Integrating verification in programming languages

Thomas Jensen, INRIA

Seminar
INRIA Rennes, 04/11/2015

Collège de France
Chaire Algorithmes, machines et langages

```
x / y
```

Types

For division to make sense, x and y should be some kind of numbers.

```
x : int;
y : int;

    … x / y …
```

P. Naur: Checking of operand types in Algol compilers (1965)

Pre-conditions

Avoid division-by-0 error.

```
require y ≠ 0;

        … x / y …
```

Pre- and post-conditions

Another possible requirement:

"x and y should be positive".

Allows us to say something about the result.

```
require x > 0, y > 0;

                x / y

ensures result > 0;
```

Compositionality

Specifications are essential for modular software development.

$$x \,/\, g(y)$$

"The post-condition of one operation may be another operation's pre-condition"

Why specify?

- Express what the code is supposed to do:
  - *a priori* requirements.
  - documentation.
- Check properties at run-time.
- Generate test cases.
- Prove formally that a program satisfies its specification.
  - *voir cours 2014-2015 : "Prouver les programmes : pourquoi, quand, comment".*

Languages with verification

In the object-oriented paradigm:

- Eiffel (1986): programming with contracts,
- JML : a specification language for Java,
- Spec#: a new version of C# with software contracts.

… and also:
Lustre, Esterel,
SparkADA, Scala,
Racket, Coq, Agda, Idris,…   An active area of research!

Today

- Overview of JML and Spec#
- Dynamic and static verification of specifications
- Specifying security and confidentiality.

The JML and Spec# approach

JML and Spec#

- Design a specification language for programmers
    - JML (Gary Leavens, Iowa State U.)
    - Spec# (Microsoft Redmond).
- Keep close to program syntax
    - specs as comments (JML) or language constructs (Spec#)
    - logic close to programmer intuition ("good enough")
- Programmer productivity as a key objective
    - bug finding is prime objective.

Pre- and post-conditions

Consider a simple bank account application with a **`debit`** method (here Java and JML)

```
public class Account
  private int balance;

/*@  requires  amount >= 0;
     ensures \result == balance;  @*/
public int debit (int amount) {
    …
}
```

Pre- and post-conditions

Pre- and post-conditions can be weakened.

```
public class Account
  private int balance;


/*@  requires  amount; >= 0
     ensures true;  @*/

public int debit (int amount) {
    …
}
```

Pre- and post-conditions

… and they can be strengthened:

```
public class Account
  private int balance;

/*@ requires  amount >= 0;
    ensures
    \result == balance &&
    balance == \old(balance) - amount;
@*/

public int debit (int amount) {
     …
}
```

Value of balance at entry to debit method

Invariants

Invariants are properties that must hold throughout the execution.

```
public class Account
  private int balance;


/*@  invariant 0 <= balance;  @*/
```

"Should never be negative"

Checking invariants dynamically may incur an important run-time overhead.

Assertions

Specify a property that should hold at one particular place in the program.

```
v = get_velocity();

//@ assert v <= SPEED_OF_LIGHT;
```

Cheaper to verify.

So useful that **`assert`** was added to most languages, including Java itself.

Goes back to von Neumann and Turing (late '40)

Quantifiers

Consider a class `Costumer` with several accounts and a table of amounts stored on each account.

```
public class Costumer {
  private int[] balances
  private Account[] accounts

/*@ invariant \forall i :
    0 <= i && i < balances.length ;
    balances[i] >= 0           @*/
```

Another example: sorting:

```
/*@ ensures \forall i :
    0 <= i && i < table.length-1;
    table[i] <= table[i+1]         @*/
```

The language of properties

Close to the programming language, but with a few essential add-ons:

- ✓ History variables: **`\old(var)`**
- ✓ Result variable **`\result`**
- ✓ Universal and existential quantification:
- Exceptions, pure methods and assignable variables, non-null types.

