2.4.6 Equational or Calculational Proofs

- The proof of Theorem 13 proceeded by massaging a formula (14) into another formula (19). This kind of proof is called

  equational since it uses logical equivalences as algebraic equations. That is, it

  considered '$\Longleftrightarrow$' as the '$=$' for the *Boolean or truth value* type $\mathbb{B}$.

  proceeded by replacing equals with equals, just like in algebra.

  calculational since it considered formulae as expressions having this Boolean type — similarly to how we calculate with expressions having a numeric type such as $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{R}$, $\mathbb{Q}$,...

- Such proofs are very handy especially in programming: we often just massage our specifications to see

  – that they do indeed fit together as pre- and postconditions

  – what kind of code we might write to meet them

  rather than proving their truth logically.

  The book by Backhouse (2003) is devoted to this style of logical reasoning. It also features prominently in e.g. the book by Gries (1981), who is another proponent of this style.

- We can even write this proof equationally as

$$\begin{aligned} \text{Formula (14)} &\Leftrightarrow \text{Formula (16)} && \text{(by Equivalence (15))} \\ &\Leftrightarrow \text{Formula (18)} && \text{(by Equivalence (17))} \\ &\Leftrightarrow \text{Formula (19)} && \text{(by Definition (12))} \end{aligned}$$

  where we have adopted the notational convention that

  long forms like $\Longleftrightarrow, \Longrightarrow, \Longleftarrow$ appear in the formulae being massaged, while their corresponding

  short forms like $\Leftrightarrow, \Rightarrow, \Leftarrow$ denote the proof steps

  to reduce the risk of confusion.

- That is, we intend that the following two notations are alike:

$$\begin{aligned} \text{logical formula} &\Leftrightarrow \text{another logical formula} \\ \text{numerical expression} &= \text{another numerical expression.} \end{aligned}$$

- Extending this analogy further, we intend also that

$$\begin{aligned} \text{logical formula} &\Rightarrow \text{another logical formula} \\ \text{numerical expression} &\leq \text{another numerical expression.} \\ \text{by thinking that FALSE} &< \text{TRUE.} \end{aligned}$$

- This would permit the following kind of equational proof step:

$$\text{(a formula containing } \phi) \Rightarrow \text{(same formula with } \phi \text{ replaced by } \psi) \quad \text{(by } \textit{implication } \phi \Longrightarrow \psi). \quad (20)$$

- However, recall that logical negation "flips the direction around" just like unary minus:

	positive	negative
numerical	$a \leq b$	$(-a) \geq (-b)$
logical	$\alpha \Rightarrow \beta$	$(\neg\alpha) \Leftarrow (\neg\beta)$

  Hence step (20) must be restriced further to "obey sign".

- This sign restriction for step (20) is that the replacements are allowed only in those places which are within the scope of an *even* number of negations '$\neg$'.
  - These negation pairs cancel each other out, so the "sign" of the replacement place is positive, and so the direction does not flip.
  - We must first expose and count also all the negations which are hiding within other connectives by expanding their definitions, such as:

hidden	exposed
$\alpha \Longrightarrow \beta$	$(\neg\alpha) \vee \beta$
$\alpha \Longleftrightarrow \beta$	$(\alpha \Longrightarrow \beta) \wedge (\beta \Longrightarrow \alpha)$

  - If the replacement place is negative (= within an *odd* number of negations) instead, then the step direction flips to the opposite of the implication direction.

2.4.7 Looping with Do

- The GCL looping command is intuitively an **if** (Section 2.4.5) command which keeps repeating itself over and over again until none of its *guard*s is TRUE any longer.

- The syntax is also the same, but instead of the **if**...**fi** we write **do**...**od**:

$$\begin{array}{ll} \textbf{do} & guard_1 \rightarrow command_1 \\ [\!] & guard_2 \rightarrow command_2 \\ [\!] & guard_3 \rightarrow command_3 \\ [\!] & \vdots \\ [\!] & guard_m \rightarrow command_m \\ \textbf{od} & \end{array}$$

- The informal semantics is "Choose freely among those alternatives whose *guard* is TRUE, execute its *command*, and continue at the **do**. But if there is nothing to choose from, then continue after the **od**".

- The formal semantics wp(**do** $\mathcal{B}$ **od**, ϕ) can be formed as follows (where $\mathcal{B}$ marks the source code of the loop body).
  - Take a fresh (= one we have never used before) predicate name $H_\phi^{\mathcal{B}}$. Note that these super- and subscripts are parts of the name itself. That is, we invent a new name for this $\mathcal{B}/\phi$ combination.
  - We shall axiomatize $H_\phi^{\mathcal{B}}(k)$ to say "if this loop is started in the current state, then it exits within $\leq k$ rounds, and ϕ holds when its exits".
  - This axiomatization can be expressed using *induction over* $k \in \mathbb{N}$.
    - This is why we included it in $\mathcal{GCL}$ (Definition 2).

do $\text{even}(n) \to$
$\quad n \;:=\; n/2$
$[\!]\;\; \neg\,\text{even} \wedge n > 1 \to$
$\quad n \;:=\; 3 \cdot n + 1$
od

Figure 7: The Collatz loop.

