Mechanical Program Verification – Part 4

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

ELTE, Budapest, 08 – 12 Oct 2007

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 4

- Problems of FPP (and others)
- Towards More Realistic VCs
- Summary

FPP is an Experimental System

- small subset of Ada
  arrays and subprograms are the elements most missed

- somewhat naive verification conditions
  e.g. assumption integer = $\mathbb{Z}$
  program developer must write longer assertions
  => main problem: no soundness
  if FPP says "proved" the program may be incorrect !!!
  no completeness is a smaller problem

- based on Mathematica (2.2) and Analytica
  Mathematica is also not sound: $0^x = 0$: $0^{-1} = 1/0^1$??? [CZ 92: 27]

Correctness - 1

AdaZ-Program	=	Program + Specification + Comments (asserted program: ap)
Program p	=	non-comment part of the AdaZ-program
Specification q	=	assertion-part of the AdaZ-program
Asserted prog ap	=	(p, q)
Comments	=	the rest

Correctness ≡ p conforms to q ($p \leq q$)

this is meant wrt to the two relations of p and q

we assume that these relations describe the behavior of p as specified by the Ada lang spec

(e.g. finite domains for number types)

"ap is correct" or "ap is valid"

Incorrectness ≡ ¬correctness

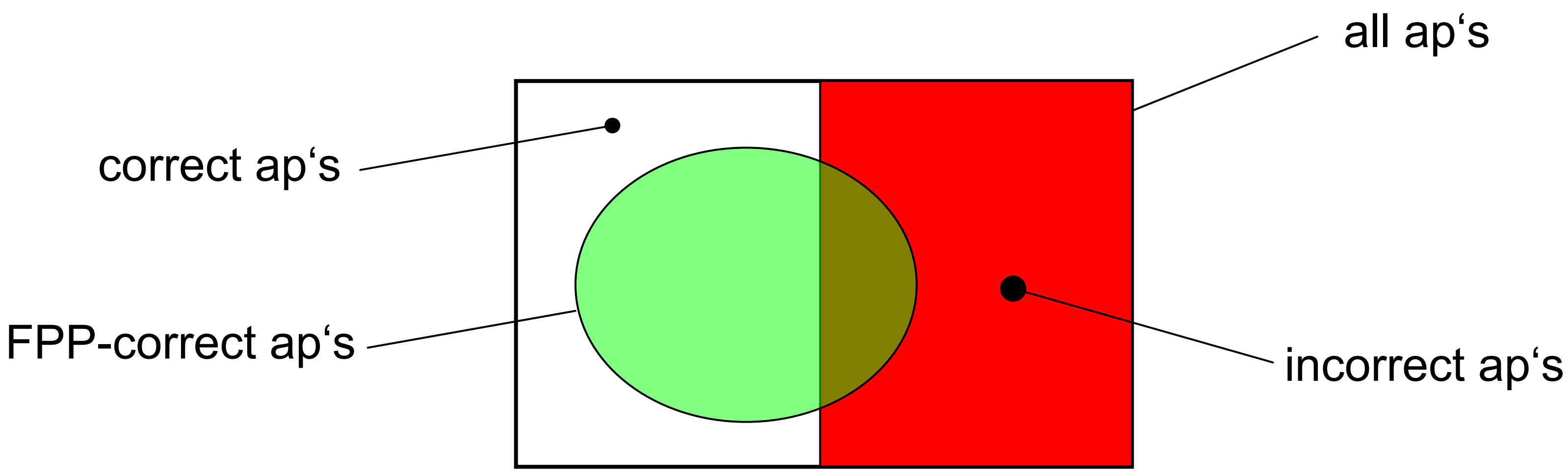
Correctness - 2

FPP-correctness $\equiv$ FPP says „proved“

FPP-Soundness $\equiv$ FPP-correctness $\Rightarrow$ Correctness

If FPP says „not proved“ there are two possibilities:

- $\neg(p \leq q)$
- $p \leq q$ but FPP cannot prove it



Integer = $\mathbb{Z}$

```
   -- naive counter
   --!pre: x = cx;
x := x+1;
   --!post: x = cx+1;
```

```
--!pre     : (x = cx)
--> wp     : (1 + x = 1 + cx)
--> vc     : (x = cx => 1 + x = 1 + cx)
--> Result: proved
x := x + 1;
--!post    : (x = 1 + cx)
```

Execution with

xc = integer’last

raised CONSTRAINT_ERROR :
naive-count.adb:7
overflow check failed

=> spec is idealistic (suitable for Cantor’s Paradise)

Integer = $\mathbb{Z}$

That was the worst that can happen for an APP (the “deadly sin of PV”)

APP says “proved”

=> people feel secure and safe

but the program (may be that which controls an air-plane) crashes at runtime

Integer = ℤ

```
   -- semi-naive counter
   --!pre: x = cx and -100 <= x and x <= 100;
x := x+1;
   --!post: x = cx+1 and -100 <= x and x <= 100;
```

Type condition

```
--!pre      : (x = cx AND -100 <= x AND x <= 100)
--> wp      : (1 + x = 1 + cx AND -100 <= 1 + x AND 1 + x <= 100)
--> vc      :     (x = cx AND -100 <= x AND x <= 100)
-->           ==> (1 + x = 1 + cx AND -100 <= 1 + x AND 1 + x <= 100)
--> Result: not proved
--> fc      : (-99 + cx >= 1 AND 100 - cx >= 0 AND 100 + cx >= 0)
x := x + 1;
--!post     : (x = 1 + cx AND -100 <= x AND x <= 100)
```

=> spec is adequate => program is not adequate

Integer = $\mathbb{Z}$

```
   -- semi-professional counter
  --!pre: x = cx and -100 <= x and x <= 100;
IF x<100
THEN x := x+1;
END IF;
  --!post: ((cx<100 ∧ x=cx+1) or (cx=100 ∧ x=cx)) ∧ -100<=x ∧ x<=100;

--!pre    :  (x = cx AND -100 <= x AND x <= 100)
--> wp    :  (100 >= 1 + x) ...
--> vc    :  (x = cx AND -100 <= x AND x <= 100)   ==>   (100 >= 1 + x)...
--> Result: proved
IF x < 100 THEN
  x := x + 1;
END IF;
--!post  (100 >= 1+cx  ∧  x = 1+cx OR cx = 100  ∧  x=cx)  ∧ (-100 <= x) ∧ (x <= 100)
```

=> professional counter ??? => exercise for the audience

Problematic expressions - 1

```
    --!pre: true;
 b := 0/0 = 0/0;
    --!post: b;
```

1. Reflexivity of equivalence: $E \equiv E$ [1)]

```
--!pre      : (True)
--> wp      : (True)
--> vc      : (True)
--> Result: proved
b := 0 / 0 = 0 / 0;
--!post     : (b)
```

Ada-compilation

undef01.adb:6:10: division by zero

undef01.adb:6:10: static expression raises „Constraint_Error“

Java

compilation successful

Exception in thread "main"
java.lang.ArithmeticException: / by zero
at Undef01.main(Undef01.java:5)

1): „Rules of Logic“: http://cerebro.xu.edu/csci370/00f/Assignments/LogicOverheads/RulesOfLogic.pdf#search=%22logical%20laws%20equivalence%20reflexivity%22 2006.Sep.29

Problematic expressions - 2

=> mechanical program verification means that the APP must deal in a sensible and SOUND manner with ANY INPUT

=> same situation as with naïve counter

In mathematics division by zero is just “forbidden” or at least evaded:

“At first sight we can't add a symbol to express 1/*x*, since all the named functions have to be defined on the whole domain of the structure, and there is no such real number as 1/0. But on second thoughts this is not a serious problem; any competent mathematician puts the condition ‘*x* is not zero’ before dividing by *x*, and so it never matters what the value of 1/0 is, and we can harmlessly take it to be 42.
But most model theorists are uncomfortable with any kind of division by zero, so they stick with plus, times and minus.”

