Mechanical Program Verification – Part 1

Jürgen F H Winkler
Institute of Informatics
Friedrich Schiller University
Jena, Germany

ELTE, Budapest, 6 - 10 Oct 2008

Material: http://psc.informatik.uni-jena.de/teach/verific/course-MPV.htm

Overview

- Historical Overview, Basic Concepts, Realistic Progr Verific
- Mechanical Program Verification (MPV)
- Comparison of 3 Automatic Program Provers (APP)
- The Frege Program Prover (FPP) in More Detail
- Mechanical Generation of Invariants for FOR-Loops
- Problems of FPP (and others)
- Towards Realistic Verification Conditions (VC)
- Summary

Part 1

- Program Verification
- Historical Overview and Basic Concepts
- References

Program Verification

Program *p conforms to* ***specification*** *q* $\equiv$ $p \leq q$

aka: p is correct (wrt q)

q is given: specifies the required behavior aka "what the program shall do"

p has been developed

Verification *tries to show* that all possible

executions of p

conform to Q

=> *The ultimate goal of program development is program execution*

Program correctness has been a concern from the very beginning

Goldstine and v. Neumann: 1 April, 1947 ("Planning and Coding Problems …")

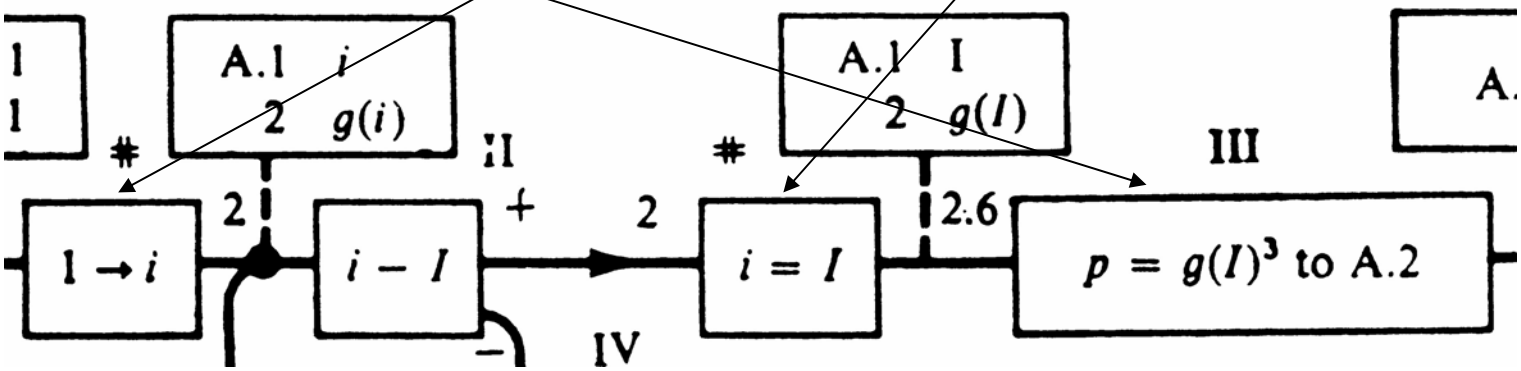
during the design phase of the EDVAC (built 1949 -1952, EDSAC 1949May06)

symbolic notation for machine instructions: S(x) → AC

„clear accumulator and add number located at position x in the Selectrons into it"

flowchart notation for programs with some higher notations:

operation boxes, substitution boxes, assertion boxes



attached to the operation boxes, the alternative boxes and the variable remote connections, and it will prove to be in the main only a process of static translation (cf. the end of 7.1 as well as 7.9). In order to prepare the ground for this final process of detailed (static) coding, we enumerate the operation boxes and the

