Programs as Relations

COMP SCI 2LC3

Ryszard Janicki

Department of Computing and Software, McMaster University, Hamilton, Ontario, Canada

Programs as Relations

Consider the well-known procedure factorial, written in a small subset of Maple:

```
factorial := proc(n::posint)
local i, fac
    i:=1;
    fac:=1;
    while i < n do
    begin
       i:=i+1
       fac:=fac*i;
    end;
end proc;
```

Since *n* does not change its value in the above program we may consider it as a constant, so we may assume the above program has two integer variables *i* and *fac*.

- Define $D = \mathbb{Z} \times \mathbb{Z}$, where $\mathbb{Z}$ is the set of integers, and denote the elements of D as (i, fac). Each assignment statement can be modeled by a function $F_i : D \to D$, $i = 1, 2, 4, 5$, in the following manner:

  `"i:=1"` corresponds to $F_1(i, fac) = (1, fac)$,
  `"fac:=1"` corresponds to $F_2(i, fac) = (i, 1)$,
  `"i:=i+1"` corresponds to $F_4(i, fac) = (i + 1, fac)$, and
  `"fac:=fac*i"` maps to $F_5(i, fac) = (i, fac \cdot i)$.

- The test `"i<n"` can be modeled by two partial identity functions, $I_3, \bar{I}_3 : D \rightsquigarrow D$, where I_3 models `"i<n"`, and $\bar{I}_3$ models its complement, i.e. "$\mathtt{i} \geq \mathtt{n}$". More precisely,

  `"i<n"` corresponds to $I_3(i, fac)$, and
  "$\mathtt{i} \geq \mathtt{n}$" corresponds to $\bar{I}_3(i, fac)$, where ($\perp$ denotes *undefined*)

$$I_3(i, fac) = \begin{cases} (i, fac) & \text{if } i < n \\ \perp & \text{otherwise} \end{cases} \qquad \bar{I}_3(i, fac) = \begin{cases} (i, fac) & \text{if } i \geq n \\ \perp & \text{otherwise} \end{cases}$$

Programs as Relations

- Let R, R_1, R_2 be relations (each function is a relation!) that model the program statements `S`, `S1`, `S2`, respectively.
- Let `T` be a test modeled by partial identities I_T and $\bar{I}_T$, and
- let the symbols "$\circ$" and "$*$" denote the composition of relations, and transitive and reflexive closure of relations (Kleene star), respectively.

Formally, if $R, R_1, R_2 \subseteq X \times X$, then

- $x(R_1 \circ R_2)y \iff \exists(z|z \in X : xR_1z \wedge zR_2y)$
- $R^* = \bigcup_{i=0}^{\infty} R^i = \bigcup(i|i = 0, \ldots, \infty : R^i)$, where $R^0 = Identity$
- Alternative definition of R^*:

$$xR^*y \iff \exists(i|0 \leq i < \infty : xR^iy)$$

We can now model the basic programming constructs as follows

- `"S1;S2"` is modeled by $R_1 \circ R_2$,
- `"if T then S1 else S2"` is modeled by $(I_T \circ R_1) \cup (\bar{I}_T \circ R_2)$, and
- `"while T do S"` is modeled by $(I_T \circ R)^* \circ \bar{I}_T$.

Using this scheme one can easily model the above program by writing the following (symbolic) relational expression:

$$F = F_1 \circ F_2 \circ (I_3 \circ F_4 \circ F_5)^* \circ \bar{I}_3,$$

or

$$F = \underbrace{F_1}_{i:=1} \circ \underbrace{F_2}_{fac:=1} \circ \underbrace{(\overbrace{I_3}^{i<n} \circ \overbrace{F_4}^{i:=i+1} \circ \overbrace{F_5}^{fac:=fac*i})^* \circ \overbrace{\bar{I}_3}^{i \geq n}}_{\text{while } i<n \text{ do } i:=i+1; fac:=fac*i \text{ od}}$$

