UNIVERSITÉ DE LORRAINE

TELECOM nancy
Ingénieurs du numérique • Inspiring your digital future

MVSI Course
Modelling and Verification of Computer Systems

Lecture 2 – Modelling, Specification and Verification (II)

Dominique Méry
Telecom Nancy, Université de Lorraine
November 20, 2026

Plan

1. The PlusCal language
   - Checking a program or an algorithm using TLA$^+$ and PlusCal
   - Annotation of PlusCal programs
   - Define algorithms and processes in PlusCal
   - Macros and procedures
   - Macros in PlusCal
   - Procedures in PlusCal
   - TLA$^+$ and PlusCal in VS Code
2. Summation of the n first integers
3. Induction principle(s)
4. Method for proving invariance properties

Part 1

The PlusCal language

PlusCal: an algorithmic language embedded in TLA/TLA$^+$

- Definition of a simple algorithmic language.
- Specific comments within the text of the TLA$^+$ module, between (* and *)
- Definition of an algorithm as a sequence of definitions of sequential processes that may share variables:

  `--algorithm name { definitions }`
- Generation of a TLA$^+$ specification with the introduction of a new variable ‘pc‘ modelling control.
- Introduction of a variable ‘`pc`‘ to model the control of the algorithm.
- The ToolBox tool has a translation feature.

Computing the quotient and the remainder of two naturals

```
1  /* data  structure */
2  struct result {
3      int quotient;
4      int reste;
5  };
6  /* Function for euclidian  division */
7  struct result division(int x, int y)
8  {
9      struct result  r;
10     r.quotient = 0;
11     r.reste = x;
12
13     while (r.reste >= y)
14     {
15         r.reste = r.reste - y;
16         r.quotient = r.quotient + 1;
17     }
18     return r;
19 }
```

Verification by annotation checking

- Translating the function called *division* as an algorithmic fragment.

Listing 1: Module TLA+

```
1  ---------------  MODULE division0 ------------
2  EXTENDS TLC, Integers, Naturals
3  CONSTANTS x0,y0,M,N
4  ----------------------------------------------
5  ASSUME x0\in 1..M /\ y0 \in 1..N
6  ----------------------------------------------
7  (*
8  --algorithm division {
9  variables x = x0,y=y0,quotient = 0, reste = x;
10 {
11     while (reste >= y)
12     {
13         reste := reste - y;
14         quotient := quotient + 1;
15     }
16 }}
17 *)
18
19 ==============
```

Verification by annotation checking

- Assuming that x0 and y0 are in two intervals defined the two constants M and N.
- Preconditions, Postconditions and Safety Properties are defined as follow:

Listing 2: Module TLA+

```
1
2
3
4  pre == x0 \in Nat /\ y0 \in Nat /\ y0#0
5  post ==
6       /\ pre /\ x=x0 /\ y=y0
7       /\ x0 = y0 * quotient + reste
8       /\ 0 \leq reste /\ reste < y0
9  D(X) == X \in -M..M
10 safetypc == pc="Done" => post
11 safetyrte == D(x) /\ D(y) /\ D(quotient) /\ D(reste)
12 safecontrol == pc \in {"Lbl_1","Done"}
```

Verification by annotation checking

- Using the translation PCAL of the toolbox produces the following TLA$^+$ module.

Verification by annotation checking

- Using the translation PCAL of the toolbox produces the following TLA$^+$ module.

Listing 4: Module TLA+

```
1
2  Lbl_1 == /\ pc = "Lbl_1"
3           /\ IF reste >= y
4                 THEN /\ reste' = reste - y
5                      /\ quotient' = quotient + 1
6                      /\ pc' = "Lbl_1"
7                 ELSE /\ pc' = "Done"
8                      /\ UNCHANGED << quotient, reste >>
9           /\ UNCHANGED << x, y >>
10
11 (* Allow infinite stuttering to prevent deadlock on termination. *)
```

Verification by annotation checking

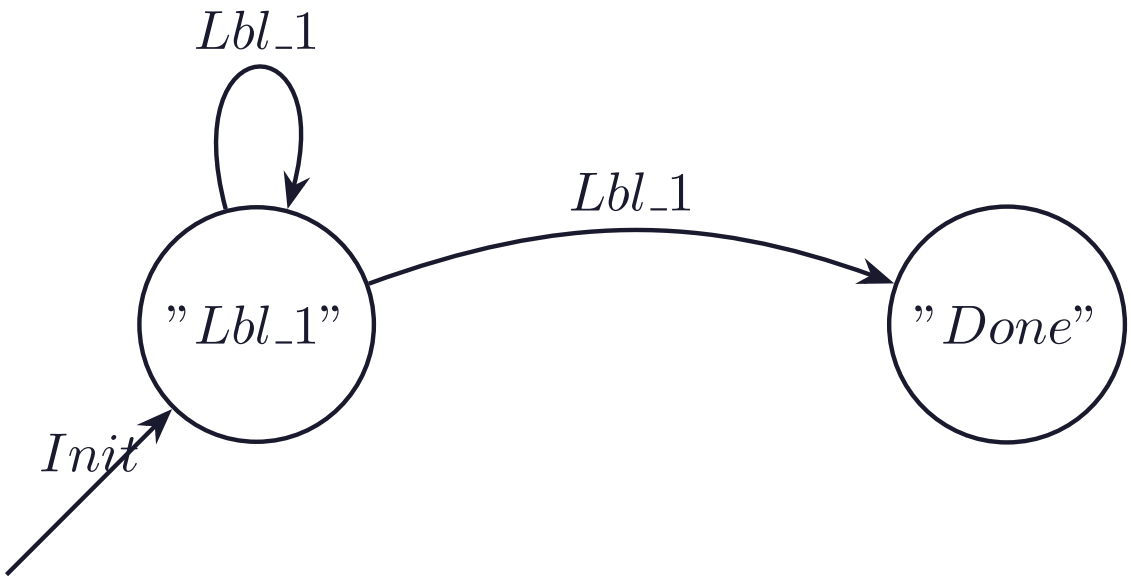
- Using the translation PCAL of the toolbox produces the following TLA$^+$ module.

Listing 5: Module TLA+

```
1  (* Allow infinite stuttering to prevent deadlock on termination. *)
2  Terminating == pc = "Done" /\ UNCHANGED vars
3
4  Next == Lbl_1
5             \/ Terminating
6
7  Spec == Init /\ [][Next]_vars
8
9  Termination == <>(pc = "Done")
10
11 \* END TRANSLATION
```

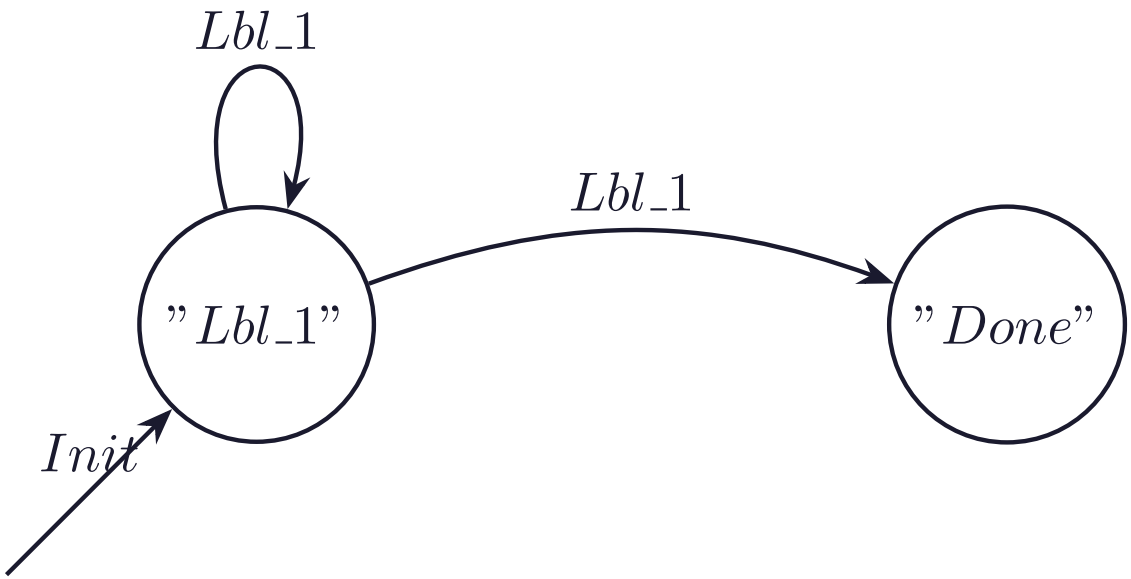
The implicit labelling

- Lbl_1 is a label assigned to a while loop.
- The translation process adds a new label called `Done` and actions are called using the labels



The implicit labelling

- Lbl_1 is a label assigned to a while loop.
- The translation process adds a new label called `Done` and actions are called using the labels



Listing 7: Module TLA+

```
1 safetypc == pc="Done" => post
2 safetyrte == D(x) /\ D(y) /\ D(quotient) /\ D(reste)
```

The implicit labelling

Listing 8: Module TLA+

```
1  ---------------  MODULE label1 ------------
2  EXTENDS TLC, Integers, Naturals
3  CONSTANTS x0,y0
4  ----------------------------------------------
5  (*
6  --algorithm ex {
7  variables x = x0,y=y0;
8  {
9      while (x # 0)
10     {
11         while (y # 0)
12         {
13         y:=y-1;
14         };
15         x:=x-1;
16     }
17     }
18     }
19
20 *)
```

The implicit labelling

Listing 9: Module TLA+

```
\* BEGIN TRANSLATION (chksum(pcal) = "3ad78380" /\ chksum(tla) = "
    a33d9efe")
VARIABLES x, y, pc

vars == << x, y, pc >>

Init == (* Global variables *)
        /\ x = x0
        /\ y = y0
        /\ pc = "Lbl_1"
```

The implicit labelling

Listing 10: Module TLA+

```
1
2  Lbl_1 == /\ pc = "Lbl_1"
3           /\ IF x # 0
4                 THEN /\ pc' = "Lbl_2"
5                 ELSE /\ pc' = "Done"
6           /\ UNCHANGED << x, y >>
```

The implicit labelling

Listing 11: Module TLA+

```
1
2  Lbl_2 == /\ pc = "Lbl_2"
3           /\ IF y # 0
4                 THEN /\ y' = y-1
5                      /\ pc' = "Lbl_2"
6                      /\ x' = x
7                 ELSE /\ x' = x-1
8                      /\ pc' = "Lbl_1"
9                      /\ y' = y
```

The implicit labelling

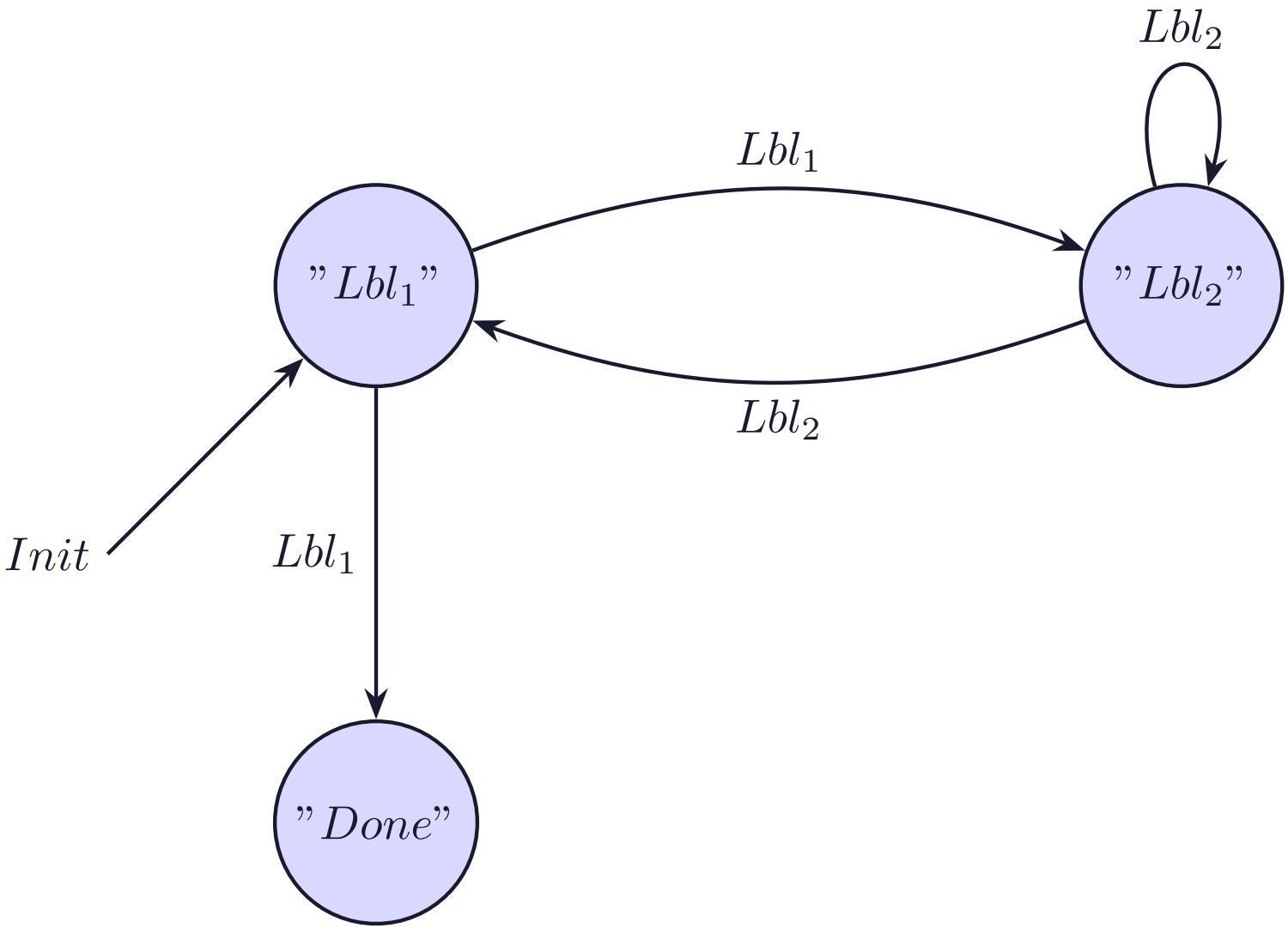
Listing 12: Module TLA+

```
1
2  (* Allow infinite stuttering to prevent deadlock on termination. *)
3  Terminating == pc = "Done" /\ UNCHANGED vars
4
5  Next == Lbl_1 \/ Lbl_2
6              \/ Terminating
7
8  Spec == Init /\ [][Next]_vars
9
10 Termination == <>(pc = "Done")
11
12 \* END TRANSLATION
```

The implicit labelling

- The set of labels is $\{"Lbl_1", "Lbl_2", "Done"\}$.
- Actions are defined as atomic computations between the three labels and each label defines an action from the label to others possible labels.
- Take the example of Lbl_2 either looping or getting out to Lbl_1.

