2.3 Weakest Preconditions

- We often have a postcondition ψ and a code S, and we need to know the *exact* circumstances ϕ in which executing S attains ψ.
- It turns out that we can *calculate* this precondition ϕ from the postcondition ψ and code S.

Definition 4 (Formula Strength)**.** A formula ϕ is *weaker* than another formula φ

iff ϕ allows all the program states allowed by φ and more besides

iff $\mathrm{Sts}(\varphi) \subsetneq \mathrm{Sts}(\phi)$

iff $\varphi \Longrightarrow \phi$ but not the other way around

iff φ entails more than ϕ.

Definition 5 (The wp Transformer)**.** The *weakest precondition* $\mathrm{wp}(S, \psi)$ is the weakest formula ϕ such that $\{ \phi \} S \{ \psi \}$ holds.

- Or viewed in another way,

  $$\{ \phi_{\mathrm{pre}} \} S \{ \phi_{\mathrm{post}} \} \quad \text{iff} \quad \phi_{\mathrm{pre}} \Longrightarrow \mathrm{wp}(S, \phi_{\mathrm{post}}). \tag{4}$$

  In this view, wp transforms a Hoare triple (and hence also the code within it) into a logical formula, which can then be verified.

- We talk of "the" weakest precondition, because it is unique as a set $\mathcal{P}$ of states.

  Of course it has several different but equivalent names: several different formulae $\phi, \phi', \phi'', \ldots$ with $\mathrm{Sts}(\phi) = \mathrm{Sts}(\phi') = \mathrm{Sts}(\phi'') = \cdots = \mathcal{P}$.

- Let us now argue the two *axioms* (Theorems 6 and 7) of wp.
  - Since we take them as our axioms, we must argue them semantically — that is, using states and execution.
  - Later we can derive new results logically from these axioms instead.

    In fact, the whole point of using logic is to avoid arguing semantically over and over again!
  - When we later give calculation rules for wp, we should check that they obey these axioms.

Theorem 6 (There Are No Miracles)**.** *There is no program code such that*

$$\mathrm{wp}(\ldots, \textsc{false}) \neq \textsc{false}.$$

Proof. Note that $\mathrm{Sts}(\textsc{false}) = \emptyset$. Hence the miraculous code would have to terminate normally in a state which does not even exist. Thus there is no state where it can be legitimately even started. □

Theorem 7 (wp Distributes over '$\wedge$')**.** $\mathrm{wp}(S, \psi) \wedge \mathrm{wp}(S, \psi') \Longleftrightarrow \mathrm{wp}(S, \psi \wedge \psi')$.

Proof. We have two cases to prove.

$\Longrightarrow$: Let $f \in \mathrm{Sts}(\mathrm{wp}(S,\psi)\wedge\mathrm{wp}(S,\psi'))$. Then both $f \in \mathrm{Sts}(\mathrm{wp}(S,\psi))$ and $f \in \mathrm{Sts}(\mathrm{wp}(S,\psi'))$. The first (second) means that if S is started in f, then it is guaranteed to terminate with ψ (ψ') true. Hence starting S in f is guaranteed to terminate with $\psi\wedge\psi'$ true: in symbols, $f \in \mathrm{Sts}(\mathrm{wp}(S,\psi\wedge\psi'))$.

Thus $\mathrm{Sts}(\mathrm{wp}(S,\psi)\wedge\mathrm{wp}(S,\psi')) \subseteq \mathrm{Sts}(\mathrm{wp}(S,\psi\wedge\psi'))$ as claimed.

$\Longleftarrow$: Let $f \in \mathrm{Sts}(\mathrm{wp}(S,\psi\wedge\psi'))$. Then starting S in f is guaranteed to terminate in a state f' where $\psi\wedge\psi'$ is true; that is, in a state f' where both ψ and ψ' are true. Hence f belongs both to $\mathrm{wp}(S,\psi)$ and to $\mathrm{wp}(S,\psi')$. Hence $f \in \mathrm{Sts}(\mathrm{wp}(S,\psi)\wedge\mathrm{wp}(S,\psi'))$. Then continue as above. $\square$

- Further properties of wp provable logically from these two axioms include monotonicity (Theorem 8) and one half of distributivity over '$\vee$' (Theorem 9).

Theorem 8 (wp is Monotonic)**.** *If* $\psi \Longrightarrow \psi'$ *then* $\mathrm{wp}(S,\psi) \Longrightarrow \mathrm{wp}(S,\psi')$.

