Gast: Generic Automated Software Testing

Pieter Koopman, Artem Alimarine, Jan Tretmans, and Rinus Plasmeijer

Nijmegen Institute for Information and Computing Sciences, The Netherlands
{pieter, alimarine, tretmans, rinus}@cs.kun.nl

Abstract. Software testing is a labor-intensive and hence expensive, yet heavily used technique to control quality. In this paper we introduce Gast, a fully automatic test-tool. Properties from first order logic can be expressed in the system, Gast automatically generates appropriate test-data, evaluates the property for these values, and analyzes the test-results. In this way it becomes easier and cheaper to test software components. The distinguishing property of our system is that the test-data are generated in a systematic and generic way using generic programming techniques. This implies that there is no need for the user to indicate how data should be generated. Moreover, duplicated tests are avoided and for finite domains Gast is able to proof a property by testing it for all possible values. As an important side-effect, it also encourages stating formal properties of the software.

1 Introduction

Testing is an important and heavily used technique to measure and ensure software quality. It is part of almost any software project. The testing phase of typical projects takes up to 50% of the total project, and hence contributes significantly to the project costs. Any change in the software can potentially influence the result of a test. For this reason tests have to be repeated often. This is error-prone, boring, time consuming, and expensive.

In this paper we introduce a tool for automatic testing. Automatic testing significantly reduces the effort of individual tests. This implies that performing the same test becomes cheaper, or one can do more tests within the same budget. In this paper we restrict ourselves to *functional testing*, i.e. examination whether the software obeys the given specification.

In this context we distinguish four steps in the process of functional testing: 1) *formulation of a property* to be obeyed, what has to be tested; 2) *generation of test data*: the decision for which input values the property should be examined, 3) *test execution*: running the program with the generated test data, and 4) *test result analysis*: making a verdict based on the results of the test execution.

The introduced Generic Automatic Software Test–system, Gast, performs the last three steps fully automatically. Gast generates test-data based on the types used in the properties, it executes the test for the generated test-values, and gives a an analysis of these test-results. The system either produces a message that the software successfully passed the specified number of tests for a particular property, or shows the arguments for which the property does not hold.

GASTmakes testing easier and cheaper. As an important side-effect it encourages the writing of properties that should hold. This contribute to the documentation of the system. Moreover, there is empirical evidence that writing specifications on its own contributes to the quality of the system [16].

GAST is implemented in the functional programming language CLEAN[14]. The primary goal is to test software written in CLEAN. However, it is not restricted to software written in CLEAN. Functions written in other languages can be called through the foreign function interface, or programs can be invoked.

The properties to be tested are expressed as functions in CLEAN. The properties have the power of first order predicate logic. The properties can state properties about individual implemented functions and datatypes as well as larger pieces of software, or even about complete programs. The definition of properties and their semantics are introduced in section 3.

Existing automatic test-systems, such as [3], use random generation of test-data. When the test involves user-defined datatypes, the tester has to indicate how elements of that type should be generated. Our test-system, GAST, improves both points. Using systematic generation of test-data, duplicated tests involving user-defined types do not occur. This makes even proofs possible. By using a generic generator the tester does not have to define how elements of a user-defined type have to be generated. Generic programming deals with the universal representation of a type instead of concrete types. This is explained in section 2. Automatic data generation is treated in section 4. If the tester wants to control the generation of data explicitly, he is able to do so (section 7).

After these preparations, the test execution is straightforward. The property is tested for the generated test-data. GAST uses the code generated by the CLEAN compiler to compute the result of applying a property to test-data. This has two important advantages. First, there cannot exist semantical differences between the ordinary CLEAN code and the interpretation of properties. Secondly, it keeps GAST simple. In this way we are able to construct a light-weight test-system. This is treated in section 5. Next, test result analysis is illustrated by some examples. In section 7 we introduce some additional tools to improve the test result analysis. Finally, we will discuss related work, open issues and conclude.

2 Generic Programming

Generic programming [7, 8, 1, 6] is based on a universal tree representation of datatypes. Whenever required, elements of any datatype can be transformed to and from that universal tree representation. The generic algorithm is defined on this tree representation. By applying the appropriate transformations, this generic algorithm can be applied to any type.

Generic programming is essential for the implementation of GAST. However, users does not have to know anything about generic programming. The reader who wants to get an impression of GAST might skip this section on first reading.

Generic extensions are currently developed for Haskell [10] and CLEAN [14]. In this paper we will use CLEAN without any loss of generality.

2.1 Generic Types

The universal type is constructed using the following type definitions [1].[1]

```
:: UNIT       = UNIT             // leaf of the type tree
:: PAIR   a b = PAIR a b         // branch in the tree
:: EITHER a b = LEFT a | RIGHT b // choice between a and b
```

As an example, we give two algebraic datatypes, `Color` and `List`, and their generic representation, $\texttt{Color}_g$ and $\texttt{List}_g$. The symbol `:==` in the generic version of the definition indicates that it are just type synonyms, they do not define new types.

```
::Color = Red | Yellow | Blue // ordinary algebraic type definition
::Color_g :== EITHER (EITHER UNIT UNIT) UNIT // generic representation

::List a = Nil | Cons a (List a)
::List_g a :== EITHER UNIT (PAIR a (List a))
```

The transformation from the user-defined type to its generic counterpart are done by automatically generated functions like[2]:

```
ColorToGeneric :: Color -> EITHER (EITHER UNIT UNIT) UNIT
ColorToGeneric Red    = LEFT (LEFT UNIT)
ColorToGeneric Yellow = LEFT (RIGHT UNIT)
ColorToGeneric Blue   = RIGHT UNIT

ListToGeneric :: (List a) -> EITHER UNIT (PAIR a (List a))
ListToGeneric Nil         = LEFT UNIT
ListToGeneric (Cons x xs) = RIGHT (PAIR x xs)
```

The generic system automatically generates these functions and their inverses.

