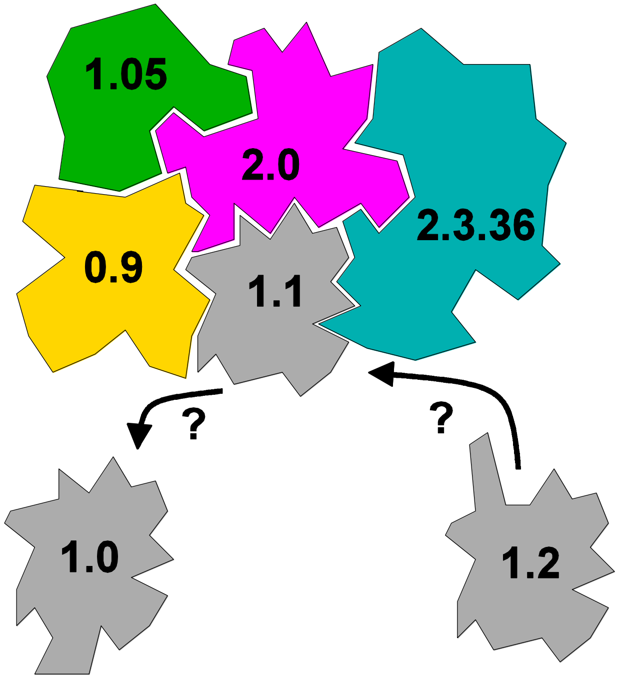
Predicting problems caused by component upgrades



Michael Ernst

MIT Lab for Computer Science

http://pag.lcs.mit.edu/~mernst/

Joint work with Stephen McCamant

An upgrade problem

1. You are a happy user of Stucco Photostand
2. You install Microswat Monopoly
3. Photostand stops working

Why?

- Step 2 upgraded **`winulose.dll`**
- Photostand is not compatible with the new version

Outline

The upgrade problem

Solution: Compare observed behavior

Capturing observed behavior

Comparing observed behavior (details)

Example: Sorting and **`swap`**

Case study: Currency

Conclusion

Upgrade safety

System *S* uses component *C*

A new version *C'* is released

Might *C'* cause *S* to misbehave?

(This question is undecidable.)

Previous solutions

Integrate new component, then test

- Resource-intensive

Vendor tests new component

- Impossible to anticipate all uses
- User, not vendor, must make upgrade decision
- (We require this)

Static analysis to guarantee identical or subtype behavior

- Difficult and inadequate

Behavioral subtyping

Subtyping guarantees type compatibility

- No information about behavior

Behavioral subtyping [Liskov 94] guarantees behavioral compatibility

- Provable properties of supertype are provable about subtype
- Operates on human-supplied specifications
- Ill-matched to the component upgrade problem

Behavioral subtyping is too strong and too weak

Too strong:

- OK to change APIs that the application does not call
- … or other aspects of APIs that are not depended upon

Too weak:

- Application may depend on implementation details
- Example:
  - Component version 1 returns elements in order
  - Application depends on that detail
  - Component version 2 returns elements in a different order
- Who is at fault in this example? It doesn’t matter!

Outline

The upgrade problem

⇒ Solution: Compare observed behavior

Capturing observed behavior

Comparing observed behavior (details)

Example: Sorting and **`swap`**

Case study: Currency

Conclusion

Features of our solution

- Application-specific
- Can warn before integrating, testing
- Minimal disruption to the development process
- Requires no source code
- Requires no formal specification
- Warns regardless of who is at fault
- Accounts for internal and external behaviors

Caveat emptor: no *guarantee* of (in)compatibility!

Run-time behavior comparison

Compare run-time behaviors of components

- Old component, in context of the application
- New component, in context of vendor test suite

Compatible if the vendor tests all the functionality that the application uses

Consider comparing test suites

- “Behavioral subtesting”

Reasons for behavioral differences

Differences between application and test suite use of component require human judgment

- True incompatibility
- Change in behavior might not affect application
- Change in behavior might be a bug fix
- Vendor test suite might be deficient
- It may be possible to work around the incompatibility

Operational abstraction

Abstraction of run-time behavior of component

Set of program properties – mathematical statements about component behavior

Syntactically identical to formal specification

Outline

The upgrade problem

Solution: Compare observed behavior

⇒ Capturing observed behavior

Comparing observed behavior (details)

