A Taste of Program Verification

Jorge Sousa Pinto
jsp@di.uminho.pt

Outline

- Formal models and the central problem of formal methods
- Introduction to Hoare Logic; verification by hand
- Specifying the behaviour of C programs
- Case study: array partition algorithm

Outline

- Program Annotation and Design by Contract
- JML Tool Demo: ESC/Java2 + Simplify
- Tool Demo: Caduceus + Coq

The Central Problem of Formal Methods

Models: Tools and Approaches

- Abstract State Machines (B)
- Automata-based Models
- Process Algebra (Esterel)
- Set and Category Theory (Z, VDM, Charity)
- Algebraic Specifications (OBJ)
- Declarative Modeling (FP, LP, TRS)
- *Preconditions and PostConditions*

The Central Problem of FM

Part 1: *model validation*

- How to enforce, at the specification level, the desired behaviour?

  Prove properties about the model

Tools for Formal Verification

- Proof Systems:

  Theorem Provers / Proof Assistants

- Model Checkers

The Central Problem of FM

Part 2: relation between specifications and implementations

- How to obtain, from a specification, an implementation with the same behaviour? *Extraction; Program Derivation*

Or alternatively,

- Given an implementation, how can it be checked that it obeys the specification? *Testing; Program Verification*

Program Extraction

- From a proof of a logical property (typically concerning existential quantifications), the Coq system is capable of extracting a program into a working programming language

Program Derivation

- Stepwise Refinement from Specifications to Programs (Z, VDM, B, …)
- Two approaches to correctness:

  (i) the refinement steps generate *proof obligations* that must be discharged. Derivations are thus formally verified.

  (ii) the refinement process is itself verified to be correct. The derived programs are then *correct by construction*.

Program Verification

- Given a program and a specification, check that the former conforms to the latter.
- This is the only applicable method in many situations
- THIS LECTURE: an approach to program verification based on *program annotation* and Hoare Logic

Hoare Logic

- A formal system that is useful for
  - Correct by construction program derivation extensive bibliography: Kaldewaij; Gries; Backhouse; Dijkstra
  - *Our focus*: Program Verification
- Formulas assert that if a given precondition holds prior to program execution, then a postcondition will hold after execution

A Toy Programming Language

Types (data and expressions):

$$\begin{array}{rcl} \tau & ::= & \mathbf{bool} \mid \mathbf{int} \\ \theta & ::= & \mathbf{var} \mid \mathbf{exp}[\tau] \mid \mathbf{com} \mid \mathbf{assert} \end{array}$$

Interpreted as expected:

$$[\![\mathbf{bool}]\!] = \{\mathit{true}, \mathit{false}\}$$

$$[\![\mathbf{int}]\!] = \{\ldots -2, -1, 0, 1, 2, \ldots\}$$

Espressions, commands and assertions are interpreted in a given state of the program

$\mathcal{I} = \{x, y, z, \ldots\}$

$$
\begin{aligned}
\Sigma &= \mathcal{I} \to [\![\mathbf{int}]\!] \\
[\![\mathbf{exp}[\tau]]\!] &= \Sigma \to [\![\tau]\!]_\perp \\
[\![\mathbf{com}]\!] &= \Sigma \to \Sigma_\perp \\
[\![\mathbf{assert}]\!] &= \Sigma \to \{true, false\}
\end{aligned}
$$