Graph of labels



Explicit labelling process for PlusCal

- Each statement `stmt` can be labelled by a label `lab` and the resulting labelled statement is `lab:stmt`.
- The rule of labelling is to label at least the while statements and possibly to add other observation points to express the control flow.
- Two syntaxes are possible for PlusCal: P-syntax (prolix) and C-syntax (compact).

Explicit labelling process for PlusCal

Listing 13: Module TLA+

```
---- MODULE label2----
EXTENDS Naturals
(* --algorithm Exemple
variables
    i = 0,j = 0,x = 0;
begin
L1:while i < 3 do
L2:         j := 0;
L3:         while j < 3 do
L4:             if i < 2 then
L5:                 if j < 2 then
                        x := x + 1;
                   else
                        x := x + 2;
                   end if;
             else
L6:                 if j = 0 then
                        x := x + 3;
                   else
                        x := x + 4;
                   end if;
             end if;
L7:             j := j + 1;
         end while;
L8:         i := i + 1;
     end while;
```

Example (I)

```
------------------- MODULE exemple1 -------------
EXTENDS Naturals, Integers, TLC
CONSTANTS x0,y0,z0,min,max,undef
-----------------------------------
(* precondition *)
ASSUME x0 = y0 + 3*z0
-----------------------------------
(*
--algorithm exemple1 {
  variables x=x0,
            y = y0,
            z=z0;

{
l0: assert x = y + 3*z/\ /\ y=y0 /\ z=z0 ;
    x := y+3*z;
l1: assert x = y0+3*z0 /\ y=y0 /\ z=z0 ;
}
}
```

Example (II)

```
---------------------- MODULE exemple1 ------------
.............

-----------------------------------------------
ISDEF(X,Y) == X # undef => X \in Y
DD(X) == X # undef => X \in min..max
-----------------------------------------------
i ==
    /\ pc \in {"l0","l1","Done"}
    /\ ISDEF(x,Int)  /\ ISDEF(x,Int) /\ ISDEF(z,Int)
    /\ pc = "l0" => x = y + 3*z
    /\ pc = "l1" => x+y+z \geq y
post ==       x = y0+3*z0 /\ y=y0 /\ z=z0

safetyrte ==DD(x) /\ DD(y) /\ DD(z)
safetypc == pc="Done" => post
=================================================================
```

General Form

Listing 14: Module TLA+

```
1  ---- MODULE module_name ----
2  \* TLA+ code
3
4  (* --algorithm algorithm_name
5  variables global_variables
6
7  process p_name = ident
8  variables local_variables
9  begin
10   \* pluscal code
11 end process
12
13 process p_group \in  set
14 variables local_variables
15 begin
16   \* pluscal code
17 end process
18
19 end algorithm; *)
20 ====
```

Variable sharing and variable scope

- Global variables are declared at the level of the algorithm and are shared by all processes, which may read and write to them in accordance with the atomicity rule.
- Local variables are defined at the level of each process, which is a sequential process.
- Parallelism is achieved through non-deterministic interleaving of the actions of the different processes.
- Control is introduced by the variable `pc`, which is an array of size equal to the number of possible concurrent processes.

Macros and Procedures

```
macro Name(var1, ...)
begin
 \* something to write
end macro;
```

```
procedure Name(arg1, ...)
variables var1 = ... \* not  \in, only =
begin
  Label:
  \* something
  return;
end procedure;
```

Macros: syntax and rules

C-syntax

```
1  macro Name(p1, ..., pk) {
2    stmt1;
3    ...
4    stmtn;
5  }
```

P-syntax

```
1  macro Name(p1, ..., pk)
2  begin
3    stmt1;
4    ...
5    stmtn;
6  end macro;
```

- Macros are declared after the global `variables` (and the `define` section), before the procedures and the processes.
- A call `Name(e1, ..., ek)` is expanded in place: the body is copied and each parameter p_i is replaced by the expression e_i.
- The body of a macro cannot contain labels, nor `while`, `call`, `return` or `goto` statements; `await` is allowed.
- A macro adds no step: its statements belong to the step of the label that precedes the call.

Example: message passing with macros (file tla/Channel.tla)

```
--algorithm Channel {
  variables chan = <<>>,
            output = <<>>;

  macro Send(m, c) {
    c := Append(c, m);
  }
  macro Recv(v, c) {
    await c # <<>>;
    v := Head(c);
    c := Tail(c);
  }
  process (Sender = "S")
    variables i = 1;
  {
  s1: while (i <= 3) {
  s2:   Send(i, chan);
        i := i + 1;
      }
  }
  process (Receiver = "R")
    variables msg = 0;
  {
  r1: while (Len(output) < 3) {
  r2:   Recv(msg, chan);
        output := Append(output,
                  msg);
      };
  r3: assert output = <<1, 2, 3>>;
  }
}
```

What the translator does with a macro

The call `Recv(msg, chan)` at label `r2` is replaced by the body of the macro, with `v` $\mapsto$ `msg` and `c` $\mapsto$ `chan`:

```
1  r2 == /\ pc["R"] = "r2"
2        /\ chan # <<>>                    \* await c # <<>>
3        /\ msg' = Head(chan)              \* v := Head(c)
4        /\ chan' = Tail(chan)             \* c := Tail(c)
5        /\ output' = Append(output, msg')
6        /\ pc' = [pc EXCEPT ![“R”] = “r1”]
7        /\ i' = i
```

- The `await` becomes an enabling condition: the receiver is blocked while the channel is empty.
- The whole step `r2` stays atomic: receiving and appending the message happen in one transition.
- TLC: no error, 42 distinct states; the assertion `output = <<1, 2, 3>>` holds (the channel is FIFO).

Procedures: syntax and rules

C-syntax

```
1  procedure Name(p1, ..., pk)
2    variables v1 = e1, ...;
3  {
4    L1: stmt;
5        ...
6        return;
7  }
```

P-syntax

```
1  procedure Name(p1, ..., pk)
2    variables v1 = e1, ...;
3  begin
4    L1: stmt;
5        ...
6        return;
7  end procedure;
```

- Declared after the macros, before the processes (or the main body).
- The body must start with a label; local variables are initialised with = only (no `\in`).
- `call Name(e1, ..., ek)` must be followed by a labelled statement, a `return` or a `goto`.
- `return` goes back after the call; there is no returned value: results are passed through global variables.
- Procedures may be recursive; the module must extend `Sequences` (the translation uses a stack).

Example: a recursive procedure (file tla/Fact.tla)

```
1  EXTENDS Naturals, Sequences, TLC
2  (* --algorithm Fact {
3    variables result = 1;
4
5    procedure fact(n) {
6    f1: if (n > 1) {
7          result := result * n;
8          call fact(n - 1);
9          return;
10       } else {
11         return;
12       }
13   }
14
15   {
16   m1: call fact(5);
17   m2: assert result = 120;
18   }
19 } *)
```

- The main body calls `fact(5)`.
- `result` is global: it carries the result $5 \times 4 \times 3 \times 2 = 120$.
- `call fact(n - 1);` `return;` is a terminal call: the translator turns it into a jump to `f1`.
- TLC: no error, 8 distinct states.
- The configuration file gives a value to the constant `defaultInitValue`: `CONSTANT defaultInitValue = defaultInitValue`

Translation of a procedure call: the stack

```
1  VARIABLES pc, result, stack, n
2  m1 == /\ pc = "m1"
3        /\ /\ n' = 5
4           /\ stack' = << [ procedure |-> "fact",
5                            pc        |-> "m2",
6                            n         |-> n ] >> \o stack
7        /\ pc' = "f1"
8        /\ UNCHANGED result
9  f1 == /\ pc = "f1"
10       /\ IF n > 1
11             THEN /\ result' = result * n
12                  /\ n' = n - 1 /\ pc' = "f1" /\ stack' = stack
13             ELSE /\ pc' = Head(stack).pc
14                  /\ n' = Head(stack).n
15                  /\ stack' = Tail(stack)
16                  /\ UNCHANGED result
```

- `call`: push the return label and the saved parameters, then jump to the first label of the procedure.
- `return`: restore them from `Head(stack)` and pop.

One stack per process (file tla/Incr.tla)

```
1  --algorithm Incr {
2    variables x = 0;
3
4    procedure incr()
5      variables tmp = 0;
6    {
7    rd: tmp := x;
8    wr: x := tmp + 1;
9        return;
10   }
11
12   process (P \in 1..2) {
13   p1: call incr();
14   p2: skip;
15   }
16 }
```

- `stack`, `pc` and `tmp` become functions indexed by the processes: `stack[self]`, `tmp[self]`.
- The labels `rd` and `wr` split the increment into two atomic steps, which may interleave.
- `Bounded`: at the end, $x \in \{1, 2\}$: holds.
- `LostUpdate`: at the end, $x = 2$: violated; TLC gives the trace where both processes read $x = 0$ before writing (file `IncrLost.cfg`).

Macro or procedure?

	macro	**procedure**
Mechanism	textual expansion at each call	call and return through a stack
Labels in the body	forbidden	required (first statement)
Steps	part of the caller's step	its own steps, may interleave
`while`, `call`, `goto`	forbidden	allowed
Local variables	none	yes, initialised with =
Recursion	no	yes
Result	assigns its arguments	through global variables
Typical use	atomic primitives: send, receive, lock	sub-algorithms, recursion



Use a macro to name an atomic operation; use a procedure when the operation has several steps.

Main commands of the TLA$^+$ extension for VS Code

Extension `tlaplus.vscode-ide`; commands from the palette (`Cmd+Shift+P` on macOS, `Ctrl+Shift+P` otherwise), or by right-clicking in the editor.

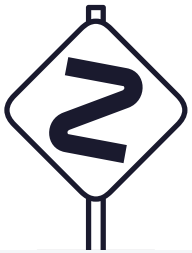
Command	**Effect**
TLA+: Parse module	translates the PlusCal algorithm into TLA$^+$, then parses the module (SANY)
TLA+: Check model with TLC	runs TLC with `M.cfg` for the module `M.tla`
TLA+: Check model with TLC using non-default config...	runs TLC with another `.cfg` file
TLA+: Run last model check again	reruns the last check
TLA+: Stop model checking process	stops TLC
TLA+: Display model checking panel	shows the results and the error trace
TLA+: Evaluate selected expression / Evaluate expression...	evaluates a constant expression
TLA+: Check and debug model with TLC	runs TLC in the debugger (breakpoints)
TLA+: Run REPL in Terminal	interactive TLA$^+$ evaluator
TLA+: Export module to LaTeX / to PDF	pretty-printed module
TLA+: Check proof step in TLAPS	checks a proof (if TLAPS is installed)

Workflow for a PlusCal algorithm in VS Code

1. Write the module `M.tla`; put the algorithm in a comment `(* --algorithm M { ...} *)`.
2. **TLA+: Parse module**: the translation is inserted between `\* BEGIN TRANSLATION` and `\* END TRANSLATION`; errors appear in the *Problems* panel.
3. Write `M.cfg`:

```
SPECIFICATION Spec
INVARIANT Bounded
CONSTANT defaultInitValue = defaultInitValue
```

4. **TLA+: Check model with TLC**: the panel shows the number of states and, for a violation, the error trace step by step.
5. Correct the algorithm, **Parse module** again, then **Run last model check again**.



Edit the algorithm, never the translation: it is regenerated at each parse.

