf{\textit{else}}\ S_2\ \textbf{\textit{fi}}, q)$
  $\equiv D(B) \wedge (B \rightarrow D(S_1)) \wedge (\neg B \rightarrow D(S_2))$
- $D(\textbf{\textit{if}}\ B_1 \rightarrow S_1 \ \square\ B_2 \rightarrow S_2\ \textbf{\textit{fi}}, q)$
  $\equiv D(B_1 \vee B_2) \wedge (B_1 \vee B_2) \wedge (B_1 \rightarrow D(S_1)) \wedge (B_2 \rightarrow D(S_2))$
  - We need $(B_1 \vee B_2)$ to avoid failure of the nondeterministic ***if-fi*** due to none of the guards holding.
  - This definition extends easily to ***if-fi*** with one or more than two guarded commands.
- $D(\textbf{\textit{while}}\ B\ \textbf{\textit{do}}\ S_1\ \textbf{\textit{od}}) \equiv D(B) \wedge (B \rightarrow D(S_1))$
- $D(\textbf{\textit{do}}\ B_1 \rightarrow S_1 \ \square\ B_2 \rightarrow S_2\ \textbf{\textit{od}})$
  $\equiv D(B_1 \vee B_2) \wedge (B_1 \rightarrow D(S_1)) \wedge (B_2 \rightarrow D(S_2))$
  - This definition extends easily to ***do-od*** with one or more than two guarded commands.
  - The domain predicate for nondeterministic ***do-od*** is like that for ***if-fi*** except that having none of the guards hold does not cause an error.
  - Note ***while*** B ***do*** S_1 ***od*** is equivalent to ***do*** $B \rightarrow S$ ***od***, and happily, their D results match.

Calculating wp for loop-free programs

- With the domain predicates, it's easy to extend *wlp* for *wp* for loop-free programs because we don't have to argue for termination of a loop.
- ***Definition***: $wp(S, q) \equiv D(S) \wedge w \wedge D(w)$, where $w \equiv wlp(S, q)$.
  - $D(S)$ tells us that running S won't cause an error
  - w tells us that running S will establish q (if S terminates).
  - $D(w)$ tells us that w makes sense.

- ***Example 10***: If a program does a division, then the *wp* and *wlp* can differ.
  - We'll calculate $w_1 \equiv wp(S_1; S_2, q)$ where
    $S_1 \equiv x := y$, $S_2 \equiv z := v / x$, and $q \equiv z > x + 2$.
  - Since $w_1 \equiv wp(S_1; S_2, q) \equiv wp(S_1, wp(S_2, q))$, we should calculate $w_2 \equiv wp(S_2, q)$ first.
    $w_2 \equiv wp(S_2, q)$
    $\equiv wp(z := v / x, z > x + 2)$

$\equiv D(z := v/x) \wedge w \wedge D(w)$ where $w \equiv wlp(z := v/x, z > x+2) \equiv v/x > x+2$
$\equiv (x \neq 0) \wedge (v/x > x+2) \wedge D(v/x > x+2)$
$\equiv x \neq 0 \wedge v/x > x+2 \wedge x \neq 0$
$\equiv x \neq 0 \wedge v/x > x+2$ [1]

- So now we can calculate $w_1 \equiv wp(S_1, w_2)$.

  $w_1 \equiv wp(S_1, w_2)$
  $\equiv wp(x := y, x \neq 0 \wedge v/x > x+2)$
  $\equiv wlp(x := y, x \neq 0 \wedge v/x > x+2)$ since the assignment $x := y$ never fails
  $\equiv y \neq 0 \wedge v/y > y+2$

- ***Example 11***: Let's calculate $w_0 \equiv wp(x := b[k], sqrt(x) \geq 1)$.
  - Let $S \equiv x := b[k]$, $q \equiv sqrt(x) \geq 1$, and $w \equiv wlp(S, q)$.
  - We can expand
    - $w \equiv wlp(S, q) \equiv wlp(x := b[k], sqrt(x) \geq 1) \equiv sqrt(b[k]) \geq 1$.
  - It's also useful to calculate

    $D(w)$
    $\equiv D(sqrt(b[k]) \geq 1)$
    $\equiv D(b[k]) \wedge b[k] \geq 0$
    $\equiv 0 \leq k < size(b) \wedge b[k] \geq 0$
  - So then

    $w_0 \equiv wp(S, q)$
    $\equiv D(S) \wedge w \wedge D(w)$
    $\equiv D(x := b[k]) \wedge (sqrt(b[k]) \geq 1) \wedge D(sqrt(b[k]) \geq 1)$
    $\equiv (0 \leq k < size(b)) \wedge (sqrt(b[k]) \geq 1) \wedge (0 \leq k < size(b) \wedge b[k] \geq 0)$
    $\equiv 0 \leq k < size(b) \wedge sqrt(b[k]) \geq 1 \wedge b[k] \geq 0$
  - If further simplification is requested, we get
    $\Leftrightarrow 0 \leq k < size(b) \wedge b[k] \geq 1$

[1] To simplify syntactic/semantic calculations, let's again extend our notion of $\equiv$ so that $p \wedge p \equiv p \vee p \equiv p$.

Syntactic Substitution

CS 536: Science of Programming, Spring 2023

A. Why

- Syntactic substitution is used in the assignment rules to calculate weakest preconditions (and later, strongest postcondition).

B. Objectives

At the end of today's class you should

- Know what syntactic substitution is and how to do it.
- Be able to carry out substitution on an expression or predicate.

C. Syntactic Substitution

- Recall that $wp(v := e, P(v)) \equiv P(e)$
- The operation of going from $P(v)$ to $P(e)$ is called ***syntactic substitution***.
- A common notation is $p[e/v]$. The advantage of this notation is that it's easier to do multiple ("iterated") substitutions. There are other notations people use, such as $p[v := e]$, $p[v \mapsto e]$, and p^{e}_{v}.

D. Substitution Into An Expression

- As part of substitution into a predicate, we need to be able to ***substitute into an expression***; the idea is to take an expression e and replace its occurrences of variable v with expression e'.
- ***Notation***: We write $e[e'/v]$, pronounced "e with e' (substituted) for v". We'll treat the substitution brackets as having very high precedence, so we'll need parentheses around e for complex expressions.
- ***Example 1***: $x + y[5/x] \equiv x + (y[5/x]) \equiv x + y$ but $(x + y)[5/x] \equiv 5 + y$.
- For the language at hand, substitution into expressions is very simple because we don't have anything that introduces a local variable (like ***let*** $x = e_1$ ***in*** e_2).
- To carry out $e[e'/v]$, we go through e. Everywhere we see an occurrence of v, we replace it by (e'). If the parentheses are redundant, we can omit them.
  - If e has no occurrence of v (there's no v to replace), then $e[e'/v] \equiv e$. Another way to say this is that if e only uses variables $\neq v$, then $e[e'/v] \equiv e$.
- Note: Substitution is a textual operation. For example, $(x + x)[2/x] \equiv 2 + 2$, which equals 4 in any state, but $(x + x)[2/x] \neq 4$.
- ***Example 2***: $(a - x)[2/x] \equiv a - (2) \equiv a - 2$ (the parentheses are redundant)

- ***Example 3***: $(x*(x+1))[b-c/x] \equiv (b-c)*(b-c+1)$ (the parentheses are required).
- ***Example 4***: $(b[x*y])[x+3/x] \equiv b[(x+3)*y]$
- ***Example 5***: $(y+b[x])\ [x*3/x] \equiv y+b[x*3]$
- ***Example 6***: $(\textbf{if } x>0 \textbf{ then } -x \textbf{ else } 0 \textbf{ fi})[z+2/x] \equiv \textbf{if } z+2>0 \textbf{ then } -(z+2) \textbf{ else } 0 \textbf{ fi}$
- ***Example 7***: $(b[x*(x+1)/2])[y+4/x] \equiv b[(y+4)*((y+4)+1)/2] \equiv b[(y+4)*(y+4+1)/2]$.
- The technical definition of $e[e'/v]$ is done by cases on the structure of e. Briefly, we have constants and variables as base cases and expressions with subexpressions as recursive cases.

Definition of $e[e'/v]$ by Structural Induction

- ***Case 1*** (base cases)
  - $c[e'/v] \equiv c$ if c is a constant
  - $v[e'/v] \equiv (e')$
  - If $v \neq w$, then $w[e'/v] \equiv w$.
- ***Case 2*** (recursive cases): Consider the expressions that have subexpressions: function calls $f(e_1, e_2, \ldots)$, array indexing expressions $b[e_1, e_2, \ldots]$, parenthesized expressions (e_1), unary operations $\oplus e_1$, binary operations $e_1 \oplus e_2$ and ternary operations $e_1 ? e_2 : e_3$ (or ***if*** e_1 ***then*** e_2 ***else*** e_3 ***fi***), we recursively process each subexpression.
  - Let $e_1' \equiv (e_1)[e'/v]$, $e_2' \equiv (e_2)[e'/v]$, etc.
    - Then $(f(e_1, e_2, \ldots))[e'/v] \equiv f(e_1', e_2', \ldots)$
    - And $(b[e_1, e_2, \ldots])[e'/v] \equiv b[e_1', e_2', \ldots]$
    - And $(e_1 \oplus e_2)[e'/v] \equiv e_1' \oplus e_2'$
    - And so on.

E. Substitution Into A Predicate

- ***Notation***: $p[e/v]$ is pronounced "p with e (substituted) for v" and stands for the result of substituting e for each (free) occurrence of v in p. (Don't worry about free and bound occurrences of a variable until we get to quantified predicates.)
- Substitution into expressions and predicates is a syntactic operation. For example, $(x>0)[1/x] \equiv 1>0$, which $\Leftrightarrow$ true, but $(x>0)[1/x] \neq T$.

Substitution Case 1: Non-Quantified Predicate

- For a predicate that is not quantified, we substitute recursively in its sub-predicates or expressions. (Note the predicate might ***contain*** a quantified subpredicate, but those predicates will get covered in the other cases, where a predicate ***is*** a quantified predicate.)
  - $(\neg p)[e/v] \equiv \neg(p[e/v])$
  - $(p_1 \wedge p_2)[e/v] \equiv p_1[e/v] \wedge p_2[e/v]$, and similarly for $\vee$, $\rightarrow$, and $\leftrightarrow$.
  - $(e_1 < e_2)[e/v] \equiv (e_1[e/v]) < (e_2[e/v])$, and similarly for the other relational operators.

- ***Example 8***: $(x > 0 \to y \geq x/2)[z+1/x]$
  $\equiv (x > 0)[z+1/x] \to (y \geq x/2)[z+1/x]$
  $\equiv (z+1 > 0 \to y \geq (z+1)/2)$. (The parentheses around $z+1$ are necessary)
- ***Note:*** If p contains no occurrences at all of v, then $p[e/v] \equiv p$. E.g., $(x < y)[e/z] \equiv x < y$.
  - This is especially true of predicates that only contain constants, such as $2 + 2 = 4$.
- This case of substitution continues recursively until we come across a quantified predicate.
  - To cover the quantified predicate case, we need to know that only some occurrences of variables do get substituted for (the "free" occurrences). The others (the "bound" occurrences) do not get substituted for.
  - However, within a predicate, the same variable can be used in different ways. This complicates things.

F. Free and Bound Variables and Occurrences of Variables

- ***Notation***: Q stands for a quantifier ($\forall$ or $\exists$).
- For the definition of $(Q\,x\,.\,q)[e/v]$, our natural instinct is to think that $(Q\,x\,.\,q)[e/v]$ should $\equiv (Q\,x\,.\,(q[e/v]))$, but in fact this isn't always true because of a distinction between "free" and "bound" occurrences of variables.
- ***Definition***: If an occurrence of a variable v in a predicate is within the scope of a quantifier over v, then it is a ***bound occurrence***, else it is a ***free occurrence***. A variable v ***is free in*** (= ***occurs free in***) p iff it has a free occurrence in p. Similarly, v ***is bound in*** (= ***occurs bound in***) p iff it has a bound occurrence in p. (In computer science terms, local variables have bound occurrences, and non-local variables have free occurrences.)
- For any variable v and predicate p, there are four possibilities:
  - v is neither free nor bound in p (this case applies when v doesn't occur at all in p).
  - v is free but not bound in p: v occurs at least once in p, and all the occurrences of v are free.
  - v is not free but is bound in p: v occurs at least once in p, and all the occurrences of v are bound.
  - v is free and bound in p: v occurs at least twice in p with at least one occurrence being free and at least one occurrence being bound.
- ***Example 9***: If $p \equiv x > z \wedge \exists\, x\,.\,\exists\, y\,.\,y \leq f(x, y)$, then
  - x is free and bound in p. (Its first occurrence is free; its second is bound.)
  - y is bound in p but not free in p.
  - z is free in p but not bound in p.
  - w is neither free nor bound in p.
- The reason we're interested in occurrences of variables being free or bound in a predicate is that ***we only substitute for free occurrences*** of a variable. In computer science terms, we're looking for non-local variables, not local variables.

- Taking polynomials as an example, $p(x) = x^2 + a * x + y$. If we want to substitute 17 for y, that's fine: $p(x) = x^2 + a * x + 17$; substituting expressions with variables that aren't bound in the definition is okay too: substituting $(z^3 + 1)$ for y gives us $p(x) = x^2 + a * x + (z^3 + 1)$. But if we want to substitute something like $(x + 3)$ for y (note: x is the defined parameter variable), we ***don't*** want $p(x) = x^2 + a * x + (x + 3)$. But if we had defined $p(w) = w^2 + a * w + y$, then substituting $(x + 3)$ for y gives us $p(w) = w^2 + a * w + (x + 3)$.

G. Substitution Into A Quantified Predicate

- In case 1 of the definition of substitution, the major operator of the predicate was not a quantifier, it was a conjunction or disjunction, etc.
- In the remaining cases, we substitute into a quantified predicate: $(Q\,x\,.\,q)[e/v]$.

Substitution Case 2: Quantified Variable ≡ Variable to Replace

- In the simplest quantifier case, the quantified variable matches the variable we're substituting for. I.e., we have $(Q\,v\,.\,q)[e/v]$.
- Since all the occurrences in q of v are bound, there are no free occurrences of v in $Q\,v\,.\,q$, so there's nothing to replace: $(Q\,v\,.\,q)[e/v] \equiv Q\,v\,.\,q$.
- ***Example 10***: $(x > 0 \wedge \exists\, x\,.\,x \le f(y))[17/x] \equiv 17 > 0 \wedge \exists\, x\,.\,x \le f(y)$. Here, the first occurrence of x (in $x > 0$) is free, so we replace it with 17, but the second occurrence of x is bound, so we don't do any replacement.

Substitution Case 3: Quantified Variable Doesn't Occur in Replacement Expression

- If $x \neq v$ and x does not occur in e, then $(Q\,x\,.\,q)[e/v] \equiv (Q\,x\,.\,(q[e/v]))$. Here, we go through the text of q and replace its free occurrences of v with e.
- ***Example 11***: $(y \ge 0 \to \forall\, x\,.\,x > y \to x * x > y \wedge \exists y\,.\,f(y) > x))[17/y]$
  $\equiv 17 \ge 0 \to \forall\, x\,.\,(x > y \to x * x > y \wedge \exists y\,.\,f(y) > x)[17/y]$
  $\equiv 17 \ge 0 \to \forall\, x\,.\,x > 17 \to x * x > 17 \wedge \exists y\,.\,f(y) > x$.
  Note the y in $f(y)$ is bound, so there's no substituting for it.
- In case 3, the restriction that the quantified variable not appear in e keeps us from having a "capture" problem, where occurrences of x in e are free, but when we we replace an occurrence of v by e in $Q\,x\,.\,q[e/v]$, the occurrences of x in e become bound, which changes their meaning.
- ***Example 12:*** $(\exists y\,.\,y = v^2)[x + 1/v] \equiv \exists y\,.\,y = (x + 1)^2$. If we were to let $(\exists\, x\,.\,x = v^2)[x + 1/v]$ be $\exists\, x\,.\,x = (x + 1)^2$, then the x in $x + 1$ would become bound to the $\exists x$ (= the x would be "***captured***").
  - (Before the substitution, the x in $\ldots[x + 1/v]$ was not quantified, so after the substitution, we also want x to not be quantified.)

- The way out of this problem is to ***rename the quantified variable*** from x to something not in e; that way the quantifier can't capture occurrences of x.

Substitution Case 4: Quantified Variable Does Occur in Replacement Expression

- This case is the most complicated one. If $x \neq v$ and x occurs in e, then what we do is replace the quantified variable with one that doesn't appear in the quantifier's body. Then we proceed as in Case 3.
  - So, $(Q\,x\,.\,q)[e/v] \equiv (Q\,z\,.\,q[z/x])[e/v] \equiv (Q\,z\,.\,(q[z/x][e/v]))$
    where z is a ***fresh variable*** (one not used in e or q).
- ***Example 13***: Using z as a fresh variable, we have

  $(g(x,v) < 0 \wedge (\exists\, x\,.\,x = v^2) \wedge h(y,v) > 0)[x+1/v]$

  $\equiv g(x,x+1) < 0 \wedge (\exists\, z\,.\,((x = v^2)[z/x])[x+1\,/\,v]) \wedge h(y,x+1) > 0$

  // Pick fresh variable, quantify over it and then substitute for it in the body

  $\equiv\ g(x,x+1) < 0 \wedge (\exists\, z\,.\,z = v^2)[x+1\,/\,v] \wedge h(y,x+1) > 0$

  $\equiv\ g(x,x+1) < 0 \wedge (\exists\, z\,.\,z = (x+1)^2) \wedge h(y,x+1) > 0$

  - Note there's some ambiguity in the definition: Which "fresh" variable should we choose?

- Substitution into a predicate is also how application of a predicate function works.
- ***Example 14:*** Define $member(x,b) \equiv \exists\, 0 \le k < size(b)\,.\,x = b[k]$. Then $member(12,\ b1)$ is calculated as $(\exists\, 0 \le k < size(b)\,.\,x = b[k])[12\,/\,x][b1\,/\,b] \equiv\ \exists\, 0 \le k < size(b1)\,.\,12 = b[k]$. Renaming occurs when an argument uses a variable that's quantified in the body.

  $member(k*c, b2)\ \equiv\ (\exists\, 0 \le k < member(b)\,.\,x = b[k])[k*c\,/\,x][b2\,/\,b]$

  $\equiv\ (\exists\, 0 \le k1 < size(b)\,.\,x = b[k1])[k*c\,/\,x][b2\,/\,b]$ -- Renaming k to $k1$

  $\equiv\ (\exists\, 0 \le k1 < size(b2)\,.\,k*c = b2[k1])$

Forward Assignment; Strongest Postconditions

CS 536: Science of Programming, Spring 2023

2023-02-27 p.7, 2023-04-07 p.8

A. Why?

- Sometimes, the forward version of the assignment rule is preferable to the backward version.
- The forward assignment rule is part of calculating the *sp* (strongest postcondition) of a loop-free program.
- The *sp* is the postcondition that includes all possible results of a program under a precondition.

B. Outcomes

At the end of this class you should

- Know the basic assignment axioms.
- Know what a strongest postcondition is and how to calculate the *sp* of loop- free programs.

C. Forward Assignment Rules

- We already have a "backwards" assignment rule, $\{P(e)\}\ v := e\ \{P(v)\}$ where P is a predicate function. If we just use the body of P as the predicate, the rule is $\{(body\_of\_P)[e/v]\}\ v := e\ \{P\}$.
  - Since $p[e/v] \equiv wlp(v := e, p)$, this is the most general possible rule.
- What about the other direction, $\{p\}\ v := e\ \{???\}$ — what can we use for the postcondition?
  - Most people's first guess is $\{p\}\ v := e\ \{p \wedge v = e\}$, which can work under certain conditions.

New Variable Introduction

- If v is a new (fresh) variable (doesn't appear free in p and doesn't appear in e) then $\{p\}\ v := e\ \{p \wedge v = e\}$.
  - For example, $\{x > y\}\ z := 2\ \{x > y \wedge z = 2\}$
- To justify this, using *wlp*, we know $\{(p \wedge v = e)[e/v]\}\ v := e\ \{p \wedge v = e\}$.
  - Expanding, $(p \wedge v = e)[e/v] \equiv p[e/v] \wedge e = e[e/v]$.
  - Since v is fresh, it doesn't occur in p or e, so $p[e/v] \equiv p$ and $e[e/v] \equiv e$. So we need $\{p \wedge e = e\}\ v := e\ \{p \wedge v = e\}$, which certainly holds.

Forward Assignment - General Case

- As an example of why $\{p\}\ v := e\ \{p \wedge v = e\}$ doesn't work in general, consider the triple $\{x > 0\}\ x := x - 2\ \{???\}$.

- We certainly don't have $\{ x > 0 \}\ x := x - 2\ \{ x > 0 \wedge x = x - 2 \}$. If we look more carefully, the relationship we're trying to capture with $x > 0 \wedge x = x - 2$ is:
  $(\text{value of } x \text{ before asgt}) > 0 \wedge (\text{the current value of } x) (\text{value of } x \text{ before asgt}) - 2$
- This example uses subtraction, which we can invert, so we can write $(x + 2 > 0 \wedge x = (x + 2) - 2)$ for the postcondition.
- But not all assignments are invertible: Consider $\{ x > 0 \}\ x := x / 2\ \{ ??? \}$. Because of truncating integer division, $(2 * x > 0 \wedge x = (2 * x / 2))$ is only true for even values of x.
- What we can do instead is to introduce a name for $(\text{the value of } x \text{ before the assignment})$. If we use x_0 as this name, we can say $\{ x_0 = x \wedge x > 0 \}\ x := x / 2\ \{ x_0 > 0 \wedge x = x_0 / 2 \}$.
- ***Definition: Aging*** x is the process of introducing a logical constant to name the value of x before a change.
- Note we don't have to actually store x_0 in memory; it's just a name we use for logical reasoning purposes — x_0 is a "fresh logical constant"; fresh in the sense that it doesn't appear in p or e, logical because it only appears in the correctness discussion, not the program, and constant because though x changes, x_0 doesn't. (Note in this context, "logical" doesn't mean "boolean".)

The General Forward Assignment Rule

- The general rule for forward assignment is $\{ p \wedge v = v_0 \}\ v := e\ \{ p[v_0 / v] \wedge v = e[v_0 / v] \}$. If it's omitted, the $v = v_0$ part of the precondition is understood.
  - ***Example 1a***: $\{ x > 0 \wedge x = x_0 \}\ x := x - 1\ \{ x_0 > 0 \wedge x = x_0 - 1 \}$
  - ***Example 2a***: $\{ s = sum(0, i) \wedge s = s_0 \}\ s := s + i + 1\ \{ s_0 = sum(0, i) \wedge s = s_0 + i + 1 \}$.
- Aging x and s using the $x = x_0$ and $s = s_0$ clauses is a bit annoying; we can drop them by using an existential in the postcondition, but that's no fun either:
  - ***Example 1b***: $\{ x > 0 \}\ x := x - 1\ \{ \exists\, x_0 .\ x_0 > 0 \wedge x = x_0 - 1 \}$
  - ***Example 2b***: $\{ s = sum(0, i) \}\ s := s + i + 1\ \{ \exists\, s_0 .\ s_0 = sum(0, i) \wedge s = s_0 + i + 1 \}$.
- Let's drop the existential as implied — when a symbol appears in the postcondition but not the precondition, then we're implicitly quantifying it existentially in the postcondition.
- We've actually been doing something similar with the precondition: Variables free in the precondition are treated as being universally quantified across both the precondition and postcondition.
  - ***Example 1c***: (For all x, there is an x_0 such that) $\{ x > 0 \}\ x := x - 1\ \{ x_0 > 0 \wedge x = x_0 - 1 \}$
  - ***Example 2c***: (For all s and i, there is an s_0 such that) $\{ s = sum(0, i) \}$ $s := s + i + 1\ \{ s_0 = sum(0, i) \wedge s = s_0 + i + 1 \}$.
- ***Example 3:*** (For all s, s_0, and i, there is an i_0 such that)
  $\{ s_0 = sum(0, i) \wedge s = s_0 + i + 1 \}\ i := i + 1\ \{ s_0 = sum(0, i_0) \wedge s = s_0 + i_0 + 1 \wedge i = i_0 + 1 \}$.
- ***Discussion:*** Simplifying the postcondition; Equivalence with *wp*

- The postcondition of Example 3 can be weakened to $s = sum(0, i)$. Combining Examples 2c and 3 gives us $\{ s = sum(0, i) \}\ s := s + i + 1\ ;\ i := i + 1\ \{ s = sum(0, i) \}$.
- Using backward assignment to calculate p in $\{ p \}\ s := s + i + 1\ ;\ i := i + 1\ \{ s = sum(0, i) \}$ produces the same triple (after simplification)

$$
\begin{aligned}
p &\equiv wp(s := s + i + 1\ ;\ i := i + 1, s = sum(0, i)) \\
&\equiv wp(s := s + i + 1, wp(i := i + 1, s = sum(0, i))) \\
&\equiv wp(s := s + i + 1, s = sum(0, i + 1)) \\
&\equiv s + i + 1 = sum(0, i + 1) && \text{Finishes calculation of } s\ p \\
&\Leftrightarrow s = sum(0, i) && \text{Logical simplification}
\end{aligned}
$$

D. Correctness of the Assignment Rules

- This section is mostly technical. The key takeaway is that the forward and backward assignment rules are equally strong because you can derive each from the other. In addition, new variable introduction is just a special case of forward assignment.
- ***Discussion:***
  - Combining Examples 2c and 3 above and weakening the postcondition gives us the triple
    $\{ s = sum(0, i) \}\ s := s + i + 1\ ;\ i := i + 1\ \{ s = sum(0, i) \}$
  - It turns out that *wp* can be used on the same program and postcondition to produce the same triple after precondition strengthening.
  - This is not accidental: The forward and backward assignment rules are equivalent in power in the sense that anything proved using forward assignment can also be proved using backward assignment, and vice versa.
  - The standard way to argue this is to show how a triple obtained using one assignment rule can be derived using the other assignment rule.

Derivation of the Forward Assignment Rule from the Backward Assignment Rule

- The forward assignment rule appears to be very different from our earlier "backward" assignment rule, but actually, we can derive the forward assignment rule using the backward assignment rule.
- ***Theorem (Forward Assignment):*** $\vDash \{ p \wedge v = v_0 \}\ v := e\ \{ p[v_0/v] \wedge v = e[v_0/v] \}$, where v_0 is a fresh logical constant.
- ***Proof***: Forward assignment tells us the triple $\{ p \wedge v = v_0 \}\ v := e\ \{ p[v_0/v] \wedge v = e[v_0/v] \}$ is correct. We'd like to prove the same triple using backward assignment. Applying the backward assignment rule to the given postcondition $p[v_0/v] \wedge v = e[v_0/v]$ will give us a precondition that is logically equivalent (but not syntactically equal) to $p \wedge v = v_0$. So we'll know that the triple $\{ p \wedge v = v_0 \}\ v := e\ \{ p[v_0/v] \wedge v = e[v_0/v] \}$ is provable using both forward and backward assignment.

  With backward assignment, we know $\{ wlp(v := e, q) \}\ v := e\ \{ q \}$ where $q \equiv p[v_0/v] \wedge$

$v = e\,[\,v_0/v\,]$. The only occurrence of v within $p\,[\,v_0/v\,] \wedge v = e\,[\,v_0/v\,]$ is the v in $v = \ldots$. This makes $wlp\,(\,v := e\,,\,q\,) \equiv q\,[\,e/v\,] \Leftrightarrow p\,[\,v_0/v\,] \wedge e = e\,[\,v_0/v\,]$. If $p \wedge v = v_0$, then $p\,[\,v_0/v\,]$, and if $v = v_0$, then $e\,[\,v_0/v\,] = e\,[\,v/v\,] = e$. So $p \wedge v = v_0$ implies $wlp\,(\,v := e\,,\,q\,)$, and we can use precondition strengthening to get $\{\,p \wedge v = v_0\,\}\,v := e\,\{\,q\,\}$. So the backward assignment rule justifies the forward assignment rule.

- For a particular example, with $\{\,x > 0 \wedge x = x_0\,\}\,x := x - 1\,\{\,x_0 > 0 \wedge x = x_0 - 1\,\}$, we find $wlp\,(\,x := x - 1\,,\,x_0 > 0 \wedge x = x_0 - 1\,) \equiv x_0 > 0 \wedge x - 1 = x_0 - 1$, which is implied by $x > 0 \wedge x = x_0$.

Derivation of New Variable Introduction

- The simpler rule for introducing a new variable is a special case of forward assignment.
- We want $\{\,p\,\}\,v := e\,\{\,p \wedge v = e\,\}$ if v doesn't occur in e or and v is not free in p. By forward assignment, $\{\,p\,\}\,v := e\,\{\,p\,[\,v_0/v\,] \wedge v = e\,[\,v_0/v\,]\,\}$, where v_0 is a fresh logical constant. Since v does not occur in e, we know $e\,[\,v_0/v\,] \equiv e$. Similarly, since v isn't free in p, we know that $p\,[\,v_0/v\,] \equiv p$. Substituting into $\{\,p\,\}\,v := e\,\{\,p\,[\,v_0/v\,] \wedge v = e\,[\,v_0/v\,]\,\}$ gives us $\{\,p\,\}\,v := e\,\{\,p \wedge v = e\,\}$.

Derivation of the Backward Assignment Rule from the Forward Assignment Rule

- We know the forward assignment rule can be derived from the backward assignment rule. The converse is also true: We can derive the forward assignment rule from the backward assignment rule.
- ***Theorem (Backward Assignment)***: $\models \{\,p\,[\,e/v\,]\,\}\,v := e\,\{\,p\,\}$ follows from the forward assignment rule.
- ***Proof:*** Using forward assignment on the precondition $p\,[\,e/v\,] \wedge v = v_0$ and assignment $v := e$ gives us the postcondition $p\,[\,e/v\,]\,[\,v_0/v\,] \wedge v = e\,[\,v_0/v\,]$. To justify backward assignment, we need this last predicate to imply $p\,[\,e/v\,]$.
- In $p\,[\,e/v\,]$, the only occurrences of v are the ones in e, so in $p\,[\,e/v\,]\,[\,v_0/v\,]$, the only occurrences of v_0 are the ones that replace the v's in e.
- Thus $p\,[\,e/v\,]\,[\,v_0/v\,]$ is logically equivalent to $p\,[\,e\,[\,v_0/v\,]/v\,]$ (where we replace the v's in e with v_0's and then replace the v's in p with the result). Let $e' \equiv e\,[\,v_0/v\,]$. Now, if $v = e'$, then $(\,p\,[\,e'/v\,] \wedge v = e'\,)$ is equivalent to $p \wedge v = e'$, which implies $p\,[\,e/v\,]$. So the backward assignment rule can be derived from the forward assignment rule.

E. The Strongest Postcondition (sp)

- ***Definition***: Given a precondition p and program S, the ***strongest postcondition*** of p and S is (the predicate that stands for) the set of states we can terminate in if we run S starting in a state that satisfies p. In symbols,
  - $sp\,(\,p\,,S\,) = \{\,\tau \mid \tau \in M\,(\,S\,,\sigma\,) - \perp$ for some σ where $\sigma \models p\,\}$.
  - Equivalently, $sp\,(\,p\,,S\,) = \cup_{\sigma}\,(\,M\,(\,S\,,\sigma\,) - \perp\,)$ where $\sigma \models p$.

- If we treat $M(S, \ldots) - \perp$ as a function over states, then $sp(p, S)$ is the image of this function over the states that satisfy p.
- Figure 1 shows the relationship between p, S, and $sp(p, S)$:
  - If $\sigma \models p$, then every state in $M(S, \sigma) - \perp$ is by definition in $sp(p, S)$, so $\models \{p\}\, S\, \{sp(p, S)\}$.
    - This is only valid for ***partial correctness***: Starting in a state that satisfies p might yield $\perp$.
    - To get total correctness, $\models_{tot} \{p\}\, S\, \{sp(p, S)\}$, we need termination, $\models_{tot} \{p\}\, S\, \{T\}$.
- ***Example 4***: Let $W \equiv$ ***while*** $i \neq 0$ ***do*** $i := i - 1$ ***od***, then $sp(i \geq 0, W) \equiv i = 0$, which implies that $\{i \geq 0\}\, W\, \{sp(i \geq 0, W)\}$ is not only partially correct, it's totally correct.
- ***Example 5***: For the same W, weakening $i \geq 0$ produces the same sp. In the limit, $sp(T, W) \equiv i = 0$. Here, the sp is partially correct but not totally correct: $\models \{T\}\, W\, \{sp(T, W)\}$ but $\not\models_{tot} \{T\}\, W\, \{sp(T, W)\}$. Of course, this is because W doesn't terminate when one starts with $i < 0$.
- ***Why strongest***? For partial correctness, $sp(p, S)$ is a postcondition. What makes it the *strongest* postcondition is that it implies any other postcondition: for any q, $\models \{p\}\, S\, \{q\}$ iff $\models sp(p, S) \rightarrow q$.

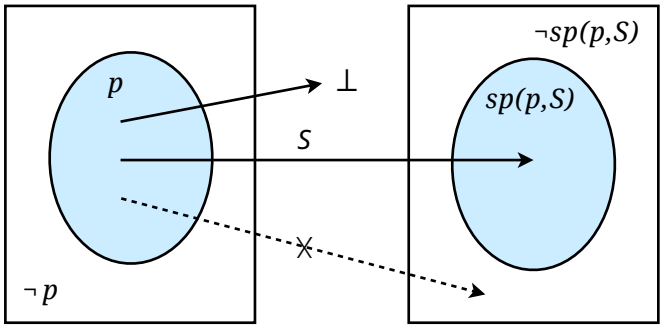


Figure 1: *sp(p,S)* is the set of states reachable via *S* from *p*

- ***Lemma:*** $\models \{p\}\, S\, \{q\}$ iff $\models sp(p, S) \rightarrow q$.
  - The $\Leftarrow$ direction holds by postcondition weakening: We have $\models \{p\}\, S\, \{sp(p, S)\}$ and $\models sp(p, S) \rightarrow q$, therefore $\models \{p\}\, S\, \{q\}$.
  - For the $\Rightarrow$ direction, assume $\models \{p\}\, S\, \{q\}$ and let $\tau \models sp(p, S)$. Since $\tau \models sp(p, S)$, we have $\tau \in M(S, \sigma) - \perp$ for some $\sigma \models p$. But $\sigma \models \{p\}\, S\, \{q\}$ tells us that $M(S, \sigma) - \perp \models q$, so $\tau \models q$. So $\tau \models sp(p, S)$ implies $\tau \models q$, so we have $\models sp(p, S) \rightarrow q$.

F. Calculating Strongest Postconditions of Loop-Free Programs

Definition: (Calculation of sp, part 1):

- As with *wlp*, the *sp* of a program can be textually calculated for loop-free programs.
- The simplest cases for calculating *sp* are for the ***skip***, assignment, and sequence statements.

- $sp(p, \textbf{skip}) \equiv p$.
  - Since ***skip*** doesn't change the state, whatever was true before the ***skip*** is true after it.
- $sp(p, v := e) \equiv p[v_0/v] \wedge v = e[v_0/v]$, where v_0 is a fresh constant (the "aged" version of v)
  - The forward assignment rule turns out to give the strongest description of the state after an assignment. We won't prove this formally, but intuitively, the value of v before the assignment isn't changed by an assignment to v: It's still the value of v before the assignment. So everything that was true about v_0 before the assignment is still true after the assignment. Similarly, the new value of v, when described relative to v_0, is the same before and after the assignment.
  - ***Example 6:*** $sp(x > y, x := x + k) \equiv (x > y)[x_0/x] \wedge x = (x + k)[x_0/x] \equiv x_0 > y \wedge x = x_0 + k$.
  - ***Example 7:*** Here's the conclusion of Example 6 used as a postcondition for a different assignment:

    $$sp(x_0 > y \wedge x = x_0 + k, y := y + k) \equiv (x_0 > y \wedge x = x_0 + k)[y_0/y] \wedge y = (y + k)[y_0/y]$$
    $$\equiv x_0 > y_0 \wedge x = x_0 + k \wedge y = y_0 + k.$$
- $sp(p, S_1; S_2) \equiv sp(sp(p, S_1), S_2)$.
  - The most we can know after $S_1; S_2$ is the most we know after executing S_2 in the state that is the most we know after S_1.
- ***Example 8:*** Combining Examples 6 and 7,

  $sp(p, x := x + k;\ y := y + k)$

  $\equiv sp(sp(p, x := x + k), y := y + k)$ Defn *sp* of sequence

  $\equiv sp(x_0 > y \wedge x = x_0 + k, y := y + k)$ Example 6

  $\equiv x_0 > y_0 \wedge x = x_0 + k \wedge y = y_0 + k$ Example 7
  - If we don't want to keep the old values x_0 and y_0, we can weaken the *sp* to $x > y$ instead.
- If we have a sequence of assignments to one variable, then we introduce multiple logical variables to talk about its values at different times in the sequence.
- ***Example 9:*** To complete $\{x > f(x, y)\}\ x := x + 1; x := x * x\ \{???\}$, we'll calculate the strongest postcondition.
- We need $sp(x > f(x, y), S_1; S_2) \equiv sp(sp(x > f(x, y), S_1), S_2)$ where $S_1 \equiv x := x + 1$ and $S_2 \equiv x := x * x$. Because x is assigned to twice, there will be three versions of x: x_0 names the value x had before the first assignment, x_1 names the value x had between the two assignments, and x will end being the name of the value after the two assignments.

  $sp(x > f(x, y), S_1)$

  $\equiv sp(x > f(x, y), x := x + 1)$

  $\equiv (x > f(x, y))[x_0/x] \wedge x = (x + 1)[x_0/x]$ (using x_0 as the fresh variable)

  $\equiv x_0 > f(x_0, y) \wedge x = x_0 + 1$

  $sp(sp(x > f(x, y), S_1), S_2)$

  $\equiv sp(x_0 > f(x_0, y) \wedge x = x_0 + 1, x := x * x)$

$\equiv (x_0 > f(x_0, y) \wedge x = x_0 + 1)[x_1/x] \wedge x = (x * x)[x_1/x]$ *(*using x_1 as the fresh variable*)*
$\equiv x_0 > f(x_0, y) \wedge x_1 = x_0 + 1 \wedge x = x_1 * x_1$

Strongest postconditions of conditional statements

- The *sp* of a conditional is the disjunction of the *sp*'s of its branches*. Disjunction is needed because, though execution will make one of those *sp*'s hold, when the conditional statement ends, we lose track of which branch was executed.
- Since the branches of a conditional can include assignments, bindings for initial values of variables ($x = x_0$, etc.) will be needed somewhere when calculating the *sp* of the conditional.
- However, instead of having these bindings turn up recursively, as we analyze the branches, they need to be part of the top level of calculation.
- Let's look at an illustrative example before seeing how to calculate the *sp* of a conditional.
- ***Notation:*** An alternate style for indicating logical constants is to use capital letters. E.g., X instead of x_0. Which notation to use is a style issue†; let's try it for an example or two.
- ***Example 10:*** Let $IF \equiv$ ***if*** $x \geq y + z$ ***then*** $x := x - 1$ ***else*** $y := y + 2$ ***fi*** and let $p \equiv T$ be our precondition, then the *sp* of the true branch and false branch are
  - $sp(T \wedge x \geq y + z, x := x - 1) \equiv X \geq y + z \wedge x = X - 1$
  - $sp(T \wedge x < y + z, y := y + 2) \equiv x < Y + z \wedge y = Y + 2$
- The disjunction of these two is $(X \geq y + z \wedge x = X - 1) \vee (x < Y + z \wedge y = Y + 2)$, which doesn't include the information that the true branch doesn't modify y and the false branch doesn't modify x. So though it is a postcondition for T and IF, it's not the strongest one.
- To define $sp(p, IF)$, it will be handy to pre-calculate some things.
- ***Definitions:***
  - $lhs(S)$ = the set of variables that appear as the lhs of assignments in statement S.
  - $rhs(S)$ = the set of variables that appear in the rhs of assignments in S or in tests in S.
  - $free(p)$ = the set of variables that are free in predicate p.
  - $aged(p, S) = lhs(S) \cap (rhs(S) \cup free(p))$ is the subset of variables of S whose assignments cause aging.

Definition: (Calculation of sp, part 2):

- $sp(p, IF)$: Let $IF \equiv$ ***if*** B ***then*** S_1 ***else*** S_2 ***fi*** and let $aged(p, IF) = \{x, y, \ldots, z\}$, then
  $sp(p, IF) \equiv sp(p_0 \wedge B, S_1) \vee sp(p_0 \wedge \neg B, S_2)$ where $p_0 = p \wedge x = X \wedge y = Y \wedge \ldots \wedge z = Z$.

---

* Since they mean the same thing, I'm going to shorten "arms / branches" to just "branches".

† A typeset X is easier to read than x_0, but on paper, handwritten X and x can be confused if you're not careful. Also, if you need multiple logical names based on x, using x, x_0, x_1, x_2, ... is easy but x, X, X, ... gets out of hand very quickly.

  - The nondeterministic case is very similar. For $NF \equiv$ ***if*** $B_1 \rightarrow S_1 \ \square \ B_2 \rightarrow S_2$ ***fi***, p_0 is the same but $sp(p, NF) \equiv sp(p_0 \wedge B_1, S_1) \vee sp(p_0 \wedge B_2, S_2)$.
- ***Example 10 re***visited: p≡T and IF≡if x≥y+z then x:=x–1 else y:=y+2 fi, so lhs(IF)={x,y}, rhs(IF)={x,y,z}, and free(p)≡∅ ~~*so p≡x≥y+z, so free(p)={x,y,z}*~~ *[2023-04-07].* This makes aged(x≥y+z,IF)={x,y}∩({x,y,z}∪∅)={x,y}, so we'll add x=X∧y=Y as a conjunct to p to get p_0≡T∧x=X∧y=Y, or simply x=X∧y=Y.

  sp(x=X∧y=Y,IF)≡(X≥y+z∧x=X–1∧y=Y)∨(x<Y+z∧x=X∧y=Y+2) *[2023-02-27]*
- This postcondition does include the information that the true branch modifies x but not y and the false branch modifies y but not x. This makes it stronger than Example 10's condition.

- Fresh variables, generalized: Given a predicate p and an assignment v:=e, we've said that v is a fresh variable if it doesn't appear in p or e. The definition aged(p,S)=lhs(S)∩ (rhs(S)∪free(p)) generalizes this. If all the assignments in S are to variables that aren't otherwise used in S and don't appear free in p, then the assignments are all to fresh variables.