2.2 Generic Functions

Based on this representation of types one can define generic functions. As example we will show the generic definition of equality[3].

```
generic gEq a  :: a a -> Bool
gEq{|UNIT|}         _          _          = True
gEq{|PAIR|}   fa fx (PAIR a x) (PAIR b y) = fa a b && fx x y
gEq{|EITHER|} fl fr (LEFT x)   (LEFT y)   = fl x y
gEq{|EITHER|} fl fr (RIGHT x)  (RIGHT y)  = fr x y
gEq{|EITHER|} _  _  _          _          = False
gEq{|Int|}          x          y          = x == y
```

The instances for `PAIR` and `EITHER` have additional arguments. These arguments (provided by the generic system) define how subtypes can be compared. In this respect generic functions differ from ordinary classes in functional programming.

In order to use this equality for `Color` an instance of `gEq` for `Color` must be derived, by: `derive gEq Color`. The system generates code equivalent to

[1] Clean uses additional constructs for information on constructors and record fields.

[2] We use the direct generic representation of result types instead of the type synonyms $\texttt{Color}_g$ and $\texttt{List}_g$ since it shows the structure of result more clearly.

[3] We only consider the basic type `Int` here. Other basic types are handled similarly.

```
gEq{|Color|} x y = gEq{|EITHER|} (gEq{|EITHER|} gEq{|UNIT|} gEq{|UNIT|})
                   gEq{|UNIT|} (ColorToGeneric x) (ColorToGeneric y)
```

The additional arguments needed by `gEq{|EITHER|}` in `gEq{|Color|}` are determined by the generic representation of the type `Color`: $\texttt{Color}_g$.

If this version of equality is not what you want, you can always define your own instance of `gEq` for `Color`, instead of deriving the default.

The infix version of this equality is defined as:

```
(===) infix 4 :: !a !a -> Bool | gEq{|*|} a
(===) x y = gEq{|*|} x y
```

This enables us to write expressions like `Red === Blue`. The necessary type conversions form `Color` to $\texttt{Color}_g$ need not to be specified, they are generated and applied at the appropriate places by the generic system.

It is important to note that the user of types like `Color` and `List` need not be aware of the generic representation of types. Types can be used and introduced as usual; the static type system also checks the consistent use of types as usual.

3 Specification of Properties

The first step in the testing process is the formulation of properties in a formalism that can be handled by Gast. In order to handle properties from first order predicate logic in Gast we represent them as functions in Clean. Each property is expressed by a function yielding a Boolean value. Expressions with value `True` indicates a successful test, `False` indicates a counter example. This solves the famous *oracle problem*: how do we decide whether the result of a test is correct or not.

The arguments of such a property represent the universal variables of the logical expression. Properties can have any number of arguments, each of these arguments can be of any non-polymorphically type.

These functions can be used to specify properties of single functions or operations in Clean, as well as properties of large combinations of functions, or even of entire programs.

In this paper we will only consider well-defined and finite values as test-data. Due to this restriction we are able to use the and-operator (`&&`) and or-operator (`||`) of Clean to represent the logical operators and ($\wedge$) and or ($\vee$) respectively.

Our first example involves the implementation of the logical `or`-function using only a two-input `nand`-function as basic building element.

```
or :: Bool Bool -> Bool
or x y = nand (not x) (not y) where not x = nand x x
```

The desired property is that the value of this function is always equal to the value of the ordinary or-operator, `||`, from Clean. That is, the `||`-operator serves as specification for the new implementation, `or`. In logic, this is:

$$\forall x \in Bool \,.\, \forall y \in Bool \,.\, x || y = or\, x\, y$$

This property can be represented by the following function in Clean. By convention we will prefix the name properties by `prop`.

```
propOr :: Bool Bool -> Bool
propOr x y = x||y == or x y
```

The user invokes the testing of this property by the main function:

```
Start = test propOr
```

GAST yields `Proof: success for all arguments after 4 tests` for this property. Since there are only finite types involved the property can be proven by testing.

For our second example we consider the classical implementation of stacks:

```
:: Stack a :== [a]

pop :: (Stack a) -> Stack a
pop [_:r] = r

top :: (Stack a) -> a
top [a:_] = a

push :: a (Stack a) -> Stack a
push a s = [a:s]
```

A desirable property for stacks is that after pushing some element onto the stack, that element is on top of the stack. Popping an element just pushed on the stack yields the original stack. The combination of these properties is expressed as:

```
propStack :: a (Stack a) -> Bool | gEq{|*|} a
propStack e s = top (push e s) === e && pop (push e s) === s
```

This property should hold for any type of stack-element. Hence we used polymorphic functions and the generic equality, `===`, here. However, GAST can only generate test-data for some concrete type. Hence, we have to specify which type GAST should use for the type argument `a`. For instance by:

```
propStackInt :: (Int (Stack Int) -> Bool)
propStackInt = propStack
```

In contrast to properties that use overloaded types, it actually does not matter much which concrete type we choose. A polymorphic property will hold for elements of any type if it holds for elements of type `Int`. The test is executed by `Start = test propStackInt`. GAST yields: `Passed after 1000 tests`. This property involves the very large type integer and the infinite type stack, only testing for a finite number of cases, here 1000, is possible.