Theorem 9 (wp Distributes over '$\vee$')**.**

$$\mathrm{wp}(S,\psi)\vee\mathrm{wp}(S,\psi') \overset{\Longrightarrow}{\underset{\Longleftarrow}{}} \mathrm{wp}(S,\psi\vee\psi')$$

where the forward direction holds for all program code S, but its inverse only for deterministic S.

Definition 10 (Deterministic Code)**.** Code S is deterministic if for every initial state f there exists at most one halting state f' in which S can terminate when its execution is started in f.

- Consider a *coin toss* as an example of nondeterminism: wp(*toss*, *heads*) is the weakest precondition which *guarantees* that the coin lands *heads* up, and so on. Hence

$$\begin{aligned}\mathrm{wp}(\mathit{toss},\mathit{heads}) &= \text{FALSE} \quad \text{and}\\ \mathrm{wp}(\mathit{toss},\mathit{tails}) &= \text{FALSE} \quad \text{but}\\ \mathrm{wp}(\mathit{toss},\mathit{heads}\vee\mathit{tails}) &= \text{TRUE}.\end{aligned}$$

- We did not assume in defining the Hoare triple (Definition 3) that all the programming code was deterministic.

- Instead, if some part of code *must* be deterministic (that is, if its correctness proof needs the inverse) then this part must be explicitly written so (for its proof to go through).

- Implementing the nondeterministic parts of the code with a real deterministic programming language poses no problems either:

  We defined the Hoare triple (Definition 3) to guarantee that *every* execution gets from ϕ_{pre} into ϕ_{post}, so *any* implementation will work.

- View (4) and these wp results let us verify logically two very general principles.
  - The first (Theorem 11) says that if program S works in a more general setting, then it (obviously) continues to work in a more restricted setting too.
  - The second (Theorem 12) says in turn that we can (obviously) forget some of the results that program S managed to accomplish, if we want to.

Theorem 11 (Strengthening the Precondition). *If $\psi \implies \phi_{pre}$ and $\{\ \phi_{pre}\ \}S\{\ \phi_{post}\ \}$, then $\{\ \psi\ \}S\{\ \phi_{post}\ \}$.*

Theorem 12 (Weakening the Postcondition). *If $\{\ \phi_{pre}\ \}S\{\ \phi_{post}\ \}$ and $\phi_{post} \implies \psi$, then $\{\ \phi_{pre}\ \}S\{\ \psi\ \}$.*

2.4 The Core Guarded Command Language

- Let us now define the commands of GCL, but leave subroutines for later.
- We shall define the meaning of a command S as the rule for computing $\mathrm{wp}(S,\phi)$ given ϕ, for any arbitrary postcondition ϕ.
- The distinguishing GCL feature is the *guarded command*

  $$\textit{guard} \rightarrow \textit{command}$$

  where the *guard* is a logical formula.

  - They appear as the contents of the **if** commands (Section 2.4.5) and **do** loops (Section 2.4.7).
  - This *command* can be executed only when its *guard* is TRUE.

    But due to nondeterminism (Definition 10), some other *command′* whose *guard′* is also TRUE can be executed instead.
  - Because the *guard* is tested during execution, it must be simple to execute: just a Boolean combination of atomic formulae over the program variables.
  - In particular, quantifiers are not allowed: if we want to check that $\forall 1 \leq i < N.\textit{something}$ holds, then we must write an explicit **do** loop over i for it.

- Guarded commands separate

  logic which is expressed with guards and just examines the current program state without modifying it

  actions expressed with commands to modify it.

- Real state-based programming languages do not enforce this separation: e.g. what is the state after the following code, if the function call TEST(x) returns FALSE?

  ```
  if TEST(x)
      then PRINT("ok");
  ```

  It *looks* like it would be exactly the same as before the **if**, but in fact running the TEST may have caused state-modifying actions (such as an assignment into a global variable) as a side effect!

  Hence the programmer should enforce the separation, even if the language does not!

- Let us now rewrite our running binary search example (Figure 1) in GCL (Figure 4). Later we will explain each of its constructs in detail.

```
$l, u$ := $1, N$;
do $l < u \rightarrow$
    $m$ := $(l + u) \operatorname{div} 2$;
    if $b \leq A[m] \rightarrow$
        $u$ := $m$
     [] $b > A[m] \rightarrow$
        $l$ := $m + 1$
    fi
od;
if $(u > 0)$ cand $(b = A[u]) \rightarrow$
    $r$ := $u$
 [] $(u = 0)$ cor $(b \neq A[u]) \rightarrow$
    $r$ := $0$
fi .
```

Figure 4: Our Binary Search in GCL.