Programs as Relations

- If R_1 and R_2 are (possibly partial) functions, calculating $R = R_1 \circ R_2$ is easy: $R(x_1, \ldots, x_n) = R_2(R_1(x_1, \ldots, x_n))$.
- If at least one of R_1, R_2 is not a function, in general, we have to use the rule:
  $(x_1, \ldots, x_n) R_1 \circ R_2 (z_1, \ldots, z_n) \iff$
  $\exists (y_1, \ldots, y_n| : (x_1, \ldots, x_n) R_1 (y_1, \ldots, y_n) \wedge (y_1, \ldots, y_n) R_2 (z_1, \ldots, z_n))$.
- Nevertheless, it might happen that $R_1 \circ R_2$ is a function even if both R_1 and R_2 are not.
- In general $R_1 \cup R_2$ is not a function, even if both R_1 and R_2 are functions.
- Similarly, $R^* = \bigcup_{i=0}^{\infty} R^i = \bigcup(i|0 \leq i < \infty : R^i)$ is almost never a function, even if R is a function, since if R is a function, then
  $(x_1, \ldots, x_n) R^* (y_1, \ldots, y_n) \iff$
  $\exists (i|i \geq 0 : (y_1, \ldots, y_n) = R^i(x_1, \ldots, x_n))$,
  and this may happen for many, even infinite number of i's.

Lemma (1)

1. *For any test T, if R_1 and R_2 are functions then* $(I_T \circ R_1) \cup (\bar{I}_T \circ R_2)$ *is always a* function.
2. *For any test T, if R is a function, then $(I_T \circ R)^* \circ \bar{I}_T$ is either a* function *or the empty relation.*
3. *For any test T, if R is a function and $(I_T \circ R)^* \circ \bar{I}_T \neq \emptyset$, then*

$$((I_T \circ R)^* \circ \bar{I}_T)(x) = R^{k(x)}(x)$$

where $k(x)$ is the smallest j such that $\bar{I}_T(R^j(x_1, ..., x_n))(x) \neq \perp$.

- Recall $F = \underbrace{F_1}_{i:=1} \circ \underbrace{F_2}_{fac:=1} \circ \underbrace{(\overbrace{I_3}^{i<n} \circ \overbrace{F_4}^{i:=i+1} \circ \overbrace{F_5}^{fac:=fac*i})^* \circ \overbrace{\bar{I}_3}^{i\geq n}}_{\text{while } i<n \text{ do } i:=i+1; fac:=fac*i \text{ od}}$

- Define $G = I_3 \circ F_4 \circ F_5$ and $H = G^* \circ \bar{I}_3$, so $F = F_1 \circ F_2 \circ H$. First note that $(F_1 \circ F_2)(i, fac) = F_2(F_1(i, fac)) = (1, 1)$, so

$$F(i, fac) = H(F_2(F_1(i, fac))) = H(1, 1).$$

- For the function G we have:

$$G(i, fac) = (I_3 \circ F_4 \circ F_5)(i, fac) = F_5(F_4(I_3(i, fac))) = \begin{cases} (i+1, fac \cdot (i+1)) & \text{if } i < n \\ \perp & \text{if } i \geq n \end{cases}$$

Similarly :

$$G^2(i, fac) = G(G(i, fac)) = \begin{cases} (i+2, fac \cdot (i+1) \cdot (i+2)) & \text{if } i+1 < n \\ \perp & \text{if } i+1 \geq n \end{cases}$$

Hence :

$$G^j(i, fac) = \begin{cases} (i+j, fac \cdot (i+1) \cdot (i+2) \cdot \ldots \cdot (i+j)) & \text{if } i+j-1 < n \\ \perp & \text{if } i+j-1 \geq n \end{cases}$$

- Recall $F = \underbrace{F_1}_{i:=1} \circ \underbrace{F_2}_{fac:=1} \circ \underbrace{(\overbrace{I_3}^{i<n} \circ \overbrace{F_4}^{i:=i+1} \circ \overbrace{F_5}^{fac:=fac*i})^* \circ \overbrace{\bar{I}_3}^{i \geq n}}_{\text{while } i<n \text{ do } i:=i+1; fac:=fac*i \text{ od}}$
- For the function G we have:

$$G(i, fac) = (I_3 \circ F_4 \circ F_5)(i, fac) = F_5(F_4(I_3(i, fac))) = \begin{cases} (i+1, fac \cdot (i+1)) & \text{if } i < n \\ \perp & \text{if } i \geq n \end{cases}$$