Specifying exceptions

```
public class Account
  private int balance;


 /*@  signals (AccountException e)
        amount > balance &&
        balance == \old(balance)…; @*/

public int debit (int amount) throws …{
    …
}
```

Only OK if enough money on account

Null pointers

"I couldn't resist the temptation to put in a **null reference**, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused **a billion dollars of pain and damage** in the last forty years."

C.A.R. Hoare, on the design of Algol W

Non-null types

Declare and check that a reference always points to something.

```
public class Costumer {
  private int [] balances
  private /*@ non-null @*/ Account[] accounts

  void deposit (/*@ non-null @*/ String id,
   int amount){
   …
   /*@ non-null @*/ Account find_account(String id)
   …
```

Very useful and easy to check locally.

Side effects

Limit the variables that can be modified in a method:

```
/*@  requires  amount; >= 0
     ensures … ;
     assignable balance @*/
public int debit (int amount) {
   …
}
```

Default: `assignable \everything`
Also useful: `assignable \nothing`

Spec#

True language integration:

- assignable variables are signaled by a `modifies` declaration in the method header.
- non-null references are declared by a `!`

```
class Costumer {
  Account[]! accounts

  void deposit modifies balance (String! id,
   int amount){ … }
```

Also addresses harder problems:

- limit side effects on internal sub-objects,
- private members must not appear in public signatures.

Pure methods

Methods that are `assignable \nothing` are called side-effect free or **pure.**

```
int /*@ pure @*/ getBalance();
int /*@ pure non-null @*/ findAccount(…)
```

Pure methods are the only methods that can be used in specifications.

```
/*@  invariant 0 <= getBalance(); @*/
```

Verification

How to verify

- Dynamic verification.
- Generating verification conditions.
    - Interactive theorem proving with programmer-specified invariants
    - See course on SMT.
- Static (automatic) program analysis.

Dynamic or static verification?

Dynamic evaluation of pre- and post-conditions and invariants:

- easy to implement
- run-time overhead (especially with invariants, history variables and recursion)
- late discovery of errors

Static checking of pre-, posts- and invariants:

- difficult program verification problem, often with approximations and false positives.
- no run-time overhead
- early detection of (some) errors

Issues with dynamic verification

Verifying **first-order** contracts is well understood - both in theory and in practice.

Contracts for **higher-order** functions pose questions: eg., how to check that a functional argument satisfies **`Even -> Even`**.

```
int M (Even -> Even f, int x) {
            … f(f(x))…
}
```

Who is to blame when M `(incr,3)` goes wrong?

Types and Blame

Type checking is one of the major success stories
of formal verification:

"Well-typed program do not go wrong"

For incremental software development it is important to mix
statically and dynamically verified code:

"Well-typed parts of code cannot be blamed"