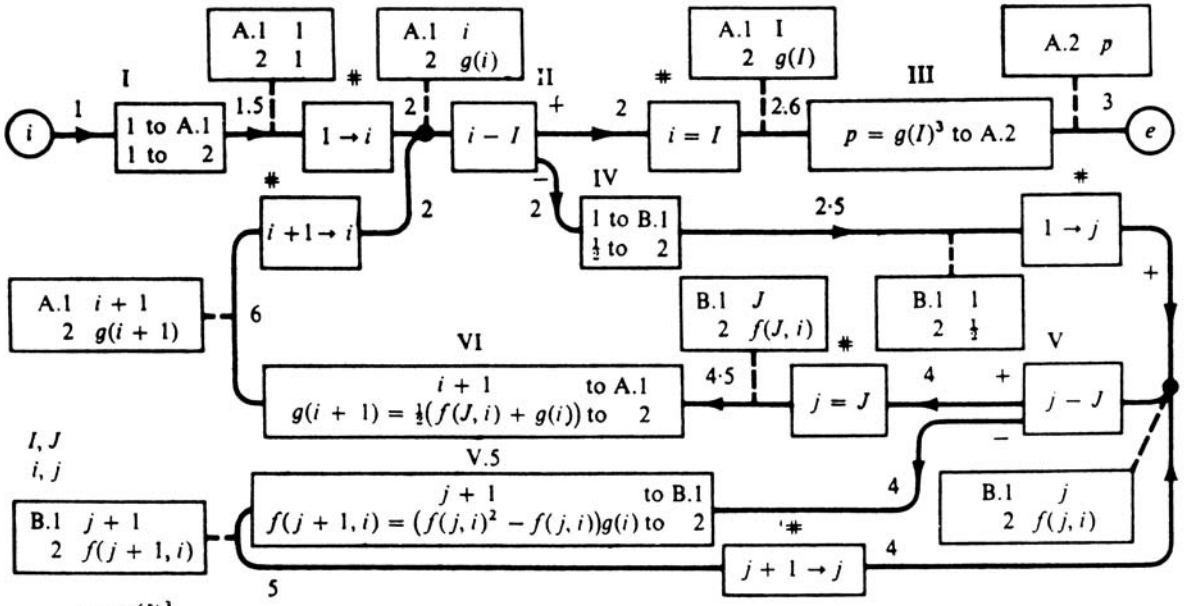
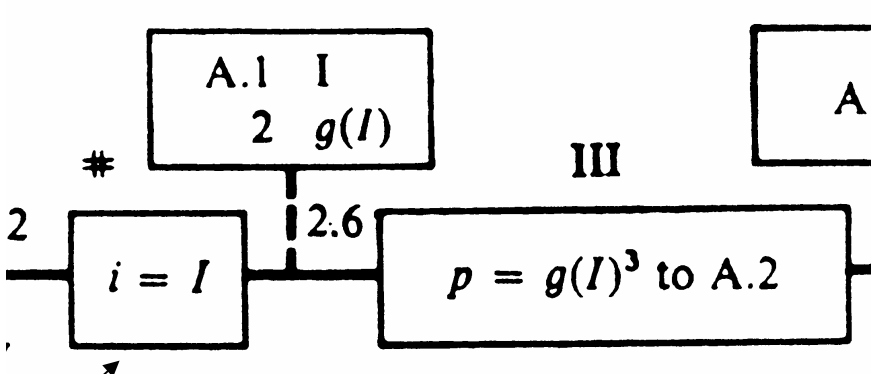


FIG. 7.10

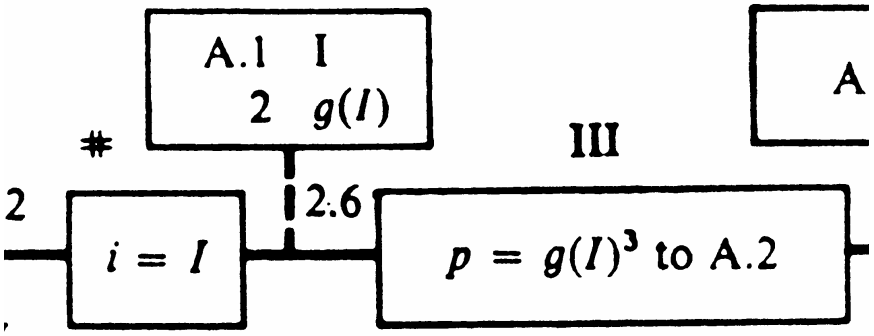


i: program variable ("bound variable") stored in variable storage

I: program constant ("free variable") stored in fixed storage

i=I: an assertion that must hold whenever the computation reaches that point

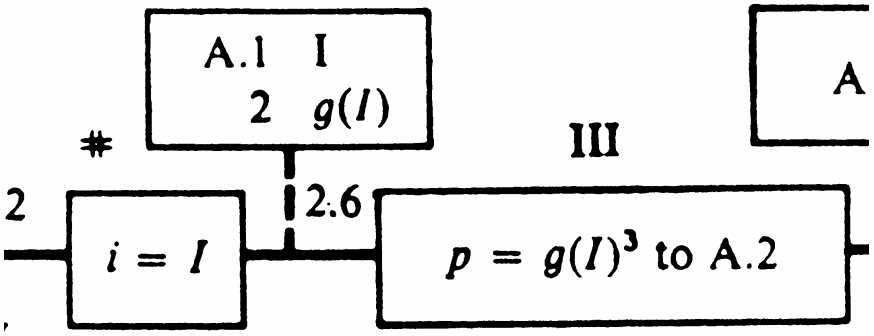
"An assertion box never requires that any specific calculations be made, it indicates only that certain relations are automatically fulfilled whenever C gets to the region which it occupies." [GN 47: 97]



looks very similar to: { i = I } A(2) := g(I)^3; Hoare

--!pre: i = I; FPP (Frege Program Prover)

A(2) := (g(I))**3;



Goldstine / v. Neumann:

no discussion about the correctness of the implementation of the operation boxes

Program correctness has been a concern from the very beginning

Turing 1949 (“Checking a Large Routine”)

“How can one check a routine in the sense of making sure that it is right?

In order that the man who checks may not have too difficult a task the

programmer should make a number of definite assertions

which can be

checked individually,

and from which the correctness of the whole programme easily follows.”