In `propOr` we used a reference implementation (`||`) to state a property about a function (`or`). In `propStack` expressed the desired property directly as a relation between functions on a datatype. Other kind of properties state relations between the input and output of functions, or use model checking based properties. For instance, we have tested a system for safe communication over unreliable channels by an alternating bit protocol with the requirement that sequence of received messages should be equal to the input sequence of messages.

One often adds the implication operator, $\Rightarrow$, as a primitive to the predicate logic. For instance $\forall x.x \geq 0 \Rightarrow (\sqrt{x})^2 = x$. We can use the law $p \Rightarrow q = \neg p \vee q$ to implement it:

```
(===>) infix 1 :: Bool Bool -> Bool
(===>) p q = not p || q
```

In first order predicate logic one also has the existential quantifier, $\exists$. If this is used to introduce values in a constructive way it can be directly transformed to local definitions in a functional programming language, for instance as: $\forall x.x \geq 0 \Rightarrow \exists y.y = \sqrt{x} \wedge y^2 = x$ can directly be expressed using local definitions.

```
propSqrt :: Real -> Bool
propSqrt x = x >=0 ===> let y = sqrt x in y*y == x
```

In general it is not possible to construct an extensionally quantified value. For instance, for a type *Day* and a function *tomorrow* we require that for each day there exists a tomorrow $\forall day.\exists d.tomorrow\ day = d$. The day d has to be searched in the type *Day*. GAST has the operator `Exists` to express this:

```
propTomorrow :: Day -> Property
propTomorrow day = Exists \d = tomorrow day === d
```

The only task of the tester is to write properties, like `propOr`, and to invoke the testing by `Start = test propOr`. Based on the type of arguments needed by the property, the test system will generate test-data, execute the test for these values, and analyze the results of the tests. In the following three sections we will explain how GAST works. The tester does not have to know this.

3.1 Semantics of Properties

For GAST we extend the standard operational semantics of CLEAN. The standard reduction to weak head normal form is denoted as $whnf\ [\![\ e\ ]\!]$. The additional rules are applied after this ordinary reduction. The implementation will follow these semantics rules directly. The possible results of the evaluation of a property are the values **Suc** for success, and **CE** for counterexample. In these rules $\lambda x.p$ represents any function. That is any expression of type $\tau \rightarrow \sigma$ (i.e. a partially parameterized function, or a lambda-expression). The evaluation of a property, $Eval\ [\![\ p\ ]\!]$, yields a list of results:

$$Eval\ [\![\ \lambda x.p\ ]\!] = [r | v \leftarrow genAll; r \leftarrow Eval\ [\![\ (\lambda x.p)\ v\ ]\!]] \quad (1)$$

$$Eval\ [\![\ \mathbf{True}\ ]\!] = [\mathbf{Suc}] \quad (2)$$

$$Eval\ [\![\ \mathbf{False}\ ]\!] = [\mathbf{CE}] \quad (3)$$

$$Eval\ [\![\ e\ ]\!] = Eval\ [\![\ whnf\ [\![\ e\ ]\!]\ ]\!] \quad (4)$$

To test property p we evaluate $Test\ [\![\ p\ ]\!]$. The rule $An\ [\![\ l\ ]\!]\ n$ analysis a list of the test-results In rule 5 N is the maximum number of tests. There are three possible test results: **Proof** indicates that the property holds for all well-defined values of the argument types, **Passed** indicated that the property passed N tests without finding a counterexample, **Fail** indicates that a counterexample is found.

$$Test\ [\![\ p\ ]\!] = An\ [\![\ Eval\ [\![\ whnf\ [\![\ p\ ]\!]\ ]\!]\ ]\!]\ N \quad (5)$$

$$An\ [\![\ [\,]\ ]\!]\ n = \mathbf{Proof} \quad (6)$$

$$An\ [\![\ l\ ]\!]\ 0 = \mathbf{Passed} \quad (7)$$

$$An\ [\![\ [\mathbf{CE} : rest]\ ]\!]\ n = \mathbf{Fail} \quad (8)$$

$$An\ [\![\ [r : rest]\ ]\!]\ n = An\ [\![\ rest\ ]\!]\ (n-1), \text{if } r \neq \mathbf{CE} \quad (9)$$

The most important properties of this semantics are:

$$Test\,[\![\ \lambda x.p\ ]\!] = \mathbf{Proof} \Rightarrow \forall v.(\lambda x.p)v \tag{10}$$

$$Test\,[\![\ \lambda x.p\ ]\!] = \mathbf{Fail} \Rightarrow \exists v.\neg(\lambda x.p)v \tag{11}$$

$$Test\,[\![\ p\ ]\!] = \mathbf{Passed} \Leftrightarrow \forall r \in (take\ N\ Eval\,[\![\ p\ ]\!]).r \neq \mathbf{CE} \tag{12}$$

Property 10 state that GAST only produces **Proof** if the property is universal valid. According to 11 the system yields only **Fail** if a counter example exists. Finally, the systems yields **Passed** if the first N tests does not contain a counterexample, (12). These properties can be proven by induction and case distinction from the rules 1 to 9 given above. Below we will introduce some additional rules for $Eval\,[\![\ p\ ]\!]$, in such a way that these properties are preserved.