  - $*$ Without induction, we would have infinitely many axioms $H^{\mathcal{B}}_{\phi}(0)$, $H^{\mathcal{B}}_{\phi}(1)$, $H^{\mathcal{B}}_{\phi}(2), \ldots$ for every $H^{\mathcal{B}}_{\phi}$.
  - $*$ If a programming language/notation/logic… has any kind of looping construct, then its semantics will have somewhere some kind of inductive construct explaining its meaning.

– The base case is

$$H^{\mathcal{B}}_{\phi}(0) \iff \text{domain}(\mathit{guards}) \wedge (\neg\, \mathit{guards}) \wedge \phi \tag{21}$$

meaning "this loop exits immediately with ϕ true".

– The inductive case is

$$\forall k \in \mathbb{N} : H^{\mathcal{B}}_{\phi}(k+1) \iff H^{\mathcal{B}}_{\phi}(0) \vee \text{wp}(\textbf{if } \mathcal{B} \textbf{ fi}, H^{\mathcal{B}}_{\phi}(k)) \tag{22}$$

meaning "… or executes one round to reach a state where there are strictly fewer rounds left".

– The semantics is then

$$\text{wp}(\textbf{do } \mathcal{B} \textbf{ od}, \phi) = (\exists k \in \mathbb{N} : H^{\mathcal{B}}_{\phi}(k)). \tag{23}$$

In this way this weakest precondition consists of exactly those states f for which there exists some k (which depends of f) such that starting the loop in f is guaranteed to stop in at most k iterations, and ϕ will be true then.

This meaning will of course need also the corresponding cases (21) and (22) to fix the intended meaning for its $H^{\mathcal{B}}_{\phi}$.

Complex Loops Have Complex Logic

- A loop **do** $\mathcal{B}$ **od** can be very complicated, so its wp definition (23) (with its associated Axioms (21) and (22)) can also be a very complicated formula.
- This formula can become *logically difficult* even for some very simple loop bodies $\mathcal{B}$.

  E.g. the so-called *Collatz conjecture* (which has other names too) claims that a certain simple loop (Figure 7) halts (with $n = 1$) for all inputs $n \in \mathbb{N}$ except $n = 0$.
- But the Collatz conjecture is still open! That is, nobody has been able to prove

$$n > 0 \Longrightarrow \text{wp}(\text{the Collatz loop (Figure 7)}, \textsc{true})$$

  (or find a counterexample) even though this formula itself is not too complicated.
- In fact, we could (but will not) even argue that there can be no proof method for verifying **do** $\mathcal{B}$ **od** loops such that it would be at the same time

correct or error-free (which we really must insist upon!)

complete or capable of verifying all loops, and

effective or programmable into a computer.

- The argument would be the same as for the celebrated *Gödel's first incompleteness theorem* (1931) (Boolos and Jeffrey, 1989, Chapter 15) which states the same for arithmetic over $\mathbb{N}$ — which we used to get our Eq. (23)...
- You may have already seen Gödel's proof: it inspired Turing's (1936) proof that the halting problem is undecidable (Hopcroft et al., 2001, Theorem 9.6).
- A way to start would be to note that every computer program $\mathcal{P}(x\colon \mathbb{N})\colon \mathbb{B}$ can be written as a single loop such that

$$\mathrm{wp}((y, s := 0, 0; \textbf{do } \mathcal{C} \textbf{ od}), s = \begin{matrix}1\\2\end{matrix}) \tag{24}$$

  means "$\mathcal{P}(x)$ returns $\begin{matrix}\text{TRUE}\\\text{FALSE}\end{matrix}$". Here the loop body $\mathcal{C}$

  - depends on the program $\mathcal{P}$, and
  - is very long in general, but
  - consists only of simple branches of the form

$$(x \underset{=}{>} 0) \wedge (y \underset{=}{>} 0) \wedge (s = \text{some constant}) \rightarrow$$
$$x, y, s := x + \begin{matrix}1\\0\\-1\end{matrix}, y + \begin{matrix}1\\0\\-1\end{matrix}, \text{some other constant}.$$

  1. First write $\mathcal{P}$ as a so-called two-counter machine (with x and y as the two counters) (Hopcroft et al., 2001, Chapters 8.5.3–8.5.4).
  2. Then structure the finite control of this Turing-style machine into this loop by introducing an explicit third variable s for its current state.

- The exact definition (23) for **do** $\mathcal{C}$ **od** is thus too complicated, both in theory and in practice.
- In its place, we will use a loop verification method which

  cannot verify all possible loops — hence we forsake completeness

  can verify most of the kinds of loops we will be verifying and designing in practice.

- The intuitive problem with the Collatz loop (Figure 7) is that it has *no simple notion of "progress":*
  - If n is even, then it gets closer to termination (which occurrs when $n = 1$), but
  - ... but if n is odd, then the gets *further away* from it!
- In contrast, our method concentrates on only those loops for which the programmer can state this "progress towards termination" easily using logic.

{ ϕ which must imply every domain($guard_i$) }
do $guard_1 \to$
{ ϕ and $guard_1$ }
$Bound := bound$;
$command_1$
{ ϕ and $(bound < Bound)$ }
$[\!]$ ⋮
od
{ ϕ and none of the $guard_i$ }

Figure 8: Propagating conditions into "do".

Bounded Loops

- This progress can be measured by a *bound*ing expression, which gives an *explicit* upper bound k for Eq. (23).