2.4.1 Skipping

- The simplest program is one which terminates without doing anything. GCL has such a primitive command **skip**.
- Its wp definition is
$$\mathrm{wp}(\mathbf{skip}, \phi) = \phi \tag{5}$$
or "**skip** terminates with ϕ true exactly when it is started with ϕ already true".
- That is, wp(**skip**, ...) is the identity function on formulae.
- Later, **skip** will turn out to be useful in

  theory as the unit element for the '**;**' operator (Section 2.4.3)

  practice within the **if** command (Section 2.4.5).

2.4.2 Aborting

- Another very simple program is one which cannot terminate normally. GCL has such a primitive command **abort** too.
- Its wp definition is in turn
$$\mathrm{wp}(\mathbf{abort}, \ldots) = \text{FALSE} \tag{6}$$
or "there are no circumstances where starting **abort** would terminate normally".
- It would take a miracle (Theorem 6) to continue after **abort**ing.
- This **abort** stands for two kinds of executions:

  crashing in a run-time error (terminates, but not normally)

  running forever in a loop (does not even terminate).
- Later, **abort** will turn out to be useful in

theory as the zero element for the '`;`' operator (Section 2.4.3) and as a run-time error indicator, but *not* in

practice since we do not want to write it in our own code. Instead, we must design our guards to avoid it from happening.

2.4.3 Sequencing

- Our first *compound* command is $S_1; S_2$ where S_1 and S_2 are simpler commands, and it executes first S_1 followed by S_2.

- Its wp definition is
$$\text{wp}(S_1; S_2, \phi) = \text{wp}(S_1, \text{wp}(S_2, \phi)) \tag{7}$$
or "executing $S_1; S_2$ is guaranteed to make ϕ true exactly when executing S_1 is guaranteed to end in those circumstances, where executing S_2 is guaranteed to make ϕ true".

- Syntactically, GCL follows the Algol 60 and Pascal tradition where '`;`' is an *operator for joining* the two commands into one larger command.

  This kind of '`;`' is (confusingly) called a command *separator*.

- Current languages like C and Java interpret '`;`' as a *terminator* instead: "e`;`" turns the expression e into a command.

  Then there is no explicit command joining operator: a sequence of commands means that they are intended to be executed one after the other.

- Viewing '`;`' as a two-place operator allows us to see its mathematical properties better. E.g.:
$$\begin{gathered}
1 \cdot x = x = x \cdot 1 \\
\textbf{skip;}S = S = S\textbf{;skip} \\
0 \cdot x = 0 = x \cdot 0 \\
\textbf{abort;}S = \textbf{abort} = S\textbf{;abort} \\
x \cdot (y \cdot z) = x \cdot y \cdot z = (x \cdot y) \cdot z \\
S_1;(S_2;S_3) = S_1;S_2;S_3 = (S_1;S_2);S_3
\end{gathered}$$

- Here we can define the equality $S = S'$ between *programs* as
  - "the programs S and S' have exactly the same pre- and postcondition pairs"
  - "their meaning functions $\text{wp}(S, \dots)$ and $\text{wp}(S', \dots)$ are the same"
  - $\text{wp}(S, \psi) \iff \text{wp}(S', \psi)$ holds for every formula ψ.

  This does capture an elusive property nicely – but it is very hard to use for any but the simplest programs. . .

- In fact, one alternative to logic-based program verification and development would be to use an *algebraic* approach instead.
  - In this approach, such equivalences between program parts would be derived — forming an *algebra of programming*.

- Then verification and development would proceed by swapping in an equivalent program part, similarly to the specification refinement approach mentioned above (Section 3).
- This algrebraic approach is best conducted in a state-free functional programming language since such code can be considered directly as an algebraic expression.
- Here instead the equivalences are proved inside wp logic, and applied in the code, which is separate. This is more cumbersome.

(One source for this approach would be the book by Bird and de Moor (1997) which uses Haskell (Peyton Jones, 2003) as its programming language for the reason given above.)

2.4.4 Assignment

- The basic command to modify the current state is to assign a new value to one of its program variables.
- The GCL syntax is

$$x := \textit{expression}$$

where the

x is a program variable name

expression is built from the program variable names and the constants and names in $\mathcal{GCL}$.

- The semantic idea is to change the current state f into the state

$$f[x \leftarrow \underbrace{\mathrm{eval}_{\mathcal{GCL},f}(\textit{expression})}_{v}] \tag{8}$$

as follows:

1. First evaluate the value v of the *expression* in the current state f (Section 2.1).
2. Then assign this value v into x but keep f otherwise as it was (Definition 1).