Part 2

Summation of the n first integers

Annotation versus comment of programs or algorithms

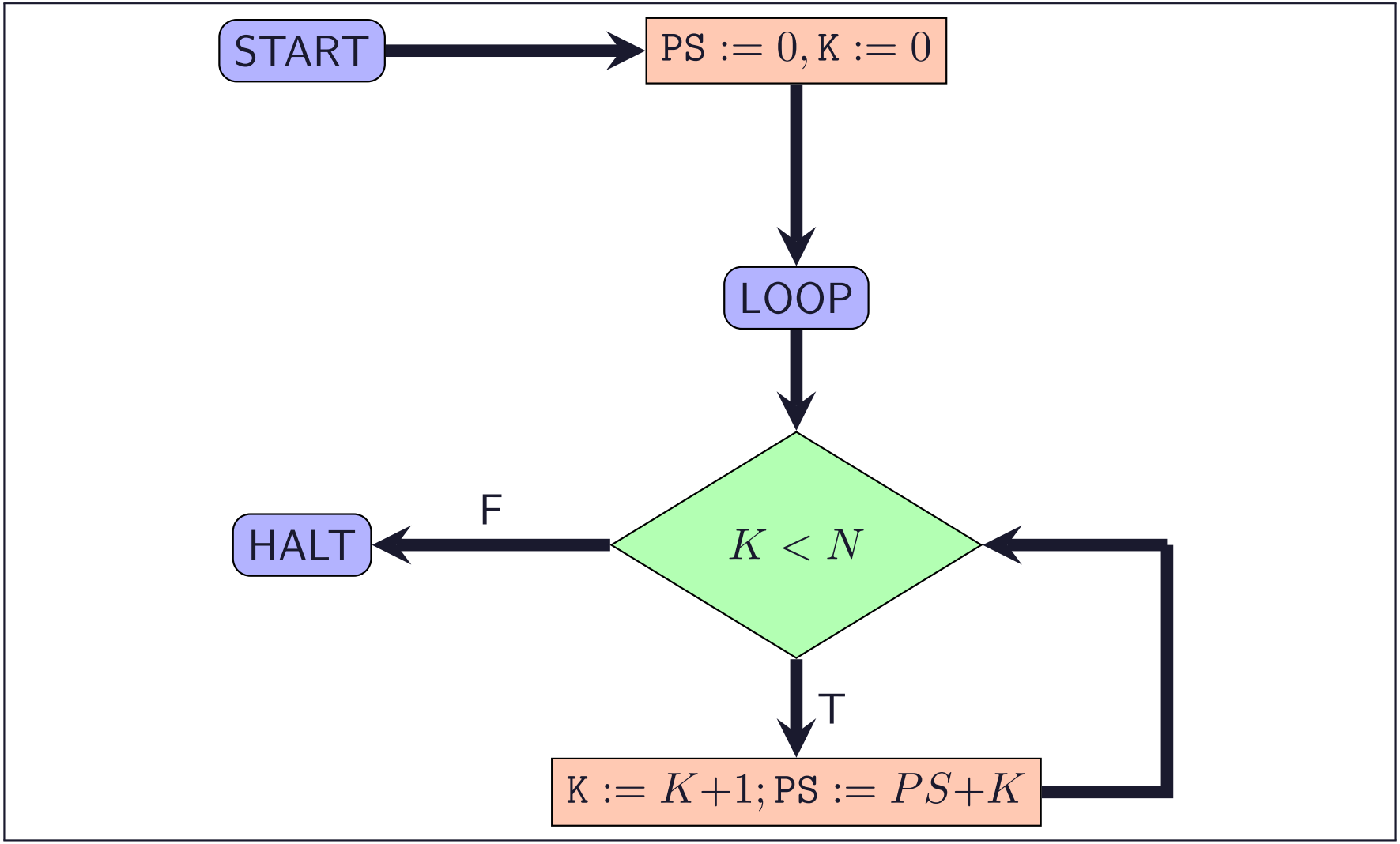
- A program or an algorithm can be annotated or commented.
- A comment is a relevant piece of information intended to be seen or read, and whose importance is relative to the mind of the designer.
- A comment gives information on the data, on the variables and hence on the assumed state of the program at run time.
- A comment is an annotation of the text of the code which lets us communicate semantic information:
  - *at this point, the variable k is smaller than n*
  - *the index e refers to a valid address of t and this value is always positive*
  - *the sum of the variables is positive*
- Annotations can be systematised and follow a specific syntax defining the annotation language;

```
1 /*@  assert l1:  z  >= 3 && y  == 3; */
2 z = z +y;
3 /*@  assert l2:  z  >= 6 && y  == 3; */
```

Computing the sum of the first n integers (v0)

```
1  int fS(int n) {
2    int ps = 0;
3    int k = 0;
4    while (k < n) {
5      k = k + 1;
6      ps = ps + k;
7    };
8    return ps;
9  }
```

Computing the sum of the first n integers (flowchart)



Computing the sum of the first n integers (v0)

```
1  // pre n>=0;
2  // post ps == n*(n+1) / 2;
3
4  int fS(int n) {
5    int ps = 0;
6    int k = 0;
7    while (k < n) {
8      k = k + 1;
9      ps = ps + k;
10   };
11   return ps;
12 }
13
14
15 int main()
16 {
17
18 }
```

Computing the sum of the first n integers (v0)

```
1  // pre n>=0;
2  // post ps == n*(n+1) / 2;
3
4  int fS(int n) {
5    int ps = 0;
6    int k = 0;
7    while (k < n) {
8      // ps = k*(k+1) / 2;
9      k = k + 1;
10     ps = ps + k;
11   };
12     // ps = n*(n+1) / 2;
13   return ps;
14 }
15
16
17 int main()
18 {
19
20 }
```

Computing the sum of the first n integers (v0)

```c
#include <stdio.h>

int fS(int n) {
  int ps = 0;
  int k = 0;
  while (k < n) {
    k = k + 1;
    ps = ps + k;

  };
  return ps;
}

int main()
{
  int   z = 3;
  printf("Value for z=%d is %d\n",z,fS(z));
  return 0;
}
```

Definition of S(n) and IS(n)

$\forall n \in \mathbb{N} : S(n) = \sum_{k=0}^{n} k$

$$\left[\begin{array}{l} IS(0) = 0 \\ n \geq 0, IS(n{+}1) = IS(n){+}(n{+}1) \end{array} \right.$$

Property for S(n) and IS(n)

$\forall n \in \mathbb{N} : S(n) = IS(n)$

- base 0: $S(0) = 0 = IS(0)$
- induction $i{+}1$: $\forall j \in 0..i : S(j) = IS(j)$
  - $S(i{+}1) = \sum_{k=0}^{i+1} k$ (definition)
  - $S(i{+}1) = (\sum_{k=0}^{i} k){+}i{+}1$ (property of summation)
  - $S(i{+}1) = S(i){+}(i{+}1)$ (by definition of S(i))
  - $S(i{+}1) = IS(i){+}(i{+}1)$ (by inductive assumption on S(i) and IS(j))
  - $S(i{+}1) = IS(i){+}(i{+}1) = IS(i{+}1)$ (by definition of IS(i+1))
- conclusion: $\forall i \in \mathbb{N} : S(i) = IS(i)$ (by induction principle)

Algorithmic reformulation of the computation of the sum of the first n integers

- $\forall n \in \mathbb{N} : S(n) = \sum_{k=0}^{n} k$
- $\begin{cases} IS(0) = 0 \\ n \geq 0, IS(n{+}1) = IS(n){+}n \end{cases}$
- $\forall n \in \mathbb{N} : S(n) = IS(n)$
-
  - base 0: $S(0) = 0$
  - induction $k{+}1$: $S(k{+}1) = S(k){+}i{+}1$
  - step $k{+}1$: $S(k{+}1) = S(k){+}k{+}1$
- $S(k) = ps$: current value of ps is S(k)
- $S(k{-}1) = ops$: current value of ops is $S(k{-}1)$
- step $k{+}1$: $ps = ops{+}k{+}1$

Computing the sum of the first n integers (v1)

```c
#include <stdio.h>

int fS(int n) {
  int ps = 0;
  int k = 0;
  int ok=k, ops = 0;
  while (k < n) {
    ok=k;ops=ps;
    k = ok + 1;
    ps = ops + k;
  };
  return ps;
}

int main()
{
  int   z = 3;
  printf("Value for z=%d is %d\n",z,fS(z));
  return 0;
}
```

Computing the sum of the first n integers (v2)

```
1
2  int fS(int n) {
3    int ps = 0;
4    int k = 0;
5    int ok=k, ops = 0;
6    while (k < n) {
7   /*@ assert 0 <= k && k <= n
8     && ps == S(k) && ops == S(ok);    */
9      ok=k;ops=ps;
10     k = ok + 1;
11     ps = ops + k;
12  /*@ assert 0 <= k && k <= n && ps == S(k)
13    && ops == S(ok);    */
14   };
15   return ps;
16 }
```

Computing the sum of the first n integers

```
1
2  /*@ axiomatic S {
3    @ logic integer S(integer n);
4    @ axiom S_0: S(0) == 0;
5    @ axiom S_i: \forall integer i; i > 0 ==> S(i) == S(i-1)+i;
6    @ } */
7
8  /*@ requires n >= 0;
9    assigns \nothing ;
10   ensures \result == S(n);
11 */
12 int fS(int n) {
13   int ps = 0;
14   int k = 0;
15   int ok=k,ops=ps;
16   /*@ loop invariant 0 <= k && k <= n && ps == S(k) && ops == S(ok)
         ;
17     loop assigns ps, k,ops,ok;
18    */
19   while (k < n) {
20 /*@ assert l0: 0 <= k && k <= n && ps == S(k) && ops == S(ok);   */
21     ops=ps;ok=k;
22      k = ok + 1;
23     ps = ops + k;
24  /*@ assert l1:  0 <= k && k <= n && ps == S(k) && ops == S(ok);
       */
25   };
26  /*@ assert  ps == S(n);   */
```

Observations on the computation

- Definition of the mathematical functions needed to express the computation of the sum of the first n integers.
- Expression of the intermediate results, called *partial sums*
- Relation between the proof by induction and the form of the body of the iteration.
- Induction and computation are linked.

Observations on the computation

- Definition of the mathematical functions needed to express the computation of the sum of the first n integers.
- Expression of the intermediate results, called *partial sums*
- Relation between the proof by induction and the form of the body of the iteration.
- Induction and computation are linked.

$$x_0 \xrightarrow{P} x \tag{1}$$

$$x_0 \xrightarrow{\star} x \tag{2}$$

$$x_0 \longrightarrow x_1 \longrightarrow \ldots \longrightarrow x \tag{3}$$

$$x_0 \longrightarrow x_1 \longrightarrow \ldots \longrightarrow x_n \xrightarrow{step} x \tag{4}$$

Part 3

Induction principle(s)

Some simplifications and notations



We adopt the following equivalent notations: $x \in E$ is equivalent to $E(x)$ for every value $x \in$ Vals. This simplification relates a set $U \subseteq$ Vals to an assertion $U(x)$, considering that $U(x)$ and $x \in U$ denote the same concept.

The two following expressions are equivalent:

- $\forall x_0, x \in \textsc{Vals}. Init(x_0) \wedge \textsc{Next}^\star(x_0, x) \Rightarrow A(x)$
- $\forall x \in \textsc{Vals}. (\exists x_0. x_0 \in \textsc{Vals} \wedge Init(x_0) \wedge \textsc{Next}^\star(x_0, x)) \Rightarrow A(x)$

- $\forall x_0, x \in \textsc{Vals}. Init(x_0) \wedge \textsc{Next}^\star(x_0, x) \Rightarrow A(x)$
- $\forall x \in \textsc{Vals}. (\exists x_0. x_0 \in \textsc{Vals} \wedge Init(x_0) \wedge \textsc{Next}^\star(x_0, x)) \Rightarrow A(x).$
- $\textsc{reachable}(M) = \{u | u \in \textsc{Vals} \wedge (\exists x_0. x_0 \in \textsc{Vals} \wedge Init(x_0) \wedge \textsc{Next}^\star(x_0, u))\}$ is the set of states reachable from the initial states, and the safety property $A(x)$ must be shown by showing the inclusion of the sets (model-checking):

$$\textsc{reachable}(M) \subseteq \{u | u \in \textsc{Vals} \wedge A(u)\}$$

Assertional induction principle

Let $(Th(s,c), x, \textsc{Vals}, Init(x), \{r_0, \ldots, r_n\})$ be a relational model M of a system S.
A property $A(x)$ is a safety property for the system S, if and only if there exists a state property $I(x)$, such that:

$$\forall x, x' \in \textsc{Vals} \ : \begin{cases} (1)\ Init(x) \Rightarrow I(x) \\ (2)\ I(x) \Rightarrow A(x) \\ (3)\ I(x) \wedge \textsc{Next}(x, x') \Rightarrow I(x') \end{cases}$$

The property $I(x)$ is called an inductive invariant of S and is a particular safety property, stronger than the other safety properties.

Justification (soundness)

Let $I(x)$ be a property such that:

$$\forall x, x' \in \textsc{Vals} \ : \begin{cases} (1)\ Init(x) \Rightarrow I(x) \\ (2)\ I(x) \Rightarrow A(x) \\ (3)\ I(x) \wedge \textsc{Next}(x, x') \Rightarrow I(x') \end{cases}$$

Then $A(x)$ is a safety property for the system S modelled by M.

Let x and $x' \in \textsc{Vals}$ such that $INIT(x) \wedge \textsc{Next}^{\cdot}(x, x')$.

- We can build a sequence such that:
  $(x = x_0) \xrightarrow[\textsc{Next}]{} x_1 \xrightarrow[\textsc{Next}]{} x_2 \xrightarrow[\textsc{Next}]{} \ldots \xrightarrow[\textsc{Next}]{} (x_i = x')$.
- Assumption (1) lets us deduce $I(x_0)$.
- Assumption (3) lets us deduce $I(x_1)$, $I(x_2)$, $I(x_3)$, ..., $I(x_i)$. Using assumption (2) for x', we deduce that x' satisfies A.

Justification (completeness)

$\forall x_0, x \cdot x, y \in \textsc{Vals} \wedge Init(x_0) \wedge x_0 \xrightarrow[\text{Next}]{\star} x \Rightarrow A(x)$

Let us prove that: there exists a property $I(x)$ such that:

$$\forall x, x' \in \textsc{Vals} \ : \begin{cases} (1)\ Init(x) \Rightarrow I(x) \\ (2)\ I(x) \Rightarrow A(x) \\ (3)\ I(x) \wedge \textsc{Next}(x, x') \Rightarrow I(x') \end{cases}$$

- We consider the following property:
  $I(x) \ \widehat{=} \ \exists x_0 \in \textsc{Vals} \cdot Init(x_0) \wedge x_0 \xrightarrow[\text{Next}]{\star} x.$
- $I(x)$ expresses that the value x is reachable from an initial value x_0.
- The three properties are easy to check for $I(x)$. $I(x)$ is called the strongest invariant of the algorithm $\mathcal{A}$.

Assertion versus relation

- P. and R. Cousot carried out a complete study of invariance and safety properties, highlighting correspondences between the different methods or systems proposed by Turing, Floyd, Hoare, Wegbreit, Manna . . . and reformulated the induction principles used to define these proof methods (see the two cubes of the 16 principles).
- Two kinds of principles are proposed: assertional and relational.
- We use the expression safety property, whereas it is generally an invariance property ($\Box$ property), and invariant instead of inductive invariant.

