High Coverage Testing of Haskell Programs

Tristan Allwood Cristian Cadar Susan Eisenbach

Department of Computing
Imperial College London
{tora,cristic,susan}@imperial.ac.uk

ABSTRACT

This paper presents a new lightweight technique for automatically generating high coverage test suites for Haskell library code. Our approach combines four main features to increase test coverage: (1) automatically inferring the constructors and functions needed to generate test data; (2) using needed narrowing to take advantage of Haskell's lazy evaluation semantics; (3) inspecting elements inside returned data structures through the use of `case` statements, and (4) efficiently handling polymorphism by lazily instantiating all possible instances.

We have implemented this technique in IRULAN, a fully automatic tool for systematic black-box unit testing of Haskell library code. We have designed IRULAN to generate high coverage test suites and detect common programming errors in the process. We have applied IRULAN to over 50 programs from the *spectral* and *real* suites of the *nofib* benchmark and show that it can effectively generate high-coverage test suites—exhibiting 70.83% coverage for *spectral* and 59.78% coverage for *real*—and find errors in these programs.

Our techniques are general enough to be useful for several other types of testing, and we also discuss our experience of using IRULAN for property and regression testing.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Testing tools*

General Terms

Reliability

Keywords

Black-box testing, Haskell, needed narrowing, IRULAN

ISSTA '11, July 17–21, 2011, Toronto, ON, Canada

1. INTRODUCTION

Writing enough tests to provide confidence in a given piece of code is difficult and time-consuming. If the code is non-trivial, such a task may not even be possible in a realistic timeframe. This has led to a large body of work on automatically generating test suites in the imperative programming community [5, 6, 12, 13, 15, 23].

The Haskell programming language contains a variety of interesting features which present unusual challenges and opportunities for automatic testing. These include a rich and expressive static type system, higher order programming, polymorphism, and laziness.

Haskell currently provides support for testing through tools such as QuickCheck [7] and (Lazy) SmallCheck [22], which allow developers to express and check functional properties that should hold in their programs. Such manually-encoded properties can include arbitrary correctness checks, but writing them is often cumbersome. The Check tool family, for example, requires programmers to explicitly write both the property they want to check, and special functions that generate values for each user-defined type.

Writing a comprehensive set of explicit properties is hard, whereas programmers already implicitly state invariants and properties that must hold in their code. Pattern matches may ignore cases the programmer believes can never happen (leaving the compiler to silently insert calls to throw an exception in the missing cases); or the programmer may explicitly throw an assertion exception if control flow reaches an unexpected program point. When packaged up into a releasable API, the programmer may wish to ensure that the user of their API can't trigger these errors. Furthermore, we can use IRULAN to automatically find higher-level errors by cross-checking different implementations of the same interface, or different versions of the same application.

The main contribution of this paper is a new lightweight technique for generating high coverage test suites for Haskell library code. Like Lazy SmallCheck, our approach is based on needed narrowing, the refinement of argument terms into values only when their evaluation is demanded, first introduced in [3]. However, we go beyond previous work [7, 22], which focuses on programmer provided property/predicate checking, and propose the follow automated techniques:

1. Generating high-coverage test suites for Haskell library functions without requiring the user to specify generator functions.

2. Detecting generic errors such as non-exhaustive pattern matches, extracting elements from an empty list,

infinite loops, and stack overflows.

3. Matching output expressions in different test suites to automatically highlight any changes in behaviour between different implementations of the same interface.

To accomplish these goals, we take advantage of the main features of the Haskell language, i.e., its rich static type system, laziness and polymorphism. To this end, we have designed techniques that:

1. Take advantage of Haskell's laziness through the use of needed narrowing.
2. Automatically infer the constants and functions needed to generate test data.
3. Inspect elements inside returned data structures by using `case` expressions.
4. Efficiently handle polymorphism by lazily instantiating all possible instances.

We have implemented these techniques in a tool called IRULAN, a high coverage test suite generator for Haskell library code. We have applied IRULAN to over 50 programs from the *spectral* and *real* suites of the *nofib* benchmarks. We show that it can effectively generate high-coverage test suites—exhibiting 70.83% coverage for *spectral* and 59.78% coverage for *real*—and discover common programming errors in these programs, such as non-exhaustive patterns errors, extracting elements from an empty list, infinite loops, and stack overflows. We have also used IRULAN to perform both property testing and regression testing, and report on three case studies where erroneous changes in behaviour have been found.

The rest of the paper is structured as follows. Section 2 gives a high-level overview of IRULAN by providing a contrived but illustrative running example. Section 3 presents the main features of the tool, namely *support set* construction (§3.1), plan generation (§3.2), decomposition of data structures (§3.3), support for polymorphism (§3.4), execution engine (§3.5), and exploration strategies (§3.6). Next, Section 4 presents our experimental evaluation. Section 5 discusses some extensions to IRULAN, namely property testing (§5.1) and regression testing (§5.2). Finally Section 6 discusses related work, and Section 7 concludes.

2. OVERVIEW

Figure 1 defines a sorted binary tree data type *IntTree* together with a function *insert* that adds a node to the tree. The *insert* function contains a bug: if the user tries to insert a node with the same value as that of a node already in the tree, the compiler throws a *Non-exhaustive patterns* exception. In this section, we show how IRULAN can automatically find this bug, as well as generate a comprehensive test suite providing 100% expression code coverage in the *IntTreeExample* module.

For those unfamiliar with Haskell, the *IntTree* data type has two constructors: *Empty* and *Branch*. *Empty* takes no arguments, while *Branch* takes two *IntTree*s for its left and right children, and an *Int* for the value of the new root node.

The type signature for *insert* says it is a function that accepts an *Int* and an *IntTree* and returns a new *IntTree* result. The function works by *pattern matching* on the existing *IntTree*. In the case of inserting an *Int* into an *Empty* tree, a new *Branch* with two *Empty* children is created. In the case of inserting an *Int* into an existing *Branch*, guards (|) are used to establish where to place the new *Int* value in the sorted binary tree.

```
module IntTreeExample where
data IntTree
  = Empty
  | Branch IntTree Int IntTree
insert :: Int → IntTree → IntTree
insert n Empty = Branch Empty n Empty
insert n (Branch left x right)
  | n < x = Branch (insert n left) x right
  | n > x = Branch left x (insert n right)
```

Figure 1: Haskell sorted binary tree.

Unfortunately, our programmer has forgotten to implement the case when $n = x$, i.e. the value n to be inserted is already in the tree. When the program has an incomplete pattern list and does not specify a default alternative, the Haskell compiler will insert a call to the library function *error*, which throws a *Non-exhaustive patterns* exception that will typically terminate the program. If IRULAN runs an expression and catches an otherwise uncaught exception, it will report that expression as a potential bug.