- The corresponding wp semantics is

$$\mathrm{wp}(x := \textit{expression}, \phi) = \phi[x \leftarrow \textit{expression}] \tag{9}$$

where the operation $\phi[x \leftarrow \ldots]$ is *textual (syntactic) substitution:*

- On the *semantic* side, Eq. (8) had to produce the same kind of thing as f itself was — that is, another state f'.
- This meant that v had to be of a kind which could be stored in x to create f' — that is, a value.
- Similarly, here Eq. (9) has to produce the same kind of thing as ϕ — that is, another formula ϕ', which is basically *text*.
- This means that the *expression* in Eq. (9) has to be of a kind which can be substituted for x to create ϕ' — that is, a suitable part of a formula *text*.

Hence this operation is rqughly "replace every free occurrence of program variable x in the formula ϕ with the text of *expression* to get another formula ϕ'".

- We choose a similar notation for Eqs. (8) and (9), even though they are distinct operations, to emphasize that the latter arises as the syntactic counterpart to the former semantic one.

- We must take care that this textual substitution $\phi[x \leftarrow \mathit{expression}]$ does not violate the meanings of the quantifiers in formula ϕ.

  A programmer's intuition is that $\phi[x \leftarrow \mathit{expression}]$ must respect the variable scoping rules of ϕ.

- The precise definition of $\phi[x \leftarrow \mathit{expression}]$ by induction/recursion over the structure of ϕ must handle the following kinds of cases:
  - The result of
  $$\underbrace{(\overset{\forall}{\underset{\exists}{}}x.\psi)}_{\phi\text{ itself}}[x \leftarrow \mathit{expression}]$$
  is ϕ itself, without any changes.

    This is because the free occurrences of the variable name x within ψ denote this quantified variable x, and not the free x which we are replacing by the *expression*.
  - The result of
  $$(\overset{\forall}{\underset{\exists}{}}y.\psi)[x \leftarrow \mathit{expression}]$$
  where the variable name y is not x but y occurrs also in the *expression*, is in turn
  $$(\overset{\forall}{\underset{\exists}{}}z.\psi[y \leftarrow z])[x \leftarrow \mathit{expression}]$$
  where z is a new variable name which does not occurr in ψ.

    That is, we first *rename* this quantified y to some other z in ψ. The occurrences of y in ψ namely denoted this "local" quantified variable, while the occurrences of y denoted some other "global" variable with the same name. Renaming the local y to z avoids getting confused about which is which.
  - Otherwise the result of
  $$(\overset{\forall}{\underset{\exists}{}}z.\psi)[x \leftarrow \mathit{expression}]$$
  just passes the replacement to the subformula ψ:
  $$(\overset{\forall}{\underset{\exists}{}}z.\psi[x \leftarrow \mathit{expression}])$$
  since this quantification mentions neither this variable name x nor its replacement text *expression*.

  - Similarly, if the formula is anything else than a quantification, the replacement is just passed to its subformulae: E.g. the result of

    $$(\psi \land \varphi)[x \leftarrow \mathit{expression}] \quad \text{is} \quad (\psi[x \leftarrow \mathit{expression}]) \land (\varphi[x \leftarrow \mathit{expression}])$$

    and so on.

  We can avoid such problems, if we never reuse program variable names as quantified variables in our formulae.

- Note that the wp semantics of Eq. (9) are "backwards" to the execution order:

  1. The given postcondition ϕ tells us what properties the new value of the program variable x must satisfy just *after* the assignment has taken place.
  2. Then the calculated precondition ϕ' tells us that the assigned *expression* must have satisfied exactly these same properties just *before* the assignment has taken place.

     Note that we reasoned like this in Step 5 of our earlier example (Section 2.2).

- E.g. "When is $x \in \mathbb{N}$ guaranteed to be odd after executing the assignment $x := y - 1$?"

$$\begin{aligned}
\mathrm{wp}(x := y - 1, &\overbrace{(\exists z \colon \mathbb{N}. x = 2 \cdot z + 1)}^{\text{definition of “}x \in \mathbb{N}\text{ is odd”}}) \\
&= (\exists z \colon \mathbb{N}. x = 2 \cdot z + 1)[x \leftarrow y - 1] \\
&= (\exists z \colon \mathbb{N}. y - 1 = 2 \cdot z + 1) \\
&= (\exists z \colon \mathbb{N}. y = 2 \cdot (z + 1)) \\
&= (\underbrace{(\exists u \colon \mathbb{N}. y = 2 \cdot u)}_{\text{“}y \in \mathbb{N}\text{ is even”}} \land (y \geq 2)).
\end{aligned}$$

Ghost Variables

- We have earlier mentioned *ghost* variables (Section 2.1) to keep track of the previous value of a variable.