- Example 11: sp(T,if y≥1 then x:=1 else z:=0 fi)
  ≡sp(y≥1,x:=1)∨sp(y<1,z:=0)
  ≡(y≥1∧x=1)∨(y<1∧z=0).
  With
  - lhs(IF)={x,z}
  - rhs(IF)={y}
  - free(p)= ∅ ~~*{y}*~~ *[2023-04-07]*
  - $aged(y \geq 1, S) = \{x, z\} \cap (\{y\} \cup \{y\}) = \{x, z\} \cap \{y\} = \varnothing$.

Proof Rules and Proofs for Correctness Triples

Part 1: Axioms, Sequencing, and Auxiliary Rules

CS 536: Science of Programming, Spring 2023

A. Why?

- We can't generally prove that correctness triples are valid using truth tables.
- We need proof axioms for atomic statements (***skip*** and assignment) and inference rules for compound statements like sequencing.
- In addition, we have inference rules that let us manipulate preconditions and postconditions.

B. Outcomes

At the end of this topic you should know

- The basic axioms for ***skip*** and assignment.
- The rules of inference for strengthening preconditions, weakening postconditions, composing statements into a sequence, and combining statements using ***if-else***.

C. Truth vs Provability of Correctness Triples

- It's time to start distinguishing between whether correctness triples are semantically true (***valid***) from whether they are ***provable***: When we write a program, how can we mechanically convince ourselves that a triple is valid?
- In propositional logic, truth/validity is about truth tables, and proofs involve rules like associativity, commutativity, DeMorgan's laws, etc.
- In predicate logic, truth/validity is about satisfaction of a predicate in a state, and one adds on rules to prove things about quantified predicates and about the kinds of values we're manipulating. We didn't actually look at those rules specifically.
- In propositional logic, it's often easier to deal with a truth table than to manipulate propositions using rules, but in predicate logic, proof rules are unavoidable because the truth table for a universal can be infinitely large. (The truth table for $\forall x \in S . P(x)$ has one row for each value of S.)
- A ***proof system*** is a set of logical formulas determined by a set of axioms and rules of inference using a set of syntactic algorithms.
- One difference between validity and provability comes from predicate logic: Not everything that is true is provable.

- (This was proved by Kurt Gödel in the 1930s in his two Incompleteness Theorems.)
- Luckily, this problem doesn't come up in everyday programming (unless your idea of an everyday program involves writing programs that read programs and try to prove things about them).

- For correctness triples, the other difference comes from ***while*** loops, basically because to describe the exact behavior of a loop may require an infinite number of cases.
  - (This is where undecidable functions come up in CS 530: Theory of Computing.)
  - Unfortunately, this problem really does come up in trying to prove that correctness triples are true.
  - Instead of proving the exact behavior of loops, we'll approximate their behavior using "loop invariants. " (Stay tuned.)

D. Reasoning About Correctness Triples

- So how do we reason about correctness triples, with proofs? We've been studying proofs formally, as textual things manipulated by textual operations (i. e. , proof rules).
  - First, we'll have ***Axioms*** that tell us that certain basic triples are true.
  - Second, we'll have ***Rules of Inference*** that tell us that if we can prove that certain triples are true, then some other triple will be true.
- In predicate logic we have axioms like "$x + 0 = x$" and rules of inference like ***Modus ponens***: If p and $p \rightarrow q$, then q or ***Modus tollens***: If $\neg q$ and $p \rightarrow q$, then $\neg p$.
- For a proof system for triples,
  - The formulas are correctness triples.
  - We'll have axioms for the ***skip*** and assignment statements.
  - We'll have rules of inference for the sequence, conditional, and iterative statements.
- ***Notation***: $\vdash \{ p \} S \{ q \}$ means "We can prove that $\{ p \} S \{ q \}$ is valid. "
- The $\vdash$ symbol is a single turnstile pronounced "prove" or "can prove". Often the $\vdash$ is left off as understood, but more generally, we can include items (predicates or correctness triples) to the left of the turnstile. This means "If we assume the items to the left then we can prove that the right hand side triple is valid. "

E. Proof System for Partial Correctness of Deterministic Programs

- To get the proof rules, we'll follow the semantics of the different statements. That way we'll know the rules are ***sound*** (if we can prove something, then it's valid). We won't try to deal with the opposite direction, ***completeness*** (if something is valid, then we can prove it). We'll have one axiom or rule for each kind of statement and we'll have some auxiliary rules that don't depend on statements.

F. Proof Formats

Proof Trees

- ***Definition: Judgements*** are the statements that proofs prove (or assume): In predicate logic, they're predicates; for us, they'll be correctness triples.
- There are various notations for writing out proofs. We'll show two of them (and you've already even seen one of them!).
- The first notation is a ***proof tree***. Here, each node is a judgement plus the name of a proof rule. Edges in the tree go from a parent to the children it relies on for use by the rule.
- For example, here are two rules you would expect in a predicate logic proof tree. On the left is ***modus ponens*** and on the right is multiplication by zero.

$$\frac{p \qquad p \rightarrow q}{q}\ \text{modus ponens} \qquad\qquad \frac{}{x * 0 = x}\ \text{multiplication by zero}$$

- The ***rule name*** is attached to a line drawn between the parent and children. For a rule of inference, the ***antecedents*** (child judgements which are required to apply the rule) are written above the line; the ***consequent*** (the parent judgement that is being proved by the rule) is below. Leafs are judgements proved by axiom or assumption; they're drawn with a labeled line above them but with no antecedents.
  - (This is the simplest kind of proof tree; there are more complicated kinds that include other features.)
- An advantage of proof trees is that you can read them top-down or bottom-up.
  - Top-down: If we know p and we know $p \rightarrow q$, then by modus ponens, we know q
  - Bottom-up: To prove q, by modus ponens it's sufficient to prove p and $p \rightarrow q$.
- The main disadvantage of proof trees is that they're difficult to draw. In a full proof, each antecedent that isn't proved by axiom is the root of its own proof subtree, and if a rule has more than one antecedent, then the tree becomes wider and wider.

Hilbert-Style Proofs

- In a Hilbert-style proof (the kind you've already seen), we have two columns. On the left, we have the judgements being proved; on the right we have the rule names being used to prove each judgement. Antecedents must above the consequent, and line numbers let us name the antecedents being used by a rule.

- In this format, modus ponens is

  1. p
  2. $p \rightarrow q$
  3. q modus ponens 1, 2

- The order of antecedents doesn't matter; they just have to be above the consequent, so we could have written

  1. $p \rightarrow q$
  2. p
  3. q modus ponens 2, 1

- The name Hilbert-style proof comes from David Hilbert, one of the first people to investigate the structure of mathematical proofs.
- Below, we'll use Hilbert-style proofs because they are more convenient to write than proof trees and because people are generally more familiar with them from high-school geometry.
- For correctness triples, we'll have rules for each kind of statement (***skip***, assignment, etc.), plus a number of auxiliary rules that allow for manipulations like precondition strengthening, etc.

G. Skip Axiom

- The ***skip*** statement is a primitive statement (it contains no substatement), so its correctness is proved by axiom.

Proof tree format:

$$\frac{}{\{\, p \,\}\ \textbf{skip}\ \{\, p \,\}}\ \text{skip}$$

Hilbert-style format:

1. $\{\, p \,\}\ \textbf{skip}\ \{\, p \,\}$ skip

H. Assignment Axioms

- The assignment statement is also primitive, so it i also proved by axiom. Unlike skip, the assignment axiom comes in two versions.

- The *wp* version of assignment:

  1. $\{\, p\,[\, e/x \,]\,\}\ x := e\ \{\, p \,\}$ assignment (backward)

- The *sp* version of assignment:

  1. $\{\, p \wedge x = x_0 \,\}\ x := e\ \{\, p\,[\, x_0/x \,] \wedge x = e\,[\, x_0/x \,] \,\}$ assignment (forward)

     where x_0 is a fresh logical constant

- In addition, the $x = x_0$ clause is implied if omitted.
- If x is itself fresh (doesn't appear in p or e), then the $x = x_0$ clause can definitely be dropped, and the rule simplifies to $\{ p \}\, x := e\, \{ p \wedge x = e \}$ because we're simply introducing a new variable.

I. Sequence (a.k.a. Composition) Rule

- The sequence rule allows us to take two statements and form a sequence from them.

Proof tree

$$\frac{\{p\}\, S_1\, \{r\} \qquad \{r\}\, S_2\, \{q\}}{\{p\}\, S_1 ; S_2\, \{q\}}\ \text{sequence}$$

Hilbert-style

sequence

1. $\{p\}\, S_1\, \{r\}$
2. $\{r\}\, S_2\, \{q\}$
3. $\{p\}\, S_1 ; S_2\, \{q\}$ sequence 1, 2

Example 1:

- Below, lines 1 and 2 are proved by axiom, and line 3 is the use of the sequence rule.

1. $\{T\}\, k := 0\, \{k = 0\}$ assignment
2. $\{k = 0\}\, s := k\, \{k = 0 \wedge s = 0\}$ assignment
3. $\{T\}\, k := 0\, ;\ s := k\, \{k = 0 \wedge s = 0\}$ sequence 1, 2

J. Conjunction and Disjunction Rules

- ***Definition:*** An ***auxiliary*** proof rule is one that's not associated with proving a particular kind of statement.
- The conjunction and disjunction rules are auxiliary rules that allow us to combine two proofs of the same triple. Their soundness relies on the semantics of $\wedge$ and $\vee$, not on the semantics of statements, which is why they are auxiliary rules.

1. $\{p_1\}\, S\, \{q_1\}$
2. $\{p_2\}\, S\, \{q_2\}$
3. $\{p_1 \wedge p_2\}\, S\, \{q_1 \wedge q_2\}$ conjunction 1, 2

- Disjunction is similar:

1. $\{p_1\}\, S\, \{q_1\}$
2. $\{p_2\}\, S\, \{q_2\}$
3. $\{p_1 \vee p_2\}\, S\, \{q_1 \vee q_2\}$ disjunction 1, 2

- In the tree format, the conjunction rule is below. (Disjunction is similar.)

$$\frac{\{p_1\}\, S\, \{q_1\} \qquad \{p_2\}\, S\, \{q_2\}}{\{p_1 \wedge p_2\}\, S\, \{q_1 \wedge q_2\}}\ \text{conjunction}$$

K. Strengthening and Weakening Rules

- The strengthening and weakening rules are also auxiliary rules. Unlike the other rules we've seen, they each include an antecedent that isn't a correctness triple; it's a predicate (an implication, specifically). Generically we call them ***predicate logic obligations***. We won't actually include predicate logic proofs of them in our correctness triple proofs, but the obligations do need to be true if our correctness triple proof is going to be true.

Strengthen Precondition

1.	$p_1 \to p_2$	predicate logic
2.	$\{ p_2 \} S \{ q \}$	
3.	$\{ p_1 \} S \{ q \}$	strengthen precondition 1, 2

- Proof tree:

$$\frac{p_1 \to p_2 \qquad \{ p_2 \} S \{ q \}}{\{ p_1 \} S \{ q \}}\ \text{strengthen precondition}$$

Example 2:

- (By adding a rule to justify line 2, we have a full proof here.)

1.	$x \geq 0 \to x^2 > 0$	predicate logic
2.	$\{ x^2 \geq 0 \}\ k := 0\ \{ x^2 \geq k \}$	assignment
3.	$\{ x \geq 0 \}\ k := 0\ \{ x^2 \geq k \}$	precondition strengthen, 1, 2

- Proof tree:

$$\frac{\dfrac{}{x \geq 0 \to x^2 > 0}\ \text{predicate logic} \qquad \dfrac{}{\{ x^2 \geq 0 \}\ k := 0\ \{ x^2 \geq k \}}\ \text{assignment}}{\{ x \geq 0 \}\ k := 0\ \{ x^2 \geq k \}}\ \text{str. precondition}$$

Weaken Postcondition Rule

- Symmetric to the precondition strengthening rule is the postcondition weakening rule. Like that rule, this one has a predicate logic obligation:

1.	$\{ p \} S \{ q_1 \}$	
2.	$q_1 \to q_2$	predicate logic
3.	$\{ p \} S \{ q_2 \}$	precond. weakening 1, 2

- Proof tree:

$$\frac{\{p\}\,S\,\{q_1\} \qquad q_1 \to q_2}{\{p\}\,S\,\{q_2\}}\ \text{postcondition weakening}$$

Example 3:

This is a slightly different proof of the conclusion from Example 2.

1.	$\{x \geq 0\}\ k := 0\ \{x \geq k\}$	assignment axiom
2.	$x \geq k \to x^2 > k$	predicate logic
3.	$\{x \geq 0\}\ k := 0\ \{x^2 > k\}$	postcond. weak. 1, 2

- Proof tree:

$$\frac{\dfrac{}{\{x^2 \geq 0\}\ k := 0\ \{x^2 \geq k\}}\ \text{assignment} \qquad \dfrac{}{x \geq 0 \to x^2 > 0}\ \text{predicate logic}}{\{x \geq 0\}\ k := 0\ \{x^2 \geq k\}}\ \text{postcondition weaken}$$

- In a correctness triple proof, there's often no unique proof of the conclusion, even ignoring how lines can be reordering. We can see this in Examples 2 and 3, which have slightly different predicate logic obligations, so they're certainly similar but not completely identical.

Example 4:

The conclusion of this proof appeared in Example 1.

1.	$\{k = 0\}\ s := k\ \{k = 0 \wedge s = k\}$	assignment (forward)
2.	$k = 0 \wedge s = k \to k = 0 \wedge s = 0$	predicate logic
3.	$\{k = 0\}\ s := k\ \{k = 0 \wedge s = 0\}$	postcond. weak. 1, 2

Example 5:

1.	$\{0 = 0 \wedge 0 = 0\}\ k := 0\ \{k = 0 \wedge k = 0\}$	assignment (backwards)
2.	$T \to 0 = 0 \wedge 0 = 0$	predicate logic
3.	$\{T\}\ \ k := 0\ \{k = 0 \wedge k = 0\}$	precondition strength. 2, 1
4.	$\{k = 0 \wedge k = 0\}\ \ s := k\ \{k = 0 \wedge s = 0\}$	assignment (backwards)
5.	$\{T\}\ k := 0\ ;\ \ s := k\ \{k = 0 \wedge s = 0\}$	sequence 3, 4

Example 6:

- Here's another proof of the same conclusion that uses forward assignment instead of backwards assignment and postcondition weakening instead of precondition strengthening. It also uses the sequence rule earlier.

1.	$\{ T \} k := 0 \{ T \wedge k = 0 \}$	assignment (forward)
2.	$\{ T \wedge k = 0 \} s := k \{ T \wedge k = 0 \wedge s = k \}$	assignment (forward)
3.	$\{ T \} k := 0 ; s := k \{ T \wedge k = 0 \wedge s = k \}$	sequence 1, 2
4.	$T \wedge k = 0 \wedge s = k \rightarrow k = 0 \wedge s = 0$	predicate logic
5.	$\{ T \} k := 0 ; \ s := k \{ k = 0 \wedge s = 0 \}$	postcondition weakening 3, 4

- Technically, the "$T \wedge$ " part of "$T \wedge k = 0$ ..." above needs to be there because it's the "$p [v_0 / v] \wedge$ ..." part of the assignment rule, $\{ p \} v := e \{ p [v_0 / v] \wedge v = e [v_0 / v] \}$. But it's annoying to write. But at this point, I think we're familiar enough with syntactic equality that we can give ourselves a bit more freedom to abbreviate things.

L. Review - Looser Syntactic Equality

- (I know we talked about this in earlier classes but I thought it would be good to write it all down somewhere.)
- Syntactic equality, you'll recall, is used as an indicator of semantic equality. Checking for it is easy and fast but we give up complete accuracy: Syntactic equality implies semantic equality but not vice versa.
- We've been using a very simple notion of syntactic equality: Ignoring redundant parentheses. Though very fast (it takes linear time) and useful, it misses out on a number of properties that are reasonably easy to check for.
- We can trade in a bit of runtime for flexibility.

- ***Definition (Version 2 of syntactic equality***: Two predicates are syntactically equal if we can show that they are identical using the following transformations:
  - Ignore redundant parentheses (including those from associative operators).
  - Identity: $p \wedge T \equiv p \vee F \equiv p$.
  - Domination: $p \vee T \equiv T$ and $p \wedge F \equiv F$.
  - Idempotentcy: $p \vee p \equiv p \wedge p \equiv p$.
  - Commutativity of $\wedge$ and $\vee$.
- We can check for version 2 of $\equiv$ by converting two predicates into some sort of normal form first.
  - (Throw away identity and domination relations, sort the result to get around commutativity, and drop duplicates for idempotentcy.)

Example 6 revisited:

- Here's Example 6 written to use the looser notion of ≡.

1.	$\{ T \}\ k := 0\ \{ k = 0 \}$	assignment (forward)
2.	$\{ k = 0 \}\ s := k\ \{ k = 0 \wedge s = k \}$	assignment (forward)
3.	$\{ T \}\ k := 0\ ;\ s := k\ \{ k = 0 \wedge s = k \}$	sequence 1, 2
4.	$k = 0 \wedge s = k \rightarrow k = 0 \wedge s = 0$	predicate logic
5.	$\{ T \}\ k := 0\ ;\ \ s := k\ \{ k = 0 \wedge s = 0 \}$	postcondition weakening 3, 4

Proof Rules and Proofs for Correctness Triples, v.2

Part 2: Conditional and Iterative Statements

CS 536: Science of Programming, Spring 2023

A. Why?

- Proof rules give us a way to establish truth with textually precise manipulations
- We need inference rules for compound statements such as conditional and iterative.

B. Outcomes

At the end of this topic you should know

- The rules of inference for ***if-else*** statements.
- The rule of inference for ***while*** statements.
- The impracticality of the *wp* and *sp* for loops; the definition and use of loop invariants.

C. Rules for Conditionals

- There are two popular ways to characterize correctness for ***if-else*** statements

If-Else Conditional Rule 1

- The *sp*-oriented basic rule is

    1. $\{ p \wedge B \}\ S_1\ \{ q_1 \}$
    2. $\{ p \wedge \neg B \}\ S_2\ \{ q_2 \}$
    3. $\{ p \}$ ***if*** B ***then*** S_1 ***else*** S_2 ***fi*** $\{ q_1 \vee q_2 \}$ if-else 1,2

  (The rule name can be "if-else" or "conditional" or anything similar.)

- In proof tree form:

$$\frac{\{ p \wedge B \}\ S_1\ \{ q_1 \} \qquad \{ p \wedge \neg B \}\ S_2\ \{ q_2 \}}{\{ p \}\ \textbf{if}\ B\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \textbf{fi}\ \{ q_1 \vee q_2 \}}\ \text{if-else}$$

- The rule says that
    - If running the true branch S_1 in a state satisfying p and B establishes q_1,
    - And running the false branch S_2 in a state satisfying p and $\neg B$ establishes q_2,
    - Then you know that running the ***if-else*** in a state satisfying p establishes $q_1 \vee q_2$.

- ***Example 1***: Here's a proof of $\{ T \}$ **if** $x \geq 0$ **then** $y := x$ **else** $y := -x$ **fi** $\{ y \geq 0 \}$. We need
  - $\{ x \geq 0 \} y := x \{ y \geq 0 \}$ for the true branch (line 1 below).
  - $\{ x < 0 \} y := -x \{ y \geq 0 \}$ for the false branch (lines 2 – 4 below).

1.	$\{ x \geq 0 \} y := x \{ y \geq 0 \}$	(backward) assignment
2.	$\{ x < 0 \} y := -x \{ x < 0 \wedge y = -x \}$	(forward) assignment
3.	$x < 0 \wedge y = -x \rightarrow y \geq 0$	predicate logic
4.	$\{ x < 0 \} y := -x \{ y \geq 0 \}$	postcondition weakening, 2, 3
5.	$\{ T \}$ **if** $x \geq 0$ **then** $y := x$ **else** $y := -x$ **fi** $\{ y \geq 0 \}$	***if-else*** 1, 4

- The proof above used forward assignment; backward assignment works also: Lines 2 – 4 become

2.	$\{ -x \geq 0 \} y := -x \{ y \geq 0 \}$	(backward) assignment
3.	$x < 0 \rightarrow -x \geq 0$	predicate logic
4.	$\{ x < 0 \} y := -x \{ y \geq 0 \}$	precondition strengthening 3, 2

If-Else Conditional Rule 2

- ***Conditional rule 2:*** An equivalent, more goal-oriented / *wp*-oriented conditional rule is:

1.	$\{ p_1 \} S_1 \{ q_1 \}$	
2.	$\{ p_2 \} S_2 \{ q_2 \}$	
3.	$\{ p_0 \}$ **if** B **then** S_1 **else** S_2 **fi** $\{ q_1 \vee q_2 \}$	***if-else*** 2, 1
	where $p_0 \equiv (B \rightarrow p_1) \wedge (\neg B \rightarrow p_2)$	

- If we add a preconditioning strengthening step of $p \rightarrow (B \rightarrow p_1) \wedge (\neg B \rightarrow p_2)$ to the rule above, we get the same effect as the old precondition $(p \wedge B \rightarrow p_1) \wedge (p \wedge \neg B \rightarrow p_2)$.
- We can derive this second version of the conditional rule using the first version. The assumptions below become the antecedents of the derived rule above; the conclusion below becomes the consequent of the derived rule above.

1.	$\{ p_1 \} S_1 \{ q_1 \}$	assumption 1
2.	$p_0 \wedge B \rightarrow p_1$	predicate logic
	where $p_0 \equiv (p \wedge B \rightarrow p_1) \wedge (p \wedge \neg B \rightarrow p_2)$	
3.	$\{ p_0 \wedge B \} S_1 \{ q_1 \}$	precondition strengthening 2, 1
4.	$\{ p_2 \} S_2 \{ q_2 \}$	assumption 2
5.	$p_0 \wedge \neg B \rightarrow p_2$	predicate logic
6.	$\{ p_0 \wedge \neg B \} S_2 \{ q_2 \}$	precondition strengthening 5, 4
7.	$\{ p_0 \}$ **if** B **then** S_1 **else** S_2 **fi** $\{ q_1 \vee q_2 \}$	***if-else*** 3, 6

If-Then Statements

- An ***if-then*** statement is an ***if-else*** with $\{ p \wedge \neg B \}$ ***skip*** $\{ p \wedge \neg B \}$ as the false branch.

1.	$\{ p \wedge B \}\, S_1 \,\{ q_1 \}$	
2.	$\{ p \wedge \neg B \}$ ***skip*** $\{ p \wedge \neg B \}$	skip
3.	$\{ p \}$ ***if*** B ***then*** S_1 ***fi*** $\{ q_1 \vee (p \wedge \neg B) \}$	if-else 1, 2

Nondeterministic Conditionals

- Perhaps surprisingly, the proof rules for nondeterministic conditionals are almost exactly the same as for deterministic conditionals.

Nondeterministic if-fi rule 1: (*sp*-like)

1.	$\{ p \wedge B_1 \}\, S_1 \,\{ q_1 \}$	
2.	$\{ p \wedge B_2 \}\, S_2 \,\{ q_2 \}$	
3.	$\{ p \}$ ***if*** $B_1 \rightarrow S_1 \,\square\, B_2 \rightarrow S_2$ ***fi*** $\{ q_1 \vee q_2 \}$	if-fi 1, 2

Nondeterministic if-fi rule 1: (*wp*-like)

1.	$\{ p_1 \}\, S_1 \,\{ q_1 \}$	
2.	$\{ p_2 \}\, S_2 \,\{ q_2 \}$	
3.	$\{ p_0 \}$ ***if*** $B_1 \rightarrow S_1 \,\square\, B_2 \rightarrow S_2$ ***fi*** $\{ q_1 \vee q_2 \}$	if-fi 1, 2
	where $p_0 \equiv (p \wedge B_1 \rightarrow p_1) \wedge (p \wedge B_2 \rightarrow p_2)$	

D. Problems With Calculating the wp or sp of a Loop

- What is $wp (W, q)$ for a typical loop $W \equiv$ ***while*** B ***do*** S ***od***? It turns out that some $wp (W, q)$ have no finite representation. ($sp (W, p)$ has the same problem.)
  - Let's look at the general problem of $wp (W, q)$.
  - First, define w_k to be the weakest precondition of W and q that requires exactly k iterations.
    - Let $w_0 \equiv \neg B \wedge q$ and for all $k \geq 0$, define $w_{k+1} \equiv B \wedge wp (S , w_k)$.
  - If we know that W will run for, say, ≤ 3 iterations, then $wp (W, q) \Leftrightarrow w_0 \vee w_1 \vee w_2 \vee w_3$.
  - But in general, W might run for any number of iterations, so $wp (W, q) \Leftrightarrow w_0 \vee w_1 \vee w_2 \vee \ldots$.
  - If this infinitely-long disjunction collapses somehow, then we can write $wp (W, q)$ finitely.
  - E.g., if $w_{k+1} \rightarrow w_k$ when $k \geq 5$, then $wp (W, q) \Leftrightarrow w_0 \vee w_1 \vee w_2 \vee w_3 \vee w_4 \vee w_5$.
  - Or, if there's a predicate function $P (k) \Leftrightarrow w_k$ (i.e., if the w_k are parameterized by k), then $wp (W, q) \Leftrightarrow \exists n \,.\, P (n)$.

E. Using Invariants to Approximate the wp and sp With Loops

Basic notions

- If we can't calculate $wp(S, q)$ or $sp(p, W)$ exactly, the best we can do is to approximate it.
- The simplest approximation is a predicate p that implies all the w_k.
  - If $p \Rightarrow w_k$ for all k, then $p \Rightarrow w_0 \vee w_1 \vee w_2 \vee \ldots$, so $p \Rightarrow wp(S, q)$.
- ***Definition:*** A ***loop invariant for*** $W \equiv$ ***while*** B do S ***od*** is a predicate p such that $\vDash \{ p \wedge B \}\, S \,\{ p \}$. It follows that $\vDash \{ p \}\, W \,\{ p \wedge \neg B \}$.[1]
  - Under partial correctness, if W terminates, it must terminate satisfying $p \wedge \neg B$.
  - Note this is for partial correctness only: To get total correctness, we'll need to prove that the loop terminates, and we'll address that problem later.
- ***Notation***: To indicate a loop's invariant, we'll add it as an extra clause: ***{ inv*** p ***} while*** B ***do*** S ***od***. This declares that p is not only a precondition of the loop, it's an invariant.

Need Useful Invariants

- Not all invariants are useful. E.g., any tautology is an invariant: $\{ T \wedge B \}\, S \,\{ T \}$, so $\{ T \}\, W \,\{ T \wedge \neg B \}$. For that matter, contradictions are invariants too, but they're even less useful.
- The key is to find an invariant that:
  1. Can be established using simple loop initialization code: $\{ p_0 \}$ initialization code $\{ p \}$.
  2. Can serve as a precondition and postcondition of a loop iteration: $\{ p \wedge B \}$ loop body $\{ p \}$.
  3. When combined with $\neg B$ and loop termination code, implies the postcondition we want: $\{ p \wedge \neg B \}$ termination code $\{ q \}$. If $p \wedge \neg B \rightarrow q$, then we don't need any termination code.
- There's no general algorithm for generating useful invariants. In a future class, we'll look at some heuristics for trying to find them.

Semantics of Invariants

- How do invariants fit in with the semantics of loops?
- Recall if we take the loop $W \equiv$ ***{ inv*** p ***} while*** B ***do*** S ***od*** and run it in state σ_0, then one iteration takes us to state σ_1, the next to σ_2, and so on: $\sigma_{k+1} = M(S, \sigma_k)$ for all k, and $M(W, \sigma_0)$ is the first σ_k that satisfies $\neg B$; if there is no such state, then we write $\perp_d \in M(W, \sigma_0)$[2].

---

[1] We've been using "p" as a generic name for a predicate. From now on, it may or may not stand for a loop invariant, depending on the context.

[2] If W is nondeterministic, it's a bit more complicated: For each possible sequence of τ_k, $M(W, \tau_0)$ either contains the first τ_k that satisfies $\neg B$ or $\perp_d$ if there is no such τ_k.

- The invariant p must be satisfied by every possible σ_0, σ_1, ..., which implies that it's an approximation to various *wp* and *sp* for the loop and loop body:

Predicate	***Approximates***	***Because***
p	the *wp* of the loop	$p \rightarrow wp(W, p \wedge \neg B)$
$p \wedge B$	the *wp* of the loop body	$p \wedge B \rightarrow wp(S, p)$
$p \wedge \neg B$	the *sp* of the loop	$sp(p, W) \rightarrow p \wedge \neg B$
p	the *sp* of the loop body	$sp(S, p \wedge B) \rightarrow p$

Loop Initialization and Cleanup

- The purpose of loop initialization code is to establish the loop invariant: $\{ p_0 \} S_0 \{ p \}$, where S_0 is the initialization code. Any variables that appear fresh in the invariant have to be initialized; e.g., $\{ n > 0 \} k := 0 \{ 0 \le k < n \}$.
- If $p \wedge \neg B \rightarrow q$, the desired postcondition for the loop, then no cleanup is necessary, otherwise we need loop termination code: $\{ p \wedge \neg B \}$ *termination code* $\{ q \}$.

F. While Loop Rule; Loop Invariant Example

- The proof rule for a loop only has one antecedent, which requires us to have a loop invariant.

1.	$\{ p \wedge B \} S \{ p \}$	
2.	$\{$ **inv** $p \}$ **while** B **do** S **od** $\{ p \wedge \neg B \}$	loop (or while), 1

- As a triple, the loop behaves like $\{ p \}$ **while** B **do** S **od** $\{ p \wedge \neg B \}$, so any precondition strengthening is relative to p, and any postcondition weakening is relative to $p \wedge \neg B$.

Example 2: Correctness of a Loop Body Using an Invariant

- We want to show that the loop W establishes $s = sum(0, n)$, given
  - $p \equiv 0 \le k \le n \wedge s = sum(0, k)$
  - $W \equiv$ **while** $k < n$ **do** $k := k+1 ; s := s+k$ **od**
- First, let's write out a full proof of correctness for this program, then we can analyze its parts:

1.	$\{ p[s+k/s] \} s := s+k \{ p \}$	(backward) assignment
2.	$\{ p[s+k/s][k+1/k] \} k := k+1 \{ p[s+k/s] \}$	(backward) assignment
3.	$\{ p[s+k/s][k+1/k] \} k := k+1 ; s := s+k \{ p \}$	sequence 2, 1
4.	$p \wedge k < n \rightarrow p[s+k/s][k+1/k]$	predicate logic
5.	$\{ p \wedge k < n \} k := k+1 ; s := s+k \{ p \}$	precondition str 4, 3
6.	$\{$ **inv** $p \} W \{ p \wedge k \ge n \}$	loop 5
7.	$p \wedge k \ge n \rightarrow s = sum(0, n)$	predicate logic
8.	$\{$ **inv** $p \} W \{ s = sum(0, n) \}$	postcondition weakening 6, 7

- The key requirement is showing that p is indeed invariant (line 5). Using the loop rule will let us conclude $\{\ \textbf{inv}\ p\ \}\ W\ \{\ p \wedge k \geq n\ \}$ (line 6).
- Once the loop terminates, we know $p \wedge k \geq n$ holds, but our final goal is to show $s = sum(0, n)$. It turns out that postcondition weakening is sufficient (we don't need any cleanup code). This completes the loop
- Turning back to the loop body $\{\ p \wedge k < n\ \}\ k := k + 1\ ;\ s := s+k\ \{\ p\ \}$, since this is a sequence, we need to show correctness of each assignment statement (lines 1 and 2) and combine them into a sequence (line 3).
  - We use the backward assignment rule twice, but the proof can certainly be done with forward assignment (see Example 3 below). The structure of the triple makes it easy to infer that backward assignment is being used, so "backward" can be omitted.
  - When we combine the assignments to form the sequence (line 3), the resulting precondition is $p\,[\,s+k/s\,]\,[\,k+1/k\,]$, so we use precondition strengthening to get $p \wedge k < n$, which is the form required by the loop rule.
- A reminder: The implication in line 4, $p \wedge k < n \rightarrow p\,[\,s+k/s\,]\,[\,k+1/k\,]$, is a predicate logic obligation. We're concentrating on correctness triples, which is why we're omitting formal proofs of the obligations. Still, it's good to convince ourselves that the implication is correct:
- First, let's expand the substitutions used. For $p \wedge k < n \rightarrow p\,[\,s+k/s\,]\,[\,k+1/k\,]$, we get
  - $p\,[\,s+k/s\,] \equiv (\,0 \leq k \leq n \wedge s = sum(0, k)\,)\,[\,s+k/s\,] \equiv 0 \leq k \leq n \wedge s+k = sum(0, k)$
  - $p\,[\,s+k/s\,]\,[\,k+1/k\,] \equiv (\,0 \leq k+1 \leq n \wedge s+k+1 = sum(0, k+1)\,)$
  - $(\,p \wedge k < n\,) \equiv (\,0 \leq k \leq n \wedge s = sum(0, k) \wedge k < n\,)$
- So $p \wedge k < n \rightarrow p\,[\,s+k/s\,]\,[\,k+1/k\,]$ expands to an implication that's easy to see is correct.
  $0 \leq k \leq n \wedge s = sum(0, k) \wedge k < n) \rightarrow 0 \leq k+1 \leq n \wedge s+k+1 = sum(0, k+1)$.
- There's also an obligation in line 7, $(\,p \wedge k \geq n \rightarrow s = sum(0, n)\,)$ but this one is easier to see: $p \wedge k \geq n$ implies $k \leq n \wedge k \geq n$, so $k = n$. Along with $s = sum(0, k)$ from p, we get $s = sum(0, n)$.

Example 3: Correctness of the Same Loop Body Using sp

- Above, we showed correctness of the loop body using *wp*; it's also possible to prove correctness using *sp* instead. We have to replace lines 1 – 5 of the proof above, but lines 6 – 8 don't change because they don't rely on how the loop body was proved to be correct.

1.	$\{\ p \wedge k < n\ \}\ k := k + 1\ \{\ p_1\ \}$	assignment
	where $p_1 \equiv (\,p \wedge k < n\,)\,[\,k_0/k\,] \wedge k = (\,k+1\,)\,[\,k_0/k\,]$	
2.	$\{\ p_1\ \}\ s := s+k\ \{\ p_2\ \}$	assignment
	where $p_2 \equiv p_1\,[\,s_0/s\,] \wedge s = (\,s+k\,)\,[\,s_0/s\,]$	
3.	$\{\ p \wedge k < n\ \}\ k := k + 1\ ;\ s := s+k\ \{\ p_2\ \}$	sequence 1, 2
4.	$p_2 \rightarrow p$	predicate logic
5.	$\{\ p \wedge k < n\ \}\ k := k + 1\ ;\ s := s+k\ \{\ p\ \}$	postcondition weak. 4, 3

- Here are the expansions of p_1 and p_2 used in the new proof:

- $p_1 \equiv (p \wedge k < n)[k_0/k] \wedge k = (k+1)[k_0/k]$
  $\equiv ((0 \le k \le n \wedge s = sum(0, k)) \wedge k < n)[k_0/k] \wedge k = (k+1)[k_0/k]$
  $\equiv 0 \le k_0 \le n \wedge s = sum(0, k_0) \wedge k_0 < n \wedge k = k_0 + 1$
- $p_2 \equiv p_1[s_0/s] \wedge s = (s+k)[s_0/s]$
  $\equiv (0 \le k_0 \le n \wedge s = sum(0, k_0) \wedge k_0 < n \wedge k = k_0 + 1)[s_0/s] \wedge s = s_0 + k$
  $\equiv 0 \le k_0 \le n \wedge s_0 = sum(0, k_0) \wedge k_0 < n \wedge k = k_0 + 1 \wedge s = s_0 + k$

Example 4: Another Loop Example

- Here's a simple loop program that calculates $s = sum(0, n) = 0 + 1 + \ldots + n$ where $n \ge 0$. (If $n < 0$, define $sum(0, n) = 0$.) Note the loop invariant appears explicitly. Also, the invariant is the same as in Example 3.

```
{ n ≥ 0 }
k := 0 ; s := 0 ;
{ inv p ≡ 0 ≤ k ≤ n ∧ s = sum ( 0 , k ) }
while k < n do
    s := s+k+1 ;
    k := k + 1
od
{ s = sum ( 0 , n ) }
```

- Informally, to see that this program works, we need
  - $\{ n \ge 0 \}\ k := 0 ; s := 0\ \{ p \equiv 0 \le k \le n \wedge s = sum(0, k) \}$
  - $\{ p \wedge k < n \}\ s := s+k+1 ; k := k + 1\ \{ p \}$
  - $p \wedge k \ge n \rightarrow s = sum(0, n)$
- It's straightforward to use *wp* or *sp* to show that the two triples are correct. A bit of predicate logic gives us the implication, which we need to weaken the loop's postcondition to the one we want.
- We'll do a detailed analysis in a little while.

G. Alternative Invariants Yield Different Programs and Proofs

- The invariant, test, initialization code, and body of a loop are all interconnected: Changing one can change them all. For example, we use $s = sum(0, k)$ in our invariant, so we have the loop terminate with $k = n$.
- If instead we use $s = sum(0, k+1)$ or $s = sum(0, k-1)$ in our invariant, we must terminate with $k+1 = n$ or $k-1 = n$ respectively, and we change the increment of s.

- ***Example 5***: Using $s = sum(0, k)$ as the invariant.

```
{ n ≥ 0 }
k := 0 ;  s := 0 ;
{ inv p ≡ 0 ≤ k ≤ n ∧ s = sum ( 0 , k ) }     // same invariant as in Examples 3 and 4
```

while $k < n$ ***do***
    $s := s+k+1$;
    $k := k+1$
od
$\{ s = sum(0, n) \}$

- ***Example 6***: Using $s = sum(0, k+1)$ as the invariant.

$\{ n > 0 \}$
$k := 0$; $s := 1$;
$\{$ ***inv*** $p_1 \equiv 0 \le k+1 < n \wedge s = sum(0, k+1) \}$
while $k+1 < n$ ***do***
    $s := s+k+2$;
    $k := k+1$
od
$\{ s = sum(0, n) \}$

- ***Example 7***: Using $s = sum(0, k-1)$ as the invariant.

$\{ n \ge 0 \}$
$k := 1$; $s := 0$;
$\{$ ***inv*** $p_2 \equiv 0 \le k-1 < n \wedge s = sum(0, k-1) \}$
while $k-1 < n$ ***do***
    $s := s+k$;
    $k := k+1$
od
$\{ s = sum(0, n) \}$

Proofs and Proof Outlines for Partial Correctness

Part 1: Full Proofs and Proof Outlines of Partial Correctness

CS 536: Science of Programming, Spring 2023

A. Why

- A formal proof lets us write out in detail the reasons for believing that something is valid.
- Proof outlines condense the same information as a proof.

B. Objectives

At the end of this class you should

- Know how to write and check a formal proof of partial correctness.
- Know how to translate between full formal proofs and full proof outlines

C. Formal Proofs of Partial Correctness

- As you've seen, the format of a formal proof is very rigid syntactically. The relationship between formal proofs and informal proofs is like the description of an algorithm in a program (very rigid syntax) versus in pseudocode (much more informal syntax).
- Just as a reminder, we're using Hilbert-style proofs: Each line's assertion is an assumption, an axiom, or follows by some rule that appeals to earlier lines in the proof. In high-school geometry, we might have used

1.	Length of AB=length of XY	Assumption
2.	Angle ABC=Angle XYZ	Assumption
3.	Length of BC=length of YZ	Assumption
4.	Triangles ABC, XYZ are congruent	Side-Angle-Side, lines 1, 2, 3

D. Sample Formal Proofs

- We can write out the reasoning for the sample summation loop we looked at. We've seen formal proofs of the loop body's correctness; all we really have to do is attach the proof of loop initialization correctness:

Example 1: Simple summation program

$\{ n \geq 0 \}$
$k := 0 ;\ s := 0 ;$
$\{ \textbf{inv}\ p_1 \equiv 0 \leq k \leq n \wedge s = sum(0, k) \}$
while $k < n$ ***do***
  $s := s+k+1 ;\ k := k+1$
od
$\{ s = sum(0, n) \}$

- Below, let $S_1 \equiv s := s+k+1 ;\ k := k+1$ (the loop body) and let $W \equiv$ ***while*** $k < n$ ***do*** S_1 ***od*** (the loop).

1.	$\{ n \geq 0 \}\ k := 0\ \{ n \geq 0 \wedge k = 0 \}$	assignment (forward)
2.	$\{ n \geq 0 \wedge k = 0 \}\ s := 0\ \{ n \geq 0 \wedge k = 0 \wedge s = 0 \}$	assignment (forward)
3.	$\{ n \geq 0 \}\ k := 0 ;\ s := 0\ \{ n \geq 0 \wedge k = 0 \wedge s = 0 \}$	sequence 1, 2
4.	$n \geq 0 \wedge k = 0 \wedge s = 0 \rightarrow p_1$ where $p_1 \equiv 0 \leq k \leq n \wedge s = sum(0, k)$	predicate logic
5.	$\{ n \geq 0 \}\ k := 0 ;\ s := 0\ \{ p_1 \}$	postcondition weakening, 3, 4
6.	$\{ p_1[k+1 / k] \}\ k := k+1\ \{ p_1 \}$	assignment (backward)
7.	$\{ p_1[k+1 / k][s+k+1 / s] \}\ s := s+k+1\ \{ p_1[k+1 / k] \}$	assignment (backward)
8.	$\{ p_1[k+1 / k][s+k+1 / s] \}\ S_1\ \{ p_1 \}$	sequence 7, 6
9.	$p_1 \wedge k < n \rightarrow p_1[k+1 / k][s+k+1 / s]$	predicate logic
10.	$\{ p_1 \wedge k < n \}\ S_1\ \{ p_1 \}$	precondition strengthening, 9, 8
11.	$\{ \textbf{inv}\ p_1 \}$ ***while*** $k < n$ ***do*** S_1 ***od*** $\{ p_1 \wedge k \geq n \}$	while loop, 10
12.	$\{ n \geq 0 \}\ k := 0 ;\ s := 0 ;\ W\ \{ p_1 \wedge k \geq n \}$ (where W is the loop in line 11)	sequence 5, 11
13.	$p_1 \wedge k \geq n \rightarrow s = sum(0, n)$	predicate logic
14.	$\{ n \geq 0 \}\ k := 0 ;\ s := 0 ;\ W\ \{ s = sum(0, n) \}$	postcond. weakening, 12, 13

- The proof uses two substitutions:
  - $p_1[k+1 / k] \equiv 0 \leq k+1 \leq n \wedge s = sum(0, k+1)$
  - $p_1[k+1 / k][s+k+1 / s] \equiv (0 \leq k \leq n \wedge s = sum(0, k+1))[s+k+1 / s]$
    $\equiv\ 0 \leq k+1 \leq n \wedge s+k+1 = sum(0, k+1)$
- The proof also gives us three predicate logic obligations (implications we need to be true, otherwise the overall proof is incorrect). Happily, all three are in fact valid.
  - $n \geq 0 \wedge k = 0 \wedge s = 0 \rightarrow p_1$
    $\equiv\ n \geq 0 \wedge k = 0 \wedge s = 0 \rightarrow 0 \leq k \leq n \wedge s = sum(0, k)$
  - $p_1 \wedge k < n \rightarrow p_1[k+1 / k][s+k+1 / s]$
    $\equiv\ (0 \leq k \leq n \wedge s = sum(0, k)) \wedge k < n \rightarrow 0 \leq k+1 \leq n \wedge s+k+1 = sum(0, k+1)$
  - $p_1 \wedge k \geq n \rightarrow s = sum(0, n)$
    $\equiv\ (0 \leq k \leq n \wedge s = sum(0, k)) \wedge k \geq n \rightarrow s = sum(0, n)$

- To review, the order of the lines in the proof is somewhat arbitrary — you can only refer to lines above you in the proof, but they can be anywhere above you.
  - For example, lines 1 and 2 don't have to be in that order, they just have to be before we use them in the sequence rule at line 3 (which in turn has to be somewhere before line 5, and so on).