Note that although Figure 1 includes the code for the *insert* function, IRULAN is in fact a *lightweight black-box tool* that does not analyse the actual implementation of Haskell modules. To construct test cases, IRULAN only makes use of the signatures of exported data types and functions, and a set of predefined constants. In our example, IRULAN generates *IntTree* values by using *IntTree*'s two constructors, *Empty* and *Branch*, together with two integer constants, 0 and 1, which are set explicitly on the IRULAN command line. In addition, IRULAN can also construct expressions of a given type by making use of publicly exported functions (§3.1) or by taking apart wrapped values (§3.3).

When IRULAN runs, it will try to build expressions to use as arguments for the function being tested. As the space of possible expressions is infinite for all but the simplest programs, IRULAN bounds its search to encourage breadth in the exploration. The bounding on the search space, given by its depth, is roughly a function of the number of steps necessary for constructing test expressions, and will be discussed in more detail in §3.6.

Figure 2 shows the output of IRULAN when used to test the *insert* function. To test a Haskell module with IRULAN, the user needs to provide several arguments: the name of the module to be tested (*IntTreeExample* in our case), the maximum exploration depth (13 in this example), the exploration strategy used (`-d`, depth first search in this example; see §3.6 for details), and (optionally) a set of predefined constants used, in our case the *Int* values 0 and 1. With these arguments, IRULAN starts generating test cases up to that maximum depth for all functions exported by the module.

As shown in Figure 2, IRULAN took 0.35 seconds to generate 29 test expressions which achieved 100% coverage of the 22 expressions in the sample module.[1] IRULAN also gen-

[1]The expression coverage metrics are obtained by HPC [11]. Expression coverage is formally defined in Section 4.

```
$ irulan --depth=13 -d --ints='[0,1]' IntTreeExample
IntTreeExample:
Error expressions:
insert 0 (Branch ?1 0 ?2) ==> !
  IntTreeExample.hs:(8,0)-(11,41):
    Non-exhaustive patterns in function insert

  insert 1 (Branch ?1 1 ?2) ==> !
    IntTreeExample.hs:(8,0)-(11,41):
      Non-exhaustive patterns in function insert

Runtime: 0.35 seconds
Test expressions generated: 29
Expression coverage: 100% (22/22)
```

Figure 2: IRULAN running on the *IntTreeExample* with additional information about the run presented.

```
 1 insert ==> OK
 2 insert ?1 ==> OK
 3 insert ?1 ?2 ==> ?2
 4   insert ?1 (Branch ?2 ?3 ?4) ==> ?1
 5     insert 0 (Branch ?1 ?2 ?3) ==> ?2
 6       insert 0 (Branch ?1 0 ?2) ==> !
          IntTreeExample.hs:(8,0)-(11,41):
            Non-exhaustive patterns in function insert
         ...
 7     insert 1 (Branch ?1 ?2 ?3) ==> ?2
 8       insert 1 (Branch ?1 0 ?2) ==> OK
 9       insert 1 (Branch ?1 1 ?2) ==> !
            IntTreeExample.hs:(8,0)-(11,41):
              Non-exhaustive patterns in function insert
         ...
10   insert ?1 Empty ==> OK
         ...
```

Figure 3: Part of the steps followed by IRULAN to discover the error in Figure 1.

erated two test cases exposing the bug discussed above: one in which the value 0 is inserted into a tree consisting of a single node with value 0, and a similar test case for value 1.

In Figure 3, we show the main steps taken by IRULAN to detect the bug in Figure 1. At each step, IRULAN generates a test case of the form `E ==> R`, where E is an expression and R the result of its evaluation. Each E consists of a function or constant exported by the module applied to zero or more arguments, which can have one of the following two forms:

1. A fully-defined expression, such as 1 or *Empty*.
2. A partially-defined expression which contains one or more arguments of the form $?n$, which throw an exception when evaluated.

IRULAN prunes program states by observing how the code uses its arguments, and not generating test expressions for the unused arguments. In order to accomplish this, IRULAN initally provides dummy arguments (the $?n$ arguments mentioned above) that throw a special exception when evaluated. For example, the compilation of `insert ?1 ?2` is `insert (error "?1") (error "?2")`[2].

[2]In the implementation, a custom IRULAN-specific exception is thrown to prevent accidental clashes with user errors, but the purpose is the same.

The use of these `?` arguments, which are then replaced later with more concrete expressions only if they are needed, is a form of needed narrowing. This works particularly well for Haskell due to the lazy evaluation at its core. As a simple example, consider the case of inserting a value n into an *Empty* tree. This operation always succeeds, and returns a tree of the form (*Branch Empty n Empty*) ***without ever evaluating the inserted value n.***

When IRULAN evaluates an expression E, there can be three possible outcomes:

1. `E ==> OK` This outcome means that E was evaluated to *Weak Head Normal Form* (WHNF) [21] without an exception being raised. An expression is in WHNF if it cannot be simplified further without being taken apart by pattern matching (i.e., it is a constructor possibly applied to some arguments), or applied to arguments (i.e., it is a function that expects an argument). Alternatively, an expression being evaluated to WHNF could throw an exception, or terminate abnormally (e.g., by infinite recursion). In order to see if an expression runs to WHNF without generating an exception IRULAN uses the built-in function *seq*, which will guarantee it's first argument has a WHNF before returning the second argument.

   As this outcome means no exceptions were raised, it also means that E ran without having to evaluate any of its $?n$ arguments. For example, line 10 contains the test case `insert ?1 Empty ==> OK` because inserting any number into an *Empty* tree always succeeds by returning a tree of the form *Branch Empty n Empty* (the first line of the definition of *insert* in Figure 1). The number inserted (`?1`) won't be evaluated unless some later code inspects it.

2. `E ==> ?k` This outcome occurs when the evaluation of E requires the evaluation of its k^{th} argument. For example, line 4 contains the test case `insert ?1 (Branch ?2 ?3 ?4) ==> ?1` meaning that in order to insert the value `?1` into a non-empty tree, *insert* needs to evaluate it (to compare it to the *Int* inside the *Branch*).

3. `E ==> !` This outcome occurs when the evaluation of E raises an uncaught exception. For example, line 6 catches our bug: trying to insert 0 into a tree that already contains a node with value 0 raises the *Non-exhaustive patterns* exception.

An important feature of IRULAN is that the test cases produced are almost completely valid Haskell expressions (modulo the `?n` arguments, and type class dictionaries). A developer could load *IntTreeExample* into an interactive Haskell system and execute

```
> insert 1 (Branch undefined 1 undefined)
```

(replacing the ? arguments in the counter-example with `undefined`) and get back the *Non-exhaustive patterns* error. IRULAN is designed to provide test cases like this, which only make use of publicly exported identifiers.

3. KEY FEATURES

We now discuss in more detail the key aspects of our technique, including some of the most important implementation

details. When IRULAN starts execution, it first establishes a *support set* (§3.1), i.e., a set of constants, constructors and functions that it uses to build arguments for the functions being tested. With the support set in place, IRULAN then generates a *Plan* (§3.2), a lazy data structure that represents the (possibly infinite) ways of building up expressions to be used for these arguments. Two particular challenges associated with building a *Plan* are the need to inspect elements inside returned data structures (§3.3) and to handle polymorphism (§3.4).