- For example, consider swapping the contents of variables x and y using a *temp*orary variable:

  $$\mathit{temp} \;:=\; x\,;\;\; x \;:=\; y\,;\;\; y \;:=\; \mathit{temp}$$

- The desired outcome is that "the *new* value of x is the *old* value of y and vice versa".

- Stating (and hence also proving) this desired outcome needed a way to refer to these old and new values.

- We adopt the notation that program variable names are (or start with) *lower*case letters, while the corresponding ghost variables are *Capitalized.* (If we should ever need more than one simultaneous ghost variable for the same program variable, then we add subscripts.)

  Hence e.g. x is a program variable and X is its corresponding ghost variable (as are also $X_0, X_1, X_2, \ldots$ if we ever need them).

- Now we can state the desired outcome as $x = Y \wedge y = X$ and calculate how the reach it:

$$\begin{aligned}\mathrm{wp}(\mathit{temp} := x; x := y; y := \mathit{temp}, x = Y \wedge y = X) = \\ \underbrace{\mathrm{wp}(\mathit{temp} := x, \underbrace{\mathrm{wp}(x := y, \underbrace{\mathrm{wp}(y := \mathit{temp}, x = Y \wedge y = X)}_{x=Y\wedge \mathit{temp}=X})}_{y=Y\wedge \mathit{temp}=X})}_{y=Y\wedge x=X}\,.\end{aligned}$$

- This calculation validates the desired Hoare triple:

$$\begin{array}{l}\{\, y = Y \wedge x = X \,\} \\ \mathit{temp} \;:=\; x\,;\;\; x \;:=\; y\,;\;\; y \;:=\; \mathit{temp} \\ \{\, x = Y \wedge y = X \,\}\end{array}$$

- Reading aloud the implicit '$\forall$' yields "for all X, Y, if $y = Y \wedge x = X$ holds before executing the swap, then $x = Y \wedge y = X$ holds afterwards".

- This reading makes intuitive sense (as it should), but as noted before, we lack formal means of saying just how far this '$\forall$' extends in our code.

Aliasing

- Treating assignment as textual substitution precludes *aliasing* where two distinct program variables x and y both *point to* the same *shared item* which has a *mutable field* (Figure 5).

- Then assignment to this $x.\mathit{field}$ also modifies $y.\mathit{field}$ invisibly without mentioning y explicitly at all.

- Textual substitution cannot see this:

$$\mathrm{wp}(x.\mathit{field} := 5, x.\mathit{field} = y.\mathit{field}) = (5 = y.\mathit{field}) \tag{10}$$

whereas it should have been

$$\begin{aligned} &= (5 = 5). \\ &= \text{TRUE}. \end{aligned}$$

- To make matters worse, Eq. (10) *is* the right answer when x and y point to *different* items!

- Because of this, GCL variables contain *only data values, not pointers* to other data values.

- If GCL had aliasing/pointers, how would we define the exact meaning of the assignment command, because it must say (implicitly) also what variables are *not changed*?

- Moreover, such a definition would have to mention more than just the source code itself:

  Whether y aliases x or not at this point of the code can depend also on *how* the computation reached this point — and ultimately on the inputs.

