(Intro to Invariants – proving iterative factorial)

Recall

```
(define (fact-iter n count result)
  (cond ((= count n) result)
        (else (fact-iter n (+ count 1)(* (+ count 1) result))))
  ))

(define (fact n)
  (fact-iter n 0 1))
```

everywhere count occurs in the scope of the formal param count, it means the same thing

Pattern of calls – useful clue to the proof – laid out in a table with formal parameters as column headings

for fact-iter

n	count	result
6	0	1
6	1	1
6	2	2
6	3	6
6	4	24
6	5	120
6	6	720

showing values each time fact-iter is called

on this call, the stopping test passes. So 720 is returned.

The clue we seek is the pattern: at each call, result = count!

For the first call,

$1 = 0!$ For the second

$1 = 1!$ For the third

$2 = 2!$

$6 = 3!$ etc

So we can see that this equation is true each time fact-iter is called!

We say that the equation — a relation among (some of) the program's variables — is INVARIANT.

How does this help us prove that the program is correct?

→ if the invariant holds each time fact-iter is called, then it holds the last time fact-iter is called.

→ according to the code, count = n the last time fact-iter is called, and when this is true, the value of result is returned.

→ But if result = count!, and count = n, this means that result = n! : the correct value is returned.

Hey — aren't you getting ahead of yourself? I mean: we did this for n = 6, not for general n

Of course — the expansion for n = 6 provides only a clue as to the invariant. We need to do some more work to conclude that $\boxed{\text{result} = \text{count!}}$ really is invariant, no matter the input. (given that pre is satisfied!)

So let's consider the more general situation — still for fact-iter, but now for an arbitrary integer $n \geq 0$.

An <u>invariant</u> is a <u>relation</u> (among a program's variables).

Here are some 'invariant-like' creatures which are NOT what we need:

17

n

$n = 6$

None of these help to prove that the program worked. Specifically: we need

$$\boxed{\begin{array}{c} \text{INV} \wedge \text{STOPPING-COND} \\ \Longrightarrow \\ \text{post-condition} \end{array}}$$

Is it the case that

$17 \wedge (count = n)$

$\Longrightarrow$

$result = n!$

?? What about

$(n = 6) \wedge (count = n) \Longrightarrow (result = n!)$

standard logical implication with <u>no</u> reference to run-time history!

?? Neither of these is a valid implication.

We will say that none of these

Guess-Invariants (GI)

is STRONG ENOUGH.

In the same vein,

$$1 = 1$$

(while invariant in the dictionary sense) is not an invariant we can use:

$$(1=1) \wedge (\text{stopping-cond}) \not\Rightarrow \text{post-condition.}$$

Intuitively — the problem with all of these is that they do not mention (enough of) the program's variables.

First test for a gI: is it strong enough [heuristic guidance: it needs to talk about the program's variables]

to imply the post-condition when logically anded with the stopping condition?

Important property of invariants:

Invariants are RELATIONS — they are NOT action statements. They are static assertions — in standard logic — about variables' values. They DO NOT REFER TO TIME. (no reference to "old" or "new" variable values)

We've seen that a guess invariant can be too weak. Can it be too strong? For example, why not just take

$$result = n!$$

as the GI? Well, this relation among variables, is NOT achieved by our initialization

We have

```
(define (fact n)
  (fact-iter n 0 1))
```

and clearly it is not the case that $1 \neq n!$ in most cases

So: second test for a GI is that it must be weak enough to be achieved (made true) by the code the first time the looping function (here fact_iter) is called.

what we're working up to here is an induction proof that the invariant is true on each call of the looping function. We are setting up an induction on the number of calls to the looping function that the inv is true on each call.

NOTE that this is different from what we did for recursive factorial. In that case, the induction was on the data (n), not on the number of calls.

amounts to the basis step.

Our IH: assume the gI was true for the current call.

Our IS: show that the gI is true for the next call

gI becomes I after it passes our tests

we have

assume count! = result

```
(define (fact-iter n count result)
  (cond ((= count n) result)
        (else (fact-iter n
                         (+ count 1)
                         (* (+ count 1) result)
   ))))
```

We don't know the order of the parameter updates – and the gI might be broken mid-stream

eg, after this update but before the last update

The beauty is that we do NOT care about midstream; all that matters is that the invariant has been re-established by the next call to fact-iter.

Showing this is just a standard logic implication. Specifically we need

$$[\text{count}! = \text{result}] \implies [(\text{count}+1)! = (\text{count}+1) * \text{result}]$$

and this is true by basic algebra and the def of factorial.

This is the third test for a GI: is it preserved true?

(ie, that the parameter updates preserve the GI)

Assuming that all three tests

→ strong enough?
→ weak enough?
→ preserved?

are passed, the $\underset{\text{guess-inv}}{GI}$ is promoted to $\underset{\text{inv}}{I}$.

We still need to give a termination argument to complete the proof.

Much more to come! Our ultimate goal is to use the logic during the coding process, not merely to retrofit logic to existing code.