The *Plan* is then traversed by an *Engine* (§3.5) that uses the GHC[3] API to execute states from the *Plan* and generate test cases. The choice of states to be run is directed by an exploration strategy (§3.6).

IRULAN has been implemented in Haskell, making extensive use of the GHC API to reflect dynamically on Haskell modules and to generate and run Haskell code. Although IRULAN presents test cases to the user in a Haskell-like syntax, which treats type application as implicit and inferred, internally it keeps the type arguments that are also applied to polymorphic functions in order to make sure that resulting expressions are well typed. We discuss some of the more interesting consequences of test generation in a fully polymorphic language in §3.4.

3.1 Support Sets

In order to test a function, IRULAN first needs to establish a *support set*. This will be a set of constants, constructors, type class instances and functions that can be used to build arguments for the function being tested. As Haskell is statically typed, IRULAN can use type information to incrementally build up its support set. In the first step, IRULAN examines the type of each argument of the test function. Then, for each argument type T, IRULAN inspects the API of the module declaring T, and adds to T's support set all the constants, constructors and functions declared in that module that return expressions of type T. This process then continues recursively for any new types that are needed to construct arguments for any newly added support functions.

The size of the state space that IRULAN needs to explore increases exponentially with the number of elements in the support set. To avoid an exponential blow-up of the state space, IRULAN uses the following important observation: if all the constructors of a type T are exported by a module, then IRULAN only adds them to T's support set, ignoring any functions that return expressions of type T. The insight here is that when all of T's constructors are available, we can (almost always) generate all expressions of type T[4]. Only when a module does not make T's constructors available do we use functions that return expressions of type T. For example, the support set for the type *IntTree* in Figure 1 consists of *IntTree*'s two constructors, *Empty* and (*Branch IntTree Int IntTree*), together with the (command-line specified) constants 0 and 1 which are used as arguments to the second constructor. Because all of *IntTree*'s constructors are exported by the *IntTreeExample* module, the function *insert*, which returns expressions of type *IntTree*, is not added to *IntTree*'s support set. In practice we have seen this optimisation give a decrease of up to 50% in the size of support sets generated.

Finally, we allow users to explicitly add additional functions to the support set, should they want some larger values available for testing, or to provide higher order functions that may not have been found by IRULAN's recursive searching algorithm. This can simply be accomplished by creating a new module that defines a set of constants, constructors and functions, and instructing IRULAN to include the module via a command-line option.

[3] `http://www.haskell.org/ghc`

[4] The exception being when a publicly exported constructor makes use of an abstract type that has no public way of being constructed, a rare situation that we haven't seen in practice.

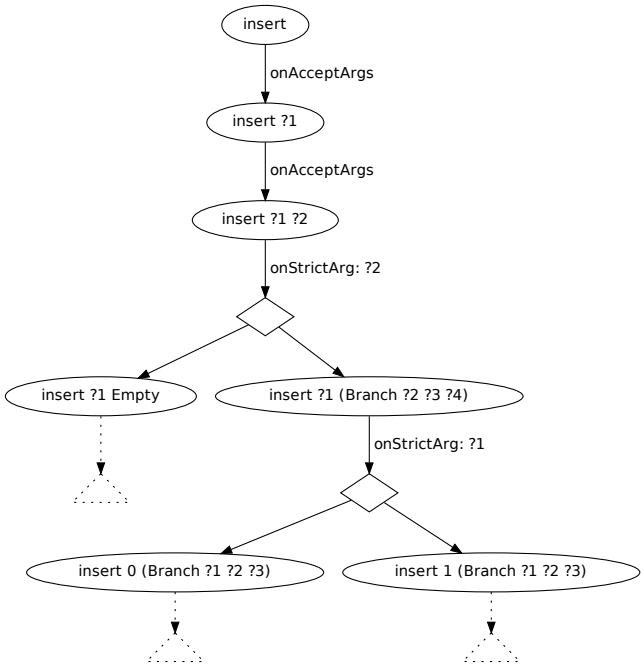


Figure 4: Part of the *Plan* followed by IRULAN while testing the *insert* function in Figure 1, which illustrates the `onAcceptArgs` and `onStrictArg` cases.

3.2 Planning

With a support set in place, IRULAN then builds a *Plan*, a lazy data structure that represents the (possibly infinite) number of ways in which IRULAN could build expressions to test the functions under testing.

Figure 4 shows part of the *Plan* dynamically generated by IRULAN while testing the *insert* function in Figure 1. The *Plan* consists of a series of steps: each step is denoted by an oval containing the expression to be evaluated in that step. Where there are several ways to generate test data, a diamond is used to split the *Plan*. The arrows linking the steps are annotated with the outcome when evaluating the expression in that step. There are four cases to consider when testing an expression e in the context of a function under test f:

1. `onAcceptArgs`: This case is triggered when IRULAN tests whether e accepts its arguments. For example, line 2 in Figure 3 shows that *insert* always accepts its first argument. When this happens, IRULAN is given a *Plan* to follow unconditionally. This new *Plan* either increases the number of arguments passed to e, or, if e

is used to build an argument to f, it instantiates that argument to e and continues f's evaluation.

2. `onStrictArg`: This case is triggered when e requires the evaluation of one of its arguments, which is looked up in a map to find a list of *Plan*(s) detailing possible ways of instantiating that argument. A non-deterministic choice can be made between these plans.

3. `onConstructor`: This case is triggered when IRULAN applies a function f to all of its arguments, and f successfully returns a data constructor d without generating an *error*-thrown exception. In this case, if d is publicly exported, IRULAN will go on to inspect (using `case` statements) the arguments passed to d by f. To do so, the *Plan*s for inspecting each argument can be retrieved by looking up d in a map from data constructors to *Plan*s. We discuss the `onConstructor` case in detail in §3.3.

4. If the evaluation of e generates an *error*-thrown exception, the current *Plan* reports the error and stops.

The first *Step* in the plan of Figure 4 tests whether *insert* accepts any of its arguments. This triggers the `onAcceptArgs` case, because running *insert* with no arguments returns successfully. Following `onAcceptArgs` means IRULAN next tries to apply *insert* to one argument. This again returns successfully, so in the third step IRULAN applies *insert* to two arguments. The application of *insert* to two arguments requires the evaluation of the second argument, which triggers the `onStrictArg` case. At this point, IRULAN returns a *Plan* with a non-deterministic choice (denoted in Figure 4 by a diamond): in the next step it must either use the *Empty* data constructor to create the `?2` argument, or use the *Branch* constructor.

When the *Branch* constructor is used, the new expression requires the evaluation of its first argument of type *Int*, so the `onStrictArg` case is again triggered. In the context of the example there are two ways of making an *Int* value, by using the constants 0 and 1, so IRULAN returns again a *Plan* with a non-deterministic choice of using either the constant 0 to create the *Int* argument, or the constant 1. These are used to instantiates the first argument of `(insert ?1 (Branch ?2 ?3 ?4))` with the respective *Int* constant.

