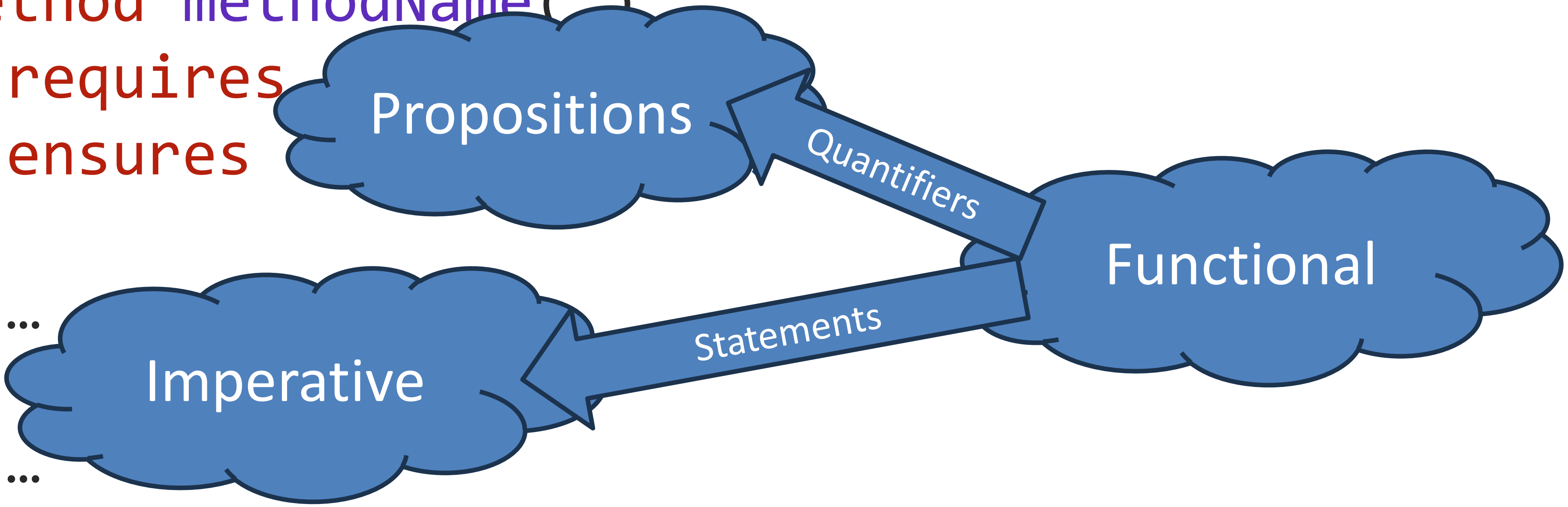
Program Specifications with Dafny

A General Pattern

```
method methodName( )
   requires
   ensures
{
   …
   …
}
```



Formal Specifications

- We use *preconditions / postconditions* to specify desired behavior of code
  - Precondition: assumptions for when the code starts executing
  - Postcondition: what must be satisfied when code terminates
- Both preconditions and postconditions will be specified in the language of propositions
- This kind of correctness is called *partial correctness*, as it doesn't require code termination
  - *Total correctness* imposes an extra termination requirement
  - We will talk about total correctness later

Formal Specifications in Dafny

- Given at the method level
  - Precondition: `requires` clause(s)
  - Postcondition: `ensures` clause(s)
- If either type of clause is missing: associated condition is assumed to be "true" (i.e. no restriction)

Dafny Formula Notation

```
||
&&
!
==>
exists x : T :: φ
forall x : T :: φ
```

- In `forall`, `exists` formulas, Dafny requires a type for x!
- Parentheses also allowed
- Boolean-valued expressions in Dafny programs can also be atomic predicates in formulas

Min

```
method Min (x : int, y : int) returns (min : int)
    ensures min <= x && min  <= y;
{
    if (x < y) {
        min := x;
    }
    else {
        return y;
    }
}
```

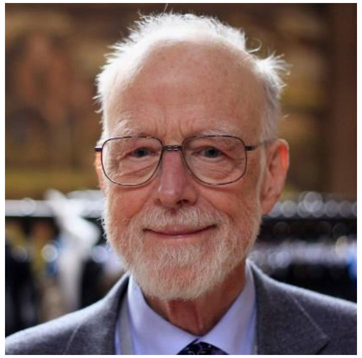
- `requires` clause is missing (so is assumed to be “true”, meaning all inputs are allowed)
- `ensures` clause states that output is ≤ both x and y

