Testing and Tracing Lazy Functional Programs using QuickCheck and Hat

Koen Claessen[*]
koen@cs.chalmers.se
(lecturer)

Colin Runciman[†]
colin@cs.york.ac.uk
(lecturer)

Olaf Chitil[†] John Hughes[*] Malcolm Wallace[†]

Preliminary Version

1 Introduction

Software often contains errors. We can all agree on that. We can also agree that this is a very undesirable situation. One motivation for using a functional programming language is that it is more difficult (or even impossible) to make low-level mistakes, and it is easier to reason about programs. But even the most advanced functional programmers are not infallible; they misunderstand the properties of their own programs, or those of others, and so commit errors.

We therefore aim to provide functional programmers with tools for testing and tracing programs. In broad terms, testing means first specifying what behaviour is acceptable in principle, then finding out whether behaviour in practice matches up to it across the input space. Tracing means first recording the internal details of a computation, then examining what is recorded to gain insight, to check hypotheses or to locate faults. Although we have emphasised the motivation of eliminating errors, tools for testing and tracing can often be useful even for programmers who rarely make mistakes. For example, the increased understanding gained by testing and tracing can lead to improved solutions and better documentation.

In these lecture notes we concentrate on QuickCheck, a tool for testing Haskell programs, and Hat, a tool for tracing them. Each tool is useful in its own right but, as we shall see, they are even more useful in combination: testing using

[*]Chalmers University of Technology, Sweden
[†]University of York, United Kingdom

QuickCheck can identify failing cases, tracing using Hat can reveal the causes of failure.

Section 2 explains what QuickCheck is and how to use it. Section 3 similarly explains Hat. Section 4 shows how to use QuickCheck and Hat in combination. Section 5 outlines a much larger application than those of earlier sections, and explains some techniques for testing and tracing more complex programs. Section 6 details almost twenty practical exercises.

We assume access to on-line source programs and data accompanying these notes, and to a Haskell system with QuickCheck *and* Hat *installed.*

In this preliminary version of these notes, there is no discussion of related work, and no list of references. We hope to make good these omissions in the final published version.

2 Testing Programs with QuickCheck

In this section we give a short introduction to QuickCheck[1], a system for specifying and randomly testing properties of Haskell programs.

2.1 Testing and Testable Specifications

Testing is by far the most commonly used approach to ensuring software quality. It is also very labour intensive, accounting for up to 50% of the cost of software development. Despite anecdotal evidence that functional programs require somewhat less testing ('Once it type-checks, it usually works'), in practice it is still a major part of functional program development.

The cost of testing motivates efforts to automate it, wholly or partly. Automatic testing tools enable the programmer to complete testing in a shorter time, or to test more thoroughly in the available time, and they make it easy to repeat tests after each modification to a program.

Functional programs are well suited to automatic testing. It is generally accepted that pure functions are much easier to test than side-effecting ones, because one need not be concerned with a state before and after execution. In an imperative language, even if whole programs are often pure functions from input to output, the procedures from which they are built are usually not. Thus relatively large units must be tested at a time. In a functional language, pure functions abound (in Haskell, only computations in the IO monad are hard to test), and so testing can be done at a fine grain.

A testing tool must be able to determine whether a test is passed or failed; the human tester must supply a passing criterion that can be automatically checked. We use formal specifications for this purpose. QuickCheck comes with a simple

[1] Available from `http://www.cs.chalmers.se/~rjmh/QuickCheck/`

domain-specific language of *testable specifications* which the tester uses to define expected properties of the functions under test. It is then checked that the properties hold in a large number of cases. We call these testable specifications *properties*. The specification language is embedded in Haskell using the class system. This means that properties are just normal Haskell functions which can be understood by any Haskell compiler or interpreter. Property declarations are either written in the same module as the functions they test, or they can be written in a separate Haskell module, importing the functions they test, which is the preferred way we use in these notes. Either way, properties serve also as checkable documentation of the behaviour of the code.

A testing tool must also be able to generate test cases automatically. QuickCheck uses the simplest method, *random testing*, which competes surprisingly favourably with systematic methods in practice. However, it is meaningless to talk about random testing without discussing the distribution of test data. Random testing is most effective when the distribution of test data follows that of actual data, but when testing reusable code units as opposed to whole systems this is not possible, since the distribution of actual data in all subsequent reuses is not known. A uniform distribution is often used instead, but for data drawn from infinite sets this is not even meaningful! In QuickCheck, distribution is put under the human tester's control, by defining a *test data generation language* (also embedded in Haskell), and a way to observe the distribution of test cases. By programming a suitable generator, the tester can not only control the distribution of test cases, but also ensure that they satisfy arbitrarily complex invariants.

2.2 Defining Properties

As a first example, we are going to test the standard function `reverse` which reverses a list. This function satisfies a number of useful laws, such as:

$$
\begin{aligned}
\texttt{reverse [x]} &= \texttt{[x]} \\
\texttt{reverse (xs++ys)} &= \texttt{reverse ys++reverse xs} \\
\texttt{reverse (reverse xs)} &= \texttt{xs}
\end{aligned}
$$

(In fact, the first two of these characterise `reverse` uniquely.)

Note that these laws hold only for *finite, total* values. In all QuickCheck properties, unless specifically stated otherwise, we quantify over completely defined finite values.

In order to check such laws using QuickCheck, we represent them as Haskell functions. To represent the second law for example, we write:

```
prop_RevApp xs ys = reverse (xs++ys) == reverse ys ++ reverse xs
```

We use the convention that property function names always start with the prefix `prop_`. Nevertheless, `prop_RevApp` is still a normal Haskell function. If this function returns `True` for every possible argument, then the properties hold. However, in order for us to actually *test* this property, we need to know on what *type* to test it! We do not know this yet since the function `prop_RevApp` has a polymorphic type. Thus the programmer must specify a fixed type at which the property is to be tested. So we simply give a type signature for each property, for example:

```
prop_RevApp :: [Int] -> [Int] -> Bool
prop_RevApp xs ys = reverse (xs++ys) == reverse ys ++ reverse xs
```

Lastly, to access the library of functions that we can use to define and test properties, we have to include the QuickCheck module. Thus we add

```
import QuickCheck2
```

at the top of our module. `QuickCheck2` is a special version of QuickCheck with a facility to interoperate with the tracing tools (explained in Section 4).

Now we are ready to test the above property! We load our module into a Haskell system (we use GHCi in these notes), and call for example:

```
Main> quickCheck prop_RevApp
OK, passed 100 successful tests.
```

The function `quickCheck` takes a property as an argument and applies it to a large number of randomly generated arguments — 100 by default — reporting "OK" if the result is `True` in every case.

If the law fails, then `quickCheck` reports the counter-example. For example, if we mistakenly define

```
prop_RevApp :: [Int] -> [Int] -> Bool
prop_RevApp xs ys = reverse (xs++ys) == reverse xs++reverse ys
```

then checking the property might produce

```
Main> quickCheck prop_RevApp
Falsifiable, after 1 successful tests:
[2]
[-2,1]
```

where the counter model can be reconstructed by taking `[2]` for `xs` (the first argument of the property), and `[-2,1]` for `ys` (the second argument). We will later see how we can use tracing to see what actually happens with the functions in a property when running it on a failing test case.

```
sort :: Ord a => [a] -> [a]
sort []      =   []
sort (x:xs)  =   insert x (sort xs)
insert :: Ord a => a -> [a] -> [a]
insert x []      =   [x]
insert x (y:ys)  =   if x <= y then x : y : ys
                     else y : insert x ys
```

Figure 1: An insertion-sort program.

2.3 Introducing Helper Functions

Take a look at the insertion sort implementation in Figure 1. Let us design a test suite to test the functions in that implementation.

First, we test the function `sort`. A cheap way of testing a new implementation of a sorting algorithm is to use an existing implementation which we trust. We can thus specify:

```
import qualified List

prop_SortIsSort :: [Int] -> Bool
prop_SortIsSort xs = sort xs == List.sort xs
```

Here, we say that our function `sort` produces the same result as the standard function `sort` which comes from the List module in Haskell.

But what if we do not trust the implementation of the standard `sort` either? Then, we have to come up with properties that say when exactly a function is a sorting function. A function sorts if and only if: (1) the output is ordered, and (2) the output has exactly the same elements as the input.