E. Full Proof Outlines

- Formal proofs are long and contain repetitive information (we keep copying the same conditions over and over). All in all, they're too tedious to use.
- A ***proof outline*** is a way to write out all the information that you would need to generate a full formal proof, but with less repetition, so they're much shorter, and they don't mask the overall structure of the program the way a full proof does.
  - To get a proof outline, we annotate program statements with their preconditions and postconditions, so that every statement in the program is part of one or correctness triples.
    - Every triple must be provable using the proof rules.
    - We include all statements, not just basic ones like assignments and ***skip***.

Proof Outlines for Individual Statements

- Each instance of a proof rule corresponds to a proof outline that combines the antecedents (if any) and consequent of the rule. (For a loop, the loop body, for conditionals, each branch.)

Assignment and skip

- These triples are annotated exactly as they are in the proof rules.
  - $\{p\}\, x := e\, \{q\}$
  - $\{p\}$ ***skip*** $\{p\}$

Sequence

- To combine $\{p_1\}\, S_1\, \{q\}$ and $\{q\}\, S_2\, \{q_1\}$ to get $\{p_1\}\, S_1\, ;\, S_2\, \{q_1\}$, we include the condition q that sits between S_1 and S_2:
  - $\{p_1\}\, S_1\, ; \{q\}\, S_2\, \{q_1\}$

While loops

- There is only one loop rule hence only one triple. It combines triple for the body, $\{p \wedge B\}\, S\, \{p\}$, and the triple for the overall statement, $\{$ ***inv*** $p\}$ ***while*** B ***do*** S ***od*** $\{p \wedge \neg B\}$.
  - $\{$ ***inv*** $p\}$ ***while*** B ***do*** $\{p \wedge B\}\, S\, \{p\}$ ***od*** $\{p \wedge \neg B\}$

Conditionals

- There are multiple possibilities for conditionals because we have multiple rules for them. Each outline includes the triples for the branches and the triple for the overall conditional statement.
  - $\{p\}$ ***if*** B ***then*** $\{p \wedge B\}\, S_1\, \{q_1\}$ ***else*** $\{p \wedge \neg B\}\, S_2\, \{q_2\}$ ***fi*** $\{q_1 \vee q_2\}$
  - $\{(B \rightarrow p_1) \wedge (\neg B \rightarrow p_2)\}$ ***if*** B ***then*** $\{p_1\}\, S_1\, \{q_1\}$ ***else*** $\{p_2\}\, S_2\, \{q_2\}$ ***fi*** $\{q_1 \vee q_2\}$

- $\{p\}$ **if** $B_1 \to \{p \wedge B_1\} S_1 \{q_1\} \ \square \ B_2 \to \{p \wedge B_2\} S_2 \{q_2\}$ **fi** $\{q_1 \vee q_2\}$
- $\{(B_1 \to p_1) \wedge (B_2 \to p_2\}$ **if** $B_1 \to \{p_1\} S_1 \{q_1\} \ \square \ B_2 \to \{p_2\} S_2 \{q_2\}$ **fi** $\{q_1 \vee q_2\}$

Strengthening and Weakening

- For strengthening or weakening operations, we include a condition for the new condition, next to the condition it replaces:
  - $\{p_1\}\{p\} S \{q\}$ For strengthening using $p_1 \to p$
  - $\{p\} S \{q\}\{q_1\}$ For weakening using $q \to q_1$.
- Just generally in an outline, if two conditions sit next to each other, say $\{p\}\{q\}$, this indicates a predicate logic implication $p \to q$.

Full Outlines Aren't Unique

- A proof outline does not stand for a unique proof. (Unless you have a one-line proof.)
  - One reason is pretty trivial: If a rule has more than one antecedent, they can be shown in any order. I.e., for a conditional, the triples for the true branch and false branch can appear in that order or the reverse.
  - The other reason is that strengthening and weakening operations within a sequence aren't unique. The overall proof ends up with the same triple, but the path there might be different.
  - E.g., take $\{p_1\} S_1 ; \{p_2\}\{p_3\} S_2 \{p_4\}$. We can read this as
    - Weakening the postcondition of S_1 from p_2 to p_3 or
    - Strengthening the precondition of S_2 from p_3 to p_2
  - Luckily, the difference is hardly ever a problem. It's often just a style issue[*].

Example 1

- One kind of problem to study is "What is the full proof that corresponds to this outline?"
- E.g., what is the outline for $\{T\}\ k := 0 ; \{k = 0\}\ x := 1\ \{k = 0 \wedge x = 1\}\{k \geq 0 \wedge x = 2 \wedge k\}$?
- The basic structure is that we form the sequence $k := 0 ;\ x := 1$ and then weaken its postcondition.

1.	$\{T\}\ k := 0\ \{k = 0\}$	assignment (forward)
2.	$\{k = 0\}\ x := 1\ \{k = 0 \wedge x = 1\}$	assignment (forward)
3.	$\{T\}\ k := 0 ;\ x := 1\ \{k = 0 \wedge x = 1\}$	sequence 1, 2
4.	$k = 0 \wedge x = 1 \to k \geq 0 \wedge x = 2 \wedge k$	predicate logic
5.	$\{T\}\ k := 0 ;\ x := 1\ \{k \geq 0 \wedge x = 2 \wedge k\}$	postcondition weakening 3, 4

[*] The weakened or strengthened triple might look nicer than the other. Also, if one of S_1 or S_2 is more painful to write, both proofs involve writing one of S_1 and S_2 once and the other twice.

Example 2

- This is like Example 1 but uses weakest preconditions instead of strongest postconditions.
- The full proof outline is $\{T\}\,\{0 \geq 0 \wedge 1 = 2 \hat{} 0\}\; k := 0;\{k \geq 0 \wedge 1 = 2 \hat{} k\}\; x := 1\,\{k \geq 0 \wedge x = 2 \hat{} k\}$.

1.	$\{k \geq 0 \wedge 1 = 2 \hat{} k\}\; x := 1\; \{k \geq 0 \wedge x = 2 \hat{} k\}$	assignment (backward)
2.	$\{0 \geq 0 \wedge 1 = 2 \hat{} 0\}\; k := 0\; \{k \geq 0 \wedge 1 = 2 \hat{} k\}$	assignment (backward)
3.	$\{0 \geq 0 \wedge 1 = 2 \hat{} 0\}\; k := 0;\; x := 1\; \{k \geq 0 \wedge x = 2 \hat{} k\}$	sequence 2, 1
4.	$T \rightarrow 0 \geq 0 \wedge 1 = 2 \hat{} 0$	predicate logic
5.	$\{T\}\; k := 0;\; x := 1\; \{k \geq 0 \wedge x = 2 \hat{} k\}$	pre. strength. 4, 3

Example 3

- Here's a full proof outline for the summation loop; note how the structure of the outline follows the partial correctness proof, which is shown below.

$\{n \geq 0\}\; k := 0;\{n \geq 0 \wedge k = 0\}\; s := 0;\{n \geq 0 \wedge k = 0 \wedge s = 0\}$
$\{\textbf{inv}\ p_1 \equiv 0 \leq k \leq n \wedge s = sum(0, k)\}$
while $k < n$ ***do***
  $\{p_1 \wedge k < n\}\{p_1[k+1 / k][s+k+1 / s]\}$
  $s := s+k+1;\{p_1[k+1 / k]\}$
  $k := k+1\,\{p_1\}$
od
$\{p_1 \wedge k \geq n\}$
$\{s = sum(0, n)\}$

- A full proof is below

1.	$\{n \geq 0\}\; k := 0\; \{n \geq 0 \wedge k = 0\}$	assignment (forward)
2.	$\{n \geq 0 \wedge k = 0\}\; s := 0\; \{n \geq 0 \wedge k = 0 \wedge s = 0\}$	assignment (forward)
3.	$\{n \geq 0\}\; k := 0;\; s := 0\; \{n \geq 0 \wedge k = 0 \wedge s = 0\}$	sequence 1, 2
4.	$n \geq 0 \wedge k = 0 \wedge s = 0 \rightarrow p_1$	predicate logic
5.	$\{n \geq 0\}\; k := 0;\; s := 0\; \{p_1\}$	post. weakening 3, 4
6.	$\{p_1[k+1 / k]\}\; k := k+1\; \{p_1\}$	assignment (backward)
7.	$\{p_1[k+1 / k][s+k+1 / s]\}\; s := s+k+1\; \{p_1[k+1 / k]$	assignment (backward)
8.	$\{p_1[k+1 / k][s+k+1 / s]\}\; s := s+k+1;\; k := k+1\; \{p_1\}$	sequence 7, 6
9.	$p_1 \wedge k < n \rightarrow p_1[k+1 / k][s+k+1 / s]$	predicate logic
10.	$\{p_1 \wedge k < n\}\; s := s+k+1;\; k := k+1\; \{p_1\}$	pre. strength. 9, 8
11.	$\{\textbf{inv}\ p_1\}\; W\; \{p_1 \wedge k \geq n\}$ where $W \equiv$ ***while*** $k < n$ ***do*** $s := s+k+1;\; k := k+1$ ***od***	while loop 10
12.	$\{n \geq 0\}\; k := 0;\; s := 0;\{\textbf{inv}\ p_1\}\; W\; \{p_1 \wedge k \geq n\}$	sequence 5, 11
13.	$p_1 \wedge k \geq n \rightarrow s = sum(0, n)$	predicate logic
14.	$\{n \geq 0\}\; k := 0; s := 0;\{\textbf{inv}\ p_1\}\; W\; \{s = sum(0, n)\}$	post. weak. 12, 13

Proof Outlines for Partial Correctness, v.2

Part 2: Partial and Minimal Proof Outlines

CS 536: Science of Programming, Spring 2023

2023-03-20, 20:00 - v.2

A. Why

- A formal proof lets us write out in detail the reasons for believing that something is valid.
- Proof outlines condense the same information as a proof.

B. Objectives

At the end of this class you should

- Know the structure of full proof outlines and formal proofs and how they are related.
- Know the difference between full, partial, and minimal proof outlines and how they are related.

C. Minimal Proof Outlines

- In a ***full proof outline*** of correctness, we include all the triples found in a formal proof of correctness, but we omit much of the redundant text, which makes them ***much*** easier to work with than formal proofs. But if you think about it, you'll realize that we can shorten the outline by omitting conditions that can be inferred to exist from the structure of the program.
- In a ***minimal proof outline***, we have the minimum amount of program annotation that allows us to infer the rest of the formal proof outline. In general, we can't infer the initial precondition and initial postcondition, nor can we infer the invariants of loops, so a minimal outline will include those conditions and possibly no others.
- A ***partial proof outline*** is somewhere in the middle: More filled-in than a minimal outline but not completely full.
- ***Example 1***: Here's a full proof outline from the previous class, with the removable parts *in blue*. (The outline comes from the previous class.)

$\{ n \geq 0 \}$

$k := 0$; $\{ n \geq 0 \wedge k = 0 \}$ // *Inferred as the sp of k := 0*

$s := 0$; $\{ n \geq 0 \wedge k = 0 \wedge s = 0 \}$ // *Inferred as the sp of s := 0*

$\{$ **inv** $p_1 \equiv 0 \leq k \leq n \wedge s = sum(0, k) \}$ // *The invariant remains — it can't be inferred*

while $k < n$ **do**

  $\{ p_1 \wedge k < n \}$ // *Loop rule requires inv ∧ loop test here*

  $\{ p_1[k+1/k][s+k+1/s] \}$ // *Inferred as the wp of s:=s+k+1*

  $s := s+k+1$; $\{ p_1[k+1/k] \}$ // *Inferred as the wp of k:=k+1*

$k := k+1$ $\{p_1\}$ *// Loop rule requires invariant at end of loop body*

od

$\{p_1 \wedge k \geq n\}$ *// Loop rule requires inv $\wedge$ $\neg$ loop test after the loop*

$\{s = sum(0,n)\}$ [Mon 2023-03-20, 14:04 not inferable]

- Dropping the inferable parts leaves us with the minimal outline:

  $\{n \geq 0\}\ k := 0;\ s := 0;$
  $\{$***inv*** $p_1 \equiv 0 \leq k \leq n \wedge s = sum(0,k)\}$
  while $k < n$ ***do***
  $s := s+k+1;\ k := k+1$
  od
  $\{s = sum(0,n)\}$

- In a language like C or Java, the conditions become comments; something like::

```
// Assume: n ≥ 0
int k, s;           // 0 ≤ k ≤ n and s = sum(0,k)
k = s = 0;          // establish k,s
while (k < n) {
   s += k+1;        // reset s
   ++k;             // Get closer to termination; reestablish k, s
}
// Established: s = sum(0,n)
```

- The following example shows how different total proof outlines can all have the same minimal proof outline.

- ***Example 2:*** The three full proof outlines below all have the same minimal proof outline, namely, $\{T\}\ k := 0;\ x := 1\ \{k \geq 0 \wedge x = 2 \hat{} k\}$

  $\{T\}\ \{0 \geq 0 \wedge 1 = 2 \hat{} 0\}\ k := 0;\ \{k \geq 0 \wedge 1 = 2 \hat{} k\}\ x := 1\ \{k \geq 0 \wedge x = 2 \hat{} k\}$

  $\{T\}\ k := 0;\ \{k = 0\}\ x := 1\ \{k = 0 \wedge x = 1\}\ \{k \geq 0 \wedge x = 2 \hat{} k\}$

  $\{T\}\ k := 0;\ \{k = 0\}\ \{k \geq 0 \wedge 1 = 2 \hat{} k\}\ x := 1\ \{k \geq 0 \wedge x = 2 \hat{} k\}$

- The reason multiple full proof outlines can have the same minimal outline is because different organizations of *wp* and *sp* can have the same minimal outline. There can also be differences in whether and where preconditions are strengthened or postconditions are weakened.

- ***Example 3:*** For the three full outlines below, the minimal outline is the same:
  $\{y = x\}$ ***if*** $x < 0$ ***then*** $y := -x$ ***fi*** $\{y = abs(x)\}$.

- The first outline uses *wp* on the branches of the conditional:

  $\{y = x\}$
  if $x < 0$ ***then***

$\{y = x \wedge x < 0\}\{-x = abs(x)\}\ y := -x\ \{y = abs(x)\}$

else

$\{y = x \wedge x \geq 0\}\{y = x \wedge x \geq 0\}$ ***skip*** $\{y = abs(x)\}$

fi

$\{y = abs(x)\}$

- The second outline uses *sp* on the branches of the conditional. Using *sp* makes the predicate logic obligation for the true branch a bit more complicated.

$\{y = x\}$

if $x < 0$ ***then***

$\{y = x \wedge x < 0\}\ y := -x\ \{y_0 = x \wedge x < 0 \wedge y = -x\}\{y = abs(x)\}$

else

$\{y = x \wedge x \geq 0\}$ ***skip*** $\{y = x \wedge x \geq 0\}\{y = abs(x)\}$

fi

$\{y = abs(x)\}$

- The third full outline calculates the *wp* of the conditional. There's only one predicate logic obligation, but it contains the same information as the two obligations of the previous outlines.

$\{y = x\}$

$\{(x < 0 \rightarrow -x = abs(x)) \wedge (x \geq 0 \rightarrow y = abs(x))\}$ *// wp of the if-else*

if $x < 0$ ***then***

$\{-x = abs(x)\}\ y := -x\ \{y = abs(x)\}$

else

$\{y = abs(x)\}$ ***skip*** $\{y = abs(x)\}$

fi

$\{y = abs(x)\}$

// end of example e

- ***Example 4:*** The minimal proof outline for

$\{n \geq 0\}\ k := n;\ \{n \geq 0 \wedge k = n\}\ s := n;\ \{n \geq 0 \wedge k = n \wedge s = n\}$

$\{$***inv*** $p \equiv 0 \leq k \leq n \wedge s = sum(k, n)\}$

while $k > 0$ ***do***

$\{p \wedge k > 0\}\{p[s+k/s][k-1/k]\}$

$k := k-1;\ \{p[s+k/s]\}$

$s := s+k\ \{p\}$

od

$\{p \wedge k \leq 0\}\{s = sum(0, n)\}$

is

$\{ n \geq 0 \}\ k := n ;\ s := n ;$
$\{ \textbf{inv}\ p \equiv 0 \leq k \leq n \wedge s = sum(k, n) \}$
while $k > 0$ ***do***
    $k := k - 1 ;\ s := s + k$
od
$\{ s = sum(0, n) \}$

D. Expanding Partial Proof Outlines

- To expand a partial proof outline into a full proof outline, basically we need to infer all the missing conditions. Postconditions are inferred from preconditions using $sp(\ldots)$, and preconditions are inferred from postconditions using $wp(\ldots)$. Loop invariants tell us how to annotate the loop body and postcondition, and the test for a conditional statement can become part of a precondition.
- Expanding a partial outline can lead to a number of different full outlines, but all the full outlines will be correct, and the differences between them are generally stylistic. Expansion can have different results because multiple full outlines can have the same minimal outline.
- For example, $\{ p \}\ v := e\ \{ q \}$ can be expanded to $\{ p \}\ \{ wp(v := e, q) \}\ v := e\ \{ q \}$ or $\{ p \}\ v := e\ \{ sp(p, v := e) \}\ \{ q \}$.
- The situation similar to how a full proof outline can expand to various formal proofs, all of which are correct but can be slightly different The different full outlines here are actually different, though generally only in small ways.
- So we can't have a deterministic algorithm for expanding minimal outlines, but with that warning, here's an informal nondeterministic algorithm. Added conditions are shown *in blue*.
- ***Notation:***
  - In $\{ p \}\ \{ \ldots \}\ S\ \{ \ldots \}$ and $\{ \ldots \}\ \{ p \}\ S\ \{ \ldots \}$, p is the first or last precondition of S respectively.
  - In $\{ p \}\ \{ \ldots \}\ S\ \{ \ldots \}$ and $\{ \ldots \}\ \{ p \}\ S\ \{ \ldots \}$, we add p as the new first or last precondition of S.
  - In $\{ \ldots \}\ S\ \{ p \}\ \{ \ldots \}$ and $\{ \ldots \}\ S\ \{ \ldots \}\ \{ p \}$, p is the first or last postcondition of S.
  - In $\{ \ldots \}\ S\ \{ p \}\ \{ \ldots \}$ and $\{ \ldots \}\ S\ \{ \ldots \}\ \{ p \}$, we add p as the new first or last postcondition of S.

- ***The algorithm:***

  Until every statement can be proved by a triple, apply one of the cases below:

  A. Add a precondition:

  1. Add wp to an assignment: $\{ \ldots \}\ \{ wp(v := e, q) \}\ v := e\ \{ q \}\ \{ \ldots \}$.
  2. Add wp to a ***skip***: $\{ \ldots \}\ \{ q \}$ ***skip*** $\{ q \}\ \{ \ldots \}$.
  3. Add precondition to second statement of a sequence: $S_1 ;\ \{ \ldots \}\ \{ p \}\ S_2\ \{ \ldots \}$.
  4. Add strongest preconditions to the branches of an ***if-else***:

     $\{ p \}$ ***if*** B ***then*** $\{ p \wedge B \}\ \{ \ldots \}\ S_1\ \{ \ldots \}$ ***else*** $\{ p \wedge \neg B \}\ \{ \ldots \}\ S_2\ \{ \ldots \}$ ***fi***.

The nondeterministic version of this is

$\{p\}$ **if** $B_1 \to \{p \wedge B_1\}\{\dots\} S_1 \{\dots\} \square B_2 \to \{p \wedge B_2\}\{\dots\} S_2 \{\dots\}$ **fi**.

5. Add a precondition to an ***if-else.***

   $\{\dots\}$ $(B \to p_1) \wedge (\neg B \to p_2)$ **if** B **then** $\{p_1\}\{\dots\} S_1$ … **else** $\{p_2\}\{\dots\} S_2$ … **fi**.

   If p_1 and p_2 are the *wp* of S_1 and S_2 respectively, then the new addition is the *wp* of the conditional.

   The nondeterministic version of this is

   $\{\dots\}$ $(B_1 \to p_1) \wedge (B_2 \to p_2)$ **if** $B_1 \to \{p_1\}\{\dots\} S_1$ … $\square$ $B_2 \to \{p_2\}\{\dots\} S_2$ … **fi**.

B. Or add a postcondition:

6. Add *sp* to an assignment: $\{p\}\, v := e\, \{sp(p, v := e)\}\{\dots\}$
7. Add *sp* to a ***skip***: $\{\dots\}\{p\}$ **skip** $\{p\}\{\dots\}$
8. Add a postcondition to the first statement of a sequence: $\{\dots\} S_1 \{q\}; \{\dots\} S_2$.

   We add q just after S_1; if we want q just before S_2, we can add it as a new precondition of S_2 (step 3 above).
9. Add a postcondition to a conditional statement:

   if B **then** $S_1\{\dots\}\{q_1\}$ **else** $S_2\{\dots\}\{q_2\}$ **fi** $\{q_1 \vee q_2\}\{\dots\}$.

   If q_1 and q_2 are the *sp* of S_1 and S_2, then we are adding the *sp* of the conditional.

   The nondeterministic version of this is

   if $B_1 \to S_1\{\dots\}\{q_1\} \square B_2 \to S_2\{\dots\}\{q_2\}$ **fi** $\{q_1 \vee q_2\}\{\dots\}$.
10. Add postconditions to the branches of a conditional statement:

    if B **then** $S_1\{\dots\}\{q_1\}$ **else** $S_2\{\dots\}\{q_2\}$ **fi** $\{q_1 \vee q_2\}\{\dots\}$.

    The nondeterministic version of this is

    if $B_1 \to S_1\{\dots\}\{q_1\} \square B_2 \to S_2\{\dots\}\{q_2\}$ **fi** $\{q_1 \vee q_2\}\{\dots\}$.

C. Or add loop conditions:

11. Add loop body pre-and post-conditions and a loop postcondition:

    $\{$**inv** $p\}$ **while** B **do** $\{p \wedge B\}\{\dots\} S_1 \{\dots\}\{p\}$ **od** $\{p \wedge \neg B\}\{\dots\}$.

D. Or strengthen or weaken some condition:

12. Strengthen q: … $\{p\}\{q\}$ … where $p \to q$.
13. Weaken p: … $\{p\}\{q\}$ … where $p \to q$.

// End loop

- ***Example 4 reversed:*** Let's expand

  $\{n \geq 0\}\, k := n;\ s := n;$
  $\{$**inv** $p \equiv 0 \leq k \leq n \wedge s = sum(k, n)\}$
  while $k > 0$ **do**

$k := k–1$;
$s := s+k$
od
$\{ s = sum(0, n) \}$

- First, we can apply case 6 (*sp* of an assignment) to $k := n$ and to $s := n$ to get

$\{ n \geq 0 \}\ k := n\ ;\ \{ n \geq 0 \wedge k = n \}\ s := n\ ;\ \{ n \geq 0 \wedge k = n \wedge s = n \}$
$\{$ **inv** $p \equiv 0 \leq k \leq n \wedge s = sum(k, n) \}$
while $k > 0$ **do**
$k := k–1$;
$s := s+k$
od
$\{ s = sum(0, n) \}$

- The next three steps are independent of the first two steps we took: First, apply case 11 to the loop:

$\{ n \geq 0 \}\ k := n\ ;\ \{ n \geq 0 \wedge k = n \}\ s := n\ ;\ \{ n \geq 0 \wedge k = n \wedge s = n \}$
$\{$ **inv** $p \equiv 0 \leq k \leq n \wedge s = sum(k, n) \}$
while $k > 0$ **do**
$\{ p \wedge k > 0 \}$
$k := k–1$;
$s := s+k$
$\{ p \}$
od
$\{ p \wedge k \leq 0 \}$
$\{ s = sum(0, n) \}$

- Then apply case 1 (*wp* of an assignment) to $s := s+k$ and to $k := k–1$:

$\{ n \geq 0 \}\ k := n\ ;\ \{ n \geq 0 \wedge k = n \}\ s := n\ ;\ \{ n \geq 0 \wedge k = n \wedge s = n \}$
$\{$ **inv** $p \equiv 0 \leq k \leq n \wedge s = sum(k, n) \}$
while $k > 0$ **do**
$\{ p \wedge k > 0 \}$ // $\equiv 0 \leq k \leq n \wedge s = sum(k, n) \wedge k > 0$
$\{ p[s+k/s][k–1/k] \}$ // $\equiv 0 \leq k–1 \leq n \wedge s+(k–1) = sum(k–1, n)$
$k := k – 1$;
$\{ p[s+k/s] \}$ // $\equiv 0 \leq k \leq n \wedge s+k = sum(k, n)$
$s := s+k$
$\{ p \}$
od

$\{ p \wedge k \le 0 \}$
$\{ s = sum(0, n) \}$

- And this finishes the expansion.
- Note using *sp* on the loop assignments works too:

$\{ n \ge 0 \}\ k := n ; \{ n \ge 0 \wedge k = n \}\ s := n ; \{ n \ge 0 \wedge k = n \wedge s = n \}$
$\{ \textbf{inv}\ p \equiv 0 \le k \le n \wedge s = sum(k, n) \}$
while $k > 0$ ***do***
  $\{ p \wedge k > 0 \}$
  $k := k-1 ;$
  $\{ p[k_0/k] \wedge k_0 > 0 \wedge k = k_0-1 \}$ // $0 \le k_0 \le n \wedge s = sum(k_0, n) \wedge k_0 > 0 \wedge k = k_0-1$
  $s := s+k$
  $\{ p[k_0/k][s_0/s] \wedge k_0 > 0 \wedge k = k_0-1 \wedge s = s_0+k \}$ // Notice: s_0+k not s_0+k_0
    // $\equiv 0 \le k_0 \le n \wedge s_0 = sum(k_0, n) \wedge k_0 > 0 \wedge k = k_0-1 \wedge s = s_0+k$
    // $\Rightarrow 0 \le k \le n \wedge s = sum(k, n) \equiv p$
  $\{ p \}$
od
$\{ p \wedge k \le 0 \}$
$\{ s = sum(0, n) \}$

Other Features of Expansion

- In Example 2, we saw that a number of full proof outlines can have the same minimal proof outline. The inverse is that a partial proof outline might expand into a number of different full proof outlines. Which one to use is pretty much a style issue.
- ***Example 5:*** In Example 4 reversed, we took

  $\{ n \ge 0 \}\ k := n ;\ s := n\ \{ p \equiv 0 \le k \le n \wedge s = sum(k, n) \}$

  and applied case 6 (*sp*) to both assignments to get

  $\{ n \ge 0 \}\ k := n ;\ \{ n \ge 0 \wedge k = n \}\ s := n ; \{ n \ge 0 \wedge k = n \wedge s = n \}\ \{ p \}$
- Another possibility would have been to use case 1 (*wp*) on both assignments; we would have gotten

  $\{n \ge 0\} \{0 \le n \le n \wedge n = sum(n,n)\}\ k := n; \{0 \le k \le n \wedge n = sum(k,n)\}\ s := n\ \{0 \le k \le n \wedge s = sum(k,n)\}$
- Or we could have used case 6 (*sp*) on the first assignment and case 1 (*wp*) on the second:

  $\{ n \ge 0 \}\ k := n ; \{ n \ge 0 \wedge k = n \}\ \{ 0 \le k \le n \wedge n = sum(k, n) \}\ s := n\ \{ p \}$
- The three versions produce slightly different predicate logic obligations, all easy to prove.
  - *sp* and *sp:* $n \ge 0 \wedge k = n \wedge s = n \rightarrow 0 \le k \le n \wedge s = sum(k, n)$
  - *wp* and *wp:* $n \ge 0 \rightarrow 0 \le n \le n \wedge n = sum(n, n)$
  - *sp* and *wp:* $n \ge 0 \wedge k = n \rightarrow 0 \le k \le n \wedge n = sum(k, n)$

- Similarly, with a conditional triple $\{p\}$ **if** B **then** $\{p_1\}\, S_1$ **else** $\{p_2\}\, S_2$ **fi**, we can get
  - With case 4: $\{p\}$ **if** B **then** $\{p \wedge B\}\, \{p_1\}\, S_1$ **else** $\{p \wedge \neg B\}\, \{p_2\}\, S_2$ **fi**
  - Or with case 5: $\{p\}\, \{(B \to p_1) \wedge (\neg B \to p_2)\}$ **if** B **then** $\{p_1\}\, S_1$ **else** $\{p_2\}\, S_2$ **fi**
- We get different predicate logic obligations for the two approaches, but the obligations are basically equally difficult to prove.
  - With case 4: $p \wedge B \to p_1$ and $p \wedge \neg B \to p_2$
  - With case 5: $p \to (B \to p_1) \wedge (\neg B \to p_2)$

Total Correctness: Avoiding Errors and Divergence

CS 536: Science of Programming, Spring 2023

2023-03-22: pp. 2,3,6-8; 2023-04-06: pp.2, 3

A. Why

- To argue that a program is totally correct, we need it to be partially correct, avoid runtime errors, and not diverge.
- To avoid runtime errors, we use domain predicates.
- To show that a loop terminates, we define a weak upper bound for the number of iterations left.

B. Objectives

At the end of this class you should understand

- How to add domain checks to partial correctness arguments.
- The loop bound method of ensuring termination.
- How to extend proofs of partial correctness to total correctness.

C. Rules for Total Correctness

- Up until now, we've been studying rules for partial correctness. Recall when we started to look at proof rules for correctness triples (back in Class 14), we had the definition for a general proof system. We can be more specific now and say:
  - ***Definition***: A ***proof system for partial correctness*** is the set of logical formulas determined by the sets of axioms and rules of inference for partial correctness.
  - ***Notation:*** $\vdash \{ p \} S \{ q \}$ means the correctness triple can be proved to be partially correct, using the rules we've seen. Since so far all we've been discussing is partial correctness, there hasn't been much call for the $\vdash$ notation, and we've been just saying "$\{ p \} S \{ q \}$ is provable" instead of "$\vdash \{ p \} S \{ q \}$".
- Now we'll look at proving total correctness. We'll take the rules for partial correctness and add avoidance of runtime errors and divergence of loops, and we'll have
  - ***Definition***: A ***proof system for total correctness*** is the set of logical formulas determined by the sets of axioms and rules of inference for total correctness.
  - ***Notation:*** $\vdash_{\text{tot}} \{ p \} S \{ q \}$ means the correctness triple can be proved to be totally correct, using the rules we've seen.

D. Avoiding Runtime Errors

- In class 11, we looked at domain predicates for expressions, statements, and predicates: $D(e)$, $D(S)$, and $D(p)$ guarantees that evaluation of e, S, or p won't cause a runtime error.

- One basic differences between rules for $\vdash$ and $\vdash_{tot}$ is that when a predicate p appears in a condition, we need it to be safe in the sense that $D(p)$ also holds. This comes up often enough that it's worth some notation:
  - ***Definition:*** A predicate p is ***safe*** if $p \to D(p)$. A correctness triple $\{p\}\,S\,\{q\}$ is ***safe*** if p and q are safe. The ***safe version*** of p, written $\downarrow p$, is $p \wedge D(p)$ and the ***safe version*** of $\{p\}\,S\,\{q\}$ is $\{\downarrow p\}\,S\,\{\downarrow q\}$. [We won't prove this, but if p is safe, then $D(p)$ is safe.]
- The second basic difference between rules for $\vdash$ and $\vdash_{tot}$ is that to conclude $\vdash_{tot} \{\downarrow p\}\,S\,\{\downarrow q\}$, we need the precondition to imply $D(S)$.
- In general, we have that for partial correctness, $\{wlp(S, q)\}\,S\,\{q\}$. For total correctness, $\{wp(S, \downarrow q)\}\,S\,\{\downarrow q\}$, where $wp(S, \downarrow q) \Leftrightarrow D(S) \wedge \downarrow wlp(S, \downarrow q)$. [2023-03-22]
  - (Note in class 11, we had $wp(S, q) \Leftrightarrow D(S) \wedge wlp(S, q) \wedge D(wlp(S, q)).$)[1]
- ***Definition:*** In outline form, the total correctness rules for the non-loop statements are:
  - $\{\downarrow p\}$ ***skip*** $\{\downarrow p\}$
  - $\{D(e) \wedge \downarrow wlp(x := e, \downarrow p)\}\ x := e\ \{\downarrow p\}$ [2023-03-22] the wp(x:=e,↓p) → D(e)]
  - $\{D(e) \wedge \downarrow p \wedge x = x_0\}\ x := e\ \{(D(e) \wedge \downarrow p)[x_0/x] \wedge x = e[x_0/x]\}$.
  - $\{D(S_1) \wedge \downarrow p\}\,S_1\,;\{D(S_2) \wedge \downarrow q\}\,S_2\,\{\downarrow r\}$.
  - $\{\downarrow B \wedge (\downarrow B \to D(S_1)) \wedge (\downarrow \neg B \to D(S_2)))\}$ [2023-03-22] include a precondition p?
    if B ***then*** $\{\downarrow B \wedge D(S_1)\}\,S_1\,\{\downarrow q_1\}$ ***else*** $\{\downarrow \neg B \wedge D(S_2)\}\,S_2\,\{\downarrow q_2\}$ ***fi*** $\{\downarrow q_1 \wedge \downarrow q_2\}$.
- ***Example 1***: What wp can we use for p in $\{p\}\,x := sqrt(y/z)\,\{x^2 \le x\}$?
  - The postcondition is already safe, so it needs no modification.
  - For $S \equiv x := sqrt(y/z)$, we have $D(S) \Leftrightarrow z \neq 0 \wedge y/z \ge 0$.
  - $w \equiv wlp(S, x^2 \le x) \Leftrightarrow sqrt(y/z)^2 \le sqrt(y/z)$ [2023-04-06] and $D(w) \Leftrightarrow z \neq 0 \wedge y/z \ge 0$
  - So $p \equiv D(S) \wedge w \wedge D(w)$
    $\Leftrightarrow (z \neq 0 \wedge y/z \ge 0) \wedge (sqrt(y/z)^2 \le y/z) \wedge (z \neq 0 \wedge y/z \ge 0)$
    $\Leftrightarrow z \neq 0 \wedge y/z \ge 0 \wedge sqrt(y/z)^2 \le y/z$.

E. Loop Divergence

- Aside from runtime errors, the other way that programs don't terminate is that they ***diverge*** (run forever). For our programs, that means infinite loops.
  - (For programs with recursion, we also have to worry about infinite recursion, but the discussion here is adaptable, especially if you remember that a loop is simply an optimized tail-recursive function.)
- For some loops, we can ensure termination by calculating the number of iterations left. E.g., at each test there are $n - k$ iterations left for $k := 0$; ***while*** $k < n$ ***do*** ...; $k := k+1$ ***od.***

(Note to self: Verify that $wp(S, q) \to D(q)$. If not, beef up wp in class 11 and here.)

  - But in general, we can't calculate the number of iterations for all loops (see theory of computation course for uncomputable functions).
- But we don't need the exact number of iterations. ***It's sufficient to find a decreasing upper bound*** for the number of iterations.
- ***Definition:*** A ***bound expression*** or ***bound function*** t for a loop is a natural number [2023-03-22] expression that, at each loop test, gives a strictly decreasing upper bound on the number of iterations remaining before termination. A bound expression can use program variables and logical variables.
- ***Syntax***: We'll attach the upper bound expression t to a loop using the syntax $\{\ \textbf{bd}\ t\ \}$, so a typical loop has the form $\{\ \textbf{inv}\ p\ \}\ \{\ \textbf{bd}\ t\ \}\ \textbf{while}\ B\ \textbf{do}\ S\ \textbf{od}\ \{\ p \wedge \neg B\ \}$.
- Note we aren't required to calculate the value of t at runtime, since it's a logical expression.

F. Properties of Bound Functions

- For t to be a valid bound expression, it needs to meet the two following properties:
  - $p \to t \geq 0$
    - Since the invariant has to be true at each loop test, making satisfaction of p imply $t \geq 0$ is a simple way to ensure that $t \geq 0$ at every loop test.
    - Another way to phrase this is that at each loop test, there must be a nonnegative number of iterations left to do.
  - $\{\ p \wedge B \wedge t = t_0\ \}\ S\ \{\ t < t_0\ \}$ where t_0 is a fresh logical variable. [2023-04-06] (For termination, we only need to prove $t < t_0$. When proving partial correctness, we need to prove p.
    - If you compare the value of the bound expression at the beginning and end of the loop body, you find that the value has decreased. I.e., if you were to print out the value of t at each while test, you would find a strictly decreasing sequence of nonnegative integers. (Since t can include logical variables, printing it out might not be possible.)
  - The variable t_0 is a logical variable (we don't actually calculate it at runtime). We're using it in the correctness proof to name of the value of t before running the loop body. It should be a fresh variable (one we're not already using) to avoid clashing with existing variables.

- ***Example 2***: For the $sum(0, n)$ program, we can use $n–k$ for the bound:

  $\{\ n \geq 0\ \}\ k := 0;\ s := 0;$
  $\{\ \textbf{inv}\ p \equiv 0 \leq k \leq n \wedge s = sum(0, k)\ \}$
  $\{\ \textbf{bd}\ n–k\ \}\ \textbf{while}\ k < n\ \ \textbf{do}\ k := k+1;\ s := s+k\ \ \textbf{od}$
  $\{\ s = sum(0, n)\ \}$

  - We need $p \to n–k \geq 0$: At the loop test, p implies $0 \leq k \leq n$, which implies $n–k \geq 0$.
- ***Definition:*** A ***progress step*** is a statement that reduces the value of the bound function. Every loop iteration needs to execute a progress step.

- We need the loop body to contain a progress step: Here, we need to decrease $k–n$: Let t_0 be our fresh logical variable, then we need $\{ p \wedge k < n \wedge n–k = t_0 \}$ *loop body* $\{n–k < t_0\}$. Since the loop body includes $k := k+1$ (and no other change to k), it works as a progress step.
  - In symbols, we find that $\{ n–k = t_0 \}\ \{ n–(k+1) < t_0 \}\ k := k+1\ \{ n–k < t_0 \}$ is a correct full outline.

Other Bound Expression Properties

- The two properties we need a bound expression to have (being nonnegative and decreasing with each iteration) imply that bound expressions have other properties but also that they don't have to have other properties.
- ***The bound expression can't be a constant***, since constants don't change values.
  - ***Example 3***: For the loop $k := 0;$ ***while*** $k < n$ ***do*** $\ldots;\ k := k+1$ ***od***, people often make an initial guess of "n" for the bound expression instead of $n–k$. When $k = 0$, the upper bound is indeed $n–k = n$, but as k increases, the number of iterations left decreases.
- ***A nonnegative bound can't imply that the loop test holds:*** If B is the while loop test, then $t \geq 0 \rightarrow B$ would cause divergence: Since $p \rightarrow t \geq 0$, if $t \geq 0 \rightarrow B$, then $p \rightarrow B$, so B would be true at every loop test.
- $\boldsymbol{p \wedge B \rightarrow t > 0}$ ***is required***: When p and B hold, we run the loop body, which should decrease t but leave it nonnegative. Equivalently, $p \wedge t = 0 \rightarrow \neg B$ because if t is zero, then there's no room for the loop body to decrease t, therefore we'd better not be able to do that iteration. $\neg p \vee \neg B \vee t > 0$ is an equivalent phrasing.
- $\boldsymbol{p \wedge \neg B \rightarrow t = 0}$ ***is not required***:. Since t doesn't have to be a strict upper bound, it doesn't have to be zero on termination. Also not required: $p \wedge t > 0 \rightarrow B$. This is effectively the contrapositive of $p \wedge \neg B \rightarrow t = 0$.
  - Let N be the number of iterations remaining at some loop test point. Then,
    - N ***must be in*** $O(t)$ because $t \geq number\ of\ iterations$.
    - N ***is not required to be in*** $\Theta(t)$ because t is not required to be a strict upper bound.

- $\boldsymbol{\{ p \wedge B \wedge t = t_0 \}}$ *loop body* $\boldsymbol{\{ t – t_0 = 1 \}}$ ***is not required***. We must decrease t by at least one, but more than one is fine.
- ***Example 4***: For searches, t is often the size of the search space. For binary search, if $p \rightarrow left < right$ (where $left$ and $right$ are the endpoints of the search), then $right–left$ is a perfectly fine upper bound even though $ceiling(\log_2(right–left))$ is tighter.

G. Heuristics For Finding A Bound Expression

- ***To find a bound expression*** t, there's no algorithm but there are some guidelines.
  - First, start with a candidate $t \equiv 0$.
  - For each variable or some variable x that the loop body decreases, add x to t.

- For each variable or some variable y that the loop body increases, subtract y from t.
- If $t < 0$ is possible, look for a manipulation that makes the resulting term nonnegative. E.g., if for some e, we have $t \le e$, then $e - t \ge 0$, so we can use $e - t$ as our new candidate t.

- ***Example 5***: Say a loop sets $k := k - 1$. First try k (i.e., add "$+ k$" to $t \equiv 0$) for t. If the invariant allows $k < 0$, then we need something that makes t larger. E.g., if the invariant implies $k \ge -10$, then it implies $k + 10 \ge 0$, so our new candidate bound function is $k + 10$.
- ***Example 6***: For a loop that sets $k := k + 1$, try $(-k)$ (i.e., $0 - k$) for t.
  - If $-k$ can be negative, we should do something to increase it. E.g., if the invariant implies $k \le e$, then it implies $e - k \ge 0$, so adding e to t would help.