Consequences on practical methods

- The invariant property I is defined by
  $\mathsf{I}(x) \stackrel{def}{=} \exists x_0 \in \textsc{Vals}.Init(x_0) \wedge \textsc{Next}^\star(x_0, x)$
- The invariant property R is defined by
  $\mathsf{R}(x_0, x) \stackrel{def}{=} Init(x_0) \wedge \textsc{Next}^\star(x_0, x)$

Relation between the two kinds of properties, assertional and relational

- $\forall x_0, x \in \textsc{Vals}. Init(x_0) \wedge \textsc{Next}^\star(x_0, x) \Rightarrow R(x_0, x)$ (R) is a relational safety property.
- Let $y = (x_0, x)$, $y_0 = (x_0, x_0)$, and $\textsc{NextR}(y, y') \stackrel{def}{=} \textsc{Next}(x, x') \wedge y = (x_0, x) \wedge y' = (x_0, x')$
- (R) is rewritten as follows:
  $\forall y_0, y \in \textsc{Vals}\times\textsc{Vals}. Init(x_0) \wedge y_0 = (x_0, x_0) \wedge \textsc{NextR}^\star(y_0, y) \Rightarrow R(y)$ (R)
- By the soundness and completeness property
- there exists a state property $IR(y)$, such that:

$$\forall y_0, y \in \textsc{Vals}\times\textsc{Vals}. \begin{cases} (1)\ Init(x_0) \wedge y_0 = (x_0, x_0) \Rightarrow IR(y) \\ (2)\ IR(y) \Rightarrow R(y) \\ (3)\ IR(y) \wedge \textsc{NextR}(y, y') \Rightarrow IR(y') \end{cases}$$

- there exists a relational property $IR(x_0, x)$, such that:

$$\forall x_0, x \in \textsc{Vals}. \begin{cases} (1)\ Init(x_0) \wedge \Rightarrow IR(x_0, x) \\ (2)\ IR(x_0, x) \Rightarrow R(x_0, x) \\ (3)\ IR(x_0, x) \wedge \textsc{Next}(x, x') \Rightarrow IR(x_0, x') \end{cases}$$

Hence we obtain two kinds of induction principles, according to the safety properties.

Assertional induction principles

Completeness and soundness

$\forall x_0, x \in \textsc{Vals}. Init(x_0) \wedge \textsc{Next}^\star(x_0, x) \Rightarrow A(x).$

if, and only if,

there exists $I \in \mathcal{P}(\textsc{Vals})$

$$\forall x_0, x, x' \in \textsc{Vals} \ : \begin{cases} (1)\ Init(x_0) \Rightarrow I(x_0) \\ (2)\ I(x) \Rightarrow A(x) \\ (3)\ I(x) \wedge \textsc{Next}(x, x') \Rightarrow I(x') \end{cases}$$

- The absence of run-time errors is characterised as an assertional property, since it states that a state is free of run-time errors if the computations are defined in this state. principle

Relational induction principles

Completeness and soundness

$$\forall x_0, x \in \textsc{Vals}. Init(x_0) \wedge \textsc{Next}^\star(x_0, x) \Rightarrow A(x_0, x).$$

if, and only if,

there exists $R \in \mathcal{P}(\textsc{Vals} \times \textsc{Vals})$

$$\forall x_0, x, x' \in \textsc{Vals} \ : \begin{cases} (1)\ Init(x_0) \Rightarrow R(x_0, x_0) \\ (2)\ R(x_0, x) \Rightarrow A(x_0, x) \\ (3)\ R(x_0, x) \wedge \textsc{Next}(x, x') \Rightarrow R(x_0, x') \end{cases}$$

- Partial correctness is characterised as a relation between the initial state and the current state.

Summary on the two induction principles

Assertional principle of safety or invariance

$$\exists I(x) \in \mathcal{P}(\textsc{Vals}).\ \left[\begin{array}{l} \forall x, x' \in \textsc{Vals}. \\ (1)\quad Init(x) \Rightarrow I(x) \\ (2)\quad I(x) \Rightarrow A(x) \\ (3)\quad I(x) \wedge \textsc{Next}(x, x') \Rightarrow I(x') \end{array}\right.$$

Relational principle of safety or invariance

$$\exists IR(x_0, x) \in \mathcal{P}(\textsc{Vals} \times \textsc{Vals}).$$
$$\left[\begin{array}{l} \forall x_0, x, x' \in \textsc{Vals}. \\ (1)\quad Init(x_0) \Rightarrow IR(x_0, x_0) \\ (2)\quad IR(x_0, x) \Rightarrow R(x_0, x) \\ (3)\quad IR(x_0, x) \wedge \textsc{Next}(x, x') \Rightarrow IR(x_0, x') \end{array}\right.$$

Part 4

Method for proving invariance properties

Designing a method for proving invariance and safety properties for a programming language

- We consider a classical programming language denoted $\textsc{Programs}$
- and we assume that this programming language provides assignment, conditional, bounded iteration, unbounded iteration, simple or structured variables such as arrays, and the definition of constants.
- We are given a program P of $\textsc{Programs}$; this program contains
  - variables globally denoted v,
  - constants globally denoted pc,
  - types associated with the variables, globally denoted $\textsc{Vals}$ and identified with a set of possible values of the variables,
  - statements following an order defined by the syntax of the programming language.

Assumptions

- we define a set of control points $\textsc{Locations}$
- for each program or algorithm P. $\textsc{Locations}$ is a finite set of values, and a hidden variable denoted pc ranges over this set following the control flow.
- the space of possible values $\textsc{Vals}$ is a Cartesian product of the form $\textsc{Locations} \times \textsc{Memory}$
- the variables x of the system are split into two independent entities $x = (pc, v)$ with the conditions $pc \in \textsc{Locations}$ and $v \in \textsc{Memory}$.

$$x = (pc, v) \wedge pc \in \textsc{Locations} \wedge v \in \textsc{Memory} \tag{5}$$

We consider an annotated program P; we are given a relational model $\mathcal{MP} = (Th(s, c), x, \textsc{Vals}, \textsc{Init}(x), \{r_0, \ldots, r_n\})$ where

- $Th(s, c)$ is a theory defining the sets, the constants and the static properties of this program
- x is a list of flexible variables, and x has a control part and a memory part.
- $\textsc{Locations} \times \textsc{Memory}$ is a set of possible values for x.
- $\{r_0, \ldots, r_n\}$ is a finite set of relations linking the values before x and the values after x', consistent with the successor relation $\longrightarrow$ between the control points.
- $\textsc{Init}(x)$ defines the set of initial values of (pc_0, v), and $x = (pc_0, v)$ with $pre(v)$ which characterises the initial values of v at the initial point.

Assumptions on the computation

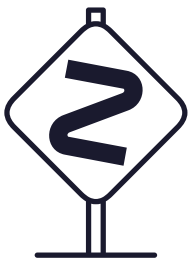
We assume that there is a graph on the set of control values defining the flow relation, and we denote this structure by $(\text{LOCATIONS}, \longrightarrow)$.

⊠ Definition

$\ell_1 \longrightarrow \ell_2 \stackrel{def}{=} pc = \ell_1 \wedge pc' = \ell_2$

⊠ Definition(Annotation of a control point)

Let $(\text{LOCATIONS}, \longrightarrow)$ be a structure and $\ell \in \text{LOCATIONS}$ a label. An annotation of a control point ℓ is a predicate $P_\ell(v)$ (assertional version) or $P_\ell(v_0, v)$ (relational version).



$P_\ell(v_0, v)$ expresses a relation between the initial value of V, denoted v_0, and v, the current value of V at point ℓ, and hence $P_\ell(v_0, v) \Rightarrow pre(v_0)$ states that v_0 is an initial value.

Relation between annotation and invariant

- The labels ℓ belong to $\textsc{Locations}$: $\ell \in \textsc{Locations}$.
- The variables v belong to $\textsc{Memory}$: $v \in \textsc{Memory}$.
- $pre(v_0)$ specifies the initial values of v.
- Whenever control is at ℓ, v satisfies $P_\ell(v)$: $pc = \ell \Rightarrow P_\ell(v_0, v)$.
- In every state (ℓ, v) of the program, the following property is true but must be proved:

$$J(\ell_0, v_0, pc, v) \stackrel{def}{=} \left[\begin{array}{l} \wedge\ pc \in \textsc{Locations} \\ \wedge\ v \in \textsc{Memory} \\ \ldots \\ \wedge\ pc = \ell \Rightarrow P_\ell(v_0, v) \\ \ldots \end{array}\right.$$

- $J(\ell_0, v_0, pc, v)$ is an invariant built from the produced annotations, but we must show that this invariant allows us to check the three conditions of the induction principle.



ℓ_0 denotes the label marking the beginning of the algorithm and ℓ_f is the end of the program. We may simply use 0 and f.

Verification to be done

$$x = (pc, v) \text{ and } J(\ell_0, v_0, pc, v) \stackrel{def}{=} \left[\begin{array}{l} \wedge\ pc \in \textsc{Locations} \\ \wedge\ v \in \textsc{Memory} \\ \ldots \\ \wedge\ pc = \ell \Rightarrow P_\ell(v_0, v) \\ \ldots \end{array}\right.$$

Let $(Th(s,c), x, \textsc{Vals}, \textsc{Init}(x), \{r_0, \ldots, r_n\})$ be a relational model for this program. A property $\mathrm{A}(x_0, x)$ is a safety property for P, if $\forall x_0, x \in \textsc{Locations} \times \textsc{Memory}. Init(x_0) \wedge x_0 \xrightarrow[\textsc{Next}]{} x \Rightarrow A(x)$. We know that this property implies that there exists a state property $\mathrm{I}(x_0, x)$ such that the three properties hold, but we apply this verification to J:

$\forall x_0, x, x' \in \textsc{Locations} \times \textsc{Memory}$:

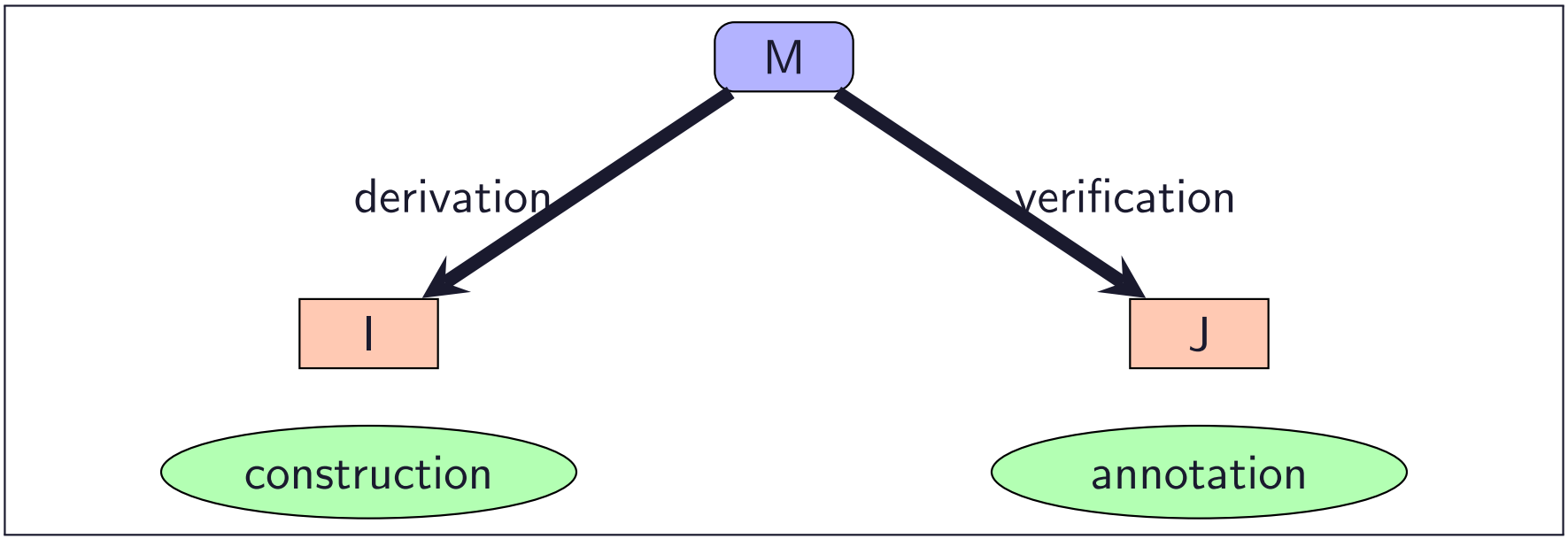
$$\left\{\begin{array}{ll} (1) & \textsc{Init}(x_0) \Rightarrow \mathrm{J}(x_0, x_0) \\ (2) & \mathrm{J}(x_0, x) \Rightarrow \mathrm{A}(x_0, x) \\ (3) & \forall i \in \{0, \ldots, n\} : \mathrm{J}(x_0, x) \wedge x\ r_i\ x' \Rightarrow \mathrm{J}(x_0, x') \end{array}\right.$$



$\forall i \in \{0, \ldots, n\} : \mathbf{J}(x_0, x) \wedge x\ r_i\ x' \Rightarrow \mathbf{J}(x_0, x')$ **is equivalent to** $\mathbf{J}(x_0, x) \wedge (\exists i \in \{0, \ldots, n\} : x\ r_i\ x') \Rightarrow \mathbf{J}(x_0, x')$