To specify the first property, we need to define a helper function `ordered` which checks that a given list is ordered.

```
ordered :: Ord a => [a] -> Bool
ordered []       = True
ordered [x]      = True
ordered (x:y:xs) = (x <= y) && ordered (y:xs)
```

Then, the orderedness property for `sort` is easy to define:

```
prop_SortOrdered :: [Int] -> Bool
prop_SortOrdered xs = ordered (sort xs)
```

For the second property, we also need to define a helper function, namely one that checks if two lists have the same elements.

```
sameElems :: Eq a => [a] -> [a] -> Bool
[]     `sameElems` [] = True
(x:xs) `sameElems` ys = (x `elem` ys) &&
                            (xs `sameElems` (ys \\ [x]))
_      `sameElems` _  = False
```

The second sorting property is then rather easy to define too:

```
prop_SortSameElems :: [Int] -> Bool
prop_SortSameElems xs = sort xs `sameElems` xs
```

2.4 Conditional Properties and Quantification

It is good to define and test properties for many functions involved in an implementation rather than just, say, the top-level functions. Applying such fine grained testing makes it more likely to find mistakes.

So, let us think about the properties of the function `insert`, and assume we do not have another implementation of it which we trust. The two properties that should hold for a correct insert function are: (1) if the argument list is ordered, so should the result list be, and (2) the elements in the result list should be the same as the elements in the argument list plus the first argument.

We can specifying the second property in a similar way to the property for `sort` defined earlier:

```
prop_InsertSameElems :: Int -> [Int] -> Bool
prop_InsertSameElems x xs = insert x xs `sameElems` (x:xs)
```

However, if we try to express the first property, we immediately run into problems. It is not just a simple equational property, but a conditional property. QuickCheck provides an *implication* combinator, written `==>`, to represent such conditional laws. Using implication, the first property for the insertion function can be expressed as:

```
prop_InsertOrdered :: Int -> [Int] -> Property
prop_InsertOrdered x xs = ordered xs ==> ordered (insert x xs)
```

Testing such a property works a little differently. Instead of checking the property for 100 random test cases, we try checking it for 100 test cases *satisfying the condition*. If a candidate test case does not satisfy the condition, it is discarded, and a new test case is tried. So, when a property with an implication successfully passes 100 test cases, we are sure that all of them actually satisfied the left hand side of the implication.

Note that the result type of a conditional property is changed from `Bool` to `Property`. This is because the testing semantics is different for conditional laws.

Checking prop_InsertOrdered succeeds as usual, but sometimes, checking a conditional property produces an output like this:

```
Arguments exhausted after 64 tests.
```

If the precondition of a law is seldom satisfied, then we might generate many test cases without finding any where it holds. In such cases it is hopeless to search for 100 cases in which the precondition holds. Rather than allow test case generation to run forever, we generate only a limited number of candidate test cases (the default is 1000). If we do not find 100 valid test cases among those candidates, then we simply report the number of successful tests we were able to perform. In the example, we know that the law passed the test in 64 cases. It is then up to the programmer to decide whether this is enough, or whether it should be tested more thoroughly.

2.5 Monitoring Test Data

Perhaps it seems that the implication operator has solved our problems, and that we are happy with the property prop_InsertOrdered. But have we really tested the law for insert thoroughly enough to establish its credibility?

Let us take a look at the *distribution* of test cases in the 100 tests that we performed on prop_InsertOrdered, by modifying prop_InsertOrdered as follows:

```
prop_InsertOrdered :: Int -> [Int] -> Property
prop_InsertOrdered x xs = ordered xs ==>
                            classify (null xs) "trivial" $
                              ordered (insert x xs)
```

Checking the law now produces the message

```
OK, passed 100 successful tests (43% trivial).
```

The classify combinator does not change the meaning of a law, but it classifies some of the test cases. In this case those where xs is the empty list were classified as “trivial”. Thus we see that a large proportion of the test cases only tested insertion into an empty list.

We can do more than just labelling some test cases with strings. The combinator collect gathers all values that are passed to it, and prints out a histogram of these values. For example, if we write:

```
prop_InsertOrdered :: Int -> [Int] -> Property
prop_InsertOrdered x xs = ordered xs ==>
                            collect (length xs) $
                              ordered (insert x xs)
```

we might get as a result:

```
OK, passed 100 successful tests.
40% 0.
31% 1.
19% 2.
8% 3.
2% 4.
```

So we see that only 29 (=19+8+2) cases tested insertion into a list with more than one element. While this is enough to provide fairly strong evidence that the property holds, it is worrying that very short lists dominate the test cases so strongly. After all, it is easy to define an erroneous version of `insert` which nevertheless works for lists with at most one element.

The reason for this behaviour, of course, is that the precondition `ordered xs` skews the distribution of test cases towards short lists. Every generated list of length 0 or 1 is ordered, but only 50% of the lists of length 2 are ordered, and not even 1% of all lists of length 5 are ordered. Thus test cases with longer lists are more likely to be rejected by the precondition. There is a risk of this kind of problem every time we use conditional laws, so it is always important to investigate the proportion of trivial cases among those actually tested.

It is comforting to be able to monitor the test data, and change the definition of our properties if we find the distribution too biased. The best solution in this case is to replace the condition with a *custom test data generator* for ordered lists. We write

```
prop_InsertOrdered :: Int -> Property
prop_InsertOrdered x = forAll orderedList $ \xs ->
                         ordered (insert x xs)
```

which specifies that values for `xs` should be generated by the test data generator `orderedList`. This test data generator can make sure that the lists in question are ordered and have a more reasonable distribution. Checking the law now gives "`OK, passed 100 successful tests`", as we would expect. QuickCheck provides support for the programmer to define his or her own test data generators, with control over the distribution of test data, which we will look at next.

2.6 Test Data Generation

So far, we have not said anything about how test data is generated. The way we generate random test data of course depends on the type. Therefore, QuickCheck provides a type class `Arbitrary`, of which a type is an instance if we know how to generate arbitrary elements in it.

```
class Arbitrary a where
  arbitrary :: Gen a
```

`Gen a` is an abstract type representing a generator for type `a`. The programmer can either use the generators built in to QuickCheck as instances of this class, or supply a custom generator using the `forAll` combinator, which we saw in the previous section.

Since we will treat `Gen` as an *abstract* type, we define a number of primitive functions to access its functionality. The first one is:

```
choose :: (Int, Int) -> Gen Int
```

This function chooses a random integer in an interval with a uniform distribution. We program other generators in terms of it.

We also need combinators to build complex generators from simpler ones; to do so, we declare `Gen` to be an instance of Haskell's class `Monad`. This involves implementing the methods of the `Monad` class:

```
return :: a -> Gen a
(>>=)  :: Gen a -> (a -> Gen b) -> Gen b
```

The first method constructs a constant generator, i.e. `return x` always generates the same value `x`; the second method is the monadic sequencing operator, i.e. `g >>= k` first generates an `a` using `g`, and passes it to `k` to generate a `b`.

Monads are heavily used in Haskell, and there are many useful overloaded standard functions which work with *any* monad; there is even syntactic sugar for monadic sequencing (the `do` notation). By making generators into a monad, we are able to use all of these features to construct them.

Defining generators for many types is now straightforward. As examples, we give generators for integers and pairs:

```
instance Arbitrary Int where
  arbitrary = choose (-20, 20)

instance (Arbitrary a, Arbitrary b) => Arbitrary (a,b) where
  arbitrary =
    do a <- arbitrary
       b <- arbitrary
       return (a,b)
```

QuickCheck contains such declarations for most of Haskell's predefined types.

Looking at the instance of pairs above, we see a pattern that occurs frequently. In fact, Haskell provides a standard operator `liftM2` for this pattern. An alternative way of writing the instance for pairs is:

```
instance (Arbitrary a, Arbitrary b) =>
                        Arbitrary (a,b) where
  arbitrary = liftM2 (,) arbitrary arbitrary
```

We will use this programming style later on too.

Since we define test data generation via an instance of class `Arbitrary` for each type, then we must rely on the user to provide instances for user-defined types.

Instead of producing generators automatically, we provide combinators to enable a programmer to define his own generators easily. The simplest, called `oneof`, just makes a choice among a list of alternative generators with a uniform distribution. For example, a suitable generator for booleans could be defined by:

```
instance Arbitrary Bool where
  arbitrary = oneof [return False, return True]
```

As another example, we could generate arbitrary lists using

```
instance Arbitrary a => Arbitrary [a] where
  arbitrary = oneof
    [return [], liftM2 (:) arbitrary arbitrary]
```

where we use `liftM2` to apply the cons operator `(:)` to an arbitrary head and tail. However, this definition is not really satisfactory, since it produces lists with an average length of one element. We can adjust the average length of list produced by using `frequency` instead, which allows us to specify the frequency with which each alternative is chosen. We define

```
instance Arbitrary a => Arbitrary [a] where
  arbitrary = frequency
    [ (1, return [])
    , (4, liftM2 (:) arbitrary arbitrary)
    ]
```

to choose the cons case four times as often as the nil case, leading to an average list length of four elements.