Similarly :

$$G^2(i, fac) = G(G(i, fac)) = \begin{cases} (i+2, fac \cdot (i+1) \cdot (i+2)) & \text{if } i+1 < n \\ \perp & \text{if } i+1 \geq n \end{cases}$$

Hence :

$$G^j(i, fac) = \begin{cases} (i+j, fac \cdot (i+1) \cdot (i+2) \cdot \ldots \cdot (i+j)) & \text{if } i+j-1 < n \\ \perp & \text{if } i+j-1 \geq n \end{cases}$$

Notice that this last step requires a small amount of human ingenuity to "see" the pattern (although it can be automated in *some* cases).

- Recall: $H = G^* \circ \bar{I}_3$, $F(i, fac) = H(1,1)$ and

$$G^j(i, fac) = \begin{cases} (i+j, fac \cdot (i+1) \cdot (i+2) \cdot \ldots \cdot (i+j)) & \text{if } i+j-1 < n \\ \perp & \text{if } i+j-1 \geq n \end{cases}$$

- From Lemma 1(3) it follows $H(i, fac) = G^k(i, fac)$ where $k = k(i, fac)$ is the smallest j such that $\bar{I}_3(G^j(i, fac)) \neq \perp$.
- In this case we can easily show that there is only one such k and that $k(i, fac) = n - i$.
- Denote $fac' = fac \cdot (i+1) \cdot (i+2) \cdot \ldots \cdot (i+j)$.
- First note that $\bar{I}_3(G^j(i, fac)) \neq \perp$ implies $G^j(i, fac) \neq \perp$, i.e. $G^j(i, fac) = (i+j, fac')$ and $i+j-1 < n$.
- Furthermore $\bar{I}_3(i+j, fac') \neq \perp$ implies $i+j \geq n$.
- From $i+j-1 < n$ and $i+j \geq n$ we immediately get $i+j = n$, or $j = n - i$.
- Hence $k(i, fac) = n - i$, i.e.

$$H(i, fac) = G^{n-i}(i, fac) = (n, fac \cdot (i+1) \cdot (i+2) \cdot \ldots \cdot n).$$

- We have proved that $k(i, fac) = n - i$, i.e.

$$H(i, fac) = G^{n-i}(i, fac) = (n, fac \cdot (i+1) \cdot (i+2) \cdot \ldots \cdot n).$$

- This means

$$F(i, fac) = H(1, 1) = (n, n!),$$

so

$$\forall (n | n \in \mathbb{N} : \texttt{factorial}(n) = n!).$$

- So we are done. In many cases, but not all, the entire calculation can be mechanized, which is a big advantage!

Hoare Logic Again

$\{P\}$

do $B \longrightarrow S$ od or **while** B **do** S **od**

$\{R\}$

Checklist for proving loop correct

(a) P is true before execution of the loop

(b) P is a loop invariant: $\{P \land B\}\ S\ \{P\}$

(c) Execution of the loop terminates

(d) R holds upon termination: $P \land \neg B \implies R$

$\{P\}$
do $B \longrightarrow S$ od or **while** B **do** S **od**
$\{R\}$

Checklist for proving loop correct

(a) P is true before execution of the loop
(b) P is a loop invariant: $\{P \ \wedge \ B\}\ S\ \{P\}$
(c) Execution of the loop terminates
(d) R holds upon termination: $P \ \wedge \ \neg B \implies R$

Example (Factorial)

Consider the following program

Pr: $\quad i := 1;\ factorial := 1;$
while $i < n$ do
begin $i := i + 1;\ factorial := factorial * i$ end
od.