Example: Sorting and **`swap`**

Case study: Currency

Conclusion

Dynamic invariant detection

Goal: recover invariants from programs

Technique: run the program, examine values

Artifact: Daikon

`http://pag.lcs.mit.edu/daikon`

Experiments demonstrate accuracy, usefulness

Goal: recover invariants

Detect invariants (as in **`assert`**s or specifications)

- **`x > abs(y)`**
- **`x = 16*y + 4*z + 3`**
- array **`a`** contains no duplicates
- for each node **`n`**, **`n = n.child.parent`**
- graph **`g`** is acyclic
- if **`ptr ≠ null`** then **`*ptr > i`**

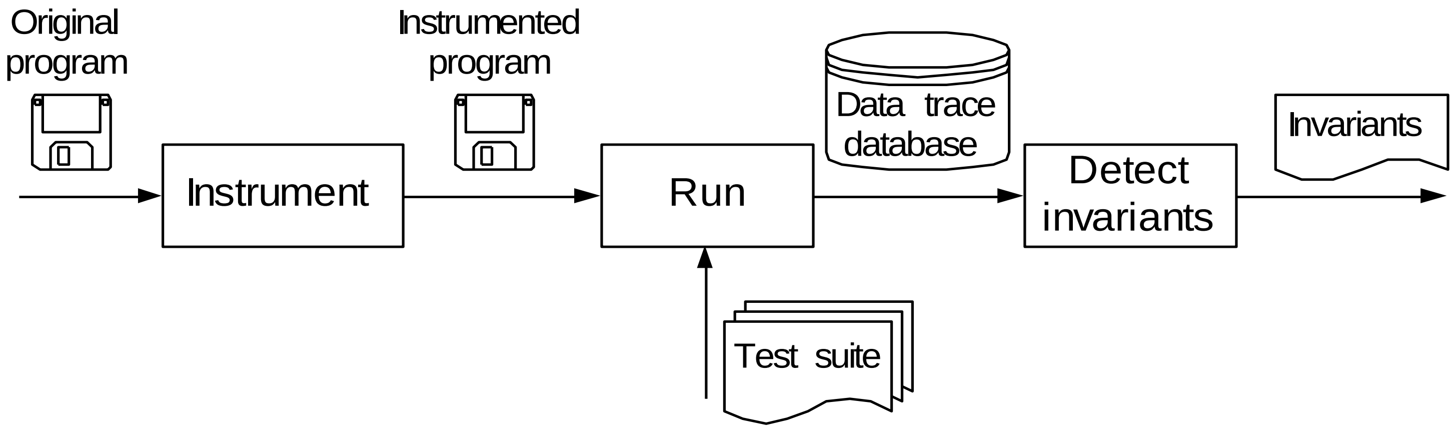
Uses for invariants

- Write better programs [Gries 81, Liskov 86]
- Document code
- Check assumptions: convert to `assert`
- Maintain invariants to avoid introducing bugs
- Locate unusual conditions
- Validate test suite: value coverage
- Provide hints for higher-level profile-directed compilation [Calder 98]
- Bootstrap proofs [Wegbreit 74, Bensalem 96]

Ways to obtain invariants

- Programmer-supplied
- Static analysis: examine the program text [Cousot 77, Gannod 96]
  - properties are guaranteed to be true
  - pointers are intractable in practice
- Dynamic analysis: run the program
  - complementary to static techniques

Dynamic invariant detection



Look for patterns in values the program computes:

- Instrument the program to write data trace files
- Run the program on a test suite
- Invariant engine reads data traces, generates potential invariants, and checks them

Checking invariants

For each potential invariant:

- instantiate
  (determine constants like a and b in $y = ax + b$)
- check for each set of variable values
- stop checking when falsified

This is inexpensive: many invariants, each cheap

Improving invariant detection

Add desired invariants: implicit values, unused polymorphism

Eliminate undesired invariants: unjustified properties, redundant invariants, incomparable variables

Traverse recursive data structures

Conditionals: compute invariants over subsets of data (if $x>0$ then $y \neq z$)

Outline

The upgrade problem

Solution: Compare observed behavior

Capturing observed behavior

⇒ Comparing observed behavior (details)

Example: Sorting and **`swap`**

Case study: Currency

Conclusion

Testing upgrade compatibility

1. User computes operational abstraction of old component, in context of application
2. Vendor computes operational abstraction of new component, over its test suite
3. Vendor supplies operational abstraction along with new component
4. User compares operational abstractions
   - OA_{app} for old component
   - OA_{test} for new component