The semantics of the `Exists`-operator is:

$$Eval\,[\![\ \mathbf{Exists}\ \lambda x.p\ ]\!] = One\,[\![\ [r|v \leftarrow genAll; r \leftarrow Eval\,[\![\ (\lambda x.p)\ v\ ]\!]\ ]\!]\ M \tag{13}$$

$$One\,[\![\ [\,]\ ]\!]\ m = [\,\mathbf{CE}\,] \tag{14}$$

$$One\,[\![\ l\ ]\!]\ 0 = [\,\mathbf{Undef}\,] \tag{15}$$

$$One\,[\![\ [\,\mathbf{Suc} : rest\,]\ ]\!]\ m = [\,\mathbf{Suc}\,] \tag{16}$$

$$One\,[\![\ [\,r : rest\,]\ ]\!]\ m = One\,[\![\ rest\ ]\!]\ (m-1), \text{if } r \neq \mathbf{Suc} \tag{17}$$

The rule $One\,[\![\ l\ ]\!]$ scans a list of semantic results, it yields success if the list of results contains at least one success within the first M results. As soon as one or more results are rejected the system cannot proof the property any more. It can, however, successfully test the property for N values. To ensure termination also the number of rejected test is limited by an additional counter. These changes for $An\,[\![\ l\ ]\!]$ are implemented by `analyse` in section 6.

4 Generating Test-Data

To test a property, step 2) in the test process, we need a list of values of the argument type. GAST will evaluate the property for the values in this list.

Since we are testing in the context of a referentially transparent language, we are only dealing with pure functions: the result of a function is completely determined by its arguments. This implies that repeating the test for the same arguments is useless: referential transparency guarantees that the results will be identical. GAST should prevent the generation of duplicated test-data.

For finite types like `Bool` or non-recursive algebraic datatypes we can generate all elements of the type as test-data. For basic types like `Real` and `Int`, generating all elements is not feasible. There are far too many elements, e.g. there are 2^{32} integers on a typical computer. For these types, we want GAST to generate some common border values, like 0 and 1, as well as random values of the type. Here, preventing duplicates is usually more work than repeating the test. Hence we do not require that GAST prevents duplicates here.

For recursive types, like list, there are infinitely many instances. GAST is only able to test properties involving these types for a finite fractionnumber of

these values. Recursive types are usually handled by recursive functions. Such a function typically contains special cases for small elements of the type, and recursive cases to handle other elements. In order to test these functions we need values for the special cases as well as some values for the general cases. We achieve this by generating a list of values of increasing size. Preventing duplicates is important here as well.

The standard implementation technique in functional languages would probably make use of classes to generate, compare and print elements of each datatype involved in the tests [3]. Instances for standard datatypes can be provided by a test-system. User-defined types however, would require user-defined instances for all types, for all classes. Defining such instances is error prone, time consuming and boring. Hence, a class based approach would hinder the ease of use of the test-system. Special about Gast is that we use generic programming techniques such that one general solution can be provided once and for all.

For the generation of test-data, Gast generates a list of generic representations of the desired type. The generic-system transforms these values to instances of the types needed.

Obviously, not any generic tree can be transformed to instances of a given type. For the type `Color` only the trees `LEFT (LEFT UNIT)`, `LEFT (RIGHT UNIT)`, and `RIGHT UNIT` represent valid values. Fortunately, the generic system provides the necessary information to guide the generation of trees in the form of additional arguments, as in the `gEq` example shown above.

For recursive types, the generic tree can grow infinitely. Without detailed knowledge about the type, one cannot predict where infinite branches occur. This implies that any systematic depth-first strategy to traverse the tree for possible values can fail to terminate. Moreover, small values which appear close to the root of the generic tree, have to be generated first. A depth-first traversal will encounter these values too late. Finally, a left-to-right strategy will favor values in the left branches and visa versa. A bias in any direction is undesirable.

In order to meet all these requirements, we use a strategy that uses a random choice at each `EITHER` in the type tree. To prevent duplicates we record the tree representation of the generated values in the datatype `Trace`.

```
:: Trace = Unit | Pair [(Trace,Trace)] [(Trace,Trace)]
         | Either Bool Trace Trace | Int [Int] | Done | Empty
```

We use a single type `Trace` to keep track of visited parts of the generic tree (rather than the generated values or their generic representation) to avoid type incompatibilities. We want to manipulate generic representations of any type in a single function `nextTrace` (instead of a generic `nextTrace`). The type `Trace` looks quite different from the ordinary generic tree since we record *all* generated values in a single tree. An ordinary generic tree just represents *one* single value.

New parts of the trace are constructed by the generic function `generate`. The function `nextTrace` prepares the trace for the generation of the next element from the list of test-data. It uses the `RandomStream`, a list of pseudo random values, to choose randomly between extending the left or right branch of the trace. When the generation of the chosen branch fails, it tries to extend the other branch. If

both branches cannot be extended, all values in this subtree are generated and the result is `Done`.