- Consider the Hoare triple $\{P\}Pr\{R\}$.
- The obvious choice for P and R is $P = (n > 0)$ and $R = (factorial = n!)$.
- Consider $\{n > 0\}$ $i := 1;\ factorial := 1;$ $\{P'\}$.
- Clearly $P' = (i = 1 \wedge factorial = 1 \wedge n > 0)$ and P' is *true* before execution of the loop.
- Hence (a) of the checklist is satisfied.

- Now we have a case:

$$\{P' = (i = 1 \wedge factorial = 1 \wedge n > 0)\}$$
$$\textbf{while}\ \ i < n\ \ \textbf{do}$$
$$\textbf{begin}\ \ i := i + 1;\ factorial := factorial * i\ \ \textbf{end}$$
$$\textbf{od}$$
$$\{R = (factorial = n!)\}$$

- We will show that $Q = (factorial = i! \wedge i \leq n)$ is a loop invariant. Assume $B = (i < n)$.
- To show that Q is a loop invariant, we have to prove that

$$\{Q \wedge B\}$$
$$i := i + 1;\ factorial := factorial * i$$
$$\{Q\},$$

or, in detail,

$$\{(factorial = i! \wedge i \leq n) \wedge i < n\}$$
$$i := i + 1;\ factorial := factorial * i$$
$$\{factorial = i! \wedge i \leq n\}$$

- Let solve:
  $\{Q'\}$
  $i := i+1;\ factorial := factorial * i$
  $\{Q = (factorial = i! \wedge i \leq n)\}$.
- From the definition of sequential composition of two assignment statements we have:
  $\{(factorial = i! \wedge i \leq n)[factorial := factorial * i][i := i+1]\}$
  $i := i+1;\ factorial := factorial * i$
  $\{factorial = i! \wedge i \leq n\}$.
- Hence:
  $(factorial = i! \wedge i \leq n)[factorial := factorial * i][i := i+1]$
  $\iff (factorial * i = i! \wedge i \leq n)[i := i+1] \iff$
  $factorial * (i+1) = (i+1)! \wedge i+1 \leq n \iff$
  $factorial * (i+1) = i! * (i+1) \wedge i < n \iff$
  $factorial = i! \wedge i < n \iff (factorial = i! \wedge i \leq n) \wedge i < n$.
- Which means $Q' = (factorial = i! \wedge i \leq n) \wedge i < n) = Q \wedge B$.
- Hence (b) holds, so Q is a loop invariant.

- What about termination of

  while $i < n$ do
  begin $i := i + 1;\ factorial := factorial * i$ end
  od?

- Initially $i = 1$ and $n > 0$. The loop contains '$i := i + 1$', so after n steps we get $i = n$, which implies $\neg B$, so the loop terminates.
- Hence (c) is also satisfied.
- Upon termination we have $(Q \wedge \neg B) = (factorial = i! \wedge i \leq n) \wedge i \geq n) \Rightarrow (factorial = i!) = R$.
- This means (d) is also satisfied.
- Hence we have proved: $\{n > 0\}Pr\{factorial = n!\}$

Another Example

- Consider the following program that already has been analysed using Hoare Logic:

$$q, r \ := \ 0, b;$$
$$\text{do } r \geq c \longrightarrow q, r \ := \ q + 1, r - c \text{ od}$$

- Define $D = \mathbb{Z} \times \mathbb{Z}$ and denote the elements of D as (q, r). Each parallel assignment statement can be modelled by a function $F_i : D \to D$, in the following manner:

  "$q, r \ := \ 0, b$" corresponds to $F_1(q, r) = (0, b)$, and
  "$q, r \ := \ q + 1, r - c$" corresponds to $F_2(q, r) = (q + 1, r - c)$.