[Tur 49: 70]

Program correctness has been a concern from the very beginning

Turing 1949 ("Checking a Large Routine")

"Finally the checker has to

verify that the process comes to an end.

Here again he should be assisted by the programmer giving a further definite assertion to be verified.

This may take the form of a

quantity which is asserted to decrease continually
and vanish when the machine stops."

[Tur 49: 71]

modern: termination function

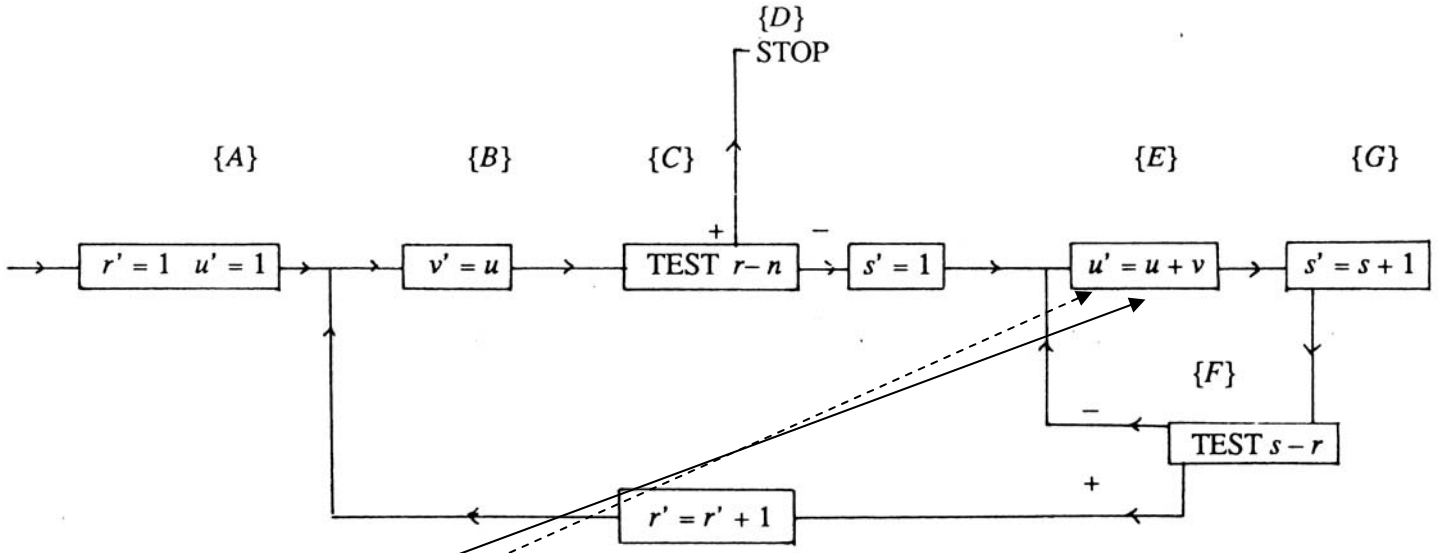


Figure 1

u: initial value of variable u

u': final value of variable u ("at the end of the process represented by the box")