2.7 Generators with Size

Suppose we have the following datatype of binary trees and the operations `size` and `flatten`:

```
data Tree a = Leaf a
            | Branch (Tree a) (Tree a)
            deriving ( Show )

size :: Tree a -> Int
size (Leaf a)     = 1
size (Branch s t) = size s + size t

flatten :: Tree a -> [a]
```

```
flatten (Leaf a)     = [a]
flatten (Branch s t) = flatten s ++ flatten s
```

An obvious property we would like to hold is that the length of a list which is a flattened tree should be the same as the size of the tree. Here it is:

```
prop_SizeFlatten :: Tree Int -> Bool
prop_SizeFlatten t = length (flatten t) == size t
```

However, to test this property in QuickCheck, we need to define our own test data generator for trees. Here is our first try:

```
instance Arbitrary a => Arbitrary (Tree a) where
  arbitrary = frequency                              -- wrong!
    [ (1, liftM  Leaf   arbitrary)
    , (2, liftM2 Branch arbitrary arbitrary)
    ]
```

We want to avoid choosing a `Leaf` too often (to avoid small trees), hence the use of `frequency`.

However, *this definition only has a 50% chance of terminating!* The reason is that for the generation of a `Branch` to terminate, *two* recursive generations must terminate. If the first few recursions choose `Branch`es, then generation terminates only if very many recursive generations all terminate, and the chance of this is small. Even when generation terminates, the generated test data is sometimes very large. We want to avoid this: since we perform a large number of tests, we want each test to be small and quick.

Our solution is to limit the *size* of generated test data. But the notion of a size is hard even to define in general for an arbitrary recursive datatype (which may include function types anywhere). We therefore give the responsibility for limiting sizes to the programmer defining the test data generator. We define a new combinator

```
sized :: (Int -> Gen a) -> Gen a
```

which the programmer can use to access the size bound: `sized` generates an `a` by passing the current size bound to its parameter. It is then up to the programmer to interpret the size bound in some reasonable way during test data generation. For example, we might generate binary trees using

```
instance Arbitrary a => Arbitrary (Tree a) where
  arbitrary = sized arbTree
    where
      arbTree n = frequency $
        [ (1, liftM  Leaf   arbitrary)
        ] ++
```

```
  [ (4, liftM2 Branch arbTree2 arbTree2)
  | n > 0
  ]
 where
  arbTree2 = arbTree (n `div` 2)
```

With this definition, the size bound limits the number of nodes in the generated trees, which is quite reasonable.

We can now test the property about `size` and `flatten`:

```
Main> quickCheck prop_SizeFlatten
Falsifiable, after 3 successful tests:
Branch (Branch (Leaf 0) (Leaf 3)) (Leaf 1)
```

Can you spot the bug?

Now that we have introduced the notion of a size bound, we can use it sensibly in the generators for other types such as integers (with absolute value bounded by the size) and lists (with length bounded by the size). So the definitions we presented earlier need to be modified accordingly. For example, to generate arbitrary integers, QuickCheck really uses the following default generator:

```
instance Arbitrary Int where
  arbitrary = sized (\n -> choose (-n, n))
```

We stress that the size bound is simply an extra, global parameter which every test data generator may access; every use of `sized` sees the same bound. We do *not* attempt to 'divide the size bound among the generators', so that for example a longer generated list would have smaller elements, keeping the overall size of the structure the same. The reason is that we wish to avoid correlations between the sizes of different parts of the test data, which might distort the test results.

We do vary the size between different test cases: we begin testing each property on small test cases, and then gradually increase the size bound as testing progresses. This makes for a greater variety of test cases, which both makes testing more effective, and improves our chances of finding enough test cases satisfying the precondition of conditional properties. It also makes it more likely that we will find a small counter example to a property, if there is one.

3 Tracing Programs with Hat

In this section we give a basic introduction to the Hat tools[2] for tracing Haskell programs.

[2] Available from `http://www.cs.york.ac.uk/fp/hat/`.

3.1 Traces and Tracing Tools

Without tracing, programs are like black boxes. We see only their input-output behaviour. To understand this behaviour our only resort is the static text of a program, and it is often not enough. We should like to see the component functions at work, the arguments they are given and the results they return. We should like to see how their various applications came about in the first place. The purpose of tracing tools like Hat is to give us access to just this kind of information that is otherwise invisible.

For more than 20 years researchers have been proposing ways to build tracers for lazy higher-order functional languages. Sadly, most of their work has never been widely used, because it has been done for locally used implementations of local dialect languages. A design-goal for Haskell was to solve the language-diversity problem. The problem will always persist to some degree, but Haskell is the nearest thing there is to a standard lazy functional language. Now the challenge is to build an effective tracer for it, depending as little as possible on the machinery of specific compilers or interpreters.

Tracers for conventional languages enable the user to step through a computation, stopping at selected points to examine variables. This approach is not so helpful for a lazy functional language where the order of evaluation is not the order of appearance in a source program, and in mid computation variables may be bound to complex-looking unevaluated expressions. Like some if its predecessors, Hat is instead based on derivation graphs for complete computations. This representation liberates us from the time-arrow of execution. For example, all arguments and results can be shown in the most fully evaluated form that they ever attain. The established name for this technique is *strictification*, but don't let the name mislead you: we do *not* force functions in the traced program into strict variants; all the lazy behaviour of the normally-executed program is preserved.

When we compile a program for tracing it is automatically transformed by a preprocessor called `hat-trans` into a self-tracing variant. The transformed program is still in Haskell, not some private intermediate language, so that Hat can be ported between compilers. When we run the transformed program, in addition to the I/O behaviour of the original, it generates a graph-structured trace of evaluation called a *redex trail*. The trace is written to file as the computation proceeds. Trace files contain a lot of detail and they can be very large — tens or even hundreds of megabytes. So we should not be surprised if traced programs run much less quickly than untraced ones, and we shall need tools to select and present the key fragments of traces in source-level terms.

There are several Hat tools for examining traces, but in these notes we shall look at the two used most: `hat-trail` and `hat-observe`. As a small illustrative application we take sorting the letters of the word 'program' using insertion sort. That is, to the definitions of Figure 1 we now add

```
main = putStrLn (sort "program")
```

to make a source program `Insort.hs`. At first we shall trace the working program; later we shall look at a variant `BadInsort.hs` with faults deliberately introduced.

3.2 Hat Compilation and Execution

To use Hat, we first compile the program to be traced, giving the `-hat` option to `hmake`:

```
$ hmake -hat Insort
hat-trans Insort.hs
Wrote TInsort.hs
ghc -package hat -c -o TInsort.o TInsort.hs
ghc -package hat -o Insort TInsort.o
```

A program compiled for tracing can be executed just as if it had been compiled normally.

```
$ Insort
agmoprr
```

The main difference from untraced execution is that as `Insort` runs it records a detailed trace of its computation in a file `Insort.hat`. The trace is a graph of program expressions encoded in a custom binary format. Two further files `Insort.hat.output` and `Insort.hat.bridge` record the output and associated references to the trace file. Trace files do not include program sources, but they do include references to program sources, so *modifying source files may invalidate existing traces.*

3.3 Hat-trail: Basics

After we have run a program compiled for tracing, creating a trace file, we can use Hat tools to examine the trace. The first such tool we shall look at is hat-trail. The idea of hat-trail is to answer the question 'Where did that come from?' in relation to the values, expressions, outputs and error messages that occur in a traced computation. The immediate answer will be a parent application or name. More specifically:

- *errors*: the application or name being reduced when the error occurred (eg. `head []` might be the parent of a pattern-match failure);
- *outputs*: the monadic action that caused the output (eg. `putStr "Hello world"` might the parent of a section of output text);

- *non-value expressions*: the application or name whose defining body contains the expression of which the child is an instance (eg. `insert 6 [3]` might be the parent of `insert 6 []`);
- *values*: as for non-value expressions, or the application of a predefined function with the child as result (eg. `[1,2]++[3,4]` might be the parent of `[1,2,3,4]`).

Parent expressions, and their subexpressions, may in turn have parents of their own. The tool is called hat-trail because it displays trails of ancestral redexes, tracing effects back to their causes.