Theorem 14 (View (4) for **do**)**.** *Suppose that the formula ϕ and the integer-valued expression $bound \in \mathbb{Z}$ satisfy the following three conditions:*

1. $\phi \implies \text{domain}(guards)$.
2. $(\phi \wedge guards) \implies (bound > 0)$.
3. *For each branch i we have*

$$(\phi \wedge guard_i) \implies \text{wp}((Bound := bound; command_i), \phi \wedge (bound < Bound))$$

where $Bound$ is a fresh ghost variable (Section 2.1) for remembering the old value of $bound$ before this branch.

Then $\phi \implies \text{wp}(\textbf{\textit{do}}\ \mathcal{B}\ \textbf{\textit{od}}, \phi \wedge \neg\, guards)$.

Outline of the proof. The central idea is to prove for every $k \geq 0$ that

$$\phi \wedge (bound \leq k) \Rightarrow \underbrace{H^{\mathcal{B}}_{\phi \wedge \neg\, guards}(k)}_{\text{essence of } \text{wp}(\textbf{do}\ \mathcal{B}\ \textbf{od}, \phi \wedge \neg\, guards)}.$$

This could be proved by induction on k.

Base case $k = 0$ can be calculated from conditions 1 and 2.

Inductive case $k + 1$ can also be calculated, after we have shown for every b that

$$\phi \wedge guards \wedge (bound \leq b + 1) \Rightarrow \text{wp}(\textbf{if}\ \mathcal{B}\ \textbf{fi}, \phi \wedge (bound \leq b))$$

which in turn takes a lengthy calculation from conditions 1 and 3. □

- Similar to our earlier View (4) for **if** (Theorem 13), this too explains how these two conditions propagate from the outside in (Figure 8).

- By condition 3, the invariant ϕ tells us what stays the same while the *bound* decreases, and we propagate both.
- This (Theorem 14) is our formalization for the "loop invariant-and-bound" idea we have used informally before in our binary search example (Figure 1).
- The invariant lets us state what we want to be true

  after the loop. But it must be stated indirectly as what we want to be true when the variables modified during the loop have reached their *final* values. Since we do not know these final values yet, we must state it so that it holds already

  before the loop starts when these variables have just been set to their *initial* values, and

  during each round of the loop for their current *intermediate* values, since these rounds will occur one after the other.

  Much of this course will be about inventing, stating and proving an appropriate invariant for the current programming situation.
- Logically, the bound provides an explicit upper limit for Eq. (23):

$$\exists k \leq (\textit{bound} \text{ with the } \textit{initial} \text{ values}).H_\phi^{\mathcal{B}}(k).$$

- The conditions restricting the bound (Theorem 14) state the following:

  Condition 3 states that the integer value of the *bound*ing expression will decrease strictly on each round.

  Condition 2 states that the loop will exit *at least* when this *bound*ing value becomes ≤ 0.

  Note that the loop is permitted to exit even sooner, when this *bound*ing value is still > 0.

  The idea is that

$$\underbrace{\textit{bound}_{\text{1st round}} > \textit{bound}_{\text{2nd}} > \textit{bound}_{\text{3rd}} > \cdots > \textit{bound}_{\text{last}}}_{\text{Only } \textit{finitely} \text{ many values!}}.$$

- As noted before, this view (4) for **do** (Theorem 14) cannot deal with every kind of terminating loop — unlike view (4) for **if** (Theorem 13), it is "if but not only if" the exact (but unusable) termination condition (23).
- E.g. the logician *Ackermann* designed his well-known function

$$\begin{aligned} A(0,n) &= n+1 \\ A(m+1,0) &= A(m,1) \\ A(m+1,n+1) &= A(m,A(m+1,n)) \end{aligned}$$

  to have the unnerving property (and more...) that although computing $A(x,x)$ does terminate, its run time (like its value) grows faster than any essentially simpler expression of x.

  Hence if we coded it as one **do** $\mathcal{C}$ **od** loop like in Eq. (24), we would get an example of a loop which cannot be handled with this method (Theorem 14) since it has no reasonable *bound* we could use.

{ invariant: the formula ϕ of the loop, and
bound: its *bound*ing expression }
do $guard_1 \to command_1$
[] ⋮
od
{ ψ }

Figure 9: Commenting a Loop.

The Checklist for a Loop

- Once we have developed our *bound*ed loop, we should also comment it — for ourselves and others.
- A good comment answers the code reader's questions:

  Why is this code here?

  Because we need to achieve a certain (sub)goal.

  This is the most important question to answer! Moreover, the code does not answer it, its programmer did.

  How does this code achieve its (sub)goal?

  Code itself is the final answer to that question. Usually the reader can see "okay, that code is indeed one way to achieve it" without much extra help.

  The reader rarely needs an explanation on *what* the code does, because it just replicates what the code already says. Such comments are helpful only for non-obvious coding tricks.

- The logical answer to the question

  why is the *postcondition* ψ after the loop

  how is its *invariant* ϕ and *bound*.

  The subproofs that each *branch* satisfies this ϕ and decreases this *bound* are rarely that interesting, as they are the logical answers to "what the code does".

  Hence we propose to add these as the loop comments (Figure 9).

