GAST: a Generic Automated Software Test-system

Pieter Koopman, Artem Alimarine, Jan Tretmans, Rinus Plasmeijer
Nijmegen, NL

Overview

- Why automatic testing
- What will be tested: Properties of (functional) programs
- How do we want to test: User specifies property of software
- Test system does all the rest
- Research question: Can we make such a system
- Conclusions / Related & future work

Why automatic testing

Manual testing is:
- Much work (up to 50% of project cost)
- Dull
- Error-prone
- To be repeated after changes

An automatic system:
- Makes testing easier, faster and more reliable
- Encourages writing specifications
- Relates specification and software
- Easy to repeat tests after changes (regression)

What will be tested

GAST tests software:
- In principle software in any language
- In the functional programming language Clean
- We restrict ourselves to: Functional testing obeys the software the specification?
- Today we consider only the technical aspects of testing

How do we test

Testing consists of four steps:
1. Definition of properties to be obeyed
2. Generation of test data
3. Execute tests with these data
4. Analyze result of these tests

A human should define the properties:
- Before or after writing the software
- GAST does 2, 3, and 4 fully automatically
- On demand of the tester
- As often and thorough as desired
- Can be used to certify software

Definition of properties

- Express properties by functions in Clean
- Clean is a functional programming language
- Most striking syntax difference for this talk:
  - Write `f :: Int Int -> Int
    f x y = g y x` instead of
  - `int f (x::int, y::int)
    { return g(y,x); }

Properties

- Topic ranges from one function to full program
- Property of software can be:
  - Equivalence to reference implementation
  - Relation between type manipulation functions
  - Relation between input and output
  - Power of first order predicate logic
- We will show some examples of properties

Property: Reference implementation

- Software to be tested: new or function
  - `or :: Bool Bool -> Bool
    or x y = nand (not x) (not y) where not x = nand x x`
- Reference implementation: or-operator `||`
- Property: result of `or` and `||` should be equal
- `propOr :: Bool Bool -> Bool
  propOr x y = x || y == or x y`
- Testing: `Start = test propOr`
- Tests executed:
  - `propOr True True`, `propOr True False`, `propOr False True`, `propOr False False`
- Answer of GAST: Passed after 4 tests

Property: Relation between manipulations

- Properties of an implementation of a stack:
  - `after pushing an element is the top`
  - `top (push e s) == e`
  - `Popping the pushed element yields the original stack`
  - `pop (push e s) == s`
  - Combined property to be tested
    - `propStack e s
     = top (push e s) == e && pop (push e s) == s`
  - Testing: `Start = test propStack`
  - Answer of GAST: Passed after 1000 tests

- Where it is impossible to test all arguments, stop after a specified number (here 1000)
- No proof, but successful test
what are generics

- uniform tree representation of all types
- handles all types by a single representation
- conversion between actual user-defined type and tree representation by generic system
- preserves full statically type check
- in run-time type errors
- best of both worlds:
- user defined types
- generic representation of constructors
- fixed class of functions can handle all types
- enables us to define one generic function
- systematic generation of generic trees
- automatic conversion of trees to needed types

what test data?

- standard approach random
- easy to implement
- works pretty well for small programs
- are all relevant cases covered?
- duplicates do occur (and are often useless)
- our approach systematic
- coverage of standard cases (guaranteed)
- instances of data types from small to large
- properties can have
- one new instance for each new type
- use generics!

requirements for data generation

- properties can have
- any number of arguments
- each argument can have any type
- we need a generator for all types occurring in the properties
- how to define one general generate function?
- one function for all types
- one new instance for each new type
- systematic generation
- use generics!

generic function generate

- generics enable us to define one generic function, that generates data for all types
- systematic generation of generic trees
- use generic type info to generate only valid trees
- choose randomly at forks
- remember branches visited to avoid duplicates
- small values will occur soon, large values later
- for each new type T we can derive a generator by deriving Generic T
- automatic conversion from tree to desired type
- type safe: all types are statically checked
- no runtime type errors possible

test execution

- fair ordering of tests
- consider a function with 2 arguments: f x y
  - y from [a, b, c, ...]
  - x from [1, 2, 3, ...]
- use fair order of tests:
  - f 1 a, f 1 b, f 2 a, f 1 c, f 2 b, f 2 c, ...
  - instead of f 1 a, f 2 b, f 1 c, f 1 c, ...
- recording arguments used
  - to tell the user what is tested now
  - to generate error messages
  - "(left x)=2" Failed after 2 tests: 1.305828e+09

test analysis

- perform a finite number of tests
- counterexample found: Failed
- all possible arguments tried: Proof
- usually 100 to 1000 tests are sufficient to obtain a reasonable confidence
conclusions

• an automatic test system is very useful
  - user writes properties to be obeyed by software
  - GAST tests these properties automatically
  - system generates test data
    - systematic generation based on type information
    - works for every data type
  - executes tests for these data
  - analyzes test results
• rapid (repeated) testing easy and fast
• direct connection between property and software
• encourages writing formal properties
• not restricted to functional programming languages
• generics are useful enabling implementation of GAST

future/related work

• GUI for GAST in IDE
  - click on property to be checked
  - construct and test properties interactively
• investigate relation with other test systems
  - TORX
  - other test systems are able to
    - generate test data systematically based on type,
    - execute tests,
    - analyze results
• more case studies
  - more and larger applications
  - test software specified in Clean,
    but implemented in other programming languages

more information

• pieter@cs.kun.nl
• www.cs.kun.nl/~pieter
  - research paper about implementation of GAST
  - demo version
  - for Clean www.cs.kun.nl/~clean
  - use software
  - documentation
  - research papers

additional features of GAST

• implication condition on test data
  propMax x y = x <= y => max x y ≡ y
• exists operator find value such that property holds
  propInvar :: Color -> Property
  propInvar c = Exists \c` = c == inverse c`
• information about test data used
  propProof c e s = classify (isEmpty s) s (propStack e s)
  - using generics, any expression can be a label
  - result for propStack:
    Passed after 1000 tests
[1]: 43 (4.3%)
• user defined test data instead of generated data
  propFib n = n >= 0 => Fib n == FibLin n
  propFibR = propFib For [0..15]

Example: generic equality

generic gEq a :: a a -> Bool
simple instances for generic type, like

gEq{UNIT} UNIT = True
\forall x y (gEq{EITHER} (LEFT x) (LEFT y)) = gEq x y
\forall x y (gEq{EITHER} (RIGHT x) (RIGHT y)) = gEq x y
\forall x y (gEq{EITHER} e1 e2) = False

derive a generic version for Color by: derive gEq Color

Generic Generic Bool

<table>
<thead>
<tr>
<th>Color</th>
<th>Color</th>
<th>equality</th>
<th>Bool</th>
</tr>
</thead>
</table>

derive gEq Color

Generic Generic Bool