We can think of redex trails as a generalisation of the stack back-traces for conventional languages, showing the dynamically enclosing call-chain leading to a computational event. Because of lazy evaluation, the call-stack may not actually exist when the event occurs, but there is sufficient information in a Hat trace to reconstruct it. When we are tracing the origins of an application using hat-trail we have five choices: we can trace the ancestry not only of (1) the application itself, as in a stack back-trace, but also of (2) the function, or (3) an argument — or indeed, any subexpression of these. We can also ask to see a relevant extract of the source program: either (4) the expression of which the application is an instance, or (5) the definition of the function being applied.

Hat-trail sessions and requests We can start a hat-trail session from a shell command line, or from within existing sessions of hat tools. If we give the shell command

```
$ hat-trail Insort
```

a new window appears with an upper part headed `Output` and a lower part headed `Trail`:

```
Output: -------------------------------------------------------
agmoprr\n

Trail: ------ hat-trail 2.00 (:h for help, :q to quit) -------
```

The line of output is highlighted[3] because it is the current selection.

Requests in `hat-trail` are of two kinds. Some are single key presses with an immediate response; others are command-lines starting with a colon and only acted upon when completed by keying return. A basic repertoire of single-key requests is:

[3]In these notes, highlighted text or expressions are shown boxed; the Hat tools actually use colour for highlighting.

return	add to the trail the parent expression of the current selection
backspace	remove the last addition to the trail display
arrow keys	select (a) parts of the output generated by different actions, or (b) subexpressions of expressions already on display

And a basic repertoire of command-line requests is:

`:source`	show the source expression of which the current selection is an instance
`:quit`	finish this hat-trail session

It is enough to give initial letters, `:s` or `:q`, rather than `:source` or `:quit`.

Some insertion-sort trails To trace the output from the `Insort` computation, we key return and the `Trail` part of the display becomes:

```
Trail: ----------------------- Insort.hs line: 10 col: 8 ------
<- putStrLn "agmoprr"
```

The source reference is to the corresponding application of `putStrLn` in the program. If we give the command `:s` at this point, a separate source window shows the relevant extract of the program. We can only do two things with a source window: (1) look at it; (2) close it. Tracing with Hat does *not* involve annotating or otherwise modifying program sources.

Back to the `Trail` display. We key return again:

```
Trail: ----------------------- Insort.hs line: 10 col: 1 ------
<-putStrLn "agmoprr"
<-main
```

That is, the line of output was produced by an application of `putStrLn` occurring in the body of `main`.

So far, so good; but what about the sorting? How do we see where `putStr`'s string argument `"agmoprr"` came from? By making that string the current selection and requesting its parent:

backspace	(removes `main`),
right-arrow	(selects `putStrLn`),
right-arrow	(selects `"agmoprr"`),
return	(requests parent expression)

```
Trail: ----------------------- Insort.hs line: 7 col: 19 -----
<- putStrLn "agmoprr"
<- insert 'p' "agmorr" | if False
```

The | symbol here is a separator between a function application and the trace of a conditional or case expression that was evaluated in its body; guards are shown in a similar way. The string `"agmoprr"` is the result of inserting `'p'`, the head of the string `"program"`, into the recursively sorted tail. More specifically, the string was computed in the else-branch of the conditional by which `insert` is defined in the recursive case (because `'p' <= 'a'` is `False`).

And so we could continue. For example, following the trail of string arguments:

```
<- insert 'p' "agmorr" | if False
<- insert 'r' "agmor" | if False
<- insert 'o' "agmr" | if False
<- insert 'g' "amr" | if False
<- insert 'r' "am" | if False
<- insert 'a' "m" | if True
<- insert 'm' []
```

But let's leave hat-trail for now.

:quit

3.4 Hat-observe: Basics

The idea of hat-observe is to answer the question 'To which arguments, if any, was that applied, and with what results?', mainly in relation to a top-level function. Answers take the form of a list of equational observations, showing for each application of the function to distinct arguments what result was computed. In this way hat-observe can present all the needed parts of an extensional specification for each function defined in a program. We also have the option to limit observations to particular patterns of arguments or results, or to particular application contexts.

Hat-observe sessions and requests We can start a hat-observe session from a shell command line, or from within an existing session of a Hat tool.

```
$ hat-observe Insort

        hat-observe 2.00  (:h for help, :q to quit)

hat-observe>
```

In comparison with hat-trail, there is more emphasis on command-lines in hat-observe, and the main user interface is a prompt-request-response cycle. Requests are of two kinds. Some are observation queries in the form of application patterns: the simplest observation query is just the name of a top-level function. Others are command-lines, starting with a colon, similar to those of hat-trail. A basic repertoire of command-line requests is

`:info`	list the names of functions and other defined values that can be observed, with application counts
`:quit`	finish this hat-observe session

Again it is enough to give the initial letters, `:i` or `:q`.

Some insertion-sort observations We often begin a hat-observe session with an `:info` request, followed by initial observation of central functions.

```
hat-observe> :info
19 <=         21 insert     1 main        1 putStrLn    8 sort
hat-observe> sort
1 sort "program" = "agmoprr"
2 sort "rogram" = "agmorr"
3 sort "ogram" = "agmor"
4 sort "gram" = "agmr"
5 sort "ram" = "amr"
6 sort "am" = "am"
7 sort "m" = "m"
8 sort [] = []
```

Here the number of observations is small. Larger collections of observations are presented in blocks of ten (by default).

```
hat-observe> <=
1 'a' <= 'm' = True
2 'r' <= 'a' = False
3 'g' <= 'a' = False
4 'o' <= 'a' = False
5 'p' <= 'a' = False
6 'r' <= 'm' = False
7 'g' <= 'm' = True
8 'o' <= 'g' = False
9 'r' <= 'g' = False
10 'p' <= 'g' = False
--more-->
```

If we key return in response to `--more-->`, the next block of observations appears. Alternatively, we can make requests in the colon-command family. Any other line of input cuts short the list of reported observations in favour of a fresh `hat-observe>` prompt.

```
--more--> n
hat-observe>
```

Observing restricted patterns of applications Viewing a block at a time is not the only way of handling what may be a large number of applications. We can also restrict observations to applications in which specific patterns of values occur as arguments or result, or to applications in a specific context. The full syntax for observation queries is

*identifier pattern** [= *pattern*] [`in` *identifier*]

where the * indicates that there can be zero or more occurrences of an argument pattern and the [...] indicate that the result pattern and context are optional. Patterns in observation queries are simplified versions of constructor patterns with _ as the only variable. Some examples for the `Insort` computation:

```
hat-observe> insert 'g' _
1 insert 'g' "amr" = "agmr"
2 insert 'g' "mr" = "gmr"
hat-observe> insert _ _ = [_]
1 insert 'm' [] = "m"
2 insert 'r' [] = "r"
hat-observe> sort in main
1 sort "program" = "agmoprr"
hat-observe> sort in sort
1 sort "rogram" = "agmorr"
2 sort "ogram" = "agmor"
3 sort "gram" = "agmr"
4 sort "ram" = "amr"
5 sort "am" = "am"
6 sort "m" = "m"
7 sort [] = []
```

Enough on hat-observe for now.

```
hat-observe> :quit
```

3.5 Tracing Faulty Programs

We have seen so far some of the ways in which Hat tools can be used to trace a correctly working program. But a common and intended use for Hat is to trace a faulty program with the aim of locating the source of the faults. A faulty

```
sort :: Ord a => [a] -> [a]
-- FAULT (1): missing equation for [] argument
sort (x:xs)  =  insert x (sort xs)
insert :: Ord a => a -> [a] -> [a]
insert x []      =  [x]
insert x (y:ys)  =  if x <= y
                    -- FAULT (2): y missing from result
                    then x : ys
                    -- FAULT (3): recursive call is same
                    else y : insert x (y:ys)
main  =  putStrLn (sort "program")
```

Figure 2: `BadInsort.hs`, a faulty version of the insertion-sort program.

computation has one of three outcomes: (1) termination with a run-time error, or (2) termination with incorrect output, or (3) non-termination.

A variant of `Insort` given in Figure 2 contains three deliberate mistakes, each of which alone would cause a different kind of fault, as indicated by comments. In the following sections we shall apply the Hat tools to examine the faulty program, as if we didn't know in advance where the mistakes were.

Tracing a Run-time Error

We compile the faulty program for tracing, then run it:

```
$ hmake -hat BadInsort
...
$ BadInsort
No match in pattern.
```

Using hat-trail We can easily trace the immediate cause of the error message, which hat-trail displays as a starting point. We key return once to see the erroneous application, then again to see its parent application:

```
$ hat-trail BadInsort

Error: -------------------------------------------------------
No match in pattern.
Trail: ------------------- BadInsort.hs line: 3 col: 25 -------
<- sort []
<- sort "m"
```

This information can be supplemented by reference to the source program. With `sort []` selected, we can give the `:source` command to see the site of the offending application in the recursive equation for `sort`. If necessary we could trace the ancestry of the `[]` argument or the `sort` application.