The function `genAll` uses `generate` to produce the list of all values of the desired type. It generates values until the next trace indicates that we are done.

```
genAll :: RandomStream -> [a] | generate{|*|} a
genAll rnd = g Empty rnd
where g Done rnd = []
      g t    rnd = let (x, t2, rnd2) = generate{|*|} t rnd
                       (t3, rnd3)    = nextTrace t2 rnd3
                   in [x: g t3 rnd3]
```

The code fragment of `nextTrace` handling `Either` is listed here. This is the most interesting case. The class `genElem` produces an element of the desired type using the random stream.

```
nextTrace (Either _ tl tr) rnd
    = let (b, rnd2) = genElem rnd in
      if b  (let (tl‘, rnd3) = nextTrace tl rnd2 in
              case tl‘ of
                Done    = let (tr‘, rnd4) = nextTrace tr rnd3 in
                          case tr‘ of
                            Done = (Done, rnd4)
                            _    = (Either Right tl tr‘, rnd4)
                        = (Either Left tl‘ tr, rnd3))
            (let (tr‘, rnd3) = nextTrace tr rnd2 in
              case tr‘ of
                Done    = let (tl‘, rnd4) = nextTrace tl rnd3 in
                          case tl‘ of
                            Done = (Done, rnd4)
                            _    = (Either Left tl‘ tr, rnd4)
                        = (Either Right tl tr‘, rnd3))
```

The corresponding instance of `generate` follows the direction indicated in the trace. When the trace is empty, it takes a boolean from the random stream and creates the desired value as well as the initial extension of the trace.

```
generic generate a :: Trace RandomStream -> (a, Trace, RandomStream)

generate{|EITHER|} fl fr Empty rnd
  = let (f,rnd2) = genElem rnd in
    if f (let (l,tl,rnd3) = fl Empty rnd2
          in  (LEFT  l, Either Left  tl Empty, rnd3))
         (let (r,tr,rnd3) = fr Empty rnd2
          in  (RIGHT r, Either Right Empty tr, rnd3))
generate{|EITHER|} fl fr (Either left tl tr) rnd
  | left = let (l,tl2,rnd2) = fl tl rnd
           in  (LEFT  l, Either left tl2 tr, rnd2)
         = let (r,tr2,rnd2) = fr tr rnd
           in  (RIGHT r, Either left tl tr2, rnd2)
```

For `Pair` we extend both branches in turn with a step. Two lists of tuples are used to keep track of this. By convention the first tuple of the first list contains

the traces to be used by `generate`. The second list is used to efficiently implement adding new tuples to the tail of the list. At each step two new tuples are added (if they exist): the current left branch, with the next right branch; and the next left branch, with a new (empty) right branch.

4.1 Generic generation of Functions as Test-Data

Since CLEAN is a higher order language it is perfectly legal to use a function as an argument or result of a function. Also in properties, the use of higher order functions can be very useful. A well-known property of the function `map` can be expressed as:

```
propMap :: (a->b) (b->c) [a] -> Bool | gEq{|*|} c
propMap f g xs = map g (map f xs) === map (g o f) xs
```

In order to test such a property we must choose a concrete type for the polymorphic arguments. Choosing `Int` for all type variables yields:

```
propMapInt :: ((Int->Int) (Int->Int) [Int] -> Bool)
propMapInt = propMap
```

This leaves us with the problem of generating functions automatically. Functions are not datatypes and hence cannot be generated by the default generic generator. Fortunately, the generic framework provides a way to create functions. We generate function of type `a->b` by defining a generic instance for `generate{|(->)|}`. Such a function is defined by generating a list of values of type `b`. The argument of type `a` is transformed in a generic way to an index in this list. The generated function just transforms its argument to an index and selects the corresponding element from the list of values. For instance, a generated function of type `Int -> Color` could look like `\a = [Red,Yellow,Blue] !! (abs a % 3)`. Due to space limitations we omit the details here.

5 Test Execution

Step 3) in the test process is the test execution. The implementation of an individual test is a direct translation of the given semantic rules introduced above. The type class `Testable` contains the function `evaluate` which directly implements the rules for $Eval\ [\![\ p\ ]\!]$.

```
class Testable a where evaluate :: a RandomStream Admin -> [Admin]
```

In order to be able to show the arguments used in a specific test, we administrate the arguments represented as strings as well as the result of the test in a record called `Admin`. There are three possible results of a test: undefined (`UnDef`), success (`OK`), and counter example found (`CE`).

```
:: Admin = {res::Result, args::[String]}
:: Result = UnDef | OK | CE
```

Instances of `TestArg` can be argument of a property. The system should be able to generate elements of such a type (`generate`) and to transform them to string (`genShow`) in order to add them to the administration.

```
class TestArg a | genShow{|*|}, generate{|*|} a
```

The semantic equations 2 and 3 are implemented by the instance of evaluate for the type `Bool`.

```
instance Testable Bool
where evaluate b rs result
 = [{result & res = if b OK CE, args = reverse result.args}]
```

The rule for function application, semantic equation 1, is complicated slightly by administrating function arguments.

```
instance Testable (a->b) | Testable b & TestArg a
where evaluate f rs result
        = let (rs, rs2) = split rs  in  forAll f (gen rs) rs2 result
forAll f list rs r
 = diagonal [apply f a (genRandInt seed) r \\ a<-list & seed<-rs ]
apply f a rs r = evaluate (f a) rs {r & args = [show a:r.args]}
```

The function `diagonal` takes care of a *fair* order of tests. For a 2-argument function f, the system generates two sequences of arguments, call them $[a, b, c, ..]$ and $[u, v, w, ..]$ respectively. The order of tests is $f\,a\,u, f\,a\,v, f\,b\,u, f\,b\,v, f\,c\,u, ..$ rather than $f\,a\,u, f\,a\,v, f\,a\,w, .., f\,b\,u, f\,b\,v, f\,b\,w, ...$

