Types, Expressions, and Arrays

CS 536: Science of Programming, Fall 2019

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 lecture, 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.
  - **Structured** variables: We have arrays. (No records or pointers.) For a 1-dimensional array reference, the syntax is the usual `b`[*e*] where *e* is an integer expression. Arrays are zero-origin and fixed-size. (You can look up the size using `size(b)`.) There are some limitations on the use of arrays (see below).
- On integers: `+`, `-`, `*`, `/`, `min`, `max`, `%`, =, ≠, <, ≤, >, ≥, `divides`
- On booleans: ¬, ∧, ∨, →, ↔, =, ≠ (note = and ↔ mean the same thing).
- On arrays: `size`
- **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.) You can also write a conditional expression as `if` *B* `then` e_1 `else` e_2 `fi`.
  - The two expressions e_1 and e_2 must have the same type so that the whole conditional has a consistent type. (Sometimes this is called "balancing".)
- **We don't have**: Assignment expressions, pointers, records, arrays as values. Also, we don't explicitly declare variables; we'll assume we know or can infer their types. (E.g., `x` must be an integer in `x+2`.) The default datatype for a variable is integer.

- **Array Limitations**:
  - We only have arrays of primitive types of values (no arrays of functions, for example). Arrays indexes are zero-origin; `size(b)` gives the length of an array. The size can be zero. At runtime, an illegal index causes a runtime error.
  - We don't have arrays as values, so we can't assign an array to a variable, and we don't have expressions that yield arrays. You can pass an array as a function argument or parameter; e.g., `sort(b)` might be a function that sorts `b` in place.
  - Multi-dimensional arrays are allowed, but you can't take a slice of an array. E.g., if `b` has 2 dimensions, you can use `b[1][3]` as an expression but not just `b[1]` (since it would yield a 1-dimensional array). To get the length along each dimension, we can use `size1(b)`, `size2(b)`, etc..
  - You can use an array in two contexts: `b`[some index] and as a function argument or parameter (including a predicate function). We don't have array variables or array assignments or expressions of type array.
- **Example 1:** `(x < 0 ? x+y : x*y)+z` 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`." (Types: `x`, `y`, and `z` must all be integers.)
- **Example 2**: `(i < 0 ? 0 : a[i])` yields 0 if `i` is negative, otherwise it yields `a[i]`. (Types: `i` is an integer and `a` is an array of integers.) Note we avoid indexes< 0 but not indexes ≥ array size.
- **Example 3**: `(i < 0 ? b[0] : i ≥ size(b) ? b[size(b)-1] : b[i])` 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[i < 0 ? 0 : i ≥ size(b) ? size(b)-1 : i]` 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
  - **Example 5a**: `(x > 1 ? min(t, u) : max(t, u))` is legal
  - **Example 5b**: `(x > 1 ? min : max)(t, u)` is illegal
- **Example 6**: We can't have array-valued expressions, so
  - **Example 6a**: `(x ? a[0] : b[0])` is legal assuming `a` and `b` are one-dimensional arrays.
  - **Example 6b**: `(x ? a : b)[0]` is illegal
- **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.

Syntactic Values and Semantic Values

- 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 so on.
- In these notes I'll try to be consistent about the following notation, but I won't do them on the blackboard, and you don't have to do them on your homeworks etc.
- **Notation**: Writing something in `this fixed-width font` means the item is syntactic (an expression or statement, typically). E.g., "`sqrt(2)`" or "`2+2 = 4`". Writing a value out in words means the item is

semantic (a value, such as a mathematical number). I'll often use italics, for extra emphasis. So `17` is syntactic, *seventeen* is semantic, and 17 (not in fixed-width font) is ambiguous.

- The basic problem is that we're talking about the meanings of programs, so some of the things we talk about are syntactic and some are semantic. Many times, the context tells you which. E.g., `x` and *p* have to be syntactic items in "Does `x` occur in the predicate *p*?" If we write z ≡ `2+2`, then `2+2` must be an expression. In "the value of 2+2 in state σ is 2+2, which is 4", the first 2+2 must be syntactic; the second 2+2 (and the 4) must be semantic. (Only expressions have values in a state, and the values are semantic.). So I might actually write "the value of `2+2` in σ is *two plus two*, which is *four*."