- With these comments, the reader can use the following 5-point *checklist* to verify that he has really understood the loop:

  1. "Is the loop invariant ϕ really true when the loop starts?"
     - A loop is usually preceded by *initializations* for its variables.
     - This means checking that these initializations set up ϕ properly.
  2. "Does the execution of the loop body really maintain the invariant ϕ true?"

     $$\{\ \phi \wedge guard_i\ \}\ command_i\{\ \phi\ \}$$

     for every branch i.

     By nondeterminism, we do not know which branches of the body gets executed, so we must check this for all of them.

3. "Do we really get the promised outcome ψ after exiting the loop?"

$$(\phi \wedge \neg\, guards) \Longrightarrow \psi$$

This is in fact the logical *design criterion* for the loop: We want to get from the precondition of the loop into this postcondition ψ, so what ϕ and *guards* should we choose?

4. "Does $bound \leq 0$ really exit the loop?"

$$(\phi \wedge guards) \Longrightarrow (bound > 0)$$

5. "Does executing the body of the loop really cause new $bound <$ old $bound$?"

$$\{\ \phi \wedge guard_i\ \}\ Bound := bound; command_i \{\ bound < Bound\ \}$$

for every branch i.

- Strictly speaking, there is a 6th point to check too: that

$$\phi \Longrightarrow \text{domain}(guards)$$

by condition 1 (Theorem 14). But this is usually easy, since domain($guards$) is usually simple (even TRUE).

Verifying the Checklist

- Let us now see an example on how to verify a given loop with this 5-point checklist.
- Our example code (Figure 10) is a small but nontrivial algorithm.
  - Is the code alone enough to tell you what the loop is for?
    - What does it compute?
    - What roles do its three variables x, p and a play?
  - Adding the pre- and postconditions tell you that.
    - Here X and P are the ghost variables for the initial values of the input variables x and p.
    - Hence the code claims to compute into a the value of x raised into the power p (destroying x and p in the process)...
    - ... and in $\mathcal{O}(\log p)$ rounds too!
    - But are these pre- and postconditions enough to convince you that the code really works as claimed?
  - Adding also the invariant and bound for the loop gives hints on how you can convince yourself that it really does work as claimed.

- Note that from now on we shorten our formulae by not repeating the parts which do not change.

  E.g. the invariant is really $x \in \mathbb{R} \wedge p \in \mathbb{N} \wedge a \cdot x^p = X^P$ but this permanent part just "trickles down" without having to be explicitly hauled along.

$\{\ x \in \mathbb{R} \wedge p \in \mathbb{N} \wedge X = x \wedge P = p\ \}$
$a\ :=\ 1\,;$
$\{$ invariant: $a \cdot x^p = X^P$
bound: $p\ \}$
do $\neg\,\mathrm{even}(p) \rightarrow$
$p\,,\ a\ :=\ p-1\,,\ a \cdot x$
$[\!]$ $\mathrm{even}(p) \wedge (p > 0) \rightarrow$
$p\,,\ x\ :=\ p\,\mathrm{div}\,2\,,\ x \cdot x$
od
$\{\ a = X^P\ \}$

Figure 10: A Not-so-obvious Loop.

Checkpoint 1 amounts to verifying the Hoare triple

$$\{\ X = x \wedge P = p\ \}a := 1\{\ a \cdot x^p = X^P\ \}$$

for the initialization of a.

- It seems obvious, but it can also be verified by calculation:

$$\begin{aligned}\mathrm{wp}(a := 1, a \cdot x^p = X^P) &= (1 \cdot x^p = X^P)\\ &\Leftrightarrow (x^p = X^P)\end{aligned}$$

  which is indeed implied by $X = x \wedge P = p$ as in View (4).
- But note that had we not known a suitable initializer, then we could have determined it: calculate for an *unknown* b

$$\mathrm{wp}(a := b, a \cdot x^p = X^P) = (b \cdot x^p = X^P)$$

  and then solve it from the three equations

$$X = x \qquad P = p \qquad b \cdot x^p = X^P.$$

Checkpoint 2 amounts to checking a Hoare triple for each branch.

- The triple for the first branch is

$$\begin{array}{l}\{\ a \cdot x^p = X^P \wedge \neg\,\mathrm{even}(p)\ \}\\ p\,,\ a\ :=\ p-1\,,\ a \cdot x\\ \{\ a \cdot x^p = X^P\ \}\end{array}$$

  and we can calculate

$$\begin{aligned}&\mathrm{wp}((p, a := p-1, a \cdot x), a \cdot x^p = X^P)\\ =&((a \cdot x) \cdot x^{(p-1)} = X^P)\\ \Leftrightarrow&(a \cdot x^{1+(p-1)} = X^P)\\ \Leftrightarrow&(a \cdot x^p = X^P)\end{aligned}$$

  and the result is indeed implied by the precondition, as required by View (4).
- The triple for the second branch is

$$
\begin{array}{l}
\{\ a \cdot x^p = X^P \wedge \mathrm{even}(p) \wedge (p > 0)\ \} \\
p\,,\ \ x\ :=\ p \operatorname{div} 2\,,\ \ x \cdot x \\
\{\ a \cdot x^p = X^P\ \}
\end{array}
$$

and again we can calculate

$$
\begin{aligned}
& \mathrm{wp}((p, x := p \operatorname{div} 2, x \cdot x), a \cdot x^p = X^P) \\
=& (a \cdot (x \cdot x)^{(p \operatorname{div} 2)} = X^P) \\
\Leftrightarrow& (a \cdot (x^2)^{(p \operatorname{div} 2)} = X^P) \\
\Leftrightarrow& (a \cdot x^{2 \cdot (p \operatorname{div} 2)} = X^P)
\end{aligned}
$$

and since the precondition assures $\mathrm{even}(p)$ we also have $2 \cdot (p \operatorname{div} 2) = p$ and hence

$$
\Leftrightarrow (a \cdot x^p = X^P).
$$

Checkpoint 3 amounts to checking that we have indeed reached the promised $a = X^P$ when neither of the *guard*s is true any more.