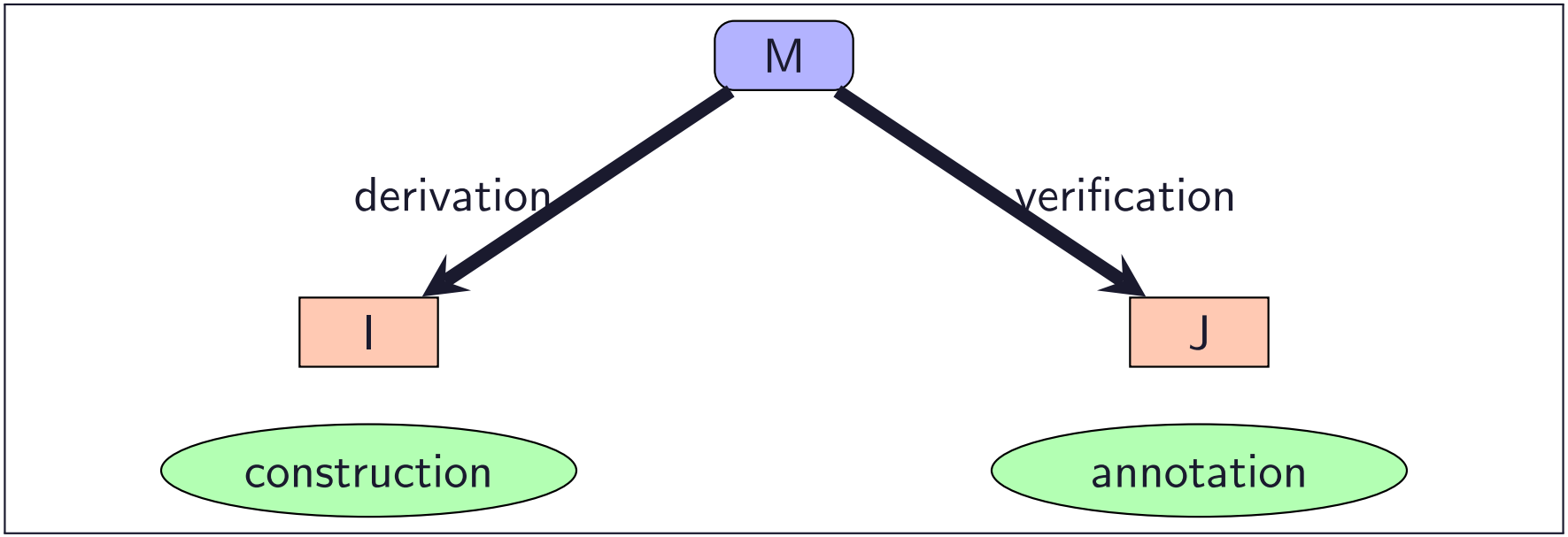
Verification to be done

- Application of the soundness of the relational induction principle: if we check the three properties, then A is a safety property for the model at hand (verification).
- If we want to show that A is a safety property, then we must use the invariant to build annotations for the model (derivation).



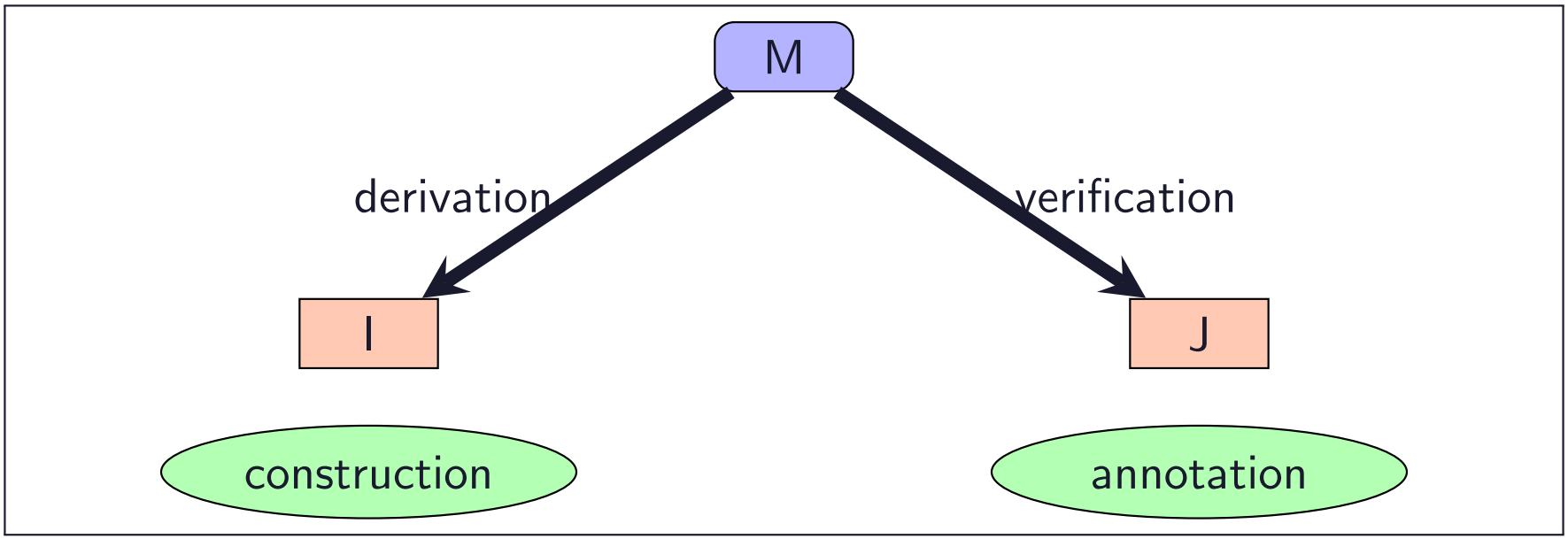
Using the relational induction principle (RI)

- $\textsc{Vals} = \textsc{Locations} \times \textsc{Memory}$
- $J(pc_0, v_0, pc, v) \stackrel{def}{=} \exists x_0, x \in \textsc{Vals}. I(x_0, x) \wedge x = (pc, v) \wedge x_0 = (pc_0, v_0)$ (deduction)
- $I(x_0, x) \stackrel{def}{=} \exists pc_0, pc \in \textsc{Locations}, v_0, v \in \textsc{Memory}. J(pc_0, v_0, pc, v) \wedge x = (pc, v) \wedge x_0 = (pc_0, v_0)$ (induction)



Using the assertional induction principle (AI)

- $\textsc{Vals} = \textsc{Locations} \times \textsc{Memory}$
- $J(pc, v) \stackrel{def}{=} \exists x \in \textsc{Vals}. I(x) \wedge x = (pc, v)$ (deduction)
- $I(x) \stackrel{def}{=} \exists pc \in \textsc{Locations}, v \in \textsc{Memory}. J(pc, v) \wedge x = (pc, v)$ (induction)



Adaptation to programs

(1) $\forall x_0, x \in \textsc{Vals}. Init(x_0) \wedge \textsc{Next}^\star(x_0, x) \Rightarrow A(x)$ $(I(x))$
(2) $\forall x_0, x \in \textsc{Vals}. Init(x_0) \wedge \textsc{Next}^\star(x_0, x) \Rightarrow R(x_0, x)$ $(IR(x))$

Relations and definitions

$x = (\ell, v)$, $x_0 = (\ell_0, v_0)$, $I(x)$, $IR(x_0, x)$ and the annotations $P_\ell(v)$, $RP_\ell(v_0, v)$ are related as follows:

- $I(x) \stackrel{def}{=} \exists \ell, v.(\ell \in \textsc{Locations} \wedge v \in \textsc{Memory} \wedge x = (\ell, v) \wedge P_\ell(v))$
- $P_\ell(v) \stackrel{def}{=} \exists x.(x \in \textsc{Vals} \wedge x = (\ell, v) \wedge I(x))$
- $IR(x_0, x) \stackrel{def}{=} \exists \ell, v, v_0(\ell \in \textsc{Locations} \wedge v \in \textsc{Memory} \wedge x = (\ell, v) \wedge x_0 = (\ell_0, v_0) \wedge RP_\ell(v_0, v))$
- $RP_\ell(v_0, v) \stackrel{def}{=} \exists x, x_0.(x, x_0 \in \textsc{Vals} \wedge x = (\ell, v) \wedge x_0 = (\ell_0, v_0) \wedge IR(x_0, x))$

The transformation is based on the transition relation defined for each pair of consecutive control labels, and is expressed very simply by the following relational form:

$$x \; r_{\ell,\ell'} \; x' \stackrel{def}{=} (pc = \ell \wedge cond_{\ell,\ell'}(v) \wedge v' = f_{\ell,\ell'}(v) \wedge pc' = \ell'$$

Transition from one label to another label

- The transition from ℓ to ℓ' is possible when the condition $cond_{\ell,\ell'}(v)$ is true for V and when control is at ℓ $(pc = \ell)$.
- When the transition is observed, the variables V are transformed as follows: $v' = f_{\ell,\ell'}(v)$.
- The definition of the transition expresses no assumption related to an execution strategy, such as fairness for instance.
- $cond_{\ell,\ell'}(v) \wedge v' = f_{\ell,\ell'}(v)$ is an expression where the expressions $cond_{\ell,\ell'}(v)$ and $= f_{\ell,\ell'}(v)$ raise definedness questions:
  - $DOM(\ell,\ell')(v) \stackrel{def}{=} DEF(cond_{\ell,\ell'}(v))(v) \wedge DEF(f_{\ell,\ell'}(v))$
  - $DEF(E(X))(x)$ means that the expression $E(X)$ is defined for x, the current value of X.
- Some transitions may lead to disasters:
  - $DEF(X{+}1)(x) \stackrel{def}{=} x{+}1 \in D$ where D is the encoding domain of X, for instance $D = -2^{31}..2^{31}-1$ for an encoding on 32 bits.
  - $DEF(T(I{+}1) < V)(t,x,v) \stackrel{def}{=} i{+}1 \in dom(t) \wedge v \in D \wedge t(i{+}1) \in D$

Relations for program statements

$\ell : P_\ell(v_0, v)$
$V := f_{\ell,\ell'}(V)$
$\ell' : P_{\ell'}(v_0, v)$

$\ell_1 : P_{\ell_1}(v_0, v)$
WHILE $B(V)$ **DO**
  $\ell_2 : P_{\ell_2}(v_0, v)$
  $\ldots$
  $\ell_3 : P_{\ell_3}(v_0, v)$
END
$\ell_4 : P_{\ell_4}(v_0, v)$

Translation

- $(pc = \ell \wedge v' = f_{\ell,\ell'}(v) \wedge pc' = \ell'$
- $cond_{\ell,\ell'}(v) \stackrel{def}{=} TRUE$

Translation

$$\begin{cases} pc = \ell_1 \wedge b(v) \wedge v' = v \wedge pc' = \ell_2 \\ pc = \ell_1 \wedge \neg b(v) \wedge v' = v \wedge pc' = \ell_4 \\ pc = \ell_3 \wedge b(v) \wedge v' = v \wedge pc' = \ell_2 \\ pc = \ell_3 \wedge \neg b(v) \wedge v' = v \wedge pc' = \ell_4 \end{cases}$$

Verification of the contract: a simple example

variables U, V
requires $u_0, v_0 \in \mathbb{N}$
ensures $u_f + v_f = u_0 + v_0$

begin
$0 : u = u_0 \wedge v = v_0 \wedge u_0, v_0 \in \mathbb{N}$
$U := U + 2$
$1 : u = u_0 + 2 \wedge v = v_0 \wedge u_0, v_0 \in \mathbb{N}$
$V := V - 2$
$2 : u = u_0 + 2 \wedge v = v_0 - 2 \wedge u_0, v_0 \in \mathbb{N}$
end

Verification to be done: back to the example

- $x = (pc, u, v)$
- $J(0, u_0, v_0, pc, u, v) \stackrel{def}{=}$
$$\left[\begin{array}{l} \wedge\ pc \in \{0,1,2\} \\ \ \wedge\ u, v \in \mathbb{Z} \\ \wedge\ pc = 0 \Rightarrow u = u_0 \wedge v = v_0 \wedge u_0, v_0 \in \mathbb{N} \\ \wedge\ pc = 1 \Rightarrow u = u_0{+}2 \wedge v = v_0 \wedge u_0, v_0 \in \mathbb{N} \\ \wedge\ pc = 2 \Rightarrow u = u_0{+}2 \wedge v = v_0{-}2 \wedge u_0, v_0 \in \mathbb{N} \end{array}\right.$$
- $A(0, u_0, v_0, pc, u, v) \stackrel{def}{=} (pc = 2 \Rightarrow u{+}v = u_0{+}v_0{-}2 \wedge u_0, v_0 \in \mathbb{N})$

$\forall pc, u, v, pc', u', v' \in \{0,1,2\}\times\mathbb{Z}:$
$$\left\{\begin{array}{ll} (1) & \textsc{Init}(0, u_0, v_0) \Rightarrow \mathrm{J}(0, u_0, v_0, 0, u_0, v_0) \\ (2) & \mathrm{J}(0, u_0, v_0, pc, u, v) \Rightarrow \mathrm{A}(0, u_0, v_0, pc, u, v) \\ (3) & \forall i \in \{0, \ldots, n\} : \mathrm{J}(0, u_0, v_0, pc, u, v) \wedge x\ r_i\ pc', u', v' \Rightarrow \mathrm{J}(0, u_0, v_0, pc', \end{array}\right.$$

Verification on the example

1. $\textsc{Init}(0, u_0, v_0,) \Rightarrow \mathrm{J}(0, u_0, v_0, 0, u_0, v_0)$:
   $pc = 0 \wedge u = u_0 \wedge v = v_0 \wedge u_0, v_0 \in \mathbb{N} \Rightarrow \mathrm{J}(0, u_0, v_0, 0, u_0, v_0)$:
2. $\mathrm{J}(0, u_0, v_0, pc, u, v) \Rightarrow \mathrm{A}(0, u_0, v_0, pc, u, v)$
   $\mathrm{J}(pc, u, v) \Rightarrow (pc = 2 \Rightarrow u{+}v = u_0{+}v_0{-}2 \wedge u_0, v_0 \in \mathbb{N})$
3. $\forall i \in \{0, \ldots, n\} : \mathrm{J}(0, u_0, v_0, pc, u, v) \wedge x \; r_i \; pc', u', v' \Rightarrow$
   $\mathrm{J}(0, u_0, v_0, pc', u', v')$

   $$\left[\begin{array}{l} r01(pc, u, v, pc', u', v') \stackrel{def}{=} pc = 0 \wedge u' = u{+}2 \wedge pc' = 1 \wedge v' = v \\ r12(pc, u, v, pc', u', v') \stackrel{def}{=} pc = 1 \wedge v' = v{-}2 \wedge pc' = 2 \wedge u' = u \end{array}\right.$$