6 Test Result Evaluation

The final step, step 4), in the test process is the evaluation of results. The system just scans the generated list of test-results as indicated by $An\,[\![\,l\,]\!]$. The only extension is the showing of the number and arguments of the current test before the test result is evaluated. In this way the tester of Gast is able to identify the data causing an runtime error or taking a lot of time. A somewhat simplified version of the function `test` is:

```
test :: p -> [String] | Testable p
test p = analyse (evaluate p RandomStream newAdmin) maxTests MaxArgs
where analyse :: [Admin] Int Int -> [String]
      analyse [] n m = ["\nProof: success for all arguments"]
      analyse l  0 m = ["\nPassed ",toString maxTests," tests"]
      analyse l  n 0 = ["\nPassed ",toString maxArgs," arguments"]
      analyse [res:rest] n m
        = [blank,toString (maxTests-n+1),":":showArgs res.args
           case res.res of
             UnDef  = analyse rest n (m-1)
             OK     = analyse rest (n-1) (m-1)
             CE     = ["\nCounterexample: ": showArgs res.args []]]
```

7 Additional features

In order to improve the power of the test-tool, we introduce some additional features. These possibilities are realized by combinators (functions) that manipulate the administration. We consider the following groups of combinators: 1) an improved implication, $p \Rightarrow q$, that discards the test if p does not hold; 2) combinators to collect information about the actual test-data used; 3) combinators to apply user-defined test-data instead of generated test-data.

7.1 Implication

Although the implication operator `===>` works correctly it has an operational drawback: if p does not hold, the property $p \Rightarrow q$ holds and is counted as a successful test. This operator is often used to put an restriction on arguments to be considered, as in $\forall x.x \geq 0 \Rightarrow (\sqrt{x})^2 = x$. Here we wants only to consider tests where $x \geq 0$ holds, in other situations the test should not be taken into account. This is represented by the result *undefined*. We introduce the operator `==>` for this purpose.

$$\mathit{Eval}\,[\![\ \mathbf{True} ==> p\ ]\!] = \mathit{Eval}\,[\![\ p\ ]\!] \tag{18}$$

$$\mathit{Eval}\,[\![\ \mathbf{False} ==> p\ ]\!] = [\,\mathbf{Rej}\,] \tag{19}$$

If the predicate holds the property `p` is evaluated, otherwise we explicitly yield an undefined result. The implementation is:

```
(==>) infixr 1 :: Bool p -> Property | Testable p
(==>) c p
    | c = Prop (evaluate p)
        = Prop (\rs r = [{r & res = Undef}])
```

Since we need to access the administration here, the property on the right-hand side is not a Boolean, but a datatype holding the function to access the administration.

```
:: Property = Prop (RandomStream Admin -> [Admin])
instance Testable Property
where evaluate (Prop p) rs result = p rs result
```

The operator `==>` can be used as `===>` in `propSqrt` above. The result of executing `test propSqrt` is `Counter-example found after 2 tests: 3.07787e-09`. The failure is caused by the finite precision of reals.

7.2 Information about Test-Data used

For properties like `propStack`, it is impossible to test all possible arguments. The tester might be curious to known more about the actual test-data used in a test. The system provides two combinators to help him:

```
label    ::      l p -> Property | Testable p & genShow{|*|} l
classify :: Bool l p -> Property | Testable p & genShow{|*|} l
```

The function `label` always adds the given label; `classify` only adds the label when the condition holds. In order to be able to collect labels we extend the administration `Admin` with a field `labels` of type `[String]`. GAST collects the strings, orders them alphabetically, counts them and computes the fraction of tests that contain this label. The label can be an expression of any type, it is converted to a string in a generic way (by `genShow{|*|}`).

These functions do not change the semantics of the specification, their only effect is the additional information in the report to the tester.

$$\mathit{Eval}\,[\![\ \mathbf{label}\ l\ p\ ]\!] = \mathit{Eval}\,[\![\ p\ ]\!] \text{ adds } l \text{ to the administration} \tag{20}$$

$$\mathit{Eval}\,[\![\ \mathbf{classify\ True}\ l\ p\ ]\!] = \mathit{Eval}\,[\![\ \mathbf{label}\ l\ p\ ]\!] \tag{21}$$

$$\mathit{Eval}\,[\![\ \mathbf{classify\ False}\ l\ p\ ]\!] = \mathit{Eval}\,[\![\ p\ ]\!] \tag{22}$$

We will illustrate the use of these functions. It is possible to view the exact test-data used for testing the property of stacks by

```
propStackL :: Int (Stack Int) -> Property
propStackL e s = label (e,s) (top (push e s)===e && pop (push e s)===s)
```

A possible result of testing `propStackL` for only 4 combinations of arguments is:

```
Passed 4 tests
(0,[0,1]): 1 (25%)
(0,[0]): 1 (25%)
(0,[]): 1 (25%)
(1,[]): 1 (25%)
```

The function `classify` can, for instance, be used to count the number of empty stacks occurring in test-data.

```
propStackC :: Int (Stack Int) -> Property
propStackC e s
 = classify (isEmpty s) s (top (push e s) === e && pop (push e s) === s)
```

A typical result for 200 tests is:

```
Passed 200 tests
[]: 18 (9%)
```

7.3 User-defined Test-Data

Gast generates sensible test-data based on the type of the arguments. Sometimes the tester is not satisfied with this behavior. This occurs if very few generated elements obey the condition of an implication, or cause enormous calculations, or overflow.