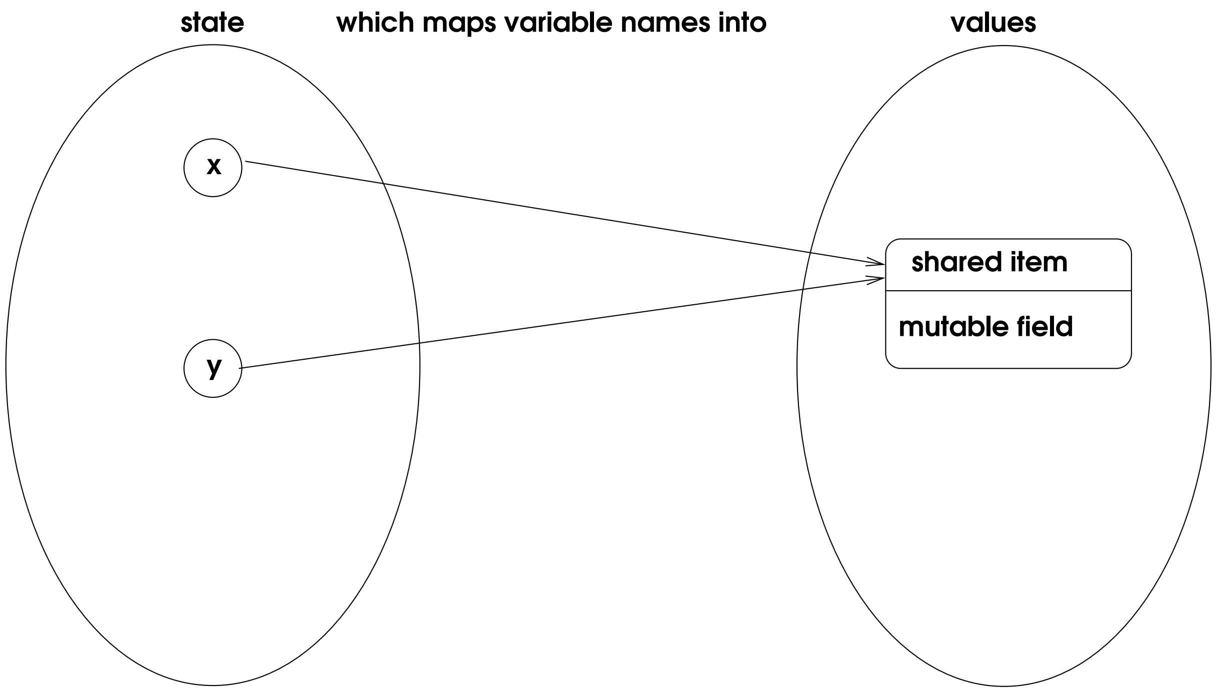

Figure 5: Aliasing.

- More generally, *side effects* not visible in the code itself make establishing correctness much harder. Aliasing is one such example.
- Furthermore, unintended side effects are a well-known source of hard-to-find bugs. . .
- It is possible to define a fully functioning programming language (and not just in theory like GCL) without any side effects. Input/Output operations become trickier, though. E.g. Haskell (Peyton Jones, 2003) is such a *pure* language.

The Well-Defined Domain

- Calculating

  $$\mathrm{wp}(x := 1/y, z = 3) = (z = 3) \qquad \text{by Eq. (9)}$$

  claims that "this assignment is guaranteed to terminate normally for all y". But it **abort**s (Section 2.4.2) instead if $y = 0$! Hence the weakest precondition we *want* to get is instead

  $$= (y \neq 0)\,\mathbf{cand}\,(z = 3)$$

  which precludes the states which would end up in this run-time error.

- Hence we turn Eq. (9) into

  $$\mathrm{wp}(x := \mathit{expression}, \phi) = \mathrm{domain}(\mathit{expression})\,\mathbf{cand}\,\phi[x \leftarrow \mathit{expression}] \quad (11)$$

  where the new part is a formula describing those states where *expression* is well-defined (= can be evaluated without **abort**ing).

- This domain(*expression*) is in fact a "macro" which prints the appropriate formula for the given input *expression*.
- This function can be defined by looking at the grammar (which we omit defining formally) for the GCL *expressions*.

  For example, the grammar rule

  $$\mathit{expression} \longrightarrow \mathit{expression} \quad \texttt{'/'} \quad \mathit{expression}$$

  for division gives us the appropriate formula building rule

  $$\mathrm{domain}(e_1 \ \texttt{'/'} \ e_2) = (\mathrm{domain}(e_1) \wedge \mathrm{domain}(e_2)) \,\mathbf{cand}\, (e_2 \neq 0)$$

  or "division e_1 `'/'` e_2 is well-defined if both its operand expressions e_1 and e_2 are and the value of e_2 is not 0".
- E.g. in our example

  $$\mathrm{domain}(1/y) = (\overbrace{\mathrm{domain}(1)}^{=\textsc{true}} \wedge \overbrace{\mathrm{domain}(y)}^{=\textsc{true}}) \,\mathbf{cand}\, (y \neq 0)$$

  which simplifies into

  $$= (y \neq 0).$$
- In practice, this means the following:
  1. Does the *expression* contain some still unhandled subexpression $e_1 \circ e_2$ such that the operation $p \circ q$ *is not* defined for all p and q?
  2. Handle any smallest such subexpression $e_1 \circ e_2$ by adding in front of the (initially TRUE) output formula the protection test

     $$(\neg\phi_\circ[p \leftarrow e_1, q \leftarrow e_2]) \,\mathbf{cand} \ldots$$

     where the formula $\phi_\circ$ specifies those p and q for which $p \circ q$ is not defined.