D. Values of Expressions

- In general, expressions have values relative to a state. E.g., relative to {`x` = 1, `y` = 2}, the expression `x+y` has the value *3*. Recall that we write σ(`x`) for the value of the variable x and extend this to σ(e) for the value of the expression *e*.
- The value of σ(*e*) depends on what kind of expression *e* is, so we use recursion on the structure of e (recursively evaluate subexpressions; the base cases are variables and constants).
  - σ(`x`) = the value that σ binds variable `x` to
  - σ(*c*) = the value of the constant *c*. E.g., σ(2) = *two*. (Note σ is irrelevant here.)
  - $\sigma(e_1 + e_2) = \sigma(e_1)$ plus $\sigma(e_2)$ [and similar for –, *, etc.]
  - $\sigma(e_1 < e_2) =$ T iff $\sigma(e_1)$ is less than $\sigma(e_2)$ [similar for ≤, =, etc].
  - $\sigma(e_1 \wedge e_2) =$ T iff $\sigma(e_1)$ and $\sigma(e_2)$ are both = T [similar for ∨, etc].
  - $\sigma(B \texttt{ ? } e_1 \texttt{ : } e_2) = \sigma(e_1)$ if σ(*B*) = T
  - $\sigma(B \texttt{ ? } e_1 \texttt{ : } e_2) = \sigma(e_2)$ if σ(*B*) = F
  - We'll put off the case σ(`b`[*e*]), the value of the array indexing expression `b`[*e*] for just a bit until we look at the value of an array variable.
- **Example 7**: 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 8**: Using the same σ as in Example 7, let τ = σ ∪ {`y` = 1} (σ as in example 7) and *e* ≡ (`x` = (`y` > `0` `?` `17` `:` `y`)), then τ(*e*) = F:
  - τ(*e*) = T iff τ(`x`) = τ(`y` > `0` `?` `17` `:` `y`)
  - For the left side, τ(`x`) = σ(`x`) = 1
  - For the right side, τ(`y` > `0`) = T iff τ(`y`) > τ(`0`) iff one is greater than zero, which is true
  - So τ(`y` > `0` `?` `17` `:` `y`) = τ(`17`) = seventeen because τ(`y` > `0`) = T

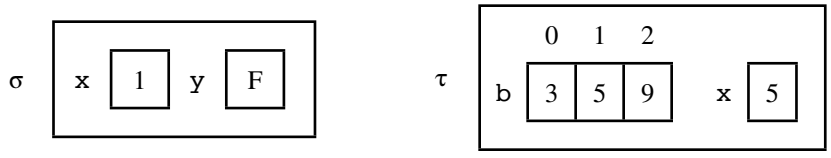
- So $\tau(e) =$ T iff one equals seventeen, which is false
- So $\tau(e) =$ F

- **Example 9**: Let $\sigma = \{x = 1, b = \alpha\}$ where $\alpha = (2, 0, 4)$, then
  - $\sigma(b[x+1]- 2) = \sigma(b[x+1]) - \sigma(2) = (\sigma(b))(\sigma(x+1))–2$
    $= (\sigma(b))(\sigma(x)+ 1))–2$
    $= \alpha(1+1)–2$
    $= \alpha(2)–2 = 4 – 2 = 2.$

- **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 10**: Let $\sigma = \emptyset$ (the empty state) then
  - $\sigma(2+2 = 4) = \sigma(2+2)$ equals $\sigma(4) = \ldots =$ four equals four = T.
- **The value of an expression has to be a semantic value**. So $\sigma(v+w) = \sigma(v)+ \sigma(w)$ is okay, as is $\sigma(v)$ plus $\sigma(w)$.
  - It's tempting to write things like $\sigma(v+w) = v+ w$ or v *plus* w, but these are errors. Since $\sigma(\ldots)$ is a semantic value, we can't write $\sigma(\ldots) =$ the expression v+w. Writing v *plus* w is even worse because it tries to run a semantic operation (addition) on two syntactic objects.

E. 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 $\tau(b[0]) = 3$, $\tau(b[1]) = 5$, 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 $\beta(0) = 3$, $\beta(1) = 5$, $\beta(2) = 9$, 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 $\beta = \{(0, 3), (1, 5), (2, 9)\}$. Since β is actually a sequence, let's allow ourselves to abbreviate this to $\beta = \{(3, 5, 9)\}$. (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 where $\beta = (3, 5, 9)$
  - $\tau = \{b[0] = 3, b[1] = 5, b[2] = 9, x = 5\}$
  - $\tau = \{b = \beta, x = 5\}$ where $\beta = \{(0, 3), (1, 5), (2, 9)\}$

- $\tau = \{b = \beta, x = 5\}$ where $\beta(0) = 3, \beta(1) = 5, \beta(2) = 9$

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 $\beta(\alpha)$ 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 $\alpha = \sigma(e)$.