Note that the existence of non-deterministic choice points in the *Plan*s generated by IRULAN gives rise to different exploration strategies (e.g. depth first search, iterative deepening), which we will discuss in §3.6.

3.3 Planning with Case Statements

Haskell makes it easy to use data structures, particularly simple ones such as tuples and lists, to wrap several items together and pass them as function arguments or return values. Consequently, the ability to take apart data structures, implemented in IRULAN by using Haskell `case` statements that operate on data constructors to extract their elements, is essential for our technique. In particular, this ability serves three purposes: (1) it allows IRULAN to increase program coverage by executing the individual elements (2) it leads to the discovery of errors in the elements that would otherwise be hidden by laziness, and (3) it allows IRULAN to use the elements as arguments to the functions under testing. We discuss each of these cases below:

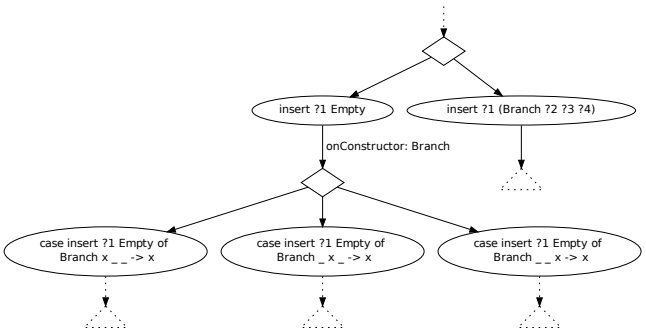
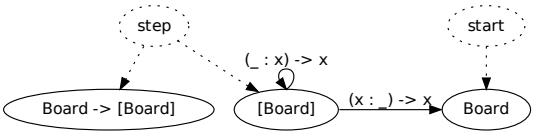


Figure 5: Part of the *Plan* followed by IRULAN while testing the *insert* function in Figure 1, illustrating the `onConstructor` case.

module *Board* (*start*, *step*, *Board*) **where**
data *Board* = ..
start :: *Board*
step :: *Board* → [*Board*]

(a) A module that needs `case` statements to test its functions.



(b) Case Constructor Graph for the *Board* module.

Figure 6: A `case` statement example.

(1) Increasing coverage and (2) discovering errors hidden by lazy evaluation: Returning to our running example, Figure 5 shows the *Plan* followed by IRULAN when the `insert ?1 Empty` expression is selected at the first non-deterministic choice point in Figure 4. Because this expression has applied the function under testing (*insert*) on all of its arguments, and successfully returned a data constructor (`Branch Empty ?1 Empty`), IRULAN proceeds to take the returned data structure apart. Specifically, the `onConstructor` case in the current *Plan* is triggered, and IRULAN inspects all of the arguments of the returned data constructor, namely `Empty`, `?1`, and `Empty`. As shown by our experimental results in Section 4, this allows IRULAN to significantly increase the coverage in the modules under testing and discover errors that would otherwise pass undetected.

(3) Using data structure elements as arguments to a tested function: To understand how IRULAN can use data structure elements to construct arguments for the functions under testing, consider Figure 6a, which represents the API for modelling a board game. In this example, *Board* is a data type representing a board, *start* constructs an initial *Board*, and *step* provides a way to generate new *Board*s based on an existing one (for example by making all the possible moves from some *Board*). In order to test the *step* function, it is essential for IRULAN to be able to take apart the list returned by *step*, so that it can test sequences of

board moves. Without this ability, IRULAN could only test the *step* function on the initial board returned by *start*.

As part of the support set discovery phase, IRULAN builds a *Case Constructor Graph*, a graph of all types for which case statements can be used to produce expressions. Figure 6b shows the graph for Figure 6a. The roots of this graph are the functions in the support set, in our case *start* and *step*. Each root points to nodes representing the types returned by applying the respective function to zero or more arguments. An arc between two types T_1 and T_2 is labelled by a case statement pattern that can be used to obtain a value of type T_2 from one of type T_1. For example, the arc between $[Board]$ and *Board* is annotated by $(x : \_) \rightarrow x$ which represents the case pattern that takes the head element out of the list of *Board*s.

To construct a value of a given type using the Constructor Graph, IRULAN first eliminates from the graph all the nodes which are not inversely reachable from the desired type (e.g., for our example, if the type of interest is *Board*, then we would eliminate the node $Board \rightarrow [Board]$). Then, starting from the roots of the graph (*start* and *step* in our example), IRULAN traverses the graph as though it were a non-deterministic finite automaton, with the accepting state being the node containing the desired type (*Board* in our case). During this non-deterministic traversal, each arc adds an appropriate `case` statement to the expression being constructed. For example, if we start with the expression *step* `?1` from the *step* root, we will begin the traversal on the $[Board]$ node. If we follow the $(\_ : x) \rightarrow x$ edge, we will build and test the expression **case** (*step* `?1`) **of** $\_ : x \rightarrow x$. If that succeeds, we will be back at the $[Board]$ node. Then following the $(x : \_) \rightarrow x$ arc, we will build and test the expression **case** (**case** (*step* `?1`) **of** $\_ : x \rightarrow x$) **of** $x : \_ \rightarrow x$, which upon success puts the algorithm on the *Board* node. As the *Board* node is the desired node, we can use the expression **case** (**case** (*step* `?1`) **of** $\_ : x \rightarrow x$) **of** $x : \_ \rightarrow x$ as a test expression of type *Board*.

3.4 Polymorphism

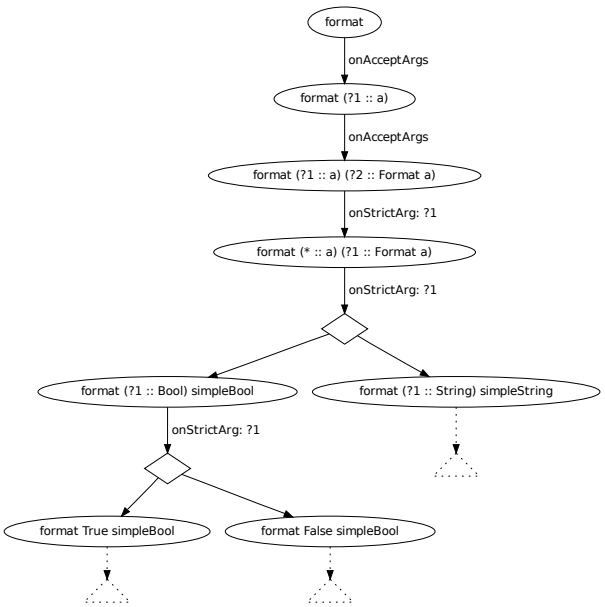
Haskell has a rich type system that allows *type variables* to be used as types. In Figure 7a, for example, values of type *Format a* represent some strategy for turning values of a generic type *a* into *String*s. The code has two example formatters for *String*s and *Bool*s, and a function *format* that takes a value of type *a* and a formatter for that type, and uses the formatter to turn the value into a string.