The property `propFib` below uses the well-known naive definition of the Fibonacci function, `fib`, as specification. The property states that the value of the efficient version of this function, `fibLin`, should be equal to the value of `Fib` for non-negative arguments.

```
fib 0 = 1
fib 1 = 1
fib n = fib (n-1) + fib (n-2)

fibLin n = f n 1 1
where f 0 a b = a
      f n a b = f (n-1) b (a+b)

propFib n = n>=0 ==> fib n == fibLin n
```

One can prevent long computations and overflow by limiting the size of the argument by an implication. For instance:

```
propFib n = n>=0 && n<=15 ==> fib n == fibLin n
```

This is a rather unsatisfactory solution. The success rate of tests in the generated list of test-values will be low, due to the condition many test results will be undefined (since the condition of the implication is false). In those situations it is more efficient if the user specifies the test-values, instead of letting the Gast generate it. For this purpose the combinator `For` is defined. The implementation is very simple using the machinery developed above:

```
(For) infixl 0 :: (x->p) [x] -> Property | Testable p & TestArg x
(For) p list = Prop (forAll p list)
```

This can for instance be used to test the equivalence of the Fibonacci functions for all arguments from 0 to 15:

```
propFibR = propFib For [0..15]
```

Testing yields `Proof: success for all arguments after 16 tests.`

The semantics of the `For` combinator is:

$$Eval\,[\![\, \lambda\, x.\, p\ \mathbf{For}\ list\ ]\!] = [r | v \leftarrow list; r \leftarrow Eval\,[\![\, (\lambda\, x.p)\, v\ ]\!]\,] \tag{23}$$

8 Related Work

Testing is labor-intensive, boring and error-prone. Moreover, it has to be done often by software engineers. Not surprisingly, a large number of tools has been developed to automate testing [2]. See [15] for an (incomplete) overview of existing tools. Although some of these tools are well engineered, none of them gives automatic support like Gast does for *all* steps of the testing process. None of the other tool is able to generate test-data systematically for arbitrary types based on the types used in properties.

In the functional programming world there are some related tools. The tool QuickCheck [3, 4] has similar ambitions as our tool. Distinguishing features of our tool are: the generic generation of test-data for arbitrary types (instead of based on a user-defined instance of a class), and the systematic generation of test-data (instead of random). As a consequence of the systematic generation of test-data, our system is able to detect that all possible values are tested and hence the property is proven. Moreover, Gast offers a complete implementation of first order predicate logic.

Auburn [13] is a tool for automatic benchmarking of functional datatypes. It is also able to generate some form of test-data, but not in a systematic and generic way. Since there is no specification of properties, only runtime errors can be detected.

HUnit [9] is the Haskell variant of JUnit [11] for Java. JUnit defines how to structure your test cases and provides the tools to run them. It executes the test defined by the user. Tests are implemented in a subclass of `TestCase`.

An important area of automatic test generation is testing of reactive systems, or control-intensive systems. In these systems the interaction with the environment in terms of stimuli and responses is important. Typical examples are communication protocols, embedded systems, and control systems. Such systems are usually modelled and specified using some kind of automaton or state machine. There are two main approaches for automatic test generation from such specifications. The first is based on Finite State Machines (FSM), and uses the theory of checking experiments for Mealy-machines [17]. Several academic tools exist with which tests can be derived from FSM specifications, e.g., Phact/The Conformance Kit [18]. Although Gast is able to test properties of an FSM, checking the actual state requires additional research.

The second approach is based on labelled transition systems and emanates from the theory of concurrency and testing equivalences [19]. Tools for this approach are, e.g., TGV [20], TESTCOMPOSER [21], TESTGEN [22], and TORX [23, 24]. State-based tools concentrate on the control flow, and cannot usually cope with complicated data structures. As shown above GAST is able to cope with these data structures.

9 Discussion

In this paper we introduce GAST, a generic tool to test software. The complete code, about 600 lines, can be downloaded from `www.cs.kun.nl/~pieter`. The tests are based on properties of the software, stated as functions based on first order predicate logic. Based on the types used in these properties the system generates automatically test-data in a systematic way, checks the property for these generated values, and analyzes the results of these tests.

One can define various kind of properties. The functions used to describe properties are slightly more powerful than first order predicate logic [5]. In our system we are able to express properties known under names as black-box tests, algebraic properties, and model based, pre- and post-conditional. Using the ability to specify the test-data, also user-guided white-box tests are possible.

Based on our experience we indicate four kinds of errors spotted by GAST. The system cannot distinguish these errors. The tester has to analyze them.

1. Errors in the implementation. The kind of mistakes you expect to find.
2. Errors in the specification. In this situation the tested software also does not obey the given property. Analysis of the indicated counter example shows that the specification is wrong instead of the software. Testing improves also the confidence in the accuracy of the properties
3. Errors caused by the finite precision of the computer used. Especially for properties involving reals, e.g. `propSqrt`, this is a frequent problem. In general we have to specify that the difference between the obtained answer and the required solution is smaller than some allowed error range.
4. Non-termination or run-time errors. Although the system does not explicitly handle these errors, the tester notices that the error occurs. Since GAST lists the arguments before executing the test, the values causing the error are known. This appears to detect partially defined functions effectively.

The efficiency of GAST is mainly determined by the evaluation of the property, not by the generation of data. For instance, on a standard PC the system generates up to 100,000 integers or up to 2000 lists of integers per second. In our experience errors pop up rather soon, if they exist. Usually 100 to 1000 tests are sufficient to be reasonably sure about the validity of a property.