Using hat-observe Although hat-trail is usually our first resort for tracing run-time errors, it is instructive to see what happens if instead we try using hat-observe.

```
$ hat-observe BadInsort

        hat-observe 2.00 (:h for help, :q to quit)

hat-observe> :info
7+0 insert    1 main        1 putStrLn    1+7 sort
```

What do the M+N counts for `insert` and `sort` mean? M is the number of applications that never got beyond a pattern-matching stage involving evaluation of arguments; N is the number of applications that were actually reduced to an instance of the function body. Applications are only counted at all if their results were demanded during the computation. Where a count is shown as a single number, it is the count N of applications actually reduced, and $M = 0$.

In the `BadInsort` computation, we see there are fewer observations of `insert` than there were in the correct `Insort` computation, and no observations at all of `<=`. How can that be? What is happening to ordered insertion?

```
hat-observe> insert
1 insert 'p' _|_ = _|_
2 insert 'r' _|_ = _|_
3 insert 'o' _|_ = _|_
4 insert 'g' _|_ = _|_
5 insert 'a' _|_ = _|_
6 insert 'm' _|_ = _|_
```

The symbol `_|_` here is an ASCII approximation to $\perp$ and indicates an undefined value. Reading the character arguments vertically `"program"` seems to be misspelt: is there an observation missing between `4` and `5`? There are in fact two separate applications `insert 'r' _|_ = _|_`, but duplicate observations are not listed (by default).

The `insert` observations explain the fall in application counts. In all the observed applications, the list arguments are undefined. So neither of the defining equations for `insert` is ever matched, there are no `<=` comparisons (as these occur only in the right-hand side of the second equation) and of course no recursive calls.

Why are the `insert` arguments undefined? They should be the results of `sort` applications.

```
hat-observe> sort
1 sort "program" = _|_
2 sort "rogram" = _|_
3 sort "ogram" = _|_
4 sort "gram" = _|_
5 sort "ram" = _|_
6 sort "am" = _|_
7 sort "m" = _|_
8 sort [] = _|_
```

Though all the `sort` results are `_|_`, the reason is not the same in every case. Observations `1` to `7` show applications of `sort` that reduced to applications of `insert`, with the `_|_` results already observed[4]. Observation `8` is an application that does not reduce at all; it also points us to the error.

Tracing a Non-terminating Computation

Suppose we correct the first fault, by restoring the equation

```
sort [] = []
```

and recompile. Now the result of running `BadInsort` is a non-terminating computation, with an infinite string `aaaaaaa...` as output. It seems that `BadInsort` has entered an infinite loop. The computation can be interrupted[5] by keying control-C.

```
$ BadInsort
Program interrupted. (^C)
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa$
```

Using hat-trail The initial hat-trail display is:

```
Error: -------------------------------------------------------
Program interrupted. (^C)
Output: ------------------------------------------------------
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa...
```

We have a choice. We can follow the trail back either from the point of interruption (the initial selection) or from the output (reached by down-arrow). In

[4]This insight requires knowledge of the program beyond the listed applications in hat-observe: for example, it could be obtained by a linked use of hat-trail (see Section 3.6)

[5]When non-termination is suspected, interrupt as quickly as possible to avoid working with very large traces.

this case, it makes little difference[6]; either way we end up examining the endless list of `'a'`s. Let's select the output:

```
Output: -------------------------------------------------------
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa...
Trail: ----------------- BadInsort.hs line: 7 col: 19 --------
<- putStrLn "aaaaaaaa..."
<- insert 'p' ('a':_) | if False
```

Notice two further features of expression display:

- the ellipsis `...` in the string argument to `putStrLn` indicates the tail-end of a long string that has been pruned from the display;
- the symbol `_` in the list argument to `insert` indicates an expression that was never evaluated.

The parent application `insert 'p' ('a':_) | if False` gives us several important clues. It tells us that in the else-branch of the recursive case in the definition of `insert` the argument's head (here `'a'`) is duplicated endlessly to generate the result without ever demanding the argument's tail (shown only as `_`). This tells us enough to discover the fault if we didn't already know it.

Using hat-observe Once again, let us also see what happens if we use hat-observe.

```
hat-observe> :info
78 <=        1+83 insert   1 main        1 putStrLn    8 sort
```

The high counts for `<=` and `insert` give us a strong clue: as `<=` is primitively defined, we immediately suspect `insert`.

```
hat-observe> insert
1 insert 'p' ('a':_) = "aaaaaaaaaa..."
2 insert 'r' ('a':_) = 'a':_
3 insert 'o' ('a':_) = 'a':_
4 insert 'g' ('a':_) = 'a':_
5 insert 'a' "m" = "a"
6 insert 'm' [] = "m"
searching ...  (^C to interrupt)
{Interrupted}
```

[6]However, the trace from point of interruption depends on the timing of the interrupt.

Many more observations would eventually be reported because hat-observe lists each observation that is distinct from those listed previously. When the computation is interrupted there are many different applications of the form `insert 'p' ('a':_)` in progress, each with results evaluated to a different extent.

But observation 1 is enough. As the tail of the argument is unevaluated, the result would be the same whatever the tail. For example, it could be `[]`; so we know `insert 'p' "a" = "aaaa..."`. This specific and simple failing case directs us to the fault in the definition of `insert`.

Tracing Wrong Output

We correct the recursive call from `insert x (y:ys)` to `insert x ys`, recompile, then execute.

```
$ BadInsort
agop
```

Using hat-observe Once again, we could reach first for hat-trail to trace the fault, but the availability of a well-defined (but wrong) result also suggests a possible starting point in hat-observe:

```
hat-observe> insert _ _ = "agop"
1 insert 'p' "agor" = "agop"
```

Somehow, insertion loses the final element `'r'`. We should like to see more details of how this result is obtained — the relevant recursive calls, for example:

```
hat-observe> insert 'p' _ in insert
1 insert 'p' "gor" = "gop"
2 insert 'p' "or" = "op"
3 insert 'p' "r" = "p"
```

Observation 3 makes it easy to discover the fault by inspection.

Using hat-trail If we instead use hat-trail, the same application could be reached as follows. We first request the parent of the output; unsurprisingly it is `putStrLn "agop"`. We then request the parent of the string argument `"agop"`:

```
Output: -------------------------------------------------------
agop\n

Trail: ----------------- BadInsort.hs line: 10 col: 26 -------
```

```
<- putStrLn "agop"
<- insert 'p' "agor" | if False
```

As in hat-observe, we see the `insert` application that loses the character `'r'`.

3.6 Linked use of hat-observe and hat-trail

Although we have so far made use of hat-observe and hat-trail separately, each can be applied within the other using the following commands:

`:o`	in hat-trail, with an application of f as the current selection, starts a new hat-observe window listing all the traced applications of f
`:t` N	in hat-observe, following the display of a list of at least N applications, starts a new hat-trail window with the Nth application as the initial expression

Returning to the last example in the previous section, suppose we begin the investigation in hat-trail

```
Trail: ----------------- BadInsort.hs line: 10 col: 26 -------
<- putStrLn "agop"
<- insert 'p' "agor" | if False
```

and see that `insert` is broken. Wondering if there is an even simpler failure among the traced applications of `insert` we use `:o` to list them all in hat-observe. The list begins:

```
1 insert 'p' "agor" = "agop"
2 insert 'r' "ago" = "agor"
3 insert 'o' "ag" = "ago"
4 insert 'g' "ar" = "ag"
5 insert 'r' "a" = "ar"
6 insert 'a' "m" = "a"
7 insert 'm' [] = "m"
8 insert 'r' [] = "r"
9 insert 'g' "r" = "g"
10 insert 'o' "g" = "go"
--more-->
```

Observation `6` (or `9`) is the simplest so we use `:t 6` to request a new session of hat-trail at this simpler starting point:

```
Trail: ----------------- BadInsort.hs line: 10 col: 26 -------
<- insert 'a' "m"
```

We sometimes find it useful to start additional windows for sessions of the *same* Hat tool but looking at different parts of the trace:

`:o` [P]	in hat-observe, where P is an application pattern, starts a new hat-observe window with P (if given) as the first observation request
`:t`	in hat-trail, starts a new hat-trail window with the current selection as the initial expression

Apart from the determination of its starting point, a hat-observe or hat-trail session created by a `:o` or `:t` command is quite independent of the session from which it was spawned.

