CSCI337
Organisation of Programming Languages

Formal Semantics 2:
Denotational and Axiomatic Semantics

Denotational Semantics

- Semantics described via functions.
- E.g. a function that maps an integer expression to its value.
  - Val : Expression $\rightarrow$ Value
- So
  - $Val\ (2+3*4) = 14$

  and
  - $Val\ ((2+3)*4) = 20$

Denotational Semantics

- The domain of a function such as *Val* is a syntactic space:
  - The set of expressions is a syntactic entity.
- The range of a function such as *Val* is a semantic space:
  - The set of integers is a semantic entity.
- In this context, '2+3*4' is said to denote 20.
- This is the origin of the term denotational semantics.

Denotational Semantics

- In many programming languages, a program can be regarded as something that receives input and produces output.
- Thus, the semantics of a program is a function mapping input to output.
- A semantic function for programs would be:
  - P : Program $\rightarrow$ {Input $\rightarrow$ Output}
- The semantic space is a set of functions, from Input to Output.

Denotational Semantics

- E.g. If p is the program:
  - ```
    int main()
    {   int x;
        scanf("%d", &x);
        printf("%d\n",x);
        return 0;
    }
    ```
- Here p denotes the identity function f from integers to integers: P(p) $= f$, where:
  - f : Integer $\rightarrow$ Integer is given by $f(x) = x$
    ```

Denotational Semantics

- The function symbol $\rightarrow$ is right associative so we can simplify functions such as:
  - P : Program $\rightarrow$ {Input $\rightarrow$ Output}

  to
  - P : Program $\rightarrow$ Input $\rightarrow$ Output
- This allows for a slightly simpler notation, particularly as semantic spaces are frequently functions.

Denotational Semantics

- A denotational semantics for a programming language consists of three elements:

1. a definition of the syntactic space on which the semantic functions act,
2. a definition of the semantic space consisting of the values of the semantic functions

and

3. a definition of the semantic functions themselves.

Syntactic Spaces

- In denotational semantics, we use a notation that is essentially the same as the abstract syntax used in operational semantics.
- The sets are listed first using capital letters to denote set elements.
- The grammar rules are then listed which recursively define the elements of the set.

Syntactic Spaces

- E.g. the syntactic spaces Number and Digit are specified as follows:
  - D: Digit
    N: Number
  - N → N D | D
    D → '0' | '1' | … | '9'
- A denotational definition views the spaces as sets of syntax trees specified by the grammar rules.
- Semantic functions are defined recursively on these sets, based on the structure of a syntax tree node.

Semantic Spaces

- These are the sets from which semantic functions take their values.
- They are sets like syntactic spaces but may have extra structure, depending on their use.
- E.g. the integers have the arithmetic operations "+", "–" and "*".
- Such extended spaces are called algebras.
- Strictly, these need formal specification but we can often get away with "well-known" spaces.

Semantic Spaces

- E.g. the semantic space of the integers:
  - Space v: Integer = {..., –2, –1, 0, 1, 2, ...}
  - Operations:

$$+: \text{Integer} \times \text{Integer} \to \text{Integer}$$
$$-: \text{Integer} \times \text{Integer} \to \text{Integer}$$
$$*: \text{Integer} \times \text{Integer} \to \text{Integer}$$

- We restrict ourselves to three operations because they are the only ones used in our sample language.
- The symbols v: indicate that the name v will be used to indicate an arbitrary member of the set.

Semantic Functions

- A semantic function is specified for each syntactic space.
- Thus the value function from the syntactic space Digit to the semantic space Integer is written:
  - $\boldsymbol{D}$: Digit $\rightarrow$ Integer
- The value of a semantic function is specified recursively on the trees of the syntactic spaces.
- This is done by giving a semantic equation for each grammar rule.