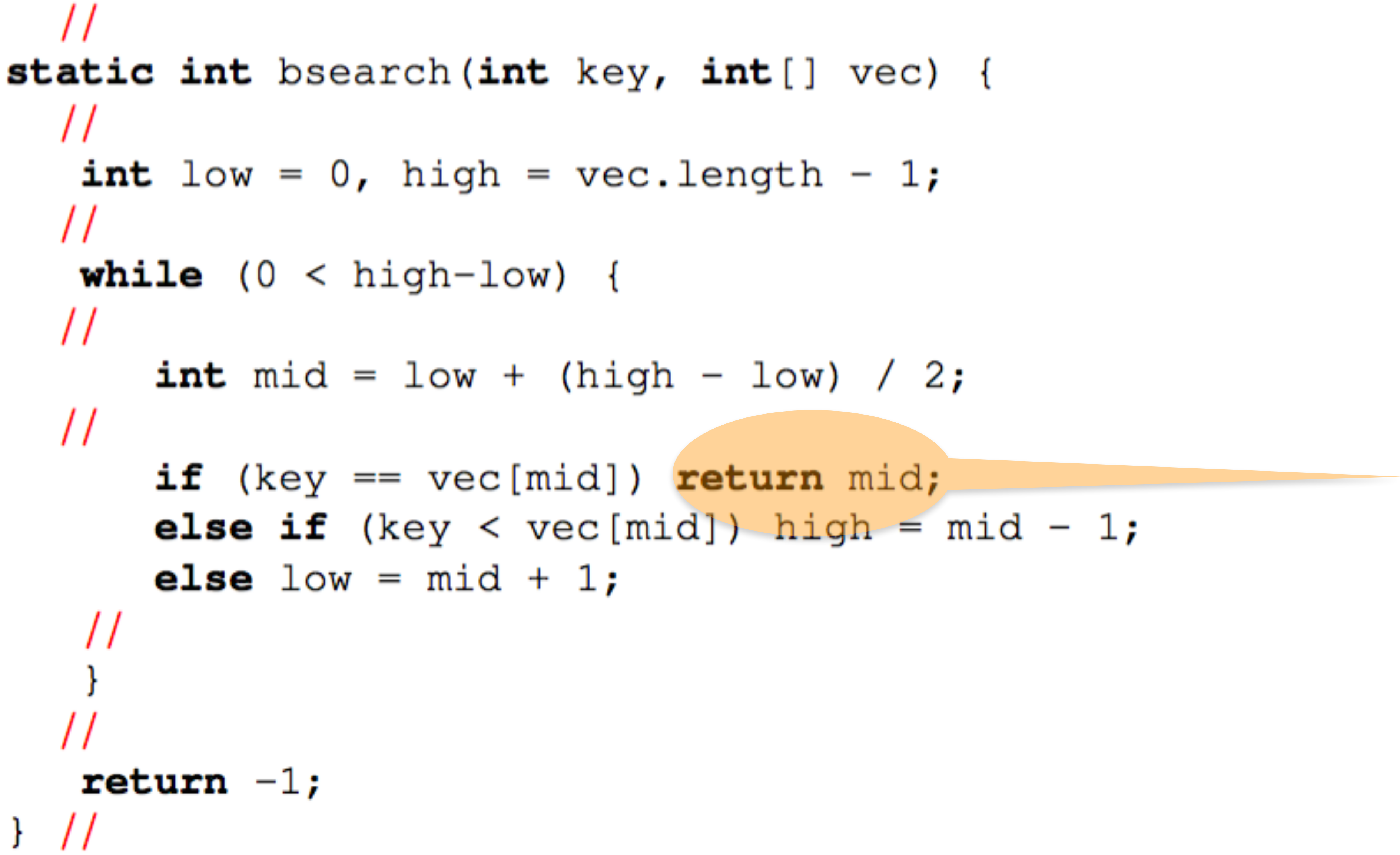
Static program analysis

Static program analysis

Infer properties about the behaviour of a program **without** running the program:

- Automatic.
- Correct.
- Approximate.