Min (cont.)

```
method Min (x : int, y : int) returns (min : int)
    ensures min <= x && min  <= y;
    ensures min == x || min == y;
{
    if (x < y) {
        min := x;
    }
    else {
        min := y;
    }
}
```

- The postcondition can actually be more precise!
- When there are two `ensures` (or `requires`) clauses they are assumed to be conjoined ("and-ed") together

Partial Correctness, Formally

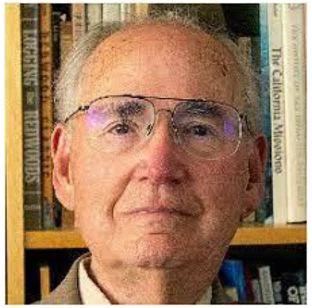


- Traditional (non-Dafny) notation: $\{P\}\ S\ \{Q\}$
  - P is the precondition
  - S is the code ("statement")
  - Q is the postcondition
  - Often $\{P\}\ S\ \{Q\}$ is called a *Hoare Triple*, after Tony Hoare (Turing Award 1980)
- $\{P\}\ S\ \{Q\}$ is *valid* if and only if:

  for every state for which P holds, if we execute statement S, we terminate in a state where Q holds
- In words: $\{P\}\ S\ \{Q\}$ is valid if and only if, when S is started in a state satisfying P and S terminates, the final state satisfies Q
- Validity of $\{P\}\ S\ \{Q\}$ = "S satisfies the precondition / postcondition specification corresponding to P and Q"

Partial Correctness (cont.)

- What does it mean for a program to “start in a state / terminate in a state”?
  - Answer: Semantics of programming languages!
  - Pioneered by Dana Scott (Turing Award 1976)
  - Other luminaries
    - Gordon Plotkin
    - Gilles Kahn
  - Take CMSC 631!



Programming Language Semantics

- Goal (imperative languages): interpret code as transformations from input states to result states
- Common approaches (S is code, Σ is the set of all states)
  - "Denotational": Define $[\![S]\!] \in \Sigma \to \Sigma$ (function from states to states)
    - Function is usually partial (i.e. not defined for all inputs)
    - Sometimes $[\![S]\!] \in \Sigma \to 2^{\Sigma}$ (i.e. returns sets of states, not single states, due to nondeterminism)
  - "Operational (Big-Step)": Define relation $\langle S, \sigma \rangle \Rightarrow \sigma'$
    - $\sigma, \sigma' \in \Sigma$
    - $\langle S, \sigma \rangle \Rightarrow \sigma'$ means S, starting in state σ, can terminate in state σ'
  - "Operational (Small-Step)": Define relation $\langle S, \sigma \rangle \to \langle S', \sigma' \rangle$
    - $\sigma, \sigma' \in \Sigma$, S' is code
    - $\langle S, \sigma \rangle \to \langle S', \sigma' \rangle$ means S, starting in state σ, can perform one execution step, with σ' being the new state and S' being the remaining code to execute.

Dafny Verifier

- Tries to prove validity of Hoare Triples for methods!
- How!
  - It constructs *annotations* of programs
  - An annotation puts a precondition in front of every statement and a postcondition after every statement
    - In Dafny: this can be done manually with `assert` statements
    - `assert` statements take a predicate-calculus formula as an argument
  - If it succeeds, i.e. if all the Hoare triples embedded in the annotated code are valid, the specification holds!
- The annotation method is often called the *intermittent invariant method*
  - Due to Bob Floyd (1967)
  - Floyd won Turing Award in 1978

Manually Annotated Min

```
method Min (x : int, y : int) returns (min : int)
    ensures min <= x && min  <= y
{
    assert true;
    if (x < y) {
        assert x < y;
        min := x;
        assert min < y;
    }
    else {
        assert x >= y;
        min := y;
        assert min <= x;
    }
    assert min <= x && min <= y;
}
```

More on Annotated Programs

- Annotation reflects "what you think is true" at the given points in code
  - If you are right: this proves precondition / postcondition!
  - If you are not right: annotation is incorrect, and is not a proof
- Dafny verifier:
  - Attempts to build annotations automatically
  - Tries to check if given annotations are indeed proofs

    If it cannot complete check, Dafny verifier complains

How Do You Build Annotations Systematically?