http://plato.stanford.edu/entries/modeltheory-fo/ 2005.Jan.30

That is no solution here

Problematic expressions - 4

```
X: integer;
. . .
   --# assert True;
X := 1/(X-X);
   --# assert 1/(X-X) = 1/(X-X);
X := 1;
   --# assert X=1;
```

SPARK95 5.01 Examiner + Simplifier:

“all conclusions proved”

2004.Jul.13

Problematic expressions - 3

let a and b be two numerical variables

What is the value of:

$$(a < b) \vee (a >= b) \qquad ???$$

where “<“ and “>=“ are predeclared operations in some programming language.

Towards realistic semantics and VCs

Semantics is specified by wp-rules

VC: pre $\Rightarrow$ wp(progr, post)
- schema for loops
- adaptation rule for procedure calls

Assumptions

- ap is syntactically legal
- ap is legal wrt static context conditions of the resp language
  - all entities properly defined
  - legal typing
  - accessibility
- no function calls in expressions

Must also be checked by APP but are not expressed in the VCs

Towards realistic semantics and VCs

Assignment

variable := expression; -- let v be scalar

v := e;

wp("v := e;", post) $\equiv$ ec('e) |$\wedge$ (e $\in$ Type(v) $\wedge$ $post^{v}_{e}$)

ec('e) : e can be effectively computed (eg no exceptions)
this includes also the eff computation of all intermediary results
'e means: no evaluation or simplification of e

|$\wedge$: evaluate from left to right
stop if value is definitely true or false

Type(v): the value set of the type of v

Towards realistic semantics and VCs

$ec(`v) \equiv true$

$ec(`literal) \equiv literal \in Type(literal)$

$v := v;$ -- $Type(v) = [1, 10]$

$wp(“v := v;”, true) \equiv ec(`e) \ |\!\wedge\ (e \in Type(v) \wedge post^v{}_e)$

$\equiv \underline{ec(`v)} \ |\!\wedge\ (v \in [1, 10] \wedge true^v{}_v)$

$\equiv \underline{true\ |\!\wedge}\ (v \in [1, 10] \wedge true^v{}_v)$

$\equiv v \in [1, 10] \wedge \underline{true^v{}_v}$

$\equiv v \in [1, 10] \underline{\wedge\ true}$

$\equiv v \in [1, 10]$

=> this solves the Pascal example

Towards realistic semantics and VCs

$\text{ec}(\text{‘}(e1/e2)) \equiv$ -- integer division

$\text{ec}(\text{‘}e1) \mathbin{|\!\wedge} \text{ec}(\text{‘}e2) \mathbin{|\!\wedge} (e2 \neq 0 \wedge e1 \in \text{Type}(/.lo) \wedge e2 \in \text{Type}(/.ro))$

wp(“x := 1/(x-x);”, true)

$\equiv$ -- assg rule: $\text{ec}(\text{‘}e) \mathbin{|\!\wedge} (e \in \text{Type}(v) \wedge \text{post}^{v}_{e})$

$\underline{\text{ec}(\text{‘}(1/(x\text{-}x)))} \mathbin{|\!\wedge} (1/(x\text{-}x) \in \text{Type}(x) \wedge \text{true}^{x}_{1/(x\text{-}x)})$

$\equiv [\underline{\text{ec}(\text{‘}1)} \mathbin{|\!\wedge} \text{ec}(\text{‘}(x\text{-}x)) \mathbin{|\!\wedge} (x\text{-}x \neq 0 \wedge 1 \in \text{Int} \wedge x\text{-}x \in \text{Int})]$
$\mathbin{|\!\wedge} (1/(x\text{-}x) \in \text{Type}(x) \wedge \text{true}^{x}_{1/(x\text{-}x)})$