s = content of line 27 of store

…

v = content of line 31 of store

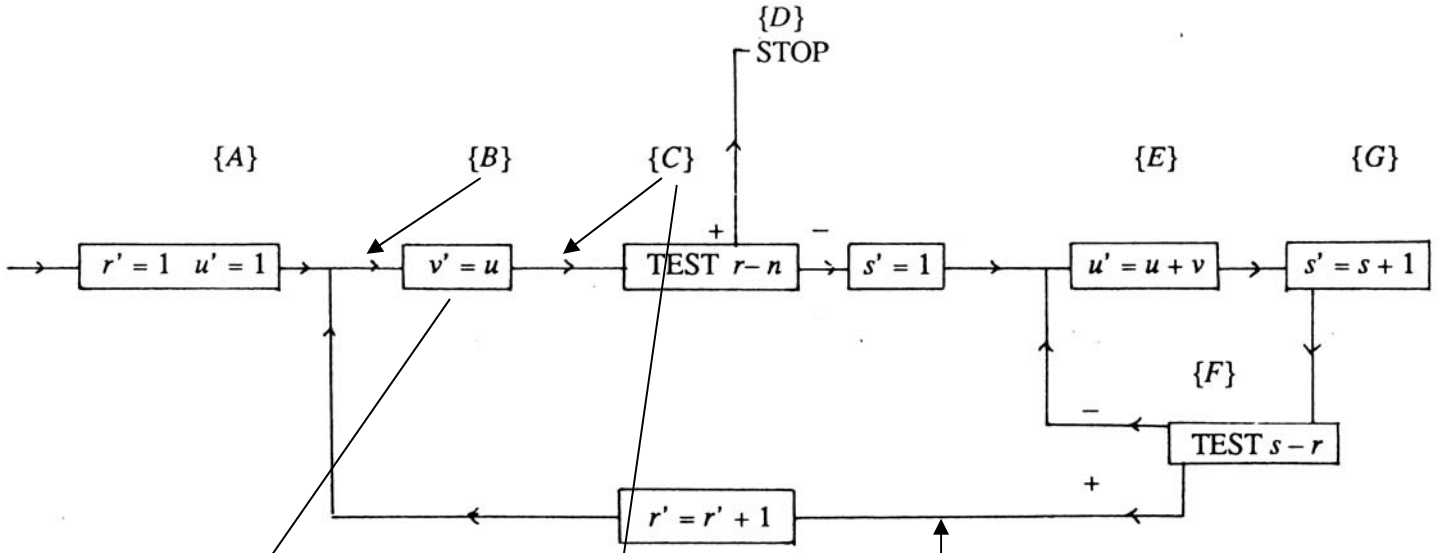


Figure 1

s = r

v := u; { C $_d\equiv$ v = r! ∧ …}

Storage Location	(INITIAL) {A} k = 6	{B} k = 5	{C} k = 4	(STOP) {D} k = 0	{E} k = 3	{F} k = 1	{G} k = 2
27					s	$s+1$	s
28		r	r		r	r	r
29	n	n	n	n	$[n]$	n	n
30		$r!$	$r!$		$s\,r!$	$(s+1)\,r!$	$(s+1)\,r!$
31			$r!$	$n!$	$r!$	$r!$	$r!$

s
r
n
u
v

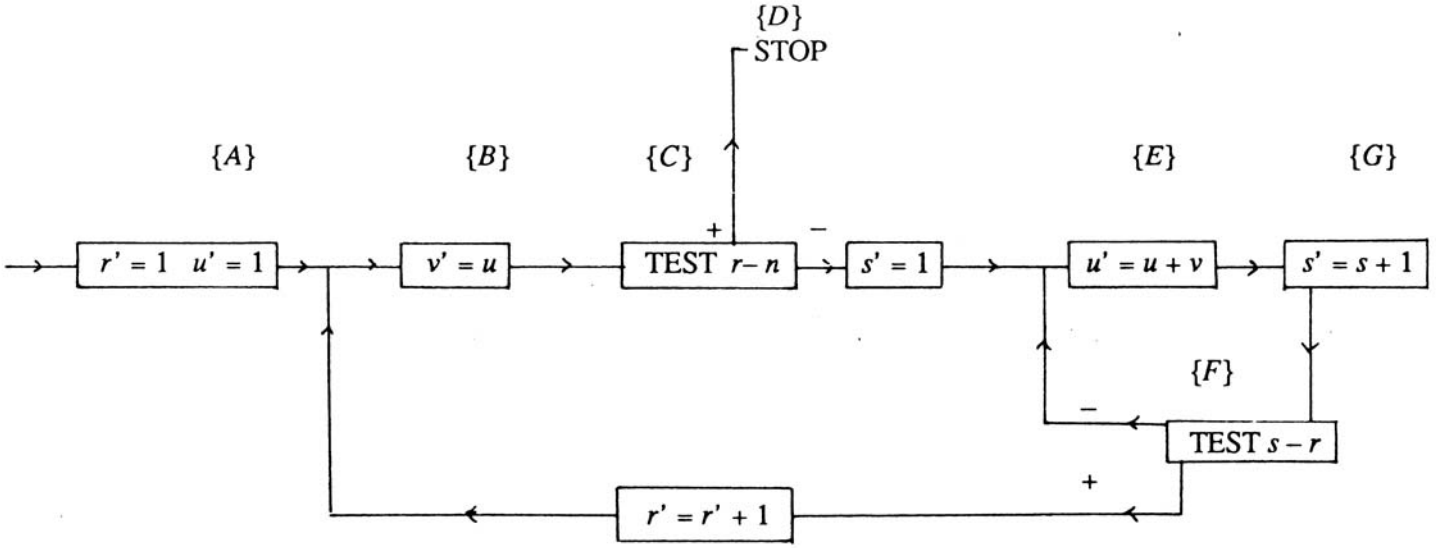


Figure 1

Turing: the operations in the boxes are very near to single machine instructions

Program Verification: History

1947	Goldstine / v. Neumann	: flow diagrams + assertions
1949	Turing	: flow diagrams + assertions
1967	Floyd	: flow diagrams + assertions
1969	Hoare	: derivation system for valid triples
		formalization
1976	Dijkstra	: function (wp) and schemas for valid triples
		mechanization
2003	Hoare	: Verifying compiler as a grand challenge

The Conformity Relation $p \leq q$

p and q are relations between program states $p, q \subseteq \Sigma \times \Sigma$

$\Sigma = Var \rightarrow Val$ the set of all mappings
from the set of all variables Var into the set of values Val

Examples: x, y: int32;

$Sem(x := 10;) = \{ (\{(x, x^i), (y, y^i)\} , \{(x, x^f), (y, y^i)\}): x^i, y^i \in int32 \wedge x^f = 10\}$
$= \{ (\{(x, x^i), (y, y^i)\} , \{(x, 10), (y, y^i)\}): x^i, y^i \in int32 \}$