Semantic Equations

- E.g. the grammar rules for digits:
  - D → '0' | '1' | … | '9'

give rise to the syntax trees

and the semantic function ***D*** is defined as follows:

Semantic Equations

$$\mathcal{D}\left(\begin{array}{c}\text{D}\\ |\\ \text{'0'}\end{array}\right)=0,\quad \mathcal{D}\left(\begin{array}{c}\text{D}\\ |\\ \text{'1'}\end{array}\right)=1,\quad \ldots,\quad \mathcal{D}\left(\begin{array}{c}\text{D}\\ |\\ \text{'9'}\end{array}\right)=9$$

- This (revolting) notation is usually shortened to:
  - $\mathcal{D}[[\text{'0'}]]=0,\ \mathcal{D}[[\text{'1'}]]=1,\ \ldots,\ \mathcal{D}[[\text{'9'}]]=9$

  where the [[…]] notation indicates that the argument is the syntax tree with the listed argument(s) as child(ren).

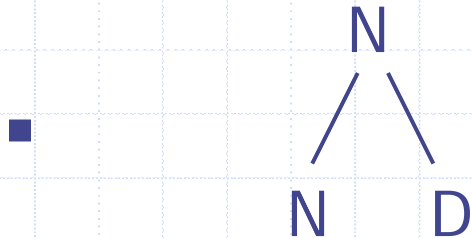
Another Example

- The semantic function
  - $\boldsymbol{N}$: Number $\rightarrow$ Integer

  from numbers to integers is based on the syntax
  - N $\rightarrow$ N D | D

  and is given by
  - $\boldsymbol{N}$[[ND]] = 10 * $\boldsymbol{N}$[[N]] + $\boldsymbol{N}$[[D]]
    $\boldsymbol{N}$[[D]] = $\boldsymbol{D}$[[D]]

Another Example

- The notation [[ND]] refers to the tree
  - 

  and [[D]] to the tree
  -

Denotational Semantics for Integer Expressions

- Using the expression language of last week and the approach described here:
- Syntactic Spaces
  - E: Expression
    N: Number
    D: Digit
  - $E \rightarrow E_1$ '+' E_2 | E_1 '–' E_2 | E_1 '*' E_2 | '(' E ')' | N
    $N \rightarrow$ N D | D
    $D \rightarrow$ '0' | '1' | … | '9'

Denotational Semantics for Integer Expressions

- Semantic Spaces
  - Space v: Integer = {..., –2, –1, 0, 1, 2, ...}
  - Operations:

$$+: \text{Integer} \times \text{Integer} \rightarrow \text{Integer}$$
$$-: \text{Integer} \times \text{Integer} \rightarrow \text{Integer}$$
$$*: \text{Integer} \times \text{Integer} \rightarrow \text{Integer}$$

Denotational Semantics for Integer Expressions

- Semantic Functions
  - $\boldsymbol{E}$: Expression $\rightarrow$ Integer
    $\boldsymbol{E}[[E_1 \text{ '+' } E_2]] = \boldsymbol{E}[[E_1]] + \boldsymbol{E}[[E_2]]$
    $\boldsymbol{E}[[E_1 \text{ '–' } E_2]] = \boldsymbol{E}[[E_1]] - \boldsymbol{E}[[E_2]]$
    $\boldsymbol{E}[[E_1 \text{ '*' } E_2]] = \boldsymbol{E}[[E_1]] * \boldsymbol{E}[[E_2]]$
    $\boldsymbol{E}[[\text{'('} \ E \ \text{')'}]] = \boldsymbol{E}[[E]]$
    $\boldsymbol{E}[[N]] = \boldsymbol{N}[[N]]$
  - $\boldsymbol{N}$: Number $\rightarrow$ Integer
    $\boldsymbol{N}[[ND]] = 10 * \boldsymbol{N}[[N]] + \boldsymbol{N}[[D]]$
    $\boldsymbol{N}[[D]] = \boldsymbol{D}[[D]]$
  - $\boldsymbol{D}$: Digit $\rightarrow$ Integer
    $\boldsymbol{D}[[\text{'0'}]]=0, \boldsymbol{D}[[\text{'1'}]]=1, \ldots, \boldsymbol{D}[[\text{'9'}]]=9$

Evaluation with Denotational Semantics

- Let's see how it works to evaluate the expression `'(2+3)*4'`