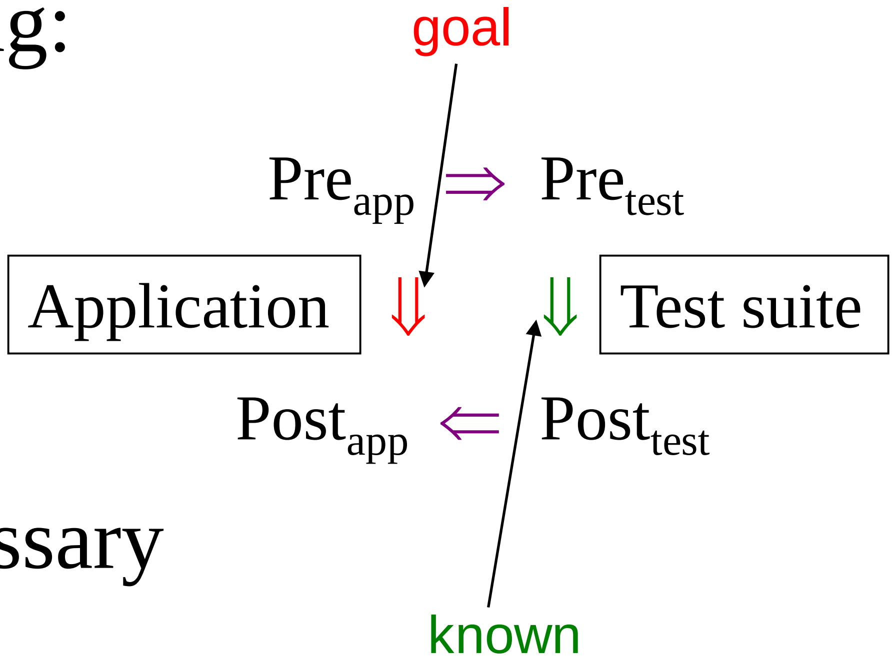
New operational abstraction must be stronger

Approximate test: $OA_{test} \Rightarrow OA_{app}$

OA consists of precondition and postcondition

Per behavioral subtyping:

- $Pre_{app} \Rightarrow Pre_{test}$
  $Post_{test} \Rightarrow Post_{app}$



goal

$Pre_{app} \Rightarrow Pre_{test}$

Application $\Downarrow$ $\Downarrow$ Test suite

$Post_{app} \Leftarrow Post_{test}$

known

Sufficient, but not necessary

Comparing operational abstractions

Sufficient but not necessary:

$\mathrm{Pre}_{\mathrm{app}} \Rightarrow \mathrm{Pre}_{\mathrm{test}}$

$\mathrm{Post}_{\mathrm{test}} \Rightarrow \mathrm{Post}_{\mathrm{app}}$

$$\begin{tikzcd}
\text{x is even} \arrow[r, Rightarrow] \arrow[d, Rightarrow, "\boxed{\text{Application}}"'] & \text{x is an integer} \arrow[d, Rightarrow, "\boxed{\begin{array}{l}\texttt{increment} \\ \text{test suite}\end{array}}"] \\
\begin{array}{l} x' = x + 1 \\ \text{x' is odd} \end{array} & x' = x + 1 \arrow[l, Rightarrow]
\end{tikzcd}$$

Sufficient and necessary:

$\mathrm{Pre}_{\mathrm{app}} \Rightarrow \mathrm{Pre}_{\mathrm{test}}$

$\mathrm{Pre}_{\mathrm{app}} \;\&\; \mathrm{Post}_{\mathrm{test}} \Rightarrow \mathrm{Post}_{\mathrm{app}}$

Outline

The upgrade problem

Solution: Compare observed behavior

Capturing observed behavior

Comparing observed behavior (details)

⇒ Example: Sorting and **`swap`**

Case study: Currency

Conclusion

Sorting application

```
// Sort the argument into ascending order
static void
bubble_sort(int[] a) {
  for (int x = a.length - 1; x > 0; x--) {
    // Compare adjacent elements in a[0..x]
    for (int y = 0; y < x; y++) {
      if (a[y] > a[y+1])
        swap(a, y, y+1);
    }
  }
}
```

Swap component

```
// Exchange the two array elements at i and j
static void
swap(int[] a, int i, int j) {
  int temp = a[i];
  a[i] = a[j];
  a[j] = temp;
}
```