--! pre: $x \in int32 \wedge y \in int32 \wedge y = Cy$

x := 10;

--! post: $x = 10 \wedge y \in int32 \wedge y = Cy$

Determinism-1

x := 10; is ***deterministic***,

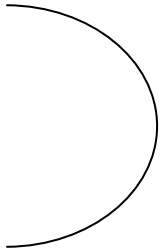
i.e. Sem(x := 10;) is right-unique

{(x, 0), (y, 0)} $\mapsto$ {(x, 10), (y, 0)}

{(x, 0), (y, 1)} $\mapsto$ {(x, 10), (y, 1)}

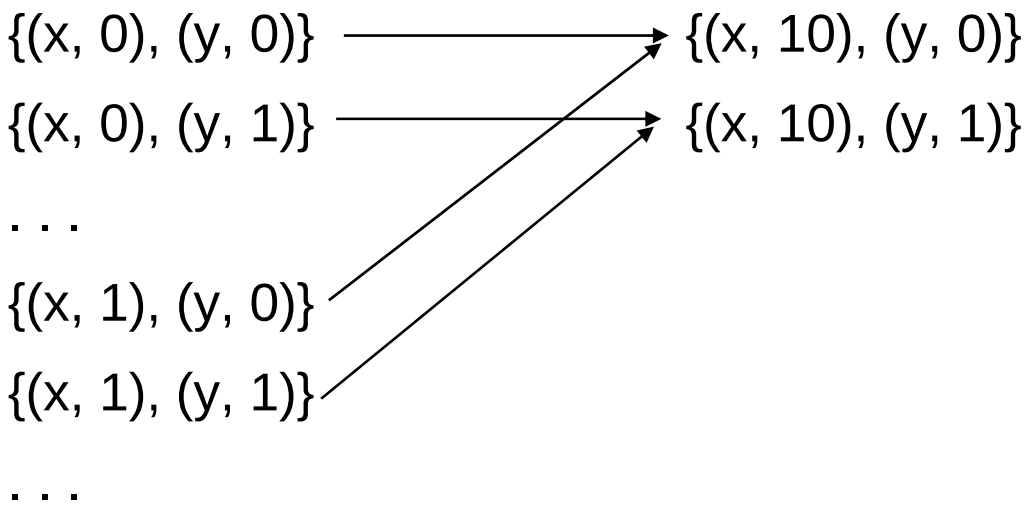
…

{(x, 1), (y, 0)} $\mapsto$ {(x, 10), (y, 0)}



In this special case Sem(x := 10;) is a mapping $\Sigma \to \Sigma$

Determinism-2



Determinism-3

read(x); is ***nondeterministic***:

{(x, 0), (y, 0)} ↦ {(x, -2147483648), (y, 0)}

{(x, 0), (y, 0)} ↦ {(x, -2147483647), (y, 0)}

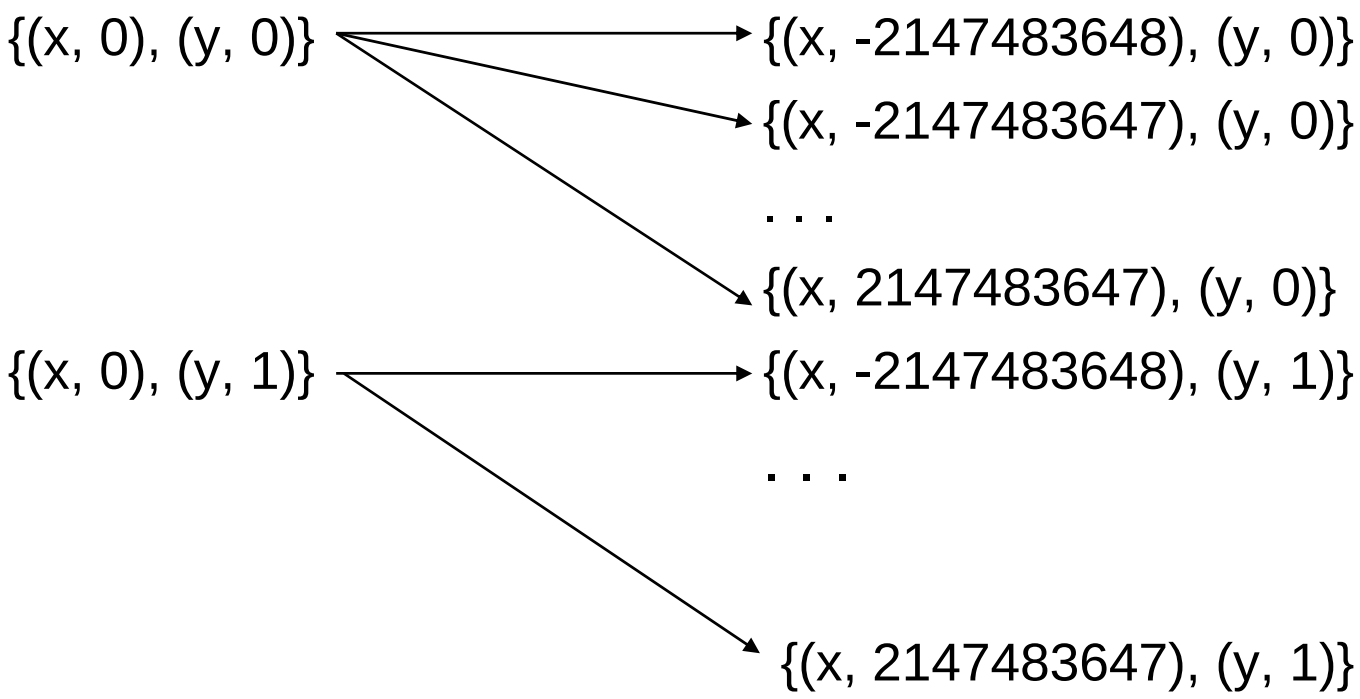
. . .