H. Increasing and Decreasing Loop Variables

- We've looked at the simple summation loop

```
{ n ≥ 0 } k := 0 ; s := 0 ;
{ inv p ≡ 0 ≤ k ≤ n ∧ s = sum ( 0 , k ) } { bd n – k }
 while k < n do
     k := k + 1 ;
     s := s + k
od
{ s = sum ( 0 , n ) }
```

- ***First bound function:***
  - Using our heuristic, since k and s are increasing, $-k-s$ is a candidate bound function that fails because it's negative.
  - For terms to add to $-k-s$ to make it nonnegative, we know $n - k \ge 0$ because the invariant includes $k \le n$, so let's add n and get $n - k - s$.
  - But $n-k-s$ can be negative, so we want to add some expression e such that $e + n - k - s \ge 0$. The invariant doesn't give an explicit bound for s, but from algebra we know that $0+1+2+\ldots+n$ grows quadratically, and it's easy to verify that $n^2 - s \ge 0$ for all $n \in \mathbb{N}$.
  - This gives $n^2 + n - k - s$ as a bound function.
- ***Second and third bound functions***
  - Just $n - k$ by itself is as a bound function: $n - k$ is nonnegative and decreases each iteration.
  - Similarly, just $n^2 - s$ by itself is a bound function: $n^2 - s$ is nonnegative and decreases with each iteration.
- ***Modifications to bound functions***
  - Bound functions are not unique: If t is a bound expression, then so is $a\,t^n + b$ for any positive a, b, and n. Similarly, if t_1 and t_2 are bound functions separately, then $t_1 + t_2$ is also a bound function. If we had found bound functions $n-k$ and n^2-s individually, we could have combined them to show that $n^2+n-k-s$ is a loop function.

I. Iterative GCD Example

- Not all loops modify only one loop variable with each iteration: Some modify multiple variables, with some being modified sometimes and others being modified another time.
- ***Definition***: For $x, y \in \mathbb{N}, x, y > 0$, the ***greatest common divisor*** of x and y, written $gcd(x, y)$, is the largest value that divides both x and y evenly (i.e., without remainder).
- If you know the prime factorizations of x and y, you can easily find their gcd. E.g., $gcd(300, 180) = gcd(2^2 * 3 * 5^2, 2^2 * 3^2 * 5) = 2^2 * 3 * 5 = 60$. But in general, finding prime factorizations is difficult, and it's been known since ancient times that there are faster simpler ways to calculate a gcd.
- The technique relies on some useful gcd properties:
  - If $x = y$, then $gcd(x, y) = x = y$
  - If $x > y$, then $gcd(x, y) = gcd(x-y, y)$
  - If $y > x$, then $gcd(x, y) = gcd(x, y-x)$
- E.g., $gcd(300, 180) = gcd(120, 180) = gcd(120, 60) = gcd(60, 60) = 60$.
- Here's a minimal proof outline for correctness of an iterative gcd-calculating loop. Since a bound function has yet to be found, the outline is for partial correctness only.

  $\{x > 0 \wedge y > 0 \wedge X = x \wedge Y = y\}$ [2023-03-22] X and Y are the same as x_0 and y_0
  $\{\textbf{inv}\ p \equiv x > 0 \wedge y > 0 \wedge gcd(X, Y) = gcd(x, y)\}$
  $\{\textbf{bd}\ ???\}$
  while $x \neq y$ ***do***
    if $x > y$ ***then*** $x := x - y$ ***else*** $y := y - x$ ***fi***
  od
  $\{x = y = gcd(X, Y)\}$ [2023-03-22]

- Expanding the minimal outline gives a full outline for partial correctness.

  $\{x > 0 \wedge y > 0 \wedge X = x \wedge Y = y\}$
  $\{\textbf{inv}\ p \equiv x > 0 \wedge y > 0 \wedge gcd(X, Y) = gcd(x, y)\}$
  $\{\textbf{bd}\ ???\}$
  while $x \neq y$ ***do***
    $\{p \wedge x \neq y\}$
    if $x > y$ ***then***
      $\{p \wedge x \neq y \wedge x > y\}\ \{p[x-y/x]\}\ x := x - y\ \{p\}$
    else
      $\{p \wedge x \neq y \wedge x \leq y\}\ \{p[y-x/y]\}\ y := y - x\ \{p\}$
    fi $\{p\}$
  od $\{p \wedge x = y\}\ \{x = gcd(X, Y)\}$

- We have a number of predicate logic obligations:

- $x > 0 \wedge y > 0 \wedge x = X \wedge y = Y \rightarrow p$
- $p \wedge x \neq y \wedge x > y \rightarrow p\,[x{-}y/x]$
- $p \wedge x \neq y \wedge x \leq y \rightarrow p\,[y{-}x/y]$
- $p \wedge x = y \rightarrow x = gcd(X, Y)$

- With $p \equiv x > 0 \wedge y > 0 \wedge gcd(X, Y) = gcd(x, y)$, the substitutions are
  - $p\,[x{-}y/x] \equiv x{-}y > 0 \wedge y > 0 \wedge gcd(X, Y) = gcd(x{-}y, y)$
  - $p\,[y{-}x/y] \equiv x > 0 \wedge y{-}x > 0 \wedge gcd(X, Y) = gcd(x, y{-}x)$
- There are other full outline expansions, for example, one using the wp of the entire ***if–fi***, which is
  $(p \wedge x \neq y) \rightarrow ((x > y \rightarrow p\,[x{-}y/x]) \wedge (x \leq y \rightarrow p\,[y{-}x/y]))$
- But the various predicate logic obligations are of basically the same proof difficulty.
- [2023-03-22] We don't have to worry about runtime errors because nothing in the program can cause one.
- What about convergence?
  - The loop body contains code that makes both x and y smaller, so our heuristic gives us $x{+}y$ as a candidate bound function. Non–negativity is easy to show: the invariant implies $x, y > 0$, so $x{+}y \geq 0$.
  - Reduction of $x{+}y$ is slightly subtle: Though the loop body doesn't always reduce x or always reduce y, it always reduces one of them, so $x{+}y$ is always reduced.
- So our minimal outline for total correctness of the program is:

  $\{x > 0 \wedge y > 0 \wedge X = x \wedge Y = y\}$ // X and Y are the initial values of x and y
  $\{$ ***inv*** $p \equiv x > 0 \wedge y > 0 \wedge gcd(X,Y) = gcd(x,y)\}$
  $\{$ ***bd*** $x{+}y\}$
  while $x \neq y$ ***do***
  if $x > y$ ***then*** $x := x - y$ ***else*** $y := y - x$ ***fi***
  od
  $\{x = gcd(X, Y)\}$

- To get a full outline for total correctness, we can take a full outline for partial correctness and add the termination requirements, shown *in blue* below.

  $\{x > 0 \wedge y > 0 \wedge X = x \wedge Y = y\}$ // X and Y are the initial values of x and y
  $\{$ ***inv*** $p \equiv x > 0 \wedge y > 0 \wedge gcd(X, Y) = gcd(x, y) \wedge x{+}y \geq 0\}$
  $\{$ ***bd*** $x{+}y\}$
  while $x \neq y$ ***do***
  $\{p \wedge x \neq y \wedge x{+}y = t_0\}$
  if $x > y$ ***then***
  $\{p \wedge x \neq y \wedge x > y \wedge x + y = t_0\}\,\{p\,[x{-}y/x] \wedge (x{-}y) + y < t_0\}\,x := x - y\,\{p \wedge x + y < t_0\}$
  else
  $\{p \wedge x \neq y \wedge x \leq y \wedge x + y = t_0\}\,\{p\,[y{-}x/y] \wedge x + (y{-}x) < t_0\}\,y := y - x\,\{p \wedge x + y < t_0\}$

fi $\{ p \wedge x + y < t_0 \}$
od $\{ p \wedge x = y \} \{ x = gcd(X, Y) \}$

- For this to work, we need $x + y = t_0$ to imply either $(x - y) + y$ or $x + (y - x) < t_0$ (depending on the ***if–else*** branch). These hold because $(x - y) + y = x < x + y$ and $x + (y - x) = y < x + y$ (because both x and y are positive).

J. Semantics of Convergence

- Here's a semantic assertion about bound functions and loop termination.
- ***Lemma (Loop Convergence)***: Let $W \equiv \{ \textbf{inv } p \} \{ \textbf{bnd } t \}$ ***while*** B ***do*** S ***od*** be a loop annotated with an invariant and bound function. Assume we can prove $\{ p \} W \{ p \wedge \neg B \}$ (partial correctness of the loop) and [2023-03-22] (p→t≥0) and $\{ p \wedge B \wedge t = t_0 \} S \{ p \wedge t < t_0 \}$ (total correctness of the loop body). Then if $\sigma \models p \wedge B \wedge t = t_0$, then $\perp_d \notin M(W, \sigma)$.
  - Proof omitted. (It's by induction on t and pretty straightforward.)

K. *** Total Correctness of a Loop ***

- To show total correctness of $\{ p_0 \} S_0 ; W$ where $W \equiv \{ \textbf{inv } p \} \{ \textbf{bd } t \}$ ***while*** B ***do*** S ***od*** $\{ q \}$, we need
  - Partial correctness: $\{ p_0 \} S_0 ; W \{ q \}$. [2023-03-22] change indentation
    1. Initialization establishes the invariant: $\{ p_0 \} S_0 \{ p \}$. E.g., $\{ p_0 \} \{ wp(S_0, p) \} S_0 \{ p \}$ or $\{ p_0 \} S_0 \{ sp(p_0, S_0) \} \{ p \}$.
    2. The loop body maintains the invariant: $\{ p \wedge B \} S \{ p \}$.
    3. The loop establishes the final postcondition: $p \wedge \neg B \rightarrow q$.
  - Termination: $\models_{tot} \{ p_0 \} S_0 ; W \{ T \}$
    4. No runtime errors during initialization $(p_0 \rightarrow D(S_0))$ nor during loop evaluation: $(p \rightarrow D(B))$ and $(p \wedge B \rightarrow D(S)))$.
    5. No divergence: The bound function is nonnegative $(p \rightarrow t \geq 0)$ and evaluation of the loop body decreases the bound: $\{ p \wedge B \wedge t = t_0 \} S \{ t < t_0 \}$.
- For a formal proof rule, let's concentrate on the loop itself and not worry about initialization or finalization. This leaves partial correctness (line 2 above) and termination (lines 4 and 5 above).

While Loop Rule for Total Correctness

1. $p \rightarrow D(B) \wedge (B \rightarrow D(S))$, where p and B are safe
2. $p \rightarrow \downarrow(t \geq 0)$ [2023-03-22] drop and D(t)
3. $\vdash_{tot} \{ p \wedge B \wedge t = t_0 \} S \{ p \wedge t < t_0 \}$
4. $\vdash_{tot} \{ \textbf{inv } p \} \{ \textbf{bnd } t \}$ ***while*** B ***do*** S ***od*** $\{ p \wedge \neg B \}$ while 1, 2, 3

answer line

Finding Invariants

Part 1: Adding Parameters by Replacing Constants by variables

CS 536: Science of Programming, Spring 2023

2023-03-27 pp. 3, 5, 6; 2023-04-29 pp. 1, 4

A. Why

- It is easier to write good programs and check them for defects than to write bad programs and then debug them.
- The hardest part of programming is finding good loop invariants.
- There are heuristics for finding them but no algorithms that work in all cases.
- Changing how we re-establish an invariant can greatly speed up the code.

B. Objectives

At the end of this class you should

- Know how to generate possible invariants using "replace a constant by a variable" or more generally "add or modify a parameter" and to be familiar with some examples of these techniques.

C. Finding Invariants

- The key (and often, hardest) part of writing correct programs involves finding invariants for our loops.
  - We need to find an invariant and loop test that establishes the desired postcondition: $\{$ **inv** $p\}$ **while** B **do** *???* **od** $\{p \wedge \neg B\}\{q\}$
  - The invariant should be easy to establish with some easy initialization code: $\{p_0\}S_0\{p\}$.
  - The loop body maintains the invariant: $\{p \wedge B\}$ *loop body* $\{p\}$.
  - When the loop terminates, the postcondition we want holds: $p \wedge \neg B \rightarrow q$. (Sometimes you have finalization code that you need, then you need $\{p \wedge \neg B\}$ *code* $\{q\}$.
- Note: $p \wedge \neg B$ is stronger than q but p itself is weaker than q ([2023-04-29] i.e., $q \rightarrow p$). This suggests the main way people find invariants: Take the postcondition q and weaken it somehow. In particular, weaken it in a way that combining with $\neg B$ gives us q.
- However, it's not required that $p \wedge B \rightarrow \neg q$. If $\neg B$ is too strong, we could be missing out on some states in q, which would make our loop too restrictive in the states it terminates in. So finding the right B is a balancing act.

General Heuristics for Finding Invariants

- There exist various general heuristics for finding invariants. Not every way applies to every situation, but all have the pattern of weakening the postcondition, with the loop test shaped by how and how much we weaken the postcondition.

Add More States to q

- Adding states to q will weaken it. We want B to include the states that are added, and $\neg B$ avoids those states so that $p \wedge \neg B$ only includes states from q. We can try to:
  - ***Add parameters***, as in Replace a Constant by a Variable, or more generally, Replace an Expression With Another Expression..
  - ***Generalize Relations*** (e.g., change an = to a ≤ or to an equivalence relation).
  - ***Add a Disjunction*** (for some r, use $q \vee r$ an invariant).

Remove Fewer States from q

- Say q is constructed as a conjunction of various properties $q_1 \wedge q_2 \wedge q_3$ etc. We can see this as starting with q_1, then adding q_2 to remove some states, then adding q_3 to remove more states, and so on. If we drop a conjunct from this removal process, then the result is weaker than q.

D. Replace A Constant By A variable

- The technique "Replace a constant by a variable" produces a candidate invariant by adding a new parameter to a predicate. We take the postcondition and replace a literal or symbolic constant c with a fresh variable x.
  - We can't phrase this as saying $p \equiv q\,[\,x/c\,]$ because substitution replaces a variable with an expression and also, in the simplest case, we only replace one occurrence of c with x, not all occurrences. So the correct phrasing is $q \equiv p\,[\,c/x\,]$, with p being the candidate invariant. Our candidate loop will be $\{$ ***inv*** $p\,\}$ ***while*** $x \neq c$ ***do*** ... ***od*** $\{\,p \wedge x = c\}\,\{\,q\,\}$.
- Depending on how many constants appear where in q, there can be multiple candidate invariants, which will typically have different loop tests, initialization code, and/or progress steps.
- In addition, not all the candidate invariants may be unusable, generally because there's no good initialization code or progress step.

Loop Initialization

- When we add a variable, we should look for the range of values the new variable can have. We always have $x = c$ where the constant is the one we're replacing. Initialization should provide a second value d the variable can have.
- If c and d form a natural boundary for a range of values, say $c \le d$, then we can initialize $x := d$ and the range for x would be $c \le x \le d$. Progressing toward termination would take x from d down to c. Symmetrically, if $d \le c$, then progression would take x from d up to c.

- Note we might find a d to initialize x to yet still have no obvious range of values for x.
- Another possibility is that there's no clear value we can initialize x to. When that happens, the candidate invariant fails.

- ***Example 1***: Summation loops
  - The postcondition $s = sum(0, n)$ has two constants 0 and n.
  - We can try replacing n by a variable k. Initialize $k = 0$ and increase it until $k = n$, the range of k will be $0, \ldots, n$.

    $\{$ **inv** $s = sum(0, k) \wedge 0 \le k \le n \}$ $\{$ **bd** $n - k \}$
    while $k \neq n$ **do**
    ... make k larger ...
    od
    $\{ s = sum(0, k) \wedge 0 \le k \le n \wedge k = n \}$
    $\{ s = sum(0, n) \}$
  - Or, we can try replacing 0 by a variable k. Initialize $k = n$ and decrease it until $k = 0$. Again, the range of k will be $0, \ldots, n$.

    $\{$ **inv** $s = sum(k, n) \wedge 0 \le k \le n \}$ $\{$ **bd** $k \}$
    while $k > 0$ **do**
    ... make k smaller ...
    od
    $\{ s = sum(k, n) \wedge 0 \le k \le n \wedge k = 0 \}$
    $\{ s = sum(0, n) \}$

- ***Example 2***: Initialization of summation loops
  - For an invariant $s = sum(0, k) \wedge 0 \le k \le n$, setting $k := 0$ or $k := n$ seems natural.
    - If we set $k := 0$, it's easy to establish $wp(i := 0, p) \equiv s = sum(0, 0) \wedge 0 \le 0 \le n$ via $s := 0$ (and the assumption $n \ge 0$).
    - But setting $k := n$ leads us to $wp(i := n, p) \equiv s = sum(0, n) \wedge 0 \le n \le n$, which is hard to satisfy (in fact, it's our original postcondition).

- ***Example 3***: Integer square root (replacing a constant by a variable)
  - To take the integer square root of an $n \ge 0$ means to find an x such that $x \le sqrt(n) < x+1$, or equivalently, $x^2 \le n < (x+1)^2$. Let that be q.
  - We can weaken *q [2023-03-27]* by replacing the 1 in $x+1$ with a new variable, say y, and get $x^2 \le n < (x+y)^2$ as a candidate invariant. For a range, we want $y \ge 1$. Loop initialization will

set x to something small like 2 or 1 and set y to something large (like n, if $n > 0$, or $n+1$ if $n \geq 0$) [*].

- Whatever we initialize y to, progress toward termination consists of making x larger or $x+y$ smaller. This suggests a bound function of $x+y-x$, which simplifies to y.

  $\{$ **inv** $x^2 \leq n < (x+y)^2 \wedge 1 \leq y \}$ $\{$ **bd** $n+y-x \}$
  while $y \neq 1$ **do** [2023-04-29]
  ... make x larger or $x+y$ smaller ...
  od
  $\{ x^2 \leq n < (x+y)^2 \wedge 1 \leq y \wedge y = 1 \}$
  $\{ x^2 \leq n < (x+1)^2 \}$

More General Replacements

- To replace a constant by a variable is the easiest way to add a parameter to a predicate. A slight generalization is to Replace an entire expression by a variable.

- ***Example 4***: Integer square root (replacing an expression by a variable)
  - For the integer square root problem, we can weaken $0 \leq x^2 \leq n < (x+1)^2$ by replacing $x+1$ with y to get $x^2 \leq n < y^2$ (and looping until $y = x+1$). For a range, we know $y \geq x+1$, and initialization is similar to that used in Example 3.

    $\{$ **inv** $0 \leq x^2 \leq n < y^2 \}$ $\{$ **bd** $y - x \}$
    while $y > x+1$ **do**
    ... make x larger or make y smaller ...
    od
    $\{ 0 \leq x^2 \leq n < y^2 \wedge y = x+1 \}$
    $\{ 0 \leq x^2 \leq n < (x+1)^2 \}$

The Loop Body

- Recall that a progress statement is a statement that gets us closer to termination. We need to execute at least one every iteration, along every execution path for the loop body.
- One way to organize a loop is $\{$ **inv** $p \}\{$ **bd** $t \}$ **while** B **do** S ; R **od**, where R is a progress step and S establishes the wp of R and the invariant:

  $\{$ **inv** $p \}$ $\{$ **bd** $t \}$ **while** B **do** $\{ p \wedge B \wedge t = t_0 \} S ; \{ wp(R, p \wedge t < t_0) \} R \{ p \wedge t < t_0 \}$ **od**
- For replacing a constant by a variable in particular, R takes us closer to the target constant by modifying the new variable.
- Say we want S such that $\{ v = e_1 \} S \{ v = e_2 \}$. Two simple ways are:

---

[*] The more we know about n, the better we might be at figuring out a range. For example, if $n > 4$, then $(n/2)^2 > n$, so we could initialize $y := n/2 - 1$.

- $\{ v = e_1 \}\ v := v + e_2 - e_1 \{ v = e_2 \}$
- $\{ v = e_1 \}\ v := v * e_2 \div e_1 \{ v = e_2 \}$ // (assuming e_1 divides e_2)

- One example was in the summation loop: We needed $s = sum(0, k+1)$ but had $s = sum(0, k)$. Following the first pattern above, we could guarantee progress by using

  $\{ s = sum(0, k) \}\ s := s + sum(0, k+1) - sum(0, k) \{ s = sum(0, k+1) \}$

- Of course, this isn't practical, but since $sum(0, k+1) - sum(0, k) = k+1$, we can (and did) instead use

  $\{ s = sum(0, k) \}\ s := s + (k+1) \{ s = sum(0, k+1) \}$

- ***Example 5***: (Integer log base 2) Find the largest power of 2 that is ≤ 2.
  - Say our invariant is $y = 2 \uparrow k \le x \wedge 0 \le k$ (we loop ***while*** $2 * y \le x$) and our progress step is $k := k+1$, so the *wp* of the progress step is $y = 2 \uparrow (k+1) \le x \wedge 0 \le k+1$.
  - So we need to establish $\{ y = 2 \uparrow k \wedge \ldots \};\ y := ??? \{ y = 2 \uparrow (k+1) \wedge \ldots \}\ k := k+1 \{ y = 2 \uparrow k \wedge \ldots \}$.
  - Both patterns above for changing y can be used here:
    - One possibility is $y := y + 2 \uparrow (k+1) - 2 \uparrow k$. Since $2 \uparrow (k+1) - 2 \uparrow k = 2 \uparrow k = y$, this simplifies to $y := y+y$.
    - Another possibility is $y := y * 2 \uparrow (k+1) \div 2 \uparrow k$, which simplifies to $y := y * 2$.

Replacing a Constant by a Variable Can Fail

- Not every constant when replaced yields an invariant that works well.
- E.g. take the postcondition $x^2 \le n < (x+1)^2$ and replace one (or say both) of the 2's with a new variable y. We loop ***while*** $y \neq 2$ [2023-03-27] with candidate invariants $x \uparrow y \le n < (x+1)^2$ or $x^2 \le n < (x+1) \uparrow y$.
- Initialization:
  - If we're trying $x \uparrow y \le n$ we could try $y := 0$, and so we'd need $1 = x^0 \le n$.
  - If we're trying $n < (x+1) \uparrow y$, the situation is much less obvious Maybe $x := n;\ y := 1$? But we'd need $n^2 \le n < (n+1)^1$, and $n^2 \le n$ requires $n = 0$ or 1. Clearly this is a dead end.
- Progress step:
  - Going back to trying $x \uparrow y \le n$ as the invariant, if we initialize y to 0, then we need to make it larger. The simplest way is $y := y+1$, and the rest of the loop body establishes the *wp* of this assignment and the invariant:

    $\{ x \uparrow y \le n < (x+1)^2 \wedge y \neq 2 \}$ [2023-03-27] // *Invariant* $\wedge$ *loop test*
    ... // *Needed code*
    $\{ x \uparrow (y+1) \le n < (x+1)^2 \}$ // *wp(progress step, invariant)*
    $y := y+1$ // *The progress step*
    $\{ x \uparrow y \le n < (x+1)^2 \}$ // *The invariant*

  - What could the missing code possibly be? Time to give up and look for a different invariant.

Adding or Modifying Parameters more Generally

- Replacing a constant by a variable is a simple version of the more general notion of adding or modifying the parameters of a predicate. Other versions of this technique include other changes:
  - Replace: One occurrence | Multiple occurrences
  - Of: A Constant | A Constant-valued Expression | A Variable | An Expression
  - With: A New variable | An Expression with one or more New variables
- For example [2023-03-27]
  - Variable with another variable: $q \equiv x < f(x)$ becomes $x < f(y)$ or $y < f(x)$ loop ***while*** $y \neq x$.
  - Expression with variable: $q \equiv x < f(x)$ becomes $x < y$ ***while*** $y \neq f(x)$.
  - Expression with one or more new variables.: $q \equiv x < f(x)$ becomes $g(y, z) < f(x)$ ***while*** $x \neq g(y, z)$.
- Whether any of these above candidates work will depend on what $f(\dots)$ and $g(\dots, \dots)$ do.

Finding Invariants

Part 2: Deleting Conjuncts; Adding Disjuncts; Examples

CS 536: Science of Programming, Spring 2023

2023-03-29: pp. 5, 6, 9, 10

A. Why

- It is easier to write good programs and check them for defects than to write bad programs and then debug them.
- The hardest part of programming is finding good loop invariants.
- There are heuristics for finding them but no algorithms that work in all cases.
- Changing how we re-establish a loop invariant can greatly speed up the code.

B. Objectives

At the end of this class you should

- Know how to generate possible invariants using “Drop a conjunct” or “Add a disjunct” and to be familiar with some examples of these techniques.

C. Finding Invariants - Review

- An invariant needs to be easy to establish with initialization code, it needs to establish the post-condition (when the loop test fails), and it has to be an approximation to the *sp* and *wp* of the loop body.
- There exist various general heuristics for finding invariants, though no heuristic works easily in every situation. The general idea is to weaken the postcondition somehow*;* the kind of weakening determines the loop test.
- We've looked at getting candidate invariants by adding parameters to the postcondition.
- ***Replacing a Constant by a Variable*** is the simplest way to add a parameter.
  - We take the postcondition q, find an occurrence of a constant c in it, and replace that occurrence with a new variable x. The result is a candidate invariant p where $p\,[\,c\,/\,x\,] \equiv q$.
  - The loop header becomes ***while*** $x \neq c$, and initialization code has to set the loop variable x to a value that satisfies p.
  - For loop termination, we make progress by changing x so that it's ”closer” to c.
  - For example, we saw changes to $q \equiv s = sum\,(\,0\,,\,n\,)$ where we replaced 0 by i or n by j.
- More generally, we can add parameters by replacing one or more occurrences of an expression with one or more new variables. The expression which might be a constant or a constant-valued expression, or a variable, or any other kind of expression.

- An example was integer square root, where $x^2 \le n < (x+1)^2$ became $x^2 \le n < y^2$ by replacing $(x+1)^2$ with y, as opposed to $x^2 \le n < (x+y)^2$ when we replaced 1 with y.

D. Adding a Disjunct

- Adding a disjunct is another way to find possible invariants. Say we want to establish postcondition q. For various possible B, we can try

  $\{$ **inv** $q \vee B\}$
  while B **do**
  $\{(q \vee B) \wedge B\}$
  Loop body
  $\{q \vee B\}$
  od
  $\{(q \vee B) \wedge \neg B\}$
  $\{q\}$
- Unlike first two methods, this one is very open-ended. The other techniques we've seen all take the postcondition and modify some identified part of it, but here we can use any testable predicate for B.
- Adding a disjunct lets us, e.g., generalize a relation like $i = n$ to $i \le n$ (i.e., $i = n \vee i < n$). This is one way to understand a loop like $\{$ **inv** $i \le n$ ... $\}$ **while** $i < n$ **do**... **od** $\{ i = n \}$: The invariant $i \le n$ is the postcondition $i = n$ plus the disjunct $i < n$ added.

E. Deleting A Conjunct

- A different way to find possible invariants is ***Deleting a Conjunct***. Say postcondition is q is a conjunction, $q_1 \wedge q_2 \ldots \wedge q_n$ where $n \ge 2$.
- We get n candidate invariants, each one being q less one conjunct, and the loop runs until the conjunct is true. For $k = 1, 2, \ldots, n$, let p_k be q less q_k; i.e., $p_k \equiv q_1 \wedge \ldots q_{k-1} \wedge q_{k+1} \wedge \ldots q_n$.
- The candidate loop using p_k is

  $\{$ **inv** $p_k\}$ //where $p_k \equiv q_1 \wedge \ldots q_{k-1} \wedge q_{k+1} \wedge \ldots q_n$.
  while $\neg q_k$ **do**
  $\{p_k \wedge \neg q_k\}$ ... $\{p_k\}$
  od
  $\{p_k \wedge q_k\}$
  $\{q\}$
- Adding a disjunct is one way to view deleting a conjunct.
  - Take $q_1 \wedge q_2$ and add the disjunct $(q_1 \wedge \neg q_2)$. The result is $(q_1 \wedge q_2) \vee (q_1 \wedge \neg q_2) \Leftrightarrow q_1$, in effect deleting q_2.

- Another example: Converting $p \wedge q$ to $p \vee q$ can be viewed as a generalization of $\wedge$ to $\vee$ or as taking $p \wedge q$ to $(p \wedge q) \vee (p \wedge \neg q) \vee q$.

F. Examples of Programs and Their Invariants

- We won't cover all of these in class and you're not expected to know these in detail. It would be good to look at each example and ask yourself what technique is being used to find the invariant, the invariant relates to the bound function, and how progress toward termination is made.

Example 1: Linear Search of an Array

- This will be an example of deleting a conjunct.
- Precondition: Array b has at least n elements $(n \geq 0)$ and the value x may or may not appear in $b[0..n-1]$.
- Postcondition: We find the index k of the leftmost occurrence of x in $b[0..n-1]$. If x doesn't appear in $b[0..n-1]$, then $k = n$. Note in either case, x doesn't appear in $b[0..k-1]$. We can formalize this as

  $0 \leq k \leq n \wedge x \notin b[0..k-1] \wedge (k < n \rightarrow b[k] = x)$

  where $x \notin b[0..k-1]$ means $\forall\, 0 \leq k' < k \,.\, x \neq b[k']$. Note if $k = 0$, then $b[0..k-1] = b[0..-1]$, which is the empty sequence of values.
- Since $0 \leq k \leq n$ is short for $0 \leq k \wedge k \leq n$, there are four conjuncts we can try deleting, which yields four possible loop/test combinations. Three of them don't yield a usable invariant, but the fourth one does.
- Dropping the first conjunct, $0 \leq k$, forces $k < 0$ in the loop body, which makes referencing $b[k]$ illegal. This sounds really unpromising.

  $\{$ ***inv*** $k \leq n \wedge x \notin b[0..k-1] \wedge (k < n \rightarrow b[k] = x)\}$ ***while*** $0 > k$ ***do*** ...
- Dropping $k \leq n$ has the symmetric problem: $k > n$ in the loop body makes $b[k]$ erroneous.

  $\{$ ***inv*** $0 \leq k \wedge x \notin b[0..k-1] \wedge (k < n \rightarrow b[k] = x)\}$ ***while*** $k > n$ ***do*** ...

[Start rewrite in v.2] [1]

- Dropping $x \notin b[0..k-1]$ means we'd use ***while*** $x \in b[0..k-1]$. First problem: How do we initialize k? Using $k := 0$ makes $x \notin$ the empty segment $b[0..k-1]$. Using $k := 1$ requires $b[0] = x$, which can be false, and using $k := n$ requires us to know $x \in b[0..n-1]$, which we don't.
- Dropping the fourth conjunct, $k < n \rightarrow b[k] = x$, however, works well.

  $\{$ ***inv*** $0 \leq k \leq n \wedge x \notin b[0..k-1]\}$ ***while*** $\neg(k < n \rightarrow b[k] = x)$ ***do*** ...
- Now we can rewrite $\neg(k < n \rightarrow b[k] = x)$ as $(k < n$ && $b[k] \neq x)$, where && is the short-circuiting operator found in C etc.: B_1 && $B_2 \equiv$ ***if*** B_1 ***then*** B_2 ***else*** F ***fi***.

---

[1] In class I said we could initialize $k=0$ but that's backward - we need $x \in b[0..k-1]$, not $\notin$.

- Initialization is easy: $k := 0$, since its *wp* is $0 \le 0 \le n \wedge x \notin b[0..0-1]$. The only nontrivial part is $n \ge 0$, which will be the initial precondition.

[End rewrite]

- Since k starts out at 0 and must increase to n, a progress step of $k := k+1$ seems pretty reasonable. The loop body so far is

  $\{ p \wedge k < n \wedge b[k] \neq x \}$ // Invariant∧ loop test
  ??? // Code to write
  $\{ 0 \le k+1 \le n \wedge x \notin b[0..k+1-1] \}$ // *wp* of progress step
  $k := k+1$ // Progress step
  $\{ 0 \le k \le n \wedge x \notin b[0..k-1] \}$ // Invariant

  where *???* is code that must take us from the precondition of the loop body to the *wp* of the loop body. But it turns out that the precondition implies the *wp*, so no code is needed.

- Convergence is easy: Since p includes $k \le n$ and k gets incremented, we can use $n-k$. So the whole loop is

  $\{ n \ge 0 \}$
  $k := 0$;
  $\{$ **inv** $p \equiv 0 \le k \le n \wedge x \notin b[0..k-1] \wedge n-k \ge 0 \}$ $\{$ **bd** $n-k \}$
  while $k < n$ && $b[k] \neq x$ **do**
  $\{ (0 \le k \le n \wedge x \notin b[0..k-1] \wedge n-k \ge 0) \wedge k < n \wedge b[k] \neq x \wedge n-k = t_0 \}$
  $\{ 0 \le k+1 \le n \wedge x \notin b[0..k+1-1] \wedge n-(k+1) < t_0 \}$
  $k := k+1$
  $\{ (0 \le k \le n \wedge x \notin b[0..k-1]) \wedge n-k < t_0 \}$
  od
  $\{ 0 \le k \le n \wedge x \notin b[0..k-1] \wedge (k < n \rightarrow b[k] = x) \}$

Example 2: Binary Search Example (Version 1)

- Binary search is a nice example of a loop that isn't a ***for*** loop. For termination, a loose upper bound (the distance between the endpoints) suffices.
- Binary search has a small subtlety about what to do when the left and right endpoints are adjacent: Because of integer division, $midpoint = (L+(L+1)) \div 2 = L$, which implies that the distance from L to the *midpoint* doesn't always decrease.
  - We'll see a couple of ways to handle this.
    - The first way uses a sentinel value, and when $R = L+1$, the loop halts.
    - The second way allows $R = L-1$, and $R < L$ causes halting.
  - The differences between the two approaches makes the postconditions different, which in turn makes the invariants different, and (as it turns out) the loop test is different.

Binary Search version 1

- Program specification: $\{ q_0 \}\ Binsearch(b, x, n)\ \{ r \}$ where
  - $q_0 \equiv Sorted(b, n) \wedge 1 \le n < |b| \wedge b[0] \le x < b[n]$ (writing $|b|$ for $size(b)$)
  - $Sorted(b, n) \equiv \forall\, 0 \le k < n-1 < |b| - 1 \,.\, b[k] \le b[k+1]$
  - $r \equiv 0 \le L < n \wedge (b[L] = x \vee b[L] < x < b[L+1]) \wedge (found \leftrightarrow x = b[L])$
- Having $x < b[n]$ means $b[n]$ is a sentinel value, not an actual data value.
- Since b and n are named constants, $Sorted(b, n)$ holds throughout the program, so we'll omit explicitly writing it in the conditions.
- For our invariant, let's generalize the loop [2023-03-29] postcondition $b[L] \le x < b[L+1]$ to $b[L] \le x < b[R]$ where $0 \le L < R \le n$. (We replaced the expression $L+1$ by R.) In addition, let's weaken the postcondition's $(found \leftrightarrow x = b[L])$ to just implication: $(found \rightarrow x = b[L])$; this lets us have $found = F$ while we search. For the bound function, we can use $R-L$; it's a loose termination bound but that's okay.
- For the loop body, we'll begin by calculating the midpoint $m := (L+R) \div 2$ (with truncating division). The search succeeds if $b[m] = x$; we can set $found$ to true and L to m and exit the loop.
- The loop so far is

$\{ q \equiv Sorted(b, n) \wedge n \ge 1 \wedge b[0] \le x < b[n] \}$
$L := 0 ; R := n ; found := F ;$
$\{ \textbf{inv}\ p \equiv 0 \le L < R \le n \wedge (b[L] = x \vee b[L] < x < b[R]) \wedge (found \rightarrow x = b[L]) \}$
$\{ \textbf{bd}\ R-L \}$ ***while*** $\neg found \wedge R \ne L+1$ ***do***
    $\{ p \wedge \neg found \wedge R \ne L+1 \wedge R-L = t_0 \}$
    $m := (L+R) \div 2 ;$
    $\{ p_1 \equiv p \wedge \neg found \wedge R \ne L+1 \wedge R-L = t_0 \wedge m = (L+R) \div 2 \}$
    if $b[m] = x$ ***then***
        $\{ p_1 \wedge b[m] = x \}\ found := T ; L := m ; R := L+1\ \{ p \wedge R-L < t_0 \}$ [2] [2023-03-29]
    else
        $\{ p_1 \wedge b[m] \ne x \}$ *... code to be filled in ...* $\{ p \wedge R-L < t_0 \}$
    fi
    $\{ p \wedge R-L < t_0 \}$
od
$\{ p \wedge (found \vee R = L+1) \}$
$\{ 0 \le L < n \wedge (b[L] = x \vee b[L] < x < b[L+1]) \wedge (found \leftrightarrow x = b[L]) \}$

- It's easy to verify that loop initialization is correct. Loop termination is also correct: Either $found$ is true and $b[L] = x$, or $found$ is false, $R = L+1$, and $b[L] < x < b[L+1]$, indicating the search has indeed failed.

[2] In class I said I thought this was a bug because we don't know $x \ne b[L+1]$, but now I remember that that's why things are written as $b[L] = x$ **OR** $b[L] < x < b[R]$.

- The loop body calculates the midpoint m and checks $b[m]$ against x. If $b[m] = x$, the search has succeeded and we set L, R, and $found$ accordingly.
- If $b[m] \neq x$, then there are two ways to make progress toward termination: $L := m$ and $R := m$. Both assignments have $p \wedge R–L < t_0$ as the postcondition, so we can calculate the wp of each assignment and see if the current precondition $p_1 \wedge b[m] \neq x$ is sufficient. We get

  $\{ p_1 \wedge b[m] \neq x \}$ ... $\{ p[m/L] \wedge R–m < t_0 \}\ L := m\ \{ p \wedge R–L < t_0 \}$

  $\{ p_1 \wedge b[m] \neq x \}$ ... $\{ p[m/R] \wedge m–L < t_0 \}\ R := m\ \{ p \wedge R–L < t_0 \}$
- Expanding,
  - $p_1 \wedge b[m] \neq x \equiv p \wedge \neg found \wedge R \neq L+1 \wedge R–L = t_0 \wedge m = (L+R) \div 2 \wedge b[m] \neq x$
  - Since $p \equiv 0 \leq L < R \leq n \wedge (b[L] = x \vee b[L] < x < b[R]) \wedge (found \rightarrow x = b[L])$
  - Substituting, $p[m / L] \equiv 0 \leq m < R \leq n \wedge (b[m] = x \vee b[m] < x < b[R]) \wedge (found \rightarrow x = b[m])$
  - And $p[m / R] \equiv 0 \leq L < m \leq n \wedge (b[L] = x \vee b[L] < x < b[m]) \wedge (found \rightarrow x = b[L])$
- Comparing (and omitting detailed calculations), we see that to imply the wp of $L := m$, the precondition $p_1 \wedge b[m] \neq x$ is not strong enough. We need to add $(m < R \wedge b[m] \leq x \wedge R–m < t_0)$.
- Similarly, to imply the wp of $R := m$ we need to add $(L < m \wedge b[m] > x \wedge m–L < t_0)$.
- We can determine $b[m] < x$ and $b[m] > x$ with a test (we already know $b[m] \neq x$).
- It turns out that *L < R [2023-03-29]* and $R \neq L+1$ imply all four of $m < R$, $R–m < t_0$, $L < m$, and $m–L < t_0$, so the wp's are satisfied[3].
- Adding the test for $b[m] <$ or $> x$ gives us a loop body partially outlined[4] as

```
{ q ≡ Sorted(b, n) ∧ n ≥ 1 ∧ b[0] ≤ x < b[n] }
L := 0 ; R := n ; found := F ;
{ inv p ≡ 0 ≤ L < R ≤ n ∧ ((b[L] = x ∨ b[L] < x < b[R]) ∧ (found → x = b[L])) }
{ bd R–L }  while ¬found ∧ R ≠ L+1  do
    { p ∧ ¬found ∧ R ≠ L+1 ∧ R–L = t0 }
    m := (L+R) ÷ 2 ;
    { p1 ≡ p ∧ ¬found ∧ R ≠ L+1 ∧ R–L = t0 ∧ m = (L+R) ÷ 2 }
    if b[m] = x  then
        found := T ; L := m
    else if b[m] < x  then
        L := m
    else // b[m] > x
        R := m
    fi fi
    { p ∧ R–L < t0 }
od
```

[3] Quick argument for $L < m < R$: Since $L+2 \leq R$, $m = (L+R) \div 2 \geq (2*L+2) \div 2 = L+1$ and also $\leq (2*R–2) \div 2 \leq R–1$.

[4] A nice at-home activity is to completely expand the annotation.

$\{ p \wedge (found \vee R = L+1) \}$
$\{ 0 \le L < n \wedge ((b[L] = x \vee b[L] < x < b[L+1])) \wedge (found \leftrightarrow x = b[L]) \}$

Example 3: Traditional Binary Search

- For contrast, let's look at a traditional version of binary search, where we stop if $L > R$.
- We won't have a sentinel value in b, so $b[n–1]$ is the last data value, and the precondition becomes $Sorted(b, n) \wedge n \ge 1 \wedge b[0] \le x \le b[n–1]$.
- The postcondition will be different: If we end with $R < L$ (in particular $R = L–1$) then the search has failed, otherwise $b[L] = x$ as before. Again, to distinguish between failure and success, we'll use $found$ to stop the search. At termination,

  $–1 \le L–1 \le R < n \wedge (found \rightarrow b[L] = x) \wedge (\neg found \rightarrow x \notin b[0..n–1])$

  - $–1 \le L–1 \le R < n$ summarizes the properties and relationships of L and R, namely $0 \le L < n$ and either $L \le R < n$ or $R = L–1$.
- For the invariant, we want to weaken $(\neg found \rightarrow x \notin b[0..n–1])$ to something that will be true during the search. We only change L and R in ways that don't alter "Is x in $b[L..R]$?" If $R < L$, we know the search has failed because $b[L..R] = \varnothing$. We should terminate the loop if $found$ or $(R < L \wedge \neg found)$.
- Now for a bound function. We can't use $R–L$ because it can be $–1$. We can almost use $R–L+1$, except that when find $b[m] = x$, all we do is set $found := T$ and $L := m$, which doesn't necessarily decrease $R–L+1$. To take $found$ into account, define $|F| = 0$ and $|T| = 1$, then the bound function can be $R–L+1 + |\neg found|$.
- Altogether, we get the following sketch for our binary search:

```
{ n > 0 ∧ Sorted(b, n) ∧ b[0] ≤ x ≤ b[n–1] } L := 0; R := n–1; found := F;
{ inv q ≡ –1 ≤ L–1 ≤ R < n ∧ (found → b[L] = x) ∧ (x ∈ b[0..n–1] ↔ x ∈ b[L..R]) }
{ bd R–L+1 + |¬found| }
while ¬found ∧ L ≤ R do
    m := (L+R) ÷ 2;
    { q ∧ ¬found ∧ L ≤ R ∧ t = t0 ∧ m = (L+R) ÷ 2 }  // where t ≡ R–L+1 + |¬found|
    if b[m] = x then
        found := T; L := m
    else if b[m] < x then
        L := m + 1
    else // b[m] > x
        R := m – 1
    fi fi { q ∧ t < t0 }
od
{ q ∧ (found ∨ L > R) }
{ –1 ≤ L–1 ≤ R < n ∧ (found → b[L] = x) ∧ (¬found → x ∉ b[0..n–1]) }
```

Example 4: Match Across Two Arrays

- We saw the three-array version of this problem when we looked at sequential nondeterminism. The context there was finding partial solutions and combining them nondeterministically. This time, we'll concentrate on the invariant and on termination. The arrays can have different lengths, and this is reflected in the bound function. To cut down on notation, we'll just match two arrays instead of three arrays.
- We start with two sorted arrays b_1 and b_2 and want to find the least indexes i and j that make $b_1[i] = b_2[j]$; if no such values exist, we halt with $i = n \vee j = m$ where $n = |b_1|$ and $m = |b_2|$.
  - We'll use a bound function of $t(i,j) \equiv (n-i) + (m-j)$. Defining it as an actual function lets us write, e.g., $t(i+1, j)$ later on, instead of $((n-i)+(m-j))[i+1/i]$.
  - We can initialize i and j to 0, increment at least one of them with each iteration and ensure that the invariant implies $0 \le i \le n \wedge 0 \le j \le m$.
- We aren't going to change b_1 or b_2, so we'll specify $Sorted(b_1, n) \wedge Sorted(b_2, m)$ in the initial precondition, but after that, omit it as being implicit.