$\equiv [\underline{1 \in \text{Int} \mathbin{|\!\wedge} \text{ec}(\text{‘}(x\text{-}x))} \mathbin{|\!\wedge} (x\text{-}x \neq 0 \wedge 1 \in \text{Int} \wedge x\text{-}x \in \text{Int})]$
$\mathbin{|\!\wedge} (1/(x\text{-}x) \in \text{Type}(x) \wedge \text{true}^{x}_{1/(x\text{-}x)})$

$\equiv [\,[\text{ec}(\text{‘}x) \mathbin{|\!\wedge} \text{ec}(\text{‘}x) \mathbin{|\!\wedge} (x \in \text{Type}(\text{-}.lo) \wedge x \in \text{Type}(\text{-}.ro))]$
$\mathbin{|\!\wedge} (x\text{-}x \neq 0 \wedge 1 \in \text{Int} \wedge x\text{-}x \in \text{Int})]$
$\mathbin{|\!\wedge} (1/(x\text{-}x) \in \text{Type}(x) \wedge \text{true}^{x}_{1/(x\text{-}x)})$

Towards realistic semantics and VCs

$\equiv$ $[\ [\underline{ec(`x)\ |}\wedge\ \underline{ec(`x)\ |}\wedge\ \ (\underline{x \in Int} \wedge \underline{x \in Int})]$
$|\wedge\ \ (x\text{-}x \neq 0 \wedge 1 \in Int \wedge x\text{-}x \in Int)]$
$|\wedge\ \ (1/(x\text{-}x) \in Type(x)\ \ \wedge\ \ true^{x}{}_{1/(x\text{-}x)})$

$\equiv$ $[\ x \in Int\ \ |\wedge\ \ (\underline{x\text{-}x} \neq 0 \wedge 1 \in Int \wedge x\text{-}x \in Int)]$
$|\wedge\ \ (1/(x\text{-}x) \in Type(x)\ \ \wedge\ \ true^{x}{}_{1/(x\text{-}x)})$

$\equiv$ $[\ x \in Int\ \ |\wedge\ \ (\underline{0 \neq 0} \wedge 1 \in Int \wedge x\text{-}x \in Int)]$
$|\wedge\ \ (1/(x\text{-}x) \in Type(x)\ \ \wedge\ \ true^{x}{}_{1/(x\text{-}x)})$

$\equiv$ $[\ x \in Int\ \ |\wedge\ \ (\underline{False} \wedge \underline{1 \in Int} \wedge \underline{x\text{-}x \in Int)}]$
$|\wedge\ \ (1/(x\text{-}x) \in Type(x)\ \ \wedge\ \ true^{x}{}_{1/(x\text{-}x)})$

$\equiv$ $[\ x \in Int\ \ |\wedge\ \ False]\ \ |\wedge\ \ (1/(x\text{-}x) \in Type(x)\ \ \wedge\ \ true^{x}{}_{1/(x\text{-}x)})$

$\equiv$ False

=> better than FPP and SPARK

Towards realistic semantics and VCs

wp(“v := e;”, post) $\equiv$ ec(‘e) |$\wedge$ (e $\in$ Type(v) $\wedge$ $post^{v}_{e}$)

everything OK now ?

what about problematic expressions in assertions ?

Towards realistic semantics and VCs

```
   --!pre: y >= 0;
x := 0;
   --!post: y/y > 0;
```

Sometimes people (and tools) apply naively: y/y = 1

```
   --!pre: y >= 0;
x := 0;
   --!post: 1 > 0;
```

$$
\begin{aligned}
wp(\text{“}x := 0;\text{”}, True) &\equiv ec(\text{‘}0) \mid\!\wedge (0 \in Type(x) \wedge True^x{}_0) \\
&\equiv True \mid\!\wedge (0 \in Type(x) \wedge True^x{}_0) \\
&\equiv 0 \in Type(x) \wedge True^x{}_0 \\
&\equiv True
\end{aligned}
$$

$$
\begin{aligned}
& pre \Rightarrow wp(\text{“}x := 0;\text{”}, True) \\
\equiv\ & y>=0 \Rightarrow True \\
\equiv\ & True
\end{aligned}
$$