     E.g. here the operator $\circ$ was division, and the formula $\phi_\circ$ was $(q = 0)$. Hence we added the protection test

     $$(\underbrace{\neg(q = 0)[p \leftarrow 1, q \leftarrow y]}_{=(y \neq 0)}) \,\mathbf{cand} \ldots$$
  3. Repeat until every such subformula has been handled.
- The above extends readily into quantifier-free formulae ϕ:

  domain(ϕ) is obtained similarly by considering its connectives $\wedge, \vee, \neg, \ldots$ as operators $\circ$ which do not need such handling (since they are defined everywhere).
- If domain($\ldots$) is TRUE, then we can drop it for brevity.

Multiple Assignment

- The first line

  $$l\,,u \;:=\; 1\,,N$$

  of our GCL binary search example (Figure 4) shows how many variables can be assigned at once.

- This avoids having to introduce *temp*orary variables which are not interesting for the algorithm or its proof.

  E.g. swapping x and y can be written directly as

  $$x\,,y \;:=\; y\,,x$$

- The general form of such a multiple assignment is

  $$x_1\,,x_2\,,x_3\,,\ldots,x_k \;:=\; e_1\,,e_2\,,e_3\,,\ldots,e_k$$

  where the k variable names $x_1, x_2, x_3, \ldots, x_k$ are all distinct from one another.

- The intuition is that all these k assignments $x_1 := e_1, x_2 := e_2, x_3 := e_3, \ldots, x_k := e_k$ are performed at the same time — in one single step.

- Semantically we extend Eq. (8) into

  $$f[x_1 \leftarrow \text{eval}_{\mathcal{GCL},f}(e_1), x_2 \leftarrow \text{eval}_{\mathcal{GCL},f}(e_2), x_3 \leftarrow \text{eval}_{\mathcal{GCL},f}(e_3), \ldots, x_k \leftarrow \text{eval}_{\mathcal{GCL},f}(e_k)]$$

  so that the k expressions $e_1, e_2, e_3, \ldots, e_k$ are indeed all first evaluated in this current state f, and only then is f updated with their values $v_1, v_2, v_3, \ldots, v_k$ into the next state f'.

- The corresponding extension of Eq. (11) is

  $$\text{wp}((x_1, x_2, x_3, \ldots, x_k := e_1, e_2, e_3, \ldots, e_k), \phi) = \phi[x_1 \leftarrow e_1, x_2 \leftarrow e_2, x_3 \leftarrow e_3, \ldots, x_k \leftarrow e_k]$$

  where all the textual replacements are also carried out *at the same time.*

- That is, only those occurrences of x_i which were present already in the original formula ϕ get replaced by e_i.

  But not those which were produced by replacements!

- Hence the following are different, as they should be:

  $$\begin{aligned}&\text{wp}((x, y := y, x), (x = Y \wedge y = X))\\ &\qquad = (x = Y \wedge y = X)[x \leftarrow y, y \leftarrow x] \qquad \text{both at the same time}\\ &\qquad = (y = Y \wedge x = X)\end{aligned}$$

  vs.

  $$\begin{aligned}&\overbrace{\overbrace{(x = Y \wedge y = X)[x \leftarrow y]}^{\text{do first this replacement}}[y \leftarrow x]}^{\text{then do this to its result}}\\ &\qquad = (y = Y \wedge y = X)[y \leftarrow x]\\ &\qquad = (x = Y \wedge x = X)\\ &\qquad = \text{wp}((y := x; x := y), (x = Y \wedge y = X))).\end{aligned}$$

2.4.5 Branching with If

- The first command with guards is choosing one of the given guarded alternatives.
- The syntax is

```
if  $guard_1 \rightarrow command_1$
 ▯  $guard_2 \rightarrow command_2$
 ▯  $guard_3 \rightarrow command_3$
 ▯  $\vdots$
 ▯  $guard_m \rightarrow command_m$
fi
```

- The informal semantics is "Choose freely (that is, nondeterministically) among those alternatives whose *guard* is TRUE, execute its *command*, and continue after the **fi**. But if there is nothing to choose from, then **abort**".
- The formal semantics is

$$\begin{aligned}\mathrm{wp}(\textbf{if } \mathcal{B} \textbf{ fi}, \phi) = \mathrm{domain}(\mathit{guards}) \wedge \mathit{guards} \wedge{} & \\ & (\mathit{guard}_1 \implies \mathrm{wp}(\mathit{command}_1, \phi))\wedge \\ & (\mathit{guard}_2 \implies \mathrm{wp}(\mathit{command}_2, \phi))\wedge \\ & (\mathit{guard}_3 \implies \mathrm{wp}(\mathit{command}_3, \phi))\wedge \\ & \vdots \\ & \wedge (\mathit{guard}_m \implies \mathrm{wp}(\mathit{command}_m, \phi)) \end{aligned} \tag{12}$$

where $\mathcal{B}$ stands for the body and

$$\mathit{guards} = (\mathit{guard}_1 \vee \mathit{guard}_2 \vee \mathit{guard}_3 \vee \cdots \vee \mathit{guard}_m) \tag{13}$$

is the disjunction of all the individual *guards*.

