Imperial College London
Department of Computing

Finding The Lazy Programmer's Bugs

Tristan Oliver Richard Allwood

Submitted in part fulfilment of the requirements for the degree of
Doctor of Philosophy in Computing of the Imperial College London and
the Diploma of the Imperial College, 2011

Abstract

Traditionally developers and testers created huge numbers of explicit tests, enumerating interesting cases, perhaps biased by what they believe to be the current boundary conditions of the function being tested. Or at least, they were supposed to.

A major step forward was the development of property testing. Property testing requires the user to write a few functional properties that are used to generate tests, and requires an external library or tool to create test data for the tests. As such many thousands of tests can be created for a single property. For the purely functional programming language Haskell there are several such libraries; for example QuickCheck [CH00], SmallCheck and Lazy SmallCheck [RNL08].

Unfortunately, property testing still requires the user to write explicit tests. Fortunately, we note there are already many implicit tests present in programs. Developers may throw assertion errors, or the compiler may silently insert runtime exceptions for incomplete pattern matches.

We attempt to automate the testing process using these implicit tests. Our contributions are in four main areas: (1) We have developed algorithms to automatically infer appropriate constructors and functions needed to generate test data without requiring additional programmer work or annotations. (2) To combine the constructors and functions into test expressions we take advantage of Haskell's lazy evaluation semantics by applying the techniques of needed narrowing and lazy instantiation to guide generation. (3) We keep the type of test data at its most general, in order to prevent committing too early to monomorphic types that cause needless wasted tests. (4) We have developed novel ways of creating Haskell case expressions to inspect elements inside returned data structures, in order to discover exceptions that may be hidden by laziness, and to make our test data generation algorithm more expressive.

In order to validate our claims, we have implemented these techniques in IRULAN, a fully automatic tool for generating systematic black-box unit tests for Haskell library code. We have designed IRULAN to generate high coverage test suites and detect common programming errors in the process.

Acknowledgements

I would like to thank my supervisors, Professor Susan Eisenbach and Professor Sophia Drossopoulou for their advice, guidence, patience and expertise throughout my time at Imperial. Having such knowledgeble advisors so interested in my work has been a constant source of encouragement.

Dr Cristian Cadar has also been a valuable mentor during the later phases of my research. I would like to thank him for the time taken to help thoroughly revise earlier draft papers on this subject, for bringing his experience with testing to the table, and suggesting refinements to experiments to focus on for my evaluation.

I would also like to thank the SLURP research group for interesting and wide ranging discussions, as well as for critiquing my ideas, papers and talks during the PhD.

For teaching me everything that is beautiful about Haskell, I would like to thank Dr Tony Field. Then later, during my internship at Microsoft Reasearch, I learned everything else about Haskell. My thanks to Simon Peyton Jones, Simon Marlow, Thomas Schilling and Max Bolingbroke for making that an enjoyable and thought provoking experience.

In addition, I have had the pleasure to meet and work with many other friends at Imperial. Francis, Matt, Marc, Steve, Dave, Khilan, Max, Mike, Azalea, Will Jones, Will Sonnex, Rob and Pedro, thank-you all for stimulating discussions, drinks, wipeboard based arguments and for helping to make research fun.

Part of this thesis involved a case study of released Haskell code, and as part of that I contacted Sterling Clover, Brendan Hickey and Iavor Diatchki for feedback on the discoveries I'd made. I'd like to thank them for taking the time to reply and provide useful information for my research.

I am grateful to my examiners, Professor Kevin Hammond and Professor Colin Runciman, for their time, detailed comments, probing questions and interesting discussion during my Viva.

Finally, I'd like to thank my other friends, Jon, Matt and Sophie for keeping me sane; Hadrian for keeping me grounded, and my parents for all their love and support.

'Paul's attention came at last to a tall blonde woman, green-eyed, a face of patrician beauty, classic in its hauteur, untouched by tears, completely undefeated. Without being told it, Paul knew her — Princess Royal, Bene Gesserit-trained, a face that time vision had shown him in many aspects: Irulan. There's my key, he thought.'
Frank Herbert, Dune

Contents

List of Tables

List of Figures

Chapter 1

Introduction

A lot of software development is based around small units of code that integrate together existing libraries to create new functionality. In addition, common combinators, code patterns, and data structures may be abstracted into utility libraries. This separation of application and library means that developers do not need to repeatedly solve the same problems in every project, which hopefully saves development time costs.

Unfortunately, where there is a problem with an application that includes library code, it may be quite difficult to debug the source of the problem. The error could be in the programmer code, a hidden bug in the library itself, or due to the fact that the programmer has misunderstood (or is just not aware of) assumptions that the library makes about its inputs and environment.

Debugging a programmer's application code is tricky in itself, but having the additional burden of libraries that may not be under the programmer's control can make things especially difficult. Determining the source of a problem which uses libraries to which one may only have an API and not the source code can be extremely challenging.

This work aims to provide support for such a scenario. Testing proceeds by creating expressions to test each of the programmer's functions in isolation. Test expressions that are produced are small and should only feature inputs that are needed to reproduce the bug. Should a bug (in our case, characterised by an uncaught exception) be found, the generated test case can be used to establish the precise location of the fault.

One hopes that a corollary of this is that this technique is then applied to libraries before they are released to be used by others. In this manner bugs present in the library can be fixed before they are discovered by application programmers.

An overriding goal of this work is to enable testing without placing any extra burden on the programmer. In practice this means we do not wish to modify the source code of the libraries being tested. We believe that a programmer already states many implicit assumptions about their code through defensive programming checks and compiler inserted error messages. These implicit assumptions are used to generate tests. We do not require the programmer to provide an oracle to classify the test cases generated a priori, we assume that it is easier for

the user to look at the test cases and the exceptions they throw and classify them after the fact. However, if the user wishes to write oracles in the form of properties to check, our techniques are general enough to perform property checking.

Such automation is desirable because writing enough tests to provide confidence in a given piece of code is difficult and time consuming. The tests themselves have to be checked for accuracy, and validation needs to take place to ensure they cover the intended parts of the code base. If the code is not trivial, it probably isn't even possible to do this in a realistic time frame. This has led to a large body of work on automatically generating test suites in the imperative programming community [MFS90, BKM02, CDE08, GLM08, VPK04, Kor96].

The Haskell programming language contains a variety of interesting language features, which present unusual challenges and opportunities for automatic testing. These include a rich and expressive polymorphic static type system, higher order programming using first class functions as values, and laziness.

The need for testing and debugging support in Haskell has been recognized already. The "flagship" Haskell compiler, GHC, has built-in support for compiling Haskell programs with code-coverage instrumentation [GR07]. In addition, testing of functional properties is currently available and widely adopted through tools such as QuickCheck [CH00] SmallCheck and Lazy SmallCheck [RNL08]. These tools require the programmer to make source-level changes to their code and to express explicitly the properties they wish to check.

Although arguably a good discipline, adding explicit properties is extra work for programmers. Haskell programs already contain many invariants and properties that must hold at runtime. For example, 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 as a releasable API, the programmer may wish to ensure that a user of their API can't trigger these assertions and exceptions.

In this work, we present novel algorithms and data structures, realised as a usable tool, IRULAN [ACE], that can automatically detect such violations in Haskell programs. In addition, IRULAN can automatically generate test suites that achieve high coverage in the modules being tested.

The user of IRULAN specifies a module to test at which point IRULAN will automatically generate a large number of expressions that test functions exported by that module. IRULAN then executes these expressions, reporting to the user those that elicit runtime exceptions.

IRULAN is designed to perform systematic black-box unit testing of Haskell libraries. It aims, through automatic code generation and execution, to see if there are any possible uses of a library's API that cause uncaught exceptions to be thrown. IRULAN supports (and takes advantage of) the most important features of the Haskell language, such as lazy evaluation and polymorphism. The large numbers of test expressions generated by IRULAN are also diverse; IRULAN uses various exploration strategies to effectively explore the state space of the programs being tested. Using the most successful strategy of iterative deepening, we evaluate IRULAN on over 50 benchmarks from the nofib suite [Par93] and show that it can effectively find errors and generate high-coverage test suites for the programs therein.

1.1 Goals and Contributions

The goal of this thesis is to explore the space of automated, black-box testing in the context of Haskell. In particular we wanted to take existing ideas and build upon them further. Our goals are to answer the following questions, and our main contributions are our answers:

- **Can we automatically infer good quality test data for use in testing functions?**

  Existing testing tools for Haskell all require some form of modification to the original sources. These changes are needed to inform the testing tool with recipes for creating test data.

  However, Haskell programs contains lots of static type information; and using that we were able to automatically infer (most of) the test data needed for testing functions. We show how we automatically collect appropriate constructors and functions that can be used to build up arguments to functions. In many cases this is all that is needed for testing, however when non-inductive values (such as integers or characters) are required, we allow the user to specify them to extend the available test data.

- **What kind of testing can you do without requiring the programmer to provide explicit assertions?**

  Automated testing techniques require some way to evaluate if a test case has passed or failed. For Haskell, popular testing libraries require the user to write the test themselves and assert if the test passed or failed (unit testing), or to write a *property function* that accepts some test data and returns a boolean indicating whether the test was passed using that test data (property testing). We wanted to explore what, if anything, can be tested for free, without requiring the user to explicitly state tests or properties.

  In this thesis we show that by applying functions that are in the user code to generated arguments and monitoring their evaluation, we can discover several classes of programming error. For example, incomplete implementations manifest through the catching of Non-exhaustive Patterns errors, and missing base cases in recursive programs can cause infinite loops that are caught by looking for high memory usage or an excessive runtime.

  We present an algorithm, based on the idea of needed narrowing (and similar in spirit to the one used by Lazy SmallCheck), that can apply arbitrary Haskell functions to test data. However we go beyond existing tools by allowing the testing of polymorphic functions, and consider the interaction of needed narrowing with polymorphic variables.

- **During testing, how important is it to ensure composite values are inspected deeply?**

  In Haskell, data values are lazy by default. This means that evaluating the root of a data structure will not necessarily force the evaluation of its children. In theory this means that just because a function returned a result successfully, does not mean that some part of the function didn't crash, because some parts of its evaluation may not have happened yet.

  As part of this thesis we investigated looking for errors inside returned data values. We demonstrate that synthesising selectors using case expressions to destruct data is useful for testing a lazy functional

language, and can substantially increase the test coverage of a library.

- **How should we explore the search space for test cases?**

  Test cases are built from a root function to test, and some generated test data that is to be used for the root function's arguments. There may be many, even infinite number of ways to apply the function to arguments.

  In this thesis we present evidence supporting the notion that, when testing for a fixed amount of time, iterative deepening is the most effective exploration strategy.

- **Can we build a prototype, autonomous testing tool to realise these ideas?**

  In order to validate our claims of usefulness, we would need an implementation that could be used to evaluate our ideas by testing real Haskell code.

  In this thesis we present a tool, IRULAN, and the algorithms and data structures that underpin it. We then present several practical experiments performed with IRULAN to look at the effectiveness of our techniques.

  Our IRULAN tool was designed to be an extensible platform, upon which we could explore some more experimental ideas. One extension was to look at regression testing, and we present several case studies using this extension that provided useful results in the real world.

1.2 Outline and Chapter Contributions

Chapter 2 outlines relevant related work, highlighting points of departure from existing techniques and those we explore here.

Chapter 3 gives an overview of different use cases of our IRULAN tool, to motivate the underlying problems we solved and applications of our ideas.

Chapter 4 outlines the implementation details of IRULAN. Here we give novel algorithms for building, organising and executing tests for Haskell functions in an autonomous manner. The main contributions of this chapter are:

- **The automatic inference of a Support Set**. IRULAN will need a set of identifiers (the support set) which it can use to create arguments for the functions being tested. This section will discuss how IRULAN can automatically infer this set of identifiers.
- **The *TypeMap*** is a novel data structure that allows polymorphic types to be used as keys in a map. Searching for a type in this map will find all values whose keys unify with the type being queried. This is used for finding functions that can be used to build arguments to the function currently being tested.
- **The Constructor Graph** is a data structure used to express (the possibly) infinite paths through constructors using case expressions, permitting the use of case expressions over composite data types in order to produce test data for the function under test.

- **The *Plan* algorithm** builds a lazy, tree like data structure that holds the current test expression to run, and, based on the result of running the test expression, what test expressions to run after it. The *Plan* encodes testing a function using a needed narrowing or lazy refinement like strategy, but adds novel features for building test data using case expressions and testing in the presence of polymorphic functions.

Chapter 5 contains our experimental evaluation. Our main experiments run IRULAN on two large existing benchmark suites. We then perform two smaller experiments to motivate configuration choices made in the larger benchmarks. The main contributions of this chapter are:

- **nofib**. We run IRULAN on the spectral and real suites of the nofib benchmark, reporting code coverage achieved on the programs contained therein and discussing the kinds of errors found by IRULAN.
- **Property Testing**. We take the benchmark comparing QuickCheck, SmallCheck and Lazy SmallCheck from [RNL08] and test IRULAN with it, reporting comparative code coverage from all four tools.
- **Runtime Caches**. We hypothesise that referential transparency could provide some optimisation opportunities through the use of caches. However our experiment shows that the overheads of using caches are in general higher than the savings we could achieve.
- **Search Strategies**. We look at the performance of different search strategies, and introduce a strategy "time split", which is a variation of depth first search, that performs well for runs of a fixed length of time.

Chapter 6 concludes with a summary of the achievements of this thesis, and a discussion on possible future directions for this work.

1.3 Statement of Originality

The implementation of IRULAN and the algorithms presented in this thesis are my own work.

Professor Susan Eisenbach, Dr. Cristian Cadar and I co-authored draft papers on IRULAN. Parts of these papers have been worked into Chapter 1, Section 3.1, Chapter 4 and Sections 5.1 and 5.5. The technical contributions of the papers are all my own.

Professor Eisenbach and I co-authored a paper on previous work describing test generation in the context of Java [AE09]. This was presented at LDTA '08 and later published in ENTCS. Extracts from this paper are included in Section 2.1.1.

Professor Sophia Drossopoulou gave suggestions for the formalisation of the lookup function in Section 4.1.3.

Professor Eisenbach has also proof read and contributed detailed suggestions throughout this thesis. Dr Tony Field and William Jones also proof read and contributed suggestions towards Chapter 1 and Chapter 6. Any mistakes remaining are my own.

Chapter 2

Background

In this chapter we give the background that led up to our work §2.1, discuss in detail relevant related work on the subject of testing §2.2, before closing by putting our work in context with respect to the related work §2.3.

2.1 Origins

This thesis is about black box testing for Haskell programs, however this is not the problem that we started with. We have spent some time looking into black box regression testing for Java compilers. This work enabled us to gain experience with working with state space explosion that occurs when generating programs or expressions, and the importance of pruning the search space as aggressively as possible. We also gained experience in using code coverage as a means to evaluate black box testing techniques.

We then, under the direction of Professor Simon Peyton Jones, investigated adding a stack tracing mechanism to the Glasgow Haskell Compiler, GHC. The work with GHC allowed us to understand the capability of GHC to not only act as a compiler, but also a reflection library to access the API of compiled Haskell code. Thinking of applications in terms of testing, we were led to the black box work of Irulan which we present in the rest of this thesis.

2.1.1 Tickling Java with a Feather

In earlier work [AE09], we investigated automated testing in the context of Java. We present here an abridged version of the paper. The work was motivated around the following observation:

> Many programming languages have been given a formal presentation; either in their entirety, or for a semantically meaningful core subset. This formalism is used to prove desirable properties of the semantics of the language, both static and dynamic. However, the language is only proved safe in theory - we still rely on a correct implementation of the compiler of the language.

Figure 2.1 Syntax of Featherweight Java

`CL ::=`	*class declarations:*
`class C extends C {`$\overline{\texttt{C f}}$`; K` $\overline{\texttt{M}}$`}`	
`K ::=`	*constructor declarations:*
`C(`$\overline{\texttt{C f}}$`) {super(`$\overline{\texttt{f}}$`);` $\overline{\texttt{this.f=f;}}$`}`	
`M ::=`	*method declarations:*
`C m(`$\overline{\texttt{C x}}$`){return t;}`	
`t ::=`	*terms:*
`x`	*variable*
`t.f`	*field access*
`t.m(`$\overline{\texttt{t}}$`)`	*method invocation*
`new C(`$\overline{\texttt{t}}$`)`	*object creation*
`(C) t`	*cast*

The work asked whether the formal presentation of the theory could also be used both as input to generate test programs and to be an oracle for them. The test programs can then be executed by the implementation of the full language to see if it conforms with the theory. We wanted to evaluate how useful these test programs would be in practice.

To start answering the question, we presented an investigation into using the theory of Featherweight Java [IPW99] to create tests for the type checker component of the OpenJDK Java compiler [Mic].

FJ is designed to be a minimal calculus for Java. The authors omitted as many features as possible while still retaining the core Java typing rules. This included omitting assignment, making FJ a purely functional calculus. FJ has its own syntax (Figure 2.1) which is a restricted subset of Java - all syntactically valid FJ programs are syntactically valid Java programs.

We generated test Java programs that were instances of the FJ grammar. We instantiate the grammar by walking it using a bounded, depth-first exploration algorithm. We use structural constraints limiting the maximum number of classes, the number of fields and methods per class, and the complexity (sum of all production rules used) of expressions in each method and the number of variables used in a method, to ensure the depth first exploration does not explore an infinite space. However the grammar of FJ also makes reference to potentially infinite domains for class names (`C`), variable names (`x`), method names (`m`) and field names (`f`). For the depth first exploration algorithm to function effectively, it requires a bounded domain for each of these infinite domains. The simple solution to this is to create constraints for the number (and names) of valid class/method/field/arguments, and whenever (for example) a class name is required in a program, n copies of the program are produced, each using a different substitution from the n available class names.

This approach will specify many programs that are isomorphic or α-equivalent to each other. For example:

```
P1:
class C1 extends Object { C2 { super(); } }
class C2 extends Object { C1 { super(); } }
```

and

```
P2:
class C2 extends Object { C1 { super(); } }
class C1 extends Object { C2 { super(); } }
```

If we can assume that the internal representation of names in the Java compiler doesn't try to inspect their values (except to compare them to each other and some built-in values such as `Object` or `this` using library methods), then we can prune away many of the isomorphic programs. To do this, we augmented FJ with a notion of *binding*. Class, field and method declarations can be thought of declaration sites for new class, field, method and argument names, and so for each site in the program, we invent new names that will be available. Program generation now happens in two phases, the first generates a skeleton with the structure of the class, field and method declarations, then the skeletons are instantiated with expressions and references to names that the skeleton makes available.