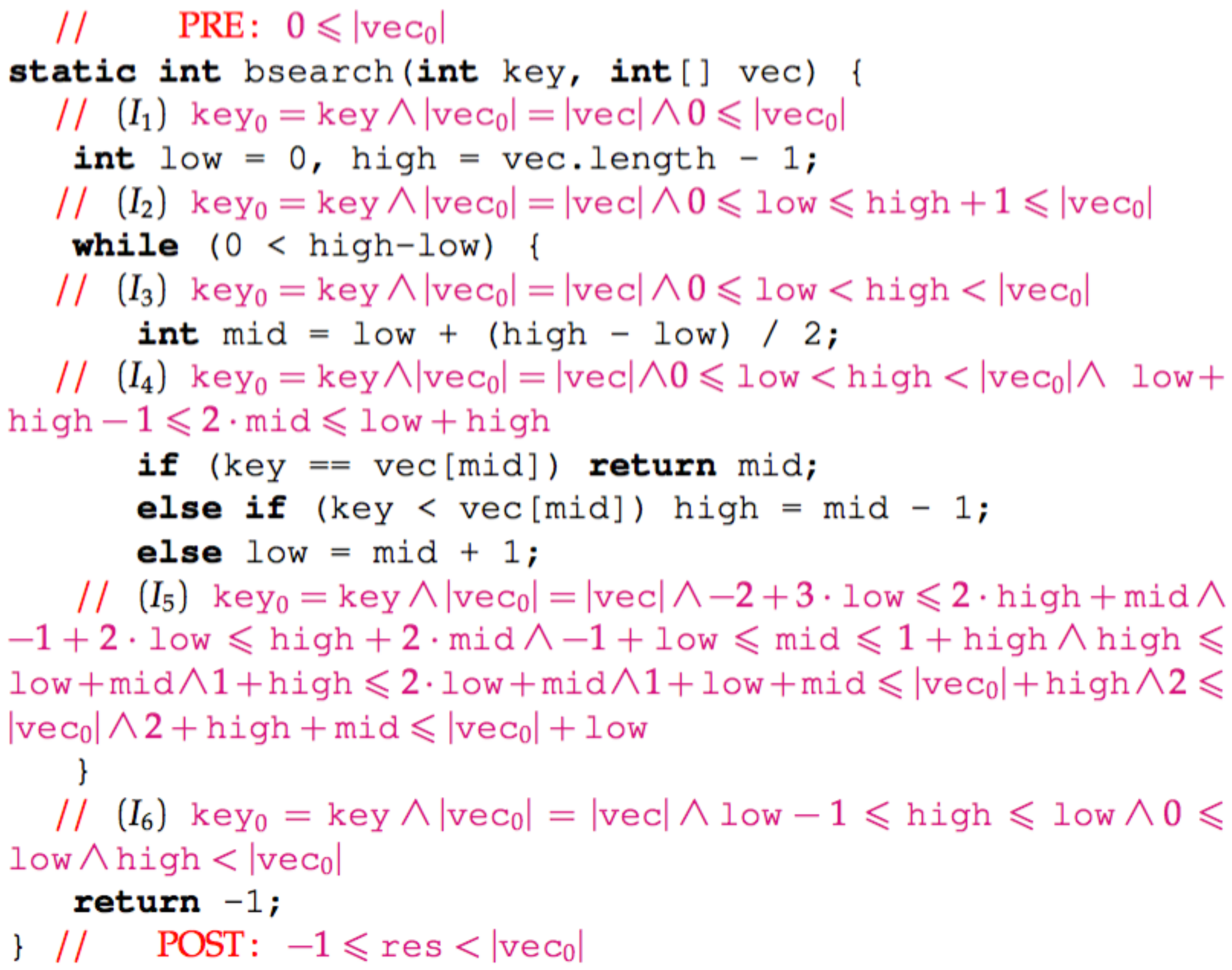
Verifying binary search



```
  //
static int bsearch(int key, int[] vec) {
  //
   int low = 0, high = vec.length - 1;
  //
   while (0 < high-low) {
  //
      int mid = low + (high - low) / 2;
  //
      if (key == vec[mid]) return mid;
      else if (key < vec[mid]) high = mid - 1;
      else low = mid + 1;
   //
   }
  //
   return -1;
} //
```

ensure: -1 ≤ \result < size_of(vec);