   - $J(0, u_0, v_0, pc, u, v) \wedge r01(pc, u, v, pc', u', v') \Rightarrow J(pc', u', v')$
   - $J(0, u_0, v_0, pc, u, v) \wedge r12(pc, u, v, pc', u', v') \Rightarrow J(0, u_0, v_0, pc', u', v')$

Some rules of logical computation

lification 1 $[A \wedge (A \Rightarrow B)] \longrightarrow [A \wedge B]$

lification 2 $[A \wedge (B = C) \wedge D \Rightarrow E \wedge (B = F) \wedge G] \longrightarrow [A \wedge (B = C) \wedge D \Rightarrow E \wedge (C = F) \wedge G]$

lification 3 $[A \wedge (B = C) \wedge D \Rightarrow E \wedge (F = F) \wedge G] \longrightarrow [A \wedge (B = C) \wedge D \Rightarrow E \wedge TRUE \wedge G]$

lification 4 $[A \Rightarrow B \wedge TRUE \wedge C] \longrightarrow [A \Rightarrow B \wedge C]$

▶

lification 5 $[A \wedge (B = C \Rightarrow U) \wedge (B = D \wedge B = C \Rightarrow V) \wedge C \neq D \wedge E] \longrightarrow [A \wedge B = C \wedge U \wedge C \neq D \wedge E]$

$J(pc, u, v) \wedge r01(pc, u, v, pc', u', v') \Rightarrow J(pc', u', v')$ **(1)**

$$
\left(\begin{array}{l}
\wedge\ pc \in \{0,1,2\} \\
\ \wedge\ u, v \in \mathbb{Z} \\
\wedge\ pc = 0 \Rightarrow u = u_0 \wedge v = v_0 \wedge u_0, v_0 \in \mathbb{N} \\
\wedge\ pc = 1 \Rightarrow u = u_0{+}2 \wedge v = v_0 \wedge u_0, v_0 \in \mathbb{N} \\
\wedge\ pc = 2 \Rightarrow u = u_0{+}2 \wedge v = v_0{-}2 \wedge u_0, v_0 \in \mathbb{N}
\end{array}\right) \wedge \left(\begin{array}{l}
\wedge pc = 0 \\
\wedge u' = u{+}2 \\
\wedge pc' = 1 \\
\wedge v' = v
\end{array}\right)
$$

$$
\Rightarrow
$$

$$
\left(\begin{array}{l}
\wedge\ pc' \in \{0,1,2\} \\
\ \wedge\ u', v' \in \mathbb{Z} \\
\wedge\ pc' = 0 \Rightarrow u' = u_0 \wedge v' = v_0 \wedge u_0, v_0 \in \mathbb{N} \\
\wedge\ pc' = 1 \Rightarrow u' = u_0{+}2 \wedge v' = v_0 \wedge u_0, v_0 \in \mathbb{N} \\
\wedge\ pc' = 2 \Rightarrow u' = u_0{+}2 \wedge v' = v_0{-}2 \wedge u_0, v_0 \in \mathbb{N}
\end{array}\right)
$$

Using the TLA Toolbox to check these elements: cours1.tla

Initialisation

The relational model $M(P)$ for the annotated program P is thus defined as follows:
$M(P) \stackrel{def}{=}$
$(Th(s,c), (pc,v), \text{LOCATIONS}\times\text{MEMORY}, Init(\ell,v), \{r_{\ell,\ell'} | \ell,\ell' \in \text{LOCATIONS} \wedge \ell \longrightarrow \ell'\})$.
The definition of $Init(x)$ depends on the precondition of P:

$$Init(x) \stackrel{def}{=} .x = (\ell_0, v) \wedge \textbf{pre}(\mathrm{P})(v).$$

Initial conditions

The two following properties are equivalent:

- $\forall x_0 \in \text{VALS} : \mathit{Init}(x_0) \Rightarrow \mathrm{J}(x_0, x_0)$
- $\forall v \in \text{MEMORY}.\textbf{pre}(\mathrm{P})(v) \wedge v = v_0 \Rightarrow P_{\ell_0}(v_0, v)$

Induction step

- The relations r_i correspond to the transitions satisfying $\ell \longrightarrow \ell'$, and we associate with each r_i the relation $r_{\ell,\ell'}$
  - $x \; r_{\ell,\ell'} \; x' \stackrel{def}{=} (pc = \ell \wedge cond_{\ell,\ell'}(v) \wedge \wedge v' = f_{\ell,\ell'}(v) \wedge pc' = \ell'$
- $J(x_0, x) \stackrel{def}{=} \exists v_0, \ell, v.(\ell \in \textsc{Locations} \wedge v_0, v \in \textsc{Memory} \wedge x = (\ell, v) \wedge P_\ell(v_0, v))$
- $P_\ell(v_0, v) \stackrel{def}{=} \exists x_0, x.(x_0, x \in \textsc{Vals} \wedge x = (\ell, v) \wedge x_0 = (\ell_0, v_0) \wedge J(x_0, x))$

Induction step

The two following properties are equivalent:

- $\forall i \in \{0, \ldots, n\} : \mathrm{J}(x_0, x) \wedge x \; r_i \; x' \Rightarrow \mathrm{J}(x_0, x')$
- $\forall \ell, \ell' \in \textsc{Locations} : \ell \longrightarrow \ell' \Rightarrow P_\ell(v_0, v) \wedge cond_{\ell,\ell'}(v) \wedge v' = f_{\ell,\ell'}(v) \Rightarrow P_{\ell'}(v_0, v')$

Induction step (explanation)

- $\mathrm{J}(x_0 n x) \wedge x \; r_{\ell,\ell'} \; x' \Rightarrow \mathrm{J}(x_0, x')$

Induction step (explanation)

- $\mathrm{J}(x_0 n x) \wedge x\ r_{\ell,\ell'}\ x' \Rightarrow \mathrm{J}(x_0, x')$
- $x\ r_{\ell,\ell'}\ x' \stackrel{def}{=} (pc = \ell \wedge cond_{\ell,\ell'}(v) \wedge \wedge v' = f_{\ell,\ell'}(v) \wedge pc' = \ell'$

Induction step (explanation)

- $\mathrm{J}(x_0 n x) \wedge x\ r_{\ell,\ell'}\ x' \Rightarrow \mathrm{J}(x_0, x')$
- $x\ r_{\ell,\ell'}\ x' \stackrel{def}{=} (pc = \ell \wedge cond_{\ell,\ell'}(v) \wedge \wedge v' = f_{\ell,\ell'}(v) \wedge pc' = \ell'$
- $I(x) \stackrel{def}{=} \exists \ell, v.(\ell \in \textsc{Locations} \wedge v \in \textsc{Memory} \wedge x = (\ell, v) \wedge P_\ell(v))$

Induction step (explanation)

- $\mathrm{J}(x_0 n x) \wedge x\ r_{\ell,\ell'}\ x' \Rightarrow \mathrm{J}(x_0, x')$
- $x\ r_{\ell,\ell'}\ x' \stackrel{def}{=} (pc = \ell \wedge cond_{\ell,\ell'}(v) \wedge \wedge v' = f_{\ell,\ell'}(v) \wedge pc' = \ell'$
- $I(x) \stackrel{def}{=} \exists \ell, v.(\ell \in \textsc{Locations} \wedge v \in \textsc{Memory} \wedge x = (\ell, v) \wedge P_\ell(v))$
- $P_\ell(v_0, v) \stackrel{def}{=} \exists x.(x \in \textsc{Vals} \wedge x = (\ell, v) \wedge x_0 = (\ell_0, v_0) \wedge J(x_0, x))$

Induction step (explanation)

- $\mathrm{J}(x_0 n x) \wedge x\ r_{\ell,\ell'}\ x' \Rightarrow \mathrm{J}(x_0, x')$
- $x\ r_{\ell,\ell'}\ x' \stackrel{def}{=} (pc = \ell \wedge cond_{\ell,\ell'}(v) \wedge \wedge v' = f_{\ell,\ell'}(v) \wedge pc' = \ell'$
- $I(x) \stackrel{def}{=} \exists \ell, v.(\ell \in \textsc{Locations} \wedge v \in \textsc{Memory} \wedge x = (\ell, v) \wedge P_\ell(v))$
- $P_\ell(v_0, v) \stackrel{def}{=} \exists x.(x \in \textsc{Vals} \wedge x = (\ell, v) \wedge x_0 = (\ell_0, v_0) \wedge J(x_0, x))$
- $\mathrm{J}(x_0 n x) \equiv pc = \ell \wedge P_\ell(v_0, v)$

Induction step (explanation)

- $\mathrm{J}(x_0 n x) \wedge x\ r_{\ell,\ell'}\ x' \Rightarrow \mathrm{J}(x_0, x')$
- $x\ r_{\ell,\ell'}\ x' \stackrel{def}{=} (pc = \ell \wedge cond_{\ell,\ell'}(v) \wedge \wedge v' = f_{\ell,\ell'}(v) \wedge pc' = \ell'$
- $I(x) \stackrel{def}{=} \exists \ell, v.(\ell \in \textsc{Locations} \wedge v \in \textsc{Memory} \wedge x = (\ell, v) \wedge P_\ell(v))$
- $P_\ell(v_0, v) \stackrel{def}{=} \exists x.(x \in \textsc{Vals} \wedge x = (\ell, v) \wedge x_0 = (\ell_0, v_0) \wedge J(x_0, x))$
- $\mathrm{J}(x_0 n x) \equiv pc = \ell \wedge P_\ell(v_0, v)$
- $\mathrm{J}(x_0, x') \equiv pc = \ell' \wedge P_\ell(v_0, v')$

Induction step (explanation)

- $\mathrm{J}(x_0 n x) \wedge x\ r_{\ell,\ell'}\ x' \Rightarrow \mathrm{J}(x_0, x')$
- $x\ r_{\ell,\ell'}\ x' \stackrel{def}{=} (pc = \ell \wedge cond_{\ell,\ell'}(v) \wedge \wedge v' = f_{\ell,\ell'}(v) \wedge pc' = \ell'$
- $I(x) \stackrel{def}{=} \exists \ell, v.(\ell \in \textsc{Locations} \wedge v \in \textsc{Memory} \wedge x = (\ell, v) \wedge P_\ell(v))$
- $P_\ell(v_0, v) \stackrel{def}{=} \exists x.(x \in \textsc{Vals} \wedge x = (\ell, v) \wedge x_0 = (\ell_0, v_0) \wedge J(x_0, x))$
- $\mathrm{J}(x_0 n x) \equiv pc = \ell \wedge P_\ell(v_0, v)$
- $\mathrm{J}(x_0, x') \equiv pc = \ell' \wedge P_\ell(v_0, v')$
- $pc = \ell \wedge P_\ell(v_0, v) \wedge (pc = \ell \wedge cond_{\ell,\ell'}(v) \wedge \wedge v' = f_{\ell,\ell'}(v) \wedge pc' = \ell' \ \Rightarrow\ pc = \ell' \wedge P_\ell(v_0, v')$

Induction step (explanation)

- $\mathrm{J}(x_0 n x) \wedge x\ r_{\ell,\ell'}\ x' \Rightarrow \mathrm{J}(x_0, x')$
- $x\ r_{\ell,\ell'}\ x' \stackrel{def}{=} (pc = \ell \wedge cond_{\ell,\ell'}(v) \wedge \wedge v' = f_{\ell,\ell'}(v) \wedge pc' = \ell'$
- $I(x) \stackrel{def}{=} \exists \ell, v.(\ell \in \textsc{Locations} \wedge v \in \textsc{Memory} \wedge x = (\ell, v) \wedge P_\ell(v))$
- $P_\ell(v_0, v) \stackrel{def}{=} \exists x.(x \in \textsc{Vals} \wedge x = (\ell, v) \wedge x_0 = (\ell_0, v_0) \wedge J(x_0, x))$
- $\mathrm{J}(x_0 n x) \equiv pc = \ell \wedge P_\ell(v_0, v)$
- $\mathrm{J}(x_0, x') \equiv pc = \ell' \wedge P_\ell(v_0, v')$
- $pc = \ell \wedge P_\ell(v_0, v) \wedge (pc = \ell \wedge cond_{\ell,\ell'}(v) \wedge \wedge v' = f_{\ell,\ell'}(v) \wedge pc' = \ell' \Rightarrow pc = \ell' \wedge P_\ell(v_0, v')$
- $pc = \ell \wedge P_\ell(v_0, v) \wedge (pc = \ell \wedge cond_{\ell,\ell'}(v) \wedge \wedge v' = f_{\ell,\ell'}(v) \wedge pc' = \ell'$ (Tautology)

Induction step (explanation)