The generated FJ programs by themselves are not very useful, as they are just programs. For them to become tests, they need to be associated with an expected result for running the test against `javac`. The expected result is provided by an oracle, in this case we have used an implementation of FJ's type checker [AE].

To help ensure our oracle is correct, we have used our generated test programs to check that it gives the same outputs as another implementation of FJ. Given the Java compiler we have chosen to test, we also expect that the implementation of `javac` is actually correct - so the oracle should agree with it in most cases (which it does). However there are some cases where the FJ oracle and `javac` do not agree.

To be as exhaustive as possible, we want to generate both positive and negative test programs for `javac`; i.e. tests that we expect to type check and tests we expect to be rejected. However we have had to be careful. FJ type checking rules on FJ programs are sound w.r.t. Java. If FJ statically accepts a program, we expect Java to accept it. However there are FJ programs that FJ statically rejects that Java will accept. For example, Java supports covariant returns in overridden methods and does not require non-final instance fields to be initialized in constructors, whereas FJ would reject programs that contained these features. There are also some classes of program where the reason FJ rejects the program is strong enough to say Java should reject it too. For example creating cycles detected in the class hierarchy or trying to declare a class named `Object` are always errors in both FJ and Java programs.

When applying the oracle to the test programs, we check whether the test program type checked or not. If it failed to type check we only pass it to `javac` if it was rejected for a reason we would expect `javac` to reject it for (e.g. there was a cycle in the class hierarchy). In this way, we are only testing `javac` (or, in the experiment run here, collecting coverage on) with programs that we can check `javac` agrees with our expectations.

We generated several suites of test FJ programs, and ran them against code-coverage instrumented versions of an FJ type checker and the OpenJDK `javac` compiler.

Using large numbers of very small and simple Featherweight Java programs, we can achieve a test coverage of around 80% - 90% of an FJ type checker. Adding the programs that were filtered out because they were

possibly correct Java programs but incorrect FJ programs hardly increases the code coverage at all. This may be because the constraints keeping the problem "small scope" and hence tractable are too limiting; for example none of our tests create method calls featuring two arguments. Achieving a near-100% code coverage for FJ type checking is a future goal, that requires a more structured approach to generating tests that doesn't have the explosion in the state space we currently experience.

The same tests run on the OpenJDK `javac` correspond to exercising 25% - 30% of the code base of a full, industrial strength Java implementation. The results indicate that the recursive nature of expressions, and the associated recursive implementation in compilers, means that testing using lots of small expressions can be effective. It is this feature that we decide was useful, and it became one of the underlying design decisions of IRULAN.

2.1.2 Finding the Needle: Stack Traces for GHC

We were fortunate to be able to undertake an internship at Microsoft Research Cambridge, under the instruction of Simon Peyton Jones. During this time, we investigated adding the ability to get lexical call stack traces out of Haskell programs. As this subject was not directly related to test generation we will refer the interested reader to our Haskell Symposium paper [APJE09] for the details.

One lesson we learned from the stack traces work is that automation is a useful feature of a tool in practice. If the programmer can run their application with or without stack traces by just toggling a compiler flag, as opposed to needing to change imports or re-write their code, it will make a big difference in practice to uptake and ease of use. Enabling this kind of automation and still having a practical tool is then an interesting research topic, and in the work presented in this thesis, we made it a goal to automate what we could during testing.

The stack traces work did enable us to gain an understanding of the internal workings of the Glasgow Haskell Compiler, GHC. One insight quickly realised was the GHC can also be used as a powerful library, to enable a Haskell program to introspect and access the exported identifiers and reflect on their types in compiled Haskell code. We realised that this ability could be leveraged to enable the automated discovery of Haskell identifiers of certain types, and to enable the dynamic construction and execution of Haskell expressions, without needing any source code modifications on the part of a normal programmer.

A motivating example for the stack trace work is identifying the cause of `Prelude.head: empty list` errors. These are a common example of difficult to debug errors that can arise in Haskell applications. They occur when the `Prelude` function *head* (which extracts the first element from a list) is applied to an empty list. Since it is not possible to extract an element from an empty list, the *head* function fails by throwing an exception. However, since there is no lexical call stack maintained in a normal Haskell runtime, it is not possible to (easily) ascertain which use of *head* caused the exception to be thrown. The stack traces work aimed to improve the situation by rewriting the application to build up lexical call stacks at runtime.

However, we realised that there is another approach to identifying the causes of exceptions. If the application's individual functions are tested under a range of inputs, should any inputs produce a `Prelude.head: empty list`

exception, then the programmer has a concrete (and hopefully small) example to work forward through the application to establish why the error was thrown. This approach could then be combined with a stack tracing approach to work backwards to help narrow the cause even further. It also brought to the forefront the interesting challenges (and potential for research) that lazy evaluation in Haskell provides, which we decided to explore further in the context of test generation.

2.2 Related Work

Given our interest in black box testing of Java programs, and our newly discovered knowledge of a major Haskell compiler, we decided to investigate automated testing in the existing research literature. We present some of that work here.

We focus first on two key areas relevant for this thesis: how search spaces are explored in test generating software §2.2.1, and how error conditions can be found automatically §2.2.2. We then consider some other related techniques and tools that are useful once a bug has been found §2.2.3 as part of a debugging effort.

2.2.1 Exploring the Search Spaces of Testing

We are interested in automatic test case generation. When testing a program, or a property, we will want to enumerate many test cases to see if we can find a test case that exhibits a fault, an erroneous computation or a violation of an invariant. For example, in the context of testing functional properties (functions that take arguments and return a Boolean), we would want to enumerate expressions representing invocations of that function, to see if any of the enumerated arguments make the function return false.

We first consider existing work, and the different approaches they take to enumerating these large and sometimes infinite spaces of expressions. Then we look at needed narrowing, an evaluation mechanism from functional logic that suggests some optimisations for enumerating the space in the context of a lazy functional language, and work related to it. Finally we look at an interesting approach to defining criterion for knowing when enough testing has been performed.

Enumerating Search Spaces

QuickCheck

One of most popular testing libraries for Haskell is QuickCheck [CH00]. This is a very lightweight tool for testing functional properties. At their core, properties are functions that take some arguments and return a Boolean result; the aim of the library is to enumerate arguments to see if any will make the function return false, and to then present such arguments as a counterexample to the property.

QuickCheck relies on type classes to detail how to generate test data. The test data is generated randomly, but the user has to declare how to generate the data for their own data types by implementing a type class *Arbitrary*. When implementing *Arbitrary* users can specify the distributions that choose how to generate their test data, and there is library of distributions and ways of making choices built into QuickCheck that make this easy.

QuickCheck can then repeatedly generate random test data, typically until some limit on the number of test cases to generate has been reached, or until it finds a counterexample.

However this approach leads to some problems. The first is that many properties are implemented with an implication at the root, meaning they have a precondition. If many test cases fail the precondition, then they will be counted as a successful test even though no meaningful testing took place.

For example, consider a function *insertSorted* that inserts an element at the right place in a list to keep it sorted. A typical implementation of a property to check for this could be:[1]

$$
\begin{aligned}
&\mathit{prop\_insertSorted} :: [\mathit{Int}] \to \mathit{Int} \to \mathit{Bool} \\
&\mathit{prop\_insertSorted}\ \mathit{xs}\ x \\
&\quad = \mathit{sorted}\ \mathit{xs} \to \mathit{sorted}\ (\mathit{insertSorted}\ x\ \mathit{xs}) \\
&\quad \textbf{where} \\
&\qquad \mathit{True} \to b = b \\
&\qquad \mathit{False} \to b = \mathit{True}
\end{aligned}
$$

Many inputs could be generated that do not satisfy *sorted xs*, and so the property will trivially be *True*. To aid the user in understanding and better expressing their intent, QuickCheck adds a small domain specific language, *Property*, that a property function can use, and features several useful combinators. For example, the two argument combinator ==> takes a precondition and a real test, and captures the intent that should the precondition fail, the input should not be considered a test at all. There are also combinators to label and classify test data passing through the test, which QuickCheck can then report at the end of a test run. This way, a user can visualise the distributions of data that has been generated, to sanity check that meaningful testing is taking place.

For example, the above example can be rewritten to use ==> and also keep track of the length of the successfully tested sorted lists:

$$
\begin{aligned}
&\mathit{prop\_insertSorted} :: [\mathit{Int}] \to \mathit{Int} \to \mathit{Property} \\
&\mathit{prop\_insertSorted}\ \mathit{xs}\ x \\
&\quad = \mathit{sorted}\ \mathit{xs} \Longrightarrow \mathit{collect}\ (\mathit{length}\ \mathit{xs})\ (\mathit{ordered}\ (\mathit{insertSorted}\ x\ \mathit{xs}))
\end{aligned}
$$

Where the output from running this may produce:

```
OK, passed 100 tests.
```

[1] This and the following example are adapted from [CH00].

```
49% 0.
32% 1.
12% 2.
 4% 3.
 2% 4.
 1% 5.
```

Here 100 lists that pass the *sorted* precondition have been tested, however 49 of them were the empty list. In this case, a more specialised random generator specifically for creating sorted lists may be more appropriate to use, but would require extra work by the user of QuickCheck to implement.

One important feature of Haskell is the ability to perform higher order programming, such that functions can accept other functions as arguments. Sometimes therefore, you may want test data that takes the form of a function. QuickCheck supports creating higher order functions which accept arguments of a certain type if it is provided with an instance of a type class for that type called *Coarbitrary*. The user is required to instantiate a function that accepts the argument of the higher order function, a random generator for the result type, and then return the random generator with its seed varied according to the value of the first argument. The intention is that different inputs to the higher order function seed the generator (and so the result value) in different ways.

QuickCheck's small and lightweight approach has helped it become a very popular and successful tool for the Haskell community. With a focus on random testing, the authors have presented many useful primitives, combinators and patterns for creating and debugging random test data generators. However there is still a burden on the user of the library to specify these generators, and to understand what are appropriate random generators to use.

SmallCheck

Other authors [RNL08] have also noted that knowing the appropriate random distribution for test data is a fine art. They also note that the small scope hypothesis [Jac06] states that if a bug exists in a program, then a small test case will likely be able to expose it. They therefore present SmallCheck (and later Lazy SmallCheck which we discuss below), which exhaustively enumerates and tests all values up to a given depth limit. The depth bounded search can also be repeated at increasing depths to give an iterative deepening search.

The implementation uses a lightweight, type-class based approach, similar to QuickCheck. However instances of SmallCheck's *Serial* type class (analogous to QuickCheck's *Arbitrary*) only need to provide a mapping from a depth limit to a finite list of all values of that type within the depth limit. By providing several useful combinators, SmallCheck makes these instances straightforward to derive, and in many cases they could be written mechanically.

Testing properties that use implication or have preconditions are also interesting in SmallCheck, but for a different reason than in QuickCheck. Consider another variant of our *prop_insertSorted* example again:

$$prop\_insertSorted :: [Int] \to Int \to Property$$
$$prop\_insertSorted\ xs\ x$$
$$= sorted\ xs \Longrightarrow ordered\ (insertSorted\ x\ xs)$$

SmallCheck also makes available a *Property* type and ==> combinator, but the behaviours are different to those in QuickCheck. SmallCheck uses *Property* to be able to maintain separate counts of tests that passed or failed the precondition. However, SmallCheck will still enumerate all inputs within the depth limit for testing the function. QuickCheck runs extra tests if some fail the precondition because it is running a user configurable number of tests, whereas SmallCheck is testing all expressions within a depth limit.

However, there is still some subtlety to using ==>. In the above example, all combinations of xs and x will be generated and tested, even though the precondition only depends on xs. In order to prevent the needless replication of trivially failing tests, it would be better for the application programmer to specify that once valid xs have been found, only then should x be generated. The user can specify this like so:

$$prop\_insertSorted :: [Int] \to Property$$
$$prop\_insertSorted\ xs = sorted\ xs \Longrightarrow \lambda x \to ordered\ (insertSorted\ x\ xs)$$

The *Property* DSL in SmallCheck can also be used to perform existential testing. To test an existential property f, an argument x must be found within the depth limit that makes $f\ x$ true. This now has an interesting effect on the interpretation of the depth limit. If the depth limit is too low then the existential may not be found, and so the only way to make the test pass is to increase the limit. This runs counter to the intuitive notion that increasing the depth limit makes it more likely for a counterexample to be found and thus to fail the test. An extra restriction offered by SmallCheck, that of requiring unique existential witnesses, can further complicate what increasing the depth limit means for the soundness of a test. A property requiring a unique existential x may fail if the depth limit is too low to find x, then pass at a sweet spot where only x is found, but as the depth limit is increased further, fail again if a new example different to x is found. SmallCheck does provide some combinators in the DSL to allow the user to alter the depth limit when interacting with existential properties to help solve these problems.

SmallCheck also has the ability to generate higher order functions if the user specifies a (mechanically derivable) *coseries* function in the *Serial* type class. As SmallCheck focuses on complete explorations of depth limited search spaces, *coseries* will enumerate all functions of $(input, output)$ pairs. As these functions are total and enumerable, if suitable *Show* instances are available for both the inputs and the outputs, then *SmallCheck* can print out what the mapping the function uses is as a counterexample if needed.

EasyCheck

Random and depth first iterative deepening aren't the only exploration strategies that have been considered for enumerating search spaces of terms for testing. The authors of EasyCheck, [CF08], for example, put forward three properties that they believe the ideal test case generation strategy should possess. Their ideal exploration

strategy for a finitely branching, but possibly infinite in depth search space would be complete, advancing and balanced:

- *Complete*: every node in the search tree is eventually visited. For example, breadth first search is complete, but unbounded depth first search may get stuck in an infinite branch, meaning some values will never be generated.
- *Advancing*: each level n of the search tree is (at least partially) visited after p (n) other nodes, where p is a polynomial. The authors argue that this avoids numerous trivial test cases, getting to larger test cases faster. This seems to run counter to the small scope hypothesis (that in general bugs will be exposed by small counterexamples), and means that test cases generated will also not be (necessarily) minimal. Depth first search (on an infinite search tree) is advancing (it explores the first node at depth n in n steps), whereas breadth first search is not (it will enumerate all of the small test cases before larger ones, requiring exponential time to reach a new level of the tree).
- *Balanced*: the order of values tested is independent of the order of child nodes in the search tree. If breadth first search collected all of layer $n+1$'s children and shuffled them before starting layer $n+1$ it would be balanced (the authors note that a normal breadth first search is nearly balanced). Depth first search is not balanced.

The tool the authors present, EasyCheck, performs functional testing in the style of QuickCheck or SmallCheck, for the functional logic programming language Curry. A feature of Curry is that expressions may have non-deterministically many values, for example:

$$bool = False$$
$$bool = True$$

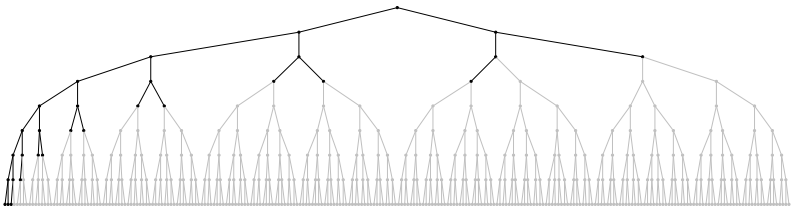
When evaluated, *bool* could be *True* or *False*. A primitive of the Curry system they use allows reification of the possible values a non-deterministic expression might take in the form of a *SearchTree*, vis:

$$searchTree :: a \rightarrow SearchTree\ a$$
$$\textbf{data}\ SearchTree\ a = Value\ a \mid Or\ [SearchTree\ a]$$

Given *SearchTree*s for the arguments of the property to be tested, the authors then present a search strategy, level diagonalisation, and then extend it with random shuffling, in an attempt to find a better search strategy based on the tenets of being complete, advancing and balanced.

All the nodes in a search space can be represented by an infinite list of lists, where each inner list represents all nodes at a particular depth. Given one of these lists of lists, then different search strategies can be thought of as different ways of transforming that list of lists into a single list, where the order of items in the single list gives the order or examining each node. For example, simply concatenating all the inner lists gives rise to breadth first search.

Figure 2.2 Level Diagonalisation Strategy on an infinite binary search space. Lines in black show the edges traversed and nodes visited in the first 40 steps. The graph is cropped at depth 10.



The level diagonalisation approach performs a diagonal interleaving of the inner lists. In Figure 2.2 we show the first 10 levels of a binary search space, and visualise in black the first 40 nodes / edges traversed. The approach is complete and advancing, however it is biased towards the left hand branches of the tree, and is therefore not balanced.

Figure 2.3 Randomised Level Diagonalisation Strategy on an infinite binary search space. Lines in black show the edges traversed and nodes visited in the first 40 steps. The graph is cropped at depth 10.

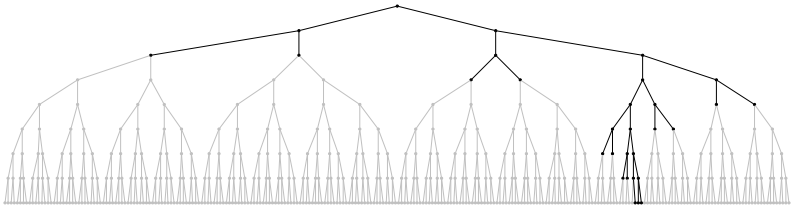
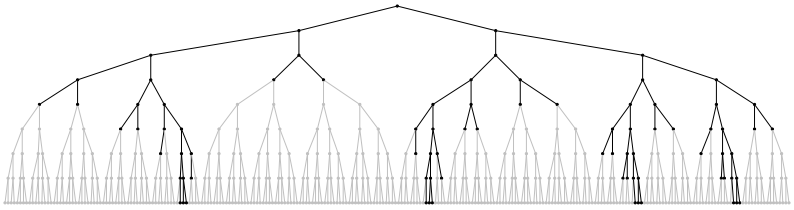


Figure 2.4 Four Randomised Level Diagonalisation Strategies interleaved on an infinte binary search space. Lines in black show the edges traversed and nodes visited in the first 160 steps. The graph is cropped at depth 10.



A refinement to this strategy to try and make it balanced, is to shuffle the children of a node recursively before attempting diagonalisation. Figure 2.3 shows the effect upon the exploration of the search space this shuffling has. While this makes the exploration balanced, it has the unfortunate property that the initially created larger test cases are all similar, as they lie close to each other in the search space. The authors initial solution to this is to interleave several randomised level diagonalisation traversals, as presented in Figure 2.4.

The authors do note, however, that generating many, "sufficiently different large test cases early" is still open future work.