Verifying binary search



```
  //     PRE: $0 \leqslant |\mathtt{vec}_0|$
static int bsearch(int key, int[] vec) {
  // $(I_1)$ $\mathtt{key}_0 = \mathtt{key} \wedge |\mathtt{vec}_0| = |\mathtt{vec}| \wedge 0 \leqslant |\mathtt{vec}_0|$
   int low = 0, high = vec.length - 1;
  // $(I_2)$ $\mathtt{key}_0 = \mathtt{key} \wedge |\mathtt{vec}_0| = |\mathtt{vec}| \wedge 0 \leqslant \mathtt{low} \leqslant \mathtt{high} + 1 \leqslant |\mathtt{vec}_0|$
   while (0 < high-low) {
  // $(I_3)$ $\mathtt{key}_0 = \mathtt{key} \wedge |\mathtt{vec}_0| = |\mathtt{vec}| \wedge 0 \leqslant \mathtt{low} < \mathtt{high} < |\mathtt{vec}_0|$
      int mid = low + (high - low) / 2;
  // $(I_4)$ $\mathtt{key}_0 = \mathtt{key} \wedge |\mathtt{vec}_0| = |\mathtt{vec}| \wedge 0 \leqslant \mathtt{low} < \mathtt{high} < |\mathtt{vec}_0| \wedge$ $\mathtt{low} +$
$\mathtt{high} - 1 \leqslant 2 \cdot \mathtt{mid} \leqslant \mathtt{low} + \mathtt{high}$
      if (key == vec[mid]) return mid;
      else if (key < vec[mid]) high = mid - 1;
      else low = mid + 1;
    // $(I_5)$ $\mathtt{key}_0 = \mathtt{key} \wedge |\mathtt{vec}_0| = |\mathtt{vec}| \wedge -2 + 3 \cdot \mathtt{low} \leqslant 2 \cdot \mathtt{high} + \mathtt{mid} \wedge$
$-1 + 2 \cdot \mathtt{low} \leqslant \mathtt{high} + 2 \cdot \mathtt{mid} \wedge -1 + \mathtt{low} \leqslant \mathtt{mid} \leqslant 1 + \mathtt{high} \wedge \mathtt{high} \leqslant$
$\mathtt{low} + \mathtt{mid} \wedge 1 + \mathtt{high} \leqslant 2 \cdot \mathtt{low} + \mathtt{mid} \wedge 1 + \mathtt{low} + \mathtt{mid} \leqslant |\mathtt{vec}_0| + \mathtt{high} \wedge 2 \leqslant$
$|\mathtt{vec}_0| \wedge 2 + \mathtt{high} + \mathtt{mid} \leqslant |\mathtt{vec}_0| + \mathtt{low}$
   }
  // $(I_6)$ $\mathtt{key}_0 = \mathtt{key} \wedge |\mathtt{vec}_0| = |\mathtt{vec}| \wedge \mathtt{low} - 1 \leqslant \mathtt{high} \leqslant \mathtt{low} \wedge 0 \leqslant$
$\mathtt{low} \wedge \mathtt{high} < |\mathtt{vec}_0|$
   return -1;
} //    POST: $-1 \leqslant \mathtt{res} < |\mathtt{vec}_0|$
```

Abstract interpretation

A foundation for static program analysis:

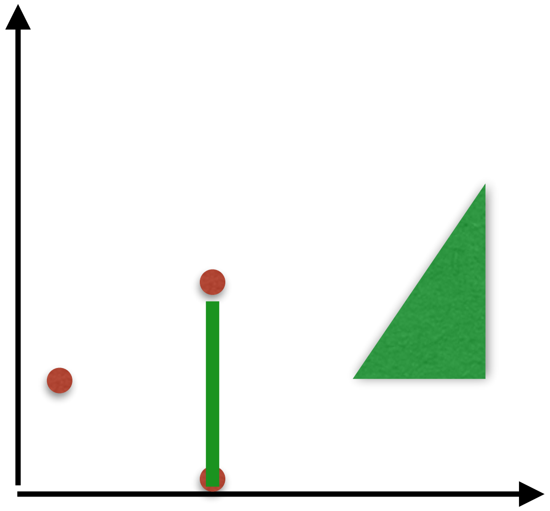
Interpret program over **abstract domain** of properties.

Eg, abstract integers by,

- signs (+,-,±)
- intervals ([1 ; 1] , [0 ; 3] , [1 ; ∞[,…)
- polyhedra

Polyhedral analysis

Describe program states by convex sets.