Upgrade to swap component

```
// Exchange the two array elements at i and j
static void
swap(int[] a, int i, int j) {
  a[i] ^= a[j];
  a[j] ^= a[i];
  a[i] ^= a[j];
}
```

Compare abstractions

```
a != null
0 <= i < size(a[])-1
1 <= j <= size(a[])-1
i < j
j == i + 1
a[i] == a[j-1]
a[i] > a[j]
```

⇒

```
a != null
0 <= i <= size(a[])-1
0 <= j <= size(a[])-1
i != j
```

`bubble_sort` application ⇓

⇓ **`swap`** test suite

```
a'[i] == a[j]
a'[j] == a[i]
a'[i] == a'[j-1]
a'[j] == a[j-1]
a'[i] < a'[j]
```

⇐

```
a'[i] == a[j]
a'[j] == a[i]
```

Compare abstractions

`a != null`
`0 <= i < size(a[])-1`
`1 <= j <= size(a[])-1`
`i < j`
`j == i + 1`
`a[i] == a[j-1]`
`a[i] > a[j]`

⇒

`a != null`
`0 <= i <= size(a[])-1`
`0 <= j <= size(a[])-1`
`i != j`

`bubble_sort`
application

⇓

⇓

`swap` test suite

`a’[i] == a[j]`
`a’[j] == a[i]`
`a’[i] == a’[j-1]`
`a’[j] == a[j-1]`
`a’[i] < a’[j]`

⇐

`a’[i] == a[j]`
`a’[j] == a[i]`

$$\mathrm{Pre}_{\mathrm{app}} \Rightarrow \mathrm{Pre}_{\mathrm{test}}$$

Compare abstractions

`a != null`
`0 <= i < size(a[])-1`
`1 <= j <= size(a[])-1`
`i < j`
`j == i + 1`
`a[i] == a[j-1]`
`a[i] > a[j]`

⇒

`a != null`
`0 <= i <= size(a[])-1`
`0 <= j <= size(a[])-1`
`i != j`

bubble_sort application ⇓

⇓ **swap** test suite

`a’[i] == a[j]`
`a’[j] == a[i]`
`a’[i] == a’[j-1]`
`a’[j] == a[j-1]`
`a’[i] < a’[j]`

⇐

`a’[i] == a[j]`
`a’[j] == a[i]`

$$\mathrm{Pre}_{\mathrm{app}} \;\&\; \mathrm{Post}_{\mathrm{test}} \Rightarrow \mathrm{Post}_{\mathrm{app}}$$

Compare abstractions

```
a != null
0 <= i < size(a[])-1
1 <= j <= size(a[])-1
i < j
j == i + 1
a[i] == a[j-1]
a[i] > a[j]
```

⇒

```
a != null
0 <= i <= size(a[])-1
0 <= j <= size(a[])-1
i != j
```

bubble_sort application ⇓

⇓ **swap** test suite

```
a’[i] == a[j]
a’[j] == a[i]
a’[i] == a’[j-1]
a’[j] == a[j-1]
a’[i] < a’[j]
```