Evaluation Guided Enumeration

Needed Narrowing

In the existing tools for Haskell (QuickCheck and SmallCheck), we have seen that the testing library takes a function to test, and searches for inputs to that function that make the test fail. This has an analogy with logic programming languages, where a search to instantiate metavariables with values is used to make a function definition succeed.

The difference, however, is that the existing tools are generating test data, and then seeing if that makes the test pass, whereas the logic programming languages use some form of narrowing to evaluate the expression, and then only instantiate the metavariables with as little structural value as necessary.

An effective narrowing strategy for functional logic is needed narrowing [AEH00], which is a sound, complete and optimal evaluation strategy for a class of functional logic programming languages (inductively sequential) which approximate the core features of functional languages such as Haskell.

The evaluation strategy that needed narrowing prescribes closely follows the lazy evaluation strategy used by Haskell. However, needed narrowing also describes how to instantiate metavariables with just enough (but no more than is necessary) concrete data to allow evaluation of a term to reach a head normal form. The analogy with test generation is that if all the arguments to a function are variables, it specifies test data for just the parts of those variables that will be evaluated, leaving the rest undefined.

The needed narrowing algorithm processes rewrite rules expressed in a *definitional tree* consisting of branch, rule and exempt nodes. These are analogous to Haskell's case statements, values and error calls respectively. Our presentation of needed narrowing will maintain the Haskell analogy in order to avoid introducing a new syntax, rather than the predicates and rewrite rules in their formalism.

For example consider the following definition of a Haskell-like less than or equals function, *lte*. Traditionally it would be expressed as an ordered list of rewrite rules, or in the declarative equivalence syntax that Haskell makes natural.

$$\textbf{data}\ Nat = Zero \mid Succ\ Nat$$
$$Zero\ `lte`\ x = True$$
$$(Succ\ x)\ `lte`\ Zero = False$$
$$(Succ\ x)\ `lte`\ (Succ\ y) = x\ `lte`\ y$$

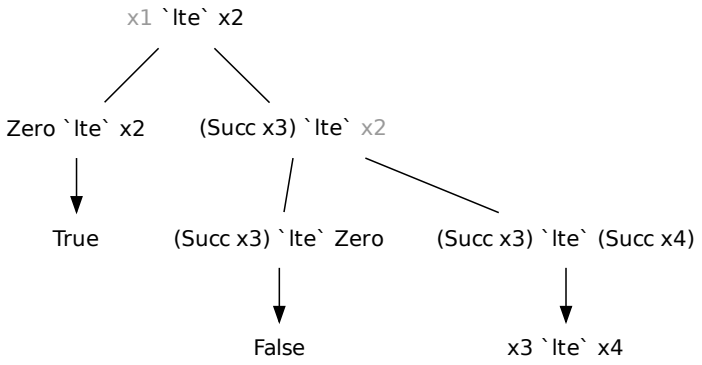
Rewriting *lte* in a definitional tree style makes the pattern matches and precedence between the rewrite rules explicit.

```
x1 `lte` x2 = case x1 of
  Zero     {-Zero `lte` x2 -}      → True
  Succ x3  {-Succ x3 `lte` x2 -}   → case x2 of
```

Zero {-Succ x3 ‘lte‘ Zero -} → *False*

Succ x4 {-Succ x3 ‘lte‘ Succ x4 -} → *x3 ‘lte‘ x4*

Figure 2.5 Graphical representation of the definitional tree for the *lte* function



In the comments next to the pattern matches we have stated explicitly what the top level term must have been for the pattern match to succeed. In Figure 2.5 we give a graphical presentation of the definitional tree, in the style used by the paper directly, which use the full terms at the roots of the branches (case statements) and highlights the variable being discriminated upon. Undecorated edges connect case statements (i.e. connect a definitional tree root to child definitional trees), whereas arrows connect a definitional tree to a final result value.

The core of the needed narrowing strategy is defined by a function λ, which takes an expression rooted at an operation, and the definitional tree for the operation. It outputs a single reduction step (consisting of a substitution to apply to the free metavariables, the part of the expression to be reduced, and the transformation to be applied to that part of the expression). In the case that no reduction can take place, the output transformation is instead ?. There can be cases were several different instantiations may lead to different reduction steps. The algorithm as presented will make a non-deterministic choice between them, however that choice can be seen to give rise to a search space of all the different ways of generating test data for the metavariables that lead to a single reduction.

Following our Haskell analogy, the implementation of λ for the most part follows the rules for lazy evaluation. However, when evaluating a case statement over a metavariable, the strategy has to know which branch to follow in the case statement. At this point, a non-deterministic choice needs to be made where the metavariable is instantiated to a constructor to make a branch followable. In definitional trees, branches must contain cases for all constructors (those that would be incomplete can be followed by an exempt node, which in Haskell would be a call to *error*), so this choice doesn't require knowledge of the definitional tree itself, only the type of the metavariable concerned.

The Haskell runtime system already performs lazy evaluation, only evaluating variables when case statements demand their values. If it were possible to observe when variables were scrutinised in case statements, and

then replaced with one of their constructors, then needed narrowing could be performed in Haskell directly. One potential bonus of being able to do this in Haskell would be to optimise some property tests that use pre-conditions. For example, consider our earlier *prop_insertSorted* example. To make the exposition simpler, we will test lists of inductively defined *Nat* as opposed to magical *Int* values.

```
prop_insertSorted :: [Nat] → Nat → Property
prop_insertSorted xs x
  = sorted xs ==> ordered (insertSorted x xs)
```

Previously in SmallCheck we saw that this definition would repeatedly test *sorted xs* while needlessly varying the value of x. However, if SmallCheck took a needed narrowing approach, a value for x wouldn't need to be instantiated until after a suitable *xs* for *sorted xs* were found. In-fact, given sensible definitions for *ordered* and *insertSorted*, in the case that *xs* is instantiated to [], x should also not need to be instantiated for the test to pass (inserting anything into an empty list should always succeed, and a single element list is always ordered).

Lazy SmallCheck

In [RNL08], the authors discuss the above problem and ideal solution for properties with pre-conditions, and then put forward an approach to make needed narrowing work in Haskell. The authors realised that to be able to observe when a metavariable is reduced can be done simply by replacing the metavariable with an expression that throws a unique exception when evaluated. Most modern Haskell systems feature a way of catching thrown exceptions within an *IO* computation.

During evaluation of an expression, if a metavariable exception is caught, then the expression is rewritten to feature a constructor applied to new metavariables (i.e. new exceptions) in place of the original metavariable, and then re-executed.

For this to work requires the representation of expressions in Lazy SmallCheck to be reified to a uniform representation. As with SmallCheck and QuickCheck, the user has to implement a type class to specify how to create test data of a certain type. However, as with SmallCheck, this type class is usually mechanically derivable. The Lazy SmallCheck type class instances express both how to construct and how to refine test data in a "universal" format.

Figure 2.6 The core datatypes underlying the implementation of Lazy SmallCheck

```
type Series a = Int → Cons a
data Cons a = Type :*: [[Term] → a]
data Type = SumOfProd [[Type]]
data Term = Hole Pos Type | Ctr Int [Term]
type Pos = [Int]
```

In Figure 2.6 we present the core types and datatypes underlying the implementation of Lazy SmallCheck. For all test data of type a to be produced, a function of type *Series a* must be available (typically through a type class method). Within the definition of *Cons a*, the *Type* argument represents the shape of type a. The outer

list has length equal to the number of constructors of the type. For each constructor represented in the outer list, its inner list represents the types of the arguments to that constructor.

In addition to knowing the structure of the type a, a *Cons a* also has to know how to create values of type a, which is what the second component of the $:*:$ represents. There is a list of functions, where each function builds one of the constructors for the type a. The argument $[Term]$ represent the list of arguments to the constructor. These *Term*s either represent yet to be refined metavariables (*Hole*s), or concrete constructors of the argument type, with a list of *Term*s for their children.

Figure 2.7 Two example *Cons* implementations.

```
boolCons :: Cons Bool
boolCons = SumOfProd [[], []] :*: [(λ[] → False), (λ[] → True)]
maybeCons :: Cons a → Cons (Maybe a)
maybeCons (ty :*: mk) = SumOfProd [[], [ty]] :*: [(λ[] → Nothing)
                                                  , (λ[child] → Just case child of
                                                      Hole p _ → error ('\0' : map toEnum p)
                                                      Ctr i ts → ((mk !! i) ts)
                                                      )
                                                  ]
```

In Figure 2.7 we give two example *Cons* values. Note that Lazy SmallCheck provides many helper combinators to make production of these values much more straightforward, but we present here expressions morally equivalent to what would be produced by Lazy SmallCheck to help show the relationship between the sum of products relationships in the types, the functions to create terms, and how "holes" (metavariables) in values get converted into errors.

Referring back to Figure 2.6, the *Hole*s carry with them their position in the ultimate Term tree, encoded as a *Pos*, which is a list of *Int*s. At runtime, Lazy SmallCheck will convert a *Term* to a value and execute it. The result will either come back as a ground *Bool* value (we are testing properties), or an exception encoding the path to a *Hole* term that needs refining. During refinement it is simply a case of walking the path described by the *Int*s and by replacing the *Hole* at that point to create a list of new possible terms from each *Ctr* available.

With this machinery in place, the authors implement implication / preconditions in properties in the natural way. The authors do note, however, that still some care needs to be taken when writing properties as now the evaluation order of conjunctions matter. For example in the expression:

$$p\ x = (f\ x \wedge g\ x) ==> h\ x$$

Assuming both f and g are total functions, then the ordering of f and g matters and will affect the number of test cases generated. If f is a strict function (i.e. it forces x to a normal form), but is permissive (in general it returns *True*) and g does not force much of the structure of x to a normal form, but does return *False* very frequently, then much more test data will be generated than is strictly necessary.

In order to help alleviate this problem, Lazy SmallCheck introduces a new combinator entitled parallel conjunction, $*\&*$. The use of it forms part of the *Property* DSL for Lazy SmallCheck. When evaluating a parallel

conjunction, $a *\&* b$, if the evaluation of a causes a hole to be reached, then Lazy SmallCheck will try to evaluate b, before going back to refine a. If either conjunct evaluates to *False* then the final result will be *False* without having to complete the evaluation of the other conjunct.

Property Directed Generation of First-Order Test Data

Part of the trick underlying parallel conjunction is that if some evaluation could happen in parallel, then in a sequential system it could be beneficial to interleave instantiating some branches with executing others, in the hopes that a different branch may reach a result without requiring instantiations.

In [Lin07], the author investigates this idea by modelling a small functional programming language, similar to the core of Haskell, but adding metavariables (analogous to Lazy SmallCheck's *Hole*), and a parallel evaluation construct. In their system, the parallel conjunction operator can be expressed as:

$$
\begin{aligned}
x *\&* y = \textbf{select}\ \{&\textbf{case}\ x\ \textbf{of}\ \{\mathit{True} \to y; \mathit{False} \to \mathit{False}\};\\
&\textbf{case}\ y\ \textbf{of}\ \{\mathit{True} \to x; \mathit{False} \to \mathit{False}\}\\
\}&
\end{aligned}
$$

Here, the **select case** expression means that the result may be chosen non-deterministically from the evaluation of either **case** branch. During evaluation, there may now be several "blocking" metavariables, where refining any of them could enable progress down one branch of a **select case** expression. If evaluation reaches a normal form down any **select case** branch, then that normal form is used, otherwise the set of blocking metavariables is returned. This set of variables gives rise to a search space for deciding the order to refine and retry them in.

This technique can be applied to property testing. If the result is not the desired one, then some backtracking has to occur to find a different metavariable to instantiate, or a different value to instantiate the metavariable with.

The author calls the technique *lazy instantiation*, and presents a prototype for a Haskell-like language. In his evaluation he shows that lazy instantiation outperforms blind enumeration. Parallel evaluation may not always perform better than the non parallel program with well chosen ordering of conjuncts, but it is not worse by much, and it gains massively if the conjunct ordering is poorly chosen.

Summary

In these techniques for enumerating and exploring search spaces of property testing we have seen that there are several choices for how to enumerate a search space. Existing literature has performed random, exhaustive, diagonalised and randomised diagonalised searches. In addition, motivated by the problems caused by preconditions in some property tests, research has looked at using meta variables and needed narrowing like techniques in order to make testing focus on test cases that actually matter. However, the ordering of conjuncts in preconditions can adversely affect these techniques, and so further work looking at parallel evaluation constructs and reduction strategies has been investigated which mitigates some of these adverse effects.

2.2.2 Automated discovery of errors

One of our objectives is the unaided, automated discovery of error conditions in software. This has been investigated through several existing techniques, we consider two here: symbolic execution and static analysis.

Symbolic Execution

Symbolic execution [Kin76] is the interpretation or execution of a real program, however instead of specifying all inputs, some are left as symbolic variables. In an imperative setting, later assignments of variables to values that involve symbols will cause the variables to be assigned expressions. Branching points in the program require the state of the program to be cloned, one clone for each branch that is possible to take. Following any particular branch will build up a path condition in that branch, which expresses the extra constraints on the symbolic values that must have been true for that branch to be selected. If execution reaches an erroneous state or undesired location, the path conditions can be solved (usually by an SMT solver) to give example values for the symbolic inputs. These example values correspond to a test case that will cause program execution to reach the undesired location.

Figure 2.8 A simple C-like increment function

```
1  void inc(int x) {
2    int y = x + 1;
3    if ( y <= x ) {
4      throw "Impossible"
5    } else {
6      return y
7    }
8  }
```

For example, consider Figure 2.8. In line 1, `x` is to be treated as a symbolic variable. In line 2, `y` is assigned a value based on an expression featuring a symbolic variable, so it gets a symbolic value, namely the expression `x + 1`. In line 3 there are two choices possible choices:

- In choosing line 4, the path condition is `y <= x`, where `y = x + 1`. Line 4 also represents a crash, so if a solution to `x` such that `x + 1 <= x` (by substituting the value of `x + 1` for `y`) can be found then we have an example bug.

  Good SMT solvers will understand that programming language `int`'s are signed, 32 bit, 2's complement numbers. So if `x` has the maximum possible positive value (e.g. 2147483647), then adding 1 will make it negative (-2147483648), and using this value as an argument to inc will cause the function to crash.

- In choosing line 6, the path condition is `!(y <= x)` which can be simplified to ($y > x$). However the return statement always succeeds, so there is no bug to be found through this path.

This is of course a simple example. Programs can contain loops and may not terminate (even symbolically), so there has been research in choosing the most effective or promising branches to follow. Full symbolic checkers

also have to take into account more complicated constraints such as arbitrary pointers into the heap, and symbolic inputs from e.g. reading from a file.

Pex

The symbolic execution tool Pex [TDH08] operates upon .NET bytecode, and can verify several different language paradigms within the .NET ecosystem (for example, C#, VB.NET and F#). The search strategy for choosing branches to symbolically execute is based upon arc coverage. Pex remembers which arcs in the control flow graph it has visited previously, and prioritises those it hasn't when exploring new paths. To be able to evaluate and solve path conditions and constraints, Pex makes use of the Z3 [DMB08] constraint solver.

KLEE

Another successful symbolic execution tool is KLEE [CDE08]. This interprets LLVM bytecodes, and has been used to find real bugs in widely used suites of programs, such as GNU CoreUtils. Unlike Pex, KLEE uses random selection when choosing branches in the program to explore. KLEE also features many optimisations to enable it to explore and keep in memory many branches at the same time. It has optimised compact space representations for path constraints, and makes use of copy on write data structures when cloning branches at decision points.

What makes KLEE particularly interesting is its simplified symbolic models for handling system calls. For example, a symbolic file system consists of a single directory with a user configurable number of symbolic files in it. System calls on non-symbolic files proceed as normal.

Reach

Symbolic execution has also been explored in the context of lazy functional languages like Haskell. In [NR07] the authors take a first order core functional language (which Haskell can be compiled to) and use symbolic execution to see if interesting program locations can be reached (and what inputs cause the location to be reached). A program location is interesting if it has been annotated with a *target* keyword by the user, or is the cause of an exception or black hole[2].

Since Reach can present example inputs that reach arbitrary program locations, it is straightforward to use Reach to perform property testing. A simple combinator, *refute* is presented that can be used to wrap the result of a Haskell *prop_* property function.

$$refute\ True = True$$
$$refute\ False = \underline{target}\ False$$

A test to see if (for example) an *insert* function maintains an ordering is then expressed as:

$$prop\_insert :: Int \to [Int] \to Bool$$

[2] A black hole is a detectable form of infinite loop where evaluation of a value depends on evaluation of itself, for example **let** $x = x$ **in** x.

$main :: Int \to [Int] \to Bool$
$main\ x\ xs = refute\ (prop\_insert\ x\ xs)$

The symbolic executor uses a lazy reduction semantics to respect the laziness of (the transformed) Haskell. Inputs to the program (for example the x and xs argument in the *main* program above) are represented as unbounded variables. When the executor reaches a case expression over an unbounded variable then conceptually execution splits, with each pattern in the case expression being followed independently. Following a pattern will then refine the unbounded variable to a more concrete form.

In order to prevent the symbolic executor getting stuck on a single path in e.g. a highly recursive or even infinite loop, there is a bound on how far it can explore a single path. In addition, the splitting of execution at case statements is implemented through backtracking once a bound has expired.

The authors explore two different bounds: the maximum tree depth of input data created through refinement; and the maximum recursion depth of function calls. The input space of the first strategy is very intuitive; it is easy for a user of Reach to understand what input values will and won't have been created within a certain depth. In contrast the second strategy does not correspond to an intuitive space of input values, but it does guarantee termination. The authors have examples where each strategy outperforms the other in finding target locations and observe that work is necessary to establish an intuition for when to use which type of bound.

However there are some limitations in the implementation of Reach. One interesting restriction is that Reach can not synthesise functions as top level inputs, although the authors state that this is further work they hope to explore. Another restriction is that Reach cannot synthesise primitive built in types. However for the purposes of running experiments, they used a Peano representation of integers.

The authors also put forward some ideas for optimisations within the symbolic executor. For example, some expressions will never reduce to a target expression and could be ignored by the executor. However Reach could then return results that crash in the ignored parts before the target was reached. As an alternative, such expressions could be delayed until after a target expression had been reached to ensure it is known if they crash or not.

Another approach is to use a backwards analysis that moves the target expressions up through the program to the root, gathering equational constraints during the lifting. Symbolic evaluation can be used to solve the constraints, resulting in only the parts of the program that directly affect reaching a target expression being considered.