  $Sorted(b, n) \equiv \forall\, 0 \le k \le n-2 \,.\, b[k] \le b[k+1]$
- It wasn't mentioned earlier, but that program stopped with the first match it found, and so will this one. We can formalize the "least indexes i and j" part of the postcondition as a property that says no value to the left of $b_1[i]$ matches any value to the left of $b_2[j]$:

  $noMatch(i,j) \equiv \forall\, 0 \le i' < i \le n \,.\, \forall\, 0 \le j' < j \le m \,.\, b_1[i'] \ne b_2[j']$
- We also define $InRange(i,j) \equiv 0 \le i \le n \wedge 0 \le j \le m$, so our postcondition is

  $q \equiv InRange(i,j) \wedge noMatch(i,j) \wedge (i < n \wedge j < m \rightarrow b_1[i] = b_2[j])$
- To get an invariant, we'll drop the third conjunct $(i < n \wedge j < m \rightarrow b_1[i] = b_2[j])$.

  $\{$ **inv** $p \equiv InRange(i,j) \wedge noMatch(i,j)\}\{$ **bd** $t(i,j)\}$
  while $\neg(i < n \wedge j < m \rightarrow b_1[i] = b_2[j])$
  do ...
  od
  $\{p \wedge (i < n \wedge j < m \rightarrow b_1[i] = b_2[j])\}\{q\}$
- As in linear search (Example 1), we'll rewrite the test as $B \equiv (i < n \wedge j < m \,\&\&\, b_1[i] \ne b_2[j])$. As a conditional expression, this is **if** $i < n \wedge j < m$ **then** $b_1[i] \ne b_2[j]$ **else** F **fi**.
- Let's consider loop initialization. As we begin, $noMatch(0,0)$ is all we know about the arrays, so we should set i and j to zero.

  $\{n \ge 0 \wedge m \ge 0 \wedge Sorted(b, n) \wedge Sorted(b_2, m)\}$
  $i := 0 ; j := 0$
  $\{inRange(0,0) \wedge noMatch(0,0)\}$
  $\{$ **inv** $p \equiv InRange(i,j) \wedge noMatch(i,j)\}\{$ **bd** $t(i,j) \equiv (n-i)+(m-j)\}$
  while $i < n \wedge j < m \,\&\&\, b_1[i] \ne b_2[j]$
  do ...

od
$\{ q \equiv p \wedge b \}$ // where $b \equiv i < n \wedge j < m \rightarrow b_1[i] = b_2[j]$

- The termination requirement that the invariant imply $t(i,j) \geq 0$ follows from $InRange(i,j)$.
- To get closer to termination, we'll use either $i := i + 1$ or $j := j + 1$. So our loop body will include finding code taking us from the invariant and loop test to the *wp* of each progress statement. With invariant $p \equiv InRange(i,j) \wedge noMatch(i,j)$, and $\neg B \Leftrightarrow i < n \wedge j < m \,\&\&\, b_1[i] \neq b_2[j]$, calculating the *wp's* of the progress steps gives us

$\{ p \wedge t(i,j) = t_0 \wedge \neg B \}$	// inv, bound, and test
???	// needed code
$\{ InRange(i+1,j) \wedge\ noMatch(i+1,j) \wedge t(i+1,j) < t_0 \}$	// *wp* of progress step
$i := i+1$	// increase i to make progress
$\{ p \wedge t < t_0 \}$	// inv and decreased bound

and

$\{ p \wedge t(i,j) = t_0 \wedge \neg B \}$	// inv, bound, and test
???	// needed code
$\{ InRange(i,j+1) \wedge\ noMatch(i,j+1) \wedge t(i,j+1) < t_0 \}$	// *wp* of progress step
$j := j+1$	// increase j to make progress
$\{ p \wedge t < t_0 \}$	// inv and decreased bound

- The *wp* range requirements are easy: $InRange(i+1,j)$ and $InRange(i,j+1)$ are both implied by $InRange(i,j) \wedge \ldots\ i < n \wedge j < m$. The bound requirements with $t(i+1,j) < t_0$ and $t(i,j+1) < t_0$, where $t_0 = t(i,j)$, are established by $i := i+1$ and $j := j+1$ respectively.
- So what remains is "How do we get from $p \wedge \neg B$ to $noMatch(i+1,j)$ or to $NoMatch(i,j+1)$?" The invariant tells us that $noMatch(i,j)$ holds, so no value in $b_1[0..i-1]$ equals any value in $b_2[0..j-1]$. Certainly if $b_1[i] = b_2[j]$, then we've found a match.
- If $b_1[i] > b_2[j]$, which is $\geq b_2[0..j–1]$, then no value in $b_1[0..i]$ equals any value in $b_2[0..j-1]$, so $noMatch(i+1,j)$ holds. Note we can't say $noMatch(i+1,j+1)$ holds because we don't know whether $b_1[i]$ is >, <, or= $b_2[j–1]$.
- Symmetrically, if $b_1[j] > b_2[i]$, which is $\geq b_1[0..i–1]$, then no value in $b_2[0..j]$ equals any value in $b_1[0..i–1]$, so $noMatch(i,j+1)$ holds. We can't say $noMatch(i+1,j+1)$ because we don't know whether $b_2[j]$ is >, <, or = $b_1[i–1]$.
- As partial solutions, we have (the nondeterministic)
  $\{ p \wedge \neg B \}$ ***if*** $b_1[i] < b_2[j] \rightarrow \{ p[i+1/i] \}\, i := i+1$ ***fi*** $\{ p \}$ and [2023-03-29] tests
  $\{ p \wedge \neg B \}$ ***if*** $b_1[i] > b_2[j] \rightarrow \{ p[j+1/j] \}\, j := j+1$ ***fi*** $\{ p \}$
- Combining these gives us

  $\{ p \wedge \neg B \}$
  if $b_1[i] < b_2[j] \rightarrow \{ p[i+1/i] \}\, i := i+1$ [2023-03-29] tests
  $\square\ b_1[i] > b_2[j] \rightarrow \{ p[j+1/j] \}\, j := j+1$
  fi
  $\{ p \}$

- We can make this ***if-fi*** the body of a loop that runs ***while*** $b_1[i] \neq b_2[j]$, and we know the ***if-fi*** won't cause a domain error (where neither of the tests hold). For a deterministic version, since we know $b_1[i] \neq b_2[j]$, then if $b_1[i] > b_2[j]$ is false, then $b_2[j] > b_1[i]$ must hold. We get

  $\{p \wedge \neg B\}$ ***if*** $b_1[i] < b_2[j]$ ***then*** $i := i+1$ ***else*** $j := j+1$ ***fi*** $\{p\}$ [2023-03-29] test
- Adding this to the loop framework (initialization and test), we get

  $\{n \geq 0 \wedge m \geq 0 \wedge Sorted(b, n) \wedge Sorted(b_2, m)\}$
  $i := 0;\ j := 0$
  $\{$ ***inv*** $p \equiv InRange(i,j) \wedge noMatch(i,j) \wedge n - i + m - j \geq 0\}$ $\{$ ***bd*** $n-i+m-j\}$
  while $\neg B$ ***do*** $\{p \wedge \neg B \wedge t(i,j) = t_0\}$ // $\neg B \Leftrightarrow i < n \wedge j < m \,\&\&\, b_1[i] \neq b_2[j]$
    if $b_1[i] < b_2[j]$ ***then*** [2023-03-29] test
      $\{p \wedge \neg B \wedge t(i,j) = t_0 \wedge b_1[i] > b_2[j]\}$
      $\{(p \wedge \neg B)[i+1/i] \wedge t(i+1, j) < t_0\}$
      $i := i+1$
      $\{p \wedge t(i,j) < t_0\}$
    else
      $\{p \wedge \neg B \wedge t(i,j) = t_0 \wedge b_1[i] < b_2[j]\}$
      $\{(p \wedge \neg B)[j+1/j] \wedge t(i, j+1) < t_0\}$
      $j := j+1$
      $\{p \wedge t(i,j) < t_0\}$
    fi $\{p \wedge t(i,j) < t_0\}$
  od
  $\{p \wedge B\}$
  $\{p \wedge (i < n \wedge j < m \rightarrow b_1[i] \neq b_2[j])\}$
- This program can easily be extended to 3 or more arrays.

Example 5: Multiply Integers x and y (version 1: Slowly)

- Our specification is $\{x = x_0 \wedge y = y_0\}\ S\ \{z = x_0 * y_0\}$. ($x_0$ and y_0 are the initial values of x and y.)
- When the loop ends, we want $z = x_0 * y_0$.
- When the loop begins, we have $x_0 * y_0 = x * y$ because $x = x_0 \wedge y = y_0$.
- To get an invariant, define z so that it covers both cases: $z = x_0 * y_0 - x * y$.
  - When the loop begins, $x = x_0$ and $y = y_0$, so $x_0 * y_0 = x * y$, so we'll set $z := 0$.
  - We can end the loop if x or $y = 0$, because $z = x_0 * y_0 - x * y = x_0 * y_0 - 0$.
  - Let's assume $x_0 \geq 0$ initially, so that we can maintain $0 \leq x \leq x_0$ and make progress toward termination by moving x from x_0 toward 0. For the progress step, let's use $x := x-1$.
- Combining everything so far with $x \neq 0$ as the loop test gives us

  $\{x = x_0 \geq 0 \wedge y = y_0\}\ z := 0;$
  $\{$ ***inv*** $p \equiv x \geq 0 \wedge z = x_0 * y_0 - x * y\}$ $\{$ ***bd*** $x\}$
  while $x \neq 0$

do
$\{ p \wedge x \neq 0 \}$
...code to write ... ;
$\{ w \}$ // where $w \equiv wp(x := x-1, p)$
$x := x-1 \{ p \}$
od
$\{ p \wedge x = 0 \} \{ z = x_0 * y_0 \}$

- Above, $w \equiv wp(x := x-1, p) \equiv p[x-1/x] \equiv (z = x_0 * y_0 - (x-1) * y \wedge x-1 \geq 0)$
- The loop body precondition $p \wedge x \neq 0 \equiv (z = x_0 * y_0 - x * y \wedge x \geq 0) \wedge x \neq 0$
- Note p implies $z = x_0 * y_0 - x * y$, but w requires $z = x_0 * y_0 - (x-1) * y$.
  - So we don't have $p \wedge x \neq 0 \rightarrow w$, so we need some code between them to establish this.
  - Recall one way to change $z = e_1$ to $z = e_2$ is $z := z + (e_2 - e_1)$. Here, $e_2 - e_1$ is $(x_0 * y_0 - x * y) - (x_0 * y_0 - (x-1) * y)$, which $= x * y - (x-1) * y$, which $= y$.
  - So $\{ p \wedge x \neq 0 \} z := z + y \{ w \} x := x-1 \{ p \}$
- Our program is

  $\{ x = x_0 \geq 0 \wedge y = y_0 \} z := 0 ;$
  $\{ \textbf{inv } p \equiv z = x_0 * y_0 - x * y \wedge x \geq 0 \} \{ \textbf{bd } x \}$
  while $x \neq 0$ ***do***
  $\{ p \wedge x \neq 0 \wedge x = t_0 \} \{ p[x-1/x][z+y/z] \wedge x-1 < t_0 \}$
  $z := z + y ; \{ p[x-1/x] \wedge x-1 < t_0 \}$
  $x := x-1 \{ p \wedge x < t_0 \}$
  od
  $\{ p \wedge x = 0 \} \{ z = x_0 * y_0 \}$
- Partial correctness of this outline is easy to verify. For total correctness, we need to make sure x can be a bound expression. This is easy: The invariant contains $x \geq 0$ as a conjunct, and the loop body always decrements x.

Example 6: Multiply Integers x and y (version 2: More Quickly)

- The program just finished to multiply integers has a runtime linear in x_0.
- ***The Progress Step Governs the Runtime:*** We can get a faster multiplication program if we make progress toward $x = 0$ more quickly. What if we try $x := x \div 2$?
  - We can still use x as the bound expression: The invariant still implies $x \geq 0$, and if $x \neq 0$, then $x := x \div 2$ brings us strictly closer to 0.
- Instead of a loop body of
  - $\{ p \wedge x \neq 0 \wedge x = t_0 \} z := z + y ; x := x-1 \{ p \wedge x < t_0 \}$

  we have

  $\{ p \wedge x \neq 0 \wedge x = t_0 \} ??? \{ w_1 \} x := x \div 2 \{ p \wedge x < t_0 \}$

where

$w_1 \equiv wp(x := x \div 2, p \wedge x < t_0)$
$\equiv (p \wedge x < t_0)[x \div 2 / x]$
$\equiv p[x \div 2 / x] \wedge x \div 2 < t_0$
$\equiv (z = x_0 * y_0 - (x \div 2) * y) \wedge x \div 2 \geq 0 \wedge x \div 2 < t_0$

- The missing statement has to take us from $p \wedge x \neq 0 \wedge x = t_0$ to w_1.
  - We're already ensured that the $x \div 2 \geq 0$ and $x \div 2 < t_0$ clauses of w_1 hold:
    - p implies $x \geq 0$, so we know $x \div 2 \geq 0$.
    - $x = t_0$ and $x \geq 0 \wedge x \neq 0$ implies $x \div 2 < t_0$.
- We need code to go from $(z = x_0 * y_0 - x * y)$ in p to $(z = x_0 * y_0 - (x \div 2) * y)$ in w_1.
  - If x is even, then $(x \div 2) * (2 * y) = x * y$.
    - So $\{p \wedge even(x)\}\, y := 2 * y;\, \{w_1\}\, x := x \div 2\, \{p\}$
- But we don't know that x is even. We could check for it:

  if $even(x)$
  then *...code above, which requires x to be even ...* $\{w_1\}$
  else
  $\{p \wedge x \neq 0 \wedge odd(x)\}$??? $\{w_1\}$ // Missing code handles x odd case?
  fi
- Or we could ***force*** x to be even:

  $\{p\}$
  if $odd(x)$ ***then*** ???; $x := x-1$ ***fi***; // Missing code enables $x := x-1$ to maintain p
  $\{p \wedge even(x)\}$
  ... above code ...
  $\{w_1\}$
- But ***we already know what we can use*** before the decrement of x.
  - We've already written it once: it's $z := z + y$.
- This completes the program:

  $\{x = x_0 \wedge y = y_0 \wedge x_0 \geq 0\}$
  $z := 0$;
  $\{$***inv*** $p \equiv z = x_0 * y_0 - x * y \wedge x \geq 0\}\, \{$***bd*** $x\}$
  while $x \neq 0$ ***do***
  if $odd(x)$ ***then*** $z := z + y;\, x := x-1$ ***fi***; $\{p \wedge even(x)\}$
  $y := 2 * y;\, x := x \div 2$
  od
  $\{p \wedge x = 0\}\, \{z = x_0 * y_0\}$
- This is a program that implements multiplication by repeated addition and bit-shifting. (Multiplication and division by 2 correspond to left and right bit shifting respectively.) It does roughly $log_2(x_0)$ iterations.

Example 7: Faster Integer Square Root

- For another example of how a faster progress step speeds up a program, recall the integer square root problem (from the previous class). One change: Instead of $n–x+y$ for the bound function, we will be able to use just y because we'll always decrease it (in addition to sometimes increasing x).

  $\{$ ***inv*** $p\}\{$ ***bd*** $y\}$ // where $p \equiv x^2 \le n < (x+y)^2 \wedge y \le 1$
  while $y \ne 1$
  do $\{p \wedge y = y_0\}$
  ... code to write ...
  $\{p\}$
  od
  $\{p \wedge y = 1\}$
  $\{x^2 \le n < (x+1)^2\}$

- To make progress, we need to decrease y. Instead of decrementing y by 1 as before, this time we'll divide it by 2: Using $y := y \div 2$, makes for a binary-search-like method: We test the midpoint $(x+y \div 2)^2$ against n and make it the new left or right endpoint accordingly.

- Here's a partial proof outline:

  $\{$ ***inv*** $p \wedge y \ge 1\}\{$ ***bd*** $y\}$
  while $y \ne 1$ ***do***
  if $(x+y \div 2)^2 > n$ ***then***
  $\{0 \le x^2 \le n < (x+y \div 2)^2 \wedge y \div 2 < t_0\}$
  $y := y \div 2$
  else // $(x+y \div 2)^2 \le n$
  $\{0 \le (x+y \div 2)^2 \le n < (x+y \div 2 + (y - y \div 2))^2 \wedge (y - y \div 2) < t_0\}$
  $x := x + y \div 2 ; y := y - y \div 2$
  fi; $\{0 \le x^2 \le n < (x+y)^2 \wedge y < t_0\}$
  od
  $\{0 \le x^2 \le n < (x+y)^2 \wedge y \ge 1\} \wedge y = 1\}$
  $\{0 \le x^2 \le n < (x+1)^2\}$

- ***Notes***: The invariant implies $y \ge 1$; that with the loop test $y \ne 1$ implies $y \ge 2$. That in turn implies that $y \div 2$ and $y - y \div 2$ are both $< y$, which ensures progress whether the ***if*** test succeeds or fails.

Array Element Assignments

CS 536: Science of Programming, Spring 2023

2023-04-03 pp. 2,4

A. Why?

- Array assignments aren't like assignments to plain variables because the actual item to change can't be determined until runtime. We can handle this by extending our notion of assignment and/or substitution.

B. Outcomes

After this class, you should

- Know how to perform textual substitution to replace an array element.
- Know how to calculate the *wp* of an array element assignment.

C. Array Element Assignments

- An array assignment $b[e_0] := e_1$ (where e_0 and e_1 are expressions) is different from a plain variable assignment because the exact element being changed may not be known at program annotation time. E.g., compare these two triples:
  - ***Valid***: $\{T\}\ x := y;\ y := y + 1\ \{x < y\}$
  - ***Invalid***: $\{T\}\ b[k] := b[j];\ b[j] := b[j] + 1\ \{b[k] < b[j]\}$
- The problem is what happens if $k = j$ at runtime: What is $wp(b[j] := b[j] + 1, b[k] < b[j])$?
- The answer should be something like "If $k \neq j$ then $b[k] < b[j] + 1$ else $b[j] + 1 < b[j] + 1$". (Note the else clause is false.)
- There are two alternatives for handling array assignments. The one we'll use involves defining the *wp* of an array assignment using an extended notion of textual substitution:

  $$wp(b[e_0] := e_1, p) \equiv p[e_1 / b[e_0]] \text{ and } \{p[e_1 / b[e_0]]\}\ b[e_0] := e_1\ \{p\}$$

- Of course, we need to figure out what syntactic substitution for an array indexing expression means: $(predicate)[expression / b[e_0]]$
- Side note: The other way to handle array assignments, the Dijkstra / Gries technique, is to introduce a new kind of expression and view the array assignment $b[e_0] := e_1$ as short for $b :=$ this new kind of expression.

D. Substitution for Array Elements

- We'll need to substitute into expressions and predicates. We'll tackle expressions first; below.

- If b and d are different arrays, then a substitution like $(b[m])[6 / d[2])$ should simply $\equiv b[m]$. The situation can be more complicated: The substitution $(b[e])[6 / d[2])$ has to recursively look for substitutions to do inside e.
  - $(b[e_2])[e_0 / d[e_1]] \equiv b[e_2']$ where $e_2' \equiv (e_2)[e_0 / d[e_1]]$. [2023-04-03]
- When the the array names match, as in $(b[k])[e_0 / b[e_1]]$, we have to check the indexes k and e_0 for equality at runtime; to do that, we can use a conditional expression.
- ***Definition (Substitution for an Array Element) — Simpler situation***
  - At runtime, if $k = e_1$, then $(b[k])[e_0 / b[e_1]] = e_0$. If $k \neq e_1$, then$(b[k])[e_0 / b[e_1]] = b[k]$. (The sense of "=" here is that the two expressions evaluate to the same value.)
    - Textually, $(b[k])[e_0 / b[e_1]] \equiv$ ***if*** $k = e_1$ ***then*** e_0 ***else*** $b[k]$ ***fi***.
- ***Example 1***:$(b[k])[5 / b[0]] \equiv (***if*** k = 0$ ***then*** 5 ***else*** $b[k]$ ***fi***)$.
- ***Example 2***: $(b[k])[e_0 / b[j]] \equiv (***if*** k = j$ ***then*** e_0 ***else*** $b[k]$ ***fi***)$.
- ***Example 3***: $(b[k])[b[j] + 1 / b[j]] \equiv (***if*** k = j$ ***then*** $b[j] + 1$ ***else*** $b[k]$ ***fi***)$.
  - Note: In $(b[k])[e_0 / b[e_1]]$, we don't substitute into e_0, even if it involves b.
- ***Example 4***:$(b[k])[b[i] / b[j]] \equiv (***if*** k = j$ ***then*** $b[i]$ ***else*** $b[k]$ ***fi***)$.

The General Case for Array Element Substitution

- When e_2 is not just a simple variable or constant, then in $(b[e_2])[e_0 / b[e_1]]$, we have to check e_2 for uses of $b[...]$ and substitute for them also.
- ***Definition (Substitution for an Array Element) — General Case***

  $(b[e_2])[e_0 / b[e_1]] \equiv$ ***if*** $e_2' = e_1$ ***then*** e_0 ***else*** $b[e_2']$ ***fi*** where $e_2' \equiv (e_2)[e_0 / b[e_1]]$.
- This subsumes the earlier case, since if $e_2 \equiv k$ then $e_2' \equiv k[e_0 / b[e_1]] \equiv k$. We get

$(b[k])[e_0 / b[e_1]] \equiv$ ***if*** $k = e_1$ ***then*** e_0 ***else*** $b[k]$ ***fi***

Example 5

- Consider $(b[b[k]])[5 / b[0]]$ — how should it behave? The inner, nested $b[k]$ should behave like 5 if $k = 0$, otherwise it should behaves like $b[k]$ as usual. The outer $b[...]$ should behave like 5 if its index behaves like 0, otherwise it should behave as $b[...]$.
- Following the definition above, we get

  $(b[b[k]])[5 / b[0]] \equiv$ ***if*** $e_2' = 0$ ***then*** 5 ***else*** $b[e_2']$ ***fi***

  where $e_2' \equiv (b[k])[5 / b[0]] \equiv (***if*** k = 0$ ***then*** 5 ***else*** $b[k]$ ***fi***$)$
- Substituting the (textual) value of e_2' gives us

$(b[b[k]])[5 / b[0]]$

$\equiv$ ***if*** $(***if*** k = 0$ ***then*** 5 ***else*** $b[k]$ ***fi***$) = 0$

then 5

else $b[$ ***if*** $k = 0$ ***then*** 5 ***else*** $b[k]$ ***fi***$]$ ***fi***

- After optimization, this is equivalent to ***if*** $k = 0$ ***then*** $b[5]$ ***else if*** $b[k] = 0$ ***then*** 5 ***else*** $b[b[k]]$ ***fi fi.***

E. Optimization of Static Cases

- Because $e[e_0 / b[e_1]]$ can result in a complicated piece of text, it can be useful to shorten it using various optimizations, similarly to how compilers can optimize code.
- All the optimizations below are intended to be done "statically" (at compile time) — we inspect the text of an expression before the code ever runs.
- For the easiest examples, if we know whether or not $k = e_1$, the index of b we're looking for, then we can optimize $\textbf{if } k = e_0 \textbf{ then } e_1 \textbf{ else } e_2 \textbf{ fi}$ to just the true branch or the false branch.
- ***Notation:*** $e_1 \mapsto e_2$ ("e_1 optimizes to e_2") means we can replace expression e_1 with e_2.

General Principle (Static Optimizations)

- (Restricted case): For $(b[k])[e_0 / b[e_1]]$
  - If[1] $k = e_1$, then $(b[k])[e_0 / b[e_1]] \mapsto e_0$.
  - If $k \neq e_1$, then $(b[k])[e_0 / b[e_1]] \mapsto b[k]$.
- (General case): For $(b[e_2])[e_0 / b[e_1]]$, let $e_2' \equiv (e_2)[e_0 / b[e_1])$
  - If $e_2' = e_1$, then $(b[e_2])[e_0 / b[e_1]] \mapsto e_0$.
  - If $e_2' \neq e_1$, then $(b[e_2])[e_0 / b[e_1]] \mapsto b[k]$.

- ***Example 6***: $(b[0])[e_1 / b[2]] \equiv \textbf{if } 0 = 2 \textbf{ then } e_1 \textbf{ else } b[0] \textbf{ fi} \mapsto b[0]$.
- ***Example 7***: $(b[2])[e_1 / b[2]] \equiv \textbf{if } 2 = 2 \textbf{ then } e_1 \textbf{ else } b[2] \textbf{ fi} \mapsto e_1$.
- ***Example 8***:
  - $(b[0])[e_0 / b[1]] \equiv \textbf{if } 0 = 1 \textbf{ then } e_0 \textbf{ else } b[0] \textbf{ fi} \mapsto b[0]$.
  - $(b[1])[e_0 / b[1]] \equiv \textbf{if } 1 = 1 \textbf{ then } e_0 \textbf{ else } b[1] \textbf{ fi} \mapsto e_0$.
  - $(b[1])[3 / b[2]] \equiv \textbf{if } 1 = 2 \textbf{ then } 3 \textbf{ else } b[1] \textbf{ fi} \mapsto b[1]$.
  - $(b[x])[e_0 / b[x]] \equiv \textbf{if } x = x \textbf{ then } e_0 \textbf{ else } b[x] \textbf{ fi} \mapsto e_0$.

F. Rules for Simplifying Conditional Expressions

- Let's identify some general rules for simplifying conditional expressions and predicates involving them. This will let us simplify calculation of *wp* for array assignments.
  - $(\textbf{if } T \textbf{ then } e_1 \textbf{ else } e_2 \textbf{ fi}) \mapsto e_1$.
  - $(\textbf{if } F \textbf{ then } e_1 \textbf{ else } e_2 \textbf{ fi}) \mapsto e_2$.
  - $(\textbf{if } B \textbf{ then } e \textbf{ else } e \textbf{ fi}) \mapsto e$.
  - If $(B \rightarrow e_1 = e_2)$, then $(\textbf{if } B \textbf{ then } e_1 \textbf{ else } e_2 \textbf{ fi}) \mapsto e_2$.
  - If $(\neg B \rightarrow e_1 = e_2)$, then $(\textbf{if } B \textbf{ then } e_1 \textbf{ else } e_2 \textbf{ fi}) \mapsto e_1$.

---

[1] The fuller version is "If we know that ... then ... ↦ ..."

- Let $\ominus$ be a unary operator or relation and $\oplus$ be a binary operation or relation
  - $\ominus$*(if* B *then* e_1 *else* e_2 *fi)* $\mapsto$*(if* B *then* $\ominus e_1$ *else* $\ominus e_2$ *fi)*
  - *(if* B *then* e_1 *else* e_2 *fi)* $\oplus e_3$ $\mapsto$*(if* B *then* $e_1 \oplus e_3$ *else* $e_2 \oplus e_3$ *fi)*
  - b*[if* B *then* e_1 *else* e_2 *fi]* $\mapsto$ *if* B *then* b*[* e_1*]* *else* b*[* e_2*]* *fi*
  - For any function $f(\ldots)$, f*(if* B *then* e_1 *else* e_2 *fi)* $\mapsto$ *if* B *then* $f(e_1)$ *else* $f(e_2)$ *fi*
- If B, B_1, and B_2 are boolean expressions, then
  - *(if* B *then* B_1 *else* F *fi)* $\Leftrightarrow (B \wedge B_1)$
  - *(if* B *then* F *else* B_2 *fi)* $\Leftrightarrow (\neg B \wedge B_2)$
  - *(if* B *then* B_1 *else* T *fi)* $\Leftrightarrow (B \rightarrow B_1) \Leftrightarrow (\neg B \vee B_1)$
  - *(if* B *then* T *else* B_2 *fi)* $\Leftrightarrow (\neg B \rightarrow B_2) \Leftrightarrow (B \vee B_2)$
  - *(if* B *then* B_1 *else* B_2 *fi)* $\Leftrightarrow ((B \rightarrow B_1) \wedge (\neg B \rightarrow B_2)) \Leftrightarrow ((B \wedge B_1) \vee (\neg B \wedge B_2))$.
- We can also do reordering of ***if-else-if*** chains. E.g.,
  - *if* B_1 *then* e_1 *else if* B_2 *then* e_2 *else* e_3 *fi* evaluates e_1 if B_1 (regardless of B_2); it evaluates e_2[2023-04-03] if $\neg B_1 \wedge B_2$; and it evaluates e_3 if $\neg B_1 \wedge \neg B_2$.
  - So we (for example) swap e_2 and e_3 by changing the test slightly:
    - *if* B_1 *then* e_1 *else if* B_2 *then* e_2 *else* e_3 *fi*
      $\mapsto$ *if* B_1 *then* e_1 *else if* $\neg B_2$ *then* e_3 *else* e_2 *fi*

      or

      $\mapsto$ *if* $\neg B_1 \wedge B_2$ *then* e_2 *else if* B_1 *then* e_1 *else* e_3 *fi*

      and so on.
- Similarly, we can move an inner ***if - else*** from the true branch of an outer ***if - else*** to the false branch of the outer ***if - else*** , in order to make an ***if-else-if*** chain. For example,
  - *if* B_1 *then if* B_2 *then* /*$B_1 \wedge B_2$*/ e_1 *else* /*$B_1 \wedge \neg B_2$*/ e_2 *fi else* /*$\neg B_1$*/ e_3 *fi*
    $\mapsto$ *if* $\neg B_1$ *then* e_3 *else if* B_2 *then* e_1 *else* e_2 *fi fi*

- ***Example 9***:

  $wp(b[j] := b[j] + 1, b[k] < b[j])$
  $\equiv (b[k] < b[j])[b[j] + 1 / b[j]]$
  $\equiv (b[k])[b[j] + 1 / b[j]] < (b[j])[b[j] + 1 / b[j]]$
  $\equiv$ *if* $k = j$ *then* $b[j] + 1$ *else* $b[k]$ *fi* $< b[j] + 1$
  $\Leftrightarrow$ *if* $k = j$ *then* $b[j] + 1 < b[j] + 1$ *else* $b[k] < b[j] + 1$ *fi*
  $\Leftrightarrow$ *if* $k = j$ *then* F *else* $b[k] < b[j] + 1$ *fi*
  $\Leftrightarrow k \neq j \wedge b[k] < b[j] + 1$

This gives us the following correctness triple:

$\{k \neq j \wedge b[k] < b[j] + 1\}\ b[j] := b[j] + 1\ \{b[k] < b[j]\}$

G. Swapping Array Elements

- To illustrate the use of array references, let's look at the problem of swapping array elements.
- To swap simple variables x and y using a temporary variable u, we can use logical variables c and d and prove

  $\{x = c \wedge y = d\}\ u := x;\ x := y;\ y := u\ \{x = d \wedge y = c\}$
- We can prove this program correct by expanding to a full proof outline; here we're using *wp*.

  $\{x = c \wedge y = d\}$
  $\{y = d \wedge x = c\}\ u := x;$
  $\{y = d \wedge u = c\}\ x := y;$
  $\{x = d \wedge u = c\}\ y := u$
  $\{x = d \wedge y = c\}$

- ***Example 10***: For swapping $b[m]$ and $b[n]$, we want to prove

  $\{b[m] = c \wedge b[n] = d\}\ u := b[m];\ b[m] := b[n];\ b[n] := u\ \{b[m] = d \wedge b[n] = c\}$

  As with simple variables, we can prove this holds by using *wp* to expand to the full proof outline.

  Let $p \equiv b[m] = c \wedge b[n] = d$ and $q \equiv b[m] = d \wedge b[n] = c$, then we can prove

  $\{p\}\ \{q_3\}\ u := b[m];\ \{q_2\}\ b[m] := b[n];\ \{q_1\}\ b[n] := u\ \{q\}$

  by using
  - $q_1 \equiv wp(b[n] := u, q) \equiv q[u / b[n]]$,
  - $q_2 \equiv wp(b[m] := b[n], q_1) \equiv q_1[b[n] / b[m]]$
  - $q_3 \equiv wp(u := b[m], q_2) \equiv q_2[b[m] / u]$
  - (and hopefully) $p \rightarrow q_3$

  We'll do this in steps.
  - $q_1 \equiv q[u / b[n]]$
    $\equiv (b[m] = d \wedge b[n] = c)[u / b[n]]$
    $\equiv (b[m] = d)[u / b[n]] \wedge (b[n] = c)[u / b[n]]$
    $\equiv (b[m])[u / b[n]] = d \wedge (b[n])[u / b[n]] = c$
    $\equiv (\textbf{if}\ m = n\ \textbf{then}\ u\ \textbf{else}\ b[m]\ \textbf{fi}) = d \wedge u = c$ // Stop here for a purely syntactic result
  - $q_2 \equiv q_1[b[n] / b[m]]$
    $\equiv ((\textbf{if}\ m = n\ \textbf{then}\ u\ \textbf{else}\ b[m]\ \textbf{fi}) = d \wedge u = c)[b[n] / b[m]]$
    $\equiv (\textbf{if}\ m = n\ \textbf{then}\ u\ \textbf{else}\ (b[m])[b[n] / b[m]]\ \textbf{fi}) = d \wedge u = c$
    $\equiv (\textbf{if}\ m = n\ \textbf{then}\ u\ \textbf{else}\ b[n]\ \textbf{fi}) = d \wedge u = c$
  - $q_3 \equiv q_2[b[m] / u]$
    $\equiv ((\textbf{if}\ m = n\ \textbf{then}\ u\ \textbf{else}\ b[n]\ \textbf{fi}) = d \wedge u = c)[b[m] / u]$
    $\equiv (\textbf{if}\ m = n\ \textbf{then}\ b[m]\ \textbf{else}\ b[n]\ \textbf{fi}) = d \wedge b[m] = c)$
    // Continuing with logical manipulation

$\Leftrightarrow$ *(**if** m = n **then** b [n] **else** b [n] **fi)** = d ∧ b [m] = c)*

// Because if $m = n$ then $b[m] = b[n]$

$\Leftrightarrow b[n] = d \wedge b[m] = c$.

- Since $p \equiv b[m] = c \wedge b[n] = d$, we get $p \rightarrow q_3$. (End of Example 10)

Basics of Parallel Programs

CS 536: Science of Programming, Spring 2023

2023-04-06: pp 3–6

A. Why?

- Parallel programs are more flexible than sequential programs but their execution is more complicated.
- Parallel programs are harder to reason about because parts of a parallel program can interfere with other parts.
- Evaluation graphs can be used to show all possible execution paths for a parallel program.

B. Objectives

After this class, you should know

- The syntax and operational & denotational semantics of parallel programs.

C. Basic Definitions for Parallel Programs

- ***Syntax*** for parallel statements: $S := [\, S \,||\, S \,||\, \ldots \,||\, S \,]$. We say $[\, S_1 \,||\, S_2 \,||\, \ldots \,||\, S_n \,]$ is the ***parallel composition*** of the ***threads*** $S_1, S_2, \ldots, S_n$.
  - The threads must be sequential: You can't nest parallel programs. (But you can embed parallel programs within otherwise-sequential programs, such as in the body of a loop.)
- ***Example 1***: $[\, x := x + 1 \,||\, x := x * 2 \,||\, y := x^2 \,]$ is a parallel program with three threads. Since it tries to nest parallel programs, $[\, x := x + 1 \,||\, [\, x := x * 2 \,||\, y := x^2 \,] \,]$ is illegal.

Interleaving Execution of Parallel Programs

- We run sequential threads in parallel by ***interleaving*** their execution. I.e., we interleave the operational semantics steps for the individual threads.
- We execute one thread for some number of operational steps, then execute another thread, etc.
- Depending on the program and the sequence of interleaving, a program can have more than one final state (or cause an error sometimes but not other times).
- As an example, since evaluation of $[\, x := x + 1 \,||\, x := x * 2 \,]$ is done by interleaving the operational semantics steps of the two threads, we can either evaluate $x := x + 1$ and then $x := x * 2$ or evaluate $x := x * 2$ and then $x := x + 1$.
- The difference between $[\, x := x + 1 \,||\, x := x * 2 \,]$ and ***if*** $T \rightarrow x := x + 1 \;\square\; T \rightarrow x := x * 2$ ***fi*** is that the nondeterministic ***if-fi*** executes only one of the two assignments whereas the parallel composition executes both assignments but in an unpredictable order. The sequential nondeterministic ***if-fi*** that simulates the parallel assignments is ***if*** $T \rightarrow x := x + 1 \,;\, x := x * 2 \;\square\; T \rightarrow x := x * 2 \,;\, x := x + 1$

fi. It nondeterministically chooses between the two possible traces of execution for the program.[1]

- Because of the nondeterminism, re-executions of a parallel program can use different orders. For example, two executions of ***while*** B ***do*** $[\, x := x + 1 \;||\; x := x * 2 \,]$ ***od*** can have the same sequence or different sequences of updates to x.

Difficult to Predict Parallel Program Behavior

- The main problem with parallel programs is that their properties can be very different from the behaviors of the individual threads.
- ***Example 2***:
  - $\vDash \{ x = 5 \}\, x := x + 1 \,\{ x = 6 \}$ and $\vDash \{ x = 5 \}\, x := x * 2 \,\{ x = 10 \}$
  - But $\vDash \{ x = 5 \}\, [\, x := x + 1 \;||\; x := x * 2 \,]\, \{ x = 11 \vee x = 12 \}$
- The problem with reasoning about parallel programs is that different threads can ***interfere*** with each other: They can change the state in ways that don't maintain the assumptions used by other threads.
- Full interference is tricky, so we're going to work our way up to it. First we'll look at simple, limited parallel programs that don't interact at all (much less interfere).
- But before that, we need to look at the semantics of parallel programs more closely.

D. Semantics of Parallel Programs

- To execute the sequential composition $S_1 ; \ldots ; S_n$ for one step, we execute S_1 for one step.
- To execute the parallel composition $[\, S_1 \;||\; \ldots \;||\; S_n \,]$ for one step, we take one of the threads and evaluate it for one step.

Operational and Denotational Semantics of Parallel Programs

- ***Definition***: Given $[\, S_1 \;||\; \ldots \;||\; S_n \,]$, for each $k = 1, 2, \ldots, n$, if $\langle S_k, \sigma \rangle \to \langle T_k, \tau_k \rangle$, then $\langle [\, S_1 \;||\; \ldots \;||\; S_n \,], \sigma \rangle \to \langle [\, S_1 \;||\; \ldots \;||\; S_{k-1} \mid T_k \;||\; S_{k+1} \;||\; \ldots \;||\; S_n \,], \tau_k \rangle$
- We write E for sequential thread that has finished execution, so a parallel program that has finished execution is written $[\, E \;||\; \ldots \;||\; E \;||\; E \,]$. We'll treat E and $[\, E \;||\; \ldots \;||\; E \;||\; E \,]$ as being syntactically equal, i.e., $E \equiv [\, E \;||\; \ldots \;||\; E \;||\; E \,]$.

The $\to$ Notation*

- ***Notation***: The $\to$* notation has the same meaning whether the configurations involved have parallel programs or not: $\to$* means $\to^n$ for some $n \geq 0$, and $C_0 \to^n C_n$ means we've omitted writing the out intermediate configurations in the sequence $C_0 \to C_1 \to \ldots \to C_{n-1} \to C_n$ (for some collection of C.)

---

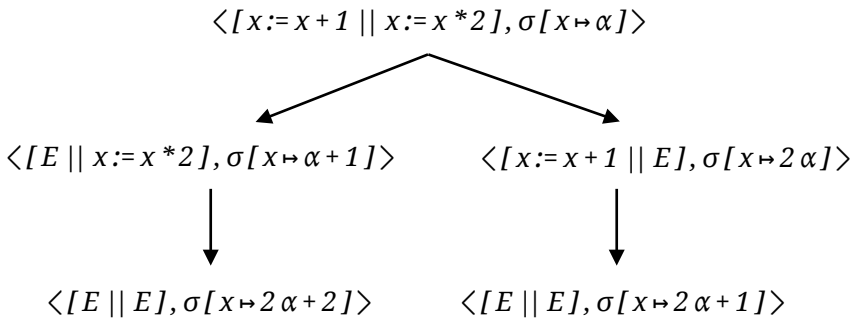
[1] This trick doesn't scale up well to larger programs, but it helps with initially understanding parallel execution.

- ***Common Mistake***: Writing $\langle [\, E \,||\, E \,], \tau \rangle \to \langle E, \tau \rangle$ is a common mistake. Since $[\, E \,||\, E \,] \equiv E$, going from $\langle [\, E \,||\, E \,], \tau \rangle$ to $\langle E, \tau \rangle$ doesn't involve an execution step. But $\langle [\, E \,||\, E \,], \tau \rangle \to^0 \langle E, \tau \rangle$ is ok because it says that in zero steps, we go from one empty configuration to itself.