Types, Expressions, and States

CS 536: Science of Programming, Fall 2019

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 today, you should

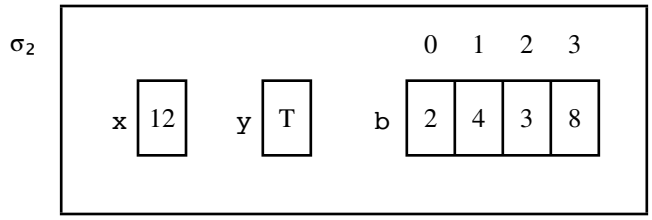
- Be able to read and write expressions we'll be using in our language.
- Be able to read and write states.
- Be able to evaluate an expression relative to a state.
- Be able to handle array names, array indexing expressions, and their values relative to a state.

C. Questions

1. Which of the following expressions are legal or illegal according to the syntax we're using? Assume `x`, `y`, `z` are integer variables and `b` is an array name.
   a. `(x > y ? x : y)` /* do you need assumptions as to the types of x and y */
   b. `(x < y ? -1 : (x = y ? 0 : 1))`
   c. `(y = 0 ? f : g)(17)`
   d. `b[0][1]` /* What type must b have for this to be legal? */
   e. `b` /* Remember we're given that b is an array */
   f. `f(b, b[0]) < 3` /* Also, if this is legal, what is the type of `f` ? */
   g. `(x < 3 ? x : F)`

2. Which of the following are legal ways to write out a state? (And if not, why not?)
   a. {x = 5, y = 2}
   b. {x = five, y = one plus one}
   c. {x = 5, y = x minus 3}
   d. {x = 5, y = α – 3} where α = 5
   e. {x = 5, y = (the value of x in this environment minus 3)}
   f. { }

3. Consider the state σ_2 described graphically below.



   a. Write a definition for $\sigma_2 = \{ \ldots \}$ using four ways described in section ***E: Arrays and Their Values***.

   b. Calculate $\sigma_2(e)$ where $e \equiv \texttt{y} \wedge \texttt{x} > \texttt{b[x/5]}$. Assume integer division truncates.

4. Let $\sigma_3 \equiv \{\texttt{z} = 4, \texttt{b}[0] = 1, \texttt{b}[1] = 5, \texttt{b}[2] = 8\}$.

   a. Abbreviate this using tuple notation for the value of `b` (i.e., $\texttt{b} = (\ldots)$).

   b. Write out the graphical representation of σ_3 (a memory diagram as in Problem 2).

   c. Calculate $\sigma_3(e)$ where $e \equiv \texttt{b[b[z-4]] > z ? z+2 : z-2}$. (Hint: Give names to parts of e and calculate the values of those parts first.)

5. Let $e_4 \equiv \texttt{x} = \texttt{y+1} \wedge \texttt{y} = \texttt{z}^2 - 3 \wedge \texttt{z} = 6$. Write out the textual definition of a state σ_4 in which e_4 evaluates to true. Use only bindings that map variables to constants. $\sigma_4 = \{ \text{x} = 34, \text{y} = 33, \text{z} = 6 \}$

6. Which of the following states are well-formed and also proper for the expression `b[i] + 0 * y`? If ill-formed, why? If taking the value might cause a runtime error, why?

   a. $\{\texttt{i} = 0, \texttt{b} = (3, 4, 8), \texttt{y} = 3, \texttt{z} = 5)$

   b. $\{\texttt{i} = 0, \texttt{b} = (6), \texttt{y} = 5)$

   c. $\{\texttt{i} = 0, \texttt{b} = 6, \texttt{y} = 5)$

   d. $\{\texttt{i} = 1, \texttt{b} = (3, 4, 8))$

   e. $\{\texttt{i} = 1, \texttt{i} = 2, \texttt{y} = 0, \texttt{b} = (2, 6))\}$

   f. $\{\text{i} = 5, \text{b} = (1, 2), \text{y} = 4\}$

CS 536 Solution to Activity 3 (Types, Expressions, and States)

1. (Legal and illegal expressions)
   a. `(x > y ? x : y)` is legal
   b. `(x < y ? -1 : (x = y ? 0 : 1))` is legal
   c. `(y = 0 ? f : g)(17)` is illegal because the conditional expression can't yield a function
   d. `b[0][1]` is legal (`b` must be a 2-dimensional array)
   e. `b` (all by itself) is illegal, since `b` we've assumed is an array
   f. `f(b, b[0]) < 3` is legal (the name `b` is being used as an argument to a function). We infer that `f` has type (int array) × int → int.
   g. `(x < 3 ? x : F)` is illegal because `x` and `F` have different types. (I.e., the expression doesn't have a fixed type because the types of its arms don't match.)

2. (Legal ways to represent states)
   a. {x = 5, y = 2} is legal
   b. {x = five, y = one plus one} is legal because "five" and "one" etc. refer to semantic objects.
   c. {x = 5, y = x minus 3} is illegal: To be legal, "x minus 3" has to be a value, so "x" has to be a value (it has to be the name of a mathematical object like 5). But the binding x = 5 tells us "x" is a variable that can appear in an expression, so "x" is a syntactic object. It can't be syntactic and semantic at the same time.