- The loop keeps executing for as long as

$$
\underbrace{\neg\,\mathrm{even}(p)}_{\text{the first}} \vee \underbrace{\mathrm{even}(p) \wedge (p > 0)}_{\text{and the second guard}} \quad \Leftrightarrow \quad (p > 0) \tag{25}
$$

  by propositional logic.
- The stopping condition $\neg(p > 0)$ is equivalent to $p = 0$ since $p \in \mathbb{N}$.
- Then we can plug this $p = 0$ into the invariant $a \cdot x^p = X^P$ to get the promised $a = X^P$.

Checkpoint 4 amounts to checking that

$$
(a \cdot x^p = X^P) \wedge \underbrace{(p > 0)}_{\text{by Eq. (25)}} \Longrightarrow (p > 0)
$$

but this is now trivial.

Checkpoint 5 amounts again to checking a Hoare triple for each branch.

- The triple for the first branch is

$$
\begin{array}{l}
\{\ a \cdot x^p = X^P \wedge \neg\,\mathrm{even}(p)\ \} \\
P_0\ :=\ p\,; \\
p\,,\ \ a\ :=\ p - 1\,,\ \ a \cdot x \\
\{\ p < P_0\ \}
\end{array}
$$

  where P_0 is another ghost variable for p (= its value before this branch), since P is already used (as the initial value of p before the whole loop).

  Calculation yields

$$
\begin{aligned}
& \mathrm{wp}((P_0 := p; p, a := p - 1, a \cdot x), (p < P_0)) \\
=& \,\mathrm{wp}((P_0 := p), ((p - 1) < P_0)) \\
=& ((p - 1) < p).
\end{aligned}
$$

- The triple for the second branch is

$$\begin{array}{l}\{\ a \cdot x^p = X^P \wedge \mathrm{even}(p) \wedge (p > 0)\ \}\\ P_0 \ := \ p\,;\\ p\,,\ \ x \ := \ p \operatorname{div} 2\,,\ \ x \cdot x\\ \{\ p < P_0\ \}\end{array}$$

  and calculation yields

$$\begin{aligned}&\mathrm{wp}((P_0 := p; p, x := p \operatorname{div} 2, x \cdot x), (p < P_0))\\ =&\,\mathrm{wp}((P_0 := p), ((p \operatorname{div} 2) < P_0))\\ =&(p \operatorname{div} 2) < p\\ \Leftrightarrow&(p > 0).\end{aligned}$$

Checkpoint 6 is trivial, since both tests $\mathrm{even}(p)$ and $(p > 0)$ in the two *guards* are defined for all $p \in \mathbb{N}$.

Hence we are done: the code (Figure 10) does indeed work as advertised.

2.4.8 Arrays

- Until now our GCL programs have only used *atomic* types such as $\mathbb{N}, \mathbb{Z}, \mathbb{B}, \ldots$ whose values are indivisible: they have no further subparts.
- GCL has also *arrays* as its basic non-atomic types.
- We can access the parts of an array a by indexing:
  - This a is another *simple variable* like the $x, y, z, \ldots$ we have seen before, but it always has an *array as its value*.
  - The expression $a[i]$ is permitted on both sides of the assignment ':='.
  - It means intuitively "the value at index i of array a" as expected.
  - On the *right* side it means reading the current value.
  - On he *left* side it means modifying the current value into the next one.
- We also assume that the standard library $\mathcal{GCL}$ contains two functions

  $\mathrm{lower}(a) \in \mathbb{Z}$ for getting the *smallest*, and
  $\mathrm{upper}(a) \in \mathbb{Z}$ for getting the *largest*

  valid index of the array a.
- The valid index range is then the contiguous

$$\mathrm{indices}(a) = \{\mathrm{lower}(a), \mathrm{lower}(a) + 1, \mathrm{lower}(a) + 2, \ldots, \mathrm{upper}(a)\}\,.$$

  Trying to access $a[k]$ for any invalid index $k \notin \mathrm{indices}(a)$ will **abort** (Section 2.4.2).
- Hence the first line of our GCL binary search (Figure 4) can be written as

$$l\,,u \ := \ \underbrace{\mathrm{lower}(A)}_{\text{instead of } 1}\,,\underbrace{\mathrm{upper}(A)}_{\text{instead of } N}\,;$$

  and it will work for all numeric arrays, regardless of their permitted indices.