Towards realistic semantics and VCs

Assume: y has a proper value: $y=0 \vee y\neq0$

Then $(y=0 \Rightarrow y/y = NaN) \wedge (y\neq0 \Rightarrow y/y = 1) \equiv True$

```
  --!pre: y >= 0;
x := 0;
   --!post: y/y > 0;
```

wp("x := 0;", y/y > 0)

$\equiv$ ec('0) $|\wedge$ $(0 \in Type(x) \wedge (y=0 \Rightarrow y/y = NaN) \wedge (y\neq0 \Rightarrow y/y = 1) \wedge (y/y > 0)^{x}_{0})$

$\equiv$ True $|\wedge$ $(0 \in Type(x) \wedge (y=0 \Rightarrow y/y = NaN) \wedge (y\neq0 \Rightarrow y/y = 1) \wedge (y/y > 0)^{x}_{0})$

$\equiv$ $0 \in Type(x) \wedge (y=0 \Rightarrow y/y = NaN) \wedge (y\neq0 \Rightarrow y/y = 1) \wedge y/y > 0$

$\equiv$ $(y=0 \Rightarrow y/y = NaN) \wedge (y\neq0 \Rightarrow y/y = 1) \wedge y/y > 0$

Towards realistic semantics and VCs

$\langle \forall x, y\text{: pre } \Rightarrow wp(\text{“}x := 0;\text{”}, y/y > 0) \rangle$

$\equiv \langle \forall x, y\text{: } y >= 0 \Rightarrow (y=0 \Rightarrow y/y = NaN) \wedge (y\neq 0 \Rightarrow y/y = 1) \wedge y/y > 0 \rangle$

$\equiv \langle y >= 0 \Rightarrow (y=0 \Rightarrow y/y = NaN) \wedge (y\neq 0 \Rightarrow y/y = 1) \wedge y/y > 0 \rangle^{y}_{0} \wedge R$

$\equiv 0 >= 0 \Rightarrow (0=0 \Rightarrow 0/0 = NaN) \wedge (0\neq 0 \Rightarrow 0/0 = 1) \wedge 0/0 > 0 \wedge R$

$\equiv \text{True} \Rightarrow (\text{True} \Rightarrow 0/0 = NaN) \wedge (\text{False} \Rightarrow 0/0 = 1) \wedge 0/0 > 0 \wedge R$

$\equiv 0/0 = NaN \wedge \text{True} \wedge 0/0 > 0 \wedge R$

$\equiv 0/0 = NaN \wedge NaN > 0 \wedge R$

$\equiv$ -- C#, Java: $NaN > 0 \equiv \text{False}$

$\equiv 0/0 = NaN \wedge \text{False} \wedge R$

$\equiv \text{False}$

This is more realistic

Towards realistic semantics and VCs

For floating point division x/y in C# (IEC 60 559) :

x, y means proper value $\neq 0$

(Table from ECMA-334 June 2006)

	+y	–y	+0	–0	+∞	–∞	NaN
+x	+z	–z	+∞	–∞	+0	–0	NaN
–x	–z	+z	–∞	+∞	–0	+0	NaN
+0	+0	–0	NaN	NaN	+0	–0	NaN
–0	–0	+0	NaN	NaN	–0	+0	NaN
+∞	+∞	–∞	+∞	–∞	NaN	NaN	NaN
–∞	–∞	+∞	–∞	+∞	NaN	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

Towards realistic semantics and VCs

Sometimes people (and tools) naively apply: (a*c)/(b*c) = a/b

```
  --!pre: a>0 and b>0 and c=0;
cond := (a*c)/(b*c) = a/b;
  --!post: cond;

User: 141.35.12.27       At: 2006.09.22, 8:59

--!pre       : (a >= 1 AND b >= 1 AND c = 0)
--> wp        : (True)
--> vc        : (True)
--> Result: proved                              !!!!!
cond := (a * c) / (b * c) = a / b;
--!post      : (cond)
```