Abstract Syntax

var, **int**, **exp**[**bool**], **exp**[**int**], **com**, and **assert**

$$
\begin{array}{rcl}
V & ::= & x, y, z, \ldots \\
L & ::= & x, y, z, \ldots \\
B & ::= & \mathbf{true} \mid \mathbf{false} \\
 & \mid & B \mathbin{\&\&} B \mid B \mathbin{||} B \mid !B \\
 & \mid & E == E \mid E < E \mid E <= E \mid E > E \mid E >= E \mid E \,!= E \\
E & ::= & \ldots -2, -1, 0, 1, 2 \ldots \mid V \mid L \\
 & \mid & -E \mid E + E \mid E - E \mid E * E \mid E \ \mathbf{div}\ E \mid E \ \mathbf{mod}\ E \\
C & ::= & \mathbf{skip} \mid C\,;C \mid V := E \mid \mathbf{if}\ E\ \mathbf{then}\ C\ \mathbf{else}\ C \mid \mathbf{while}\,(E)\,\mathbf{do}\,C \\
A & ::= & \mathbf{true} \mid \mathbf{false} \\
 & \mid & A \mathbin{\&\&} A \mid A \mathbin{||} A \mid !A \mid \forall\, L.\, A \mid \exists\, L.\, A \mid A \rightarrow A \\
 & \mid & E == E \mid E < E \mid E <= E \mid E > E \mid E >= E \mid E \,!= E
\end{array}
$$

Interpretation of Expressions

$$
\begin{aligned}
[\![\mathbf{true}]\!](s) &= \mathit{true} \\
[\![\mathbf{false}]\!](s) &= \mathit{false} \\
[\![n]\!](s) &= n \qquad \text{for } n \in \{\ldots -2, -1, 0, 1, 2, \ldots\}
\end{aligned}
$$

$$
[\![x]\!](s) = \begin{cases} s(x) & \text{if } x \in \mathrm{dom}(s) \\ \bot & \text{otherwise} \end{cases}
$$

$$
[\![\, !e]\!](s) = \begin{cases} \mathit{true} & \text{if } [\![e]\!](s) = \mathit{false} \\ \mathit{false} & \text{if } [\![e]\!](s) = \mathit{true} \\ \bot & \text{if } [\![e]\!](s) = \bot \end{cases}
$$

$$
[\![-e]\!](s) = \begin{cases} -[\![e]\!](s) & \text{if } [\![e]\!](s) \neq \bot \\ \bot & \text{if } [\![e]\!](s) = \bot \end{cases}
$$

$$
[\![e_1 \oplus e_2]\!](s) = \begin{cases} [\![e_1]\!](s) \oplus [\![e_2]\!](s) & \text{if } [\![e_1]\!](s) \neq \bot \text{ and } [\![e_2]\!](s) \neq \bot \\ \bot & \text{otherwise} \end{cases}
$$

$$
\text{for } \oplus \in \{\ \&\&\ ,\ ||\ , ==, <, <=, >, >=, !=, +, -, *, \ \mathbf{div}\ ,\ \mathbf{mod}\ \}
$$

Interpretation of Commands

$$
\begin{array}{rcl}
[\![\mathbf{skip}]\!](s) & = & s \\[2ex]
[\![C_1 \,;\, C_2]\!](s) & = & ([\![C_2]\!] \odot [\![C_1]\!])(s) \\[2ex]
 & & \text{where } [\![g \odot f]\!](s) = \begin{cases} \bot & \text{if } [\![f]\!](s) = \bot \\ g(f(s)) & \text{otherwise} \end{cases} \\[2ex]
[\![x := E]\!](s) & = & \begin{cases} s[x \mapsto [\![E]\!](s)] & \text{if } [\![E]\!](s) \neq \bot \\ \bot & \text{otherwise} \end{cases} \\[2ex]
[\![\mathbf{if}\ E_1\ \mathbf{then}\ E_2\ \mathbf{else}\ E_3]\!](s) & = & \mathsf{cond}([\![E_1]\!], [\![E_2]\!], [\![E_3]\!])(s) \\
 & & \text{and } \mathsf{cond}(p, f, g)(s) = \begin{cases} f(s) & \text{if } p(s) = \mathit{true} \\ g(s) & \text{if } p(s) = \mathit{false} \\ \bot & \text{otherwise} \end{cases} \\[2ex]
[\![\mathbf{while}\,(E)\,\mathbf{do}\,C]\!] & = & \mathrm{fix}\, F \\
 & & \text{where } F(f) = \mathsf{cond}([\![E]\!], f \odot [\![C]\!], \lambda x.x)
\end{array}
$$