- *Strongest postconditions*
- *Weakest preconditions*
- *Loop invariants*

Strongest Postconditions

- If P is a precondition (so, a proposition) and S is code, then Q is the *strongest postcondition for P and S* if and only if:
  - $\{P\}\ S\ \{Q\}$ is valid
  - Q is the "most precise" among all postconditions Q' such that $\{P\}\ S\ \{Q'\}$ is valid

    "Most precise" means that if Q' is such that $\{P\}\ S\ \{Q'\}$ is valid, then $Q \Rightarrow Q'$
- Some facts
  - For traditional imperative languages: strongest postconditions always exist!
    - Regardless of form of P and S, strongest postcondition can be written down as a formula
    - Notation: $sp(P, S)$ used for strongest postcondition of P, S
  - $sp(P, S)$ can (often) be computed syntactically!

Computing $sp(P, S)$: Assignment

```
assert P;
x := 1;
assert Q;
```

What is the strongest Q?

Computing $sp(P, S)$: Assignment

- Suppose S is (Dafny) statement `x := 1`. What is $sp(P, S)$?
  - First guess: $P \wedge (x = 1)$
  - But this doesn't always work!
    - Suppose P is $x \neq 1$
    - This proposed definition would make $sp(P, S) = (x \neq 1 \wedge x = 1) \equiv \text{false}$
    - $\{P\}\ S\ \{sp(P, S)\}$ is not valid in this case!
- Problem:
  - P can mention the variable being assigned to
  - P might no longer be true after the assignment

Computing $sp(P, S)$: Assignment

- Another approach for $sp(P, S)$ when S is `x := 1`
  - Introduce a new variable u (not free in P) that represents the “old value” of x
  - Define $sp(P, S) = \exists u. (P[x \coloneqq u] \wedge x = 1)$
- Recall the previous example, where P is $x \neq 1$
  - $P[x \coloneqq u]$ is $u \neq 1$
  - Then $sp(P, S)$ is $\exists u. (u \neq 1 \wedge x = 1)$
  - This works!
  - Note that $\exists u. (u \neq 1 \wedge x = 1)$ can be simplified to $x = 1$ (why?)

Computing $sp(P, S)$: Assignment

```
assert P;
x := t;
assert ∃u. (P[x := u] ∧ x = t[x := u]);
```

Example

Suppose P is $x \geq 1$ and S is $x := x + 1$.

– In this case

$$sp(P,S) = \exists u.\,(P[x := u] \land x = (x+1)[x := u])$$
$$= \exists u.\,(u \geq 1 \land x = u + 1)$$

– This can be simplified to $x \geq 2$!

- Since $x = u + 1, u = x - 1$
- You can replace u by $x - 1$ in formula and simplify!

$$\exists u.\,(u \geq 1 \land x = u + 1) \equiv \exists u.\,(x - 1 \geq 1 \land x = x - 1 + 1)$$
$$\equiv (x \geq 2 \land x = x)$$
$$\equiv x \geq 2$$

Computing $sp(P, S)$: Statement Blocks

```
assert P;
s1; s2;
assert Q;
```

What is the strongest Q?

Computing $sp(P, S)$: Statement Blocks

- Suppose S is $S_1; S_2; \cdots S_n$; (like one would find in if-then-else or a loop body). What is $sp(P, S)$?
- Answer: chain them together!

That is, $sp(P, S) = Q_n$, where:

$$
\begin{aligned}
Q_1 &= sp(P, S_1) \\
Q_2 &= sp(Q_1, S_2) \\
&\vdots \\
Q_n &= sp(Q_{n-1}, S_n)
\end{aligned}
$$

Computing $sp(P, S)$: Statement Blocks

```
assert P;
s1;
assert sp(P,s1);
s2;
assert sp(sp(P,s1),s2);
```

Example

- Suppose P is $x \geq 1$ and $S = S_1; S_2;$, where S_1 is `x := x + 1` and S_2 is `x := x + 2`. What is $sp(P, S)$?

$$
\begin{aligned}
Q_1 &= sp(P, S_1) \\
&= \exists u. (u \geq 1 \wedge x = u + 1) \\
&\equiv x \geq 2 \\
Q_2 &= sp(Q_1, S_2) \\
&= \exists u. (u \geq 2 \wedge x = u + 2) \\
&\equiv x \geq 4
\end{aligned}
$$

- So $sp(P, S)$ is $x \geq 4$

Computing $sp(P, S)$: if-then-else

```
assert P;
if b {
    assert P ∧ b;
    s1;
    assert sp(P ∧ b, s1);
} else {
    assert P ∧ !b;
    s2;
    assert sp(P ∧ !b, s2);
}
assert sp(P ∧ b, s1) ∨ sp(P ∧ !b, s2);
```