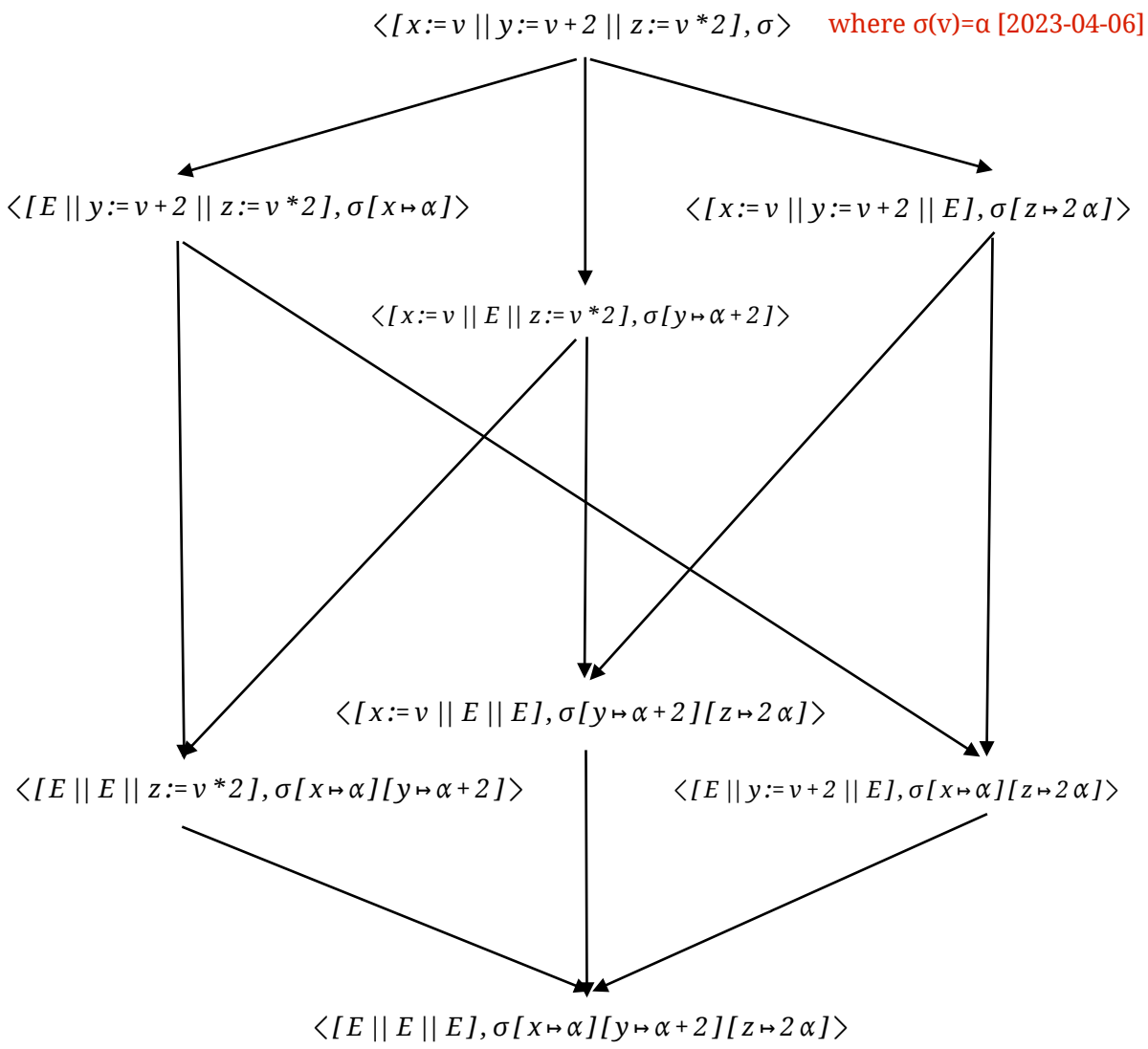
Evaluation Graph and Denotational Semantics

- Recall that the ***evaluation graph*** for $\langle S, \sigma \rangle$ is the directed graph of configurations and evaluation arrows leading from $\langle S, \sigma \rangle$.
- When drawing evaluation graphs, the configuration nodes need to be different.
  - (I.e., if the same configuration appears more than once, show multiple arrows into it — don't repeat the same node.)
- An evaluation graph shows all possible executions.
  - A program with n threads will have n out-arrows from its configuration.
  - (Exception: Evaluation graphs are not multigraphs: If two arrows go to exactly the same configuration, we write the configuration just once and write exactly one arrow to it.)
- A path through the graph corresponds to one possible evaluation of the program.
- The ***denotational semantics*** of a program in a state is the set of all possible terminating states (plus possibly the pseudostates $\perp_d$ and $\perp_e$). I.e., the states found in the sinks (i.e., at the leaves) of an evaluation graph. (We'll modify this definition when we get to deadlocked programs.)
  - $M(S, \sigma) = \{ \tau \in \sigma \mid \langle S, \sigma \rangle \to^* \langle E, \tau \rangle \}$
    $\cup \{ \perp_d \}$ if S can diverge; i.e., if $\langle S, \sigma \rangle \to^* \langle E, \perp_d \rangle$ is possible [2023-04-06]
    $\cup \{ \perp_e \}$ if S can produce a runtime error; i.e., $\langle S, \sigma \rangle \to^* \langle E, \perp_e \rangle$ is possible. [2023-04-06]

- ***Example 3***: The evaluation graph below is for the same program as in Example 2, but starting with an arbitrary state σ where $\sigma(x) = \alpha$. The graph has two sinks for the two possible final states, so $M([\, x := x+1 \,||\, x := x*2 \,], \sigma) = \{ \sigma[x \mapsto 2\alpha+2], \sigma[x \mapsto 2\alpha+1] \}$.



Example 3

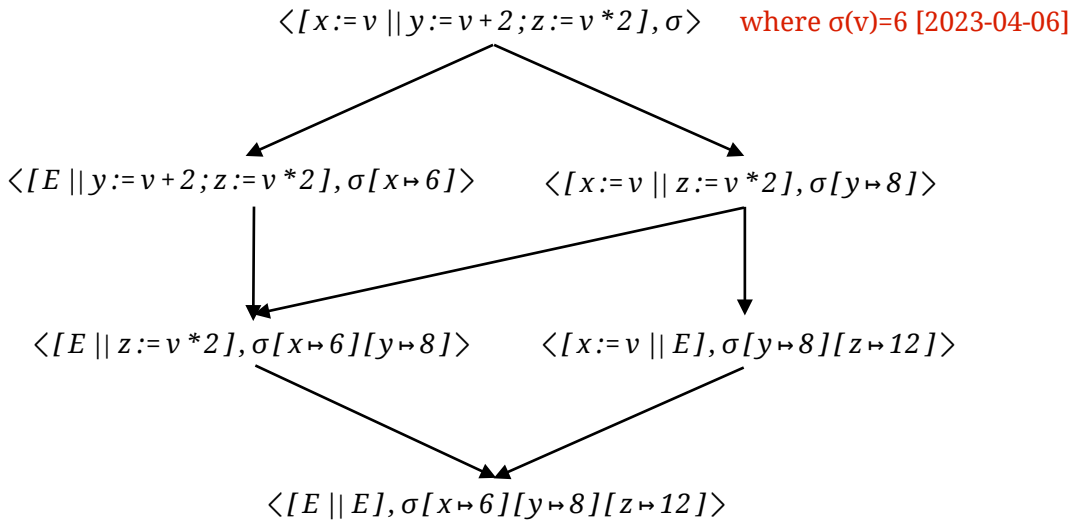


Example 4

- ***Example 4***: For this example, the evaluation graph is for $\langle [\, x := v \;||\; y := v + 2 \;||\; z := v * 2 \,], \sigma \rangle$, where $\sigma(v) = \alpha$. $M([\, x := v \;||\; y := v + 2 \;||\; z := v * 2 \,], \sigma) = \{\sigma[\, x \mapsto \alpha \,][\, y \mapsto \alpha + 2 \,][\, z \mapsto 2\,\alpha \,]\}$. Note even though the program is nondeterministic, it produces the same result no matter what execution path it uses.

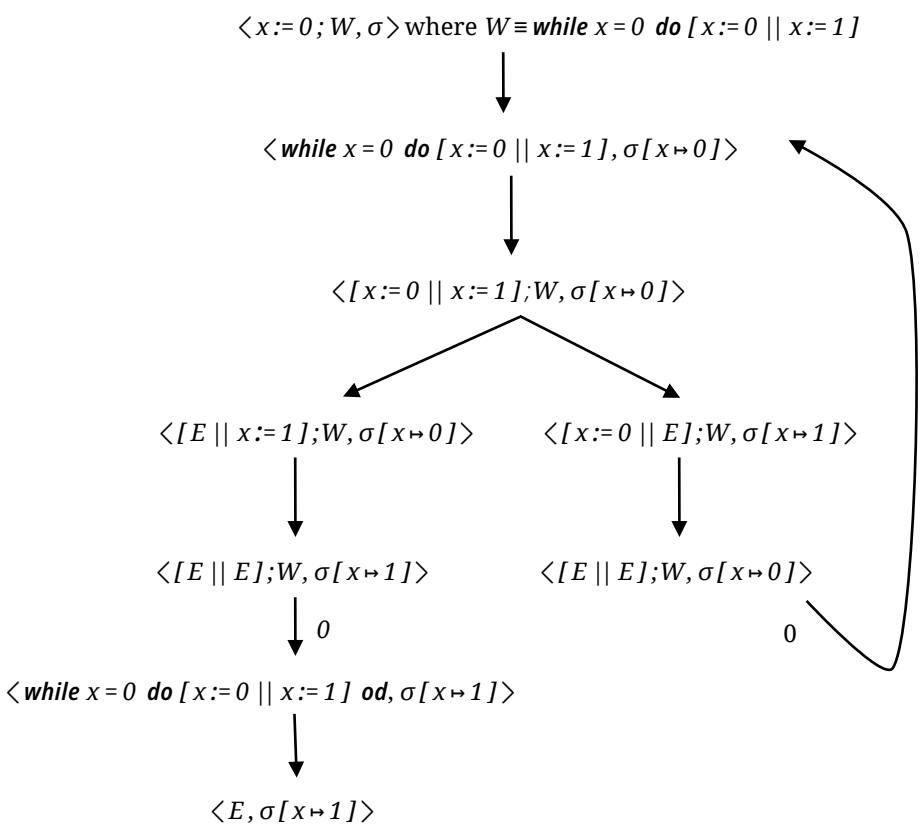
  (More generally, if S is parallel, then $M(S, \sigma)$ can have more than 1 member, but the converse is not true: Having $M(S, \sigma)$ of size 1 does not imply that S is nondeterministic.)

- ***Example 5***: If we take the program from Example 4 and combine the last two threads sequentially, then the evaluation graph for the resulting program is a subgraph of the graph from Example 4. Below, $\sigma(v) = 6$, and $M([x := v \,||\, y := v + 2 \,||\, z := v * 2], \sigma) = \{\sigma[x \mapsto 6][y \mapsto 8][z \mapsto 12]\}$.



Example 5

- ***Example 6***: Let $W \equiv x := 0;$ ***while*** $x = 0$ ***do*** $[x := 0 \,||\, x := 1]$ ***od***. Then $M(W, \sigma) = \{\sigma[x \mapsto 1], \perp_d\}$. as shown in the evaluation graph. Note the transitions $\langle [E \,||\, E]; W, \sigma[x \mapsto \ldots] \rangle \rightarrow^0 \langle W, \sigma[x \mapsto \ldots] \rangle$ take 0 steps because $[E \,||\, E]; W \equiv E; W \equiv W$; that is, they're all the same program, textually.
- The problem in this example is that there is possible divergence.
  - On the other hand, it only happens if we ***always*** choose thread 1 when we have to make the nondeterministic choice of $[x := 0 \,||\, x := 1]$.
  - This is definitely unfair behavior, but it's allowed because of the unpredictability of our nondeterministic choices. In real life, we would want a fairness mechanism to ensure that all threads get to evaluate once in a while.
- If each thread is on a separate processor, then the nondeterministic choice corresponds to which processor is fastest, so the possible divergence of the program is a ***race condition***, where the correct behavior of a program depends on the relative speed of the processors involved. Here, divergence occurs if the processor for x:=1 is always faster than the processor for x:=0. [2023-04-06]
- Note that it's not necessarily a race condition to have a parallel program producing different results when run multiple times. As long as all results satisfy the specification, there's no race condition.



Example 6

- ***Example 7***: The correctness triple *{ T } [x := 0 || x := 1] { x ≥ 0 }* does not have a race condition, but *{ T } [x := 0 || x := 1] { x > 0 }* does. [2023-04-06] The program terminates with *x = 0* or *1*. With postcondition *x ≥ 0* , both states are correct even though they're different. But with post-condition *x > 0* , the relative speed of the threads means we may or may not produce a correct result.

Disjoint Programs

CS 536: Science of Programming, Spring 2023

A. Why?

- Parallel programs are harder to reason about because parts of a parallel program can interfere with other parts.
- Reducing the amount of interference between threads lets us reason about parallel programs by combining the proofs of the individual threads.
- Disjoint parallel programs ensure that no thread can interfere with the execution of another thread.
- The sequentialization rule (though imperfect) gives us a way to prove the correctness of disjoint parallel programs.

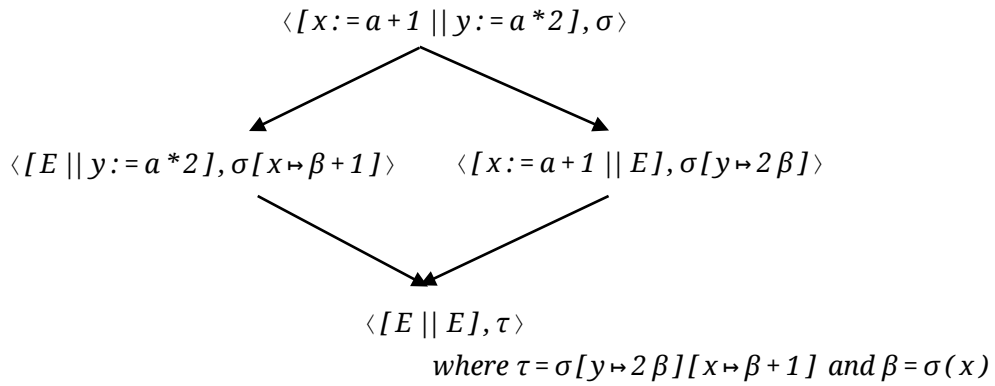
B. Objectives

After this class, you should know

- What distinguishes disjoint parallel programs
- The sequentialization rule for disjoint parallel programs

C. Disjoint Parallel Programs

- The following example shows a program with an innocuous kind of parallelism: no matter what order we execute the threads in, we end up in the same final state.
- ***Example 1***: Here is the the evaluation graph for $\langle [\, x := a + 1 \;||\; y := a * 2 \,], \sigma \rangle$ where $\beta = \sigma(a)$. The final state is $\sigma[\, x \mapsto \beta + 1 \,][\, y \mapsto 2\,\beta \,]$ if we take the left-hand path and $\sigma[\, y \mapsto 2\,\beta \,][\, x \mapsto \beta + 1 \,]$ if we take the right-hand path, but since $x \neq y$, these two states are exactly the same, so we show two arrows going to the final state configuration.



Disjoint Parallel Programs (DPPs) model computations with n processors that share readable memory but not writable memory. In a disjoint parallel program, for every variable x that appears in the program, either

- One or more threads reads x (i.e., look up its value) and no thread writes to x (i.e., assigns it a value).
- Exactly one thread writes to x and that thread can read x; no other thread can read or write x.

- ***Definition***: *vars (S)* is the set of variables that appear in S and *change (S)* is the set of variables that appear on the left-hand side of assignments in S. Since these sets are statically calculable, they are $\supseteq$ the sets of variables actually read or written at runtime. Another way to say this is that execution order isn't taken into account. E.g., If $S \equiv$ ***if*** B ***then*** $x := 1$ ***else*** $y := 1$ ***fi*** then $change(S) = \{x, y\}$.
- ***Definition***: The threads $S_1, S_2, \ldots, S_n$ are ***pairwise disjoint*** if no thread can change the variables used by any other: I.e., $change(S_i) \cap vars(S_j) = \varnothing$ for all $1 \le i \ne j \le n$.
- ***Example 2***: $S_1 \equiv a := a + x$ and $S_2 \equiv y := y + x$ are disjoint: $change(S_1) = \{a\}$ and $vars(S_2) = \{x, y\}$ and these sets don't intersect. Similarly, $change(S_2) = \{y\}$ and $vars(S_1) = \{a, x\}$ and those sets don't intersect.
- ***Definition***: For $n > 1$, if $S_1, S_2, \ldots, S_n$ are pairwise disjoint, then $[S_1 \,||\, \ldots \,||\, S_n]$ is their ***disjoint parallel composition***. We also say $[S_1 \,||\, \ldots \,||\, S_n]$ is a ***disjoint parallel program*** (DPP).
- ***Example 3:***
  - $a := a + x$ and $y := y + x$ are disjoint, so $[a := a + x \,||\, y := y + x]$ is a DPP.
  - $a := x + 1$ and $y := x + 2$ are disjoint, so $[a := x + 1 \,||\, y := x + 2]$ is a DPP.
  - $a := x$ and $x := c$ are not disjoint so $[a := x \,||\, x := c]$ isn't a DPP.
  - $a := x$ and $x := x + 1$ are not disjoint so $[a := x \,||\, x := x + 1]$ isn't a DPP.
  - $x := a + 1$ and $x := b * 2$ are not disjoint so $[x := a + 1 \,||\, x := b * 2]$ isn't a DPP.
- An easy way to calculate whether or not programs are pairwise disjoint is to use a table listing the $change(S_j)$ and $vars(S_k)$ sets for each pair of pair of threads.
- ***Definitions***
  - Thread S_j ***(apparently) interferes with*** thread S_k if $change(S_j) \cap vars(S_k) \ne \varnothing$.
  - Thread S_j ***is disjoint with*** thread S_k if $change(S_j) \cap vars(S_k) = \varnothing$.
  - Threads S_j and S_k are ***disjoint*** if they are disjoint with each other (S_j with S_k and S_k with S_j).
  - A collection of threads is ***pairwise disjoint*** if each pair of two different threads is disjoint. Note for a collection of n threads, there are $n * (n - 1)$ such pairs.
- For convenience and flexibility, we'll often omit the "apparently" in "apparently interferes with" and we'll allow phrases like "doesn't interfere with" and "isn't disjoint with" as synonyms or "is disjoint with" and "(apparently) interferes with". Similarly, "can/can't change the variables of" means "interferes with/is disjoint with".

- ***Example 4***: Here is a table for $a := a + x$ and $y := y + x$, showing that they are pairwise disjoint:

j	*k*	*Change j*	*Vars k*	*j Disjoint with k*
1	2	a	$x\ y$	yes
2	1	y	$a\ x$	yes

Conclusion: The two programs are pairwise disjoint.

- ***Example 5:*** Here's a table for and $a := x$ and $x := c$ showing that while the first doesn't interfere with the second, the second does interfere with the first, which makes the pair not disjoint.

j	*k*	*Change j*	*Vars k*	*j Disjoint with k*
1	2	a	$c\ x$	yes
2	1	x	$a\ x$	no

Conclusion: The two programs are not pairwise disjoint.

- ***Example 6:*** Here's a table showing the interference relationships for the three threads
  - $a := v;\ \ v := c + b$
  - ***if*** $b > 0$ ***then*** $b := c * b$ ***else*** $c := c * 2$ ***fi***
  - ***while*** $d \geq 0$ ***do*** $d := d \div 2 - c$ ***od***

j	*k*	*Change j*	*Vars k*	*j Disjoint with k*
1	2	$a\ v$	$b\ c$	yes
1	3	$a\ v$	$c\ d$	yes
2	1	$b\ c$	$a\ b\ c\ v$	no
2	3	$b\ c$	$c\ d$	no
3	1	d	$a\ b\ c\ v$	yes
3	2	d	$b\ c$	yes

Conclusion: Thread 2 interferes with threads 1 and 3; the other combinations are disjoint

- ***Example 7:*** This example is similar to Example 6 but only causes interference in one arm of the conditional.
  - $a := v$
  - ***if*** $b \le 0$ ***then*** $v := c + b$ ***else*** $v := b * 2$ ***fi***
  - ***while*** $d \ge 0$ ***do*** $d := d \div 2 - c$ ***od***

j	*k*	*Change j*	*Vars k*	*j Disjoint with k*
1	2	a	$b\ c\ v$	yes
1	3	a	$c\ d$	yes
2	1	$b\ v$	$a\ v$	no
2	3	$b\ v$	$c\ d$	yes
3	1	d	$a\ v$	yes
3	2	d	$b\ c\ v$	yes

Conclusion: Thread 2 interferes with thread 3; the other combinations are disjoint

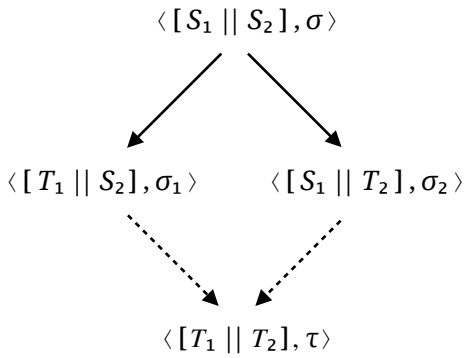
- ***Disjointedness Test Can Overestimate Amount of Interference:*** The disjointedness test is a static (compile-time) that aims for safety over accuracy when it comes to looking for interference. Not all the variables in *change (…)* and *vars (…)* are necessarily used at runtime. The tests for ***if*** B ***then*** S_1 ***else*** S_2 ***fi*** use the union of the variables for S_1 and for S_2, so a variable that is appears only in one branch of the ***if-fi*** is counted regardless of B or the runtime state.
- Passing a disjointedness test of thread *j* against thread *k* guarantees that interference cannot happen, no matter what the starting state is, and no matter what execution path gets taken.
- Failing a disjointedness test simply says we can't guarantee that thread *j* interferes with thread *k*. Without knowing more about the threads and the starting state, we can't say anything about whether interference in fact doesn't occur, or occurs only with some start states, or only along some execution paths. Failing a test certainly does not guarantee that interference is inevitable at runtime.

D. The Diamond Property; Confluence

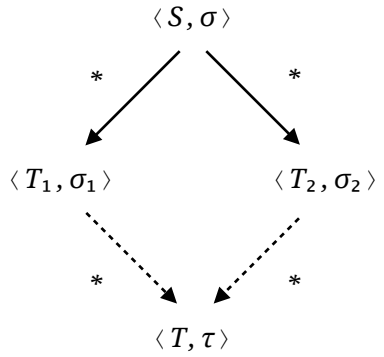
- The parallelism in DPPs is innocuous because different threads don't interfere with each other's execution: If one thread modifies a variable, that modification can’t be overwritten by any other thread. Also, since the modified variable can’t even be inspected by other threads, we know the modification won’t affect how the other threads execute. This “disjointedness” causes all the evaluation paths to end in the same configuration.
- In general, with $[\, S_1 \,||\, S_2 \,]$, we can execute S_1 or S_2 for one step. In an evaluation graph, the current evaluation path splits into two paths. With parallel programs in general, there might be

no way for those two paths to eventually merge back together into one path, but DPP's are different.

- Let $[\, S_1 \,||\, S_2 \,]$ be a DPP. If $\langle S_1, \sigma \rangle \rightarrow \langle T_1, \sigma_1 \rangle$ and $\langle S_2, \sigma \rangle \rightarrow \langle T_2, \sigma_2 \rangle$ then there is a state τ such that $\langle [\, T_1 \,||\, S_2 \,], \sigma_1 \rangle$ and $\langle [\, S_1 \,||\, T_2 \,], \sigma_2 \rangle$ both $\rightarrow \langle [T_1 \,||\, T_2 \,], \tau \rangle$. (Note: the same τ.)
- This is called the ***diamond property*** because people often draw it as in the diagram shown below. The claim is that if the solid arrows exist then the dashed arrows will exist.

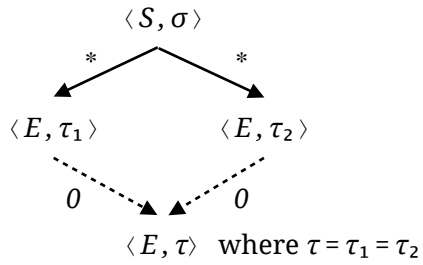


- The diamond property holds because the threads are disjoint so that it doesn't matter which thread you execute first: Any change in state caused by S_1 will be the same whether or not you execute part of S_2 (and vice-versa).
- The diamond property is actually stronger than what we discussed earlier, where an execution path splits and then eventually can merge back together. This weaker property is called confluence (or Church-Rosser, after two investigators of the lambda calculus), where the one-step arrows are replaced by zero-or-more-step arrows ($\rightarrow$ becomes $\rightarrow^*$). The diamond property is stronger because it implies confluence, but the converse is not true.



- Basically, a computation system in general (not just parallel programs) is confluent if execution doesn't have side effects. Everyday arithmetic expressions are confluent; C expressions with assignment operators are not.
- Because execution of disjoint parallel programs is confluent, if execution terminates, it terminates in a unique state.

- ***Theorem*** (***Unique Result of Disjoint Parallel Program***): If S is a disjoint parallel program then either $M(S, \sigma) = \{\tau\}$ (for some $\tau \in \Sigma$), $\{\perp_d\}$, or $\{\perp_e\}$.
- ***Proof***:If $\langle S, \sigma \rangle \rightarrow^* \langle E, \tau_1 \rangle$ and $\langle S, \sigma \rangle \rightarrow^* \langle E, \tau_2 \rangle$, then by confluence, there exists some common $\langle S', \tau \rangle$that both $\langle E, \tau_1 \rangle$and $\langle E, \tau_2 \rangle$ can $\rightarrow^*$ to. Since no semantics rule take $\langle E, ... \rangle \rightarrow$ anything, the $\rightarrow^*$ relations must both involve zero steps, so S' is E and $\tau = \tau_1 = \tau_2$.



E. Sequentialization Proof Rule for Disjoint Parallel Programs

- We'll have three rules for proving disjoint parallel programs correct: a sequential rule and two parallel rules. The sequential rule is powerful but burdensome.
- ***Definition***: The ***sequentialization*** of the parallel statement $[S_1 \,||\, ... \,||\, S_n]$ is the sequence $S_1; ...; S_n$. The ***sequentialized execution*** of the parallel statement is the execution of its sequentialization: We evaluate S_1 completely ,then S_2 completely, and so on.
- Since it doesn't matter how we interleave evaluation of pairwise disjoint parallel threads, their total effect will be the same as if we had evaluated them sequentially.

Sequentialization Rule

- If the sequential threads $S_1, ..., S_n$ are pairwise disjoint, then
  1. $\{p\} S_1; ...; S_n \{q\}$
  2. $\{p\} [S_1 \,||\, ... \,||\, S_n] \{q\}$ Sequentialization, 1
- ***Example 4:*** First, prove $\{T\} a := x + 1; b := x + 2 \{a + 1 = b\}$:

  $$\{T\} a := x + 1 \{a = x + 1\}; b := x + 2 \{a = x + 1 \wedge b = x + 2\} \{a + 1 = b\}$$

  - From the sequentialization rule for disjoint parallel programs, it follows that

    $$\{T\} [a := x + 1 \,||\, b := x + 2] \{a + 1 = b\}$$
- ***Example 5***: From $\{x = y\} \{x + 1 = y + 1\} x := x + 1; \{x = y + 1\} y := y + 1 \{x = y\}$
  - We can prove $\{x = y\} x := x + 1; y := y + 1 \{x = y\}$
  - So by the sequentialization rule for disjoint parallel programs,

    $$\{x = y\} [x := x + 1 \,||\, y := y + 1] \{x = y\}$$
- Since the order of evaluation the threads doesn't matter for a DPP, we can actually shuffle the order of the threads in the sequentialized program. E.g., since $\{p\} [S_1 \,||\, S_2] \{q\}$ and $\{p\} [S_2 \,||\, S_1] \{q\}$ produce the same final state, so do $\{p\} S_1; S_2 \{q\}$ and $\{p\} S_2; S_1 \{q\}$.

- ***Example 6***: As a concrete example of reordering, take Example 4:
  - $\{ T \}\ a := x + 1\ ;\ b := x + 2\ \{ a + 1 = b \}$ [before reordering]
  - $\{ T \}\ b := x + 2\ ;\ a := x + 1\ \{ a + 1 = b \}$ [after reordering]

Disjoint Conditions

CS 536: Science of Programming, Spring 2023

A. Why?

- Combining arbitrary threads are in parallel can yield programs that have surprisingly different results from how each thread works when run sequentially.
- When threads don't interfere with each other's work, we can combine them into a parallel program without worrying about these strange and unexpected kinds of behavior.
- Simply having disjoint parallel programs for threads isn't sufficient to avoid interference problems.
- What's needed are disjoint conditions, which ensure that no thread can interfere with the conditions of another thread.

B. Objectives

After this class, you should know

- What disjoint conditions are, why we need them, and how to recognize them.
- What the disjoint parallelism rule for disjoint parallel programs with disjoint conditions allows.
- That maybe not all apparent interference actually causes problems.

C. Parallelism Rule for Parallel Programs?

- The ***sequentialization proof rule*** for DPPs lets us reason about DPPs, which is nice, but it has two major flaws.
- First, It requires disjoint parallel programs, and we know disjoint parallelism is a strong constraint on what programs we can write.
- Second, to use it sequentialization, we need many intermediate conditions. To prove $\{p\}[S_1 \,||\, \ldots \,||\, S_n]\{q\}$, we need to prove $\{p\}S_1;\ldots;S_n\{q\}$, which (if we use wp) means finding a sequence of preconditions $q_1, \ldots, q_n$ and proving $\{p\}\{q_n\}S_1;\{q_{n-1}\}S_2;\{q_{n-2}\}\ldots\{q_1\}S_n\{q\}$. The proofs of $q_1, q_2, \ldots q_n$ can get increasingly complicated because each q_i can depend on all the threads and conditions to its right.
- ***Parallelism Rule***: Ideally, we'd like to take advantage not only of parallel execution of programs but also parallel writing of programs. Can we have n programmers write n sequential threads and then combine them into a parallel program? This behavior describes a ***parallelism rule***. For two threads, the form would be as follows (it generalizes to n threads).

1. $\{ p_1 \} S_1 \{ q_1 \}$
2. $\{ p_2 \} S_2 \{ q_2 \}$
3. $\{ p_1 \wedge p_2 \} [S_1 \,||\, S_2] \{ q_1 \wedge q_2 \}$ by (some kind of) parallelism 1, 2

In proof outline format, we write this as $\{ p_1 \wedge p_2 \} [\{ p_1 \} S_1 \{ q_1 \} \,||\, \{ p_2 \} S_2 \{ q_2 \}] \{ q_1 \wedge q_2 \}$.

- ***Example 1***: In proof format, we write
  1. $\{ x \geq 0 \} z := x \{ z \geq 0 \}$
  2. $\{ y \leq 0 \} w := -y \{ w \geq 0 \}$
  3. $\{ x \geq 0 \wedge y \leq 0 \} [z := x \,||\, w := -y] \{ z \geq 0 \wedge w \geq 0 \}$ by ??? parallelism 1, 2

  The proof outline form is

  $\{ x \geq 0 \wedge y \leq 0 \}$
  $[\{ x \geq 0 \} z := x \{ z \geq 0 \}$
  $\,||\, \{ y \leq 0 \} w := -y \{ w \geq 0 \}$
  $] \{ z \geq 0 \wedge w \geq 0 \}$
- It's nice when this pattern works, but we'll see examples where the pattern produces incorrect programs. To get correctness, we have to impose side conditions that limit how we can combine threads. Altogether, we'll look at a couple of different kinds of side conditions, which work in different situations.

- ***Example 2***: Here are some outlines where using a parallelism rule works. Outlines *(b)* and *(c)* use postcondition weakening to get a final postcondition.
  a. $\{ x \geq 0 \wedge y \leq 0 \} [\{ x \geq 0 \} z := x \{ z \geq 0 \} \,||\, \{ y \leq 0 \} w := -y \{ w \geq 0 \}] \{ z \geq 0 \wedge w \geq 0 \}$
  b. $\{ z = 0 \} [\{ z = 0 \} x := z - 1 \ \{ x \leq z = 0 \} \,||\, \{ z = 0 \} y := z \{ y = z = 0 \}] \{ x \leq z = 0 \wedge y = z = 0 \}$
     $\{ x \leq y = z = 0 \}$
  c. $\{ T \} [\{ T \} a := x + 1 \{ a = x + 1 \} \,||\, \{ T \} b := x + 2 \{ b = x + 2 \}] \{ a = x + 1 \wedge b = x + 2 \} \{ a + 1 = b \}$
  d. $\{ x = y = z = c \}$
     $[\{ x = c \} x := x^2 \{ x = c^2 \}$
     $\,||\, \{ y = c \} y := y^2 \{ y = c^2 \}$
     $\,||\, \{ z = c \} z := (z - d) * (z + d) \{ z = c^2 - d^2 \}$
     $]$
     $\{ x = c^2 \wedge y = c^2 \wedge z = c^2 - d^2 \}$
     $\{ x = y = z + d^2 \}$
  e.
- ***Example 3:*** Here are some outlines where using a parallelism rule fails. Outline *(a)* is incorrect because it wants to end with $x = 1 \wedge x = y = 0$ Outline *(b)* doesn't always leave $x > y$ (e.g., if we start with $x = y = 2$). Outline *(c)* never works.
  a. $\{ x = 0 \} [\{ x = 0 \} x := 1 \{ x = 1 \} \,||\, \{ x = 0 \} y := 0 \{ x = y = 0 \}] \{ x = 1 \wedge x = y = 0 \}$

b. $\{ x \geq y \wedge y = z \} [\{ x \geq y \} x := x + 1 \{ x > y \} \,||\, \{ y = z \} y := y * 2 ; z := z * 2 \{ y = z \}] \{ x > y \wedge y = z \}$

c. $\{ T \} [\{ T \} [x := 2 + 2 \{ x = 2 + 2 \} \,||\, \{ T \} x := 5 \{ x = 5 \}] \{ x = 2 + 2 \wedge x = 5 \} \{ 2 + 2 = 5 \}$

D. Disjoint Conditions

- Why do the proof outlines in ***Example 3*** fail? Since outline *(c)* ***should*** fail because the semantics don't support it. But outlines *(a)* and *(b)* fail even though they have disjoint parallel programs.
- Looking more closely, we see outline *(a)* fails because thread 1 sets x to *1*, which invalidates the precondition $x = 0$ needed by thread 2. Similarly, outline *(b)* fails because thread 2 modifies y, which might invalidate the $x \geq y$ and $x > y$ pre- and postconditions in thread1.
- It's not enough for outlines *(a)* and *(b)* to have disjoint parallel programs. I.e., it's not enough to guarantee that threads don't change each others' states. For correctness, we need each thread to ***not modify*** the variables that appear in the conditions of other threads.
- ***Definition***: $Free(p, ..., q)$ is the set of variables that appear free in any of the predicates p, ..., q. (Recall that a variable can have both free and bound occurrences, "bound" meaning "in the scope of a quantifier" for it. If a variable has at least one free occurrence in a predicate, then it is free in that predicate, regardless of whether or not there are bound occurrences.)
- ***Definition***: With $\{ p_1 \} S_1 \{ q_1 \}$ and $\{ p_2 \} S_2 \{ q_2 \}$, the first triple ***interferes with the conditions*** of the second triple if $Change(S_1) \cap Free(p_2, q_2) \neq \emptyset$. We may use ***disjoint from the conditions*** as a synonym for "not interfering with." The two triples have ***disjoint conditions*** if neither interferes with the conditions of the other. (I.e., $Change(S_1) \cap Free(p_2, q_2) = \emptyset$ and $Change(S_2) \cap Free(p_1, q_1) = \emptyset$.)
- ***Definition***: A parallel program outline $\{ p \} [\{ p_1 \} S_1 \{ q_1 \} \,||\, ... \,||\, \{ p_n \} S_n \{ q_n \}] \{ q \}$ has ***disjoint conditions*** if its threads have pairwise disjoint conditions. I.e., threads i and j have disjoint conditions, for all indexes i and j (with $i \neq j$).

- ***Example 4:*** All of the outlines in ***Example 2*** have pairwise disjoint programs and conditions. To show this, we can add a Disjoint Conditions column to the table we used for checking for disjoint programs. (Note that not all the sequential thread outlines are full outlines.) [Note: In the table below, I don't think it makes any difference if you write (e.g.) *w y* or *w, y* or $\{ w, y \}$.]

a. $\{ x \geq 0 \wedge y \leq 0 \} [\{ x \geq 0 \} z := x \{ z \geq 0 \} \,||\, \{ y \leq 0 \} w := -y \{ w \geq 0 \}] \{ z \geq 0 \wedge w \geq 0 \}$

i	***j***	***Change i***	***Vars j***	***Free j***	***Disj Pgm***	***Disj Cond***
1	2	*z*	*w y*	*w y*	*Y*	*Y*
2	1	*w*	*x z*	*x z*	*Y*	*Y*

b. $\{ z = 0 \} [\{ z = 0 \} x := z + 1 \ \{ x \le z = 0 \} \ || \ \{ z = 0 \} y := z \{ y = z = 0 \}] \{ x \le z = 0 \wedge y = z = 0 \}$
$\{ x \le y = z = 0 \}$

i	*j*	*Change i*	*Vars j*	*Free j*	*Disj Pgm*	*Disj Cond*
1	2	x	$y\ z$	$y\ z$	*Y*	*Y*
2	1	y	$x\ z$	$x\ z$	*Y*	*Y*

c. $\{ T \} [\{ T \} a := x + 1 \{ a = x + 1 \} \ || \ \{ T \} b := x + 2 \{ b = x + 2 \}] \{ a = x + 1 \wedge b = x + 2 \} \{ a + 1 = b \}$

i	*j*	*Change i*	*Vars j*	*Free j*	*Disj Pgm*	*Disj Cond*
1	2	a	$b\ x$	$b\ x$	*Y*	*Y*
2	1	b	$a\ x$	$a\ x$	*Y*	*Y*

d. $\{ x = y = z = c \}$
$[\{ x = c \} x := x^2 \{ x = c^2 \}$
$|| \{ y = c \} y := y^2 \{ y = c^2 \}$
$|| \{ z = c \} z := (z - d) * (z + d) \{ z = c^2 - d^2 \}$
$] \{ x = c^2 \wedge y = c^2 \wedge z = c^2 - d^2 \}$
$\{ x = y = z + d^2 \}$

i	*j*	*Change i*	*Vars j*	*Free j*	*Disj Pgm*	*Disj Cond*
1	2	x	y	$c\ y$	*Y*	*Y*
1	3	x	$d\ z$	$c\ d\ z$	*Y*	*Y*
2	1	y	x	$c\ x$	*Y*	*Y*
2	3	y	$d\ z$	$c\ d\ z$	*Y*	*Y*
3	1	z	x	$c\ x$	*Y*	*Y*
3	2	z	y	$c\ y$	*Y*	*Y*

- ***Example 5:*** All of the outlines in ***Example 4*** lack pairwise disjoint conditions. Outlines *(a)* and *(b)* have disjoint programs but outline *(c)* does not.

a. $\{ x = 0 \} [\{ x = 0 \} x := 1 \{ x = 1 \} \,||\, \{ x = 0 \} y := 0 \{ x = y \}] \{ x = 1 \wedge x = y \}$

i	*j*	*Change i*	*Vars j*	*Free j*	*Disj Pgm*	*Disj Cond*
1	2	x	y	$x\ y$	*Y*	*N*
2	1	y	x	x	*Y*	*Y*

b. $\{ x \le y \wedge y = z \} [\{ x \le y \} x := x - 1 \{ x < y \} \,||\, \{ y = z \} y := y * 2 ; z := z * 2 \{ y = z \}] \{ x < y \wedge y = z \}$

i	*j*	*Change i*	*Vars j*	*Free j*	*Disj Pgm*	*Disj Cond*
1	2	x	$y\ z$	$y\ z$	*Y*	*Y*
2	1	$y\ z$	x	$x\ y$	*Y*	*N*

c. $\{ T \} [\{ T \} x := 2 + 2 \{ x = 2 + 2 \} \,||\, \{ T \} x := 5 \{ x = 5 \}] \{ x = 2 + 2 \wedge x = 5 \} \{ 2 + 2 = 5 \}$

i	*j*	*Change i*	*Vars j*	*Free j*	*Disj Pgm*	*Disj Cond*
1	2	x	x	x	*N*	*N*
2	1	x	x	x	*N*	*N*

E. Disjoint Conditions; Disjoint Parallelism Rule

- If two outlines have disjoint programs and conditions, then their evaluations can be arbitrarily interleaved without fear. Having disjoint programs guarantees that no thread modifies the parts of the state used by the calculations of other threads. Having disjoint conditions guarantees that no thread modifies the parts of the state used to determine satisfaction of other threads' conditions.
- Because of this lack of runtime interference, we can use a a parallelism rule for disjoint parallel programs with disjoint conditions. The rule has the form of a parallelism rule (the parallel program uses the conjunctions of its threads' preconditions and postconditions as its precondition and postcondition). Having disjoint programs and conditions is the side condition that guarantees correctness of the rule.

DisjointParallelism Rule

$1.\quad \{p_1\}\, S_1\, \{q_1\}$
$2.\quad \{p_2\}\, S_2\, \{q_2\}$
...
$n\quad \{p_n\}\, S_n\, \{q_n\}$
$n+1\ \{p_1 \wedge p_2 \wedge \ldots \wedge p_n\}\, [S_1\ ||\ \ldots\ ||\ S_n]\, \{q_1 \wedge q_2 \wedge \ldots \wedge q_n\}$ Disjoint Parallelism, $1, 2, \ldots, n$

where threads $1, 2, \ldots, n$ are pairwise disjoint programs with pairwise disjoint conditions

- A reminder: The pairwise tests check thread i for interference with thread j, for all $i, j \in \{1, 2, \ldots, n\}$ where $i \neq j$.