{(x, 0), (y, 0)} ↦ {(x, 2147483647), (y, 0)}

{(x, 0), (y, 1)} ↦ {(x, -2147483648), (y, 1)}

. . .

Determinism-4



The Conformity Relation $p \leq q$

p and q are relations between program states $p, q \subseteq \Sigma \times \Sigma$

$dom(p) \subset \Sigma$ or $dom(q) \subset \Sigma$ may hold,

i.e. specs or programs need not be lefttotal

$q = \{ (\{(x, x^i), (y, y^i)\} , \{(x, x^f), (y, y^i)\}):$

$x^i, y^i \in int32 \wedge \mathbf{x^i > 0} \wedge (x^f = \sqrt{x^i} \vee x^f = -\sqrt{x^i}) \}$

The spec q is not lefttotal and is nondeterministic:

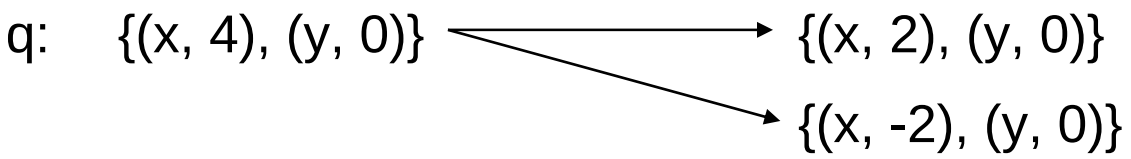
the final value of x may be $\sqrt{x^i}$ or may be $-\sqrt{x^i}$ (completely arbitrary choice)

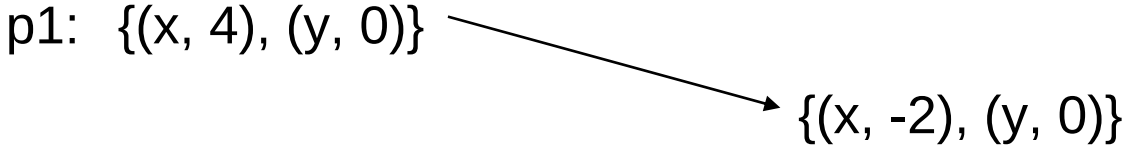
If a program p1 computes always $-\sqrt{x^i}$ then this is OK:

$p1 = \{ (\{(x, x^i), (y, y^i)\} , \{(x, x^f), (y, y^i)\}): x^i, y^i \in int32 \wedge x^i > 0 \wedge \mathbf{x^f = -\sqrt{x^i}} \}$

i.e. $p1 \leq q$ in this case $p1 \subset q$, p1 is less nondeterministic than q

The Conformity Relation $p \leq q$





$p1 \subset q$ p1 is less nondeterministic than q

The Conformity Relation $p \leq q$

On the other hand,

another program p2 may work for more initial values of x:

$p2 = \{ (\{(x, x^i), (y, y^i)\} , \{(x, x^f), (y, y^i)\}):$

$x^i, y^i \in \text{int32} \wedge \mathbf{x^i \geq 0} \wedge (x^f = \sqrt{x^i} \vee x^f = -\sqrt{x^i}) \}$

Again we say that $p2 \leq q$ i.e. p2 conforms to q

In this case we have $dom(p2) \supset dom(q)$

Definition of the Conformity Relation $p \leq q$

p conforms to q

$p \leq q \quad \stackrel{d}{\equiv} \quad p|_{dom(q)} \subseteq q \quad \wedge$ less-equal nondeterministic

$dom(p) \supseteq dom(q)$ may work for more i-states

$p3 = \{ (\{(x, x^i), (y, y^i)\} , \{(x, x^f), (y, y^i)\}):$

$x^i, y^i \in int32 \wedge \mathbf{x^i \geq 0} \wedge x^f = -\sqrt{x^i} \}$ again: $p3 \leq q$

Liberal conformity

(more strict: $p \leq q \stackrel{d}{\equiv} dom(p) = dom(q) \wedge p \subseteq q$)

Definition of the Conformity Relation $p \leq q$

p conforms to q

$p \leq q \quad \underset{d}{\equiv} \quad p|_{dom(q)} \subseteq q \quad \wedge$ less-equal nondeterministic