FPP: AdaZ: wp

d) if-statement

wp("if Cond Then S1 Else S2 End If;", post) $\equiv$

Cond $\wedge$ wp("S1", post) $\vee$ $\neg$Cond $\wedge$ wp("S2", post)

Cond must also be effectively computable

wp("if Cond Then S1 Else S2 End If;", post) $\equiv$

ec('Cond) |$\wedge$ [Cond $\wedge$ wp("S1", post) $\vee$ $\neg$Cond $\wedge$ wp("S2", post)]

Cond $\in$ Boolean seems guaranted (at least in IEC 60559)

Towards realistic semantics and VCs

f) FOR-loop: VC (based on [Hoa 72])

```
    -- pre
FOR  id  IN  e1 .. e2
LOOP
        -- e1≤e2 ∧ inv^id_pred(id)
    statm-sequence (ss)
        --  inv
END LOOP;
    -- post
```

$$\begin{aligned}
\text{init} &\equiv e1 \le e2 \wedge \text{pre} \Rightarrow \text{inv}^{id}_{pred(e1)} \\
\text{null} &\equiv e1 > e2 \wedge \text{pre} \Rightarrow \text{post} \\
\text{ind} &\equiv e1 \le e2 \wedge \text{inv}^{id}_{pred(id)} \Rightarrow \text{wp}(ss, \text{inv}) \\
\text{final} &\equiv e1 \le e2 \wedge \text{inv}^{id}_{e2} \Rightarrow \text{post} \\
\text{VC} &\equiv \text{init} \wedge \text{null} \wedge \text{ind} \wedge \text{final}
\end{aligned}$$

e1 and e2 must also be effectively computable

$\Rightarrow$ VC $\equiv$ ec('e1) $|\wedge$ ec('e2) $|\wedge$ e1,e2 $\in$ Type(id) $\wedge$ init $\wedge$ null $\wedge$ ind $\wedge$ final

Towards realistic semantics and VCs

f) WHILE-loop: VC

```
    -- pre
    -- inv
WHILE  cond
LOOP
        -- inv ∧ term>0 ∧ term=T
    statm-sequence (ss)
        -- inv ∧ term<T
END LOOP;
    -- post
```

$init_f \equiv pre \Rightarrow inv$

$null_f \equiv inv \wedge \neg cond \Rightarrow post$

$init_t \equiv cond \wedge inv \Rightarrow term>0$

$ind_f \equiv cond \wedge inv \Rightarrow wp(ss, inv)$

$ind_t \equiv cond \wedge inv \Rightarrow [wp(ss, term<T)]^T_{term}$

$final_f \equiv inv \wedge \neg cond \Rightarrow post$

$VC \equiv init_f \wedge ind_f \wedge final_f \wedge init_t \wedge ind_t$

cond must also be effectively computable

=> $VC \equiv ec(\text{'}cond) \mid\!\wedge\ init_f \wedge ind_f \wedge final_f \wedge init_t \wedge ind_t$

Summary

- Historical overview: GvN, Turing, Floyd, Hoare, Dijkstra
- Serious Program Verification
- Mechanical Verification of real programs
  APP computes and tries to prove VC
- Comparison of APPs (FPP, NPPV, SPARK-aut)
- Mechanical generation of invariants for FOR-loops
- No Soundness due to idealistic APPs (e.g. integer = $\mathbb{Z}$)
- Towards more realistic VCs
  how do real programs work ? ($(a < b) \vee (a \geq b)$)
- Undefined expressions have to be tackled ($(a*c)/(b*c) = a/b$)
- More logic at school and university necessary ($e \Rightarrow true$)
  especially: more practice in logical calculations