Finally, some facilities so far shown only in one tool are available in the other in a slightly different form. Two frequently used examples are:

`:s` N	in hat-observe, following the display of a list of at least N applications, creates a source window showing the expression of which application N is an instance
`=`	in hat-trail, if the outermost expression enclosing the current selection is a redex, complete the equation with that redex as left-hand side, adding `=` and a result expression (which becomes the new selection); or if the current selection is within an already completed equation, revert to the display of the left-hand-side redex only (which becomes the new selection)

There is a particular reason for the `=` command in hat-trail. Following trails of ancestral redexes means working *backwards*, from results expressed in the body of a function to the applications and arguments that caused them. The movement is outward, from the details inside a function to an application context outside it. Using `=` is one way to go *forwards* when the key information is what happens within an application, not how the application came about. Returning once more to our running example, here is how `=` can be used to reach inside the `insert 'a' "m"` computation.

```
Trail: ----------------- BadInsort.hs line: 8 col: 28 -------
<- insert 'a' "m" = "a"
<- insert 'a' "m" | if True
<- 'a' <= 'm'
```

4 Combined Testing and Tracing

When testing identifies a test that fails, we may need to trace the failing computation to understand and correct the fault. But it is usually too expensive to trace *all* tests just in case one of them fails. In this section, we describe a way of working with the tools that addresses this requirement.

We have defined a variant of the top-level testing function `quickCheck`, called `traceCheck`. It has two modes of working:

- In *running mode*, `traceCheck` seems to behave just like `quickCheck`, but actually keeps track of what test cases succeeded and what test case failed. In does this in a special file in the working directory called `.tracecheck`.
- In *tracing mode*, `traceCheck` reads this file, and will repeat the exact test case that led to failure, and *only* that one.

Suppose, for example, that we wish to test an `insert` function defined (incorrectly) in the file `Insert.hs`:

```
module Insert where

insert :: Ord a => a -> [a] -> [a]
insert x []     = [x]
insert x (y:ys) = if x <= y then x : ys
                  else y : insert x ys
```

We could write a test program (`Iprop.hs`) like this:

```
import Insert
import QuickCheck2

sameElems :: Eq a => [a] -> [a] -> Bool
[]     `sameElems` [] = True
(x:xs) `sameElems` ys = (x `elem` ys) &&
                              (xs `sameElems` (ys \\ [x]))
_      `sameElems` _  = False

prop_InsertSameElems :: Int -> [Int] -> Bool
prop_InsertSameElems x xs = insert x xs `sameElems` (x:xs)
```

We can load this module into GHCi as usual, and run `quickCheck` on the property:

```
Main> quickCheck prop_InsertSameElems
Falsifiable, after 1 successful tests:
1
[0,1,0]
```

But when we want to trace what is going on here we will have to use `traceCheck`. Notice that `traceCheck` is a bit slower than `quickCheck` since it has to save its random seed to a file each time *before* it evaluates a test — the test might crash or loop, and it might never come back.

```
Main> traceCheck prop_InsertSameElems
Falsifiable, after 0 successful tests:
0
[-1,1,0]
(Seed saved -- trace the program to reproduce.)
```

We now add the following definition of `main` to our property module `Iprop.hs` so that we have a complete Haskell program which can be compiled for tracing.

```
main :: IO ()
main = traceCheck prop_SortIsSort
```

Leaving this definition of `main` in the file `Iprop.hs` does not hurt, and reduces work when tracing the property again later. Let us compile and trace the program:

```
$ hmake -hat Iprop
...
$ ./Iprop
Again, the property is falsifiable:
0
[-1,1,0]
```

You can see that the output from `traceCheck` is now a little different. It carries out only the failing test that was saved by the previous `traceCheck` application, and confirms the result.

A .hat file is now generated, and the tracing tools can be applied as usual. We recommend that you always start with `hat-observe` when tracing a failed property, and observe the function calls of the functions mentioned in the property. The `hat-trail` tool can then be called upon from `hat-observe`.

For our illustrative example `hat-observe` immediately reveals that the recursive call is an even simpler failed application:

```
hat-observe> insert
1 insert 0 [-1,1,0] = [-1,0,0]
2 insert 0 [1,0] = [0,0]
```

If necessary, `:t 2` allows us to examine details of the derivation using `hat-trail`.

```
Trail: ----------------------- Insert.hs line: 5 col: 24 -----
<- insert 0 [1,0] = [0,0]
<- insert 0 [1,0] | if True
<- 0 <= 1
```

And so on.

5 Working with a Larger Program

So far our example programs have been miniatures. In this section we introduce a rather larger program —a multi-module interpreter and compiler— and discuss how to handle some of the problems it poses for testing and tracing.

5.1 An Implementation of Imp

We begin by outlining an interpreter and compiler for *Imp*, a simple imperative language. Imp programs are command sequences with the following grammar:

comseq	=	com {**;** com}
com	=	**skip**
	|	**print** exp
	|	**if** exp **then** comseq **else** comseq **fi**
	|	**while** exp **do** comseq **od**
	|	name **:=** exp
exp	=	term {op2 term}
term	=	name | value | op1 term | (exp)

Names are lower-case identifiers; values are integers or booleans; op1 and op2 are the usual assortments of unary and binary operators. Here, for example, is an Imp program (`gcd.in`) to compute the GCD of 148 and 58:

```
x := 148; y := 58;
while ~(x=y) do
  if x < y then y := y - x
  else x := x - y
  fi
od;
print x
```

The operational behaviour of an Imp program can be represented by a value of type `Trace String`, with `Trace` defined like this:

```
data Trace a = a :> Trace a | End | Crash | Step (Trace a)
```

Each printing of a value is represented by a `:>`. A program may terminate normally (`End`), terminate with an error (`Crash`) or fail to terminate by looping infinitely. To deal with non-termination, we introduce the `Step` constructor, as we explain later (in Section 5.3).

Our implementation of Imp includes both direct interpretation of syntax trees and a compiler for a stack machine. Here is a module-level overview.

`Behaviour`	Defines behavioural traces, their concatenation and approximate equality based on bounded prefixes.
`Compiler`	Generates machine instructions from the abstract syntax of a program. (Depends on `Machine`, `Syntax`, `StackMap`, `Value`.)
`Interpreter`	Computes the behavioural trace of a program by directly interpreting abstract syntax. (Depends on `Syntax`, `Behaviour`, `Value`)
`Machine`	Defines stack machine instructions and the execution rules mapping instruction sequences to behavioural traces. (Depends on `Behaviour`, `Value`.)
`Main`	Reads an Imp program and reports the behavioural traces obtained when it is interpreted and when it is compiled. (Depends on `Syntax`, `Parser`, `Interpreter`, `Machine`, `Compiler`.)
`Parser`	Defines parser combinators and an Imp parser using them. (Depends on `Syntax`, `Value`.)
`StackMap`	Models the run-time stack during compilation.
`Syntax`	Defines an abstract syntax for the language. (Depends on `Value`.)
`Value`	Defines basic values and primitive operations over them.

5.2 Tracing bigger computations

Even compiling an Imp program as simple as `gcd.in`, the binary-coded Hat trace exceeds half a megabyte. If we were tracing a fully-fledged compiler processing a more typical program, the .hat file could be a thousand times larger. The development of Hat was motivated by a lack of tracing information for Haskell programs, but clearly we could have too much of a good thing! How do we cut down the amount of information presented when tracing larger programs? (1) At *compile-time* we identify some modules as trusted — details of computation within these modules are not recorded in traces. (2) At *run-time* we use simple inputs. It is helpful that QuickCheck test-case generators usually start with the simplest values. (3) At *trace-viewing time* we set options in the Hat tools to control how much information is shown and to what level of detail.

Working with trusted modules Usually, untrusted modules depend on trusted ones, rather than the other way round, so trusted modules need to be compiled first[7]. It may be simplest first to compile *all* modules as trusted, then to recompile selected modules for full tracing. For example, if we want to compile the Imp system to trace only the details of computation in module `Compiler`:

$ *hmake -hat -trusted Main*
Compiles everything, with all modules trusted.
$ *touch Compiler.hs*
$ *hmake -hat Main*
Recompiles just `Compiler` *(and* `Main` *which depends on it) as fully traced modules.*

How effectively does this reduce the amount of trace information? With no modules trusted (apart from the prelude and libraries), and `gcd.in` as input, the `:info` table in hat-observe lists 88 top-level functions; more than a dozen have over 100 traced applications and several have over 300. With all but `Main` and `Compiler` trusted the `:info` table has just 23 entries; all but four of these show fewer than 10 applications and all have less than 30.