      Also, in this nicely word-processed document, "x" is presented in `this font`, so we know it's supposed to be a syntactic object. On paper, you see the difference between "x" and "x". Even so, if someone wrote { x = 5, y = x minus 1 } on the blackboard*, it would have to be illegal because of using X in two incompatible ways.
   d. {x = 5, y = α − 3} where α = 5 — is legal. We infer that symbols x and y are syntactic objects and α is the name of the semantic object 5.
   e. {x = 5, y = (the value of x in this environment, minus 3)} is legal. Since "the value of x in this environment" is just another name (albeit complicated) for the mathematical object 5, it's legal to use here.
   f. { } is legal, since it's just another way to write ∅, the empty state.

3. (Graphically defined state)
   a.. $\sigma_2 = \{b = \beta, x = 12, y = T\}$ where where $\beta = (2, 4, 3, 8)$.
   $\sigma_2 = \{b[0] = 2, b[1] = 4, b[2] = 3, b[3] = 8, x = 12, y = T\}$.
   $\sigma_2 = \{b = \beta, x = 12, y = T\}$ where $\beta = \{(0, 2), (1, 4), (2, 3), (3, 8)\}$.
   $\sigma_2 = \{b = \beta, x = 12, y = T\}$ where $\beta(0) = 2, \beta(1) = 4, \beta(2) = 3, \beta(3) = 8$.

---

* Using a felt marker, apparently :-)

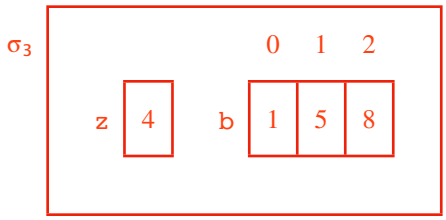
b. You can write out these kinds of calculations to different levels of detail, but a brief answer is that $\sigma_2(e) = \sigma_2(\texttt{y} \wedge \texttt{x} > \texttt{b[x/5]}) = \text{T} \wedge 12 > 3 = \text{T}$ You can certainly show intermediate steps:

$$\sigma_2(e) = \sigma_2(\texttt{y}) \wedge \sigma_2(\texttt{x}) > \sigma_2(\texttt{b})(\sigma_2(\texttt{x/5}))$$
$$= \text{T} \wedge 12 > \sigma_2(\texttt{b})(12/5)$$
$$= \text{T} \wedge 12 > \sigma_2(\texttt{b})(2) = \text{T} \wedge 12 > 3 = \text{T}$$

4. (Alternative ways to represent a state with an array value)

a. $\sigma_3 = \{\texttt{z} = 4, \texttt{b} = (1, 5. 8)\}$

b.



c. We have $e \equiv \texttt{b[b[z-4]] > z ? z+2 : z-2}$. To make this easier to deal with, let's break it down. Let $e \equiv e_1 \texttt{ > z ? z+2 : z-2}$ where $e_1 \equiv \texttt{b}[e_2]$ and $e_2 \equiv \texttt{b[z-4]}$.

- First, $\sigma_3(e_2) = \sigma_3(\texttt{b[z-4]}) = (\sigma_3(\texttt{b}))(\sigma_3(\texttt{z-4})) = (\sigma_3(\texttt{b}))(4–4) = (\sigma_3(\texttt{b}))(0) = 1$
- So $\sigma_3(e_1) = \sigma_3(\texttt{b}[e_2]) = (\sigma_3(\texttt{b}))(\sigma_3(e_2)) = (\sigma_3(\texttt{b}))(1) = 5$
- Then $\sigma_3(e_1 > \texttt{z}) = (\sigma_3(e_1) > \sigma_3(\texttt{z})) = 5 > 4 = \text{F}$
- So $\sigma_3(e) = \sigma_3(e_1 \texttt{ > z ? z+2 : z-2}) = \sigma_3(\texttt{z+2})$ because the test $\sigma_3(e_1 > \texttt{z}) = \text{F}$
- So finally, $\sigma_3(e) = \sigma_3(\texttt{z+2}) = \sigma_3(\texttt{z}) + \sigma_3(2) = 4+2 = 6$

5. $\sigma_4 = \{\texttt{z} = 6, \texttt{y} = 33, \texttt{x} = 34\}$

6. (Proper states)

a. Proper: The extra binding for `z` isn't a problem

b. Proper: The value of `b` is an array of length 0.

c. Improper: The value of `b` can't be an integer.

d. Improper: We need a binding for `y` even though we're multiplying it by zero.
[So our semantics uses eager evaluation, not lazy evaluation.]

e. Illegal: We have two bindings for `i`.

f. Proper but causes a runtime error, since `b` has size 2.