Static Checking

Symbolic execution cannot always guarantee that programs are devoid of reachable error conditions; loops and recursive calls may require large amounts of time to be executed, and, by appealing to the halting problem, it is impossible to know in general if all possible followable branches in the program have been followed.

Another approach, in static analysis, aims to prove that error conditions are not reachable. We consider two approaches here, focused on proving certain properties of Haskell.

Not All Patterns, But Enough

While programming in Haskell, a programmer may deliberately create partial functions, i.e. a function that does not have an implementation for all possible values its arguments may take. The most common example of this is the *head* function which returns the first element of a list, and throws an exception if the list is empty. In [MR08], the authors present a static technique (implemented in a tool called Catch) that proves that partial pattern matches in a program are safe, i.e. there is no dynamic path that can lead to an unimplemented case being reached.

The technique operates upon a first order language, however the tool can work upon full Haskell '98 programs by first transforming them into a first order language; In order to reason about non-algebraic data types (such as *Int*s and *Char*s) they are converted into algebraic data types representing an abstracted value. For example:

data $Int = Neg \mid Zero \mid One \mid Pos$
data $Char = Char$

Additionally, in order to reason about functions that return *IO* values, it is assumed that any value of the returned type could be returned.

The technique used by Catch is to calculate preconditions on the arguments on functions, and, assuming those preconditions hold, some guarantees on the return values of the functions. The preconditions are expressed by constraints on the shape that the arguments may take. In the paper, the authors put forwards three different types of constraint, and explain their positive and negative attributes.

For example, basic constraints say a value can take any shape, or must be rooted at a particular constructor, with a list of recursive constraints for the arguments to that constructor:

data $Constraint = Any \mid Con\ CtorName\ [Constraint]$

All preconditions to functions are initially assumed to be true, apart from the one for the *error* function (which is false and used to represent partial pattern matches). The algorithm used by Catch iterates building preconditions by conjoining old preconditions with newly constructed ones. While this means constraints get more restrictive, the basic constraints presented have an infinite space (data types can be infinite in space). This therefore means the algorithm may not terminate.

In order to guarantee termination, two other forms of constraint are presented:

- Regular expression constraints represent a path through a value and a set of constructors that the value at the end of the path must be in the form of. The regular expression language is restricted so that for a particular value, there are a finite number of possible regular expression constraints that could be generated for it.

- Multipattern constraints consist of two constraints, $p_1 \star p_2$. A value satisfies this multipattern constraint if the root satisfies p_1, and all values recursively reachable from the root satisfy p_2.

Catch has been able to analyse real programs, and the evaluation section of the paper makes clear the strengths and weaknesses of the static analysis approach. For example, several programs within the nofib benchmark suite needed to have some small source modifications before Catch could perform meaningful testing on them. However the approach is useful, and found real bugs within the Haskell applications HsColour and XMonad.

Extended Static Checking for Haskell

An alternative approach to property testing checking is to allow the user to state explicit pre and post conditions on their functions, and then use static checking to ensure the pre and post conditions hold. In [Xu06] the authors outline such an approach for Haskell. The technique puts forwards transformation rules to turn a function with pre and post conditions into a functional programming expression. The authors then explore using supercompilation (a combination of static inlining, simplification and other aggressive compiler optimisation techniques) to simplify the resulting expression. The methodology arranges it so that the originally transformed expression contains *BAD* terms that supercompilation should only be able to remove if they are not reachable (i.e. they are guarded by a pre-condition that ensures they are not reachable). If the expression can be simplified so that no *BAD* terms are reachable, then, assuming the preconditions to the function hold, the function is error free and cannot crash (modulo non-termination).

In order to make their technique scalable, functions are analysed in isolation, and the pre and post conditions of referenced functions are assumed to hold when used. This requires that pre and post conditions of all referenced functions are precise enough to specify the correct behaviours needed for the function being analysed.

One other feature of this technique is counterexample guided unrolling, which means that the supercompilation will unroll some functions towards removing *BAD* expressions. If ever a state is reached were there are no function calls between the root of the function and a *BAD*, then a program slicing technique can be used to generate a concrete example argument that leads to the error or post-condition violation.

Discussion

Symbolic evaluation is related to some of the functional logic techniques shown earlier. The splitting of the search space for different instantiations of metavariables in a functional logic system is related to the branching and path constraints built up during symbolic execution. Some of the techniques adopted by the symbolic execution community, particularly the use of SMT solvers for working with more primitives types could be applied in the functional logic world.

Recent advances in the Haskell ecosystem means that symbolic evaluation could also be explored more directly for Haskell. The KLEE tool, for example, performs symbolic execution upon LLVM bytecodes. Recently the Glasgow Haskell Compiler has added a new back end for outputting LLVM bytecode, and thus compiling via LLVM. It could be an interesting experiment try symbolic evaluation on Haskell in that manner.

One interesting aspect drawn out of these techniques is the approach to dealing with *IO* operations. In the static analysis world, Catch assumes that an *IO* value could take on any value. KLEE however allows the user to decide how to abstract some IO operations. However, other *IO* operations are allowed to execute directly. This obviously requires care during testing to ensure that adverse operations do not destroy the very system being run.

2.2.3 Debugging Tools

Code Coverage

One measure of the effectiveness of a testing technique is to look at the coverage of the code being tested that it achieves. In the context of Haskell this can mean looking at the number of source expressions and subexpressions in the program that actually get executed at runtime. The Glasgow Haskell Compiler, GHC, has built into it a code coverage mechanism entitled Haskell Program Coverage (HPC) [GR07].

The GHC integration of HPC works by rewriting GHC's intermediate *Core* language so that every expression and subexpression entered executes some code to update an array of counts. Each element of the array represents one subexpression in the original program. Upon the completion of execution of a rewritten program, a "tix" file is output which contains a serialisation of these arrays. During the rewriting and compilation phase, "mix" files are generated that describe the link between the arrays and the original source code. HPC has a suite of tools for calculating the union, difference, etc. of multiple tix files, and pretty printing them to output numbers of expressions, conditional branches, top level functions and various other statistics.

One other interesting feature of HPC is the ability to access the tix arrays at runtime. This allows a program to reset or reflect upon the code coverage currently obtained so far during the run of the program.

HPC has been designed to work in large systems. It is possible for Haskell modules that have not been rewritten to be compiled with modules that have, and for the resulting system to work seamlessly. To achieve this, HPC maintains one tix array per module. Modules that have been rewritten write their coverage information into that modules tix array. Modules that haven't been rewritten won't have a tix array, and do not have to worry.

While HPC is a mature and well integrated tool, there are also other possible approaches to code coverage that could be taken. In the context of test generation, the authors of [FK07] investigate two alternative notions of code coverage from simple expression and subexpression coverage. They note that lazy declarative languages frequently have complicated control flow, and so control flow graphs are hard to represent. They then argue that imperative notions of code coverage do not easily map to declarative languages. In the context of the functional logic programming language Curry, the authors present a rewriting system to transform Curry programs into ones that can record their own coverage. They present two coverage criteria, and they evaluate using them as criteria for when to stop testing:

- Global Branch Coverage: all alternatives in case expressions inside function definitions are labelled differently. Testing stops when all labelled branches are reached in functions directly or indirectly called from

the function being tested.

- Function Coverage: In addition to achieving Global Branch Coverage, all recursive calls to the function being tested must achieve full coverage of all branches inside the recursive call.

The authors evaluate their strategies, and show that using Function Coverage as a stopping criteria will expose bugs that Global Branch Coverage would otherwise miss.

Debugging Haskell Programs

There is active research into testing and debugging Haskell programs, and there are several other tools available to help the Haskell programmer in doing this. In this work we have focused our efforts on helping a Haskell programmer debug their programs by finding techniques to automatically produce small expressions that expose bugs. The programmer then has to reason through what their application does when given the expression as an input, hopefully finding and fixing the bug in the process.

When debugging an application, a programmer tries to find some way to visualise the internal states of their program at runtime, to try and find out where their model of what the program should be doing, and what it is actually doing differs. By narrowing down on the part of the program that causes the discrepancy between model and actual states, the programmer should hopefully be able to identify and fix the bug.

Very simple forms of this debugging are available in Haskell through *Debug.Trace* from the base libraries, that allows for an arbitrary string message to be printed to standard error when an expression is evaluated. This is analogous to debugging through `printf` statements in imperative languages like C. In Haskell this kind of debugging can easily be misleading due to the subtle effects of lazy evaluation. The very act of printing some data may force it to a normal form earlier than normal, possibly hiding the bug, possibly causing it to manifest in different ways. There is also the issue that the evaluation order using lazy evaluation means that the order of print statements may not be one that users intuitively expect.

An alternative approach to visualising data to aid in debugging is offered through the Haskell Object Observational Debugger (Hood) [Gil01]. This is a library that allows a user to tag certain computations to enable observation of the concrete values they evaluate to at runtime. This tagging is enabled by wrapping the computation in a function from the library, $observe :: (Observable\ a) \Rightarrow String \rightarrow a \rightarrow a$. The *String* is used to describe the value when it is output.

The library is implemented in such a way as not to alter the strictness properties of the data. If only the first element of an observed list value is evaluated, then only that element will be visualised, with a sentinel "_" used in place of the unevaluated tail.

Hood can also visualise functions. It will observe both the evaluated parts of the inputs to the function, and the evaluated results returned by it. The visualisation of functions is by presenting a list of (input, output) pairs. However, because of nested lexical scopes, the output of a function may also depend on values that are

in scope, but not an explicit argument to that function. Hood has no mechanism to detect and display these, so there may be cases when a visualised function has multiple (input, output) pairs with the same input but different output.

The downside is that Hood requires some changes to the programmer's application to be able to use it, custom data types to be observed need to be made an instance of its type class *Observable*, and the *observe* function needs to be imported and placed in the programmer's source code. Additionally the programmer has to alter the entry point to their program to enable Hood to record observation data and print it at the end of execution. Hood also suffers from a problem that the user of *Debug.Trace* also faces, knowing which values are the correct ones to focus on in the first place.

An alternative to editing source code for printing values is to use an actual debugger application. GHC ships with one, [MIPG07], integrated into its interactive environment, GHCi. GHCi can load both interpreted and compiled code, however only interpreted code can be fully understood by the debugger. It has many features, such as the ability to set breakpoints at arbitrary points, single step reductions, safe visualisation (and optional forcing) of available bindings in scope, and to record a trace (and stepping through the history) of the execution of an expression. These traces represent the evaluation order followed by lazy evaluation, and so, again, may not always be straightforward for a user to understand. However the ability to set breakpoints and inspect the local environment can be a massive gain for a programmer trying to understand what their code is doing.

The problem of understanding program execution traces in Haskell has also been looked into, and several tools addressing the problem have been produced for example Freja [Nil98] and Hat [WCBR01].

Freja does not support full Haskell 98, however it does suggest an interesting debugging technique. It runs the application, recording a trace of all reductions that took place during evaluation. It presents the user with an expression and the value it reduces to, and then asks the user if this reduction was correct. Initially it will start at the top level of the program or function of interest. As the user specifies if a reduction was correct or incorrect, Freja will then search for the reduction sequences that caused the parent behaviour, and interactively ask the user if their behaviour was correct or not. Continuing this search, Freja will search to find the faulty reduction sequence that comes from the program definition and state this to the user.

Hat also runs the application and records a trace of all reductions that took place. However instead of starting at the top level, it presents the user with the result of running the program (which could be a value or an exception). The user can then browse the parent redexes that caused the final result, working backwards from the final result, in an effort to try and locate the bug in their code.

A more thorough description and comparison of Hood, Freja and Hat, which also features feedback based on experience of using all three tools is available in [CRW01].

Algorithmic Debugging

Freja and Hat both support a general line of research known as Algorithmic Debugging, a detailed summary of which can be found in [Sil07]. Algorithmic debugging is concerned with taking a program's execution trace

that features a fault, and locating the cause of the fault. In order to do this an oracle (usually the programmer answering yes or no questions) is asked whether the results of sub-computations from within the trace are correct, and the algorithmic debugging strategy then uses these answers to determine the next sub computation to ask about, or to present the source of the bug.

For functional programming languages, an execution trace is a tree representing the reductions that took place during execution. Each node of the tree corresponds to a function applied to arguments, along with the result of that application. Child nodes correspond to each function application in the definition of the function application in the parent node.

If a function applied to arguments gives the wrong result, then the bug could be in the definition of the function when applied to those arguments. Alternatively, the bug could be in the definition of one of the child functions that are called in the definition of the parent function. In this case, the bug will also be present in a child of the current node of the execution trace.

An interactive search for bugs proceeds by considering whether the nodes of the execution trace are known to be bug free, or *suspicious*, i.e. it is unknown whether they contain a bug. Initially, every node in the execution trace can be considered suspicious. The strategy then presents nodes to the user, and asks the user if the function application and result in the node are correct. The user's response will then remove some nodes from suspicion. When a single suspicious node is left, the definition of the function that corresponds to the application and result in the node can be identified as buggy.

When the user classifies a node as being correct or not, the suspicion in the execution trace is updated as follows:

- The node is not correct: the bug is either in the current node, or one of its descendants. All other nodes can be removed from suspicion.
- The node is correct: then this node and all of its descendants can be removed from suspicion.

The research literature features many different strategies for exploring the execution trace in order to find bugs. To motivate the main issues that should be factored in when exploring, consider two of the simplest strategies: post-order traversal, and pre-order traversal.

With a post-order traversal, the first node a user identifies as buggy will be the source of the bug. However in the worst case this strategy will ask the user to classify every single node in the execution trace. There is also another, more human issue with this traversal scheme. Since the nodes will be visited in a bottom up order, it will require the user to reorient themselves with what the program is (meant to be) doing at almost every question; there will be no continuity between questions and the user would find it hard to anticipate what questions could come next as they would be going backwards through the control flow (and then occasionally jumping back down very deep to the bottom).

A pre-order traversal however can perform much better. This traversal strategy allows for pruning of the execution trace. Identifying a node as correct means the current subtree can be entirely pruned, and identifying

a node as incorrect means the traversal can move down a layer in the search. In addition, the downward movements may make it more natural for a programmer to think about the nodes they are being presented, although movements between siblings can still be complicated to think about.

The naive pre-order traversal can still be improved, for example by looking at the size of the subtrees when moving down, and ordering the children according to size. However, because of the human element (requiring a user to classify nodes as buggy or not), it is important to trade off a predictable exploration, with one that minimizes the number of nodes looked at.

The order in which the children of a suspicious node are explored can have a further effect. In situations where there are multiple bugs contained in the execution trace, the order in which the children of a suspicious node are explored will affect which bug is found. If a user determines that the bug they find following a particular strategy is not the one they are interested in, then it is useful if they can restart and request a different ordering for the exploration of child nodes as they move down the tree.

Algorithmic debugging is a powerful tool, however it does require a user to act as an oracle. On the other hand, it can be seen as a way to force the user through a structured debugging process, and as such, provide mechanisation to an otherwise manual process. The requirement on gathering an execution trace may make the approach difficult to use for large programs, however when combined with a technology that finds small test cases for bugs, it could be very useful.

2.3 Context

We now draw together the related work above, and place the contributions of this thesis in context with existing research. Table 2.1 summarises the key different techniques, their examples and salient features.

The first component of Table 2.1 highlights different libraries written to enable *Property Testing*. The Haskell based implementations (Quick, Small and Lazy Small Check) do require two things from the user of the testing library; the properties to be tested need to be explicitly written, and some declarative description of how test data is made needs to be specified. It is not always the case that they way of generating test data needs to be specified, for example the EasyCheck tool is implemented in the Curry programming language, which the required form of reflection in a first class way), and for the most part Small and Lazy SmallCheck's descriptions could be mechanically derived.

In IRULAN we decided to focus on automating as much of the test data discovery as possible, but do allow the user to specify extra test data. However there are advantages in allowing the user to specify how test data is to be used. For SmallCheck and Lazy SmallCheck, power users can specify precisely how the depth cost of testing changes across different constructions. For QuickCheck, it is essential that good specifications of the distributions to draw test data from are given, otherwise testing may be probabilistically confined to a small region of argument space. In the case of Quick Check this requirement for good specifications is a disadvantage for the novice programmer, as they may otherwise gain a false sense of security in their tests.

Table 2.1 Summary of different testing techniques and their examples

Property Testing

Tool	Exploration Strategy	Advantage	Disadvantage
QuickCheck	Random	Produces varied, larger test cases	Requires good generators for data
SmallCheck	Depth Limited Enumeration	Enumerates all inputs within a bound	Creates many test cases where test data isn't looked at
EasyCheck	Randomised Level Diagonalisation	Creates small and large test cases	Larger test cases are similar
Lazy SmallCheck	Needed Narrowing	Prunes data that is never evaluated	No support for synthesising higher order functions

Crash Testing

Tool	Strategy	Advantage	Disadvantage
IRULAN	Dynamic Test Case Generation	Automatic, Real code execution	Reflective overheads
Pex	Symbolic Execution (.NET bytecode)	Arc coverage based exploration	-
KLEE	Symbolic Execution (LLVM bytecode)	Symbolic model for system calls	-
Reach	Symbolic Execution (Functional core)	User configurable targets (not just errors or property violations)	No support for synthesising higher order functions
Catch	Static Analysis	Guarantees of freedom from pattern match failure	Requires source code changes.
ESC/Haskell	Static Analysis (Contracts)	Flexible and powerful pre and post condition expression and checking	No available implementation

Debugging Tools

Tool	Strategy	Advantage	Disadvantage
HPC	Code coverage tool	Low overhead	Can only identify executed and unexecuted code
Freja	Tracer and interactive debugger	Guide programmer through what happened	Requires trace of execution
Hat	Tracer and trace visualiser	Allow exploration of everything that happened	Requires trace of execution
Hood	Debugging Library	Low impact, precise information	User needs to know precisely what to investigate

For Haskell specifically, requiring the user to explicitly state how to produce test data also confers a performance benefit, as the high costs of reflection and dynamic linking (which are incurred by IRULAN) do not need to be paid.

However the key difference between these tools, as highlighted in the table, is the way they explore the space of possible test data. Random testing (QuickCheck), as noted above, requires programmer input to specify the underlying distributions of test data. However when used well it can explore many larger, more varied tests. The level diagonalised approach of EasyCheck attempts to achieve this property in a more structured way, however it suffers the problem of the larger tests being similar to each other. The authors of SmallCheck and Lazy SmallCheck argue that when counter examples to a property are found, it is almost always the case that a small counter example will highlight the problem. This argument is drawn from the small scope hypothesis that underpins the theory behind model checking tools. SmallCheck simply enumerates all inputs up to a finite depth bound (which can be iteratively deepened), however this approach highlights some potential optimisations across implications in properties which are taken advantage of by Lazy SmallCheck, using a needed narrowing approach.