Interpretation of Assertions (1)

$$
\begin{array}{rcl}
[\![\mathbf{true}]\!](s) & = & \mathit{true} \\
[\![\mathbf{false}]\!](s) & = & \mathit{false}
\end{array}
$$

$$
[\![\,!a]\!](s) = \begin{cases} \mathit{true} & \text{if } [\![e]\!](s) = \mathit{false} \\ \mathit{false} & \text{if } [\![e]\!](s) = \mathit{true} \end{cases}
$$

$$
\begin{array}{rcl}
[\![a_1 \mathbin{\&\&} a_2]\!](s) & = & [\![a_1]\!](s) \text{ and } [\![a_2]\!](s) \\
[\![a_1 \mathbin{||} a_2]\!](s) & = & [\![a_1]\!](s) \text{ or } [\![a_2]\!](s) \\
[\![a_1 \longrightarrow a_2]\!](s) & = & \text{if } [\![a_1]\!](s) \text{ then } [\![a_2]\!](s)
\end{array}
$$

$$
[\![e_1 < e_2]\!](s) = \begin{cases} [\![e_1]\!](s) < [\![e_2]\!](s) & \text{if } [\![e_1]\!](s) \neq \perp \text{ and } [\![e_2]\!](s) \neq \perp \\ \mathit{false} & \text{otherwise} \end{cases}
$$

$$
[\![e_1 == e_2]\!](s) = \begin{cases} [\![e_1]\!](s) == [\![e_2]\!](s) & \text{if } [\![e_1]\!](s) \neq \perp \text{ and } [\![e_2]\!](s) \neq \perp \\ \mathit{false} & \text{otherwise} \end{cases}
$$

Interpretation of Assertions (2)

$$
\begin{aligned}
[\![e_1 <= e_2]\!](s) &= [\![(e_1 < e_2) \mid\mid (e_1 == e_2)]\!](s) \\
[\![e_1 > e_2]\!](s) &= [\![\,!(e_1 <= e_2)]\!](s) \\
[\![e_1 >= e_2]\!](s) &= [\![\,!(e_1 < e_2)]\!](s) \\
[\![e_1 \,!= e_2]\!](s) &= [\![\,!(e_1 == e_2)]\!](s)
\end{aligned}
$$

$$[\![\forall\, x.\, a]\!](s) = \text{for every } v \in [\![\mathbf{int}]\!]_\perp,\ [\![a]\!](s)[x \mapsto v] = \mathit{true}$$

$$[\![\exists\, x.\, a]\!](s) = \text{for some } v \in [\![\mathbf{int}]\!]_\perp,\ [\![a]\!](s)[x \mapsto v] = \mathit{true}$$

Hoare Triple Formulas

$$\{P\}\, C\, \{Q\}$$

P, Q : **assert** are closed with respect to logical variables

C : **com** contains no occurrences of logical variables

meaning that if C executes in a state where P holds, then *if C terminates* Q will hold upon termination

Semantics of Hoare Triples

Given by the following interpretation in *{true, false}*, using the semantics of assertions

$$[\![\{P\}\, C\, \{Q\}]\!] \quad = \quad \text{if } [\![P]\!](s) \text{ then } [\![Q]\!](s')$$

$$\text{for all states } s, s' \in \Sigma \text{ such that } [\![C]\!](s) = s'.$$

P, Q may contain occurrences of program variables that do not occur in C. Such variables are called auxiliary

This is a *partial* notion of correctness since the program is not guaranteed to terminate.