Represented by sets of linear inequalities:

$$0 \leq x \,,\quad x < 2y.$$

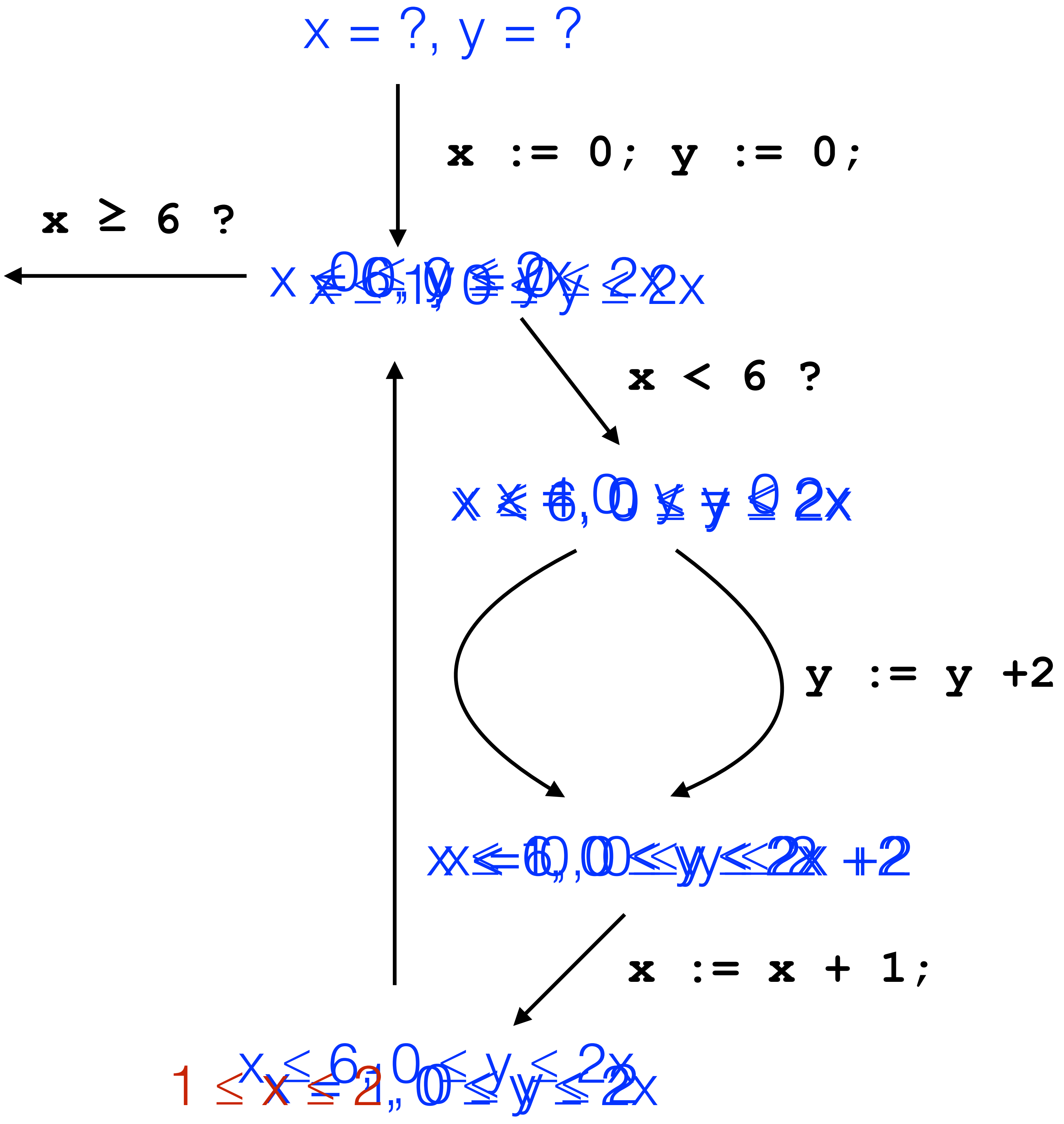
Example

```
x := 0; y := 0;
while (x < 6) {
  if (...) {
    y := y + 2;
  }
  x := x +1;
}

assert (y ≤ 12)
```

Polyhedral analysis

Stabilisation



x = ?, y = ?