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
1976	Dijkstra	: function (wp) and schemas for valid triples
2003	Hoare	: Verifying compiler as a grand challenge
2006	Hoare (Budapest*))	: Program Verifier as Grand Challenge of Informatics

*): http://www.cs.bme.hu/~szeredi/ae-is-budapest/symp.html#SECTION00031000000000000000

Program Verification (serious viewpoint)

{pre} p_s {post}

$$p_s \downarrow \text{Comp} \downarrow p_o$$

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

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

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

Correctness of the compiler

And you gave the hint: correctness of the OS, Processor, Loader, …

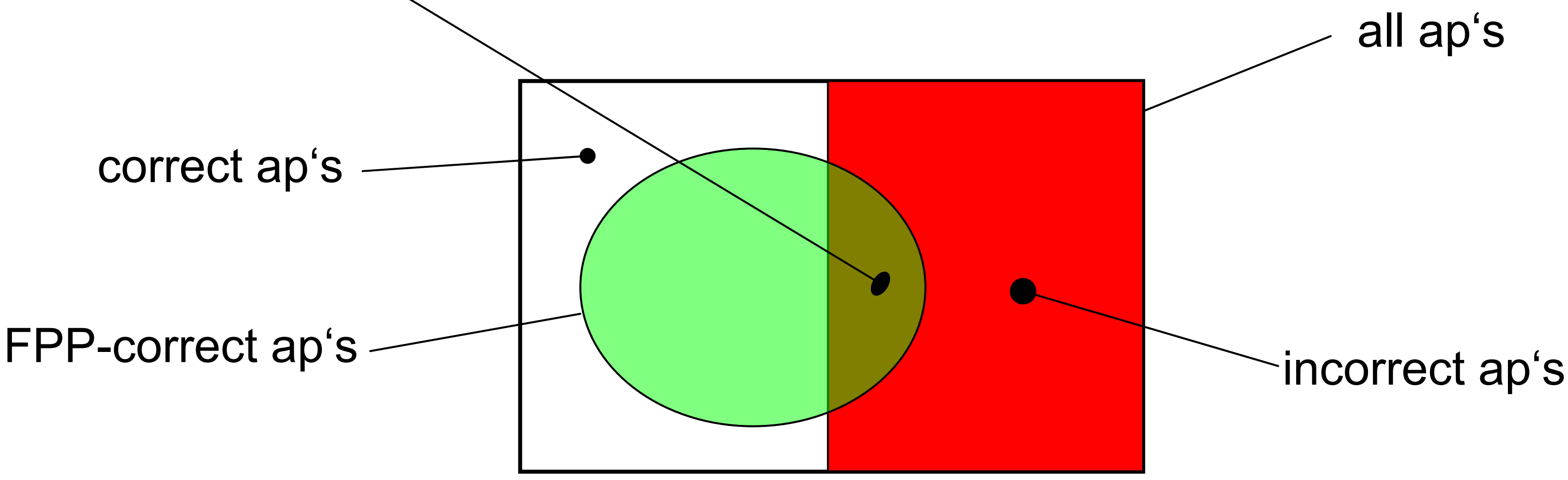
And we all forgot: correctness of the APP

was the first question to Hoare after his talk on 2006.Sep.19

Correctness - 2

Soundness: FPP-correctness $\Rightarrow$ Correctness

```
--!pre     : (a >= 1 AND b >= 1 AND c = 0)
--> wp     : (True)
--> vc     : (True)
--> Result: proved   !!!!!
cond := (a * c) / (b * c) = a / b;
--!post    : (cond)
```



Conclusion

Mechanical Program Verification

- promises great advantages
- is still in its infancy
- requires a realistic logic and realistic VCs
- is a whole new technology (as e.g. compiling)
- a “Great Challenge”

=> a lot of work to do (beginning at school)

and … don’t be afraid of heuristics

Thank you very much

I hope you‘ve learnt something

Good Bye

Rest

Mechanical Program Verification – Part 4

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

ELTE, Budapest, 25 – 29 Sep 2006