Finding inputs that REACH a target expression

Matthew Naylor

(joint work with Colin Runciman)

The problem that REACH solves

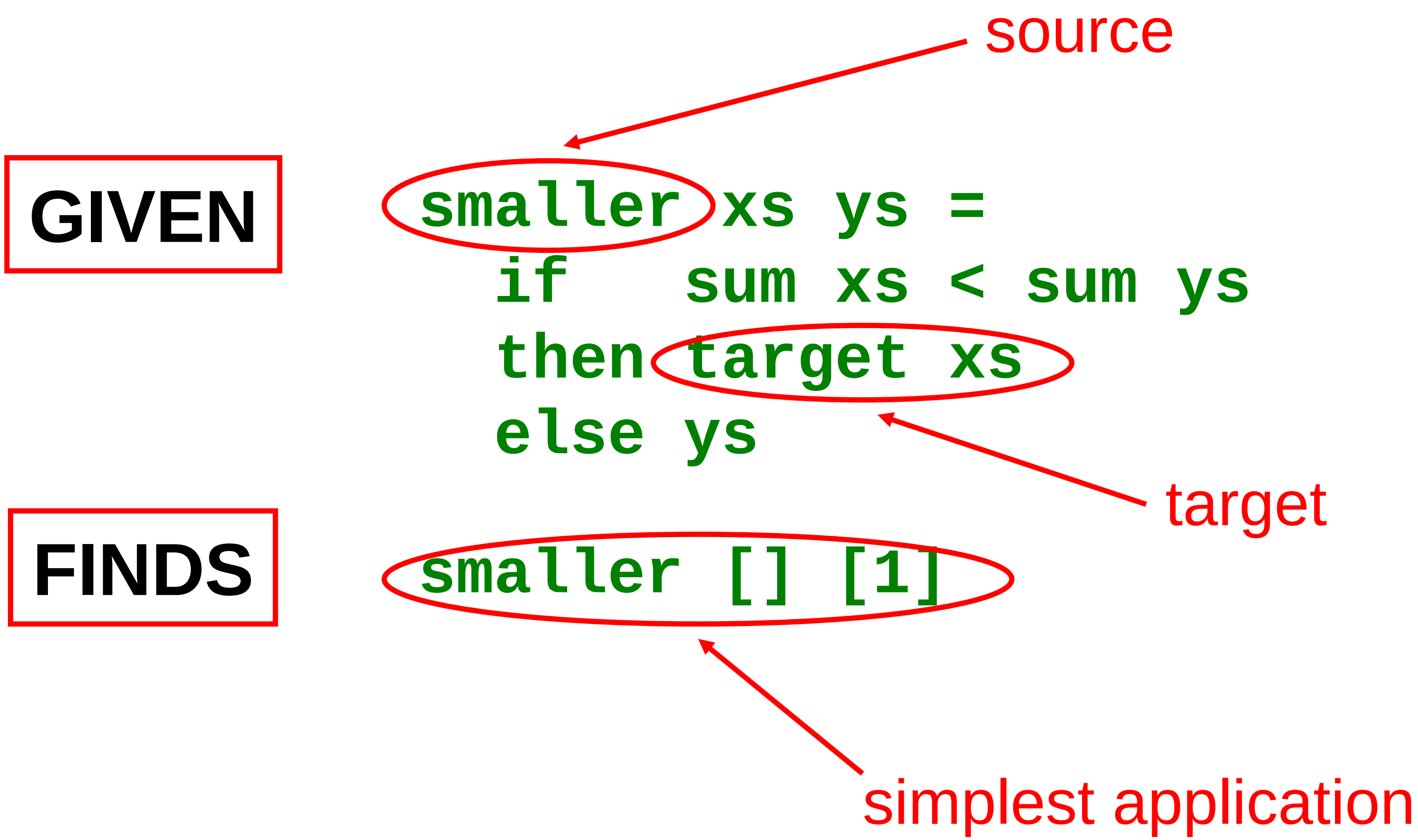
GIVEN

a program with some top-level functions marked as **sources** and some expressions marked as **targets**

FIND

for each source the **simplest applications** that entail evaluation of one or more targets

Simple example of REACH



source

GIVEN

```
smaller xs ys =
  if   sum xs < sum ys
  then target xs
  else ys
```

target

FINDS

```
smaller [] [1]
```

simplest application

Application 1: Refuting Properties

- Properties with conditions often arise when testing – e.g. deletion from a binary search tree should satisfy

```
prop_ordDel x t =
      ordered t ==> ordered (delete x t)
```

condition

- Such properties can be very difficult to test
  - Condition must be satisfied if a counter example is to be found
  - Many inputs do not satisfy the condition
  - Blind test generators cannot see the condition