For IRULAN we felt that exploring Haskell's laziness in the context of general testing was interesting, and since the needed narrowing like approach of Lazy SmallCheck could work autonomously, we decided to adopt that as our base approach to generating test data.

The second component of Table 2.1 highlights the different automated crash testing tools discussed, with the strategy each tool uses to find errors. The tools have been sorted according to how static or dynamic their approaches are.

IRULAN does not statically analyse programs, but instead builds test cases dynamically and executes them, looking to see if errors are thrown at runtime. This has the advantage of being fast (the testing is running native code), whereas the symbolic execution based tools (Pex, KLEE and Reach) have to embed interpreters for the respective bytecodes of the languages they test. However the symbolic execution tools do have the advantage that they can abstract away from concrete testing data and avoid re-execution of identical code paths with different data. However if the program does not allow for abstraction of the data, or is (for example) heavily numeric then there can be higher costs incurred as the original program essentially becomes interpreted, as opposed to executed.

Employing further abstraction gives rise to static analysis tools. The trade off here is memory and time, versus guarantees. In the examples presented, if the static analysis tool tells you the program is bug free, then it is guaranteed to be so. Dynamic test generation (such as used by IRULAN) and symbolic execution can only provide partial guarantees about correctness with respect to the inputs and bounds that have been specified. However static analysis tools can not always provide a binary yes there is a bug, or no there isn't; there will necessarily always be a range of programs they cannot successfully analyse.

We have also discussed debugging tools. While IRULAN and the other technologies mentioned so far aim to locate bugs and provide test cases to trigger them, it is also important to think about how the user then

identifies the cause of, and hopefully the fix for the bug. Many of the tools listed so far will produce small if not minimal test cases, which should narrow the potential source code at fault, however other techniques can narrow this further. The third part of Table 2.1 lists some of these tools for Haskell. The four tools cover four different ideas, HPC allows the user to identify exactly which subexpressions were and were not executed during a particular run. Freja and Hat both record an execution trace, and then allow the user to either interactively explore what happened, or use an algorithmic debugging strategy to try and identify the bug. Finally Hood is a lightweight debugging library that allows the user to annotate their code to see information about only some function calls. The trade off here depends on how much the user knows a priori before debugging. HPC will quickly allow the user to discard some parts of the source code from suspicion. Freja will hopefully interactively guide the user towards finding the bug, but will require them to consider many reductions. Hat, as a visualiser, will allow a user who has a rough idea where their bug is to focus on that area and see exactly what happened. The size of the execution traces considered by Freja and Hat can be reduced by specifying that some modules are trusted and that reductions of their functions need not be traced. Hood then requires the most knowledge, but will then give very focused output to the user.

Chapter 3

Overview of IRULAN

IRULAN is designed to aid in the development of Haskell libraries. There are three main ways to use IRULAN, incrementally during development to see if there are ways to crash exposed functions; before and after refactoring or optimising a library to see if its behaviour has changed; and to perform property testing. In general, IRULAN is invoked with an option to set the mode of behaviour, the name of a module (source or compiled) to test, and optionally some configuration flags that alter how IRULAN finds and creates test expressions.

In Section 3.1 we develop example library functions and demonstrate using IRULAN to find inputs that cause crashes. IRULAN can also identify changes in behaviour between different implementations of the same API, and we build up an example of this use in Section 3.2. The third major application of IRULAN is to perform property testing, which is discussed in Section 3.3. IRULAN was designed to be an experimental platform, and in Section 3.4 and Section 3.5 we discuss some small advanced features of IRULAN and some potential future uses, before concluding in Section 3.6.

3.1 General *error* Finding

IRULAN is able to use its automatic expression generation technique to find arbitrary *error* conditions in Haskell functions. In this section we demonstrate this through a simple example.

Consider the beginnings of a sorted binary tree implementation, as shown in Figure 3.1. The *IntTree* data type has two constructors: *Leaf* and *Branch*. *Leaf* takes no arguments, representing the empty leaf nodes of the tree, while *Branch* represents splits in the tree, with two *IntTree*s for its left and right children, and an *Int* for the value of the new root node.

The *insert* function is used to add an *Int* into the tree, building a *Branch* for the value when inserting into a *Leaf*, and navigating left or right down the tree whenever inserting into a *Branch*, in order to keep the values in the tree in order.

Figure 3.1 Simple Haskell *IntTreeExample* module.

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

Figure 3.2 Part of the output from running IRULAN on the *IntTreeExample* with *Int*s 0 and 1, with case statements enabled, in iterative deepening mode (`-a`) for 1 second (output cut and neatened).

```
$irulan --ints='[0,1]' --enable-case-statements -a --maximumRuntime=1 source IntTreeExample
  ...
  insert 1 (Branch ? 1 ?1) ==> !
    IntTreeExample.hs:(8,0)-(11,41): Non-exhaustive patterns in function insert

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

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

  case case insert 0 (Branch (Branch (Branch ? 0 ?1) 1 ?2) 1 ?3) of
       Branch x _ _ -> x of
         Branch x _ _ -> x ==> !
           IntTreeExample.hs:(8,0)-(11,41): Non-exhaustive patterns in function insert
  ...
```

Unfortunately we have forgotten to implement the case when the value we are inserting, n, is already in the tree. Luckily, the Haskell compiler will implement that case for us by throwing a *Non − exhaustive patterns* exception that would normally 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 3.1 includes the implementation of the *insert* function, IRULAN is in fact a *lightweight black-box tool* that does not look at 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* instances by using *IntTree*'s two constructors, *Leaf* and *Branch*, together with two integer constants, 0 and 1, which are set explicitly on the IRULAN command line, as shown in the example trace in Figure 3.2.

In the trace, we have set IRULAN to run for a second, using an iterative deepening scheme (so it tries progressively larger inputs, but restarts from the beginning increasing the depth after a previous depth has been explored). We have picked out four example invocations that it finds to cause the `Non-exhaustive patterns` errors.

IRULAN's output usually consists of *expression* ==> *result* lines, where when *expression* is executed, *result*

Figure 3.3 Part of the output from running IRULAN in trace mode on the *IntTreeExample* with *Int*s 0 and 1, with case statements enabled, in depth bounded mode to depth 15 (`-d --depth=15`) (output cut and neatened).

```
$ irulan --ints='[0,1]' --enable-case-statements --trace -d --depth=15 source IntTreeExample
IntTreeExample:
Results:
 1 insert ==> .
 2 insert ? ==> .
 3 insert ? ?1 ==> ?1
 4   insert ? (Branch ?1 ?2 ?3) ==> ?
 5     insert 1 (Branch ? ?1 ?2) ==> ?1
 6       insert 1 (Branch ? 0 ?1) ==> .
 7       insert 1 (Branch ? 1 ?1) ==> !
 8         IntTreeExample.hs:(8,0)-(11,41): Non-exhaustive patterns in function insert
 9       ...
10     insert 0 (Branch ? ?1 ?2) ==> ?1
11       insert 0 (Branch ? 0 ?1) ==> !
12         IntTreeExample.hs:(8,0)-(11,41): Non-exhaustive patterns in function insert
13       insert 0 (Branch ? 1 ?1) ==> .
14       ...
15   insert ? Leaf ==> .
16   case insert ? Leaf of Branch x _ _ -> x ==> .
17   case insert ? Leaf of Branch _ _ x -> x ==> .
18   case insert ? Leaf of Branch _ x _ -> x ==> ?
19     case insert 0 Leaf of Branch _ x _ -> x ==> .
20     case insert 1 Leaf of Branch _ x _ -> x ==> .
       ...
```

occurs. By default IRULAN will only present expressions that cause errors to be thrown, indicated by the leading '!' in the four shown results.

IRULAN's expressions are close to normal Haskell expressions. However IRULAN uses an incremental approach to testing, using a form of needed narrowing [AEH00] or lazy instantiation [Lin07]. Undefined arguments are passed to functions and only refined to values if they are needed. These undefined arguments can sometimes appear in expressions, and are represented as '?' arguments. These can safely be replaced by $\perp$ by the Haskell programmer. So, for example, the expression *insert* 1 (*Branch* $\perp$ 1 $\perp$) entered in an interactive Haskell environment would yield the first error in the trace.

By enabling case statements (`--enable-case-statements`) IRULAN will also generate selector case expressions over result values in order to peek inside them. For example, the third test case shown uses a case selector to reach inside the returned *Branch* constructor to force the evaluation of the unimplemented pattern.

It is important to note that as iterative deepening has been used, IRULAN will often output multiple identical test cases as it rediscovers them as the depth increases. However it is easy to post-process the results with unix tools such as sort, uniq, awk and grep to remove duplicates. The indenting and linebreaks in the traces presented have been manually inserted to make reading the examples easier, but are normally not present to make interaction with these tools easier.

In order to gain some intuition into how IRULAN is finding these test cases, we can ask IRULAN to present a full trace of all expressions it executes, shown in Figure 3.3.

As we have mentioned, in its output, IRULAN presents the expressions (on the left hand side of the arrow) it has run followed by a result (on the right hand side of the arrow); there are three main forms of possible result shown in a full trace; which are:

- **Success:** `expression ==> .` This means that `expression` successfully ran to a *W*eak Head Normal Form (WHNF) [Pey87].

  An expression runs to WHNF if it cannot be simplified further without being taken apart by pattern matching, or applied to arguments. Values in WHNF are therefore either constructors (possibly applied to some arguments) or functions expecting explicit arguments.

  For example, line 15 contains the test case `insert ? Leaf ==> .` because inserting any number into a *Leaf* tree always succeeds by returning a tree of the form (*Branch Leaf n Leaf*). The number inserted (`?`) won't be evaluated unless some later code inspects it.

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

  `?k` arguments are implemented as values that throw exceptions that carry which `k` they are. IRULAN then catches these exceptions, recognises them and then carries on to build real values to use in their place.

- **Failure:** `expression ==> ! Error`. This is reported if the evaluation of *expression* raises an uncaught exception (other than the argument exception mentioned above). For example, on line 7, `insert 1 (Branch ? 1 ?1)` causes the non implemented case of *insert* to be tripped as we try to insert 1 into a branch that already features a 1.

For normal use, a user of IRULAN is only interested in the failure case, and by default IRULAN hides success and refinement evaluations.

Looking in detail at Figure 3.3 we see that IRULAN finds the one exported function from the module, *insert*, and begins (on line 1) by testing it with no arguments. This is to see if the function is defined. Since it is defined (and is not e.g. *insert* = *error* `"TODO: implement insert"`), the test reports success with a "`.`".

On line 2 IRULAN then checks to see if *insert* is strict by passing in a `?` argument, which is a value that throws a special exception when it is evaluated. This is done so that arguments that are not evaluated due to Haskell's lazy evaluation scheme do not waste computation by having real expressions computed for them. In this case, evaluation completes successfully again, and IRULAN can then add a second argument to *insert* in line 3. However, since the evaluation on line 3 causes the `?1` exception to be thrown, IRULAN then goes in search of real values to use for that `?1` argument. The `?1` argument can be refined in two ways, either to a *Branch* `?1 ?2 ?3` (line 4), or a *Leaf* (line 15).

With the original `?1` argument refined to a *Branch*-constructed value, evaluation proceeds on line 4 and IRULAN

Figure 3.4 Haskell *IntTreeExample* module, with *insert* corrected and extended with a *delete* function.

```
module IntTreeExample where
data IntTree
  = Leaf
  | Branch IntTree Int IntTree
insert :: Int → IntTree → IntTree
insert n Leaf = Branch Leaf n Leaf
insert n (Branch left x right)
  | n < x = Branch (insert n left) x right
  | n > x = Branch left x (insert n right)
  | n ≡ x = Branch left x right
delete :: Int → IntTree → IntTree
delete n Leaf = Leaf
delete n (Branch left x right)
  | n < x  = Branch (delete n left) x right
  | n > x  = Branch left x (delete n right)
  | n ≡ x  = attachRight left right
attachRight :: IntTree → IntTree → IntTree
attachRight Leaf t = t
attachRight (Branch left x right) Leaf
  = Branch left x right
attachRight (Branch left x right) t@(Branch _ x' _)
  | x ⩾ x' = error "Precondition failure: x >= x'"
  | x < x' = Branch left x (attachRight right t)
```

discovers that *insert* now needs to refine its first argument (the *Int*). The possible choices for primitive values come from IRULAN's specified constant pool, and so the `?` argument is refined to either 1 (line 5) or 0 (line 10).

The process of refinement continues until the bugs are found (lines 7, 11), or the insert function is successfully executed (line 6, 13, 15). However, since the user has enabled case expressions, after the successful executions in lines 6, 13 and 15 (and other elided places), IRULAN will continue testing by inspecting the fields (if any) of the returned values, as can be seen on lines 16 - 18. Notice on line 18 this inspection causes the first argument of insert to be demanded, which causes it to be refined under the selector (lines 19 and 20). Through this process in the *Branch* case (the elided results in lines 9 and 14) the test cases involving case expressions in Figure 3.2 that cause crashes will also be found.

The bug can be fixed by add adding the following guard to the end of the *insert* function, and IRULAN will not report any test cases that cause crashes.

```
  | n ≡ x = Branch left x right
```

Carrying on development of our example, we have added a *delete* function, as shown in Figure 3.4. When the value to be deleted is found, the *attachRight* helper function is used to replace the rightmost *Leaf* in the left hand tree with the right hand tree. This maintains the implicit ordered invariant of the tree. Since it's also fairly easy to check part of that invariant while traversing in *attachOnRight*, we have added an explicit check for it.

Figure 3.5 Part of the output from running IRULAN in trace mode on the *IntTreeExample* with correct *insert* and new *delete* function. (output examples run through unix `sort` and `uniq`).

```
$ irulan --ints='[0,1]' -a --maximumRuntime=1 source IntTreeExample
IntTreeExample:
  Results:
  attachRight (Branch ? 0 ?1) (Branch ?2 0 ?3) ==> ! Precondition failure: x >= x'
  attachRight (Branch ? 1 ?1) (Branch ?2 0 ?3) ==> ! Precondition failure: x >= x'
  attachRight (Branch ? 1 ?1) (Branch ?2 1 ?3) ==> ! Precondition failure: x >= x'
  delete 0 (Branch (Branch ? 0 ?1) 0 (Branch ?2 0 ?3)) ==> ! Precondition failure: x >= x'
  delete 0 (Branch (Branch ? 1 ?1) 0 (Branch ?2 0 ?3)) ==> ! Precondition failure: x >= x'
  delete 0 (Branch (Branch ? 1 ?1) 0 (Branch ?2 1 ?3)) ==> ! Precondition failure: x >= x'
  delete 1 (Branch (Branch ? 0 ?1) 1 (Branch ?2 0 ?3)) ==> ! Precondition failure: x >= x'
  delete 1 (Branch (Branch ? 1 ?1) 1 (Branch ?2 0 ?3)) ==> ! Precondition failure: x >= x'
  delete 1 (Branch (Branch ? 1 ?1) 1 (Branch ?2 1 ?3)) ==> ! Precondition failure: x >= x'
```

However, when IRULAN is run on this several counter examples are found, as shown in Figure 3.5. What has happened is that we have exposed enough internal structure to mean that users of the IntTree API could violate the implicit invariant. For a start, *attachOnRight* is an internal helper function and shouldn't be exported, in addition, ideally, all *IntTree*s should be build using *insert* and *delete*, from an original *Leaf*. Adding an explicit export list would make this clear to other programmers, and also IRULAN. In addition an *empty* function can be exported to produce an empty tree:

> **module** *IntTreeExample* (*empty*, *insert*, *delete*, *IntTree*) **where**
>
> *empty* :: *IntTree*
>
> *empty* = *Leaf*
>
> ...

With this, IRULAN will see that the constructors for *IntTree* are not exported, and will automatically proceed to test by using *empty*, *insert* and *delete* to build *IntTree* values to test the *insert* and *delete* functions. With this change, rerunning IRULAN reports no errors.

3.2 Regression Testing

The next part of our example is going to add a simple balancing operation to the tree. It would be nice to precisely see what changes in behaviour this does. To do this, we will use IRULAN's regression testing functionality. When using IRULAN's regression testing functionality, a new form of success is added to IRULAN's output:

- **Regression Testing**: To enable regression testing (§3.2) the right hand side of an ==> may also be an arbitrary Haskell String drawn from the *show* function of the result.

Before making the change, we can make a snapshot of (input,output) pairs on the module. In order to get meaningful test data for output pairs, we have to make the *IntTree* renderable to a String using Haskell's built

in *Show* type class:

data *IntTree*
= *Leaf*
| *Branch IntTree Int IntTree*
deriving *Show*

Next, we run IRULAN specifying a location to create a test suite (a compressed log of (input,output) pairs). This file can be thought of as containing a snapshot of the behaviour of the module. In this case, we increased the number of *Int* values available, and up the runtime allocated to the iterative deepening exploration to 60 seconds.

```
$ irulan --int-range='(0,5)' -a --maximumRuntime=60 --full-testsuite=beforeBalance.tst
    source IntTreeExample
```

IRULAN can be used to print out the compressed test suite by passing it as a single argument in its test suite analyser mode (`tsa`):

```
$ irulan tsa beforeBalance.tst
...
A#  delete 0 (delete 0 (insert 1 empty)) ==> Branch Leaf 1 Leaf
A#  delete 0 (delete 0 (insert 1 (insert 0 empty))) ==> Branch Leaf 1 Leaf
A#  delete 0 (delete 0 (insert 1 (insert 1 empty))) ==> Branch Leaf 1 Leaf
A#  delete 0 (delete 0 (insert 1 (insert 2 empty))) ==> Branch (Branch Leaf 1 Leaf) 2 Leaf
A#  delete 0 (delete 0 (insert 1 (insert 3 empty))) ==> Branch (Branch Leaf 1 Leaf) 3 Leaf
A#  delete 0 (delete 0 (insert 1 (insert 4 empty))) ==> Branch (Branch Leaf 1 Leaf) 4 Leaf
A#  delete 0 (delete 0 (insert 1 (insert 5 empty))) ==> Branch (Branch Leaf 1 Leaf) 5 Leaf
A#  delete 0 (delete 0 (insert 2 (delete ? (delete ?1 empty)))) ==> Branch Leaf 2 Leaf
A#  delete 0 (delete 0 (insert 2 (delete ? empty))) ==> Branch Leaf 2 Leaf
A#  delete 0 (delete 0 (insert 2 empty)) ==> Branch Leaf 2 Leaf
...
```

In this case, there are 13,959 unique test cases stored in a 68k file.