  - $\mathbf{E}$[['(' '2' '+' '3' ')' '*' '4']]
    = $\mathbf{E}$ [['(' '2' '+' '3' ')']] * $\mathbf{E}$[['4']]
    = ($\mathbf{E}$ [['2' '+' '3']]) * $\mathbf{N}$[['4']]
    = ($\mathbf{E}$ [['2']] + $\mathbf{E}$ [['3']]) * $\mathbf{D}$[['4']]
    = ($\mathbf{N}$ [['2']] + $\mathbf{N}$ [['3']]) * 4
    = ($\mathbf{D}$ [['2']] + $\mathbf{D}$ [['3']]) * 4
    = (2 + 3) * 4
    = 5 * 4
    = 20

Environments and Assignment

- Once again, if we add the concept of variables to our language, we must add environments to the semantics.
- This time, the set of environments is simply a new semantic space.
  - Space Env : Environment = Identifier $\rightarrow$ Integer $\cup$ {undef}
- The value undef is given a special name, bottom, and is denoted by the symbol $\perp$.
- We write Integer $\cup$ {undef} as $\text{Integer}_{\perp}$

Environments

- We can now define Env_0, the empty environment by
  - $Env_0(\mathrm{I}) = \bot$ for all identifiers I
- The evaluation of expressions within an environment must include the environment as a parameter.
- Thus the semantic value of an expression now becomes a function mapping environments to integers:
  - $\boldsymbol{E}$: Expression $\to$ Environment $\to$ $\mathrm{Integer}_{\bot}$

Environments

- The value of an identifier is its value in the environment:
  - $\boldsymbol{E}[[\mathrm{I}]](Env) = Env\ (\mathrm{I})$
- The value of a number is independent of the environment:
  - $\boldsymbol{E}[[\mathrm{N}]](Env) = \boldsymbol{N}[[\mathrm{N}]]$
- In other expression cases, the environment is simply passed on to the subexpressions.

Statements and Statement Lists

- To extend the semantics to these; we note that these are functions mapping environments to environments.
- Execution of a statement simply adds its value to the environment.
- We will use the same '&' notation we saw last week for adding to an environment.
- A statement list is simply the composition of the equivalent functions:

$$(f \circ g)(x) = f(g(x))$$

Denotational Semantics for the Augmented Language

- Syntactic Spaces
  - P: Program
    L: Statement-List
    S: Statement
    E: Expression
    N: Number
    D: Digit
    I : Identifier
    A: Letter

Denotational Semantics for the Augmented Language

- Abstract syntax
  - $P \rightarrow L$
    $L \rightarrow L_1$ ';' L_2 | S
    $S \rightarrow I$ ':=' E
    $E \rightarrow E_1$ '+' E_2 | E_1 '–' E_2 | E_1 '*' E_2 | '(' E ')' | N
    $N \rightarrow N\ D$ | D
    $D \rightarrow$ '0' | '1' | … | '9'
    $I \rightarrow I\ A$ | A
    $A \rightarrow$ 'a' | 'b' | …| 'z'

Denotational Semantics for the Augmented Language

- Semantic Spaces
  - Space v: Integer $= \{\ldots, -2, -1, 0, 1, 2, \ldots\}$
  - Operations:

    $+$: Integer $\times$ Integer $\rightarrow$ Integer

    $-$: Integer $\times$ Integer $\rightarrow$ Integer