- "We can evaluate all the *guards* without **abort**ing. At least one of them evaluates to TRUE. No matter which of these TRUE branches we choose, its *command* ensures ϕ."
- The practical reason for having an explicit **skip** (Section 2.4.2) is that

```
if $p$
   then $q$
   else  $r$
```

must be written as

```
if  $p \rightarrow q$
 ▯  $\neg p \rightarrow r$
fi
```

in GCL.

- This is because we want to show explicitly that the **else** branch r can (and probably must) assume $\neg p$ in its correctness proof.

```
{ $\phi$ which must imply both
      every domain($guard_i$)
      and at least one of the $guard_i$ }
if  $guard_1$ $\rightarrow$
    { $\phi \wedge guard_1$ }
    $command_1$
    { $\psi$ }
 $[\!]$ $\vdots$
fi
{ $\psi$ }
```

Figure 6: Propagating conditions into "if".

- Hence if we want to omit the **else** branch, we must instead use $\neg p \rightarrow$ **skip** in its place.
- Note also that it is much easier to add or delete branches in **if**...**fi** than in complicated **if**...**then**...**else if**...**else** structures!
- In writing and verifying GCL programs, we often know the pre- and postconditions surrounding the **if** command. Hence it is useful to restate its formal semantics in terms of view (4).
- Condition

  1 is "the precondition ϕ must ensure (at least one of) the *guards*"

  2 is "every branch must independently ensure the postcondition ψ".

Theorem 13 (View (4) for **if**). $\phi \implies \mathrm{wp}(\textbf{\textit{if}}\ \mathcal{B}\ \textbf{\textit{fi}}, \psi)$ *if and only if the following two conditions hold:*

1. $\phi \implies (\mathrm{domain}(guards) \wedge guards)$*, and*

2. for each branch i we have

$$(\phi \wedge guard_i) \implies \mathrm{wp}(command_i, \psi).$$

- These conditions 1 and 2 (Theorem 13) explain how the Hoare assertions { ... } propagate from the outside in (Figure 6).

- Let us start proving this theorem (Theorem 13) with the formula describing its two conditions:

$$\overbrace{(\phi \implies (\mathrm{domain}(guards) \wedge guards))}^{\text{condition 1}} \wedge \\ \bigwedge_{i \text{ in } \mathcal{B}} \underbrace{(\phi \wedge guard_i) \implies \mathrm{wp}(command_i, \psi)}_{\text{condition 2}}. \quad (14)$$

- Propositional logic has (among others) the tautology

$$((\alpha \land \beta) \implies \gamma) \quad \iff \quad (\alpha \implies (\beta \implies \gamma)) \tag{15}$$

(which you can verify by e.g. forming its truth table). We can apply it in the "$\implies$" direction to these conditions 2 to rewrite formula (14) into the equivalent form

$$(\phi \implies (\mathrm{domain}(\mathit{guards}) \land \mathit{guards})) \land \bigwedge_{i \text{ in } \mathcal{B}} \phi \implies (\mathit{guard}_i \implies \mathrm{wp}(\mathit{command}_i, \psi)). \tag{16}$$

- Next we can apply another propositional tautology

$$((\alpha \implies \beta) \land (\alpha \implies \gamma)) \quad \iff \quad (\alpha \implies (\beta \land \gamma)) \tag{17}$$

repeatedly to rewrite formula (16) into yet another equivalent form

$$\phi \implies (\mathrm{domain}(\mathit{guards}) \land \mathit{guards} \land \bigwedge_{i \text{ in } \mathcal{B}} (\mathit{guard}_i \implies \mathrm{wp}(\mathit{command}_i, \psi))). \tag{18}$$

- But this form (18) is in fact

$$\phi \implies \mathrm{wp}(\mathbf{if}\ \mathcal{B}\ \mathbf{fi}, \psi) \tag{19}$$

by Definition (12).

- Hence formulae (14) and (19) are indeed equivalent, and this is what this theorem (Theorem 13) claims. Thus we have now proved it. □