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*** $i \geq size(b)$ ***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*** $i \geq size(b)$ ***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 ≡ symbol is for syntactic equality, so both z and 2+2 are syntactic.
  - ***Example 9:*** In “σ(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.

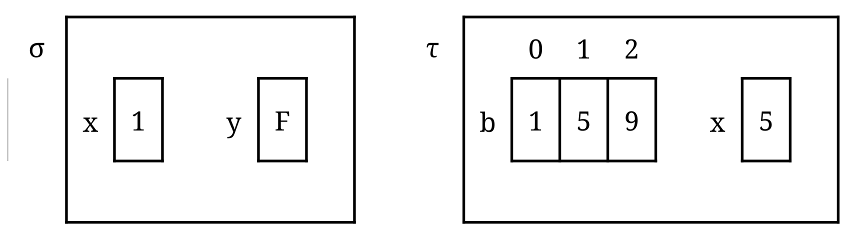
F. Semantic Values and Values of Expressions

- ***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 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 ≡ (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 ≤, =, 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 σ(*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 τ(***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* τ(***if*** $y > 0$ ***then*** *17* ***else*** y ***fi***) = τ(*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 τ(***if*** $y > 0$ ***then*** *17* ***else*** y ***fi***) = $\underline{17}$.
  - For the overall expression, we're comparing $\tau(x)$ and τ(***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 ∅ is a state (the empty state). It's proper for any expression or predicate that doesn't include variables. E.g., In state ∅, 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 σ = ∅ (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 $\sigma(v+w) = \sigma(v)$ $\underline{plus}$ $\sigma(w)$; here, $\underline{plus}$ 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 $\sigma = \{x = 1, y = \text{F}\} = \{(x, 1), (y, \text{F})\}$. 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 $\tau(b[0])$ = 3, $\tau(b[1])$ = 5, and $\tau(b[2])$ = 9. We could write $\tau = \{b[0] = 3, b[1] = 5, b[2] = 9, x = 5\} = \{(b[0], 3), (b[1], 5), (b[2], 9), (x, 5)\}$, 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 $\tau = \{b = \beta, x = 5\} = \{(b, \beta), (x, 5)\}$. (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
  - $\tau = \{b = \beta, x = 5\}$ where β = (3, 5, 9) — A sequence
  - $\tau = \{b[0] = 3, b[1] = 5, b[2] = 9, x = 5\}$ — A set of individual bindings
  - $\tau = \{b = \beta, x = 5\}$ where β = {(0, 3), (1, 5), (2, 9)} — A set of ordered pairs
  - $\tau = \{b = \beta, x = 5\}$ 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:
- $\sigma(b[e]) = \beta(\alpha)$ where $\beta = \sigma(b)$ and $\alpha = \sigma(e)$. The variable b is an array name, so $\sigma(b)$ = a function we're calling β. We call β on the ***value*** of the index expression e, hence $\alpha = \sigma(e)$, and the value β(α) is the meaning of $b[e]$.
- You can also write $\sigma(b[e]) = (\sigma(b))(\sigma(e))$ if you don't want to define α and β. Function application is left-associative, so $\sigma(b)(\sigma(e)) = (\sigma(b))(\sigma(e))$. I.e., $\sigma(b)$ is a function we're applying to $\sigma(e)$.
- So another way to write the definition is $\sigma(b[e]) = \sigma(b)(\sigma(e)) = \beta(\alpha)$ where $\beta = \sigma(b)$ and $\alpha = \sigma(e)$.
- With our earlier example then, $\sigma(b[x\text{-}4]) = \sigma(b)(\sigma(x\text{-}4))$ = 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 $\sigma = \{x = 1, b = \alpha\}$ where α = (2, 0, 4). Then
  - $\sigma(x) = 1$
  - $\sigma(x+1) = \sigma(x) + \sigma(1) = 1+1 = 2$
  - $\sigma(b) = \alpha$
  - $\sigma(b[x+1]) = (\sigma(b))(\sigma(x+1)) = \alpha(2) = 4$
  - If we don't want to write out the intermediate steps first, we could write
    - $\sigma(b[x+1]) = (\sigma(b))(\,\sigma(x+1)\,) = \alpha(\underline{\sigma(x)+1}) = \alpha(\underline{1+1}) = \alpha(\underline{2}) = \underline{4}$.
- ***Example 16***: Let $\sigma = \{x = 1, b = \alpha\}$ where α = (2, 0, 4), then
  - $\sigma(b[x+1]\text{-}\,2) = \sigma(b[x+1]) - \sigma(2) = (\sigma(b))(\sigma(x+1)) - \underline{2}$
    $= (\sigma(b))(\underline{\sigma(x)+1}) - \underline{2}$

$= \alpha(\underline{1+1}) - \underline{2}$
$= \underline{\alpha(2)-2} = \underline{4-2} = \underline{2}$.