    $*$: Integer $\times$ Integer $\rightarrow$ Integer
  - Space Env: Environment $=$ Identifier $\rightarrow$ $\text{Integer}_{\perp}$

Denotational Semantics for the Augmented Language

- Semantic Functions
  - $\boldsymbol{P}$: Program → Environment
    $\boldsymbol{P}$[[L]] = $\boldsymbol{L}$[[L]](Env_0)
  - $\boldsymbol{L}$: Statement-List → Environment → Environment
    $\boldsymbol{L}$[[L$_1$ ';' L$_2$]](Env) = $\boldsymbol{L}$[[L$_2$]] o $\boldsymbol{L}$[[L$_1$]](Env)
    $\boldsymbol{L}$[[S]](Env) = $\boldsymbol{S}$[[S]](Env)
  - $\boldsymbol{S}$: Statement → Environment → Environment
    $\boldsymbol{S}$[[I ':=' E]](Env) = Env & {I = $\boldsymbol{E}$[[E]](Env)}

Denotational Semantics for the Augmented Language

- $\boldsymbol{E}$: Expression → Environment → Integer
  $\boldsymbol{E}[[E_1 \text{ '+' } E_2]](Env) = \boldsymbol{E}[[E_1]](Env) + \boldsymbol{E}[[E_2]](Env)$
  $\boldsymbol{E}[[E_1 \text{ '–' } E_2]](Env) = \boldsymbol{E}[[E_1]](Env) - \boldsymbol{E}[[E_2]](Env)$
  $\boldsymbol{E}[[E_1 \text{ '*' } E_2]](Env) = \boldsymbol{E}[[E_1]](Env) * \boldsymbol{E}[[E_2]](Env)$
  $\boldsymbol{E}[[\text{'(' } E \text{ ')'}]](Env) = \boldsymbol{E}[[E]](Env)$
  $\boldsymbol{E}[[I]](Env) = Env(I)$
  $\boldsymbol{E}[[N]] = \boldsymbol{N}[[N]]$
- $\boldsymbol{N}$: Number → Integer
  $\boldsymbol{N}[[ND]] = 10 * \boldsymbol{N}[[N]] + \boldsymbol{N}[[D]]$
  $\boldsymbol{N}[[D]] = \boldsymbol{D}[[D]]$
- $\boldsymbol{D}$: Digit → Integer
  $\boldsymbol{D}[[\text{'0'}]]=0, \boldsymbol{D}[[\text{'1'}]]=1, \ldots, \boldsymbol{D}[[\text{'9'}]]=9$

Denotational Semantics for Control Statements

- To complete the discussion, we extend our semantics to if- and while- statements.
- Syntactic Space
  - S: Statement
    $S \rightarrow I$ ':=' E | 'if' E 'then' L_1 'else' L_2 'fi' |
    'while' E 'do' L 'od'
- We also need to extend the semantic function $\mathcal{S}$.

Semantic Function for if-statements

- $\mathcal{S}$: Statement → Environment → Environment
  $\mathcal{S}$[['if' E 'then' L_1 'else' L_2 'fi']] =
  if $\mathcal{E}[[E]](Env) > 0$ then $\mathcal{L}[[L_1]](Env)$ else
  $\mathcal{L}[[L_2]](Env)$

Where, given:

- F: Environment → Integer
  G: Environment → Environment
  H: Environment → Environment
- (if F then G else H)(Env) = G(Env), if F$(Env) > 0$
  H(Env), if F$(Env) \leq 0$

Semantic Function for while-statements

- This is more difficult.
- If we define F: Environment → Environment as:
  - $F(Env) = \mathcal{S}[[$'while' E 'do' L 'od'$]]$

then

  - $F(Env) = \text{if } \mathcal{E}[[E]](Env) \leq 0 \text{ then } Env \text{ else } F(\mathcal{L}[[L]](Env))$
- This is a recursive equation for F.

Semantic Function for while-statements

- To use this as a semantic function we must be sure that there exists a unique solution for F in some sense.
- How this termination is assured is beyond the scope of this subject.