Computing $sp(P, S)$: if-then-else

- Suppose $S = \texttt{if}\ B\ \texttt{\{}\ S'\ \texttt{\}}\ \texttt{else}\ \texttt{\{}\ S''\ \texttt{\}}$, where B is condition and S, S' are blocks of statements. What is $sp(P, S)$?
- To execute S:
  - Check if B is true, and if so, execute S'
  - If instead B is false, execute S''
- $sp(P, S)$ mimics this!
  - Suppose $Q_1 = sp(P \wedge B, S')$ and $Q_2 = sp(P \wedge \neg B, S'')$
  - Then $sp(P, S) = Q_1 \vee Q_2$!

Computing $sp(P, S)$: if-then-else

```
assert P;
if b {
    s1;
} else {
    s2;
}
assert Q;
```

What is the strongest Q?

Using sp To Generate Annotations

- Start from precondition, beginning of code
- Statement-by-statement, apply sp to (current) precondition and statement to generate postcondition for statement
- When you move from one statement to the next, use the postcondition of the previous statement as the precondition for the current one

Weakest Preconditions

- Strongest postconditions start from a precondition P and code S
- Weakest preconditions start from code S and postcondition Q!
  - If Q is a postcondition (so a proposition in Dafny) and S is code, then P is the *weakest precondition for S and Q* if and only if:
  - $\{P\}\ S\ \{Q\}$ is valid
  - P is the "most general" among all preconditions P' such that $\{P'\}\ S\ \{Q\}$ is valid

    "Most general" means that for all P' such that $\{P'\}\ S\ \{Q\}$ is valid, $P' \Rightarrow P$
- Some facts
  - For traditional imperative languages: weakest preconditions always exist!
    - Regardless of form of S and Q, weakest precondition can be written down as a formula
    - Notation: $wp(S, Q)$ used for weakest precondition of S, Q
  - Like $sp(P, S)$, $wp(S, Q)$ can (often) be computed syntactically!

Computing $wp(S, Q)$: Assignment

- Suppose S is `x := t`. What is $wp(S, Q)$?
  - For sp we needed to keep track of the old and new values of x
  - If we do the same for wp then we should introduce variable u for the new value of x
  - This would yield:
    $wp(S, Q) = \exists u. (Q[x \coloneqq u] \wedge u = t)$
- This can be simplified!
  - Since $u = t$, $\exists u. (Q[x \coloneqq u] \wedge u = t) \equiv \exists u. Q[x \coloneqq t]$
  - But now there is no u in $Q[x \coloneqq t]$, and $\exists u$ can be dropped!
  - So $\boldsymbol{wp(S, Q) = Q[x \coloneqq t]}$
  - No quantifier (i.e. $\exists u$) needed!
- Example
  - Suppose S is `x := x + 1`, Q is $x \leq 1$
  - Then $wp(S, Q) = Q[x \coloneqq x + 1] = x + 1 \leq 1 \equiv x \leq 0$

Computing $wp(S, Q)$: Statement Blocks

- Suppose S is $S_1; S_2; \cdots S_n;$
- $wp(S, Q)$ is computed like sp, but starting at the end of the block and working forward

$wp(P, S) = P_1$, where:

$$
\begin{aligned}
P_n &= wp(S_n, Q) \\
P_{n-1} &= wp(S_{n-1}, P_n) \\
&\vdots \\
P_1 &= sp(S_1, P_{n-1})
\end{aligned}
$$

Computing $wp(P, S)$: if-then-else

- Suppose $S = \texttt{if}\ B\ \texttt{\{}\ S'\ \texttt{\}}\ \texttt{else}\ \texttt{\{}\ S''\ \texttt{\}}$, where B is condition and S, S' are blocks of statements. What is $wp(S, Q)$?
  - Suppose we compute $P_1 = wp(S', Q)$, $P_2 = wp(S'', Q)$
  - This gives the preconditions under the assumption that B is true (P_1)and under the assumption that B is false (P_2)
  - So $wp(S, P) = (B \Rightarrow P_1) \wedge (\neg B \Rightarrow P_2)$!

Using wp To Generate Annotations

- Start from postcondition, end of code
- Working backwards, statement-by-statement, apply wp to (current) postcondition and statement to generate precondition for statement
- When you move from backward to the next statement, use the precondition of the just-processed statement as the postcondition for the current one