When a module T is compiled as trusted, applications of exported T functions in untrusted modules are still recorded, but the details of the corresponding computation within T are not. For example, in the `StackMap` module there is a function to compute the initial map of the stack when execution begins:

```
stackMap :: Command -> StackMap
stackMap c = (0, comVars c)
```

Details of the `Command` type (the abstract syntax of Imp programs) and the significance of the `StackMap` values need not concern us here; the point is that even with `StackMap` trusted, hat-observe reports the application of `stackMap`:

```
1 stackMap
    ("x" := Val (Num 148) :->
      ("y" := Val (Num 58) :->
        (While (Uno Not (Duo Eq (Var "x") (Var "y")))
          (If (Duo Less (Var "x") (Var "y"))
            ("y" := Duo Sub (Var "y") (Var "x"))
            ("x" := Duo Sub (Var "x") (Var "y"))) :->
          Print (Var "x"))))
  =
  (0,["x","y"])
```

But hat-observe does *not* report any applications of the auxiliary function `comVars` that computes the second component `["x","y"]`. This component is not just left orphaned —with no trace of a parent— instead it is adopted by the `stackMap` application, as this is the nearest ancestral redex recorded in

[7] The Haskell prelude and standard libraries are pre-compiled and trusted by default.

the trace. In hat-trail, if we select the `["x","y"]` component of the result and request the parent redex it is the `stackMap` application that is displayed.

Some applications within a trusted module are still recorded. For example, there may be applications of untrusted functional arguments in trusted higher-order functions, and there may be applications of constructors recorded because they are part of a result.

Controlling the volume of displayed information Even when traces are confined to specific functions of interest, there may be many applications of these functions, and the expressions for their arguments and results may be large and complex. In hat-trail, the *number* of applications need not concern us: only explicitly selected derivations are explored, and each request causes only a single expression to be displayed. In hat-observe, the counts in `:info` tables warn us where there are large numbers of applications, by default only unique representatives are shown when a function is applied more than once to the same arguments, and patterns can be used to narrow the range of a search. But if the volume of output from hat-observe is still too high, we have two options:

`:set recursive off`	Recursive applications (ie. applications of f in the body of f itself) are not shown.
`:set group` N	Show only N observations at a time — the default is 10.

In both hat-trail and hat-observe, large expressions can be a problem. Within a single window, the remedy[8] is to control the level of detail to which expressions are displayed. The main way we can do so is:

`:set cutoff` N	Show expression structure only to depth N — the default is 10.

Rectangular placeholders (shown here as ▮) are displayed in place of pruned expressions, followed by ellipses in the case of truncated lists. For example, here once again is the application of `stackMap` to the abstract syntax for `gcd.in`, but lightly pruned (`:set cutoff 8`):

```
stackMap
  ("x" := Val (Num 148) :->
     ("y" := Val (Num 58) :->
       (While (Uno Not (Duo Eq (Var ▮) (Var ▮)))
         (If (Duo Less (Var ▮) (Var ▮)) ("y" := Duo ▮ ▮ ▮)
           ("x" := Duo ▮ ▮ ▮)) :-> Print (Var "x"))))
```

More severely pruned (`:set cutoff 4`) it becomes a one-liner:

```
stackMap ("x" := Val ▮ :-> (▮ := ▮ :-> (▮ :-> ▮)))
```

One limitation of depth-based pruning is its uniformity. We face a dilemma if two parts of an outsize expression are at the same depth, the details of one are

[8]Apart from making the window larger! After which a `:resize` command may be needed.

irrelevant but the details of the other are vital. In hat-trail we can explicitly over-ride pruning for any selected ■ expression by keying `+`, and we can explicitly prune any other selected expression by keying `-`. A more extravagant solution is to view the expression in a cascade of hat-trail windows. Returning once more to the `stackMap` example, in a first hat-trail window suppose we have the heavily pruned redex, with a subexpression of interest selected:

```
stackMap ("x" := Val ■ :-> (■ := ■ :-> (■ :-> ■)))
```

We give the command `:t` to spawn a fresh hat-trail session starting with this subexpression. Pruned to the same cutoff depth it is now revealed to be:

```
While (Uno Not (Duo ■ ■ ■)) (If (Duo ■ ■ ■) (■ := ■) (■ := ■)) :->
  Print (Var "x")
```

Within this subexpression, we can select still deeper subexpressions recursively. We can continue (or close) the hat-trail session for each level of detail quite independently of the others.

5.3 Specifying properties of Imp

Let us think about how we are going to test the compiler and interpreter. There might be many properties we would like to test for, but one important property is the following:

> [**Congruence Property**] For any program p, interpreting p should produce the same result as first compiling and then executing p.

To formulate this as a QuickCheck property, the first thing we need to do is to define test data generators for all the types that are involved. We will show how to define the test data generators for the types `Name` and `Expr`. The other types have similar generators — see `Compiler/Properties.hs` for details.

For the type `Name`, we will have to do something more than merely generating an arbitrary string. We want it to be rather likely that two independently generated names are the same, since programs where each occurrence of a variable is different make very boring test data. One approach is to pick the name arbitrarily from a limited set of names (say {"a", . . . , "z"}). It turns out that it is a good idea to make this set small when generating small test cases, and larger when generating large test cases.

```
arbName :: Gen String
arbName = sized gen
 where
  gen n = elements [ [c] | c <- take (n `div` 2 + 1) ['a'..'z'] ]
```

To generate elements of type `Expr` (the datatype representing Imp expressions), we assume that we know how to generate arbitrary `Val`s (representing Imp values), Op1s and Op2s (representing unary and binary operators, respectively). The Expr generator is very similar to the one for binary trees in Section 2.7. We keep track of the size bound explicitly when we generate the tree recursively. When the size is not strictly positive any more, we generate a leaf of the tree.

```
instance Arbitrary Expr where
  arbitrary = sized arbExpr
    where
      arbExpr n =
        frequency $
          [ (1, liftM Var arbName)
          , (1, liftM Val arbitrary)
          ] ++
          concat
          [ [ (2, liftM2 Uno arbitrary arbExpr')
            , (4, liftM3 Duo arbitrary arbExpr2 arbExpr2)
            ]
          | n > 0
          ]
       where
        arbExpr' = arbExpr (n-1)
        arbExpr2 = arbExpr (n `div` 2)
```

There is no right or wrong way to choose frequencies for the constructors — you simply get what you write. The idea is to try to predict what kind of expressions we are likely to get in practice, and what kind of expressions are likely to generate counter-examples to our properties. The rationale for the above frequencies is the following: We do not want to generate leaves too often, since this means that the expressions are small. We do not want to generate a unary operator too often, since nesting `Not` or `Minus` a lot does not generate really interesting test cases. Also, the above frequencies can easily be adapted after monitoring test data in actual runs of QuickCheck on properties.

Finally, we can direct our attention towards specifying the congruence property. Without thinking much, we can come up with the following property, which pretty much directly describes what we mean by congruence; for all `p`, `obey p` should be equal to `exec (compile p)`.

```
prop_Congruence :: Command -> Bool
prop_Congruence p = obey p == exec (compile p)          -- wrong!
```

However, what happens when the program `p` is a non-terminating program? In the case where `obey` works correctly, the trace will either be an infinite trace of printed values, or the computation of the trace will simply not terminate. In both cases, the comparison of the two traces will not terminate either! So, for non-terminating programs, the above property does not terminate.

We have run into a limitation of using an embedded language for properties, and testing the properties by running them like any other function. Whenever one of the functions in a property does not terminate, the whole property does not terminate. Similarly, when one of the functions in a property crashes, the whole property crashes. To avoid solving the Halting Problem, we take the pragmatic viewpoint that properties are allowed to crash or not terminate, but only in cases where *they are not valid*.

The solution to the infinite trace problem consists of two phases.

First, we have to make the passing of time during the execution of a program explicit in its trace. We do this so that any non-terminating program will generate an infinite trace, instead of a trace that is stuck somewhere. The `Step` constructor is added to the `Trace` datatype for that reason — the idea is to let a trace make a 'step' whenever the body of a while-loop in the program has completed, so that executing the body of a while loop infinitely often produces infinitely many `Step`s in the trace.

The second change we make is that when we compare these possibly infinite traces for equality, we only do so approximately, by comparing a *finite prefix* of each trace. The function `approx n` compares the first `n` events in its argument traces for equality:

```
approx :: Eq a => Int -> Trace a -> Trace a -> Bool
approx 0 _        _        = True
approx n (a :> s) (b :> t) = a == b && approx (n-1) s t
approx n (Step s) (Step t) = approx (n-1) s t
approx n End      End      = True
approx n Crash    Crash    = True
approx n _        _        = False
```

We could imagine a somewhat looser definition of `approx` which does not require each occurrence of `Step` to match up, which allows more freedom in the compiler, but the current definition will do for now.