- What about the loop:
  - i := 1 ;
    while i do i := i + 1 od

  which does not terminate?
- This cannot be assigned a function value.

Semantic Function for while-statements

- We define loops of this kind to have an undefined value.
- The semantic space of statements now becomes:
  - $\mathcal{S}$: Statement $\rightarrow$ Environment$_\perp$ $\rightarrow$ Environment$_\perp$

where

  - Environment$_\perp$ = (Identifier $\rightarrow$ Integer$_\perp$)$_\perp$

Axiomatic Semantics

- Here we define the semantics of a program by describing the effect that its execution has on assertions made about the program.
- Assertions are of two kinds:
  - Pre-conditions

  and

  - Post-conditions

Pre- and Post-conditions

- E.g. given the statement:
  - **`x := x + 1`**

  we would expect that, whatever the value of x was before execution, its value after execution is one greater.
- This is equivalent to setting a pre-condition of $x = A$ and a post-condition of $x = A + 1$.
- We write this:
  - $\{x = A\}$ **`x := x + 1`** $\{x = A + 1\}$

Pre- and Post-conditions

- We can also write this as:
  - $\{x = A\}$
    `x := x + 1`
    $\{x = A + 1\}$
- A second example is:
  - **`x := 1 / y`**

  where the pre-condition is that y be non-zero and the post-condition is that $x = 1/y$.
  - $\{y \neq 0\}$ **`x := 1 / y`** $\{x = 1/y\}$

Pre- and Post-conditions

- We can extend this notion to more than a single statement:
  - $\{n \geq 1$ and for all i, $1 \leq i \leq n$, $a[i] = A_i\}$
    `sort-program`
    {sorted(a) and permutation(a, A)}
- We can often test such conditions at execution time.
- E.g.
  - The C assert mechanism.

Axiomatic Specification

- An axiomatic specification of the program construct C takes the form:
  - {P}C{Q}

  where P and Q are assertions.
- The semantics of this is that if P is true prior to execution of C then Q is true after execution.
- This specification is not unique.

Axiomatic Specification

- In general, given the assertion Q there are many possible assertions P with the property {P}C{Q}.
- E.g. For:
  - C = `x := 1/y`
  - Q = $x=1/y$

  then
  - P = $y \neq 0$ or $y > 0$ or $y < 0$

  are all appropriate pre-conditions

Weakest Pre-condition

- Where we have a range of possible pre-conditions it is possible to identify the most general or weakest assertion which will result in Q being true.
- We call this the weakest pre-condition and write:
  - $wp(C,Q)$
- In the x=1/y example, $wp(C,Q)$ is $y \neq 0$.

Weakest Pre-condition

- We can now redefine the property {P}C{Q} as follows:
  - {P}C{Q} iff P $\rightarrow$ *wp*(C,Q)
- Now, we define the axiomatic semantics of the language construct C as the function *wp*(C,Q) from assertions to assertions.
- This is called a predicate transformer in that it takes a predicate (assertion) as argument and returns a predicate (the weakest precondition) as result.

Examples of *wp*(C,Q)

- *wp*(‘**`x := 1 / y`**’, x=1/y) = {y ≠ 0}
- *wp*(‘**`x := x + 1`**’, x>0) = {x > -1}
- *wp*(‘**`x := x + 1`**’, x=A) = {x = A-1}

To completely determine the semantics of an assignment such as:

- **`x :=`** E

we need to specify *wp*(‘**`x :=`** E’, Q) for all post-conditions Q.

Properties of *wp*

- Law of the Excluded Miracle:
  - $wp(C, \text{false}) = \text{false}$
- Distributivity of Conjunction:
  - $wp(C, Q \text{ and } R) = wp(C, Q) \text{ and } wp(C, R)$
- Distributivity of Disjunction:
  - $wp(C, Q) \text{ or } wp(C, R) = wp(C, Q \text{ or } R)$
- Law of Monoticity
  - If $Q \rightarrow R$ then $wp(C, Q) \rightarrow wp(C, R)$

Axiomatic Semantics of the Sample Language

- Note, first, that the specification of the semantics of isolated expressions is not normally undertaken in axiomatic semantics.