In contrast to proof-systems like SPARKLE [12], GAST is restricted to well-defined and finite arguments. In proof-systems one also investigates the property for non-terminating arguments, usually denoted as $\perp$, and infinite arguments (for instance a list with infinite length). Although it is possible to generate undefined

and infinite arguments, it is impossible to stop the evaluation of the property when such an argument is used. This is a direct consequence of our decision to use ordinary compiled code for the evaluation of properties.

Restrictions of our current system are that the types should be known to the system (it is not possible to handle abstract types by generics); if there are restrictions on the types used they should be enforced explicitly; and world access is not supported. In general it is very undesirable when the world (e.g. the file system on disk) is effected by random tests.

Currently the tester has to indicate that a property has to be tested by writing an appropriate `Start` function. In the near future we want to construct a tool that extracts properties from CLEAN modules and tests these properties fully automatically.

GAST is not restricted to testing software written in its implementation language, CLEAN. It is possible to call a function written in some other language through the foreign function interface, or to invoke another program.

Acknowledgements

Many people have contributed to GAST and this paper. We would like to name Marko van Eekelen, Maarten de Mol, Peter Achten, and Susan Evens.

References

1. A. Alimarine and R. Plasmeijer *A Generic Programming Extension for Clean.* IFL2001, LNCS 2312, pp.168–185, 2001.
2. G. Bernot, M.C. Gaudel, B.Marre: *Software Testing based on Formal Specifications: a theory and a tool*, Software Engineering Journal, 6(6), pp 287–405, 1991.
3. K. Claessen, J. Hughes. *QuickCheck: A lightweight Tool for Random Testing of Hasskell Programs.* International Conference on Functional Programming, ACM, pp 268–279, 2000. See also `www.cs.chalmers.se/~rjmh/QuickCheck`.
4. K. Claessen, J. Hughes. *Testing Monadic Code with QuickCheck*, Haskell Workshop, 2002.
5. M. van Eekelen, M. de Mol: *Reasoning about explicit strictness in a lazy language using mixed lazy/strict semantics*, Draft proceedings IFL2002, Report 127-02, Computer Science, Universidad Complutense de Madrid, pp 357–373, 2002.
6. R. Hinze and J. Jeuring. *Generic Haskell: Practice and Theory*, Summer School on Generic Programming, 2002.
7. R. Hinze, and S. Peyton Jones *Derivable Type Classes*, Proceedings of the Fourth Haskell Workshop, Montreal Canada, 2000.
8. Hinze, R. *Polytypic values possess polykinded types*, Fifth International Conference on Mathematics of Program Construction, LNCS 1837, pp 2–27, 2000.
9. HUnit home page: `hunit.sourceforge.net`
10. S. Peyton Jones, J. Hughes: *Report on the programming language Haskell 98 – A Non-strict, Purely Functional Language*, 2002 `www.haskell.org/onlinereport`.
11. JUint home page: `junit.sourceforge.net`
12. M. de Mol, M. van Eekelen, R. Plasmeijer. *Theorem Proving for Functional Programmers,* , LNCS 2312, pp 55–72, 2001. See also `www.cs.kun.nl/Sparkle`.

13. Graeme E. Moss and Colin Runciman. *Inductive benchmarking for purely functional data structures*, Journal of Functional Programming, 11(5): pp 525–556, 2001. Auburn home page: `www.cs.york.ac.uk/fp/auburn`
14. Rinus Plasmeijer and Marko van Eekelen: *Concurrent Clean Language Report (version 2.0)*, 2002. `www.cs.kun.nl/~clean`.
15. Maurice Siteur: *Testing with tools–Sleep while you are working.* See also `www.siteur.myweb.nl`.
16. J. Tretmans, K. Wijbrans, M. Chaudron: *Software Engineering with Formal Methods: The development of a storm surge barrier control system–revisiting seven myths of formal methods*, Formal Methods in System Design, 19(2) pp 195–215, 2001.
17. D. Lee, and M. Yannakakis, M *Principles and Methods for Testing Finite State Machines– A Survey*, The Proceedings of the IEEE, 84(8), pp 1090-1123, 1996.
18. L. Feijs, F. Meijs, J. Moonen, J. Wamel *Conformance Testing of a Multimedia System Using PHACT* in *Workshop on Testing of Communicating Systems 11* pp 193–210, 1998.
19. E. Brinksma, J. Tretmans *Testing Transition Systems: An Annotated Bibliography*", in *Modeling and Verification of Parallel Processes–*4^{th} *Summer School MOVEP 2000* LNCS 2067, pp 186-195, 2001.
20. J. Fernandez, C. Jard, T. Jéron, C. Viho *Using On-the-Fly Verification Techniques for the generation of test suites*, LNCS 1102, 1996.
21. A. Kerbrat, T. Jéron, R. Groz *Automated Test Generation from SDL Specifications*, in *SDL'99, The Next Millennium–Proceedings of the* 9^{th} *SDL Forum*, pp 135–152, 1999.
22. J. He, K. Turner *Protocol-Inspired Hardware Testing*, in *Int. Workshop on Testing of Communicating Systems 12* pp 131–147, 1999.
23. A. Belinfante, J. Feenstra, R. Vries, J. Tretmans, N. Goga, L. Feijs, S. Mauw, L. Heerink *Formal Test Automation: A Simple Experiment*, in *Int. Workshop on Testing of Communicating Systems 12* pp 179-196, 1999".
24. J. Tretmans, E. Brinksma *Côte de Resyste – Automated Model Based Testing*, in *Progress 2002 –* 3^{rd} *Workshop on Embedded Systems*, pp 246–255, 2002.