Refuting Conditional Properties with REACH

- Define implication as

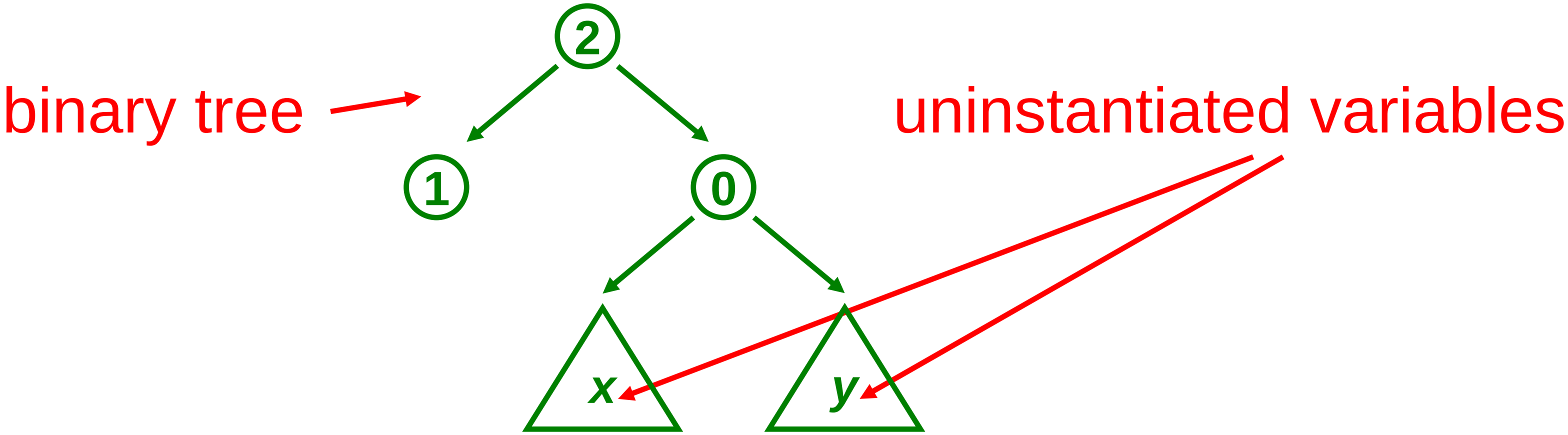
```
cond ==> x  =  if   cond && not x
               then target False
               else True
```

- Time taken to find all counter-examples with a depth bound of four:
  - Reach: 272 seconds
  - Exhaustive testing: 2523 seconds

- The more restrictive the condition, the bigger the benefit of using REACH over exhaustive testing

What is REACH doing?

- Sources are evaluated with uninstantiated inputs
  - Inputs are progressively instantiated as evaluation proceeds



binary tree

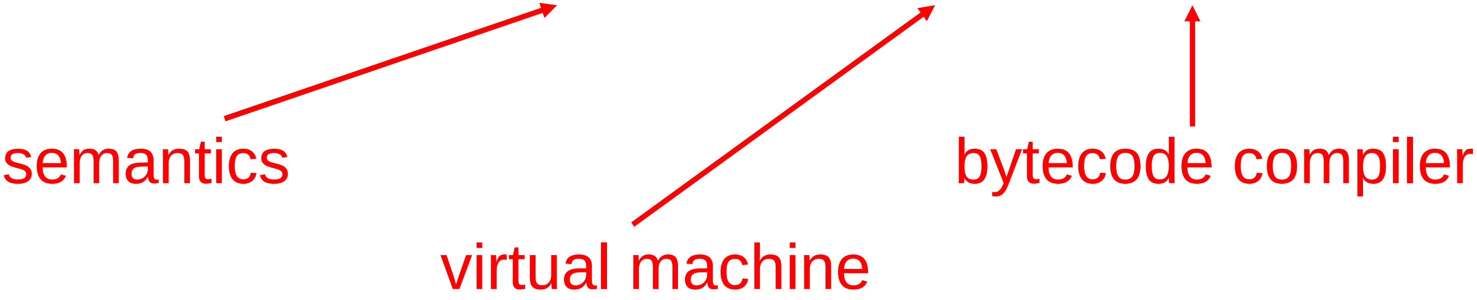
uninstantiated variables

- This partial input tree represents *many* concrete trees
- But it is not ordered, no matter what ***x*** and ***y*** are
- So a whole set of trees is pruned from the search space

Application 2: Property Coverage

- Congruence properties are also common – e.g. a compiler for an imperative language should satisfy

```
prop_comp p = interp p == exec (comp p)
```



semantics

virtual machine

bytecode compiler

- Pruning is only possible when **`interp p`** and **`exec (comp p)`** return for partially instantiated **`p`**
- But **`comp`** requires whole program before returning
  - No search space pruned; smaller benefit in refuting using REACH

- But REACH can help obtain coverage

- For example, a coverage tool reported this expression as unreached after a random testing

```
run (While c p) env =
  case eval c env of
    True  -> …
    False -> …
```

good place to insert `target`

- Random testing did not check the property for programs containing while loops that iterate at least once

Other Applications

- Crashing Programs – put **`target`** before calls to **`error`**

```
head [] = target (error “no head of []”)
head (x:xs) = x
```

program crashes on error

- Assertion Breaking – redefine **`assert`** as:

```
assert c x = if c then x else target x
```

- Program Understanding – e.g. find the simplest binary tree that involves the most complex rebalancing rotations

Restrictions

- Programs may contain higher-order functions but only first-order functions can be REACH sources
- Requires algebraic data types; primitive numeric types are redefined accordingly – e.g. integers as signed unary

  ```
  data Nat = Zero | Succ Nat
  ```

- Search is bounded by two parameters
  - Maximum construction depth of source arguments
    - Know what class of inputs have been tested (bounded verification)
  - Maximum recursion depth of function calls
    - Always terminates

Conclusion and Future Work

- REACH is a simple analysis with many useful applications
  - Has a concise, clearly correct, operational semantics
  - Applications: property and assertion refutation, program coverage, crashing, and understanding
  - Demonstrated on several published programs (some multi-module, several hundred lines long)
  - Up to 2 orders of magnitude speed-up observed over exhaustive testing when refuting a range of properties at small depths

- Sometimes REACH explores computational branches that can be avoided if function-call-graph is known
  - Semantics extended with 3 pruning rules
  - Order of magnitude speed-up observed on some examples