Example 6: The outlines from Example 2 all have disjoint programs and conditions, so all of them can be proved using the disjoint parallelism rule. The presentations in Example 2 illustrate the uses of the rule. Just to be different, we can write them in nonlinear form. The comments marked * indicate parts of outline that are partial but not full. E.g., $\{y \leq 0\}\, w := -y\, \{w \geq 0\}$ is not a full outline; $\{y \leq 0\}\, w := -y\, \{y \leq 0 \wedge w = -y\}\, \{w \geq 0\}$ is; so is $\{y \leq 0\}\, \{-y \geq 0\}\, w := -y\, \{w \geq 0\}$.

a. $\{x \geq 0 \wedge y \leq 0\}$
$[\{x \geq 0\}\, z := x\, \{z \geq 0\}$
$||\ \{y \leq 0\}\, w := -y\, \{w \geq 0\}]$ (*) (Not a full outline for the sequential thread)
$\{z \geq 0 \wedge w \geq 0\}$

b. $\{z = 0\}$ //I.e., $z = 0 \wedge z = 0$
$[\{z = 0\}\, x := z - 1\ \{x \leq z = 0\}$ (*)
$||\ \{z = 0\}\, y := z\, \{y = z = 0\}]$
$\{x \leq z = 0 \wedge y = z = 0\}$
$\{x \leq y = z = 0\}$

c. $\{T\}$
$[\{T\}\, a := x + 1\, \{a = x + 1\}$
$||\ \{T\}\, b := x + 2\, \{b = x + 2\}]$
$\{a = x + 1 \wedge b = x + 2\}$
$\{a + 1 = b\}$

d. $\{ x = y = z = c \}$
$\{ x = c \wedge y = c \wedge z = c \}$
$[\{ x = c \} x := x^2 \{ x = c^2 \}$ (*)
$|| \{ y = c \} y := y^2 \{ y = c^2 \}$ (*)
$|| \{ z = c \} z := (z - d) * (z + d) \{ z = c^2 - d^2 \}$ (*)
$]$
$\{ x = c^2 \wedge y = c^2 \wedge z = c^2 - d^2 \}$
$\{ x = y = z + d^2 \}$

F. Removing Interference of Conditions

- Sometimes, interference with conditions can be removed by adding logical variables.

Example 7: All of the outlines from Example 3 have interfering conditions, but we can remove the interference for outlines *(a)* and *(b)*. Since *(c)* doesn't have disjoint programs, modifying its conditions won't help.

a. Original: $\{ x = 0 \} [\{ x = 0 \} x := 1 \{ x = 1 \} || \{ x = 0 \} y := 0 \{ x = y \}] \{ x = 1 \wedge x = y \}$.
The interference is that thread 1 setting x to 1 interferes with the x in $x = y$ in thread2. Said the other way, the x in $x = y$ refers to the value of x before $x := 1$. Introducing a logical variable for the initial value of x takes care of that. We also have to change the final post-condition to get one that's true.

Without interference: (note the outline for thread 2 is not full).
$\{ x = x_0 \wedge x = 0 \}$
$[\{ x = x_0 \wedge x = 0 \} x := 1 \{ x = 1 \wedge x_0 = 0 \}$
$|| \{ x = x_0 \wedge x = 0 \} y := 0 \{ x_0 = y \}]$
$\{ x = 1 \wedge x - 1 = y \}$

b. Original:
$\{ x \le y \wedge y = z \} [\{ x \le y \} x := x - 1 \{ x < y \} || \{ y = z \} y := y * 2 ; z := z * 2 \{ y = z \}] \{ x < y \wedge y = z \}$.
The interference is $y := y * 2$ in thread 2 versus $x \le y$ and $x < y$ in thread 1. Again, introducing logical variables helps.

Without interference:

$\{ x = x_0 \wedge y = y_0 \wedge z = z_0 \wedge x \le y \wedge y = z \}$
$[\{ x = x_0 \wedge x \le y_0 \}$
$x := x - 1$
$\{ x_0 \le y_0 \wedge x = x_0 - 1 \}$
$\| \{ y = y_0 \wedge z = z_0 \wedge y = z \}$
$y := y * 2 ; z := z * 2$
$\{ y_0 = z_0 \wedge y = y_0 * 2 \wedge z = z_0 * 2 \}$
$]$
$\{ (x_0 \le y_0 \wedge x = x_0 - 1) \wedge (y_0 = z_0 \wedge y = y_0 * 2 \wedge z = z_0 * 2) \}$
$\{ x < y \div 2 \wedge y = z \}$

- For another example of removing interference, let's look at a modified version of Example 6.d where we have the initial precondition $x = y = z = c$ but no annotation for the sequential threads. In that case, it would be natural to use $x = y = z = c$ as the precondition for all three threads:

  $\{ x = y = z = c \}$ (partial outline)
  $[\{ x = y = z = c \} x := x^2 \{ x = c^2 \wedge y = z = c \}$
  $\| \{ x = y = z = c \} y := y^2 \{ y = c^2 \wedge x = z = c \}$
  $\| \{ x = y = z = c \} z := (z - d) * (z + d) \{ z = c^2 - d^2 \wedge x = y = c \}$
  $]$
  $\{ x = c^2 \wedge y = z = c \wedge y = c^2 \wedge x = z = c \wedge z = c^2 - d^2 \wedge x = y = c \}$
  $\{ x = y = z + d^2 \}$
- But this has interference all over the place. E.g., $x := x^2$ interferes with all the conditions $x = \ldots$ in the other threads, and similarly for the assignments to y and z.
- A possible fix that works very well for this example is to look at the sequential annotations and drop everything that isn't directly related to the sequential thread's programs. E.g., with $\{ x = y = z = c \} x := x^2 \{ x = c^2 \wedge y = z = c \}$, we need x and c but not y or z. Removing y and z gives us $\{ x = c \} x := x^2 \{ x = c^2 \}$. Similarly, we can remove x and z from the thread for $y := y^2$ and then x and y from the thread for $z := (z - d) * (z + d)$. The result is exactly Example 6.d:

  $\{ x = y = z = c \}$ (partial outline)
  $[\{ x = c \} x := x^2 \{ x = c^2 \}$
  $\| \{ y = c \} y := y^2 \{ y = c^2 \}$
  $\| \{ z = c \} z := (z - d) * (z + d) \{ z = c^2 - d^2 \}$
  $]$
  $\{ x = c^2 \wedge y = c^2 \wedge z = c^2 - d^2 \}$
  $\{ x = y = z + d^2 \}$
- Another analysis that adds x_0, y_0, and z_0 turns $\{ x = y = z = c \} x := x^2 \{ x = c^2 \wedge y = z = c \}$ into $\{ x = x_0 \wedge x = c \} x := x^2 \{ x_0 = c \wedge x = x_0^2 \}$ and similarly for y and z:

$\{ x = y = z = c \}$ (partial outline)
$[\{ x = x_0 \wedge x = c \}\ x := x^2\ \{ x_0 = c \wedge x = x_0{}^2 \}$
$\|\ \{ y = y_0 \wedge y = c \}\ y := y^2\ \{ y_0 = c \wedge y = x_0{}^2 \}$
$\|\ \{ z = z_0 \wedge z = c \}\ z := (z - d) * (z + d)\ \{ z_0 = c \wedge z = z_0{}^2 - d^2 \}$
$]$
$\{ x_0 = c \wedge x = x_0{}^2 \wedge y_0 = c \wedge y = x_0{}^2 \wedge z_0 = c \wedge z = z_0{}^2 - d^2 \}$
$\{ x = c^2 \wedge y = c^2 \wedge z = c^2 - d^2 \}$
$\{ x = y = z + d^2 \}$

G. Not All Changes are Interference

- We'll look into this in more detail next class, but can we loosen the requirement of disjoint conditions? (After all, requiring disjoint threads already disallows too many programs we want to write.)
- The basic observation is that often we can change the values of variables without changing whether a condition is satisfied or not. E.g., the condition $x > 0$ isn't invalidated by setting $x := 1$ or $x := x + 1$ (or $x := x$, for that matter).
- Say a thread includes a condition p (either as a precondition or postcondition; it doesn't matter), and a different thread is about to execute $\{ q_1 \}\ x := e\ \{ q_2 \}$. Surprisingly, q_2 isn't important; what is important is that executing $x := e$ only causes harm if it makes p false. More precisely, if p and q_1 hold and we execute $x := e$, we don't have interference if p is still satisfied after the assignment.
- As long as $\{ p \wedge q_1 \}\ x := e\ \{ p \}$ is valid, we know that whatever change $x := e$ causes, it won't disturb execution of the other thread, at least as far as p holding.
- More generally, with p and $\{ q_1 \}\ S_1\ \{ \ldots \}$ (we don't care about the postcondition of S_1), we're free of interference if $\{ p \wedge q_1 \}\ S_1\ \{ p \}$ is valid.
- We can build on this observation to get a notion of interference-freedom that works as a side condition for a nicer parallelism rule. It's not trivial because a thread has multiple statements that we have to check against multiple conditions in the other thread. But we can reduce the burden by looking more carefully at exactly what statements we need to check for interference.
- E.g., $x_1 := e_1 ; x_2 := e_2 ; x_3 := e_3$ comprises 5 statements (three assignments and two sequences). We'll have the check the assignments, of course, but do we need to check the two sequences? As it turns out, no, we don't.

Shared Variables and Interference-Freedom

CS 536: Science of Programming, Spring 2023

2023-04-21: p.7

A. Why

- Parallel programs can coordinate their work using shared variables, but it's important for threads to not interfere (to not invalidate conditions that the other thread relies on).

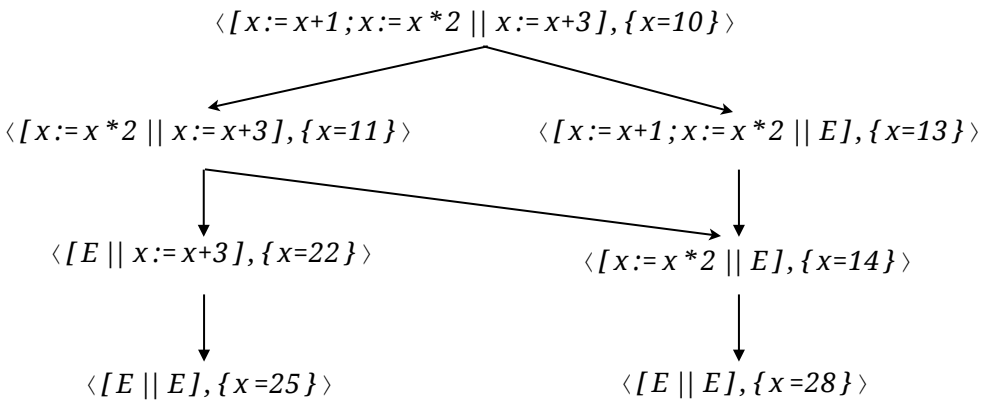
B. Objectives

At the end of this class you should know how to

- Check for interference between the correctness proofs of the sequential threads of a shared memory parallel program.

C. Parallel Programs with Shared Variables

- Disjoint parallel programs are nice because no thread interferes with another's work. They're bad because threads can't communicate or combine efforts.
- Let's start looking at programs that aren't disjoint parallel and allow threads to share variables. We've seen examples, but here's another one.
- ***Example 1***: Below is the evaluation graph for $\langle [\, x := x+1 \,;\, x := x*2 \;||\; x := x+3 \,], \{ x=10 \} \rangle$. Since $11+1+3 = 11+3+1$, two of the intermediate states are equal.



- The problem with shared variables is that threads that work correctly individually might stop working when you combine them in parallel.
- Depending on the execution path it takes, some piece of code in one thread may invalidate a condition needed by a second thread.

- ***Race condition***: A situation where correctness of a parallel program depends on the relative speeds of execution of the threads. (If different relative speeds produce different results but the results are correct, then we don't have a race condition.) To avoid race conditions
  - We control where interleaving can occur by using ***atomic regions***.
  - We ensure that when interleaving occurs, it causes no harm (threads are ***interference-free***).

D. Critical Sections

- The basic ***critical section*** problem involves two threads, each with an identified subset of code (the ***critical section***), where we must avoid both threads executing code in their critical sections simultaneously. Or said another way, if one thread is executing code in its critical section, then we must keep the other thread from entering its critical section. Race conditions caused by interleaving two pieces of code is a form of the critical section problem.
- Critical sections don't only involve what state changes pieces of code do, it also depends on the form of the code. For example, we normally don't think of there being a difference between $x := y+y$ and $x := y\,;\ x := x+y$; they both set x to $2\,y$. But if their opportunities for interleaving are different, then these two pieces of code can behave very differently.
  - For us, $x := y+y$ cannot be interleaved but $x := y\,;\ x := x+y$ can be interleaved with at the semicolon (i.e., between statements). There are three possible interleavings:
    - $\{y = \alpha\}\ x := y+y\ ;\{x = 2\,\alpha\}\ x := y\ ;\{x=\alpha\}\ x := x+y\ \{x = 2\,\alpha\}$
    - $\{y = \alpha\}\ x := y\ ;\ \{x=\alpha\}\ x := x+y\ ;\{x = 2\,\alpha\}\ x := y+y\ \{x = 2\,\alpha\}$
    - $\{y = \alpha\}\ x := y\ ;\ \{x=\alpha\}\ x := y+y\ ;\ \{x = 2\,\alpha\}\ x := x+y\ \{x = 3\,\alpha\}$
  - The third interleaving involves a race condition because running $x := y+y$ by overwriting x after the other thread sets $x := y$ but before that thread gets to use x in $x := x+y$.
- Historically, the critical section problem was very difficult to solve using software alone. (Numerous proposed solutions were in fact wrong.) Eventually, the problem was solved by adding new hardware instructions ("test and set").

E. Atomic Regions

- People control the amount of possible interleaving of execution by declaring pieces of code to be ***atomic***: Their execution cannot be interleaved with anything else.
- ***Definition (Syntax)***: If S is a statement, then $<S>$ is an ***atomic region*** statement with body S.
- ***Operational semantics of atomic regions:*** Evaluation of $<S>$ behaves like a single step:
  - If $\langle S, \sigma \rangle \rightarrow^* \langle E, \tau \rangle$ then $\langle <S>, \sigma \rangle \rightarrow \langle E, \tau \rangle$.
- Our operational semantics definition makes assignment statements atomic automatically, so making $v := e$ its own atomic section causes no change. However, embedding $v := e$ within a larger atomic region does make a difference.
- A ***normal assignment*** is one not inside an atomic area. We worry about interleaving of normal assignments; we don't worry about the non-normal ones.

- ***Example 2***: For example, since it can't be interleaved with $< x := y;\ \ x := x+y >$ has the same effect as $x := y+y$. Indeed, the evaluation graph for both of them involve a single$\rightarrow$ arrow:
  - $\langle x := y+y, \sigma \rangle \rightarrow \langle E, \sigma[x \mapsto 2\,\sigma(y)] \rangle$
  - $\langle < x := y;\ \ x := x+y >, \sigma \rangle \rightarrow \langle E, \sigma[x \mapsto 2\,\sigma(y)] \rangle$
    - This is because $\langle x := y;\ \ x := x+y, \sigma \rangle \rightarrow^2 \langle E, \sigma[x \mapsto 2\,\sigma(y)] \rangle$
- Using atomic regions gives us control over the size of pieces of code that can be interleaved ("granularity of code interleaving"). However, making more or larger atomic sections is no panacea: The more or larger atomic regions code has, the less interleaving and hence parallelism we have.

F. Interleaving with skip, if, and while statements

- There's no interleaving with a ***skip*** statement: ***skip*** executes atomically, so nothing can execute "in the middle of a" ***skip***. Since ***skip*** doesn't change the state, interleaving it between two statements of other threads causes no change.
- For ***if-else*** and ***while*** statements (and nondeterministic ***if-fi*** and ***do-od***), although evaluation of a boolean expression is atomic, interleaving can occur between inspection of the result and the jump to the next configuration.
  - For example, sequentially, if $\sigma(B) = T$, then $\langle$ ***if*** B ***then*** S_1 ***else*** S_2 ***fi***$, \sigma \rangle \rightarrow \langle S_1, \sigma \rangle$. Thus in parallel, another statement can be executed between the ***if*** test and the start of the true branch. If statement U changes σ to τ, then we can have
    $$\langle [\textbf{if } B \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi} \,||\, < U >], \sigma \rangle \rightarrow \langle [S_1 \,||\, < U >], \sigma \rangle \rightarrow \langle [S_1 \,||\, E], \tau \rangle$$
  - In this last configuration, we're about to execute S_1 not in σ, but in τ. We can't use annotations like $\{ p \}$ ***if*** B ***then*** $\{ p \wedge B \} S_1 \ldots$ ***fi*** because there's no guarantee that B is still true when the true branch begins executing.
  - Exactly the same problem occurs with ***while***: if $W \equiv$ ***while*** B ***do*** S ***od***, then
    $$\langle [\textbf{while } B \textbf{ do } S \textbf{ od} \,||\, < U >], \sigma \rangle \rightarrow \langle [S; W \,||\, < U >], \sigma \rangle \rightarrow \langle [S; W \,||\, E], \tau \rangle$$
    So B may be false when the loop body S starts executing.

G. Interference Between Threads

- ***Interference*** occurs when one thread invalidates a condition needed by another thread. In the previous class we had an example where $x := 1$ interfered with $\{ x=0 \}\, y := 0\, \{ x=y \}$ because we didn't have disjoint conditions. If the triple had been $\{ x \geq 0 \}\, y := 0\, \{ x \geq y = 0 \}$, then the conditions would still not be disjoint, but $x := 1$ would not have caused them to become invalid.
- ***Example 3***: Let $\{ x > 0 \}\, S_1\, \{ x \geq 0 \}$ and $\{ x > 0 \}\, S_2\, \{ x > 0 \}$ be two threads. If $x > 0$, then the preconditions for S_1 and S_2 hold, so we can run either one.
  - If S_1 runs before S_2, then S_1 terminates with $x \geq 0$, which interferes with the precondition $x > 0$ of S_2.

- If S_1 runs after S_2, then S_1 again takes $x > 0$ to $x \geq 0$, which interferes with the postcondition $x > 0$ of S_2.
- If S_2 runs before S_1, it terminates with $x > 0$, so it doesn't interfere with the precondition $x > 0$ or the postcondition $x \geq 0$ of S_1.
- To remove the interference caused by S_1, we might strengthen its postcondition from $x \geq 0$ to $x > 0$, or we might weaken the precondition of S_2 to $x \geq 0$. Another possibility is to make running S_1 atomically with the code that follows it (and presumably requires $x \geq 0$).
  - $\{ x > 0 \}\ S_1\ \{ x \geq 0 \}\ ;\ <U>\ \{ q \}$ could become $\{ x > 0 \}\ <S_1\ \{ x \geq 0 \}\ ;\ U>\ \{ q \}$, so we wouldn't be able to run S_2 between S_1 and $<U>$.

H. The Interference-Freedom Checks

- ***Definition***: The ***interference-freedom check*** for $\{ p \} < S > \{ \dots \}$ ***versus a predicate*** q is the predicate $\{ p \wedge q \} < S > \{ q \}$. If this check is valid, then we say that $\{ p \} < S > \{ \dots \}$ ***does not interfere with*** q. (Note we don't care what $< S >$ does if p holds but q does not: $\{ p \wedge \neg q \} < S > \{ \dots \}$.)
- ***Example 4***: $\{ p \}\ x := x+1\ \{ \dots \}$ does not interfere with $x \geq 0$, but it does interfere with $x < 0$.
- ***Example 5***: $\{ x \leq -1 \}\ x := x-1\ \{ \dots \}$ does not interfere with $x \leq 0$, but it does interfere with $x \geq -1$.
- ***Example 6***: $\{ x \,\%\, 4 = 2 \}\ x := x+4\ \{ \dots \}$ does not interfere with $even(x)$ (i.e., $x \,\%\, 2 = 0$).
- Note interference freedom of $\{ p \} < S > \{ \dots \}$ with q doesn't mean that S can't change the values of variables free in q, it means that S is restricted to changes that maintain satisfaction of q. Interference freedom is less restrictive than disjointedness of programs or conditions.

Failing an interference freedom check

- Proving that the interference freedom check for $\{ p \} < S > \{ \dots \}$ with q is valid tells us that we know interference cannot occur at runtime. That means failing to prove the check only tells us that we don't know that interference cannot occur at runtime. Said another way, the negation of "does not interfere with" is "possibly interferes with".
- Specifically, failing an interference freedom check does not guarantee that interference will occur at runtime. Interference occurs if running the program uses $\langle < S >, \tau_0 \rangle \rightarrow \langle E, \tau_1 \rangle$ for some $\tau_0 \vDash p \wedge q$ and some $\tau_1 \nvDash q$. If program execution along some path never involves $\langle < S >, \tau_0 \rangle \rightarrow \langle E, \tau_1 \rangle$, then interference doesn't occur*.

Checking for Interference-Freedom of larger structures

- Once we have a notion of interference freedom of an atomic triple versus a predicate, we can build up to a notion of interference between threads.
- ***Notation***: S^* is a proof outline of the program S.

---

*In actual execution, $< S >$ would be the next step of execution of a thread, so we would get something like $\langle [< S > ; S' \,||\, \dots], \tau_0 \rangle \rightarrow \langle [S' \,||\, \dots], \tau_1 \rangle$.

- ***Definition***: The atomic statement $\{p_1\} < S_1 > \{...\}$ ***does not interfere with*** the proof outline $\{p_2\} S_2^* \{q_2\}$ if it doesn't interfere with p_2, nor with q_2, nor with any precondition before an atomic statement in S_2^*. I.e., $\{p_1\} < S_1 > \{...\}$ does not interfere with any r where $\{r\} < ... > \{...\}$ appears in S_2^*.
- ***Definition***: A proof outline $\{p_1\} S_1^* \{q_1\}$ ***does not interfere with*** another proof outline $\{p_2\} S_2^* \{q_2\}$ if every atomic statement $\{r\} < S > \{...\}$ in S_1^* does not interfere with the outline $\{p_2\} S_2^* \{q_2\}$.
- It's sufficient to look at only the atomic statements in S_1^* because complex statements don't cause state changes until they get to atomic sub-statements. E.g., with $\{p_1\}$ ***while*** B ***do*** $\{p_2\} S \{p_3\}$ ***od*** $\{p_4\}$, when execution is at p_1 or p_3, the next execution step involves testing B and jumping to p_2 or p_4; this doesn't change the state. When execution is at p_3, execution might cause a state change, but only if S begins with (or is) an atomic statement. The situations with ***if*** statements and nondeterministic ***if*** and ***do*** statements are similar.
- ***Definition***: Two proof outlines $\{p_1\} S_1^* \{q_1\}$ and $\{p_2\} S_2^* \{q_2\}$ are ***interference-free*** if neither interferes with the other.
- ***Example 7:*** $\{x \bmod 4 = 0\}\ x := x+3\ \{...\}$ interferes with $\{even(x)\}\ x := x+1\ \{odd(x)\}$ because it interferes with $even(x)$: $\{x \bmod 4 = 0 \wedge even(x)\}\ x := x+3\ \{even(x)\}$ is not valid. However, the first triple doesn't interfere with $odd(x)$: $\{x \bmod 4 = 0 \wedge odd(x)\}\ x := x+3\ \{odd(x)\}$ is valid because the precondition implies false (and $\{F\} S \{q\}$ is valid for all S and q).
- ***Example 8***: Two different copies of $\{even(x)\}\ x := x+1\ \{odd(x)\}$ interfere with each other. I.e., copies of $\{even(x)\}\ x := x+1\ \{odd(x)\}$ in different threads will interfere with each other.
- ***Example 9***: $\{even(x)\}\ x := x+1\ \{odd(x)\}$ and $\{x \geq 0\}\ x := x+2\ \{x > 1\}$ are interference-free.
  - Precondition of triple 1: $\{x \geq 0 \wedge even(x)\}\ x := x+2\ \{even(x)\}$ is valid.
  - Postcondition of triple 1: $\{x \geq 0 \wedge odd(x)\}\ x := x+2\ \{odd(x)\}$ is valid.
  - Precondition of triple 2: $\{even(x) \wedge x \geq 0\}\ x := x+1\ \{x \geq 0\}$ is valid.
  - Postcondition of triple 2: $\{even(x) \wedge x > 1\}\ x := x+1\ \{x > 1\}$ is valid.

I. Parallelism with Shared Variables and Interference-Freedom

- ***Theorem (Interference-Freedom)***: Let $\{p_1\} S_1^* \{q_1\}$, $\{p_2\} S_2^* \{q_2\}$, ..., and $\{p_n\} S_n^* \{q_n\}$ be sequentially valid and pairwise interference-free. Then their parallel composition

$$\{p_1 \wedge p_2 \wedge ... \wedge p_n\}\ [S_1^* \,||\, ... \,||\, S_n^*]\ \{q_1 \wedge q_2 \wedge ... \wedge q_n\}$$

  is also valid. (Proof omitted.)
- The interference freedom theorem enables the use of a new parallelism rule:

Parallelism with Shared Variables Rule

1. $\{p_1\} S_1^* \{q_1\}$
2. $\{p_2\} S_2^* \{q_2\}$

...
n. $\{p_n\} S_n^* \{q_n\}$
n+1. $\{p_1 \wedge p_2 \wedge \ldots \wedge p_n\}$ Parallelism w/ Shared Vars, *1*, *2*, ..., *n*
$[S_1^* \,||\, \ldots \,||\, S_n^*]$
$\{q_1 \wedge q_2 \wedge \ldots \wedge q_n\}$

where the $\{p_k\} S_k^* \{q_k\}$ are pairwise interference-free.

- One feature of this rule is that it talks about proof outlines, not correctness triples. (Before this, the rules only concerned triples.) We can no longer compose correctness triples because we can't guarantee correctness without knowing that the different threads don't invalidate conditions inside the other threads.
- ***Example 12***: A proof outline for $\{x=0\}\,[\,x := x+2 \,||\, x := 0\,]\,\{x=0 \vee x=2\}$ using parallelism with shared variables is below:

  $\{x=0\}$
  $\{x=0 \wedge T\}$
  $[\,\{x=0\}\, x := x+2\, \{x=0 \vee x=2\}$
  $||\, \{T\}\, x := 0\, \{x=0 \vee x=2\}$
  $]$
  $\{(x=0 \vee x=2) \wedge (x=0 \vee x=2)\}$
  $\{x=0 \vee x=2\}$

- The side conditions are
  - $\{x=0\}\, x := x+2\, \{\ldots\}$ does not interfere with T or $x=0 \vee x=2$, which holds by
    - $\{x=0 \wedge T\}\, x := x+2\, \{T\}$
    - $\{x=0 \wedge (x=0 \vee x=2)\}\, x := x+2\, \{x=0 \vee x=2\}$
  - $\{T\}\, x := 0\, \{\ldots\}$ does not interfere with $x=0$ or $x=0 \vee x=2$
    - $\{T \wedge x=0\}\, x := 0\, \{x=0\}$
    - $\{T \wedge (x=0 \vee x=2)\}\, x := 0\, \{x=0 \vee x=2\}$
- No matter which assignment executes first, when $x := x+2$ runs, it sees $x=0$ and sets it to 2. When $x := 0$ runs, it sees $x=0$ or 2 and makes it 0.
- Sequentially, the disjunct $x=0$ is not needed in $\{x=0\}\, x := x+2\, \{x=0 \vee x=2\}$, nor is $x=2$ needed in $\{T\}\, x := 0\, \{x=0 \vee x=2\}$. To run the threads in parallel, however, we need to add these disjuncts to account for the interactions that parallel execution causes. (Or said the other way, we add these disjuncts to avoid interference in the final result.)

J. An Example With Shared and Auxiliary Variables[†]

- Recall the program $\{x=0\}\,[\,x := x+2 \,||\, x := 0\,]\,\{x=0 \vee x=2\}$, which we proved correct using parallelism with shared variables. Sequentially, we had $\{x=0\}\, x := x+2\, \{x=0 \vee x=2\}$ and $\{T\}\, x := 0\, \{x=0 \vee x=2\}$, and interference freedom allowed us to compose these threads in parallel.

[†] We'll look in detail at auxiliary variables in the next class.

- We can weaken the precondition $x=0$ to just true and the program still works, but it's annoyingly difficult to verify. If we try to annotate the program using

  $$\{ T \wedge x=x_0 \} [\{ T \} x := 0 \{ x=0 \} \,||\, \{ x=x_0 \} x := x+2 \{ x=x_0+2 \}] \{ \dots \}$$

  we find that each thread's assignment to x interferes with one or both conditions of the other thread.
- However, just because two proof outlines interfere, that doesn't mean the programs are wrong.[‡] It may just be that one or more of the proofs need to be modified in order to prove interference freedom.
- We could try adding the $x=0$ postcondition of thread 1 to thread 2. [2023-04-21]

  $\{ T \wedge x=x_0 \}$
  $[\{ T \} x := 0 \{ x=0 \}$
  $\,||\, \{ x=x_0 \} \{ x=0 \vee x=x_0 \} x := x+2 \{ x=2 \vee x=x_0+2 \}$
  $]$
  $\{ x=0 \wedge (x=2 \vee x=x_0+2) \}$

[2023-04-21] start new version

- If thread 1 runs first, its $x := 0$ interferes with thread 2's postcondition $x=2 \vee x=x_0+2$.
  - We can fix this by changing the thread 2 postcondition to $x=0 \vee x=2 \vee x=x_0+2$.
- If thread 2 runs first, it interferes with thread 1's $x=0$.
  - We could make the first thread $\{ T \} x := 0 \{ x=0 \vee x=2 \}$, which reflects the possibility that $x := 0$ runs and then $x := x+2$ runs.
  - But thread 2's $x := x+2$ interferes with $x=0 \vee x=2$. Adding $x = 4$ to get $x=0 \vee x=2 \vee x=4$ doesn't solve the problem because now $x := x+2$ interferes with that. Adding a disjunct $x=6$ leads to adding $x=8$, ad infinitum.
  - Even worse,, adding $x=4$ or 6 or 8 ... doesn't reflect the reality of what the program does: Either $x=x_0$ or 0 or x_0+2 or 2. It should never be 4 (unless $x_0 = 2$).

[2023-04-21] end new version

- The problem here is that we don't know which case ran first: If thread 1 runs first then we have $x=x_0$ and then $x=0$ and then thread 2 sets $x=2$. If thread 2 runs first then we have $x=x_0$ and then $x=x_0+2$ and then $x=0$. We can solve this problem by adding a new boolean variable *inc* that tells us whether or not the $x := x+2$ increment has been done.
- $\{ T \}\, inc := F ; [\, x := 0 \,||\, < x := x+2 ;\ inc := T >] \{ x=0 \vee x=2 \}$
- First, let's look at how adding *inc* lets us prove correctness, then we'll look at (and eliminate) the inefficiency caused by adding a new (what we'll eventually call an ***auxiliary***) variable.
  - The increment of x and setting of *inc* are done together atomically so we don't have to worry about $x := 0$ being done between them. Since *inc* will be removed from the program, there's no actual increase in granularity of atomicity.

[‡] That's especially true here, since the program in fact works correctly.

  - The annotation of the first thread, is key: $\{ T \}\ x := 0\ \{ x=0 \vee (inc \wedge x=2) \}$. The postcondition can't just be $x=0$, since that's interfered with by thread 2. If thread 2 runs, however, it sees $x=0$ and sets $x=2$ and inc to true, so we can make that a second disjunct of the postcondition.
  - For the second thread, $\{ \neg\ inc \}\ \ < x := x+2 ;\ \ inc := T > \{ inc \}$ is all we need. The important information about the value of x is held in the conditions of thread 1. We could add information to the conditions of the second thread, but it just makes for more complicated interference-freedom checks.
- Here is a full annotation for our program, with inc added

  $\{ T \}\ \ inc := F \{ T \wedge \neg\ inc \} ;$
  $[\{ T \}\ x := 0\ \{ x=0 \vee (inc \wedge x=2) \}$
  $||\ \{ \neg\ inc \} < x := x+2 ;\ \ inc := T > \{ inc \}$
  $]$
  $\{ (x=0 \vee (inc \wedge x=2)) \wedge inc \}$
  $\{ x=0 \vee x=2 \}$

- Adding inc lets us know whether or not the increment of x has occurred. There's a symmetric alternative, which is to use a variable that tells us whether or not $x := 0$ has executed; let's call it z (short for $zeroed$). To avoid interference, we need to make $z := T$ atomic with $x := 0$ so that thread2 can never observe $x=0 \wedge \neg\ z$.

  $\{ T \}\ z := F ; \{ \neg\ z \wedge (z \rightarrow x=0) \}$
  $[\{ \neg\ z \} < x := 0 ; z := T > \{ z \}$
  $||\ \{ z \rightarrow x=0 \}\ x := x+2\ \{ z \rightarrow x=0 \vee x=2 \}$
  $]$
  $\{ z \wedge (z \rightarrow x=0 \vee x=2) \}$
  $\{ x=0 \vee x=2 \}$

  Once again, we don't need to include all the variables in all the conditions. The postcondition of thread 1 doesn't need to mention x because all the relevant information is contained in the postcondition of thread 2. Sequential correctness of the threads is easy to verify, as is interference freedom: Thread 1's $< x := 0 ; z := T >$ doesn't interfere with $z \rightarrow x=0$ or with $z \rightarrow x=0 \vee x=2$, and thread 2 doesn't modify z, so it can't interfere with z.

Auxiliary Variables

- Whether we add inc or z, the resulting program works correctly, but we've also added to its computation, which doesn't seem efficient. On the other hand, assignments to x don't rely on inc or z: We don't have $x := \ldots$ expression involving inc or z $\ldots$, so for purposes of calculating x, the actual value of inc or z in memory isn't relevant. When we look at ***auxiliary variables*** in the next class, we'll see that the code involving inc or z can be removed without affecting the

overall correctness of the program. This will get us back to the program and annotation we wanted,

$$\{ T \} [x := 0 \,||\, x := x+2] \{ x=0 \vee x=2 \}$$

Auxiliary and Logical Variables

CS 536: Science of Programming, Spring 2023

2023-04-24 pp. 3, 6

A. Why

- Auxiliary variables help us reason about our programs without adding unnecessary computations.

B. Objectives

At the end of this class you should

- Recognize whether or not a set of variables is auxiliary for a program.
- Be able to add auxiliary variables to a program or remove auxiliary variables from a program, consistently.

C. Why Auxiliary Variables?

- We've used logical variables, which only appear in the correctness proof in:
  - The forward assignment rule to name the value a variable had before the assignment statement.
  - Program specifications to name the value a variable had when the program began.
- Since they only appear in proofs, we haven't been calculating the values of logical variables because it's clearly unnecessary to do so.
- Auxiliary variables are an extension of the notion of logical variables. Normally, we calculate the values of all of our program variables; with auxiliary variables, we won't.
- Auxiliary variables added to the program to enable a correctness proof but aren't relevant to the calculation of the values of variables we're actually interested in: Their actual values at runtime, however, don't affect the calculations that we're interested in. It's in that sense that auxiliary variables are unnecessary.
- To illustrate, consider forward assignment: $\{ p \wedge x = x_0 \}\ x := e\ \{ p[x_0 / x] \wedge x = e[x_0 / x] \}$.
  - Without introducing x_0, we're kind of stuck for how to describe forward assignment.
- Now consider $\{ p \}\ x_0 := x ; \{ p \wedge x = x_0 \}\ x := e\ \{ p[x_0 / x] \wedge x = e[x_0 / x] \}$
  - The assignment $x_0 := x$ sets our "logical" variable but doesn't affect the calculation of $x := e$.
  - We could calculate x_0 at runtime, but why bother if all we're interested in is x?
  - So we can argue that in some sense the assignment $x_0 := x$ doesn't really need to be executed because it doesn't affect $x := e$.
  - We've had an implicit quantifier over x_0 where the range of the quantifier is both conditions.

$$\{ \exists\, x_0 \,.\, p \wedge x = x_0 \}\; x := e \;\{ p\,[\,x_0 \,/\, x\,] \wedge x = e\,[\,x_0 \,/\, x\,] \}$$

- Here, x_0 doesn't change once we set it. Using auxiliary variables will let us change variables like x_0 as long as those changes don't affect the calculations we're interested in, so we'll still be able to avoid calculating their values.

Example 1:

- In the program below, we search through $x, f(x), f(f(x)), f(f(f(x)))$, ... for the first value that meets property $P(x)$. For termination, let's assume that in this sequence, the difference between adjacent values decreases: $|x - f(x)| > |f(x) - f(f(x))| > |f(f(x)) - f(f(f(x)))|$ ... ≥ 0.

```
x0 := x ;                          // Previous value of x
x := f(x) ;                        // New value of x
delta_x := x – x0 ;
{ inv ... } { bd | delta_x | }     // Absolute value of delta_x
while ¬ P(x) do
    ( ... computations that don't use x0 or delta_x ...)
    // Update old x, current x, and delta_x
        x0 := x ;
        x := f(x) ;
        delta_x := x – x0
od
// After the loop, we don't use delta_x or x0.
```

- If $delta\_x$ isn't used anywhere (except in the bound function), then calculating its actual value doesn't really serve any purpose. Similarly, if x_0 is used only to calculate $delta\_x$, then its value doesn't serve any useful purpose. We use x in $delta\_x := x - x_0$, but since we're not calculating $delta\_x$, we can ignore the value of x here.
- We can't just treat x_0 and $delta\_x$ as named logical constants because they change over time. We can't just write the program without them, since we need $delta\_x$ for the bound function and x_0 for $delta\_x$.
- $delta\_x$ and x_0 will be ***auxiliary variables***: They're program variables, so we can discuss their logical properties, but they're like logical variables in that we don't compute their values.

D. Auxiliary Variables

- ***Definition***: Let S be a program and let $V = Vars(S)$. A set of variables $A \subseteq V$ is an ***auxiliary set*** (for S) if:
  - All computations in S of values in $V - A$ depend only on variables in $V - A$; and
  - All boolean tests in S use only variables from $V - A$.
- The empty set is trivially auxiliary, and if S includes no boolean tests, then V is trivially auxiliary.
- ***Definition***: The ***required variables*** (with respect to A) are the ones in $V - A$.

  - The idea is that when it comes to calculating the values of variables, we're interested only in the values of required variables. Use of required variables can for us to treat other variables as required.
  - E.g., if we're interested in x, then having assignments like $x := y$ and $y := z$ force us to be interested in y and then z too.
  - We can get away with not actually calculating and storing the values of auxiliary variables because their values can't affect the values of required variables.
- ***Definition***: A variable of S is a ***primary variable*** if it is not a member of any auxiliary set of variables for S.
  - All variables that appear in tests are primary, as are the variables needed to calculate their values, directly and indirectly. (I.e, if x is primary, then $x := y$ and $y := z$ force y and z to be primary also.)
- ***Notation:*** To indicate in a program that we intend a variable to be auxiliary, we'll parenthesize it. In Example 1, we would write $(x_0) := x;$ and $(delta\_x) := x - (x_0);$ (We can omit parenthesizing them in conditions.)
- ***Definition***: An ***auxiliary labeling*** for a program tells us which program variables are auxiliary vs required.
- ***Definition***: An auxiliary labeling is ***consistent*** if
  - No auxiliary variable appears in a ***if*** or ***while*** test and
  - For every assignment statement $v := e$, if v is required then all the variables of e are also required. Contrapositively, if any variable in e is auxiliary, then v must be auxiliary.
- A case analysis shows us which usages of auxiliary variables are allowed and which are disallowed. Here, a and a' are auxiliary and r and r' are required. [2023-04-24]
- ***Allowed***: [2023-04-24]
  - $(a) := \ldots r \ldots (a') \ \ldots$ If the rhs contains an auxiliary variable, then the lhs must also be auxiliary. (The assignment forces a dependency from a' to a.) Also, if the lhs is auxiliary, then the rhs can include auxiliary and required variables.
  - $r := \ldots r' \ \ldots$ If the lhs is required, then the rhs can include required variables.
  - ***if*** / ***while*** $\ldots r \ldots$ Required variables can appear in tests.
- ***Disallowed***:
  - $r := \ldots (a) \ldots$ If the lhs is required, then the rhs cannot include auxiliary variables.
  - ***if*** / ***while*** $\ldots (a) \ldots$ Auxiliary variables cannot appear in tests.

Expanding an auxiliary labeling

- Let's call a labeling ***fully expanded*** if it is includes all the variables forced to be auxiliary. I.e., if $v := e$ is an assignment and e includes an auxiliary variable, then v is marked auxiliary.

- A fully expanded labeling is consistent if no ***if / while*** test includes a labeled variable.
- There's a simple algorithm for fully expanding a starting set of variables: If the program contains an assignment notated *y := ... (x) ...* , then mark all occurrences of *y* as *(y)*. Repeat until no such assignment exists.

Example 2:

- Let's expand the initial labeling *{ v }* for the following program. We start with

  x := y ; y := (v) + w ; ***if*** *w ≥ 0* ***then*** *x := x+1 ; w := w–1* ***fi***

- Because of *y := (v) ...* , mark *y*:

  x := (y) ; (y) := (v) + w ; ***if*** *w ≥ 0* ***then*** *x := x+1 ; w := w–1* ***fi***

- Because of *x := (y) ...*, mark *x*:

  (x) := (y) ; (y) := (v) + w ; ***if*** *w ≥ 0* ***then*** *(x) := (x) + 1 ; w := w–1* ***fi***

- No more variables need to be marked as auxiliary, and there are no disallowed uses of auxiliary variables, so *{ v , x , y }* is a consistent set of auxiliary variables.
- More generally for this program, the assignments *x := y* and *y := v + w* generate the following dependencies: *y* being auxiliary forces *x* to be auxiliary, and *v* forces *y*.
  - The assignment *x := x+1* makes *x* force *x*, which is trivial, and since *w* appears in the test, it's primary, so it doesn't matter that *y := v + w* makes *w* force *y* .)
- Altogether, there are three consistent labelings.
  - *(x) := y ; y := v + w ;* ***if*** *w ≥ 0* ***then*** *(x) := (x) + 1 ; w := w–1* ***fi*** // *{ x }* auxiliary
  - *(x) := (y) ; (y) := v + w ;* ***if*** *w ≥ 0* ***then*** *(x) := (x) + 1 ; w := w–1* ***fi*** // *{ x , y }* auxiliary
  - *(x) := (y) ; (y) := (v) + w ;* ***if*** *w ≥ 0* ***then*** *(x) := (x) + 1 ; w := w–1* ***fi*** // *{ v , x , y }* auxiliary
- In the other direction, since three of the *2⁴ – 1 = 15* nontrivial labelings are consistent, the other twelve are inconsistent:
  - Since *w* appears in the ***if*** test, it's primary, so the 8 labelings that include it are inconsistent.
  - Since *x := (y)* is inconsistent, *{ y }* and *{ y , y }* are inconsistent.
  - Since *y := (v) + w* is inconsistent, *{ v }* and *{ v , y }* are inconsistent.