- $\mathrm{J}(x_0 n x) \wedge x\ r_{\ell,\ell'}\ x' \Rightarrow \mathrm{J}(x_0, x')$
- $x\ r_{\ell,\ell'}\ x' \stackrel{def}{=} (pc = \ell \wedge cond_{\ell,\ell'}(v) \wedge \wedge v' = f_{\ell,\ell'}(v) \wedge pc' = \ell'$
- $I(x) \stackrel{def}{=} \exists \ell, v.(\ell \in \textsc{Locations} \wedge v \in \textsc{Memory} \wedge x = (\ell, v) \wedge P_\ell(v))$
- $P_\ell(v_0, v) \stackrel{def}{=} \exists x.(x \in \textsc{Vals} \wedge x = (\ell, v) \wedge x_0 = (\ell_0, v_0) \wedge J(x_0, x))$
- $\mathrm{J}(x_0 n x) \equiv pc = \ell \wedge P_\ell(v_0, v)$
- $\mathrm{J}(x_0, x') \equiv pc = \ell' \wedge P_\ell(v_0, v')$
- $pc = \ell \wedge P_\ell(v_0, v) \wedge (pc = \ell \wedge cond_{\ell,\ell'}(v) \wedge \wedge v' = f_{\ell,\ell'}(v) \wedge pc' = \ell' \ \Rightarrow \ pc = \ell' \wedge P_\ell(v_0, v')$
- $pc = \ell \wedge P_\ell(v_0, v) \wedge (pc = \ell \wedge cond_{\ell,\ell'}(v) \wedge \wedge v' = f_{\ell,\ell'}(v) \wedge pc' = \ell'$ (Tautology)
- $pc = \ell \wedge P_\ell(v) \wedge (pc = \ell \wedge cond_{\ell,\ell'}(v) \wedge \wedge v' = f_{\ell,\ell'}(v) \wedge pc' = \ell' \ \Rightarrow P_{\ell'}(v_0, v')$

Induction step (explanation)

- $\mathrm{J}(x_0 n x) \wedge x\ r_{\ell,\ell'}\ x' \Rightarrow \mathrm{J}(x_0, x')$
- $x\ r_{\ell,\ell'}\ x' \stackrel{def}{=} (pc = \ell \wedge cond_{\ell,\ell'}(v) \wedge \wedge v' = f_{\ell,\ell'}(v) \wedge pc' = \ell'$
- $I(x) \stackrel{def}{=} \exists \ell, v.(\ell \in \textsc{Locations} \wedge v \in \textsc{Memory} \wedge x = (\ell, v) \wedge P_\ell(v))$
- $P_\ell(v_0, v) \stackrel{def}{=} \exists x.(x \in \textsc{Vals} \wedge x = (\ell, v) \wedge x_0 = (\ell_0, v_0) \wedge J(x_0, x))$
- $\mathrm{J}(x_0 n x) \equiv pc = \ell \wedge P_\ell(v_0, v)$
- $\mathrm{J}(x_0, x') \equiv pc = \ell' \wedge P_\ell(v_0, v')$
- $pc = \ell \wedge P_\ell(v_0, v) \wedge (pc = \ell \wedge cond_{\ell,\ell'}(v) \wedge \wedge v' = f_{\ell,\ell'}(v) \wedge pc' = \ell' \ \Rightarrow \ pc = \ell' \wedge P_\ell(v_0, v')$
- $pc = \ell \wedge P_\ell(v_0, v) \wedge (pc = \ell \wedge cond_{\ell,\ell'}(v) \wedge \wedge v' = f_{\ell,\ell'}(v) \wedge pc' = \ell'$ (Tautology)
- $pc = \ell \wedge P_\ell(v) \wedge (pc = \ell \wedge cond_{\ell,\ell'}(v) \wedge \wedge v' = f_{\ell,\ell'}(v) \wedge pc' = \ell' \ \Rightarrow P_{\ell'}(v_0, v')$
- $P_\ell(v_0, v) \wedge cond_{\ell,\ell'}(v) \wedge v' = f_{\ell,\ell'}(v) \Rightarrow P_{\ell'}(v_0, v')$

Conclusion

- $J(x_0, x) \stackrel{def}{=} \exists \ell, v, v_0.(\ell \in \textsc{Locations} \wedge v, v_0 \in \textsc{Memory} \wedge x = (\ell, v) \wedge x_0 = (\ell_0, v_0) \wedge P_\ell(v_0, v))$
- $P_\ell(v_0, v) \stackrel{def}{=} \exists x.(x, x_0 \in \textsc{Vals} \wedge x = (\ell, v) \wedge x_0 = (\ell_0, v_0) \wedge J(x_0), x)$
- $\mathrm{J}(x_0, x) \Rightarrow \mathrm{A}(x_0, x)$
- $\exists \ell, v, v_0.(\ell \in \textsc{Locations} \wedge v, v_0 \in \textsc{Memory} \wedge x = (\ell, v) \wedge x_0 = (\ell_0, v_0) \wedge P_\ell(v_0, v)) \Rightarrow \mathrm{A}(x_0, x)$
- $\forall \ell, v, v_0.(\ell \in \textsc{Locations} \wedge v, v_0 \in \textsc{Memory} \wedge x = (\ell, v) \wedge x_0 = (\ell_0, v_0) \wedge P_\ell(v_0, v)) \Rightarrow \mathrm{A}(x_0, x)$
- $\forall \ell \in \textsc{Locations}, v, v_0 \in \textsc{Memory}.P_\ell(v_0, v) \Rightarrow \mathrm{A}(\ell_0, v_0, \ell, v)$

Conclusion

The two following properties are equivalent:

- $\mathrm{J}(x_0, x) \Rightarrow \mathrm{A}(x_0, x)$
- $\forall \ell \in \textsc{Locations}, v, v_0 \in \textsc{Memory}.P_\ell(v_0, v) \Rightarrow \mathrm{A}(\ell_0, v_0, \ell, v)$

Translation of the verification conditions

The following verification conditions are equivalent:

- $\forall x_0, x, x' \in \text{LOCATIONS} \times \text{MEMORY}$:

$$\begin{cases} (1) \quad \text{INIT}(x_0) \Rightarrow \text{J}(x_0, x_0) \\ (2) \quad \text{J}(x_0, x) \Rightarrow \text{A}(x_0, x) \\ (3) \quad \forall i \in \{0, \ldots, n\} : \text{J}(x_0, x) \wedge x \; r_i \; x' \Rightarrow \text{J}(x_0, x') \end{cases}$$

- $\forall v_0, v, v' \in \text{MEMORY}$:

$$\begin{cases} (1) \quad \textbf{pre}(\text{P})(v_0) \wedge v = v_0 \Rightarrow P_{\ell_0}(v_0, v) \\ (2) \quad \forall \ell \in \text{LOCATIONS}. P_\ell(v_0, v) \Rightarrow \text{A}(\ell_0, v_0, \ell, v) \\ (3) \quad \forall \ell, \ell' \in \text{LOCATIONS} : \\ \ell \longrightarrow \ell' \Rightarrow P_\ell(v_0, v) \wedge cond_{\ell,\ell'}(v) \wedge v' = f_{\ell,\ell'}(v) \Rightarrow P_{\ell'}(v_0, v') \end{cases}$$

- The program is annotated.
- The annotations define an invariant to be checked according to the verification conditions.
- $A(\ell, v)$ is the statement of the safety property to be checked.

Relational method for the correctness of safety properties

Let $\mathrm{A}(\ell_0, v_0, \ell, v)$ be a property of a program P. Let $\{P_\ell(v_0, v) : \ell \in \textsc{Locations}\}$ be a family of annotations (properties) for this program. If the following conditions hold:
then $\mathrm{A}(\ell_0, v_0, \ell, v)$ is a safety property for the program P.

Equivalence Floyd/Hoare

⊠ DefinitionVerification condition
The expression $P_\ell(v_0, v) \wedge cond_{\ell,\ell'}(v) \wedge v' = f_{\ell,\ell'}(v) \Rightarrow P_{\ell'}(v_0, v')$, where ℓ, ℓ' are two labels related by the relation $\longrightarrow$, is called a verification condition.

Floyd and Hoare

- $\forall v_0, v, v' \in \text{MEMORY}.\forall \ell, \ell' \in \text{LOCATIONS}.\ell \longrightarrow \ell' \Rightarrow P_\ell(v_0, v) \wedge cond_{\ell,\ell'}(v) \wedge v' = f_{\ell,\ell'}(v) \Rightarrow P_{\ell'}(v_0, v')$ is equivalent to $\forall \ell, \ell' \in \text{LOCATIONS}.\ell \longrightarrow \ell' \Rightarrow \forall v' \in \text{MEMORY}.P_\ell(v_0, v) \wedge cond_{\ell,\ell'}(v) \Rightarrow P_{\ell'}(v_0, v \mapsto f_{\ell,\ell'}(v))$
- $\forall v_0, v, v' \in \text{MEMORY}.\forall \ell, \ell' \in \text{LOCATIONS}.\ell \longrightarrow \ell' \Rightarrow P_\ell(v_0, v) \wedge cond_{\ell,\ell'}(v) \wedge v' = f_{\ell,\ell'}(v) \Rightarrow P_{\ell'}(v_0, v')$ is equivalent to $\forall \ell, \ell' \in \text{LOCATIONS}.\ell \longrightarrow \ell' \Rightarrow \forall v' \in \text{MEMORY}.\ (\exists v \in \text{MEMORY}.P_\ell(v_0, v) \wedge cond_{\ell,\ell'}(v) \wedge v' = f_{\ell,\ell'}(v)) \Rightarrow P_{\ell'}(v_0, v')$

Verification condition for the assignment

We can summarise the two possible forms of the following assignment:

- $\forall v, v' \in \text{MEMORY}. P_\ell(v_0, v) \wedge cond_{\ell,\ell'}(v) \wedge v' = f_{\ell,\ell'}(v) \Rightarrow P_{\ell'}(v_0, v')$

$\ell : P_\ell(v_0, v)$
$V := f_{\ell,\ell'}(V)$
$\ell' : P_{\ell'}(v_0, v)$

Verification condition for the assignment

We can summarise the two possible forms of the following assignment:

- $\forall v, v' \in \textsc{Memory}. P_\ell(v_0, v) \wedge cond_{\ell,\ell'}(v) \wedge v' = f_{\ell,\ell'}(v) \Rightarrow P_{\ell'}(v_0, v')$
- $\forall v, v' \in \textsc{Memory}. P_\ell(v_0, v) \wedge TRUE \wedge v' = f_{\ell,\ell'}(v) \Rightarrow P_{\ell'}(v_0, v')$

$\ell : P_\ell(v_0, v)$
$V := f_{\ell,\ell'}(V)$
$\ell' : P_{\ell'}(v_0, v)$

Verification condition for the assignment

We can summarise the two possible forms of the following assignment:

$\ell : P_\ell(v_0, v)$
$V := f_{\ell,\ell'}(V)$
$\ell' : P_{\ell'}(v_0, v)$

- $\forall v, v' \in \text{MEMORY}. P_\ell(v_0, v) \wedge cond_{\ell,\ell'}(v) \wedge v' = f_{\ell,\ell'}(v) \Rightarrow P_{\ell'}(v_0, v')$
- $\forall v, v' \in \text{MEMORY}. P_\ell(v_0, v) \wedge TRUE \wedge v' = f_{\ell,\ell'}(v) \Rightarrow P_{\ell'}(v_0, v')$
- $\forall v, v' \in \text{MEMORY}. P_\ell(v_0, v) \wedge v' = f_{\ell,\ell'}(v) \Rightarrow P_{\ell'}(v_0, v')$

Verification condition for the assignment

We can summarise the two possible forms of the following assignment:

$$\ell : P_\ell(v_0, v)$$
$$V := f_{\ell,\ell'}(V)$$
$$\ell' : P_{\ell'}(v_0, v)$$

- $\forall v, v' \in \text{MEMORY}. P_\ell(v_0, v) \wedge cond_{\ell,\ell'}(v) \wedge v' = f_{\ell,\ell'}(v) \Rightarrow P_{\ell'}(v_0, v')$
- $\forall v, v' \in \text{MEMORY}. P_\ell(v_0, v) \wedge TRUE \wedge v' = f_{\ell,\ell'}(v) \Rightarrow P_{\ell'}(v_0, v')$
- $\forall v, v' \in \text{MEMORY}. P_\ell(v_0, v) \wedge v' = f_{\ell,\ell'}(v) \Rightarrow P_{\ell'}(v_0, v')$
- $\forall v \in \text{MEMORY}. P_\ell(v_0, v) \Rightarrow P_{\ell'}(v_0, v \mapsto f_{\ell,\ell'}(v))$ (Hoare's axiomatics).

Verification condition for the assignment

We can summarise the two possible forms of the following assignment:

$\ell : P_\ell(v_0, v)$
$V := f_{\ell,\ell'}(V)$
$\ell' : P_{\ell'}(v_0, v)$

- $\forall v, v' \in \text{MEMORY}. P_\ell(v_0, v) \wedge cond_{\ell,\ell'}(v) \wedge v' = f_{\ell,\ell'}(v) \Rightarrow P_{\ell'}(v_0, v')$
- $\forall v, v' \in \text{MEMORY}. P_\ell(v_0, v) \wedge TRUE \wedge v' = f_{\ell,\ell'}(v) \Rightarrow P_{\ell'}(v_0, v')$
- $\forall v, v' \in \text{MEMORY}. P_\ell(v_0, v) \wedge v' = f_{\ell,\ell'}(v) \Rightarrow P_{\ell'}(v_0, v')$
- $\forall v \in \text{MEMORY}. P_\ell(v_0, v) \Rightarrow P_{\ell'}(v_0, v \mapsto f_{\ell,\ell'}(v))$ (Hoare's axiomatics).
- $\forall v \in \text{MEMORY}. (\exists v' \in \text{MEMORY}. P_\ell(v_0, v) \wedge v' = f_{\ell,\ell'}(v)) \Rightarrow P_{\ell'}(v_0, v')$ corresponds to Floyd's assignment rule.

