

Types, Expressions, and States

CS 536 Lecture 3, Wed Jan 18, 2012

version Mon, Mar 5, 2012

A. Why

- Expressions represent values in programming languages, relative to a state.
- Types describe common properties of sets of values.
- States describe memory; an expression has a value relative to a state; ~~a predicate is satisfied or unsatisfied relative to a state.~~

B. Outcomes

At the end of today, you should

- Understand what expressions we'll be using in our language.
- Understand what a state is, how we're representing them, and how they connect to expressions.
- Understand what it means for a predicate to be satisfied in a state or valid.

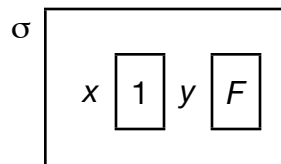
C. Types and Expressions

- 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), and maybe characters.
 - Composite types: Multi-dimensional arrays of primitive types of values, with integer indexes.
- Expressions built from
 - Constants: Integers (0, 1, -1, ...) and Boolean constants (T, F).
 - "Simple" variables of primitive types.
 - The "Complex" variables are the array indexing expressions $b[e_1, e_2, \dots, e_n]$.
 - Array elements have values of primitive types. (Can't have an array of functions, for example.)
 - **No arrays of arrays** (unlike C, C++, Java).
 - Arrays will be zero-origin and have a fixed size ≥ 0 .
 - (Size zero is for an empty array; usually uninteresting.)
 - On integers: +, -, *, /, min, max, %, =, ≠, <, ≤, >, ≥, divides
 - On booleans: ¬, ∧, ∨, →, ↔, =, ≠ (note = and ↔ mean the same thing).
 - **If you don't have symbols in your type font**, let's use all for \forall , some for \exists , in for \in , ! for ¬, and for ∧, or for ∨, -> for →, <-> for ↔, != for ≠.
- **Conditional expressions:** $(B ? e_1 : e_2)$ as in C or Java etc., where B is a boolean expression and e_1 and e_2 have the same simple type. (Can't be arrays or functions, e.g.)

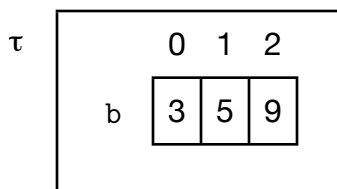
- May also write this as **if** B **then** e_1 **else** s_2 **fi**.
- **We don't have:** Assignment expressions, pointers, records, arrays of arrays.
- **Notation:** c and d are constants; e and s are general expressions; B and C are boolean expressions, a and b are arrays, x, y , etc. are variables, and u and v stand for variables.
- No explicit declarations of variables; just assume we know what they are and what their types are. (E.g., x must be an integer in $x+2$.)
- **Examples of expressions:**
 - $z+(x<0 ? x+y : x*y)$ [if $x<0$ then this whole thing equals $z+(x+y)$; if not, this whole expression equals $z+(x*y)$.]
 - $(i<0 ? 0 : a[i])$
 - $a[(i<0 ? 0 : (i\geq n ? n-1 : i))]$
- **Examples of non-expressions:**
 - A (conditional) expression can't yield a function, so
 - OK: $(x>1 ? \min(t,u) : \max(t,u))$
 - BAD: $(x>1 ? \min : \max)(t,u)$
 - No array-valued expressions, so
 - OK: $(x ? a[0] : b[0])$
 - BAD: $(x ? a : b)[0]$

D. States of Memory

- The value of an expression or the truth of a predicate can depend on the values of the variables.
 - A (**memory**) **state** (typically σ, τ) specifies values of variables.
 - Technically, a state is a function from variables (considered as pieces of text) to semantic values. E.g., $\sigma = \{("x", 1), ("y", F)\}$ for the memory state we'd usually draw on the blackboard as



- (The double quotes are annoying, so we'd write $\sigma = \{(x, 1), (y, F)\}$.)
- For an array variable, a state maps the variable to a function from indexes to values.
- E.g., $\tau = \{(b, \alpha)\}$ where $\alpha = \{(0, 3), (1, 5), (2, 9)\}$ means



(We could also write this as $\tau = \{(b, \{(0, 3), (1, 5), (2, 9)\})\}$).

- **Notation:** If the context is clear, let's abbreviate $\{(0, 3), (1, 5), (2, 9)\}$ to $(3, 5, 9)$, so we could abbreviate τ to $\{(b, (3, 5, 9))\}$. In general, (x_0, x_1, \dots, x_n) will stand for $\{(0, x_0), (1, x_1), \dots, (n, x_n)\}$.