- The test $r \geq c$ can be modelled by two partial identity functions, $I_3, \overline{I}_3 : D \rightsquigarrow D$, where I_3 models $r \geq c$, and $\overline{I}_3$ models its complement, i.e. $r < c$. More precisely,

  "$r \geq c$" corresponds to $I_3(q, r)$, and
  "$r < c$" corresponds to $\overline{I}_3(q, r)$, where ($\perp$ denotes *undefined*)

$$I_3(q, r) = \begin{cases} (q, r) & \text{if } r \geq c \\ \perp & \text{otherwise} \end{cases} \qquad \overline{I}_3(q, r) = \begin{cases} (q, r) & \text{if } r < c \\ \perp & \text{otherwise} \end{cases}$$

We can now model the basic programming constructs as follows

- `"S1;S2"` is modeled by $R_1 \circ R_2$,
- `"if T then S1 else S2"` is modeled by $(I_T \circ R_1) \cup (\bar{I}_T \circ R_2)$, and
- `"while T do S"` is modeled by $(I_T \circ R)^* \circ \bar{I}_T$.

Using this scheme one can easily model the above program by writing the following (symbolic) relational expression:

$$F = F_1 \circ (I_3 \circ F_2)^* \circ \bar{I}_3,$$

or

$$F = \underbrace{F_1}_{q,r:=0,b} \circ \underbrace{(\overbrace{I_3}^{r \geq c} \circ \overbrace{F_2}^{q,r:=q+1,r-c})^* \circ \overbrace{\bar{I}_3}^{r<c}}_{\text{do } r \geq c \rightarrow q,r:=q+1,r-c \text{ od}}$$

- Recall $F = \underbrace{F_1}_{q,r:=0,b} \circ \underbrace{(\overbrace{I_3}^{r \geq c} \circ \overbrace{F_2}^{q,r:=q+1,r-c})^* \circ \overbrace{\bar{I}_3}^{r<c}}_{\text{do } r \geq c \to q,r:=q+1,r-c \text{ od}}$

- Define $G = I_3 \circ F_2$ and $H = G^* \circ \bar{I}_3$, so $F = F_1 \circ H$. First note that $F_1(q,r) = (0,b)$, so $F(q,r) = H((F_1(q,r)) = H(0,b)$.

- For the function G we have:

$$G(q,r) = (I_3 \circ F_2)(q,r) = F_2(I_3(q,r)) = \begin{cases} (q+1, r-c) & \text{if } r \geq c \\ \perp & \text{if } r < c \end{cases}$$

Similarly :

$$G^2(q,r) = G(G(q,r)) = \begin{cases} (q+2, r-2\cdot c) & \text{if } r-c \geq c \equiv r \geq 2\cdot c \\ \perp & \text{if } r-c < c \equiv r < 2\cdot c \end{cases}$$

Hence :

$$G^i(q,r) = \begin{cases} (q+i, r-i\cdot c) & \text{if } r \geq i\cdot c \\ \perp & \text{if } r < i\cdot c \end{cases}$$

Notice that this last step requires a small amount of human ingenuity to "see" the pattern (although it can be automated in *some* cases).

- Recall: $H = G^* \circ \bar{I}_3$, $F(q,r) = H(0,b)$ and

$$G^i(q,r) = \begin{cases} (q+i, r-i\cdot c) & \text{if } r \geq i\cdot c \\ \perp & \text{if } r < i\cdot c \end{cases}$$

- From Lemma 1(3) it follows $H(q,r) = G^k(q,r)$ where $k = k(q,r)$ is the smallest j such that $\bar{I}_3(G^j(q,r)) \neq \perp$.
- In this case we can easily show that there is only one such k and $k = k(q,r)$ is the biggest i such that $r \geq i \cdot c$.
- Let $(q_F, r_F) = H(0,b) = G^k(0,b)$. Then k is the biggest i such that $b \geq i \cdot c$, $q_F = k$, $r_F = b - k \cdot c$.
- Hence q_F is the quotient of integer division, i.e. $q_F = b \div c$, and r_F is the reminder of $b \div c$, i.e. $r_F = b - q_F \cdot c$.

Problem

Use the checklist to prove that the annotation in this program is correct.