`x := 0; y := 0;`

x ≤ 6, 0 ≤ y ≤ 2x

x = 0, y = 0

0 ≤ x ≤ 1, 0 ≤ y ≤ 2x

`x ≥ 6 ?`

x = 6, 0 ≤ y ≤ 12

`x < 6 ?`

x < 6, 0 ≤ y ≤ 2x

x = 0, y = 0

`y := y +2`

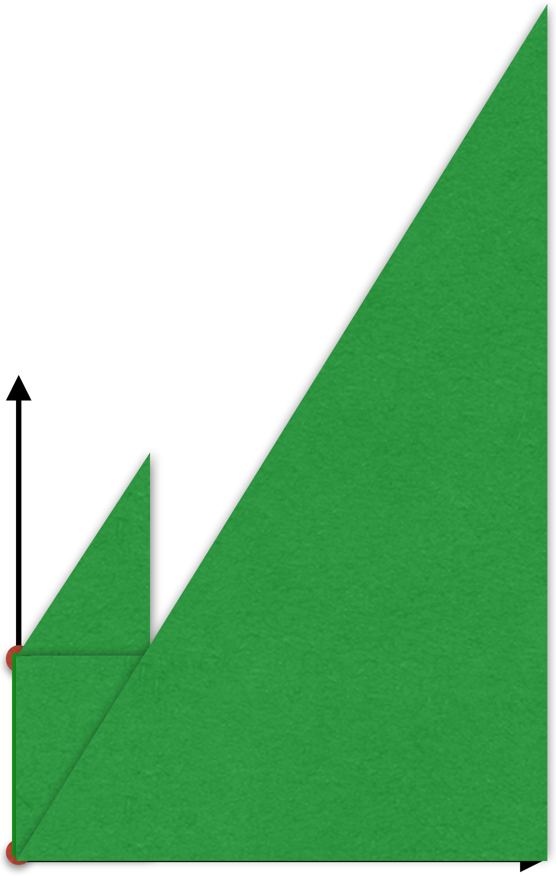
x ≤ 6, 0 ≤ y ≤ 2x +2

x = 0, 0 ≤ y ≤ 2

`x := x + 1;`

x ≤ 6, 0 ≤ y ≤ 2x

1 ≤ x ≤ 2, 0 ≤ y ≤ 2x



One analysis of many

- Numerical domains for integers, floats,…
- Alias, null-pointer and shape analysis of memory.

Principle of program analysis:

- translate to flow equations over partial orders,
- general solver based on iteration.

Specifying security

Information security

Three main properties of information security

- Confidentiality,
- Integrity of data,
- Availability.