E. Predicates as Standing for (Sets of) States

- When programming, we don't use states as defined above, we use predicates that stand for one or more states.
 - E.g., $x = 0 \wedge y = F$ stands for the state $\{(x, 0), (y, F)\}$.
 - Any predicate equivalent to this one stands for the same state. (E.g., $x = 1-1 \wedge \neg y$.)
- **Definition:** A **standard state predicate** is one of the form $variable = constant \wedge variable = value \wedge \dots$. E.g., $x = 0 \wedge y = F$ is standard but $x = 1-1 \wedge \neg y$ is not. We'll also allow T (true) and F (false) as standard state predicates.
 - Each subpredicate of the form $variable = constant$ is a **binding** — a standard state predicate is the conjunction of some bindings (or is T or F).
 - We'll use associativity and commutativity of \wedge without worrying about it, so we'll consider $x = 0 \wedge y = F$ and $y = F \wedge x = 0$ as being the "same" state.
 - Note a variable might be an array indexing expression (with a constant index). E.g., $b[0] = 3 \wedge b[1] = 5 \wedge b[2] = 9$. **Notation:** We'll write $b = (c_0, c_1, \dots, c_n)$ as an abbreviation for the standard state predicate $b[0] = c_0 \wedge b[1] = c_1 \wedge \dots \wedge b[n] = c_n$. So we could abbreviate $b[0] = 3 \wedge b[1] = 5 \wedge b[2] = 9$ to $b = (3, 5, 9)$.
- **Notation.** In general, if a state is needed, we'll allow ourselves to write a standard state predicate for that state. (We'll allow ourselves to be a little loose with the notation for states.)
- **Definition:** A state is **well-formed** or **proper** for an expression if it defines type-correct values for all the variables of the expression. Similarly, we'll say a state is proper for a predicate if it defines type-correct values for all the free variables of the predicate. E.g., assuming x and y are integer variables, the state $x = 0 \wedge y = 1$ is proper but the state $x = (2, 4)$ is improper for two reasons: It gives x an array value, and it doesn't define a value for y . Note that for predicates, it doesn't matter if the predicate doesn't actually need the value of a variable — the state must still define a type-correct value. E.g., $x = 0 \wedge y = 1$ is proper for $x = (T ? 17 : y)$ but just $x = 0$ is not.
- In general, when we talk about states, **we'll ignore the improper ones**. E.g., if we say "Let X and Y be the set of states in which $x = y+1$ is true and false respectively", then all states in X and Y are proper and $X \cup Y$ = the set of all (proper) states.

F. Values of Expressions

- Expressions have values relative to standard states. E.g., relative to $x = 1 \wedge b = (2, 0, 4)$, the expression $b[b[x]]$ has the value 2.

- We get the value by recursively evaluating subexpressions and combining their values using operators or functions. The base cases are variables (we get their values from the state) and constants (which have values independent of any state).
- To get the value, we simplify by substituting values for variables, applying operations on constants, and looking up array values given constant indexes.
 - E.g., in state $\sigma \equiv x = 1 \wedge b = (2, 0, 4)$, we have $b[b[x]] = b[b[1]] = b[0] = 2$.
 - In $\sigma \wedge y = 1$, $(x = (T ? 17 : y)) = (1 = (T ? 17 : y)) = (1 = 17) = F$.
 - (Since the test in $(T ? 17 : y)$ made us evaluate the true branch 17 and not the false branch y , we don't have to evaluate y .)
- **Notation:** $\sigma(e)$ is the **value of expression** e in state σ . E.g., above, we had $\sigma(b[b[x]]) = 2$ and $(\sigma \wedge y = 1)(x = (T ? 17 : y)) = F$.

G. Truth (Satisfaction and Validity) of Predicates (to be covered Mon Jan 23)

- **Definition:** A predicate p is **satisfied in state** σ if it evaluates to true in state σ . (Note σ must be proper for p .) We also say σ **satisfies** p .
 - E.g., $x = y \wedge y = 0$ is satisfied in $x = 0 \wedge y = 0$; it's not satisfied in $x = 0 \wedge y = 2$.
- **Notation:** $\sigma \models p$ means σ satisfies p . (The " \models " symbol is a "double turnstile".) Similarly, $\sigma \not\models p$ means σ (is proper but) does not satisfy p .
- One reason to be interested in satisfaction is that our program specifications will only tell us things about state/program pairs where the state initially satisfies some predicate. (E.g., "if $x > 0$, then after assigning $x - 1$ to y , we know $y \geq 0$ ".)
- **Definition:** A predicate p is **valid** if it is satisfied in all states: For all σ , $\sigma \models p$.
 - E.g., $x = x + 0$ is valid. So is $\forall x : x > x + 1$ (where x ranges over integers).
- **Note:** If p is not valid, that means there are one or more states in which it is not satisfied. That's different from saying it's not satisfied in any state. So $x^2 > x$ is not valid because it's not satisfied in $x = 0$ (or $x = 1$ or $x = -1$).