In Figure 3.6 we add a *balance* function which attempts to balance the tree by rotating subtrees left and right as necessary. The previous *insert* and *delete* functions (and their recursive calls) are renamed to *insert′* and *delete′*, and new *insert* and *delete* functions are created that balance the results of their helper functions.

Again, IRULAN can be used to create a test suite of this new version:

```
>irulan --int-range='(0,5)' -a --maximumRuntime=60 --full-testsuite=afterBalance.tst
  source IntTreeExample
```

Figure 3.6 Haskell *IntTreeExample* module, with rebalancing after *insert* or *delete*.

```
module IntTreeExample (empty, insert, delete, IntTree) where
data IntTree
  = Leaf
  | Branch IntTree Int IntTree
  deriving Show
empty :: IntTree
empty = Leaf
insert :: Int → IntTree → IntTree
insert x t = balance $ insert' x t
insert' :: Int → IntTree → IntTree
...
delete :: Int → IntTree → IntTree
delete x t = balance $ delete' x t
delete' :: Int → IntTree → IntTree
...
depth :: IntTree → Int
depth Leaf = 0
depth (Branch left _ right) = 1 + max (depth left) (depth right)
balance :: IntTree → IntTree
balance Leaf = Leaf
balance (Branch left x right)
  = doRotate ldepth rdepth left' x right'
  where
    left' = balance left
    right' = balance right
    ldepth = depth left'
    rdepth = depth right'
doRotate :: Int → Int → IntTree → Int → IntTree → IntTree
doRotate lDepth rDepth (Branch ll l lr) x rt
  | lDepth > (rDepth + 1) = doRotate (lDepth − 1) (rDepth + 1) ll l (Branch lr x rt)
doRotate lDepth rDepth lt x (Branch rl r rr)
  | rDepth > (lDepth + 1) = doRotate (lDepth + 1) (rDepth − 1) (Branch lt x rl) r rr
doRotate lDepth rDepth lt x rt = Branch lt x rt
```

IRULAN can then be used to compare two test suites, and identify where changes in behaviour or strictness have occurred. To do this, IRULAN matches up equivalent inputs from the test suites, and sees if the outputs are the same or different. It then prints out the matched up inputs and outputs, prefixed by a symbol to indicate status:

- **:** - The inputs give the same output.
- **~** - The inputs give different output.
- # - There is no corresponding input in the other test suite (or in the one test suite case, there is no other test suite).

In Figure 3.7 we present a few of the matched inputs and outputs from IRULAN's test suite; many test cases have not changed behaviour (e.g. *empty* or simple *delete*s), however the rebalancing operation has altered the

shape of the tree in some of the more nested *insert* and *delete* cases.

There is an interesting edge case to consider, which is when the strictness of a function has changed. For example, imagine if a new *insert* function was faulty, ignoring the *Int* argument and always inserting 0. The test cases produced for such a function could look like:

```
insert ? empty             ==> Branch Leaf 0 Leaf
insert ? (insert ?1 empty) ==> Branch Leaf 0 (Branch Leaf 0 Leaf)
```

The test expression *insert* ? *empty* is more general than the test expression *insert* 1 *empty* or *insert* 2 *empty*. When IRULAN matches up test expressions, it groups together more general ones with their more specific counterparts from the other test suite. For example, a test suite analysis including the faulty *insert* function may report:

```
A: Original.tst
B: Faulty.tst
...
A: insert 0 empty ==> Branch Leaf 0 Leaf
A~ insert 1 empty ==> Branch Leaf 1 Leaf
A~ insert 2 empty ==> Branch Leaf 2 Leaf
B~ insert ? empty ==> Branch Leaf 0 Leaf
...
```

Here we see that IRULAN has grouped the three more specific inputs in the original test suite together with the one more general input in the faulty test suite. When deciding correctness, IRULAN will mark a more specific input correct (with a `:`) if it has the same output as the more general input, however the more general test case will only be correct if all of the more specific examples agree with it (in this case they don't, so the B input is marked faulty with a `~`).

3.3 Property Testing

In addition to just checking for runtime exceptions, IRULAN can also perform property testing. If IRULAN encounters a property function (one that starts with the prefix *prop_* and has a *Bool* result type), it will check whether the resulting *Bool* is *True* or *False*, and report test cases that produce *False* as errors. Property testing adds new forms of success and failure output to IRULAN:

- **Property Testing**: `expression ==> True`, `expression ==> False`. These are two specialisations of `expression ==> .` which denote that `expression` represents the testing of a Haskell property (a function with the prefix *prop_* and ultimate result type *Bool*), and that the test returned *True* or *False* respectively. From the point of view of testing, `expression ==>  False` is considered a failure.

Figure 3.7 Running IRULAN's test suite analyser on the test suites created before and after adding the *balance* function (output cut)

```
$ irulan tsa beforeBalance.tst afterBalance.tst
A: beforeBalance.tst
B: afterBalance.tst
  ...
A:  empty ==> Leaf
B:  empty ==> Leaf

A:  delete ? empty ==> Leaf
B:  delete ? empty ==> Leaf
  ...
A~  insert 0 (insert 1 (insert 2 empty)) ==> Branch (Branch (Branch Leaf 0 Leaf) 1 Leaf) 2 Leaf
B~  insert 0 (insert 1 (insert 2 empty)) ==> Branch (Branch Leaf 0 Leaf) 1 (Branch Leaf 2 Leaf)
  ...
A~  delete 0 (insert 1 (insert 4 (insert 5 empty))) ==> Branch (Branch (Branch Leaf 1 Leaf) 4 Leaf) 5 Leaf
B~  delete 0 (insert 1 (insert 4 (insert 5 empty))) ==> Branch (Branch Leaf 1 Leaf) 4 (Branch Leaf 5 Leaf)
  ...
```

Figure 3.8 Haskell *IntTreeExample* module, with a *flatten* function.

module *IntTreeExample* (*empty*, *insert*, *delete*, *flatten*, *IntTree*) **where**

...

flatten :: *IntTree* → [*Int*]
flatten Leaf = []
flatten (*Branch left x right*) = *flatten left* ++ [*x*] ++ *flatten right*

Figure 3.9 Haskell *IntTreeSort* module, with a faulty property to test the IntTreeExample.

module *IntTreeSort* **where**
import *Data.List hiding* (*insert*)
import *IntTreeExample*

prop_sort :: [*Int*] → *Bool*
prop_sort xs = *sort xs* ≡ (*flatten* ∘ *build* $ *xs*)

build :: [*Int*] → *IntTree*
build = *foldr insert empty*

Figure 3.10 Running IRULAN in property checking mode (`-p`) for 1 second of iterative deepening on the *IntTreeSort* module.

```
$ irulan --int-range='(0,5)' -a --maximumRuntime=1 -p source IntTreeSort
IntTreeSort:
 Results:
  prop_sort (: 0 (: 0 ([]))) ==> False
  prop_sort (: 1 (: 1 ([]))) ==> False
  prop_sort (: 2 (: 2 ([]))) ==> False
  prop_sort (: 3 (: 3 ([]))) ==> False
  prop_sort (: 4 (: 4 ([]))) ==> False
  prop_sort (: 5 (: 5 ([]))) ==> False
  prop_sort (: 0 (: 0 (: 0 ([])))) ==> False
  prop_sort (: 0 (: 0 (: 1 ([])))) ==> False
  prop_sort (: 0 (: 0 (: 2 ([])))) ==> False
  ...
```

One possible use for the *IntTreeExample* module we have been developing is to sort a list of numbers. To enable this, we need a *flatten* function to turn an *IntTree* into a list by performing a pre-order traversal (Figure 3.8), and a way to build an *IntTree* from a list by repeated calls to *insert*. In Figure 3.9 we show a new module, *IntTreeSort* which contains such a function, *build*, and also the property (*prop_sort*) expressing that normal list *sort* should be equivalent to $flatten \circ build$.

With no further work, IRULAN can be used to check the property, as shown in Figure 3.10. The `-p` switch puts IRULAN in property testing mode, which tells it to only test property functions (so it won't try and test the *build* function).

IRULAN identifies many inputs that show the property fails when there are duplicate items in the list. This is because the *IntTree* actually discards duplicates and our *prop_sort* property is faulty. The fault can be corrected by changing the property to:

$$prop\_sort\ xs = sort\ (nub\ xs) \equiv (flatten \circ build\ \$\ xs)$$

And rerunning IRULAN will not find any counter examples.

3.4 Minimized Test Suite Generation

IRULAN integrates with the Haskell Program Coverage (HPC, [GR07]) extension to GHC. During compilation, HPC instruments all subexpressions in the modules being compiled. The instrumentation records which subexpressions are entered during program execution. Typically at the end of a program's run this code coverage information is serialised to disk and can then be analysed to present to the user statistics and marked up source code showing which expressions were and were not executed. This mode of execution works with IRULAN, so we can see how effective IRULAN is at covering the expressions in the source code, a feature we will use during our experimental analysis in Chapter 5.

However, it is also possible to reify and alter this code coverage information at runtime. By resetting the coverage information so no subexpression thinks it has been executed, and then running a single test expression, IRULAN can ascertain precisely the coverage *footprint* of each test expression. These footprints can then be used to build a minimized test suite that achieves the same amount of code coverage as the full suite of expressions IRULAN has tried, but with many fewer expressions. The idea is that as IRULAN executes, it keeps track of the current minimized test suite. If a newly executed test expression has a footprint that covers subexpressions that have not yet been executed, it is added to the test suite. In addition, test expressions that subsume other test expressions are added (and the subsumed test expressions in both cases are removed). At the end of execution, IRULAN can then report the surviving test suite.

For example, running on our latest *IntTreeExample*, with HPC code coverage (`-fhpc`) and test suite generation (`--enable-testsuite`) enabled yields the trace in Figure 3.11. Many, many simple expressions (such as *empty* or *insert* 0 *empty*) have been pruned as their behaviour is also covered by these larger examples. The potential applications of this technology are still to be explored, but we present this as an interesting first step.

Figure 3.11 Running IRULAN on the *IntTreeExample* to generate a minimized testsuite

```
$ irulan --ints='[0,1]' --hpc-testsuite -a --maximumRuntime=5 --disable-show-results
    --enable-case-statements source --ghc-options='-fforce-recomp -fhpc' IntTreeExample
IntTreeExample:
 Results:

Test Suite:
  insert 0 (delete 1 (insert 0 (insert 1 empty))) ==> .
  delete 1 (insert 1 (insert 0 empty)) ==> .
  delete 0 (insert 0 (insert 1 empty)) ==> .
  insert 0 (delete 1 (insert 0 empty)) ==> .
  insert 0 (delete 0 (insert 1 empty)) ==> .
  insert 1 (insert 1 (insert 0 empty)) ==> .
  insert 0 (insert 0 (insert 0 empty)) ==> .
  case flatten (insert ? empty) of _ : x -> x ==> .
  case flatten (insert 0 empty) of x : _ -> x ==> .
```

3.5 Code Coverage and Advanced Features

We now present a couple of more advanced features of IRULAN through a final evolution of example. We show that IRULAN can instantiate and test polymorphic functions, that it has some support for automatically satisfying type class constraints, and that it can interact with HPC to produce standard code coverage output.

Figure 3.12 Haskell *TreeSort* module, with a property to test the *TreeExample*.

```
module TreeSort where
import Data.List hiding (insert)
import TreeExample
prop_sort :: (Ord a) ⇒ [a] → Bool
prop_sort xs = sort (nub xs) ≡ (flatten ∘ build $ xs)
build :: Ord a ⇒ [a] → Tree a
build = foldr insert empty
```

Figure 3.13 Haskell *Nat* module, with a definition of peano naturals.

```
module Nat where
data Nat
   = Zero
   | Succ Nat
  deriving (Eq, Ord)
```

In Figure 3.14 we have generalised our *IntTree* implementation to work on more general *Tree*s. The *Tree* data type is parameterised by a type a which is the type of the elements in the tree. New, empty trees are built by *empty*, *insert* and *delete* as before, however *insert* and *delete* require that the elements in the tree are comparable using the *Ord* type class.

We have also updated the *IntTreeSort* to work with the new *Tree* type, as shown in Figure 3.12. In addition,

we have implemented a small peano natural data type which implements *Ord*, and placed that in a new module *Nat*, shown in Figure 3.13.

IRULAN is able to test the *prop_sort* property even though it has a polymorphic type. If an argument with a polymorphic type is refined, IRULAN will attempt to find values that will safely unify with that type. If for example we run IRULAN and explicitly add the *Nat* module to IRULAN's set of support (but no *Int*s), we will may see output such as the below:

```
$ irulan -p -a -iNat --maximumRuntime=10 --trace source --ghc-options='-fhpc -fforce-recomp' TreeSort
...
prop_sort $fOrdNat (: (Succ Zero) (: Zero ([]))) ==> True
...
```

A small limitation is that IRULAN will make explicit the type class argument to the polymorphic function, and state the instance that has been used, e.g. in the above example the $*fOrdNat* parameter indicates the *Ord* instance for *Nat* is being used.

By also enabling HPC (`--ghc-options='-fhpc'`), IRULAN will create an `irulan.tix` file that can be processed by the HPC toolchain:

```
$ hpc report --per-module irulan.tix
-----<module Nat>-----
100% expressions used (0/0)
100% boolean coverage (0/0)
     100% guards (0/0)
     100% 'if' conditions (0/0)
     100% qualifiers (0/0)
100% alternatives used (0/0)
100% local declarations used (1/1)
 66% top-level declarations used (2/3)
-----<module TreeExample>-----
 68% expressions used (107/156)
 40% boolean coverage (4/10)
      40% guards (4/10), 1 always True, 5 unevaluated
     100% 'if' conditions (0/0)
     100% qualifiers (0/0)
 61% alternatives used (13/21)
100% local declarations used (4/4)
 58% top-level declarations used (7/12)
-----<module TreeSort>-----
100% expressions used (12/12)
100% boolean coverage (0/0)
     100% guards (0/0)
```

```
      100% 'if' conditions (0/0)
      100% qualifiers (0/0)
 100% alternatives used (0/0)
 100% local declarations used (0/0)
 100% top-level declarations used (2/2)
```

From the above run, we can see that the *prop_sort* property and IRULAN have tested most of the *TreeExample* module but not all. In fact, *prop_sort* does not test the *delete* and *delete'* functions or the *Show* instance for *Tree*, which accounts for the missing code coverage.

3.6 Discussion

In this chapter we have looked at several use cases of IRULAN through a growing example. We started by using IRULAN as a quick and easy way to see if there are inputs that cause a library to crash. The ideas used to make this happen (automatic inference of identifiers, enumerating values of the appropriate type to pass in as arguments to a function, using needed narrowing to avoid generating test data that isn't evaluated) are all used by existing tools, but we believe this is the first work to integrate them together to search for arbitrary error conditions in Haskell programs.

In addition, we have shown some novel features for Haskell testing tools. IRULAN can synthesise selectors using case expressions, which allows it to explore within returned values. This exploration can uncover errors that would otherwise be hidden by laziness, and the case expression will precisely identify which field of the returned value was hiding the error. Of course, this technology isn't needed by property testing tools, as the resulting value is always a *Bool*, however IRULAN's case expression synthesis can also be used to generate test data to be used as arguments to functions, by picking apart more complicated data structures (such as tuples or lists) to extract useful test data that may be held within.

Also, unlike many of the existing Haskell library based testing approaches to testing, IRULAN is able to perform error finding (and thus property) testing on polymorphic functions, where it instantiates values that would be polymorphic by unifying them with possible values from the inferred data set.

The regression testing demonstration is a demonstration that the ideas within IRULAN (automatically generating input and output pairs) can be put to other uses than just error finding. In Section 5.5 we talk in detail about some small case studies where we managed to apply the regression testing work; this looks to be a fruitful line of research we hope to develop further.

One important consideration when looking at any testing tool is ensuring that the user is really sure about what has and has not been tested. A common problem when using random or iterative deepening testing tools is that a huge number of tests may be reported, but many may be identical. For example, a naive random generator for lists would make 50% of all tests the empty list.

One way to quickly ascertain what is not being tested is to see what a code coverage tool reports as not being explored. For this reason we have made sure that IRULAN can interoperate well with Haskell Program Coverage. The ability to quantitatively measure the code coverage achieved by IRULAN on arbitrary source code modules is made use of throughout our evaluation in Chapter 5.

Using HPC, we also hinted at some experimental open work with IRULAN. By being able to reify code coverage results, we are able to generate a minimized test suite of expressions. This could have potential applications in several areas, for example for quick regression testing or helping to classify the expressions that cause errors. However, it is important to stress that this is a minimized, not a minimal test suite. The order of execution of test cases can affect what test suite is reported. To generate a fully minimized test suite would reduce to the set covering problem, which is NP-complete.

We have also showed that IRULAN can interact with some of Haskell's more advanced language features, such as type classes. Internally IRULAN sees a type class constraint as an extra parameter to the function that accepts a value witnessing the instance of the type class for that particular type. A limitation of IRULAN is that it currently makes these extra parameters and witnesses visible to the user in its expressions, although it should be possible with some further work to hide this. IRULAN currently does not support testing type class instance declarations directly, and while this should be possible to do in theory, it is an aspect of implementation we have not yet explored.

One very visible difference between IRULAN and the existing testing tools such as SmallCheck is how the depth of a test is measured. As we will discuss in Section 4.4, IRULAN's default depth metric is not as intuitive; for that reason we recommend IRULAN be used in an iterative deepening fashion. The use of iterative deepening does mean that at times IRULAN can report duplicate test cases that cause errors, however IRULAN's output has been designed to operate well with standard filtering tools (such as Unix `sort` and `uniq`), so this is not a real problem in practice.

Another difference with the library based techniques for property testing is that their raw performance (in terms of speed of expression generation) is much higher than that of IRULAN, a point we will discuss in Section 5.2.1. For property testing it is then a trade-off for the user between faster testing and automatic inference of test data and the ability to test polymorphic functions. It would be interesting future work to integrate IRULAN's automatic discovery of test data with a type class based testing routine (such as used by Lazy SmallCheck) to gain performance in the property testing case.