$\{Q: \ \ b \geq 0 \ \wedge \ c > 0\}$

$q, r \ := \ 0, b;$

$\{invariant\ P : b = q \cdot c + r \ \wedge \ 0 \leq r\}$

$\text{do } r \geq c \longrightarrow q, r \ := \ q + 1, r - c \text{ od}$

$\{R: \ \ b = q \cdot c + r \ \wedge \ 0 \leq r < c\}$

$\{Q: \ b \geq 0 \ \wedge \ c > 0\}$
$q, r \ := \ 0, b;$
$\{\text{invariant } P : b = q \cdot c + r \ \wedge \ 0 \leq r\}$
do $r \geq c \longrightarrow q, r \ := \ q + 1, r - c$ od
$\{R: \ b = q \cdot c + r \ \wedge \ 0 \leq r < c\}$

(a) We need to prove $Q \Rightarrow P[q, r := 0, b]$.

$$
\begin{aligned}
& P[q, r := 0, b] \\
= \ & \langle \text{ Definition of } P\text{; textual substitution} \rangle \\
& b = 0 \cdot c + b \wedge 0 \leq b \\
\Leftarrow \ & \langle \text{Arithmetic; definition of } Q \ \rangle \\
& Q
\end{aligned}
$$

(b) $\{P \wedge B\}S\{P\}$, hence we have to prove $P \wedge B \Rightarrow P[q, r := q + 1, r - c]$.

$$
\begin{aligned}
& P[q, r := q + 1, r - c] \\
= \ & \langle \text{Definition of } P \text{ and textual substitution} \rangle \\
& b = (q + 1) \cdot c + (r - c) \wedge 0 \leq r - c \\
= \ & \langle \text{Arithmetic} \rangle \\
& b = q \cdot c + r \wedge r \geq c \\
\Leftarrow \ & \langle \text{Definition of } P \text{ and } B \rangle \\
& P \wedge B
\end{aligned}
$$

(c) Note that each iteration decreases r by c ($c > 0$), so that after a finite number of iterations $r < c$ is achieved.

(d) $P \wedge \neg B \Rightarrow$ is obvious. So we are done.

Final (Almost) Comment

- The first example (Factorial) is probably easier to mechanize.
- To make this technique feasible for bigger, more realistic programs, we need a tool that would be able to do all those symbolic calculations.
- The reasoning presented above rely heavily on Lemma 1(3) and is rather typical for human beings.
- Many steps and observations are not easy to mechanize.
- Nevertheless, this technique has most likely better prospects to eventually lead to almost automatic theorem provers (at least for some special kind of programs), than Hoare Logic.
- On the other hand, for human beings skillful in finding loop invariant, Hoare Logic is probably more convenient.

Finite Induction

- Consider the program:

```
factorial-1000
local i, fac
    i:=1;
    fac:=1;
    while i < 1000 do
    begin
       i:=i+1
       fac:=fac*i;
    end;
end proc;
```

- It is relatively easy using almost every theorem prover that factorial-1000 $= 1000!$, or in fact that fectorial-$n = n!$ for any *constant* n.

- Consider the following program Pr:

$$
\begin{aligned}
b, c &:= 73458, 73;\\
q, r &:= 0, b;\\
\text{do } r \geq c \longrightarrow q, r &:= q+1, r-c \text{ od}
\end{aligned}
$$

- It is relatively easy using almost every theorem prover that for $b = 73458, c = 73$, the program Pr calculate proper quotient and reminder, i.e. the program ends with $q = 1006$ and $r = 20$. In fact it can be proved for any constants b and c.
- A proving properties of programs software developed using Maple can deal with both Factorial and quotient/reminder cannot prove correctness of some sorting procedures for a variable n, which is the size of the data to be sorted.
- However it can handle all cases when n is fixed, for example $n = 6758$, etc.
- The same is true for other similar software.

- In science, laws of nature are proved by conducting a finite number of experiments.
- We may say that in science we use *finite induction*.
- Would you trust a given sorting procedure that was *proven correct* for several different values n?
- How does it differ from *testing*?
- *Testing*: You would test several random sequences and verify if they were really sorted correctly. Quite often programs works correctly for most of inputs but not for all.
- *Verifying by Finite Induction*: You have formal proofs that a program is correct for some specific constant parameters. From this you conclude that is works correctly for all values of these parameters.
- I believe that verification by finite induction is more trustworthy than testing.