$dom(p) \supseteq dom(q)$ may work for more i-states

applies generally to specs and programs:

$q1 \leq q2$

$p1 \leq p2$

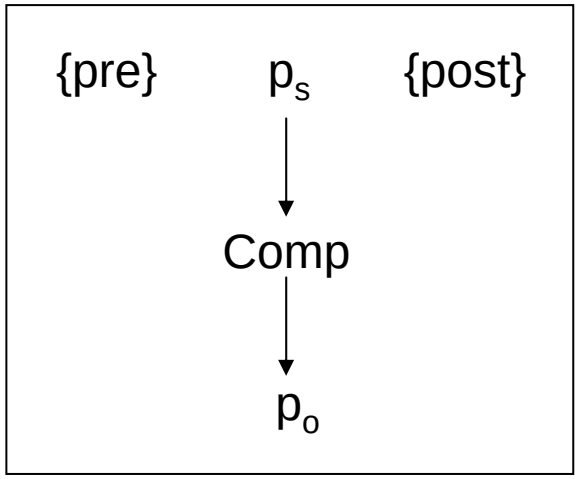
$\leq$ is transitive $p \leq q \wedge q \leq r \Rightarrow p \leq r$

Program Verification (serious viewpoint)

Program p : typically means high-level source-program p_s
(Ada, C#, Fortran, etc.)

Specification q : typically refers to the IO-behavior: q = (pre, post)
(high-level = low-level)

Program execution: refers to the object-program p_o produced by a compiler
(sometimes also execution by an interpreter)

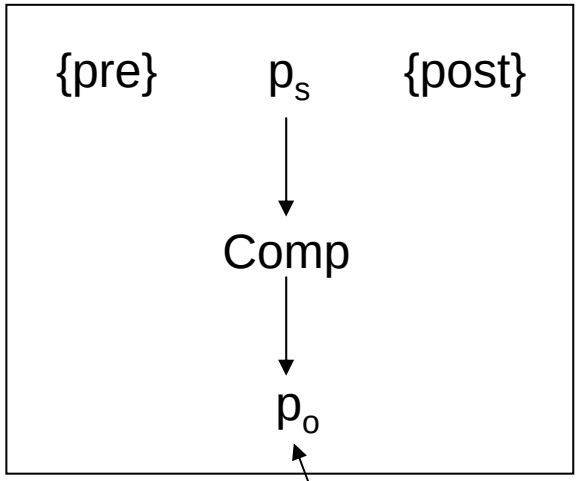


$p_s \leq$ (pre, post) is not sufficient

$p_o \leq$ (pre, post) is the really important thing

$p_o \leq p_s \ \wedge \ p_s \leq$ (pre, post) $\Rightarrow \ p_o \leq$ (pre, post)

Program Verification (serious viewpoint) - 2



$p_o \leq p_s$ refers to compiler correctness:

$\langle \forall\ p_s \in PL: Comp(p_s) \leq p_s \rangle$

Instead of proving $p_o \leq p_s$ each time

prove once that Comp is correct

12 Test the system to ensure that your compiler, linker etc. have not introduced errors and to make sure the required performance has been achieved. If improved performance is still needed in some areas, repeat from step 8.

http://www.eschertech.com/products/step_by_step_guide.php (2006.Aug.31)

Program Verification (serious viewpoint) - 3

Further aspects

Finiteness	:	value sets of types are finite
Several sorts	:	real programs often use several types also: user defined types also: complex types (e.g. class, task) also: pointers
Definedness	:	real programs may contain undefined expressions 1/0, X := 1/(X-X);
Exceptions	:	real programs may contain exception handling
Concurrency	:	becomes more important in practice (e.g. multi-core)

Program Verification (in the literature)

Program p : typically means high-level source-program fragment p_s (e.g. guarded commands [Gri 83])

Specification q : typically refers to the values of program variables (pre, post)

Verification : shows that $p_s \leq$ (pre, post) holds

Program execution: no execution takes place

Usually not done : $q_{IO} \leq$ (pre, post) IO-oriented spec q_{IO}

$p_{PL} \leq q_{IO}$ program in real progr lang p_{PL}

$p_o \leq p_{PL}$

$p_o \leq p_{PL} \leq q_{IO} \leq$ (pre, post) $\Rightarrow$ $p_o \leq$ (pre, post)

surprising observation: $p_s \leq$ (pre, post) seems to be superfluous

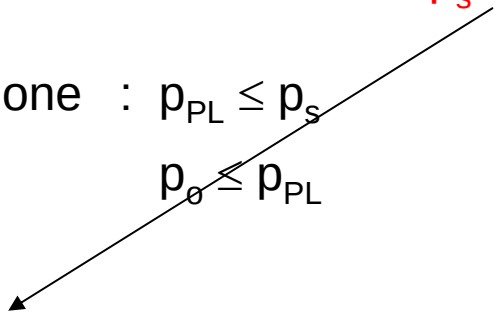
Program Verification (in the literature) - 2

If the spec q refers to I/O we obtain:

Verification : shows that $p_s \leq (pre, post) = q_{IO}$ holds

Usually not done : $p_{PL} \leq p_s$ program in real progr lang p_{PL}

$p_o \leq p_{PL}$



$p_o \leq p_{PL} \leq p_s \leq (pre, post) \Rightarrow p_o \leq (pre, post)$

Program Verification (in the literature) - 3

Further aspects

Finiteness	:	typically: integer = $\mathbb{Z}$ “the numerical size of MAXINT does not belong to a language definition” [TZ 88: 7]
Several sorts	:	typically: integer and Boolean
Definedness	:	typically: only well-defined expressions e.g. no division operation in [TZ 88]
Exceptions	:	typically: no exception handling
Concurrency	:	typically: only simple sequential commands

Why this Confrontation of Theory and Practice ?

- NOT to blame any theoretician !!!
- remind us of our ultimate goal
- show that we are a far cry from this goal
- indicate which path we should follow in our work towards (mechanical) program verification

Turing mentioned already the essential aspects

- correctness: “routine … is right?”
- interspersed assertions to ease verification
  compositionality of the verification of the whole program
- initial and final values of program variables
  very variable nature of program variables
- finite value ranges <======
- termination function

Missing:

- formal semantics (the semantics of the operations was quite obvious)
- specification as a formal object
- calculus for verification

References

Bar 2000 Barnes, John G. P.: High integrity Ada: the SPARK approach. Addison-Wesley, Harlow etc., 2000.
0-201-17517-7

Bou 2006 Boute, Raymond T.: Calculational Semantics: Deriving Programming Theories from Equations by Functional Predicate Calculus. ACM TOPLAS 28, 4 (2006) 747..793

CZ 92 Clarke, Edmund; Zhao, Xudong: Analytica – A Theorem Prover for Mathematica. School of Computer Science, Carnegie Mellon University, Report CMU-CS-92-117, September 1992

Fei 2005 Feinerer, Ingo: Formal Program Verification: a Comparison of Selected Tools and Their Theoretical Foundations. Master Thesis, Vienna Univ. of Technology, Institute of Computer Languages, Jan. 2005

FKW 2002 Freining, Carsten; Kauer, Stefan; Winkler, Jürgen F. H.: Ein Vergleich der Programmbeweiser FPP, NPPV und SPARK. In: Winkler, Jürgen F. H.; Dencker, Peter; Keller, Hubert B.; Tonndorf, Michael (Hrsg): Ada Deutschland Tagung 2002 - Software für sicherheitskritische Systeme - Shaker Verlag, Aachen, 2002. 3-8265-9956-X, S. 127..145

References

GH 99 Hehner, Eric C. R.; Gravell, Andrew M.: Refinement Semantics and Loop Rules. In: FM'99, Vol. II, 1497..1510. LNCS 1709, Springer, Berlin etc., 1999. 3-540-66588-9

GN 47 Goldstine, Herman H.; Neumann, John von: Planning and Coding Problems for an Electronic Computing Instrument – Part II, Volume 1. In: Taub, A. H. (gen. ed.): John von Neumann – Collected Works – Volume V. Pergamon Press, Oxford etc. 1961, repr. 1976, 0-08-009571-2, pp. 80..151

Gri 83 Gries, David: The Science of Programming. Springer, New York etc., 2nd pr. 1983, 0-387-90641-X

Hoa 72 Hoare, C. A. R.: A Note on the FOR-Statement. BIT 12, 3 (1972) 334..341

Kau 99 Kauer, Stefan: Automatische Erzeugung von Verifikations- und Falsifikationsbedingungen sequentieller Programme. Dissertation, Friedrich Schiller University, 1999.Jan.27

KW 99 Kauer, Stefan; Winkler, Jürgen F. H.: A Comparison of the Program Provers NPPV and FPP. Friedrich Schiller University, Institute of Computer Science, Report Math / Inf / 99 / 28

References

KW 2007 Kauer, Stefan; Winkler, Jürgen F. H.: *Mechanical Generation of Invariants for FOR-Loops*. WING 2007, 1st Int. Workshop on Invariant Generation, Linz/Hagenberg, 2007Jun25-26

Tur 49 Turing, A.: Checking a Large Routine. In: Williams, L. R.; Campbell-Kelly, Martin (eds.): The Early British Computer Conferences. MIT Press, Cambridge/London, Tomash Publ., Los Angeles/San Francisco, 1989, 0-262-23136-0, pp. 70..72

TZ 88 Tucker, J. V.; Zucker, J: I.: Program Correctness over Abstract Data Types, with Error-State Semantics. CWI Monograph 6. North-Holland, Amsterdam etc, 1988.

Win 90 Winkler Jürgen F. H.: Functions not equivalent. Letter to the editor, IEEE Software 7,3 (1990)10.

Win 98 Winkler, Jürgen F. H.: New Proof Rules for FOR-Loops. Friedrich Schiller University, Dept. of Math. & Comp. Sci., Report Math / Inf / 98 / 13 1998.Nov.07