If additionally the existence of s' is required, we are in the presence of total correctness formulas.

$$\{P\}\, C\, \{Q\} \text{ and } C \text{ terminates} \equiv [P]\, C\, [Q]$$

Inference System

An inference system can be defined that derives only valid Hoare triples: if

$$\{P\}\,C\,\{Q\}$$

is derived then

$$[\![\{P\}\,C\,\{Q\}]\!] \;=\; true$$

Skip and Composition

$$\frac{}{\{P\}\,\mathbf{skip}\,\{P\}}$$

$$\frac{\{P\}\,C_1\,\{Q\} \qquad \{Q\}\,C_2\,\{R\}}{\{P\}\,C_1; C_2\,\{R\}}$$

Assignment

Works backwards

$$\frac{}{\{Q[x \mapsto e]\}\; x := e\; \{Q\}}$$

Example: $\{x+1=4\}\; x := x+1\; \{x=4\}$

Conditional

$$\frac{\{P \;\&\&\; B\}\, C_t\, \{Q\} \qquad \{P \;\&\&\; !B\}\, C_f\, \{Q\}}{\{P\}\, \textbf{if}\ B\ \textbf{then}\ C_t\ \textbf{else}\ C_f\, \{Q\}}$$

Can you spot a minor imprecision here?

Loops

A fundamental notion: a *loop invariant* is a property that is preserved by the body of a loop, i.e. if it holds as a precondition *together with the loop condition* then it holds as a post-condition

$$\frac{\{I \;\&\&\; B\}\; C\; \{I\}}{\{I\}\;\mathbf{while}\,(B)\,\mathbf{do}\; C\;\{I \;\&\&\; \neg B\}}$$

The identification of loop invariants is a crucial task

Logical Rules

We also need rules that relate assertions with specifications. Preconditions can be strengthened or made disjuncts

$$\frac{P' \rightarrow P \qquad \{P\}\, C\, \{Q\}}{\{P'\}\, C\, \{Q\}}$$

$$\frac{\{P_1\}\, C\, \{Q\} \qquad \ldots \qquad \{P_n\}\, C\, \{Q\}}{\{P_1 \mid\mid \ldots \mid\mid P_n\}\, C\, \{Q\}}$$

More Logical Rules

Postconditions can be weakened or made conjuncts

$$\frac{\{P\}\,C\,\{Q\} \qquad Q \to Q'}{\{P\}\,C\,\{Q'\}}$$

$$\frac{\{P\}\,C\,\{Q_1\} \qquad \ldots \qquad \{P\}\,C\,\{Q_n\}}{\{P\}\,C\,\{Q_1 \,\&\&\, \ldots \,\&\&\, Q_n\}}$$

More Logical Rules

0-ary cases for conjunction and disjunction

$$\frac{}{\{\mathbf{false}\}\, C\, \{Q\}}$$

$$\frac{}{\{Q\}\, C\, \{\mathbf{true}\}}$$

Example: Verification by Hand

Take the exponentiation function

$$\begin{aligned} \exp(x, 0) &= 1 \\ \exp(x, n+1) &= x * \exp(x, n) \end{aligned}$$

We intend to write a program **calcexp** such that

$$\{\textbf{true}\}\ \textbf{calcexp}\ \{w = \exp(x,y)\}$$

In fact this needs to be refined with the help of auxiliary variables, not used by **calcexp**

```
z := 1;
w := 1;
while z <= y do
  w := w * x;
  z := z + 1;
```

$$\{\, x = X_0 \wedge\ y = Y_0 \}\ \textbf{calcexp}\ \{\, w = \exp(x, y) \wedge\ x = X_0 \wedge\ y = Y_0 \}$$

$$\{\, x = X_0 \wedge\ y = Y_0 \}\ \textbf{calcexp}\ \{\, w = \exp(X_0, Y_0) \}$$

The loop condition:

$$B \equiv z \leq y$$

```
while z <= y do
  w := w * x;
  z := z + 1;
```

The loop invariant P:

$$P \equiv I \wedge R \wedge w = \exp(X_0, z-1)$$

$$I \equiv x = X_0 \wedge y = Y_0$$

$$R \equiv 1 \leq z \leq y + 1$$

P will grant the postcondition upon termination

$$P \equiv I \wedge R \wedge w = \exp(X_0,\, z - 1)$$

Invariant preservation

$$\{P \wedge B\}\ C\ \{P\}$$

```
while z <= y do
  w := w * x;
  z := z + 1;
```