- Essentially, the assertions of axiomatic semantics are statements about the side effects of language constructs.
- That is, they are statements about values of identifiers in the environment of the construct.

Axiomatic Semantics of the Sample Language

- The abstract syntax for which we will define *wp* is:
  - P → L
    L → L_1 ';' L_2 | S
    S → I ':=' E | 'if' E 'then' L_1 'else' L_2 'fi'
    | 'while' E 'do' L 'od'
- Syntax rules such as P → L and L → S do not need separate specification, since these rules state that the *wp* function for a program is the same as for its associated statement-list and that if a statement-list is a single statement then *wp* for the list is the same as that of the statement.

Axiomatic Semantics of the Sample Language

- Statement lists:
  - $wp(L_1 ; L_2, Q) = wp(L_1, wp(L_2, Q))$
- Assignment statements:
  - $wp(I := E, Q) = Q[E/I]$
- This introduces a new notation Q[E/I] which is defined to be the assertion Q in which all free occurrences of I are replaced with E.
- A free occurrence is one which is not subject to “for all” or “there exists”.

Free Variables.

- E.g. with
  - $Q = \{\text{for all } i, a[i] > a[j]\}$

  we can evaluate
  - $Q[1/j] = \{\text{for all } i, a[i] > a[1]\}$

  but
  - $Q[1/i] = Q$
- This is because j is free in Q but i is not.

An Example of *wp* for Assignments

- Consider the example:
  - *wp*('`x := x + 1 ;`', x > 0)

  here:
  - I = x
  - E = x+1
  - Q = {x > 0}

  so, from
  - *wp*(I := E, Q) = Q[E/I]

  we get
  - Q[E/I] = {x+1 > 0}
    = {x > -1}

  which is the result we obtained informally before.

Axiomatic Semantics for if-statements

- The weakest pre-condition for the if-statement in our sample language is defined as:
  - wp(if E then L_1 else L_2, Q) =
    (E > 0 $\rightarrow$ $wp(L_1, Q)$) and (E $\leq$ 0 $\rightarrow$ $wp(L_2, Q)$)
- This looks worse than it in fact is.

An Example if-statement

- Consider:
  - wp(if x then x := 1 else x := -1 fi, x=1)
- Here:
  - Q = {x=1}
  - E = x
  - L1 = x := 1
  - L2 = x := -1
  - $wp(C,Q) = (x > 0 \rightarrow wp(x{:=}1, x=1))$ and
    $(x \leq 0 \rightarrow wp(x{:=}-1, x=1))$

An Example if-statement

- This can be simplified:
  - wp(C,Q) = (x > 0 → wp(x:=1,x=1)) and
    (x < 0 → wp(x:=-1,x=1))
  - = (x > 0 → 1 = 1) and (x ≤ 0 → -1 = 1)
- Now:
  - (P → Q) is the same as (Q or not P)

  so:
  - (x > 0 → 1 = 1) = ((1 = 1) or not (x > 0)) = true

  and
  - (x ≤ 0 → -1 = 1) = ((-1 = 1) or not (x ≤ 0)) = not (x ≤ 0) = x > 0

An Example if-statement

- Substituting back:
  - wp(if x then x := 1 else x := -1 fi, x=1) =

    $= (x > 0 \rightarrow 1 = 1)$ and $(x \leq 0 \rightarrow -1 = 1)$

    $=$ true and $x > 0$

    $= x > 0$

  which is what we would expect.

Axiomatic Semantics for while-statements

- The while-statement **`while E do L od`** executes as long as E > 0.