⇐

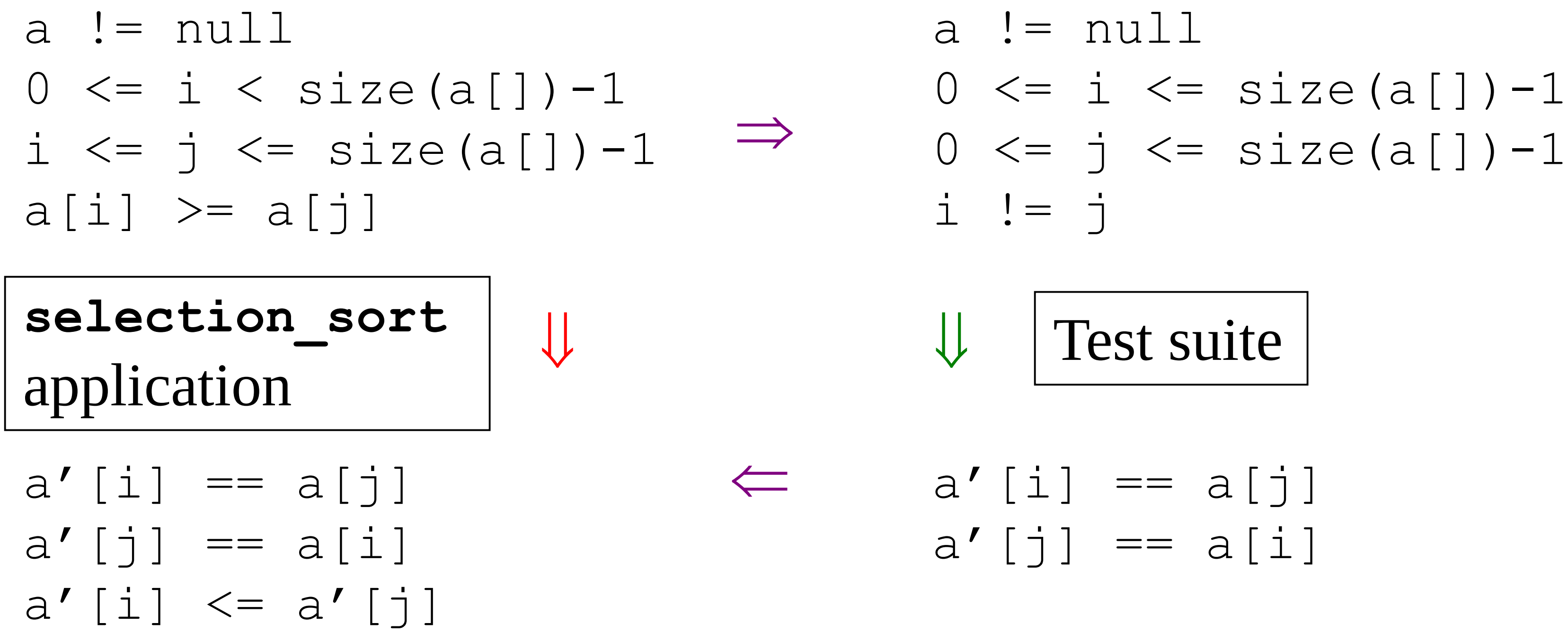
```
a’[i] == a[j]
a’[j] == a[i]
```

Upgrade succeeds

Another sorting application

```
// Sort the argument into ascending order
static void
selection_sort(int[] a) {
  for (int x = 0; x <= a.length - 2; x++) {
    // Find the smallest element in a[x..]
    int min = x;
    for (int y = x; y < a.length; y++) {
      if (a[y] < a[min])
        min = y;
    }
    swap(a, x, min);
  }
}
```

Compare abstractions



```
a != null
0 <= i < size(a[])-1
i <= j <= size(a[])-1
a[i] >= a[j]
```

⇒

```
a != null
0 <= i <= size(a[])-1
0 <= j <= size(a[])-1
i != j
```

selection_sort application ⇓

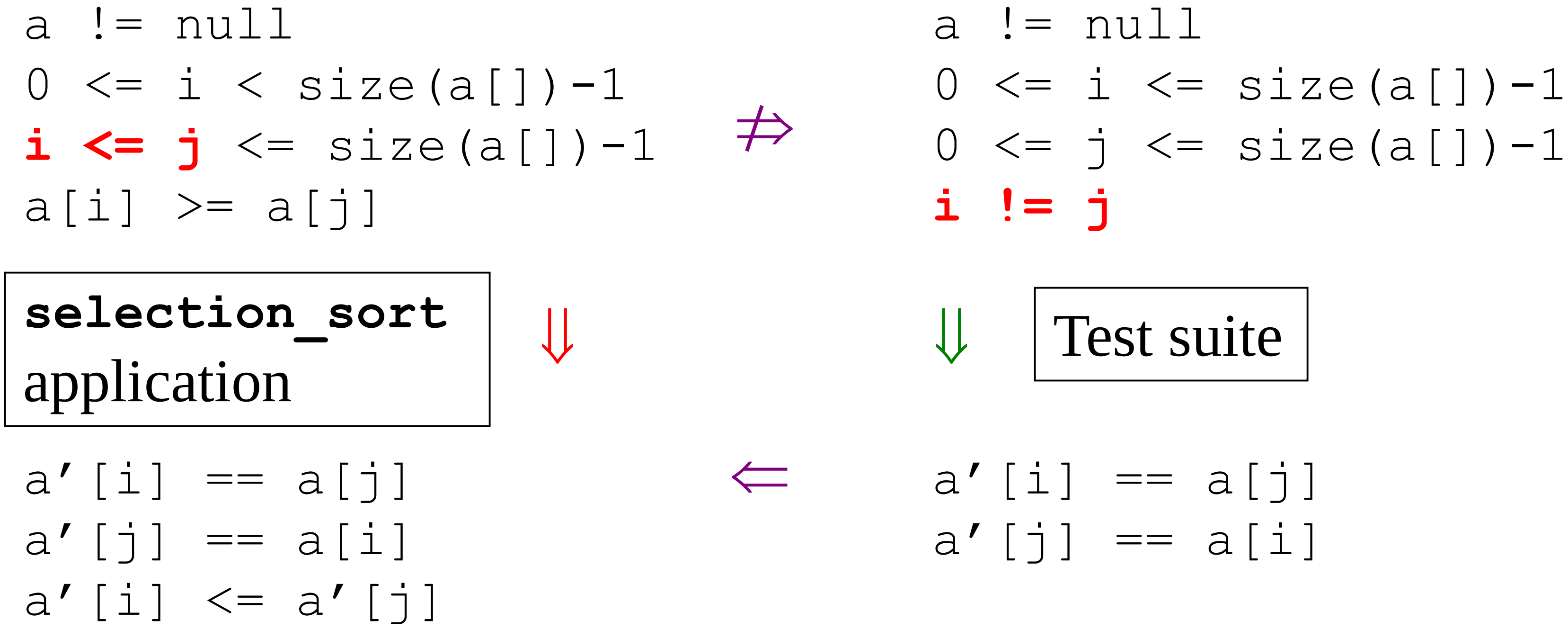
⇓ Test suite

```
a'[i] == a[j]
a'[j] == a[i]
a'[i] <= a'[j]
```