Now we can define a trace comparison operator on the property level, which compares two traces approximately: For arbitrary strictly positive n, the traces should approximate each other up to n steps. (We choose strictly positive n since for $n = 0$ the approximation is trivially true which makes an uninteresting test case.)

```
(=~=) :: Eq a => Trace a -> Trace a -> Property
s =~= t = forAll arbitrary $ \n ->
            n > 0 ==>
              approx n s t
```

The new version of the congruence property thus becomes:

```
prop_Congruence :: Command -> Property
prop_Congruence p = obey p =~= exec (compile p)
```

Note that this is still not the final version of the property; there are some issues related to test coverage, which will be discussed in the exercises in Section 6.3.

6 Practical Exercises

Exercises in this section use example programs available to Summer School participants in the directory `Test+Trace`.

6.1 About merge-sort (in the `Sorting` directory)

Exercise 1 Look at the simple merge-sort program in the source files `Mmain.hs` and `Msort.hs`. If `Mmain` is run with `words.in` as input, what lengths of list arguments occur in the applications of `merge` in `pairwise`, and how many applications are there for each length? Try to answer by inspection before verifying your answers using Hat. **Hint:** in hat-observe, either give a context to a `merge` application query or `:set recursive off`. □

Exercise 2 Examine the recursive `pairwise` computation. How deep does the recursion go? Are all equations in the definition of `pairwise` really necessary? **Hint:** in hat-trail, trace the ancestry of the list of strings from which the output is formed. □

Exercise 3 How many comparisons and `merge`'s does it take to sort a list that is already sorted? What about a list that is reverse-sorted? □

Exercise 4 Write QuickCheck properties that characterise what each function in the `Msort` module does. Check that your properties hold. What can you say about test coverage? □

Exercise 5 Look at the improved(?) version of merge-sort in `Nmain.hs` and `Nsort.hs`. Instead of starting the pairwise merging process merely with unit lists, the idea is to find the largest possible ascending and descending sublists. However, we have made a deliberate mistake! Find a test case where the property of the `msort` function does not hold. Can you locate and fix the bug? Do all your previously defined `msort` properties now hold? □

Exercise 6 How many comparisons and `merge`'s does the improved (and now correct!) merge-sort take for already-sorted input? □

Exercise 7 What property should the function `ascending` have? Check that it holds. How lazy is the `ascends` function? What happens if an element of its list argument is undefined? Trace the computation to see why. Can you improve the definition of `ascends`? □

6.2 About cryptarithmetic (in the `SumPuzzle` directory)

The next few exercises are about a program that solves cryptarithmetic puzzles (source files `SumPuz.hs` and `Main.hs`). Inputs are lines such as `SEND + MORE = MONEY` — an example provided in the file `puzzle3.in`. The program has to find a mapping from letters to digits that makes the sum correct. Your task is to understand how exactly the program works, and to formulate your understanding in tested properties.

Exercise 8 Compile the program for tracing, and run it with `puzzle3.in` as input. In the process of searching for a solution, the program carries out many additions of two digits. The digits are candidate values for letters in the same column of the encrypted sum:

```
  SEND
+ MORE
```

What is the maximum result actually obtained from any such digit addition? The result occurs more than once: how many times? (Use `:set all` and appropriate application patterns in hat-observe.) Select one example of a maximal sum to investigate further using hat-trail. Which letters are being added and with what candidate values? What values are assigned to other letters at that point? Why does this branch of the search fail to reach a complete solution? □

Exercise 9 The function `solutions` is the heart of the program. As you can see in the function `solve` in `SumPuz.hs`, the standard top-level way to call the function `solutions` is with `0` as the fourth argument and `[]` as the fifth argument. In `Properties.hs`, we have predefined a function `find` that does exactly that:

```
find :: String -> String -> String -> [Soln]
find xs ys zs = solutions xs ys zs 0 []
```

In this and the following exercises we are going to write properties about this function `find`.

The first property to define is a *soundness* property: the program only reports genuine solutions. It should say something like:

> For all puzzles, every element in the found list of solutions is arithmetically correct.

Check that your property holds! Remember that your task is to characterise exactly what kind of puzzles the program solves, and in what way. So if your property does not hold, use the tracing tools to understand why, and then revise your property (not the program) until it is correct. □

Exercise 10 Use a test data monitor to check how interesting the test cases are. For example, is a test case where there are no solutions interesting? Try to eliminate uninteresting tests by adding an appropriate precondition to your property. How does this influence the size of the tested puzzles? □

Exercise 11 The next property to define is a *completeness* property: the program always finds a solution if there is one. A handy way to do this is to say something like:

> For all numbers x and y, if I supply as input the digits of x, plus, the digits of y, equals, and the digits of $x + y$, then the list of found solutions should include this digit-identity.

Again, check that your property holds. If not, use the tracing tools to understand why, and revise your property accordingly. □

Exercise 12 Hypothetically, how would you change the soundness and completeness properties if the solutions function worked in such a way that it always only returned one solution even if there are many?

6.3 About Imp (in the `Compiler` directory)

The final group of exercises involve testing, tracing, fixing, specifying and extending the Imp interpreter and compiler.

Exercise 13 Recall the QuickCheck congruence property that should hold for the Imp compiler and the interpreter. The version of the Imp system in the `Compiler` directory has been deliberately broken, so it does not satisfy this property. Indeed, it hardly works at all: try running it on `gcd.in`. Use QuickCheck and Hat to find the two bugs we have introduced. Fix them! □

Exercise 14 There are some functions in which we can apparently introduce as many bugs as we want; the congruence property will still hold! Which are these functions? **Hint:** Which functions are used both by the compiler and the interpreter? □

Exercise 15 Random testing works best if it is applied at a fine grain! Therefore, formulate a property that is only going to test compilation and interpretation of *expressions*. **Hint:** You can reuse the congruence property of programs, but generate only programs that print a single expression (which cannot contain variables). Is non-termination still an issue? □

Exercise 16 Now investigate the test coverage of QuickCheck for your property. Insert a test data monitor that checks what kind of traces are generated by the programs during test, and check the distribution. What do you think of the distribution of test data? Most generated expressions are type incorrect! Adapt your property by using the implication operator `==>` to discard this rather large amount of useless test data.

Note: To show that without this fix, your property does not have good test coverage, introduce the following bug: flip the order of the arguments of binary operators in the expression compiler. Can your old property find the bug? Can your new one? □

Exercise 17 The original congruence property for programs has a similar problem; the whole program crashes if the condition in an `if` or `while` statement is type incorrect, and this happens a lot during testing. Adapt the program congruence property to overcome this problem. □

Exercise 18 Suppose the Imp language is extended by generalising assignments to multiple assignments. Instead of just one variable name on the left of each `:=` there are one or more, separated by commas, and on the right an equal number of expressions, also separated by commas. A multiple assignment is executed by first evaluating all the right-hand expressions and then storing the results in corresponding left-hand variables in left-to-right order. Here is an example program (`power.in`) which raises `3` to the power `6`:

```
a, n, x := 3, 6, 1;
while 0 < n do
  if (n\2) = 1 then n, x := n-1, x*a else skip fi;
  a, n := a*a, n/2
od;
print x
```

By making changes in the following places, revise the Imp interpreter and compiler to work with multiple assignments.

`Syntax`	Change the `:=` construction in the `Command` type.
`Parser`	Change the final alternative in `nonSeqCommand`. **Hint:** define `listOf :: Parser a -> Parser [a]`.
`Interpreter`	Change the `:=` equation in the definition of `run`. **Hint:** generalise the definition of `update`.
`StackMap`	Change the `:=` equation in the definition of `comVars`.
`Compiler`	Change the `:=` equation in the definition of `compObey`. **Hint:** none — we hope you get it wrong!

Test your extension first on `power.in`, using Hat to investigate any faults. When you have an interpreter and compiler that work for this example program do you suppose it will work for all parallel-assignment programs? Revise the program generator in the `Properties.hs` so that the congruence property is asserted over the extended language. Apply QuickCheck and Hat as necessary to achieve a solution that passes an appropriate range of tests. □

Exercise 19 Compare the assignments:

```
x, y := e1, e2          and          y, x := e2, e1
```

Under what conditions do these two assignments mean the same thing? Formulate this conditional equivalence as a QuickCheck property and check that the property holds. □

Acknowledgements

The Hat tracing system was developed under grant number GR/M81953 of the Engineering and Physical Sciences Research Council of the United Kingdom.