Example 3:

- Consider the program *y := r ;* ***while*** *t > 1* ***do*** *y := y * t ; t := t – k* ***od***.
- The consistent labelings are
  - *(y) := r ;* ***while*** *t > 1* ***do*** *(y) := (y) * t ; t := t – k* ***od*** // *{ y }* auxiliary
  - *(y) := (r) ;* ***while*** *t > 1* ***do*** *(y) := (y) * t ; t := t – k* ***od*** // *{ r , y }* auxiliary
- For inconsistent labelings, we have
  - From ***while*** *t ...*, we know that no labeling can include *t*.
  - From *t := t – (k)*, we know that no labeling can include *k* . Since no labeling with *t* is consistent, *(t) := (t) – (k)* is also inconsistent.)

- From $y := (r)$, we know that r without y is inconsistent.

Example 4:

- Let's go back to the x_0 and $delta\_x$ program from Example 1. (To save space, I've compressed it and removed the ***inv*** and ***bd*** headers.)

  $x_0 := x;\ x := f(x);\ delta\_x := x - x_0;$
  while $\neg P(x)$ ***do*** $x_0 := x;\ x := f(x);\ delta\_x := x - x_0$ ***od***

- Since x appears in the ***while*** test, it must be primary. The assignment $delta\_x := x - x_0$ forces a dependency from x_0 to $delta\_x$, but $delta\_x$ forces no dependencies because it doesn't appear on the rhs of a any assignment.
- There are two consistent labelings. One is $\{delta\_x\}$ and $\{delta\_x, x_0\}$.

  $x_0 := x;\ x := f(x);\ (delta\_x) := x - x_0;$
  while $\neg P(x)$ ***do*** $x_0 := x;\ x := f(x);\ (delta\_x) := x - x_0$ ***od***

- The other consistent labeling is $\{delta\_x, x_0\}$.

  $(x_0) := x;\ x := f(x);\ (delta\_x) := x - (x_0);$
  while $\neg P(x)$ ***do*** $(x_0) := x;\ x := f(x);\ (delta\_x) := x - (x_0)$ ***od***

Example 5:

- As a general example of using auxiliary variables, let's consider the following disjoint parallel program. Recall for the program to be a DPP, x and y do not in appear e_2 and e_1 respectively. On the other hand, the outline does not have disjoint conditions because p_1 and p_2 depend on y and q_1 and q_2 depend on x.

  $\{p_1(x,y) \wedge q_1(x,y)\}$
  $[\ \{p_1(x,y)\}\, x := e_1\, \{p_2(x,y)\}$ // y does not appear in e_1
  $\|\ \{q_1(x,y)\}\, y := e_2\, \{q_2(x,y)\}$ // x does not appear in e_2
  $]\, \{p_2(x,y) \wedge q_2(x,y)\}$

- If we modify the outlines to have disjoint conditions, we can use disjoint parallelism to prove correctness. We'll introduce auxiliary variables x_0 and y_0, change the uses of x in thread 2 to x_0, and change the uses of y in thread 1 to y_0.

  $\{p_1(x,y) \wedge q_1(x,y)\}$
  $x_0 := x;\ y_0 := y;$
  $\{p_1(x,y_0) \wedge q_1(x_0,y)\}$
  $[\ \{p_1(x,y_0)\}\, x := e_1\, \{p_2(x,y_0)\}$
  $\|\ \{q_1(x_0,y)\}\, y := e_2\, \{q_2(x_0,y)\}$
  $]\, \{p_2(x,y_0) \wedge q_2(x_0,y)\}$

- This modified outline concludes $p_2(x,y_0) \wedge q_2(x_0,y)$. We can get the original conclusion $p_2(x,y) \wedge q_2(x,y)$ only if this modified conclusion implies the original conclusion.

Example 6:

- Let's look at a concrete instance of Example 5, starting with

  $\{ x–y = d \}\, [\, x := x+1 \;||\; y := y+1 \,]\, \{ x–y = d \}$

- Neither thread itself maintains $x–y = d$ by itself; it's only the combination that does, so we cannot just make $x–y = d$ the preconditions and postconditions of the threads.
- We can start following the pattern of Example 5:

  $\{ x–y = d \}\, x_0 := x ;\, y_0 := y\, \{ x_0–y_0 = d \wedge x_0 – y_0 = d \}$ [2023-04-24]
  $[\, \{ x_0–y_0 = d \}\, x := x+1\, \{ ??? \}$
  $\;||\; \{ x_0 – y_0 = d \}\, y := y+1\, \{ ??? \}$
  $\,]\; \{ ??? \wedge ??? \}\, \{ x–y = d \}$

- We need to figure out the missing conditions. If we use *sp* on each thread, we get
  - $\{ x_0 = x \wedge x_0 – y_0 = d \}\; x := x+1\; \{ x = x_0+1 \wedge x_0 – y_0 = d \}$
  - $\{ y_0 = y \wedge x_0 – y_0 = d \}\; y := y+1\; \{ y = y_0+1 \wedge x_0 – y_0 = d \}$
- Since $x = x_0 + 1 \wedge x_0 – y_0 = d$ and $y = y_0 + 1 \wedge x_0 – y_0 = d$ implies $x–y = (x_0 + 1) – (y_0 + 1) = x_0 – y_0 = d$, we can combine the two threads and get

  $\{ x–y = d \}\, x_0 := x ;\, y_0 := y\, \{ x_0 = x \wedge x_0 – y_0 = d \wedge y_0 = y \wedge x_0 – y_0 = d \}$
  $[\, \{ x_0 = x \wedge x_0 – y_0 = d \}\, x := x+1\, \{ x = x_0 + 1 \wedge x_0 – y_0 = d \}$
  $\;||\; \{ y_0 = y \wedge x_0 – y_0 = d \}\, y := y+1\, \{ y = y_0 + 1 \wedge x_0 – y_0 = d \}$
  $\,]\, \{ x = x_0+1 \wedge x_0 – y_0 = d \wedge y = y_0+1 \wedge x_0 – y_0 = d \}$
  $\{ x–y = d \}$ // x and y have been modified in the same way [2023-04-24]

- We use x_0 and y_0 here in the conditions of the threads but not the code, so they can be seen as logical variables:

  $\{ x_0 = x \wedge y_0 = y \wedge x–y = d \}\; \ldots \textit{program} \ldots\, \{ x–y = d \}$

E. Removing Auxiliary Variables

- We need to connect the behavior of programs with and without auxiliary variables. It turns out to be easier to discuss the behavior of removing auxiliary variables instead of adding them, so we'll do it that way.
- ***Definition***: Let S be a program and A be a set of auxiliary variables. Then $S – A$ (*"S with A removed"*) is S where where each assignment to a variable in A has been replaced by a ***skip*** statement.
- It's easy to optimize $S – A$ by replacing ***skip***; S' and S '; ***skip*** with just S' and repeating until this can't be done. If B cannot cause a runtime error, then there's also the optimization of replacing ***if*** B ***then skip else skip fi*** with just ***skip***.
  - Note it's possible to cycle through the pair of optimizations.

Example 7:

- Going back to the program and labelings of Examples 1 and 4, we had two consistent labelings: $\{ delta\_x \}$ gave

  $x_0 := x ;\ x := f(x) ;\ (delta\_x) := x - x_0 ;$ **while** $\neg P(x)$ **do** $x_0 := x ;\ x := f(x) ;\ (delta\_x) := x - x_0$ **od**

  - Removal of $\{ delta\_x \}$ yields

    $x_0 := x ;\ x := f(x) ;$ **skip;** **while** $\neg P(x)$ **do** $x_0 := x ;\ x := f(x) ;$ **skip od**

  - This optimizes to

    $x_0 := x ;\ x := f(x) ;$ **while** $\neg P(x)$ **do** $x_0 := x ;\ x := f(x)$ **od**

- The other consistent labeling was $\{ delta\_x , x_0 \}$:

  $(x_0) := x ;\ x := f(x) ;\ (delta\_x) := x - (x_0) ;$
  while $\neg P(x)$ **do** $(x_0) := x ;\ x := f(x) ;\ (delta\_x) := x - (x_0)$ **od**

  - Removal gives

    skip; $x := f(x) ;$ **skip;** **while** $\neg P(x)$ **do skip;** $x := f(x) ;$ **skip od**

  - This optimizes to

    $x := f(x) ;$ **while** $\neg P(x)$ **do** $x := f(x)$ **od**

Example 8:

- Let's go back to Example 2, where we had a program with three auxiliary labelings.
- First was the labeling $\{ x \}$. Marking, removing, and optimizing gives
  - $S \equiv (x) := y ;\ y := v + w ;$ **if** $w \geq 0$ **then** $(x) := (x) + 1 ;\ w := w - 1$ **fi**
  - $S - \{ x \} \equiv$ **skip;** $y := v + w ;$ **if** $w \geq 0$ **then skip;** $w := w - 1$ **fi**
  - $S - \{ x \}$ after optimization: $y := v + w ;$ **if** $w \geq 0$ **then** $w := w - 1$ **fi**
- For $\{ x , y \}$ we get
  - $S \equiv (x) := (y) ;\ (y) := v + w ;$ **if** $w \geq 0$ **then** $(x) := (x) + 1 ;\ w := w - 1$ **fi**
  - $S - \{ x , y \} \equiv$ **skip;** **skip;** **if** $w \geq 0$ **then skip;** $w := w - 1$ **fi** [2023-04-24]
  - $S - \{ x , y \}$ after optimization: **if** $w \geq 0$ **then** $w := w - 1$ **fi**
- For $\{ v , x , y \}$, we get a different marking from $\{ x , y \}$ but the same results after removal and optimization:
  - $S \equiv (x) := (y) ;\ (y) := (v) + w ;$ **if** $w \geq 0$ **then** $(x) := (x) + 1 ;\ w := w - 1$ **fi**
  - $S - \{ v , x , y \} \equiv$ **skip;** **skip;** **if** $w \geq 0$ **then skip;** $w := w - 1$ **fi** [2023-04-24]
  - $S - \{ v , x , y \}$ after optimization: **if** $w \geq 0$ **then** $w := w - 1$ **fi**

- ***Know this for the exam***[1]: You should be able to fully expand a labeling, be able to verify that a labeling is consistent, and be able to remove the auxiliary variables from a program. (The practice will help with these skills.)

F. Programs With Auxiliary Variables: Execution and Proof Rules

- ***Know this for the exam***: The goal is to argue that removing auxiliary variables from a program does not change how the program works on required variables.
- To phrase this, it helps to start with a lemma about a single operational semantics step ($\rightarrow$), which makes it easy to go to overall operational semantics ($\rightarrow^*$).
- ***Theorem*** (***Preservation of State Changes***):
  - ***Know this for the exam:*** Removing a program's auxiliary variables yields a program that modifies the non-auxiliary variables in exactly the same way as the original program.
- More formally, let S be a program with auxiliary and required variables A and R. Let $\sigma \cup \tau$ be a state for S where σ covers R and τ covers A. (I.e., their domains are A and R respectively) If $\langle S, \sigma \cup \tau \rangle \rightarrow^* \langle S', \sigma' \cup \tau' \rangle$, then $\langle S - A, \sigma \rangle \rightarrow^* \langle S' - A, \sigma' \rangle$.
- ***Proof***: It's sufficient to verify that single-step execution of S and $S - A$ behave the same on non-auxiliary variables. (We can iterate correctness of $\rightarrow$ to get correctness of $\rightarrow^*$.) Since S and $S - A$ differ only in $S - A$ having ***skip*** where S has assignments to auxiliary variables, this is the important case; ***if*** and ***while*** also have to be discussed; ***skip*** is trivial and omitted.

  Say S includes $v := e$, then $\langle v := e, \sigma \cup \tau \rangle \rightarrow \langle E, (\sigma \cup \tau)[v \mapsto a] \rangle$ where $a = (\sigma \cup \tau)(e)$. If v is auxiliary, the update to $\sigma \cup \tau$ can only affect σ, so we have $\langle v := e, \sigma \cup \tau \rangle \rightarrow \langle E, \sigma \cup \tau[v \mapsto a] \rangle$. The corresponding execution in $S - A$ is $\langle$ ***skip***$, \sigma \rangle \rightarrow \langle E, \sigma \rangle$, so the programs behave the same way on R.

  For ***if*** and ***while*** statements, removing A does not change the tests in ***if*** B and ***while*** B, so S jumps depend on $(\sigma \cup \tau)(B)$ and $S - A$ jumps depend on $\sigma(B)$. But B contains only required variables, so $(\sigma \cup \tau)(B) = \sigma(B)$, so the behavior in S and $S - A$ are the same on R. //
- Now that we understand the semantics of adding and removing auxiliary variables, we can formalize these as sound proof rules.
- ***Theorem (Preservation of Validity)***:
  - ***Know this for the exam***: If a program's specification doesn't involve auxiliary variables, then we can remove the auxiliary variables from the program without changing the specification.
- More formally, let $\models \{p\}\, S'\, \{q\}$ with auxiliary and required variables A and R. If no variables of A are free in p and q, then $\models \{p\}\, S' - A\, \{q\}$.
  Proof: Let $\sigma \models p$ be a state that covers R, and let τ cover A so that $\sigma \cup \tau$ is a state for S. Say $\langle S, \sigma \cup \tau \rangle \rightarrow^* \langle E, \sigma' \cup \tau' \rangle$ where σ' and τ' cover A and R. By the preservation theorem, we

[1] Things labeled "Know this for the exam" are important. Unimportant parts, like the proofs of the theorems in section F, are here because I had to write them out to convince myself they were correct.

know $\langle S - A, \sigma \rangle \to^* \langle E - A, \sigma' \rangle$. For satisfaction of q, validity of $\{ p \} S \{ q \}$ implies $\sigma' \cup \tau' \vDash q$. Since q depends only on R, this implies $\sigma' \vDash q$. So $\sigma \vDash \{ p \} S - A \{ q \}$.

Auxiliary Variable Removal

1. $\{ p \} S \{ q \}$
2. $\{ p \} S - A \{ q \}$ Auxiliary variable removal, *1*, A
   where no free variables of p or q appear in A.

Await and Deadlocks

CS 536: Science of Programming, Spring 2023

A. Why?

- Avoiding interference isn't the same as coordinating desirable activities.
- It's common for one thread to wait for another thread to reach a desired state.
- Care needs to be taken to avoid a program that waits with no hope of completing.

B. Objectives

At the end of this lecture you should know

- The syntax and semantics of the ***await*** statement.
- How to draw an evaluation diagram for a parallel program that uses ***await***.
- How to recognize deadlocked configurations in an evaluation diagram.
- How to list the potential deadlock predicates for a parallel program that uses ***await***.

C. Synchronization

The Need for Synchronization

- We've looked at parallel programs whose threads avoid bad interactions.
- They don't interfere because they don't interact (disjoint programs/conditions).
- They interact but don't interfere (interference-freedom).
- To supporting good interaction between threads, we often have to have one thread wait for another one. Some examples:
  - Thread 1 should wait until thread 2 is finished executing a certain block of code.
  - Thread 1 has to wait until some buffer is not empty
  - Thread 2 has to wait until some buffer is not full.
- The general problem is that we often want threads to ***synchronize***: We want one thread to wait until some other thread makes a condition come true.
- ***Example 1***: For a more specific example, in the following program, the calculation of u doesn't start until we finish calculating z, even though u doesn't depend on z.

  $[x := \ldots \,||\, y := \ldots \,||\, z := \ldots];\ u = f(x, y);\ v := g(u, z)$

  On the other hand, we can't nest parallel programs, so we can't write

  $[[[x := \ldots \,||\, y := \ldots];\ u = f(x, y) \,||\, z := \ldots];\ v := g(u, z)$

  which would be a natural way to do the calculations of u and z in parallel. In some sense, what we'd like is to run something like

$[\, x := \ldots \,||\, y := \ldots \,||$ wait for x and y; $u = f(x, y) \,||\, z := \ldots \,]\,; v := g(u, z)$

D. The Await Statement

- It's time to introduce a new statement, the ***await*** statement, whose semantics implements the notion of waiting until some condition is true.
  - Busy wait loops like ***while*** $\neg B$ ***do skip od*** $\{ B \}$ work but are wasteful.
- ***Syntax***: ***await*** B ***then*** S ***end*** where B is a boolean expression and S is a statement.
  - S isn't allowed to have loops, ***await*** statements, or atomic regions.
  - ***await*** statements can only appear in sequential threads of parallel programs. (I.e., in some thread S_k in an $[\, S_1 \,||\, S_2 \,||\, \ldots \,]$.)
- An ***await*** statement is a ***conditional atomic region***. Suppose that some thread begins with ***await*** B ***then*** S ***end***, then
  - We nondeterministically choose between all the available threads. I.e., there's no insistence that we must check the ***await*** before trying other threads. (See case 1 of Example 2.)
  - If we choose the thread that begins with ***await*** B ***then*** S ***end***,
  - If B is true, then immediately jump to S and execute all of it.
    - The test, jump, and execution of S are atomic — the combination executes as one step. E.g., with the configuration below, we can't set x to 1 between looking up the two x's to use for calculating $x+x$. (See case 2 of Example 2.)
  - If B is false, we ***block***: We wait until B is true. Instead, we nondeterministically choose between the other threads and execute it. (See case 3 of Example 2.)
- An ***await*** is similar to an atomic ***if-then*** statement, but not identical.
  - With $\langle$ ***if*** B ***then*** S ***else skip fi*** $\rangle$, if B is false, we execute ***skip*** and complete the ***if-fi***. (See case 4, Example 2.)
  - With ***await*** B ***then*** S ***end***, if B is false, nothing happens until B becomes true. (See the note with case 3, Example 2.)

Example 2:

- (See the discussion above)
  - Case 1: Let $A \equiv$ ***await*** B ***then*** $x := x+x$ ***end*** in $\langle\, [\, $***await*** B ***then*** $x := x+x$ ***end*** $\,||\, x := 1\,]\,\{ b=T, x=2 \}\,\rangle \rightarrow \langle\, [\, $***await*** B ***then*** $x := x + x$ ***end*** $\,||\, E\,],\ \{ b = T, x=1 \}\,\rangle$.
  - Case 2: $\langle\, [\, $***await*** B ***then*** $x := x+x$ ***end*** $\,||\, x := 1\,], \{ b=T, x=2 \}\,\rangle \rightarrow \langle\, [\, E \,||\, x := 1\,], \{ b=T, x=4 \}\,\rangle$. (This is the only transition that executes the ***await***.)
  - Case 3: $\langle\, [\, $***await*** B ***then*** S ***end*** $\,||\, x := 1\,], \{ b=F \}\,\rangle \rightarrow \langle\, [\, $***await*** B ***then*** S ***end*** $\,||\, E\,], \{ b=F, x=1 \}\,\rangle$. (The second configuration is blocked, with no other thread available to unblock it.)
  - Case 4: $\langle\, [\, $***if*** B ***then*** $x := 0$ ***else skip fi*** $\,||\, S'\,], \{ b=F \}\,\rangle \rightarrow \langle\, [\, E \,||\, S'\,], \{ b=F, x=0 \}\,\rangle$.

- ***Example 3:*** Execution of a non-atomic ***if-fi*** can be interleaved with. In Figure 1, the **dashed red lines** show how execution of ***await*** $x \geq 0$ ***then*** $x := x+1 ; y := x+2$ ***end*** takes just one step to execute the entire body.

Solid black lines show execution steps taken only when $S \equiv$ ***if*** $x \geq 0$ ***then*** $x := x+1; y := x+2$ ***fi***

Dashed red lines show steps taken only when $S \equiv$ ***await*** $x \geq 0$ ***then*** $x := x+1; y := x+2$ ***end***

Dashed black lines are common to both executions.

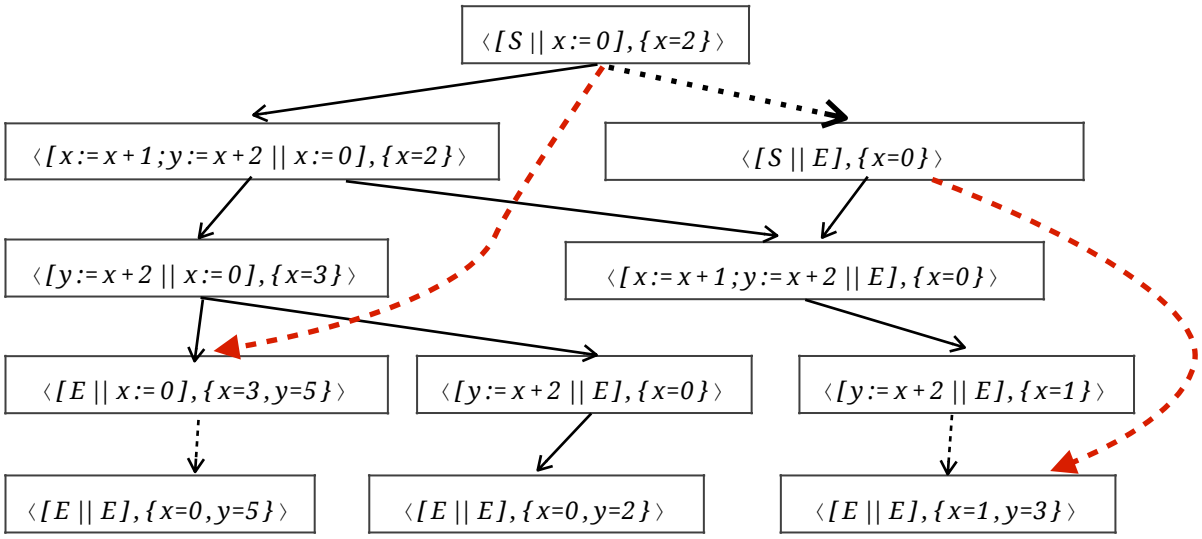


Figure 1: Execution of ***await*** vs ***if-fi***

- ***Example 4***: In the introduction, we looked at a situation where we want to wait for some calculations to finish before stating others.

  $[x := \ldots \,||\, y := \ldots \,||$ ***wait*** *for* x *and* y; $u = f(x,y) \,||\, z := \ldots]$; $v := g(u,z)$

  can be implemented using

  $x\_done := F$; $y\_done := F$;
  $[x := \ldots;\ x\_done := T \,||\, y := \ldots;\ y\_done := T$
  $||$ ***await*** $x\_done \wedge y\_done$ ***then*** $u := f(x,y)$ ***end*** $||\, z := \ldots]$;
  $v := g(u,z)$;

E. await, wait, if, and ⟨ *S* ⟩

The Abbreviations ⟨ *S* ⟩ *and wait B*

- With ***await*** *B* ***then*** *S* ***end***, there are two simple cases: When *B* is trivial and when *S* is trivial.

- ***Definition***: We can redefine $\langle S \rangle$ to stand for ***await*** T ***then*** S ***end***. When the test is trivially true, we don't need to wait, we simply execute the body atomically. So atomic execution is just conditional atomic execution with a trivial test.
- ***Definition***: ***wait*** $B \equiv$ ***await*** B ***then skip end***. When the body is trivial, we simply wait; when B is true, execution is complete.
- There's a important difference between ***wait*** B ***;*** S and ***await*** B ***then*** S ***end***.
  - With ***await*** B ***then*** S ***end***, once B is true, we immediately atomically execute S, so no other statement can interleave between the test and running S. Therefore S can rely on B being true when it starts executing. If $\sigma(B) = T$, then $\langle [$ ***await*** B ***then*** S ***end*** $|| \ldots], \sigma \rangle \rightarrow \langle [E || \ldots], \tau \rangle$, where $\tau \in M(S, \sigma)$.
  - ***wait*** B ***;*** S means ***await*** B ***then skip end;*** S, so it allows another thread to be executed after the ***wait*** but before running S. If $\sigma(B) = T$, then $\langle [$ ***wait*** B ; $S || \ldots], \sigma \rangle \rightarrow \langle [S || \ldots], \sigma \rangle$ $\rightarrow {}^*\langle [E || \ldots], \tau \rangle$ (if no interleaving occurs). Since interleaving can occur, we rely on B being true when S starts execution.

F. Await Statement Proof Rule and Outlines

- The proof rule for the ***await*** statement is similar to an ***if fi***, but there's no false clause (not even ***else skip***).

await Statement (a.k.a. Synchronization Rule)

1. $\{p \wedge B\}\, S\, \{q\}$
2. $\{p\}$ ***await*** B ***then*** S ***end*** $\{q\}$ ***await***, *1*

- ***Minimal Proof Outline***: $\{p\}$ ***await*** B ***then*** S ***end*** $\{q\}$
- ***Full Proof Outline***: $\{p\}$ ***await*** B ***then*** $\{p \wedge B\}\, S^*\, \{q\}$ ***end*** $\{q\}$ where S^* is a full proof outline for program S.
- ***Weakest Preconditions***: $wp($ ***await*** B ***then*** S ***end***$, q) \equiv B \rightarrow wp(S, q)$.
  - This guarantees $\{B \rightarrow wp(S, q)\}$ ***await*** B ***then*** $\{wp(S, q)\}\, S^*\, \{q\}$ ***end*** $\{q\}$
- Note: It may be tempting to write $\{p \wedge \neg B\}$ ***await*** B ***then*** ..., but that's guaranteed to self-deadlock; the outline is

$$\{p \wedge \neg B\}\ \textbf{await}\ B\ \textbf{then}\ \{p \wedge \neg B \wedge B\}\, S^*\, \{q\}\ \textbf{end}\{q\}$$

G. The Producer/Consumer Problem

- The ***Producer/Consumer Problem*** (a.k.a. ***Bounded Buffer Problem***) is a standard problem in parallel programming.
- We have two threads running in parallel: The producer creates things and puts them into a buffer; the consumer removes things from the buffer and does something with them.

- The problem is that if the buffer is full, the producer shouldn't add anything to the buffer; if the buffer is empty, the consumer shouldn't remove anything from the buffer.
- ***Example 5***: The rough code to solve this problem is

```
Initialize(buffer);
[ while ¬ done do                          // Producer
    created := Create();
    await NotFull(buffer) then
        BufferAdd(buffer, created)
    end
 od
|| while ¬ done do                          // Consumer
    await NotEmpty(buffer) then
        removed := BufferRemove(buffer);
    end;
    Consume(removed)
    od
]
```

- Buffer operations need to be synchronized because the threads share the buffer. The threads don't share the created or removed objects, so the *Create* and *Consume* calls can go outside the ***await*** and interleave execution.

H. Deadlock

Blocked Threads; Deadlock

- Recall that $\langle$ *[**await** B **then** S **end**; ... || ...]* , $\sigma \rangle$ is blocked (must wait) if $\sigma(B) = F$.
  - If some other thread can make B true, then the await may eventually unblock.
  - E.g., $\langle$ *[**await** x > y **then** S **end**; ... || ...; x := y+1 ; ...]* , $\sigma \rangle$ could unblock.
  - But if all the other threads have either completed or are themselves blocked, then there's no way for our ***await*** to unblock. E.g., take $\langle$ *[**await** B **then** S **end**; ... || E]* , $\sigma \rangle$. If thread 1 is stuck at the ***await*** but thread 2 has completed, the program can't evaluate further.
- ***Definition***: A parallel program is ***deadlocked*** if it has not finished execution and there's no possible evaluation step to take. I.e., all the threads are either complete or blocked and at least one thread is blocked.
- If all the other threads are complete or are blocked, the program is ***deadlocked:*** There's no possible evaluation step leaving from the configuration.
- ***Example 6***: If $A \equiv$ ***await*** $x \geq 0$ ... ***end***, then there's no arrow out of $\langle$ *[A || A]* , σ *[* $x \mapsto -1$ *]* $\rangle$, so this configuration is deadlocked. If the value of x had been ≥ 0, then both ***await*** statements would have been eligible for execution.

Since only one blocked thread is required for deadlock, $\langle$ *[**await** $x \geq 0$ ... **end** || E] , σ [$x \mapsto -1$] $\rangle$ is also deadlocked.

- Threads can block themselves (trivial example: ***await false then*** S ***end***).
  - More often, threads block because they're waiting for conditions they expect other threads to establish. E.g., if we're running in a state where $y=0$ and $x=0$, then these two threads deadlock:
  - Thread 1: $\{ p_1 \}$ **await** $y \neq 0$ **then** $x := 1$; ...
  - Thread 2: $\{ p2 \}$ **await** $x \neq 0$ **then** $y := 1$; ...
- A program might deadlock under all execution paths or only certain execution paths.
- ***Example 7***: The program

  [**await** $y \neq 0$ **then** $x := 1$ **end** || **await** $x \neq 0$ **then** $y := 1$ **end**]

  deadlocks iff you execute in a state where x and y are both zero
- ***Example 8***: If thread 1 sets $x := 0$ before thread2 evaluates its ***wait*** x, then thread 2 will block. (Recall ***wait*** x ≡ ***await*** x ***then skip end.***)

  $\{ T \}$ $x := 1$; $y := 1$;
  [**wait** $y=1$; $x := 0$ || **wait** $x=1$; $y := 0$]
  $\{ x=0 \wedge y=0 \}$

Figure 2 contains an execution graph for this program in state {$x=1$, $y=1$ } (somewhat abbreviated). There are two deadlocking paths (and four paths that terminate correctly).

- Obviously, we'd like to know if a program is going to deadlock. The following test identifies a set of predicates that indicate potential problems with a program; if none of these predicates is satisfiable, then deadlock is guaranteed not to occur.
- If one or more of these predicates is satisfiable, then we can't guarantee that deadlock will not occur, but we aren't guaranteeing that deadlock ***must*** occur. (So the deadlock conditions are sufficient to show deadlock is impossible but they are not necessary conditions.)
- Let $\{ p \} [\{ p_1 \} S_1 * \{ q1 \} \ || \ \{ p2 \} S_2 * \{ q2 \} \ || \ ... \ || \ \{ p_n \} S_n * \{ q_n \}] \{ q \}$ be a full outline for a parallel program, where $p \equiv p_1 \wedge ... \wedge p_n$ and $q \equiv q_1 \wedge ... \wedge q_n$.
- ***Definition***: A (***potential***) ***deadlock condition*** for the program outline above is a predicate of the form $r1' \wedge r2' \wedge ... \wedge r_n'$ where each r_k' is either
  - q_k , the postcondition for thread S_k or
  - $p \wedge \neg B$ where $\{ p \}$ **await** B... appears in the proof outline for thread S_k.
  - In addition, at least one of the r_k' must involve waiting. I.e., $q \equiv q_1 \wedge ... \wedge q_n$ is not a potential deadlock condition.
- A program outline is ***deadlock-free*** if every one of its potential deadlock conditions is unsatisfiable (i.e., a contradiction):
  - I.e., for each deadlock condition r', we have $\models \neg r'$ (or the equivalent $\models r' \rightarrow F$).

Parallelism with Deadlock Freedom

1. $\{p_1\}\, S_1^*\, \{q_1\}$
2. $\{p2\}\, S_2^*\, \{q_2\}$
...
n. $\{p_n\}\, S_n^*\, \{q_n\}$
n+1. $\{p_1 \wedge p_2 \wedge \ldots \wedge p_n\}$
$[S_1 \,||\, S_2 \,||\, \ldots \,||\, S_n]$
$\{q_1 \wedge q_2 \wedge \ldots \wedge q_n\}$ D.P. w/o deadlock, *1, 2, ..., n*

where the $\{p_k\}\, S_k^*\, \{q_k\}$ are pairwise interference-free standard proof outlines and the parallel program outline is deadlock-free.

I. Examples of Deadlock Conditions

- ***Example 9***: Let's take the program from Example 7:

  [***await*** $y \neq 0$ ***then*** $x := 1$ ***end*** || ***await*** $x \neq 0$ ***then*** $y := 1$ ***end***]

  and develop an annotation for it:

  $\{T\}$
  $[\{T\}$ ***await*** $y \neq 0$ ***then*** $\{y \neq 0\}\, x := 1\, \{x \neq 0 \wedge y \neq 0\}$ ***end*** $\{x \neq 0 \wedge y \neq 0\}$
  $\,||\, \{T\}$ ***await*** $x \neq 0$ ***then*** $\{x \neq 0\}\, y := 1\, \{x \neq 0 \wedge y \neq 0\}$ ***end*** $\{x \neq 0 \wedge y \neq 0\}$
  $]\, \{x \neq 0 \wedge y \neq 0\}$

- Let set $D_1 = \{x \neq 0 \wedge y \neq 0, y = 0\}$ be the choices for p_1'.
- $x \neq 0 \wedge y \neq 0$ is the thread postcondition
- $y = 0$ indicates thread 1 is blocked at the ***await*** statement.
- Similarly, let set $D_2 = \{x \neq 0 \wedge y \neq 0, x = 0\}$ be the choices for p_2' (the postcondition of thread 2 and the blocking condition for its ***await***).
- There are three choices for the potential deadlock predicate $r1' \wedge r2'$:
- $(x \neq 0 \wedge y \neq 0) \wedge (x = 0)$, which is a contradiction.
- $(y = 0) \wedge (x \neq 0 \wedge y \neq 0)$, which is a contradiction.
- $(y = 0) \wedge (x = 0)$, which is not a contradiction, therefore, it's a potential deadlock condition, and our program does not pass the deadlock-freedom test.
- Recall $(x \neq 0 \wedge y \neq 0) \wedge (x \neq 0 \wedge y \neq 0)$ is not a potential deadlock predicate because it says that the two threads have both completed.
- One way out of this predicament is to make the initial precondition the negation of $y = 0 \wedge x = 0$. Let p be $(x \neq 0 \vee y \neq 0)$ in

  $\{p\}$
  $[\{p\}$ ***await*** $y \neq 0$ ***then*** $\{p \wedge y \neq 0\}\, x := 1\, \{x \neq 0 \wedge y \neq 0\}$ ***end*** $\{x \neq 0 \wedge y \neq 0\}$
  $\,||\, \{p\}$ ***await*** $x \neq 0$ ***then*** $\{p \wedge x \neq 0\}\, y := 1\, \{x \neq 0 \wedge y \neq 0\}$ ***end*** $\{x \neq 0 \wedge y \neq 0\}$
  $]\, \{x \neq 0 \wedge y \neq 0\}$

- Let $D_1 = \{ x \neq 0 \wedge y \neq 0 , p \wedge y = 0 \}$ and let $D_2 = \{ x \neq 0 \wedge y \neq 0 , p \wedge x = 0 \}$.
- The three potential deadlock predicates are now contradictory
  - $(x \neq 0 \wedge y \neq 0) \wedge (p \wedge x = 0)$ (is false because of $x \neq 0 \wedge x = 0$)
  - $(p \wedge y = 0) \wedge (x \neq 0 \wedge y \neq 0)$ (is false because of $y = 0 \wedge y \neq 0$)
  - $(p \wedge y = 0) \wedge (p \wedge x = 0)$
    $\equiv ((x \neq 0 \vee y \neq 0) \wedge y = 0) \wedge ((x \neq 0 \vee y \neq 0) \wedge x = 0)$
    $\Rightarrow (x \neq 0 \wedge y = 0) \wedge (y \neq 0 \wedge x = 0)$
    $\Rightarrow F$
- (end of example 9)
- ***Example 10***: Since it has three threads, the deadlock conditions for this program are a bit more involved than for Example 9. Thread 1 has one ***await*** statement, thread 2 has two ***await*** statements, and thread 3 has no ***await*** statements.

  $[\ldots \{p_{11}\}$ ***await*** $B_{11} \ldots \{q_1\}$
  $\| \ldots \{p_{21}\}$ ***await*** $B_{21} \ldots \{p_{22}\}$ ***await*** $B_{22} \ldots \{q_2\}$
  $\| \ldots \{q_3\}]$
- The deadlock conditions are built using the three sets
  - $D_1 = \{ p_{11} \wedge \neg B_{11}, q_1 \}$
  - $D_2 = \{ p_{21} \wedge \neg B_{21}, p_{22} \wedge \neg B_{22}, q_2 \}$
  - $D_3 = \{ q_3 \}$.
- Let D be the set of deadlock conditions, $D = \{ r_1 \wedge r_2 \wedge r_3 \mid r_1 \in D_1, r_2 \in D_2, r_3 \in D_3 \} - \{ q_1 \wedge q_2 \wedge q_3 \}$. Specifically, we get the following $(2 \times 3 \times 1 - 1 = 5)$ conditions:

  $D = \{ (p_{11} \wedge \neg B_{11}) \wedge (p_{21} \wedge \neg B_{21}) \wedge q_3,$ — Thread 1 blocked; thread 2 blocked at 1st await
  $(p_{11} \wedge \neg B_{11}) \wedge (p_{22} \wedge \neg B_{22}) \wedge q_3,$ — Thread 1 blocked; thread 2 blocked at 2nd await
  $(p_{11} \wedge \neg B_{11}) \wedge q_2 \wedge q_3,$ — Thread 1 blocked
  $q_1 \wedge (p_2 \wedge \neg B_{21}) \wedge q_3,$ — Thread 2 blocked at 1st await
  $q_1 \wedge (p_{22} \wedge \neg B_{22}) \wedge q_3 \}$ — Thread 2 blocked at 2nd await
- The program will be deadlock-free if every predicate in D is a contradiction (i.e., unsatisfiable).

J. Strengthening Deadlock Conditions

- Having all deadlock conditions be contradictory is sufficient for guaranteeing that no program execution will deadlock.
- It's not a necessary condition, however. Just because some $r \in D$ is satisfiable, that doesn't mean that there exists a program execution that can get to the corresponding deadlocked configuration.

- ***Example 11***: Here's an example of strengthening conditions so that we can prove deadlock freedom. The program is small enough for us to be able to hand-verify that it never deadlocks (by figuring out all possible interleavings).

$\{ T \}\ n := 0\ ;\ [\ \textbf{await}\ n=0\ \textbf{then}\ n := 1\ \textbf{end} \,||\, \textbf{wait}\ n=1\ ]\ \{ n > 0 \}$

- If we annotate the program as below, we have sequential correctness for each thread, plus the threads are interference-free:

  $\{ T \}\ n := 0\ ;\ \{ T \}$
  $[\ \{ T \}\ \textbf{await}\ n=0\ \textbf{then}\ n := 1\ \textbf{end}\ \{ n > 0 \}$
  $||\ \{ T \}\ \textbf{wait}\ n=1\ \{ n > 0 \}$
  $]\ \{ n > 0 \}$

- On the other hand, we can't prove deadlock freedom. There are $2 \times 2 - 1 = 3$ deadlock conditions and all of them are satisfiable:
  - $n \neq 0 \wedge n \neq 1$ — Both threads blocked
  - $n \neq 0 \wedge n > 0$ — Thread 1 blocked
  - $n > 0 \wedge n \neq 1$ — Thread 2 blocked
- The problem here is that the proof outline's conditions are too weak. We want each deadlock condition to be logically equivalent to false, the strongest predicate.
- To make a conjunctive formula stronger, we need to strengthen its conjuncts. For a deadlock-freedom test, we have two kinds of conjuncts:
  - (postcondition of thread)
  - (precondition of ***await*** statement)$\wedge\neg$ *(* test of ***await*** statement)
- By strengthening the postcondition of the initial assignment of $n := 0$ from true to $n=0$, we can strengthen the precondition of the first ***await***:

  $\{ T \}\ n := 0\ ;\ \{ n=0 \}$
  $[\ \{ n=0 \}\ \textbf{await}\ n=0\ \textbf{then}\ \{ n=0 \wedge n=0 \}\ n := 1\ \textbf{end}\ \{ n > 0 \}$
  $||\ \{ T \}\ \ \textbf{await}\ n=1\ \textbf{then}\ \{ n=1 \}\ \ \textbf{skip}\ \{ n=1 \}\ \textbf{end}\ \{ n > 0 \}\ ]$
  $\{ n > 0 \}$

- The potential deadlock conditions for the proof outline above are now
  - $(n=0 \wedge n \neq 0) \wedge n \neq 1$ — Both threads blocked (contradiction)
  - $(n=0 \wedge n \neq 0) \wedge n > 0$ — Thread 1 blocked (contradiction)
  - $n > 0 \wedge n \neq 1$ — Thread 2 blocked (satisfiable)
- So two of the conditions are contradictory, but one condition is still satisfiable. To prove deadlock-freedom, we need to strengthen the conditions even more to include the state we get to when the first thread has executed and the second thread hasn't.
- Unfortunately, if we annotate the two threads as
  - $\{ n=0 \}\ \textbf{await}\ n=0\ \textbf{then}\ n := 1\ \textbf{end}\ \{ n=1 \}$
  - $\{ n=1 \}\ \textbf{wait}\ n=1\ \{ n=1 \}$
- Then the precondition of the parallel program has to be $(n=0) \wedge (n=1)$, which isn't possible. Even if it were, we'd need to be sure it follows form the strongest postcondition of $n := 0$.

{ T } n := 0 ;
{ n=0 ∧ n=1 } // ← error
[{ n=0 } ***await*** *n=0* ***then*** *n := 1* ***end*** *{ n=1 }*
|| { n=1 } ***wait*** *n=1 { n=1 }*
] { n=1 ∧ n=1 } { n=1 }

- Before thread 2 runs, it sees *n=0* or *n=1* depending on whether thread 1 has run yet. If we use that as the precondition for thread 2, then we get *n=0 ∧ (n=0 ∨ n=1)* as the precondition for the parallel program, which works:

  { T } n := 0 ;
  n=0 ∧ (n=0 ∨ n=1)
  [{ n=0 } ***await*** *n=0* ***then*** *n := 1* ***end*** *{ n=1 }*
  || { n=0 ∨ n=1 } ***wait*** *n=1{ n=1 }*
  { n=1 ∧ n=1 } { n=1 }

- Better still, the deadlock conditions are now all contradictions, so we have deadlock-freedom
  - *(n=0 ∧ n≠0) ∧ ((n=0 ∨ n=1) ∧ n≠1)* — Both blocked (contradiction)
  - *(n=0 ∧ n≠0) ∧ n=1* — Thread 1 blocked (contradiction)
  - *n=1 ∧ ((n=0 ∨ n=1) ∧ n≠1)* — Thread 2 blocked (contradiction)
- Unfortunately, one of the interference freedom tests now fails:
  - Pass: *{ n=0 ∧ (n=0 ∨ n=1) }* ***await*** *n=0* ***then*** *n := 1* ***end*** *{ n=0 ∨ n=1 }*
  - Pass: *{ n=0 ∧ n=1 }* ***await*** *n=0* ***then*** *n := 1* ***end*** *{ n=1 }*
  - Fail: *{ (n=0 ∧ n=1) ∧ n=0 }* ***wait*** *n=1 { n=0 }* — ***wait*** *n=1* definitely doesn't preserve *n=0*
- We can solve this problem by adding an auxiliary variable to say whether or not the first thread has run and set *n=1* . (end of Example 11)