⇐

```
a'[i] == a[j]
a'[j] == a[i]
```

Compare abstractions



`a != null`
`0 <= i < size(a[])-1`
`i <= j` `<= size(a[])-1`
`a[i] >= a[j]`

⇏

`a != null`
`0 <= i <= size(a[])-1`
`0 <= j <= size(a[])-1`
`i != j`

`selection_sort`
application

⇓

⇓

Test suite

`a'[i] == a[j]`
`a'[j] == a[i]`
`a'[i] <= a'[j]`

⇐

`a'[i] == a[j]`
`a'[j] == a[i]`

Upgrade fails

Outline

The upgrade problem

Solution: Compare observed behavior

Capturing observed behavior

Comparing observed behavior (details)

Example: Sorting and **`swap`**

⇒ Case study: Currency

Conclusion

Currency case study

Application: Math-Currency

Component: Math-BigInt (versions 1.40, 1.42)

Both from Comprehensive Perl Archive Network

Our technique is needed: a wrong version of BigInt induces two errors in Currency

Downgrade from BigInt 1.42 to 1.40

(Why downgrade? Fix bugs, porting.)

Inconsistency is discovered:

- In 1.42, **`bcmp`** returns -1, 0, or 1
- In 1.40, **`bcmp`** returns any integer

Do not downgrade without further examination

- Application might do **`(a <=> b) == (c <=> d)`**

(This change is not reflected in the documentation.)

Upgrade from BigInt 1.40 to 1.42

Inconsistency:

- In 1.40, `bcmp($1.67, $1.75)` ⇒ 0
- In 1.42, `bcmp($1.67, $1.75)` ⇒ -1

Our system did not discover this property …

… but it discovered differences in behavior of other components that interacted with it

Do not upgrade without further examination

Outline

The upgrade problem

Solution: Compare observed behavior

Capturing observed behavior

Comparing observed behavior (details)

Example: Sorting and **`swap`**

Case study: Currency

⇒ Conclusion

Getting to Yes: Limits of the technique

Rejecting an upgrade is easier than approving it

- Application postconditions may be hard to prove
- Can check the reason for the rejection

Key problem is limits of the theorem prover

Adjust grammar of operational abstractions

- Stronger or weaker properties may be provable
- Weak properties may depend on strong ones

Implementation status

- Operational abstractions are automatically generated (by the Daikon invariant detector)
- In Currency case study, operational abstractions were compared by hand
- Operational abstractions are automatically compared (by the Simplify theorem prover)
  - Requires background theory for each property

Contributions

New technique for early detection of upgrade problems

Compares run-time behavior of old & new components

Technique is

- Application-specific
- Pre-integration
- Lightweight
- Source-free
- Specification-free
- Blame-neutral
- Output-independent
- Unvalidated

Questions?