Figure 7b shows the first steps taken by IRULAN while testing the function *format*. In addition, the figure also shows the type of some of the generated arguments. At runtime, type information is erased, and the runtime assumes the code it executes is type correct, meaning that IRULAN must only create correctly typed expressions.

The first three steps shown in Figure 7b proceed as before, testing the *format* function with an increasing number of arguments. The execution of the expression in the third step requires a value for the first argument, which currently has generic type *a*. At this point, IRULAN could try to populate the first argument with every single value from the support set. However, the only thing that the *format* function can safely do with a variable of any type is force it to WHNF. We call this an *unconstrained* value, and represent it as `*` (which at the implementation level is compiled to (), the unit value).

module *Format* (*Format*, *simpleString*, *simpleBool*, *format*) **where**
data *Format a* = ...
simpleString :: *Format String*
simpleBool :: *Format Bool*
format :: *a* → *Format a* → *String*

(a) A polymorphic formatter.



(b) Part of the *Plan* followed by IRULAN while testing the formatter, with additional type annotations shown.

Figure 7: An example containing polymorphism.

IRULAN next determines that *format* is strict in its second argument, which is of type *Format a*. As the type of this argument is not a type variable, but a constructor application, it attempts to unify it with elements from the support set. In our example, there are two possibilities: *simpleBool* and *simpleString*. Thus IRULAN makes a non-deterministic choice between the two, and instantiates the type variable *a* to *Bool* or *String*. At this point, IRULAN re-abstracts the `*` back to `?1`. The *Plan* algorithm then proceeds as before, instantiating the new `?1` with *True* and *False* in the *simpleBool* branch, and appropriate strings in the *simpleString* one.

3.5 The Execution Engine

The actual execution of test expressions in IRULAN is handled by an execution *Engine*. The *Engine* is responsible for converting IRULAN test expressions into runtime values, evaluating them, and then inspecting the result to see if it is an error, a `?k` argument, or a WHNF value. For efficiency, IRULAN uses GHC to compile the test modules to native code, and test expressions are built by manipulating function and data closures in the heap.

The compilation and execution of test cases happens in the same process as IRULAN itself, to remove inter-process communication overheads. Once an expression has been converted into an executable entity, it is evaluated to WHNF

using the built-in function *seq*; this has several possible outcomes:

- **Evaluation terminates normally:** This case occurs if the test expression evaluates to some WHNF. In the *onConstructor* cases of the *Plan*, the name of the constructor at the head of the WHNF value will need to be looked up in a map. To find its name, the *Engine* interrogates the closure on the heap that represents the value.
- **An exception is thrown:** This could be due either to an *error* call inside the function, or the evaluation of a `?k` argument (which stands for `(error ?k)`). To distinguish between the two cases, IRULAN inspects the caught exception.
- **A time-out is reached:** Some test expressions may not terminate, or may take a very long time to complete. To avoid becoming blocked on such expressions, a time-out mechanism is used to abort execution after a user-configurable time limit has expired.
- **Evaluation allocates too much memory:** If a test expression uses up large amounts of memory, it could cause the IRULAN process to start thrashing, significantly degrading performance. To guard against this, IRULAN monitors the allocations performed by test expressions, and kills any test expression that allocates more than a user-configurable amount of memory.

3.6 Exploration Strategies

As discussed in Sections 3.2 and 3.4, the *Plan* followed by IRULAN has a series of non-deterministic choice points, denoted by diamonds in Figures 4, 5 and 7. For example, IRULAN has to choose the expression to be used to instantiate a strict argument to the function under test, or the argument to explore from a data constructor.

The state space defined by IRULAN's *Plan* is exponential in the number of choice points, and the strategy used to explore it has significant impact on the errors found and the coverage achieved by IRULAN in practice.

We have experimented with several kinds of exploration strategies in IRULAN. This includes the depth limited, depth first strategy used in Figure 2; a random strategy starting at the root of the *Plan*, choosing random branches, and restarting at the root if it hits a leaf (inspired by [6]); and an iterative deepening of the depth first strategy. After experimentation, we have determined that the iterative deepening strategy appears to be the most effective strategy for IRULAN and so this is the strategy used in our evaluation.

4. EXPERIMENTAL EVALUATION

Haskell has an established set of benchmarking programs, called *nofib* [20]. From nofib, we took the library components of its constituent programs for testing. The benchmark programs we used came from two suites:

- The `spectral` suite, consisting of the algorithmic cores of real programs, such as a chess end-game solver and a minimal expert system implementation. We tested 24 programs in this suite.
- The `real` suite, consisting of real Haskell programs. This includes implementations of a Prolog interpreter, a grep utility and an LZW compression tool. We tested 27 programs in this suite.

Table 1: Code size (in number of expressions, as reported by HPC) of the benchmark programs from the `real` and `spectral` suites.

# Expressions	Real	Spectral
0 - 100	0	2
100 - 1000	6	17
1000 - 10,000	19	4
10,000 - 100,000	2	1

In order to evaluate coverage, we used HPC [11], the standard tool used by Haskell programmers to record program coverage. HPC associates every expression in the program being tested (not including the standard libraries) with a "tick box". Everytime an expression is evaluated during program execution, its corresponding tick box is ticked. Expression coverage is the number of tick boxes ticked over the total number of tick boxes inserted.

Table 1 shows the approximate sizes of the 51 programs tested in these suites—in terms of number of expressions, as reported by HPC—after filtering out the modules that only export a *main :: IO()* function.[5] In Section 4.1, we report coverage results for these programs in terms of percentage of expressions executed.

We ran IRULAN with various configuration options on all the functions exported by the modules in these programs. In total we tested 4,030 different functions in 403 modules. For all runs, we configured IRULAN to use the constants 0, 1, and -1 of type *Int* and *Integer*, -1, 0, 0.5 and 1 of types *Double* and *Float*, and `'a'`, `'0'`, and `'\NUL'` of type *Char*, for a total of 17 constants. All experiments were run on a heterogeneous cluster of 64-bit Linux machines, most of which have dual core Intel Core^{TM}2 CPUs at 2-3 GHz, with 2 GB of RAM.

4.1 Coverage Results

Figure 8a gives the code coverage for the `spectral` suite, where each module in each program was tested for 5 minutes using iterative deepening. The average coverage per program is 70.83%, with a minimum of 18.48% (for `awards`) and a maximum of 97.86% (for `minimax`).

There are several reasons why IRULAN achieves such high coverage. For example, `minimax` is an implementation of a mini-max search tree for a noughts and crosses game. It has several top level functions that produce (using all the exported helper functions for creating boards and placing pieces) enumerations of the entire search space and that also evaluate that search space. Since IRULAN can test those top level functions thoroughly, all of the dependant source code also gets tested.