Figure 3.14 Haskell *TreeSort* module, a polymorphic, more generalised version of the *IntTreeExample*.

```
module TreeExample (empty, insert, delete, flatten, Tree) where
data Tree a
   = Leaf
   | Branch (Tree a) a (Tree a)
  deriving Show
empty :: Tree a
empty = Leaf
insert :: (Ord a) ⇒ a → Tree a → Tree a
insert x t = balance $ insert' x t
insert' :: (Ord a) ⇒ a → Tree a → Tree a
insert' n Leaf = Branch Leaf n Leaf
insert' n (Branch left x right)
   | n < x = Branch (insert' n left) x right
   | n > x = Branch left x (insert' n right)
   | n ≡ x = Branch left x right
delete :: (Ord a) ⇒ a → Tree a → Tree a
delete x t = balance $ delete' x t
delete' :: (Ord a) ⇒ a → Tree a → Tree a
delete' n Leaf = Leaf
delete' n (Branch left x right)
   | n < x  = Branch (delete' n left) x right
   | n > x  = Branch left x (delete' n right)
   | n ≡ x = attachRight left right
attachRight :: (Ord a) ⇒ Tree a → Tree a → Tree a
attachRight Leaf t = t
attachRight (Branch left x right) Leaf
   = Branch left x right
attachRight (Branch left x right) t@(Branch _ x' _)
   | x ⩾ x' = error "Precondition failure: x >= x'"
   | x < x' = Branch left x (attachRight right t)
depth :: Tree a → Int
depth Leaf = 0
depth (Branch left _ right) = 1 + max (depth left) (depth right)
balance :: Tree a → Tree a
balance Leaf = Leaf
balance (Branch left x right) = doRotate ldepth rdepth left' x right'
  where
    left'  = balance left
    right' = balance right
    ldepth = depth left'
    rdepth = depth right'
doRotate :: Int → Int → Tree a → a → Tree a → Tree a
doRotate lDepth rDepth (Branch ll l lr) x rt
   | lDepth > (rDepth + 1) = doRotate (lDepth − 1) (rDepth + 1) ll l (Branch lr x rt)
doRotate lDepth rDepth lt x (Branch rl r rr)
   | rDepth > (lDepth + 1) = doRotate (lDepth + 1) (rDepth − 1) (Branch lt x rl) r rr
doRotate lDepth rDepth lt x rt
   = Branch lt x rt
flatten :: Tree a → [a]
flatten Leaf = []
flatten (Branch left x right) = flatten left ++ [x] ++ flatten right
```

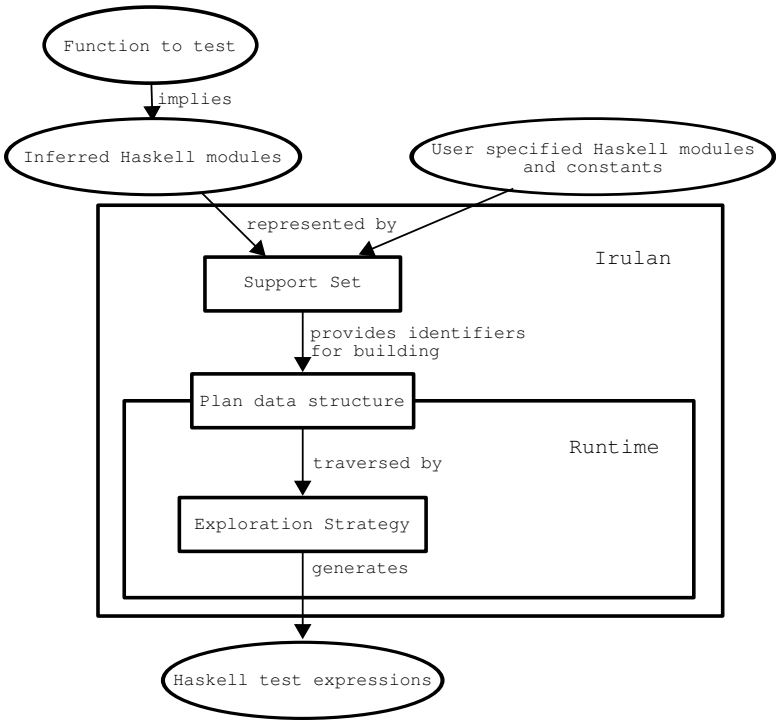
Chapter 4

Implementation of IRULAN

In the previous chapter, we saw how a user would typically interact with IRULAN, and showed some of the parameters that can be used to change its internal behaviour. In general, the user asks IRULAN to load some Haskell modules (using GHC to pre-compile them if necessary), and then IRULAN will automatically generate expressions to test the functions exported from those modules. In order to construct the expressions, IRULAN will have to find functions in imported modules, and may need to be provided with constants by the user. The order and number of test expressions that IRULAN generates is controlled by an exploration strategy (such as depth first or random search), which the user can also specify and parametrise.

In this chapter we discuss the core concepts, algorithms and data structures that underpin the implementation of IRULAN to make the above happen. Figure 4.1 presents these concepts graphically.

Figure 4.1 An overview of the main components of IRULAN



The core of IRULAN has three inputs; a function to test, a set of Haskell modules and a set of primitive constants. The set of modules is inferred by looking at the dependencies of the function to test, i.e. which modules declared the argument and result types of the test function. As Haskell has some built in values (e.g. *Int*, *Double* etc.) that do not have a data constructor representation, IRULAN requires that values of these types that it should use be stated explicitly. From these inputs, IRULAN automatically tests the given function, using the functions and data constructors exported from the support modules, and constants provided.

IRULAN first uses the modules and constants provided to build a *support set*. The support set is IRULAN's own database of identifiers, constructors and constants that can be used to generate test data. The identifiers are extracted from modules using a reflection-like mechanism presented by the GHC API. When concrete test data is needed as an argument to the function being tested, the support set will be queried to provide values of the type of that argument. There are two query functions provided by the support set; one provides identifiers that can be applied to arguments to get the type required, the other builds a state machine that expresses sequences of case expressions that can extract a value of the desired type out of a more complicated value.

IRULAN's discovery and representation of a support set differs from comparable Haskell techniques in several ways. Constructors and functions that can be used to build test data are discovered by loading compiled or source Haskell modules into GHC and then using the GHC API as a reflection library to introspect their exported APIs. IRULAN then extracts identifiers as appropriate from GHC and uses two novel data structures and associated algorithms to implement the query functions that the support set provides.

IRULAN uses the support set to generate *test expressions*, which are Haskell expressions to be executed under observation to see if they crash. The test expressions are built in a *Plan*. This is a lazy, tree-like data structure that encodes what test expressions to execute, and, based on the result of running a test expression, which test expressions should be run next. The *Plan* has many features; it takes advantage of lazy evaluation to only create test expressions for arguments when needed (hence the *Plan* box being half in and half out of the Runtime in Figure 4.1), it is novel in that it can build expressions using Haskell's case expressions to uncover bugs that may be hidden by laziness and to create extra test data, and another novel feature is that it can test polymorphic functions without requiring them to have an up front fully monomorphic type. IRULAN differs from existing techniques in that it makes the *Plan* an explicit structure, as opposed to implicit in the execution routines of other tools. This allows for easy experimentation, for example different ways of traversing the *Plan* can be expressed abstractly without worrying about how it is produced. In addition it allows for easy visualisation.

The *Plan* is used by IRULAN's runtime. Since the *Plan* is potentially infinite in size with a high branching factor, the way it is explored and the order of the execution of test expressions from it will affect ultimate testing coverage. We have implemented several different strategies in IRULAN for exploring the *Plan*. All of these strategies use the *Runtime* to execute test expressions and interrogate them to find out their results.

The *Runtime* also differs from existing tools. It is unique amongst existing Haskell tools in that it monitors the expressions it executes to safeguard itself from excessively long running or high allocating expressions. We also investigated adding caches in at the runtime layer, looking at the actual values returned and pruning executing expressions to see if there could be any potential benefits to be gained.

Figure 4.2 The syntax of types that IRULAN uses

```
Type ::= forall tyVar . Type  [ Parameterised type ]
       | RawType              [ Raw Type ]

RawType ::= tyvar             [ Type variable ]
          | TyCon             [ Type constructor ]
          | RawType RawType   [ Type application ]
```

As a practical matter, the user of IRULAN will usually specify a module to test, and IRULAN will then test all of the exported functions in that module. Although the presentation will focus on just a single function being tested (as that is conceptually simpler), we will detail how each component moves from working with a single function to test to a module's worth.

4.1 Support Set

In order to test a function, IRULAN will need to generate concrete test data to pass to it as arguments. To build this test data, 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.

IRULAN uses type information to incrementally build up its support set (§4.1.1). During construction of the support set (§4.1.2), two novel data structures are built; a *TypeMap* (§4.1.3), which maps wanted types to the functions that can provide them; and a *Constructor Graph* (§4.1.4), which links data type constructors to the types they can provide via case expressions.

In order to test all the functions in a module, it would be possible to build a support set for each individual function. However, it is simpler to just build a single support set which is the union of the support sets for each individual function.

4.1.1 Types

The construction of the support set is driven by the static type information attached to the functions to be tested. IRULAN assumes a simple subset of GHC's *Core* [SCJD07] model of types, which can represent the types seen in a Haskell '98 program. A syntax corresponding to these types is given in Figure 4.2.

Types explicitly declare their free type variables with `forall`s at the top level, restricting all types to be rank one. Raw types can then be the ground type constructors (`TyCon`s), type variables (`tyvar`) that were introduced in the root `forall.` , or the application of one raw type to another.

Function types are constructed using the type constructor `(->)`. In many cases we will discuss the argument types and return type of a function. Since functions in Haskell are usually Curried, we will assume that a function of the form $T1 \rightarrow T2 \rightarrow .. Tn \rightarrow Tr$ will have argument types $T1, T2 .. Tn$ and result type Tr.

However, there may be cases where we are discussing higher order functions as test data. In relation to this, we have a notion of the *provided* result types of an identifier. Since functions can be used as values, these provided types are all the types the identifier (when applied to some arguments) could be used as a value of. For example, consider an identifier $foo :: A \to B \to C \to D$. Its normal argument types are A, B, and C, and its result type is D. Its provided types are D, $C \to D$, $B \to C \to D$ and $A \to B \to C \to D$

Since IRULAN operates on GHC's internal representation of data and types, there are some features of Haskell syntax that IRULAN does not need to deal with directly. For example, type classes are compiled by GHC using the dictionary passing transform [JW91, HHPJW96]. Here, type class declarations are turned into data declarations, type class instances are identifiers that return a value of the data type, and type class contexts on functions are turned into extra arguments to the function of the appropriate data type.

4.1.2 Construction Algorithm

The construction of the support set happens in two phases. In the first phase the given module to test is traversed and IRULAN infers useful constructors and functions from that module and the modules it may (transitively) depend upon. In the second phase, any command-line specified constants, and the functions/constructors exported by command-line specified modules are included.

The module to be tested is examined to find the set of functions to be tested. Normally this would consist of all of the exported function symbols from that module, however if property checking mode is enabled only the property functions (those that are prefixed with *prop* and return a *Bool*) will be selected. The types of the arguments of each test function are used to guide discovery of identifiers that will be useful for testing.

In the first step, IRULAN examines the type of each argument of the test function. i.e. it splits a function type $T1 \to T2 \to .. Tn \to TR$ to produce a list of argument types $T1 .. Tn$ and a result type TR. Then, for each argument type Ti, IRULAN extracts the type constructors mentioned within the type. For each of these type constructors Tc, IRULAN inspects the APIs of the module declaring Tc, and adds all the constructors and functions declared in that module that return expressions where Tc is applied to type arguments. This process then continues recursively on any newly added support functions.

This means that if the argument type T consists of an application of a type constructor to some argument types (i.e. is of the form $Tc\ T1\ T2$, e.g. *Maybe Bool*) then IRULAN will recursively search both the type constructor's declaring module, and the declaring modules of the argument types. This is useful for capturing all the types an instantiated data type argument will need to create instances of that data type.

For example, when processing the test function $isJustTrue :: Maybe\ Bool \to Bool$, IRULAN will look at the argument of type *Maybe Bool*. If IRULAN only added ways of making *Maybe* to its support set (i.e. $Nothing ::$ *Maybe a* and $Just :: a \to$ *Maybe a*) then there would be no way to create values of type *Maybe Bool* as *Bool* had not been added. In general, parameterised data types of the form $T\ a$ will have functions $a \to T\ a$ to produce them (e.g. *Just* for *Maybe* or (:) for lists). In the cases where they don't (perhaps the T in $T\ T1$ is abstract

and a function *mkT* :: *T T1* is exported) IRULAN may have conservatively added definitions for constructing values of type *T1*, but this will not adversely affect the construction of tests.

Visible Constructor Optimisation

The size of the state space that IRULAN needs to explore increases exponentially with the number of elements in the support set for a given type. To avoid an exponential blow-up of the state space, IRULAN uses the following important heuristic: if all the constructors of a type constructor *Tc* are exported by a module, then IRULAN only adds them to the support set, ignoring any functions that return expressions of type *Tc*. The insight here is that when all of *Tc*'s constructors are available, we can (almost always) generate all expressions of a saturated *Tc*[1]. Only when a module does not make *Tc*'s constructors available do we need to use functions that return expressions of type *Tc*.

There are frequently cases where IRULAN will see a type constructor that it has previously processed during support set construction. To prevent IRULAN repeating work, and potentially going into an infinite loop, IRULAN keeps track of all type constructors it has seen, and does not attempt to process a type constructor if it has already done so.

4.1.3 The *TypeMap*

When IRULAN runs, it will query the support set to find functions or constructors that can provide values of a type that are needed for a test functions' argument. In order to facilitate this, a generalized trie [CM95], called the *TypeMap*, maps *Type*s to sets of *Id*entifiers that provide them. We use a custom data type to support this query, as opposed to reusing an existing map library, as we wish to provide a lookup function that can return values whose keys unify with the type being queried for. In this section, we will construct in Haskell a simplified form of the *TypeMap* that underpins the implementation of IRULAN.

We will need a Haskell-equivalent form of the types from Section 4.1.1. These are shown in Figure 4.3, along with an assumed API for *Set*s and normal key-value *Map*s that will be needed during development of the *TypeMap*.

The representation of *Type*s and *RawType*s follows directly from the abstract syntax in Figure 4.2. The assumed *Map* API expects *Monoid* values, to make usage slightly more natural. The *mappend* operator from *Monoid* is used to combine values during *mapInsert* if an existing value exists, and *mempty* is used to provide a default value should *mapLookup* fail. The *Monoid* instance for *Map* also uses the *mappend* operator from the value to combine them in a similar manner to *mapInsert*. The *Monoid* instance for *Set*s is standard, with *mempty* being the empty set, and *mappend* as union. We also include an API for managing explicit substitutions (mappings from *TyVar*s to *RawType*s). The identity *Subst*itution is *mempty* from its *Monoid* instance, and two substitutions can be unioned using *mappend*. *Subst*itutions are built from a successful unification of two *RawType*s, and a *Subst*itution can be applied to a *RawType*.

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

Figure 4.3 The Haskell form of *Type*s, and an API for the exposition.

```
data Type
  = Forall TyVar Type
  | RawType RawType
data RawType
  = TV TyVar
  | TC TyCon
  | RawType `TA` RawType
newtype TyVar
newtype TyCon
newtype Map k v
instance (Eq k, Monoid a) ⇒ Monoid (Map k a)
mapInsert     :: (Eq k, Monoid a) ⇒ k → a → Map k a → Map k a
mapSingleton :: (Eq k, Monoid a) ⇒ k → a → Map k a
mapLookup     :: (Eq k, Monoid a) ⇒ k → Map k a → a
mapHasKey     :: (Eq k) ⇒ k → Map k a → Bool
mapElems      :: Map k a → [(k, a)]
newtype Set a
instance (Eq a) ⇒ Monoid (Set a)
setInsert     :: Eq a ⇒ a → Set a → Set a
setSingleton  :: Eq a ⇒ a → Set a
setNotElem    :: Eq a ⇒ a → Set a → Bool
newtype Subst
instance Monoid Subst
unify :: RawType → RawType → Maybe Subst
applySubst :: Subst → RawType → RawType
newtype Id
```

The generalized trie structure that corresponds to *RawType* can be derived:

```
data TypeMap a
  = Empty
  | Split { tyVars :: Map TyVar a
          , tyCons :: Map TyCon a
          , appTys :: TypeMap (TypeMap a)
          }
```

The intuition is that a *TypeMap a* maps a given *RawType* to some value *a*. The value corresponding to a *TyVar* type and a *TyCon* type can be looked up in the *tyVars* and *tyCons* maps respectively. For type applications, we have to look up a value based on both the left and right type children of the type application. To facilitate this, *appTys* uses a *TypeMap* to map the left child type to a map for the right child, and the nested map maps to the ultimate value.

TypeMap has a straightforward *Monoid* instance, with *mempty* being the empty map, and *mappend* the union of keys and values:

```
instance (Monoid a) ⇒ Monoid (TypeMap a) where
  mempty = Empty
  Empty `mappend` x = x
  x `mappend` Empty = x
  (Split tvl tcl atl) `mappend` (Split tvr tcr atr)
    = Split { tyVars = tvl `mappend` tvr
            , tyCons = tcl `mappend` tcr
            , appTys = atl `mappend` atr
            }
```

To make the *TypeMap* useful, we need to be able to insert *RawType* to value mappings into it.

```
tmInsert :: (Monoid a) ⇒ RawType → a → TypeMap a → TypeMap a
```

Inserting a value into an *Empty* typemap first builds a *Split* with empty children, and then recursively calls *tmInsert* on that:

```
tmInsert ty it Empty = tmInsert ty it emptySplit
  where
    emptySplit :: Monoid a ⇒ TypeMap a
    emptySplit = Split { tyVars = mempty
                       , tyCons = mempty
                       , appTys = mempty
                       }
```

Inserting a value at a type variable or a type constructor is then just a case of updating the appropriate map in the *Split*:

$$tmInsert\ (TV\ tyVar)\ it\ s@(Split\ \{tyVars\}) = s\ \{tyVars = mapInsert\ tyVar\ it\ tyVars\}$$
$$tmInsert\ (TC\ tyCon)\ it\ s@(Split\ \{tyCons\}) = s\ \{tyCons = mapInsert\ tyCon\ it\ tyCons\}$$

Inserting a type application happens in two parts. The *appTys TypeMap* needs to be updated to include a mapping for the left sub-type that maps to a map mapping the right sub-type to the desired value:

$$tmInsert\ (TA\ tyl\ tyr)\ it\ s@(Split\ \{appTys\}) = s\ \{appTys = tmInsert\ tyl\ part2\ appTys\}$$
where
$$part2 = tmInsert\ tyr\ it\ emptySplit$$

With the implementation of *tmInsert*, we can build an example *TypeMap*. In Figure 4.4 and Figure 4.5 we show an example support set and the *TypeMap* it induces. The nodes (solid dots) represent *TypeMap*s or *Map*s, depending on the type of edges leaving them. Solid edges specify the node was a *TypeMap*'s *Split*, and the edge is labelled by the constructor of *RawType* we are looking up in a *Split*; *TV* goes into *tyVars*, *TC* goes into *tyCons* and *TA* goes into *appTys*. Dashed edges specify the node was a *Map*, and the edge is labelled by a key in the map.

mkFilling and *mkPie* have been inserted twice, as these are functions that can provide values of two types. For example *mkPie* can provide test data of type $Filling\ a \to Pie\ a$, but if it is applied to an argument, it can also provide test data of type $Pie\ a$. During our experimental evaluation, we have seen *TypeMap*s need to store up to 229 identifiers. However the size of that particular *TypeMap* was 1031, i.e. each identifier provided 4.5 types on average.