Start with assignment axioms

$$\frac{}{\{I \wedge 1 \leq z+1 \leq y+1 \wedge w = \exp(X_0, (z+1)-1)\}\ \texttt{z := z+1}\ \{P\}}$$

$$\frac{}{\{I \wedge 0 \leq z \leq y \wedge w = \exp(X_0,\, z)\}\ \texttt{z := z+1}\ \{P\}}\ (A_1)$$

$$P \equiv I \wedge R \wedge w = \exp(X_0,\ z - 1)$$

Invariant preservation

$$\{P \wedge B\}\ C\ \{P\}$$

```
while z <= y do
  w := w * x;
  z := z + 1;
```

A second assignment axiom

$$\{I \wedge 0 \leq z \leq y \wedge w * x = \exp(X_0,\ z)\}\ \texttt{w := w*x}\ \{I \wedge 0 \leq z \leq y \wedge w = \exp(X_0,\ z)\}$$

Simplifying and strengthening the precondition we get:

$$\{I \wedge 1 \leq z \leq y \wedge w = \exp(X_0,\ z - 1)\}\ \texttt{w := w*x}\ \{I \wedge z \geq 0 \wedge w = \exp(X_0,\ z)\}$$

$$P \equiv I \wedge R \wedge \; w = \exp(X_0,\; z-1)$$

Thus

$$\frac{}{\{P \wedge B\}\ \texttt{w := w*x}\ \{I \wedge \; z \geq 0 \wedge \; w = \exp(X_0,\; z)\}}\ (A_2)$$

And these can now be sequenced

$$\frac{\dfrac{A_2 \qquad\qquad A_1}{\{P \wedge B\}\ C\ \{P\}}\ ;}{\{P\}\ \texttt{while B do C}\ \{P \wedge \neg B\}}$$

Similarly for the initializations, going backwards

$$\frac{}{\{I \wedge 1 \leq z \leq y+1 \wedge 1 = \exp(X_0, z-1)\}\ \texttt{w := 1}\ \{P\}}\ (A_3)$$

$$\frac{}{\{I\}\ \texttt{z := 1}\ \{I \wedge 1 \leq z \leq y+1 \wedge 1 = \exp(X_0, z-1)\}}\ (A_4)$$

$$\frac{A_4 \qquad A_3}{\{I\}\ \texttt{z := 1; w := 1}\ \{P\}}$$

Sequencing:

$$\dfrac{\dfrac{A_4 \qquad\qquad A_3}{\{I\}\ \texttt{z := 1; w := 1}\ \{P\}}\ ; \qquad \dfrac{\dfrac{A_2 \qquad\qquad A_1}{\{P \wedge B\}\ C\ \{P\}}\ ;}{\{P\}\ \texttt{while B do C}\ \{P \wedge \neg B\}}}{\{I\}\ \texttt{z := 1; w := 1; while B do C}\ \{P \wedge \neg B\}}$$

and the postcondition can be weakened:

$$\begin{aligned} P \wedge \neg B &\iff I \wedge R \wedge w = \exp(X_0, z - 1) \wedge \neg B \\ &\iff I \wedge w = \exp(X_0, z - 1) \wedge z = y + 1 \\ &\implies I \wedge w = \exp(X_0, y) \\ &\implies I \wedge w = \exp(X_0, Y_0) \end{aligned}$$

Dealing with Arrays

Arrays can be treated as families of variables indexed by integers.

Naive axiom:

$$\overline{\{Q[a_i \mapsto e]\}\, a_i := e\, \{Q\}}$$

What's wrong?