- Then the correctness proof reads "for all initial values of A, which are arrays, we have…".
- We can also have lower(a) > upper(a): the array a can be *empty* — without any valid indices.
- *Multidimensional* arrays are represented as *arrays of arrays*. That is, arrays which contain other arrays as their elements.
- E.g. if m is a two-dimensional integer matrix, then
  - indices(m) gives its *row numbers*
  - $m[3]$ is its row number 3
  - indices($m[3]$) gives the *column numbers* for its row 3
  - $m[3][5]$ is the integer element at column number 5 of row number 3.
    It can be parenthesized as $(m[3])[5]$ to make clear the order in which the indices apply.
- We assume that an array variable does not change its number of dimensions nor element type during execution.

  E.g. this matrix variable m is assigned only arrays of arrays of integers, and not values of another type such as
  - arrays of integers — different (too small) dimensions
  - arrays of arrays of Booleans — different element type
  - arrays of arrays of arrays of integers — different (too large) dimensions,…

Arrays as Finite Functions

- Let a be a variable whose type is **array of** τ where τ is the element type.
- So for multidimensional arrays, τ would be another **array of** τ' and so on, until we reach the atomic types.
- Semantically, the value $f(a)$ of a in the current state f is some *function*

$$g\colon (\text{some } \textit{finite contiguous subrange} \text{ of } \mathbb{Z}) \mapsto \tau$$

  such that $g(i)$ is the element $a[i]$. Now

  lower(a) is the *smallest* integer in the subrange of this g, and

  upper(a) is the *largest*

  and $g(k)$ is undefined everywhere else.
- Logically, the standard library $\mathcal{GCL}$ already has (names for) standard functions such as '+', so these functions g are nothing new… almost.
- The difference is that program code can *modify* them.
- This GCL arrays-as-functions approach means that array modification will have *copying* semantics:

$s\,,\ \ i\ :=\ 0\,,\ \ \mathrm{lower}(a)\,;$
$\{\ \text{invariant: } s = \sum_{j=\mathrm{lower}(a)}^{i-1} a[j]$
$\quad \text{bound: } \mathrm{upper}(a) + 1 - i\ \}$
do $i \le \mathrm{upper}(a) \to$
$\quad s\,,\ \ i\ :=\ s + a[i]\,,\ \ i + 1$
od

$$
\begin{aligned}
& \mathrm{wp}((s, i := s + a[i], i + 1), \text{invariant}) \\
= & \left(s + a[i] = \sum_{j=\mathrm{lower}(a)}^{(i+1)-1} a[j] \right) \\
\Leftrightarrow & \left(s \mathbin{/\!\!/\!\!/} a[i] = \left(\sum_{j=\mathrm{lower}(a)}^{i-1} a[j] \right) \mathbin{/\!\!/\!\!/} a[i] \right) \\
\Leftrightarrow & \text{ invariant.}
\end{aligned}
$$

Figure 11: Summing a One-dimensional Array.

$$b\ :=\ a\,;$$

stores into the array variable b a separate copy of the contents of the array variable a.

This is like e.g. in the Pascal programming language.

- The other choice (taken in e.g. the C and Java programming languages) would have been to make b point to the same memory area as a — but this would reintroduce the dreaded aliasing!

- Looking up values from arrays is semantically a straightforward extension to Eqs. (8) and (11):

$$
\begin{aligned}
\mathrm{eval}_{\mathcal{GCL},f}(a[i]) &= \underbrace{(\mathrm{eval}_{\mathcal{GCL},f}(a))}_{\text{look up the } g}\underbrace{(\mathrm{eval}_{\mathcal{GCL},f}(i))}_{\text{evaluate } g \text{ at } i} \\
\mathrm{domain}(a[i]) &= \mathrm{domain}(a)\,\mathbf{cand}\ \mathrm{domain}(i)\,\mathbf{cand} \\
&\qquad (\mathrm{lower}(a) \le i \wedge i \le\ \mathrm{upper}(a))\,\mathbf{cand}\ldots
\end{aligned}
$$

where

$$\mathrm{domain}(\textit{variable}) = \textsc{true}.$$

- This works for multidimensional arrays as well: then a is another array expression such as $m[j]$, and we can apply the same rule again.

- This also means that code (Figure 11) where arrays appear only on the *right* side of the assignments ':=' can be handled with the tools we already know.

- Explaining array appearences on the *left* side turns out to be trickier.

Single Assignment into Array

- The single assignment

$$name[index] \;:=\; value\,;$$

is semantically evaluated similarly to Eq. (8):

1. First evaluate first both *index* and *value* in the current state f.
2. Then create a new array function g' from the array function g obtained for *name* by mapping the *index* into this *value* instead.
3. Finally create the next state by *re*assigning this g' into *name*.

$$f\,\underbrace{[name \leftarrow \overbrace{\mathrm{eval}_{\mathcal{GCL},f}(name)}^{\text{1. Get the function } g}\overbrace{[\mathrm{eval}_{\mathcal{GCL},f}(index) \leftarrow \mathrm{eval}_{\mathcal{GCL},f}(value)]}^{\text{2. then modify } g \text{ into another } g'}]}_{\text{3. then reassign this new } g' \text{ into } name \text{ to get the next state } f'.}\,.$$

- However, capturing this semantic idea syntactically is more difficult than before:

$$\begin{aligned}\mathrm{wp}(a[j] := 5, a[i] = 5) &= (a[i] = 5)[a[j] \leftarrow 5] \\ &= (a[i] = 5)\end{aligned}$$

by the straightforward textual substitution of Eq. (9) — but this is wrong! The result should have been

$$= ((a[i] = 5) \vee (j = i))$$

instead, because the assignment *can* set $a[i]$ to 5 — if the indices are the same!