As another example, the reason why IRULAN achieves such high coverage for `fft2` (a Fourier transform library and application) is that `fft2` is mostly a numeric library that takes

[5]The modules we removed were determined by the following simple rule: if a module is called *Main* and only exports a function *main* :: *IO* () we removed it from our tests; if that *Main* was the only module in the benchmark program, we removed the program.

Int arguments, where our constants (-1, 0 and 1) are enough to trigger the different cases. IRULAN completely explores every exported function, the only un-executed code being an unexported and unreachable function that should be removed.

The reasons for which IRULAN achieves low code coverage for certain benchmarks vary. In many cases, our simplistic library filtering rule has meant that a program which has been factored into smaller parts that run in *IO* and are all exported is still being tested. Unfortunately (but by design), IRULAN does not perform any meaningful testing of functions that return an *IO* type. This is because *IO* values are opaque, non-referentially transparent operations representing system side-effects (e.g. writing to disk).

IRULAN could also provide greater functionality for creating support sets for type classes; this affects a few of the benchmark programs, such as `awards`.

Figure 8a also shows the effect that `case` statements (§3.3) have on coverage. With `case` statements disabled, the average coverage decreases from 70.83% to only 48.35%. We achieve similar results for `real`, with code coverage decreasing from 59.78% to 34.38%. In some extreme cases, the inability to use `case` statements prevents testing almost entirely: e.g., for `hartel` IRULAN achieves only 0.09% coverage, compared to 77.64% when `case` statements are used. This is because `hartel` consists of constant definitions which include large data structures containing lists of values. Without `case` statements, none of these wrapped values are explored.

Figure 8b gives the code coverage for the `real` suite after 1 second and respectively 5 minutes of testing. The average coverage per program after 5 minutes is 59.78%, with a minimum of 10.65% (for `maillist`) and maximum of 93.91% (for `cacheprof`). The `maillist` program (a mailing list generator) achieves such low coverage because it consists solely of a *Main* module that exports lots of functions that have an *IO* result type. The few constants and non-*IO* functions in the module are tested thoroughly by IRULAN, but they represent a very small amount of the application's code.

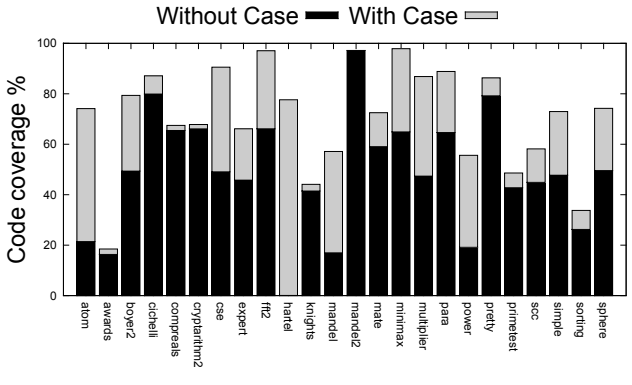
It is worth noting that testing each module for just 1 second achieves useful results, getting two thirds the coverage achieved after 5 minutes (40.62%).

4.2 Errors Found

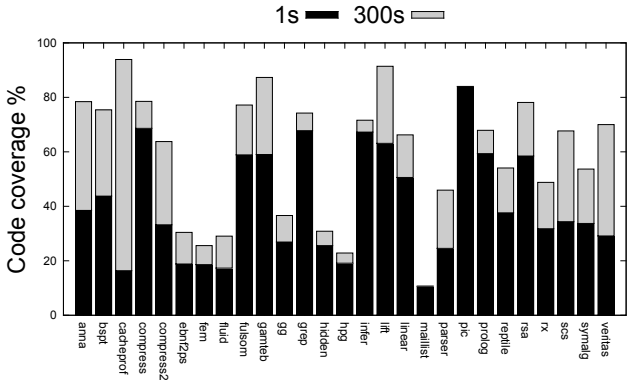
In this section, we present the main types of errors found by IRULAN in the *nofib* benchmarks. When run for 5 minutes per module, IRULAN reported 880,345 unique expressions that caused errors, spanning 47 different programs.

Given the large number of error expressions generated by IRULAN, the first step is to group them into clusters of unique errors. First, we group error expressions based on the type of exception thrown, e.g., *Non-exhaustive patterns* or *Prelude head* exceptions. Then, for those error types that include the location of the error, we group expressions based on the source location. For example, of the 191,388 error expressions generated for the `spectral` suite, 95,200 include source locations, which correspond to 37 unique program locations. (This is of course a rather crude method of grouping errors, as Haskell error messages do not contain the equivalent of stack traces [1]; we plan to address this issue in future work.)

As mentioned in Section 3.3, the use of `case` statements allows IRULAN to discover errors that would otherwise be left undetected. Of the 37 unique locations mentioned above, 7



(a) Spectral; effect of `case` statements.



(b) Real; effect of longer runtimes.

Figure 8: Code coverage as a percentage of expressions executed for the `spectral` and `real` suites.

of them were identified only by expressions with `case` statements in them.

We next give examples of some errors found by IRULAN:

Non-exhaustive pattern errors: these are errors in which the pattern matching of an expression reaches a case that the programmer has not considered. While some of the non-exhaustive pattern errors found involve relatively simple cases, IRULAN was also able to generate more complicated expressions that lead to a non-exhaustive pattern error in a function that is not mentioned in the expression. For example when testing `Game.hs` in `minimax` in `spectral`, IRULAN discovers the following:

```
case searchTree ?1 ([]) of
  Branch _ x -> x ==> !
    Board.hs:(34,0)-(36,35):
    Non-exhaustive patterns in function empty
```

The second argument to *searchTree* (the `[]`) represents a *Board*, which as a precondition is expected to have three elements in it. However, *searchTree* does not check this and happily returns a *Branch* value. It is only when that *Branch* is unpacked and the second argument to the branch inspected that the precondition violation results in an exception being thrown. This error also demonstrates a difficulty with working with Haskell, where laziness often causes errors to manifest themselves far away from their root cause.

Prelude.head (empty list) errors: these are errors

where the program tries to access an element from an empty list. For example, IRULAN discovered such an error in the `max_list` function of the `simple` program from the `spectral` suite. The function extracts the first element of the given list without checking that the list is not empty:

```
max_list :: [Double] → Double
max_list list
  = reduce_list max (head (list :: [Double])) list
```

Infinite loops and stack overflow errors: While IRULAN cannot detect these types of errors per se, cases in which the evaluation of an expression exceeds the resources allocated by IRULAN (see §3.5) are often indicative of pathological cases caused by bugs in the program.

In our experiments, the limits were set to 1 second and 128 MB of memory allocation per expression evaluation. These limits were exceeded 4,265 times: 143 times for the 1 second timeout, and 4,122 times for the 128 MB allocation limit. On further examination, we found that these events were often caused by missing base cases in the functions under test.