Verification conditions for the iteration

$\ell_1 : P_{\ell_1}(v_0, v)$
WHILE $B(v)$ **DO**
  $\ell_2 : P_{\ell_2}(v_0, v)$
  $\ldots$
  $\ell_3 : P_{\ell_3}(v_0, v)$
END
$\ell_4 : P_{\ell_4}(v_0, v)$

For the iteration structure, the verification conditions are the following:

- $P_{\ell_1}(v_0, v) \wedge B(v) \Rightarrow P_{\ell_2}(v_0, v)$
- $P_{\ell_1}(v_0, v) \wedge \neg B(v) \Rightarrow P_{\ell_4}(v_0, v)$
- $P_{\ell_3}(v_0, v) \wedge B(v) \Rightarrow P_{\ell_2}(v_0, v)$
- $P_{\ell_3}(v_0, v) \wedge \neg B(v) \Rightarrow P_{\ell_4}(v_0, v)$

Verification conditions for the conditional

$\ell_1 : P_{\ell_1}(v_0, v)$
IF $B(v)$ **THEN**
  $\ell_2 : P_{\ell_2}(v_0, v)$
  $\ldots$
  $\ell_3 : P_{\ell_3}(v_0, v)$
ELSE
  $m_2 : P_{\ell_2}(v_0, v)$
  $\ldots$
  $m_3 : P_{\ell_3}(v_0, v)$
FI
$\ell_4 : P_{\ell_4}(v_0, v)$

For the conditional structure, the verification condi

- $P_{\ell_1}(v_0, v) \wedge B(v) \Rightarrow P_{\ell_2}(v_0, v)$
- $P_{\ell_3}(v_0, v) \Rightarrow P_{\ell_4}(v_0, v)$
- $P_{\ell_1}(v_0, v) \wedge \neg B(v) \Rightarrow P_{m_2}(v_0, v)$
- $P_{m_3}(v_0, v) \Rightarrow P_{\ell_4}(v_0, v)$

Correctness properties of programs

Let v be a state variable of P. $\mathbf{pre}(\mathrm{P})(v)$ is the precondition of P for v; it characterises the initial values of v. $\mathbf{post}(\mathrm{P})(v_0, v)$ is the postcondition of P for v; it characterises the final values of v in relation with the initial value v_0

Example

1. $\mathbf{pre}(\mathrm{P})(x, y, z) = x, y, z \in \mathbb{N}$ and
   $\mathbf{post}(\mathrm{P})(x_0, y_0, z_0, x, y, z) = z = x_0 \cdot y_0$
2. $\mathbf{pre}(\mathrm{Q})(x, y, z) = x, y, z \in \mathbb{N}$ and
   $\mathbf{post}(\mathrm{Q})(x_0, y_0, z_0, x, y, z) = z = x_0 + y_0$

$$\forall \underline{x}, \underline{y}, \underline{r}, \underline{q}, \overline{x}, \overline{y}, \overline{r}, \overline{q}.$$
$$\mathbf{pre}(\mathrm{P})(\underline{x}, \underline{y}, \underline{r}, \underline{q}) \wedge (\underline{x}, \underline{y}, \underline{r}, \underline{q}) \xrightarrow{\mathrm{P}} (\overline{x}, \overline{y}, \overline{r}, \overline{q})$$
$$\Rightarrow \mathbf{post}(\mathrm{P})(\underline{x}, \underline{y}, \underline{r}, \underline{q}, \overline{x}, \overline{y}, \overline{r}, \overline{q})$$

Partial correctness of a program

Partial correctness aims at establishing that a program P is partially correct with respect to its precondition and its postcondition.

- the specification of the data of P **pre**$(\mathrm{P})(\mathrm{v}_0)$
- the specification of the results of P **post**$(\mathrm{P})(\mathrm{v}_0,\mathrm{v})$
- a family of annotations (properties) $\{P_\ell(v_0, v) : \ell \in \textsc{Locations}\}$ for this program.
- a safety property defining partial correctness, $pc = \ell_f \Rightarrow$ **post**$(\mathrm{P})(v_0, v_f)$, where ℓ_f is the label marking the end of the program P

⊠ Definition

The program P is partially correct with respect to **pre**$(\mathrm{P})(\mathrm{v}_0)$ and **post**$(\mathrm{P})(\mathrm{v}_0,\mathrm{v})$, if the property $pc = \ell_f \Rightarrow$ **post**$(\mathrm{P})(v_0, v)$ is a safety property for this program.

Partial correctness of a program

If the following conditions hold:

- $\forall v_0, v \in \text{MEMORY} : \textbf{pre}(\text{P})(v_0) \wedge v = v_0 \Rightarrow P_{\ell_0}(v_0, v)$
- $\forall v_0, v \in \text{MEMORY} : P_{\ell_f}(v_0, v) \Rightarrow \textbf{post}(\text{P})(v_0, v)$
- $\forall \ell, \ell' \in \text{LOCATIONS} : \ell \longrightarrow \ell' : \forall v_0, v, v' \in \text{MEMORY}. (P_\ell(v_0, v) \wedge cond_{\ell,\ell'}(v) \wedge v' = f_{\ell,\ell'}(v) \Rightarrow P_{\ell'}(v_0, v'))$,

then the program P is partially correct with respect to **pre**(P)(v_0) and **post**(P)(v_0,v).

- Partial correctness states that if the program terminates normally, then the postcondition is satisfied by the current variables.
- The semantics of the contract is thus rather simple to give:

Reformulation of the computation

- $\mathsf{pre}(v_0) \wedge v_0 \xrightarrow{\mathsf{P}} v_f \Rightarrow \mathsf{post}(v_0, v_f)$

  (expression of partial correctness)

- $pc_0 = \ell_0 \wedge \mathsf{pre}(v_0) \wedge (pc_0, v_0) \overset{\textsc{Next}^\star}{\longrightarrow} (pc, v) \wedge pc = \ell_f \Rightarrow \mathsf{post}(v_0, v_f)$

  (big-step semantics and small-step semantics equivalence)

- $pc_0 = \ell_0 \wedge \mathsf{pre}(v_0) \wedge (pc_0, v_0) \overset{\textsc{Next}^\star}{\longrightarrow} (pc, v) \Rightarrow (pc = \ell_f \Rightarrow \mathsf{post}(v_0, v_f))$

  (implication and conjunction property)

- $Init(x_0) \wedge x_0 \overset{\textsc{Next}^\star}{\longrightarrow} x \Rightarrow \mathsf{PC}(x_0, x)$

  $(Init(x_0) \overset{def}{=} pc_0 = \ell_0 \wedge \mathsf{pre}(v_0)$

  $x_0 \overset{def}{=} (\ell_0, v_0)$ and $x \overset{def}{=} (pc, v)$

  $\mathsf{PC}(x_0 x) \overset{def}{=} x_0 = (\ell_0, v_0) \wedge x = (pc, v) \Rightarrow (pc = \ell_f \Rightarrow \mathsf{post}(v_0, v_f))$



Partial correctness is a safety property and the relational method for safety properties is applied.

Verification of the contract

A program P *satisfies* a contract (pre,post):

- P transforms a variable v from an initial value v_0 and produces a final value v_f: $v_0 \xrightarrow{P} v_f$
- v_0 satisfies pre: $pre(v_0)$ and v_f satisfies post : $post(v_0, v_f)$
- $pre(v_0) \wedge v_0 \xrightarrow{P} v_f \Rightarrow post(v_0, v_f)$

requires $pre(v_0)$
ensures $post(v_0, v_f)$
variables V

begin
$0 : P_0(v_0, v)$
$\texttt{instruction}_0$
$\ldots$
$i : P_i(v_0, v)$
$\ldots$
$\texttt{instruction}_{f-1}$
$f : P_f(v_0, v)$
end

- $pre(v_0) \wedge v = v_0 \Rightarrow P_0(v_0, v)$
- $pre(v_0) \wedge P_f(v_0, v) \Rightarrow post(v_0, v)$
- For every pair of labels ℓ, ℓ' such that $\ell \longrightarrow \ell'$, we check that, for all values $v, v' \in \textsc{Memory}$

$$\left(\begin{array}{l} \left(\begin{array}{l} P_\ell(v_0, v)) \\ \wedge cond_{\ell,\ell'}(v) \wedge v' = f_{\ell,\ell'}(v) \end{array} \right) \\ \Rightarrow P_{\ell'}(v_0, v') \end{array} \right),$$

Floyd/Turing method of intermittent assertions

An Early Program Proof by Alan Turing

Turing, A. M. 1949. "Checking a Large Routine." In Report of a Conference on High Speed Automatic Calculating Machines, Univ. Math. Lab., Cambridge,pp. 67-69.

- Turing raised a fundamental question on the correctness of routines or programs in 1949.
- It is probably (Jones!) the method of annotation and induction on programs which Floyd finalised in 1967.

Floyd's method

- At point 0, $pre(x_0) \wedge x = x_0 \Rightarrow P_0(x_0, x)$
- Annotations: at point i, the assertion $P_i(x_0, x)$ is true.
- At the final point f, $pre(x_0) \wedge P_f(x_0, x) \Rightarrow post(x_0, x)$

Absence of run-time errors

- The transition to execute is the one going from ℓ to ℓ', characterised by the condition or guard $cond_{\ell,\ell'}(v)$ on v and by a transformation of the variable v, $v' = f_{\ell,\ell'}(v)$.
- An error-absence condition is defined by $\mathbf{DOM}(\ell,\ell')(v)$ for the transition at hand. $\mathbf{DOM}(\ell,\ell')(v)$ means that the transition $\ell \longrightarrow \ell'$ is possible and does not lead to an error.
- An error is an arithmetic overflow, a reference to an array element that does not exist, a reference to a null pointer, ...

example

1. The transition corresponds to an assignment of the form $x := x+y$ or $y := x+y$:
   $\mathbf{DOM}(x+y)(x,y) \stackrel{def}{=} \mathbf{DOM}(x)(x,y) \wedge \mathbf{DOM}(y)(x,y) \wedge x+y \in int$
2. The transition corresponds to an assignment of the form $x := x+1$ or $y := x+1$:
   $\mathbf{DOM}(x+1)(x,y) \stackrel{def}{=} \mathbf{DOM}(x)(x,y) \wedge x+2 \in int$

Absence of run-time errors

RTE definition

The absence of run-time errors aims at establishing that a program P will not produce errors during its execution with respect to its precondition and its postcondition.

- the specification of the data of P **pre**$(\mathrm{P})(\mathrm{v})$
- the specification of the results of P **post**$(\mathrm{P})(\mathrm{v}_0,\mathrm{v})$
- a family of annotations (properties) $\{P_\ell(v) : \ell \in \textsc{Locations}\}$ for this program.
- a safety property defining the absence of run-time errors:

$$\bigwedge_{\ell \in \textsc{Locations} - \{output\}, n \in \textsc{Locations}, \ell \longrightarrow n} (\mathbf{DOM}(\ell, n)(v))$$

⊠ Definition

The program P will not produce run-time errors with respect to **pre**$(\mathrm{P})(\mathrm{v})$ and **post**$(\mathrm{P})(\mathrm{v}_0,\mathrm{v})$, if the property

$$\bigwedge_{\ell \in \textsc{Locations} - \{output\}, n \in \textsc{Locations}, \ell \longrightarrow n} (\mathbf{DOM}(\ell, n)(v))$$ is a safety property for this program.

Induction principle for RTE

RTE = Run Time Error

If the following conditions hold:

- $\forall v_0, v \in \text{MEMORY} : \textbf{pre}(\text{P})(v_0) \wedge v = v_0 \Rightarrow P_{\ell_0}(v_0, v)$
- $\forall m \in \text{LOCATIONS} - \{\ell_f\}, n \in \text{LOCATIONS}, \forall v_0, v, v' \in \text{MEMORY} :$ $m \longrightarrow n : \textbf{pre}(\text{P})(v_0) \wedge P_m(v_0, v) \Rightarrow \textbf{DOM}(m, n)(v)$
- $\forall \ell, \ell' \in \text{LOCATIONS} : \ell \longrightarrow \ell' : \forall v_0, v, v' \in$ $\text{MEMORY}. (P_\ell(v_0, v) \wedge cond_{\ell,\ell'}(v) \wedge v' = f_{\ell,\ell'}(v) \Rightarrow P_{\ell'}(v_0, v')),$

then the program P will not produce run-time errors with respect to **pre**(P)(v_0) and **post**(P)(v_0,v).

- We must first check partial correctness, then strengthen the assertions of partial correctness with domain conditions.
- We can thus derive a contract which also includes the verification of the absence of run-time errors.

Relational verification method for RTE

A program P *satisfies* a contract (pre,post):

- P transforms a variable v from an initial value v_0 and produces a final value v_f: $v_0 \xrightarrow{P} v_f$
- v_0 satisfies pre: $pre(v_0)$ and v_f satisfies post : $post(v_0, v_f)$
- $pre(v_0) \wedge v_0 \xrightarrow{P} v_f \Rightarrow post(v_0, v_f)$
- $\mathbb{D}$ is the RTE domain of V

requires $pre(v_0)$
ensures $post(v_0, v_f)$
variables V

begin
$0 : P_0(v_0, v)$
$\texttt{instruction}_0$
...
$i : P_i(v_0, v)$
...
$\texttt{instruction}_{f-1}$
$f : P_f(v_0, v)$
end

- $pre(v_0) \wedge v = v_0 \Rightarrow P_0(v_0, v)$
- $pre(x_0) \wedge P_f(v_0, v) \Rightarrow post(v_0, v)$
- For every pair of labels ℓ, ℓ' such that $\ell \longrightarrow \ell'$, we check that, for all values $v, v' \in \textsc{Memory}$

$$\left(\begin{array}{l} \left(\begin{array}{l} P_\ell(v_0, v)) \\ \wedge cond_{\ell,\ell'}(v) \wedge v' = f_{\ell,\ell'}(v) \end{array} \right) \\ \Rightarrow P_{\ell'}(v_0, v') \end{array} \right),$$

- $\forall m \in \textsc{Locations} - \{\ell_f\}, n \in \textsc{Locations}, \forall v_0, v, v' \in \textsc{Memory}$: $m \longrightarrow n$: $pre(v_0) \wedge P_m(v_0, v) \Rightarrow \mathbf{DOM}(m, n)(v)$