- Once again, this recursive behaviour creates problems for formal semantics.
- We will give an inductive definition based on the number of times the loop executes.
- Let H_i(while E do L od, Q) mean that the loop executes i times and terminates in a state satisfying Q.

Axiomatic Semantics for while-statements

- Clearly:
  - H_0(while E do L od, Q) $=$ E $\leq$ 0 and Q

  and

  - H_1(while E do L od, Q)
    $=$ E $>$ 0 and *wp*(L, Q and E $\leq$ 0)
    $=$ E $>$ 0 and *wp*(L, H_0(while E do L od, Q))
- In general:
  - H_{i+1}(while E do L od, Q)
    $=$ E $>$ 0 and *wp*(L, H_i(while E do L od, Q))

Axiomatic Semantics for while-statements

- We can now define *wp* for while-statements as:
  - *wp*(while E do L od, Q)

  = there exists an i such that H_i(while E do L od, Q)
- Note that the loop must terminate for this to make sense.
- A non-terminating loop always has false as its *wp*.
- E.g.
  - *wp*(while 1 do L od, Q) = false for all L, Q.

Axiomatic Semantics and Correctness Proofs

- Axiomatic semantics were developed as a tool for proving the correctness of programs.
- Let us examine an example:
  - {x = X and y = Y}
`swapxy`
{x = Y and y = X}

Axiomatic Semantics and Correctness Proofs

- We assert that the program:
  - {x = X and y = Y}
    ```
    t := x ;
    x := y ;
    y := t ;
    ```
    {x = Y and y = X}

  is correct.
- To prove this we must compute *wp*(C,Q) and then show that $P \rightarrow wp(C, Q)$

Axiomatic Semantics and Correctness Proofs

- wp(**t:=x;x:=y;y:=t**, x=Y and y=X)

  = wp(**t:=x**, wp(**x:=y;y:=t**, x=Y and y=X))

  = wp(t:=x, wp(**x:=y**, wp(**y:=t**, x=Y and y=X)))

  =wp(t:=x, wp(**x:=y**, wp(**y:=t**, x=Y)
  and wp(**y:=**t,y=X)))

  =wp(t:=x,wp(**x:=y**, wp(**y:=t**, x=Y))
  and wp(**x:=y**, wp(**y:=**t,y=X)))

  = wp(**t:=x**, wp(**x:=y**, wp(**y:=t**, x=Y)))
  and wp(**t:=x**, wp(**x:=y**, wp(**y:=t**, y=X)))

Axiomatic Semantics and Correctness Proofs

- Now:
  - $wp($**t:=x**$, wp($**x:=y**$, wp($**y:=t**$, x=Y))) =$
    $wp($**t:=x**$, wp($**x:=y**$, x = Y)) =$
    $wp($**t:=x**$, y = Y) =$
    $\{y = Y\}$

  and
  - $wp($**t:=x**$, wp($**x:=y**$, wp($**y:=t**$, y=X))) =$
    $wp($**t:=x**$, wp($**x:=y**$, t = X)) =$
    $wp($**t:=x**$, t = X) =$
    $\{x = X\}$

Axiomatic Semantics and Correctness Proofs

- wp(**t:=x;x:=y;y:=t**, x=Y and y=X)
  = wp(**t:=x**, wp(**x:=y**, wp(**y:=t**, x=Y)))
  and wp(**t:=x**, wp(**x:=y**, wp(**y:=t**, y=X)))
  = {y = Y} and {x = X}
- In this case P = wp(C, Q) so the proof of P → wp(C, Q) is trivial.

Axiomatic Semantics and Correctness Proofs

- As can be seen from even this simple example, formal proof of program correctness is extremely complex.
- The prospect of proving the correctness of a 100,000-line program, even a particularly carefully written one, is terrifying.
- We can conclude that only trivial programs are provably correct.
- Does this imply that all correct programs are trivial?