For example, consider the following code in the `primetest` program of the `spectral` suite:

```
log2 :: Integer → Integer
log2 = genericLength ∘ chop 2
chop :: Integer → Integer → [Integer]
chop b = chop' []
  where chop' a n = if n ≡ 0 then a
                              else chop' (r : a) q
    where (q, r) = n `divMod` b
```

IRULAN generates the expression `log2 (-1)` whose evaluation exceeds the allocation limit, and which in fact causes the code to loop indefinitely. The problem here is that the helper function *chop′* misses the base case for negative numbers.

5. EXTENSIONS TO IRULAN

Once we had built IRULAN, it became clear that the core could be applied to other types of testing. In this section we explore two such examples: property testing (§5.1) and regression testing (§5.2).

5.1 Property Testing

Testing of functional properties, also known as *property testing*, is available to Haskell programmers through tools such as QuickCheck [7], SmallCheck and Lazy SmallCheck [22]. In this section, we show that IRULAN can also be applied to property testing, and discuss its relative advantages and disadvantages for this problem domain.

Property testing is used to check that program invariants hold. For example, a programmer might write the property that list reversal is invertible as:

```
prop_reverse :: [Int] → Bool
prop_reverse xs = xs ≡ reverse (reverse xs)
```

We extended IRULAN so that if it tests functions that are prefixed with *prop_* and return a *Bool*, it will also report test cases that make these functions return *False*.

To find property violations, existing tools require the user to provide functions that facilitate the generation of test values for any custom data type that is needed for testing. At runtime, such tools use these functions to create test values, feeding them into property functions and reporting (typically using the *Show* instance for the test value) any values that make the test fail.

IRULAN provides two advantages over existing tools. First, its support set inference system means that it does not require the user to write any functions for generating test data. Second, IRULAN has the advantage of producing test *expressions* that are compiled on the fly into test values, whereas QuickCheck and (Lazy) SmallCheck produce test *values* directly. This means that the counterexample values presented by the other tools do not always explain how to construct the produced test values.

As an informal comparison between the different testing tools, we took the benchmark of properties made available in [22] and set up IRULAN, QuickCheck, SmallCheck, and Lazy SmallCheck to run them. Full details of these results are available on-line (§8). Currently the overhead of reflecting, dynamically compiling and running test expressions, as opposed to using type-class overloading to create values, means IRULAN is much slower when compared to the other tools. However, we believe the advantages of automatic argument discovery and of test expression generation outweigh the performance losses in many cases. We plan to investigate the possibility of designing a hybrid tool that combines IRULAN's set of support and reflection features with a runtime loop similar to the Check family of tools, in hope of eliminating the need for manually-written generation functions whilst obtaining better runtime performance.

5.2 Regression Testing

We have used IRULAN to find high-level regression bugs by cross-checking different versions of the same application. To compare two different versions, we first use IRULAN to generate a test suite for each, and then compare the two test suites to detect changes in behaviour.[6] Such changes are either made intentionally (in which case the test case generated by IRULAN can act as an illustrative example), or can indicate a bug introduced in the newer release.

Given two test suites, T_1 and T_2, we try to match input-output pairs in T_1 with corresponding pairs from T_2. Consider a pair (i_1, o_1) from T_1; there are several cases to consider:

1. There is exactly one test case (i_2, o_2) in T_2 where i_2 is identical to i_1. We then report a change in behaviour if o_1 and o_2 differ.

2. If i_1 contains ?s, then there may be many test cases (i_2, o_2) in T_2 such that i_1 is more general than i_2. By more general, we mean that i_1 is identical to i_2, except that where i_1 features a ?, i_2 may feature any subexpression. In general, this case shows a change in strictness of a function, which is indicative of an algorithm performing more work in T_2 than in T_1.

   We match the single more general test expression i_1 with all of its more specific instances in the second

[6] Alternatively, we could generate one test suite for one implementation and run it on the second implementation. However, this would lose information about new functions in the second implementation and, due to subtleties introduced by Haskell's laziness, would miss some of the errors that our approach can detect.

suite. We report an error if any of the specific output instances o_2 have a different result value from the output o_1.

3. The symmetric case, where i_1 is one of several test inputs in T_1 that are more specific than a single test input i_2 in T_2 is treated in a similar way.

4. The input i_1 may have no corresponding test input in the other suite. We report i_1 as unmatched by the other test suite. This is not necessarily an error, but should be brought to the attention of the programmer.

   This case can arise due to a complete change in strictness of the implementation of the function being tested e.g. *foo ? X ==> True* in T_1, but *foo Y ? ==>False* in T_2. This could also occur if one implementation has more functions to test, or provides more ways to create test data.

To ensure only one of the above cases is true, our test suites must obey the following invariant: If a test suite T features a test case with input i, then all other test inputs i' in T will be disjoint from i. That is, i will not equal, be more general or be less general than i'. We guarantee this by removing all the test cases that return ? from the test suites generated by IRULAN.

We applied this automated regression testing technique in two different contexts: an undergraduate Haskell programming exam (§5.2.1), and several libraries that had been uploaded to the Hackage library database (§5.2.2).

5.2.1 Undergraduate Programming Exam

The first year undergraduate Computing students at Imperial undergo a three hour on-line Haskell exam. This year the exam included implementing unification and the Martelli-Montanari polymorphic type inference algorithm for a small functional language. We used IRULAN's regression testing to cross-check the official sample answer with two PhD student answers before giving the exam to the students. All three solutions had passed a manually written test suite. The cross-checking revealed errors in all three solutions, including an infinite loop in an implementation of unification by one of the PhD students, and several subtle bugs in the implementation of type inference in the sample answer. The test questions, the three solutions, regression testing output and explanations can be found on-line (§8).

5.2.2 Hackage Libraries

Hackage[7] is a public collection of Haskell libraries and applications. Hackage retains snapshots of all previous versions of released software, and it also allows authors to provide links to home pages and version control repositories for the latest development versions of the libraries.

We took a selection of libraries from the Data Structures and Algorithms sections of Hackage, and built test suites for each exported module in their released versions. In addition, we also built test suites for the current development version, if available. We present our findings from two of these libraries.

TreeStructures: The TreeStructures library provides implementations of heap and tree data structures. Test suites for each exported module in each version of the library were built by running IRULAN for 60 seconds using iterative deepening with *Int* constants 0,1,2 and 3.

The comparison of the test suites revealed that the building of a binary heap from a list of elements had changed between two released versions. Upon closer inspection, the example inputs and outputs showed that the new implementation was not building balanced trees; and that this incorrect behaviour remained between the second release and the development version.

Using the version control history, we were able to work backwards from the example inputs and establish the commit that caused the bug to manifest, which was due to a contributed patch with the commit message *Changed definition of fromList (get rid of ugly lambda). Fixed heap.* Contacting the library author with the relevant examples revealed that the test cases do indicate a bug, and that the author had run a QuickCheck test suite prior to applying the patch, which didn't catch the bug.