- Hence the construction of $\mathrm{wp}(name[index] := value, \phi)$ must carry also information all the possible (in)equalities between the indices appearing in it.

- This information is carried as follows:
  - We *temporarily* add to our intermediate formulae a new construct of the form

$$\langle name; [index] \leftarrow value \rangle \tag{26}$$

which stands for "the array which is otherwise like *name* but assigns *value* into position $[index]$ instead".

These carry the information during the construction.

  - We substitute these constructs textually instead:

$$\phi[name \leftarrow \text{construct (26)}]. \tag{27}$$

The idea is again the same: ϕ *will be* true about (the elements of) the array *name* after the assignment if and only if formula (27) *was* true before it.

  - We then *simplify* this resulting formula (27) to get rid of these introduced constructs (26).
  - The (in)equalities between indices will appear during this simplification phase. When all these constructs have been simplified away, we have our final formula.

- The simplification rule for constructs (26) is:

– Let ϕ still contain some

$$\langle name; [index] \leftarrow value \rangle\, [lookup]. \tag{28}$$

It means "*look up* this entry from that modified array" since it arose from having $name[lookup]$ in the original formula ϕ.

– Expand this ϕ into the disjunction

$$\overbrace{((index = lookup) \wedge (\phi \text{ with (28) replaced by } value))}^{\text{looking up the modified value}} \vee \underbrace{((index \neq lookup) \wedge (\phi \text{ with (28) replaced by } name[lookup]))}_{\text{looking up some other old value}} \tag{29}$$

which separates the two cases.

– Although this new ϕ is longer than the old ϕ was, it has fewer constructs (26) left, so this this simplification does eventually end.

- Hence the correct construction for our example is instead

$$\begin{aligned} \text{wp}(a[j] := 5, a[i] = 5) &= (a[i] = 5)[a \leftarrow \langle a; [j] \leftarrow 5 \rangle] \\ &= (\langle a; [j] \leftarrow 5 \rangle\, [i] = 5) \\ &\Leftrightarrow ((j = i) \wedge (5 = 5)) \vee ((j \neq i) \wedge (a[i] = 5)) \\ &\Leftrightarrow (j = i) \vee (a[i] = 5) \end{aligned}$$

which is correct: "the weakest precondition for $a[j] := 5$ to guarantee $a[i] = 5$ is

either that $j = i$, in which case the assignment guarantees it,

or $j \neq i$, in which case the assignment does not alter $a[i]$, so it must already be 5".

- For a more complicated example, consider the question "Can the assignment $b[b[i]] := i$ change the truth value of test $b[i] = i$?"

Solving this via operational reasoning gets rather tricky...

Calculating it yields

$$\begin{aligned} &\text{wp}(b[b[i]] := i, b[i] = i) \\ =&(b[i] = i)[b \leftarrow \langle b; [b[i]] \leftarrow i \rangle] \\ =&(\langle b; [b[i]] \leftarrow i \rangle\, [i] = i) \\ \Leftrightarrow&((b[i] = i) \wedge \underbrace{(i = i)}_{\text{TRUE}}) \vee \underbrace{((b[i] \neq i) \wedge (b[i] = i))}_{\text{FALSE}} \\ \Leftrightarrow&(b[i] = i) \end{aligned}$$

and the answer is "No, it cannot".

Aliasing within an Array

- In fact, this construction of $\text{wp}(name[index] := value, \phi)$ takes into account all the possible *aliasings* in $name$:

all the possibilities that $name[i]$ and $name[j]$ and $name[k]$ and... might be the same element in ϕ.

- Here we could solve the aliasing problem, because we have *explicit* aliasing information — the indices $i, j, k, \ldots$ — which we can express in the weakest precondition formula being constructed.
- However, the formula becomes longer and longer, since it spells out explicitly all the possible (in)equality combinations between i vs. j, j vs. k, i vs. k,... as its disjuncts.

Multidimensional Assignment

- Let us next extend this construction to assignments with more than one index:

$$\mathrm{wp}(m[i_1][i_2][i_3]\ldots[i_k] := e, \phi).$$

- First we extend the constructs (26) into whole *sequences* of indices:

$$\langle m; [i_1][i_2][i_3]\ldots[i_k] \leftarrow e\rangle\,.$$

- Then we say that the expansion (29) is always applied to the *first index* i_1 in the sequence:

$$\langle m; [i_1][i_2][i_3]\ldots[i_k] \leftarrow e\rangle\,[j]$$

expands into

$$(i_1 = j) \wedge (\phi \text{ with } \langle m[j]; [i_2][i_3]\ldots[i_k] \leftarrow e\rangle \text{ instead}) \\ \vee \\ (i_1 \neq j) \wedge (\phi \text{ with } m[j] \text{ instead})$$