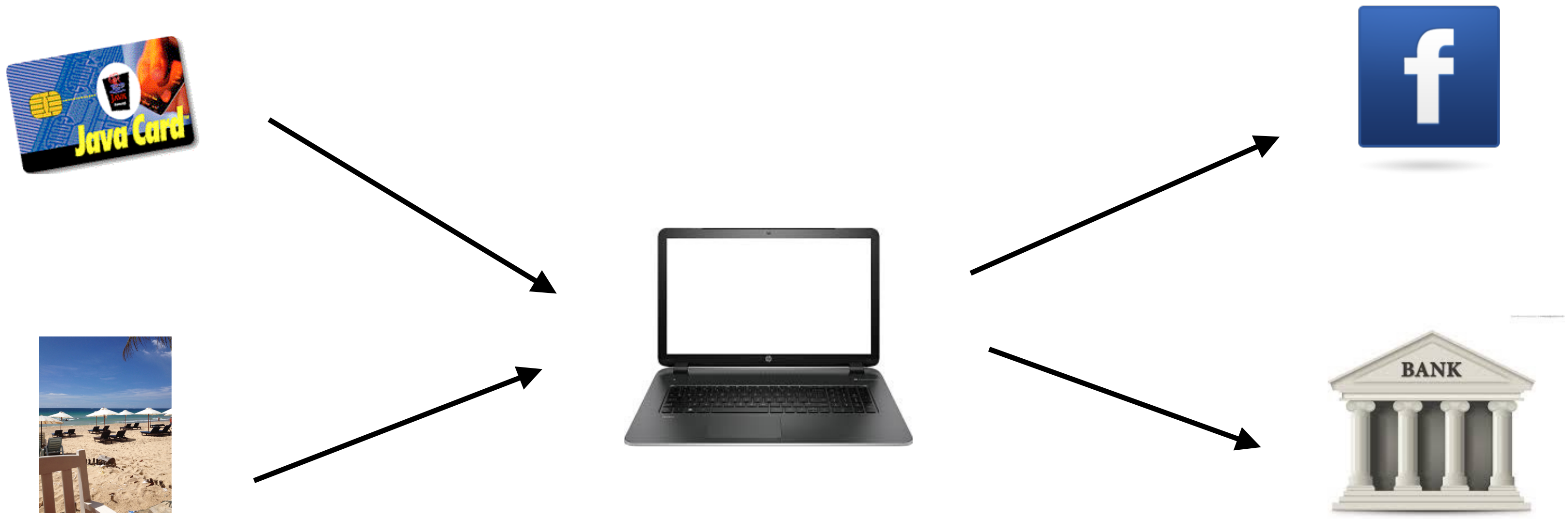
Most are **non-functional** properties

Confidentiality

Classify data as
- private/secret/confidential
- public

A basic security policy:

"Confidential data should not become public"



Breaking confidentiality

```
int secret s;    // s ∈ {0,1}
int public p;
```

```
p := s;             // direct flow
```

```
if s == 1 then
 p := 1
else             // indirect flow
 p := 0
```

Dynamic verification

Add a security level to all data and variables

Security levels evolve due to assignments

```
p := s;            // direct flow
```

and when we assign under secret control:

```
if s == 1 then
 p := 1
```

Secure?

Not enough to enforce confidentiality

```
int secret s;   // s ∈ {0,1}
int public p,q;
                  s=0      s=1
p := 0; q := 1;   p=0,q=1  p=0,q=1
if s == 0 then
  q := 0;         p=0,q=0  skip
if q == 1 then
  p := 1;         skip     p=1,q=1
                  p=0      p=1
```

The **"no-sensitive-upgrade"** principle

Static information flow control

Information flow types:

$$T, T_{\mathbf{x}}, T_{pc} \in \{\mathbf{public} \sqsubseteq \mathbf{secret}\}$$

Typing rules:

$$\frac{\vdash \mathbf{e} : T \qquad T \sqsubseteq T_{\mathbf{x}} \qquad T_{pc} \sqsubseteq T_{\mathbf{x}}}{T_{pc} \vdash \mathbf{x}\ \mathbf{:=}\ \mathbf{e}} \quad \textit{assign}$$

$$\frac{\vdash \mathbf{e} : T \qquad T_{pc} \sqcup T \vdash \mathbf{S_i} \qquad \mathbf{i} = \mathbf{1,2}}{T_{pc} \vdash \mathbf{if\ e\ then\ S_1\ else\ S_2}} \quad \textit{if}$$

"Real" information flow control

More elaborate policies would also specify how to **declassify** confidential data:

- what to declassify?
- when to declassify?

Proposals for information flow control for Java:

- JIF (Cornell)
- Paralocks (Chalmers)

Integrating verification in programming languages

- Specification and verification increasingly present
    - Robust code.
    - More productive programmers.
    - Both in academia and in industry.
- Functional and non-functional properties.
- Mix of dynamic and static verification.

Integrating verification in programming languages

Thomas Jensen, INRIA

Seminar
INRIA Rennes, 04/11/2015

Collège de France
Chaire Algorithmes, machines et langages