Dealing with Arrays

The solution is to substitute arrays monolithically

$$\frac{}{\{Q[a \mapsto a^{(i,e)}]\}\, a_i := e\, \{Q\}}$$

$$a_k^{(i,e)} = \begin{cases} a_k & \text{for } k \neq i \\ e & \text{for } k = i \end{cases}$$

Procedures / Functions

- Introduce functional component in the language (ALGOL-style)
- Allows for recursive definitions and an additional source of non-termination
- Two classes of identifiers: assignable variables and abstraction variables
- Quantifiers can be formalized with lambdas

Interference

```
f(x) {
 return  x+k;
}
```

$$\{a = f(b)\}\ k := k+1\ \{a = f(b)\}$$

Pointers

Classic problems...

$$\{*q = x\}\ *p := *\mathrm{p}{+}1\ \{*q = x\}$$

Total Correctness

The identification of a decreasing *variant* expression is necessary to gurantee that every loop terminates

$$\frac{[I \;\&\&\; B \;\&\&\; V == n]\; C\; [I \;\&\&\; V < n] \qquad\qquad I \;\&\&\; B \;\rightarrow\; V >= 0}{[I]\; \textbf{while}\,(B)\, \textbf{do}\; C\; [I \;\&\&\; \neg B]}$$

Realistic Languages

The problems that need to be addressed seem daunting, however:

-all have been studied at the theoretical level (beyond our scope)

-most importantly, tools exist that support full languages (including object-oriented features)

Exercise 1

```
void swap(int X[], int a, int b)
{ aux = X[a]; X[a] = X[b]; X[b] = aux; }
```

1. Write specification
2. Prove correctness of function

Exercise 2

Recall the *partition* function used by the quicksort algorithm. Verify informally:

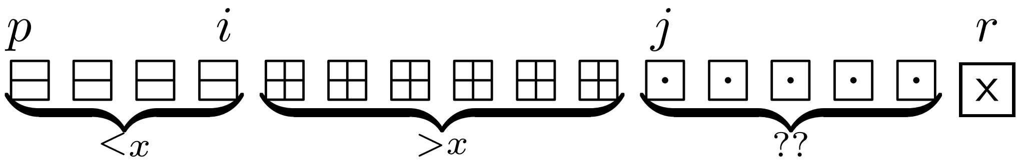
1. Write a Specification
2. Examine suggested implementation
3. Identify loop invariant
4. Check initial conditions and presevation
5. Identify loop variant
6. Check final conditions

```
int partition (int A[], int p, int r)
{
  x = A[r];
  i = p-1;
  for (j=p ; j<r ; j++)
    if (A[j] <= x) {
      i++;
      swap(A, i, j);
    }
  swap(A, i+1, r);
  return i+1;
}
```

Análise de Correcção – Invariante

No início de cada iteração do ciclo `for` tem-se para qualquer posição k do vector:

1. Se $p \leq k \leq i$ então $A[k] \leq x$;
2. Se $i + 1 \leq k \leq j - 1$ então $A[k] > x$;
3. Se $k = r$ então $A[k] = x$.



⇒ Verificar as propriedades de *inicialização* ($j = p$, $i = p - 1$), *preservação*, e *terminação* ($j = r$)

⇒ o que fazem as duas últimas instruções?

Algorithms slide

Something Missing!

- It is still required to check that the elements are the same in the input and in the output arrays!
- A particular case of the problem of specifying that two arrays contain the same elements
- *And* same number of occurences: *multiset equality*, rather than *set equality*

A first attempt

$$\forall k : p \leq k \leq r : (\, \exists l : p \leq l \leq r : A[k] = B[l] \wedge A[l] = B[k] \,)$$

What's wrong with it?

Second attempt

$$\forall k : p \leq k \leq r : (\, \exists l : p \leq l \leq r : A[k] = B[l] \,)$$

$$\wedge$$

$$\forall k : p \leq k \leq r : (\, \exists l : p \leq l \leq r : B[k] = A[l] \,)$$

What's wrong with it?