Presburger: The Presburger library provides an implementation of a decision procedure for Presburger arithmetic. The released versions on Hackage are drawn from a published algorithm [8], however the latest development version has switched to an alternative approach. IRULAN identified several base cases to do with checking for the existence of numbers that divide by 0 or -1, where behaviour had changed: in particular, certain test cases which were returning a value in previous versions, now throw a *divide by zero* exception.

6. RELATED WORK

Like IRULAN, Lazy SmallCheck uses a form of *needed narrowing* [3] in order to avoid generating arguments that are not strict. This technique is well known in the functional-logic programming community, and has been employed in different settings. In particular, in the context of the Curry programming language, narrowing and coverage information have been used to generate regression test suites and to give a metric on their effectiveness [9]. Before Lazy SmallCheck, Lindblad [14] proposed a similar technique of instantiating lazy variables to generate property testing data, but for a small term language. Their technique, *lazy instantiation*, uses explicit unrefined meta-variables, represented by a `?`. These meta-variables are analogous to IRULAN's `?` arguments, and are where IRULAN drew the name from.

An alternative to the dynamic execution of expressions that IRULAN performs would be a static analysis of Haskell source code. Work in this area has proved successful, managing to analyse and prove safety and/or find bugs in several heavily used Haskell libraries and programs [16, 17]. Static analysis has the advantage that it can prove the safety of code, but in terms of bug-finding, the costs are false positives and the inability to generate actual test expressions that trigger the discovered errors.

The testing and model checking community uses the *small scope hypothesis* which states that testing with small inputs can give high confidence in the reliability of an implementation [2]. In relation to this work, IRULAN can be seen as testing the small scope hypothesis for Haskell programs.

There is a large body of work on testing imperative programs. Testing tools developed in the past for languages such as C and Java make use of a variety of techniques, including random testing [10, 18] systematic testing [2, 5], model checking [23, 24], and symbolic execution [6, 12]. IR-

[7] `http://www.hackage.haskell.org`

ULAN's building up of expressions and discarding those that cause errors (and not using them as arguments later) is analogous to part of the feedback loop used in the RANDOOP tool [19], although RANDOOP (and related tools such as ECLAT [18]) also have ways of expressing pre and post conditions (filters and contracts), which IRULAN currently lacks and would be useful for pruning the number of error expressions presented to the user. It is interesting to note here that work such as RWSet [4] and Korat [5] try to prune large parts of the program state space by monitoring the values read by the program, in order to avoid constructing values that will not be subsequently read. This is somewhat equivalent to inferring the values that are *lazy*, which comes almost for free in a non-strict language like Haskell.

7. CONCLUSION

We have presented IRULAN, a black-box unit testing tool designed to automatically generate high-coverage test suites, and find programming errors in Haskell library code. Our experience building IRULAN and applying it to real Haskell code has demonstrated that: (1) the static types attached to the functions to be tested can be used to guide test data generation; (2) lazy evaluation can be exploited to prune large parts of the search space; and (3) taking apart result values from functions using **case** expressions can dramatically increase code coverage in a lazy language like Haskell.

We have applied IRULAN to over 50 benchmarks from the *spectral* and *real* suites of the *nofib* benchmarks and showed that it can effectively generate high-coverage test suites—exhibiting 70.83% coverage for *spectral* and 59.78% coverage for *real*—and discover common programming errors in these programs, such as non-exhaustive patterns errors, extracting elements from an empty list, infinite loops, and stack overflows.

Furthermore, we have adapted the core of IRULAN to perform property and regression testing and demonstrated IRULAN's ability to find high-level errors by cross-checking different releases of several third-party libraries.

8. AVAILABILITY

IRULAN is freely available, together with our experimental evaluation, property checking experiments and regression testing case studies, at the following URL:

`http://www.doc.ic.ac.uk/~tora/irulan`.

Acknowledgements

The authors would like to thank the SLURP reading group and especially Sophia Drossopoulou for lengthy and insightful discussions during the development of this work. Many thanks also to our anonymous reviewers, and Petr Hosek, Will Jones, and Paul Marinescu for detailed comments on the paper.

9. REFERENCES

[1] T. Allwood, S. Peyton Jones, and S. Eisenbach. Finding the Needle: Stack Traces for GHC. In *Haskell*, 2009.

[2] A. Andoni, D. Daniliuc, S. Khurshid, and D. Marinov. Evaluating the "Small Scope Hypothesis", 2002.

[3] Antoy, Echahed, and Hanus. A Needed Narrowing Strategy. *JACM: Journal of the ACM*, 47, 2000.

[4] P. Boonstoppel, C. Cadar, and D. Engler. RWset: Attacking Path Explosion in Constraint-Based Test Generation. In *TACAS*, 2008.

[5] C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated testing based on Java predicates. In *ISSTA*, 2002.

[6] C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *OSDI*, 2008.

[7] K. Claessen and J. Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In *ICFP*, 2000.

[8] D. C. Cooper. Theorem proving in arithmetic without multiplication. In *Machine Intelligence*, 1972.

[9] S. Fischer and H. Kuchen. Systematic generation of glass-box test cases for functional logic programs. In *PPDP*, 2007.

[10] J. E. Forrester and B. P. Miller. An empirical study of the robustness of windows NT applications using random testing, 2000.

[11] A. Gill and C. Runciman. Haskell Program Coverage. In *Haskell*, 2007.

[12] P. Godefroid, M. Y. Levin, and D. A. Molnar. Automated Whitebox Fuzz Testing. In *NDSS*, 2008.

[13] B. Korel. Automated test data generation for programs with procedures. In *ISSTA*, 1996.

[14] F. Lindblad. Property directed generation of first-order test data. In *TFP*, 2007.

[15] B. P. Miller, L. Fredriksen, and B. So. An empirical study of the reliability of UNIX utilities. *CACM*, 1990.

[16] N. Mitchell and C. Runciman. A Static Checker for Safe Pattern Matching in Haskell. In *TFP*, 2007.

[17] N. Mitchell and C. Runciman. Not All Patterns, But Enough - an automatic verifier for partial but sufficient pattern matching. In *Haskell Symposium*, 2008.

[18] C. Pacheco and M. D. Ernst. Eclat: Automatic generation and classification of test inputs. In *ECOOP 2005*.

[19] C. Pacheco, S. Lahiri, M. Ernst, and T. Ball. Feedback-directed random test generation. Technical Report MSR-TR-2006-125, Microsoft Research, 2006.

[20] W. Partain. The nofib benchmark suite of Haskell programs. In *Functional Programming*, Workshops in Computing, 1992.

[21] S. L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice–Hall, 1987.

[22] C. Runciman, M. Naylor, and F. Lindblad. SmallCheck and Lazy SmallCheck: Automatic exhaustive testing for small values. In *Haskell Symposium*, 2008.

[23] W. Visser, C. S. Pasareanu, and S. Khurshid. Test input generation with Java PathFinder. In *ISSTA*, 2004.

[24] J. Yang, P. Twohey, D. Engler, and M. Musuvathi. Using Model Checking to Find Serious File System Errors. In *OSDI*, 2004.