- Then $\langle m[j]; [i_2][i_3]\ldots[i_k] \leftarrow e\rangle$ gets further expanded in the same way. Each expansion shortens the sequence, so we do not keep expanding forever.
- Eventually we get the *empty* sequence ε. Then we can eliminate the construct: $\langle n; \varepsilon \leftarrow e\rangle$ "expands" to just e.
- In fact, we can even consider the familiar scalar (= non-array) variables x as *dimensionless* array variables:

$$\begin{aligned}\mathrm{wp}(x := e, \phi) &= \phi[x \leftarrow e] \\ &= \phi[x \leftarrow \langle x; \varepsilon \leftarrow e\rangle] \\ &= \mathrm{wp}(x\varepsilon := e, \phi).\end{aligned}$$

- E.g. we can calculate

$$\begin{aligned}&\mathrm{wp}(m[i][j] := 2 \cdot m[p][q], m[i][k] > 8) \\ =&(m[i][k] > 8)[m \leftarrow \langle m; [i][j] \leftarrow 2 \cdot m[p][q]\rangle] \\ =&(\langle m; [i][j] \leftarrow 2 \cdot m[p][q]\rangle\,[i][k] > 8) \\ \Leftrightarrow&\overbrace{(i = i)}^{\text{TRUE}} \wedge (\langle m[i]; [j] \leftarrow 2 \cdot m[p][q]\rangle\,[k] > 8) \\ &\quad \vee \underbrace{(i \neq i) \wedge (m[i][k] > 8)}_{\text{FALSE}} \\ \Leftrightarrow&(j = k) \wedge (\langle m[i][k]; \varepsilon \leftarrow 2 \cdot m[p][q]\rangle > 8) \\ &\quad \vee (j \neq k) \wedge (m[i][k] > 8) \\ \Leftrightarrow&(j = k) \wedge (2 \cdot m[p][q] > 8) \\ &\quad \vee (j \neq k) \wedge (m[i][k] > 8).\end{aligned}$$

Multiple Assignment

- We still need to extend this construction to the case where multiple elements of the same array are assigned at the same time:

$$\mathrm{wp}((a[i], a[j] := 1, 2), \phi)$$

  and so on.

- Like before, the operational intention is that everything else is precomputed in the current state before doing the actual assignments:

  the values for the indices (i,j) and the expressions (1,2).

- However, now we must also *fix the order* in which the actual assignments are done:
  - If $i = j$ then we must know whether $a[i]$ and $a[j]$ will get the value 1 or 2.
  - We choose *left-to-right* order, so they get the last value 2.

- We model this left-to-right assignment order by extending construct (26) to ';'-separated single assignments:

$$\begin{aligned}&\mathrm{wp}((a[i_1], a[i_2], a[i_3], \ldots, a[i_k] := e_1, e_2, e_3, \ldots, e_k), \phi) = \\ &\qquad\qquad \phi[a \leftarrow \langle a; [i_1] \leftarrow e_1; [i_2] \leftarrow e_2; [i_3] \leftarrow e_3; \ldots; [i_k] \leftarrow e_k\rangle]\end{aligned}$$

- Then we say that expansion (29) is always applied to the *last assignment:*

$$\begin{aligned}&\langle a; [i_1] \leftarrow e_1; [i_2] \leftarrow e_2; [i_3] \leftarrow e_3; \ldots; [i_k] \leftarrow e_k\rangle\,[j] \\ &\qquad\qquad\qquad\qquad \text{is treated as} \\ &\qquad \langle\langle a; [i_1] \leftarrow e_1; [i_2] \leftarrow e_2; [i_3] \leftarrow e_3; \ldots; [i_{k-1}] \leftarrow e_{k-1}\rangle\,; [i_k] \leftarrow e_k\rangle\,[j].\end{aligned}$$

- E.g. let us verify swapping two elements using two ghost variables X and Y.

$$\begin{aligned}&\mathrm{wp}((a[i], a[j] := a[j], a[i]), (a[i] = Y \wedge a[j] = X)) \\ =&(a[i] = Y \wedge a[j] = X)[a \leftarrow \langle a; [i] \leftarrow a[j]; [j] \leftarrow a[i]\rangle] \\ =&(\langle a; [i] \leftarrow a[j]; [j] \leftarrow a[i]\rangle\,[i] = Y \\ &\quad \wedge\ \langle a; [i] \leftarrow a[j]; [j] \leftarrow a[i]\rangle\,[j] = X) \\ =&(\langle\langle a; [i] \leftarrow a[j]\rangle\,; [j] \leftarrow a[i]\rangle\,[i] = Y \\ &\quad \wedge\ \langle\langle a; [i] \leftarrow a[j]\rangle\,; [j] \leftarrow a[i]\rangle\,[j] = X) \\ =&((j = i) \wedge (a[i] = Y) \vee (j \neq i) \wedge \langle a; [i] \leftarrow a[j]\rangle\,[i] = Y) \\ &\quad \wedge (\underbrace{(j = j)}_{\text{TRUE}} \wedge (a[i] = X) \vee \underbrace{(j \neq j) \wedge (\langle a; [i] \leftarrow a[j]\rangle\,[j] = X)}_{\text{FALSE}}) \\ \Leftrightarrow&((j = i) \wedge (a[i] = Y) \vee (j \neq i) \wedge (a[j] = Y)) \\ &\quad \wedge (a[i] = X) \\ \Leftrightarrow&(a[j] = Y) \wedge (a[i] = X).\end{aligned}$$

  Note that the case $i = j$ (which is often forgotten...) is automatically included and checked.