Third attempt

Use a logical theory for multi-sets and a function mset that abstracts an array into the multiset of its elements

$$\mathrm{mset}\,(A) = \mathrm{mset}\,(B)$$

This requires a prover with support for theories like *sets*, *multisets*, *sequences*… or else user-defined theories

Program Annotation and Automated Static Checking

Why Annotate Programs?

- A practical and accessible interface *specification* method
- Specify the semantics together with the syntax
- Do not worry about following a prescribed design method, as is the case with most formal methodologies
- “Light” formal methods for everyday programmers?

Applications

- Dynamic checking
- Test-case generation
- Static Checking
- Documentation: register design decisions and implementation steps
- Design by Contract

Design by Contract

- A software development method, initiated with Eiffel, based on contracts between clients and classes (dynamically-checked)
- Client guarantees certain (pre-)conditions before invoking methods and may then assume other (post-)conditions after invocation

Design by Contract

- Class must ensure certain (post-)conditions hold after methods have been called and may for this effect assume given (pre-)conditions
- Advantages: reasoning/modularity; *blame assignment*; eliminate *defensive checking* (practical and efficient!!!)

JML
(Java Modelling Language)

- A *standard* annotation language for JML
- Is itself very close to Java (easy to learn)
- Many tools have adhered to the standard and are now JML-compliant
- Imperative subset has been adapted to other languages (C)

JML Assertions

- *preconditions*: keyword `requires`
- *postconditions*: keyword `ensures`
- (class and loop) *invariants*:

  keywords `invariant`

  and `loop_invariant`

JML Assertions

- Added as special comments in Java files

```
/*@ ... @*/

//@ ...
```

- Properties written as Java boolean expressions
- With extra operators...

JML Operators

- Quantification:

```
(\forall ... ; ... ; ...)
(\exists ... ; ... ; ...)
```

- variable value at entry: `\old(...)`
- method return value: `\result`

Class Invariants

- Universal properties of class and instance variables (valid all the time)
- Must be preserved by all the methods in a class
- Implicitly, it is as if they were part of every pre- and postcondition

Other JML Stuff

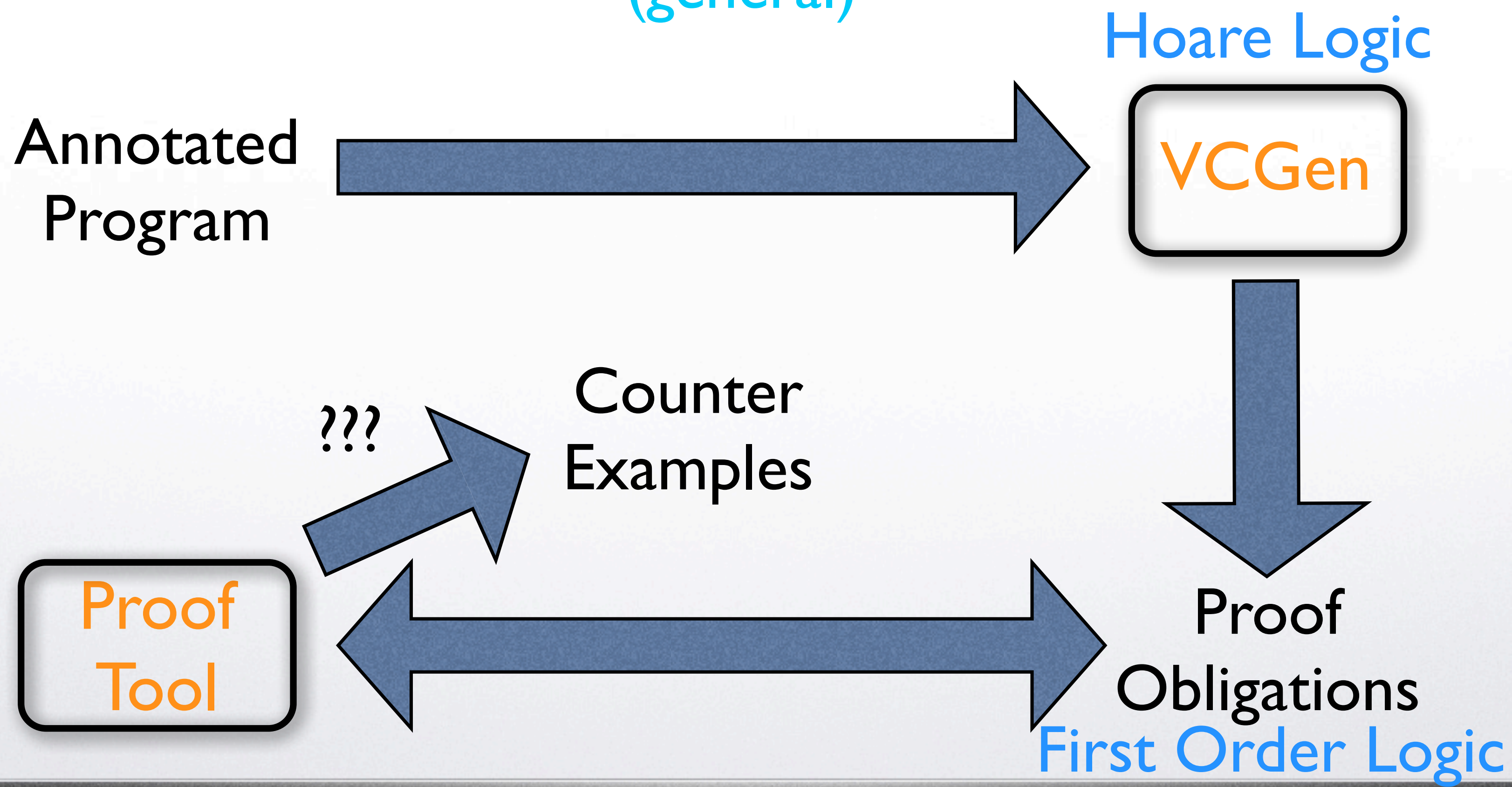
- exceptions (keyword `signals`)
- frame conditions
- pure methods: `pure`
- `non_null` annotations
- ad hoc assertions: `\assert`

Static Checking

- Dynamic checking verifies only the execution paths followed in one run of the program
- Static checking examines *all possible execution paths*
- The location of the warnings that are issued is not where they occur (as in run-time) but where they are *created*
- Typically *unsound* and *incomplete* to increase cost-effectiveness (automatic theorem prover, not interactive)

Underlying Architecture

(general)



ESC/Java and ESC/Java2

- Development Story: DEC / Compaq / HP research labs
- ESC/Java2: Kodak and UC Dublin researchers

  (update to cover *full* JML and recent versions of Java)
- JML-based; attempts to check consistency of code with annotations *automatically*
- Current versions use the **Simplify** theorem prover

ESC/Java and ESC/Java2

- Typical successful checks: null dereferencing; out-of-bounds array indexes (run-time exceptions). *Safety* checking
- Annotations may both suppress warnings (pre-condition prevents warning) and generate new warnings (pre-conditions may possibly not be met)

DEMO

ESC/Java2 eclipse plugin

swap / partition example

Limitations Highlighted by Partition Example!

$$\forall k : p \le k \le r : (\,\exists l : p \le l \le r : A[k] = B[l] \wedge A[l] = B[k]\,)$$

What's wrong with it?
Too Strong! However, ESC/Java proves this
(an example of unsoundness)

Second attempt

$$\forall k : p \leq k \leq r : (\, \exists l : p \leq l \leq r : A[k] = B[l] \,)$$
$$\wedge$$
$$\forall k : p \leq k \leq r : (\, \exists l : p \leq l \leq r : B[k] = A[l] \,)$$

What's wrong with it?
Too weak! However, ESC/Java fails to prove it
(an example of incompleteness)