If the *TypeMap* were treated as a normal trie structure then the lookup function, which finds identifiers that can provide test data of a certain type, would be achieved by navigating through the structure of the trie based on the type wanted, and returning the set of identifiers at the end (if any). However polymorphism complicates this story as other types beyond the syntactical matches could be appropriate. For example, imagine we need test data of (i.e. we want to lookup identifiers that can provide) type *Pie Apple*. In our example support set there are no functions that provide that type directly, but *mkPie* would be suitable if a is instantiated to *Apple*.

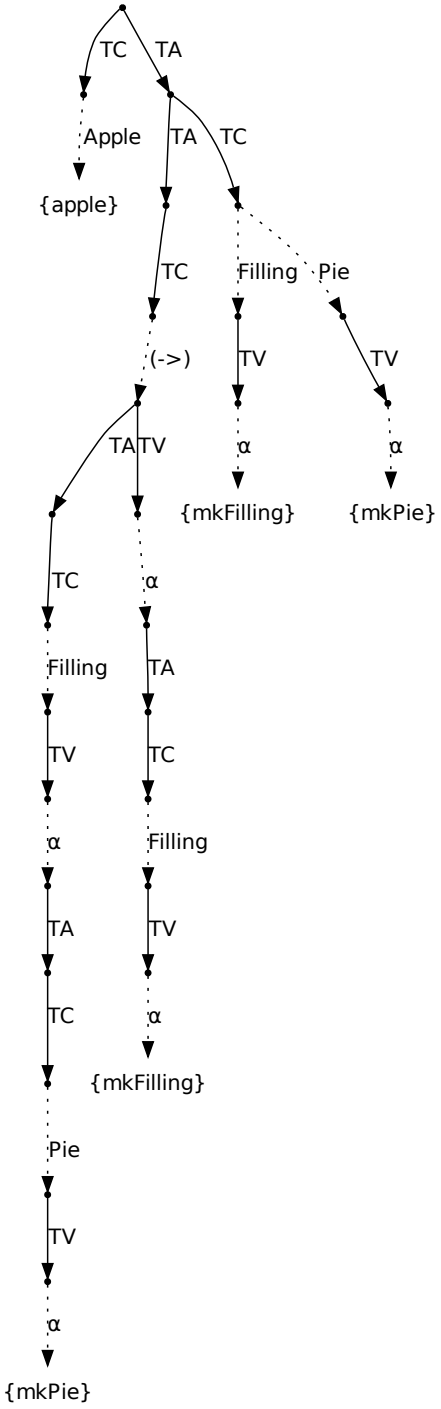
Possibly counter-intuitively, the instantiations can also work in the other direction. If a function needs test data with a type variable in it, for example of type $Pie\ a$, then that means the function being instantiated could be, for example $::Pie\ a \to Box \to Present\ a$. We could use a test value of type *Pie Apple*, unifying a to *Apple*, and test the function at type $Pie\ Apple \to Box \to Present\ Apple$.

Before presenting the Haskell implementation of lookup in *TypeMap*s, we will first outline how lookup works. This will require being explicit about the unifications and substitutions that allow finding the two examples above. We begin by defining abstractly a *TypeMap* that is either `Empty` or a triple of mappings for type variables, type constructors, and type applications:

Figure 4.4 An example set of functions to include in a support set

apple :: *Apple*
mkFilling :: $a \rightarrow$ *Filling a*
mkPie :: *Filling a* $\rightarrow$ *Pie a*

Figure 4.5 The *TypeMap* induced by the support set in Figure 4.4. Solid arcs are *TypeMap* lookups, dashed arcs are *Map* lookups



$$\texttt{TyVars}_\alpha \in \mathcal{P}(\texttt{TyVar} \times \alpha) \tag{4.1}$$

$$\texttt{TyCons}_\alpha \in \mathcal{P}(\texttt{TyCon} \times \alpha) \tag{4.2}$$

$$\texttt{AppTys}_\alpha \in \texttt{TypeMap}_{\texttt{TypeMap}_\alpha} \tag{4.3}$$

$$\texttt{TypeMap}_\alpha \in \texttt{Empty} + \texttt{TyVars}_\alpha \times \texttt{TyCons}_\alpha \times \texttt{AppTys}_\alpha \tag{4.4}$$

As a syntactical nicety, we will also define element inclusion ($\in$) for *TypeMaps*, based on inclusion from flattening the *TypeMap*. We will use this definition during the specification of *lookup*.

$$x \in (\texttt{tyVars} \times \texttt{tyCons} \times \texttt{appTys}) \leftrightarrow \quad x \in \texttt{tyVars} \cup \texttt{tyCons} \cup \{(t_1\ t_2, v) | (t_1, m) \in \texttt{appTys} \wedge (t_2, v) \in m\} \tag{4.5}$$

Lookup ($\simeq$) will take a *RawType* to extract values from a *TypeMap*, where the value's key unifies with the *RawType*. Since a unification has taken place, the returned values are paired with the substitution created by the unification.

$$\texttt{RawType} \simeq \texttt{TypeMap}_\alpha \in \mathcal{P}(\sigma \times \alpha) \tag{4.6}$$

$$t \simeq \texttt{Empty} = \emptyset \tag{4.7}$$

$$\begin{aligned} \alpha \simeq (\texttt{tyVars}, \texttt{tyCons}, \texttt{appTys}) =& \{(\alpha/\beta, v) | (\beta, v) \in \texttt{tyVars}\} \\ & \cup \{(\alpha/C, v) | (C, v) \in \texttt{tyCons}\} \\ & \cup \{(\alpha/t, v) | (t, v) \in \texttt{appTys}\} \end{aligned} \tag{4.8}$$

$$\begin{aligned} C \simeq (\texttt{tyVars}, \texttt{tyCons}, \texttt{appTys}) =& \{(\alpha/C, v) | (\alpha, v) \in \texttt{tyVars}\} \\ & \cup \{(\emptyset, v) | (C, v) \in \texttt{tyCons}\} \end{aligned} \tag{4.9}$$

$$\begin{aligned} t_1\ t_2 \simeq (\texttt{tyVars}, \texttt{tyCons}, \texttt{appTys}) =& \{(\alpha/t_1\ t_2, v) | (\alpha, v) \in \texttt{tyVars}\} \\ & \cup \{(\sigma' \cup \sigma, v) | (\sigma, m) \in (t_1 \simeq \texttt{appTys}) \wedge (\sigma', v) \in (\sigma(t_2) \simeq \sigma(m))\} \end{aligned} \tag{4.10}$$

In definition 4.10, it is syntactically convenient to apply the first substitution, σ to both the raw type t_2 and the `TypeMap` m. In the Haskell implementation, instead of substituting the entire *TypeMap* and then matching it, we explicitly only substitute through what we need to.[2]. We achieve this by passing into the lookup function a substitution to be applied to type variables in the *TypeMap* before trying to match them. The type signature

[2] In Irulan the *TypeMap* is a strict data type for performance reasons, so a lazily evaluated substitution would not work

and default case for the empty *TypeMap* are therefore:

```
tmLookup :: (Monoid a) ⇒ RawType → Subst → TypeMap a → [(Subst, a)]
tmLookup _ _ Empty = []
```

Looking up a type variable requires finding all elements (*RawType*, value pairs) in the *TypeMap*, where the inclusion relation ($\in$) is implemented as the descriptive *flattenTypeMap* (that also has to take a substitution to apply to type variables). Then each type *ty* found in the *TypeMap* is unified with the original type *t* (the type variable *tv*):

```
tmLookup t@(TV tv) s tm = [(s `mappend` subst, a)
                          | (ty, a) ← flattenTypeMap tm s
                          , let Just subst = unify t ty
                          ]
```

The other two cases for *tmLookup* (looking up a type constructor or a type application) share having to look up possible values in the *tyVars* field of the *TypeMap*, we factor this common code into *possiblesInTyVars*, implemented later.

In the case of looking up a type constructor *tc* , we check that the *tyCons* map does actually have *tc* as a key before returning the substitution passed into us and the value it maps to.

```
tmLookup t@(TC tc) s tm = [(s, mapLookup tc (tyCons tm))
                          | True ← [mapHasKey tc (tyCons tm)]
                          ] ++ possiblesInTyVars t s (tyVars tm)
```

Finally, looking up a type application requires threading through the incoming substitution *s* to make sure it is applied to the left hand side (*tl*), and then taking the modified substitution *s1* and threading that through the right hand side (*tr*) before returning it and any resulting value.

```
tmLookup t@(TA tl tr) s tm = [(s2, a2)
                             | (s1, a1) ← tmLookup tl s (appTys tm)
                             , (s2, a2) ← tmLookup (applySubst s1 tr) s1 a1
                             ] ++ possiblesInTyVars t s (tyVars tm)
```

For completeness, we define the *flattenTypeMap* and *possiblesInTyVars* to complete our *TypeMap* definition.

```
flattenTypeMap :: TypeMap a → Subst → [(RawType, a)]
flattenTypeMap Empty _ = []
flattenTypeMap Split {tyVars, tyCons, appTys} subst
  = tyVars' ++ tyCons' ++ appTys'
  where
```

```
    tyVars' = [(applySubst subst (TV tv), v) | (tv, v) ← mapElems tyVars]
    tyCons' = [(TC tc, v)  | (tc, v) ← mapElems tyCons]
    appTys' = [(TA l r, a) | (l, next) ← flattenTypeMap appTys subst
                           , (r, a) ← flattenTypeMap next subst
                           ]

possiblesInTyVars :: (Monoid a) ⇒ RawType → Subst → Map TyVar a → [(Subst, a)]
possiblesInTyVars t s tyVars
  = [(s `mappend` subst, a)
    | (tv, a) ← mapElems tyVars
    , let Just subst = unify t (applySubst s (TV tv))
    ]
```

4.1.4 The Constructor Graph

As part of the support set discovery phase, IRULAN builds a *Constructor Graph*, which represents the graph of reachable types through the use of case statements on data constructors. The graph is used during the planning phase to guide expression generation to include case expressions. A *ConstructorGraph* can be queried for a type, and it returns identifiers paired with a subset of the constructor graph in non-deterministic finite automata (NFA) form. The NFA edges express a case statement pattern to apply to a previous expression based on the value returned by executing the previous expression. The NFA accepting states represent expressions of the type that was originally looked up.

For example, in Figure 4.6a we present a small module for exploring the states of a *Board* game, Figure 4.6b shows the constructor graph for Program 4.6a.

The roots of this graph are the functions in the support set, in our case *start*, *step* and *searchForBest*. Each root points to nodes representing the ultimate type returned by applying the respective function all of its arguments. An arc between two types *T1* and *T2* is labelled by a constructor pattern match that can be used to obtain a value of type *T2* from one of type *T1*. For example, the arc between [*Board*] and *Board* is annotated by $(x : \_) \to x$ which represents the case statement that takes the head element out of a list of *Board*s built by a cons (:) constructor.

Note, the roots of the graph do not include data constructors, only functions. This is because the graph encodes case expressions over values built up from the roots of the graph. To build a case statement rooted over a data constructor would be wasteful (e.g. there is no point constructing **case** (*foo* : *bar*) **of** (x: _) $\to x$, as this is the same as *foo*).

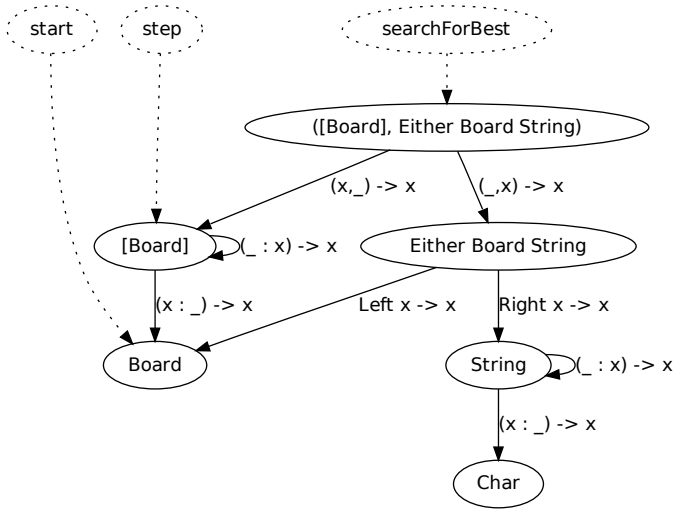
Table 4.1 shows some example expressions built using the constructor graph in Figure 4.6b. Imagine we wanted to use case expressions to build an expression of type *Board*. We could start from the identifier *step*. Assuming

Figure 4.6 A case statement example

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

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

(b) Constructor Graph for the *Board* module



we have another algorithm (for IRULAN this is the *Plan* algorithm, discussed in Section 4.2) that can apply *step* to enough arguments, we can start with an expression of type [*Board*], e.g by building the expression *step start* as shown in the first line of Table 4.1.

If *step start* is successfully evaluated to a WHNF, it could either return an empty list [] or a cons cell (:). In our constructor graph, there are no arcs from [*Board*] corresponding to the [] case (there are no case expressions over [] that yield sub values), so in that situation we have to stop, as shown in the second line of Table 4.1.

The third and fourth lines of Table 4.1 are for when *start step* builds a list with at least one element. Here there is a choice of arc to follow in the graph, either taking the head element of the list (line 3) or the tail (line 4). When taking the head element (via **case** (*step start*) **of** (x: _) → x), the resulting type is *Board* which is what we want. Note that the case statement generated for the next expression is not exhaustive, i.e. there is no alternative for the [] case. This is because we have already analysed the root constructor of *step start* and know that it returns a (:) cell before choosing one of the (:) based arcs in the graph.

If we follow the tail case statement in line 4, then we can execute the resulting case statement, and possibly

Table 4.1 Examples of test expressions with their types and a possible runtime (RT) value. Then based on the type and runtime value, one arc from Figure 4.6b to follow is given (Arc / Id Followed), and what the next expression based on the arc followed would be.

Expression	Type	RT Value	Arc / Id Followed	Next Expression	Next Type
-	-	-	*step*	*step start*	[*Board*]
step start	[*Board*]	[]	-	-	-
step start	[*Board*]	(:)	$(x\colon \_) \to x$	**case** (*step start*) **of** $(x\colon \_) \to x$	*Board*
step start	[*Board*]	(:)	$(\_ : x) \to x$	**case** (*step start*) **of** $(\_ : x) \to x$	[*Board*]
case (*step start*) **of** $(\_ : x) \to x$	[*Board*]	(:)	$(x\colon \_) \to x$	**case** (**case** (*step start*) **of** $(\_ : x) \to x$) **of** $(x\colon \_) \to x$	*Board*
case (*step start*) **of** $(\_ : x) \to x$	[*Board*]	(:)	$(\_ : x) \to x$	**case** (**case** (*step start*) **of** $(\_ : x) \to x$) **of** $(\_ : x) \to x$	[*Board*]

build a larger case statement if it returns a (:)-based value, as shown in lines 5 and 6.

Thus far, we have assumed that the root identifier(s) chosen and the arcs followed will always lead to useful case expressions. We have not yet made precise how we ensure that only suitable root identifiers, and suitable arcs are followed. For example when trying to generate expressions of type *Board*, if we started from the identifier *searchForBest* and built up an expression of type *Either Board String* that returned a value *Right x*, it would be wasteful to follow that arc, as the reachable types from that arc (*String* and *Char*) do not include *Board*.

During the planning phase (§4.2), IRULAN will query the constructor graph when it wants to use case expressions in expressions. The constructor graph uses a *TypeMap* (§4.1.3) to map queried types to nodes in the graph that could be suitable. For example, querying for a type $[a]$ would return the [*Board*] node with a substitution `a/Board`, and the *String* node with the substitution `a/String`. For each of the suitable (target) nodes, the constructor graph will then extract a subset of itself as an NFA where the accepting state is the target node. The subset has the rule that the target node is always reachable from any other node in the graph.

To build this subset, the constructor graph first inverts all arrows in the graph, and finds all reachable nodes from the target node (e.g. in Figure 4.7a the target/accepting node is *Board*). Only the reachable nodes are kept and the arrows are re-reversed (Figure 4.7b). These NFAs are then traversed by the *Plan* (§4.2) to build up case expressions of a particular type.

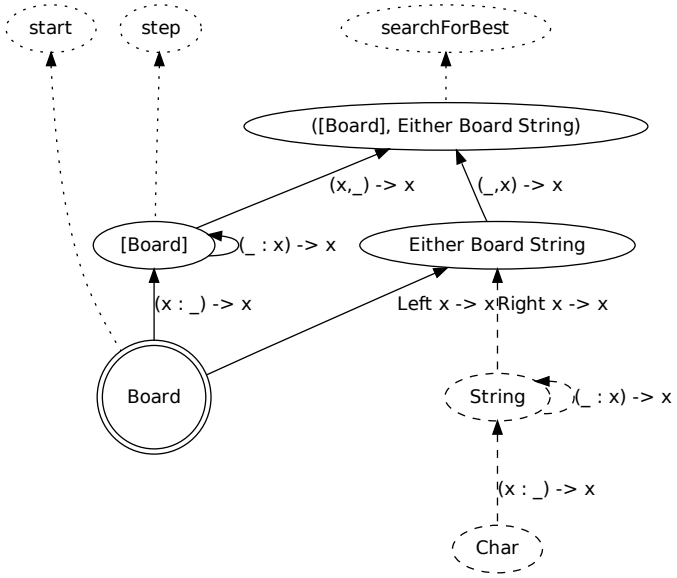
4.2 The Plan

4.2.1 Test Expressions

Using a function to test from the module to be tested as a starting identifier, and the support set as a source of identifiers and constants, IRULAN then generates and executes test expressions.

Figure 4.7 Turning a Constructor Graph into a NFA to build case expressions

(a) Selecting *Board* as the target node,
and reversing all arrows to find inversely-reachable states
(dashed nodes/arrows are unreachable)



(b) Keeping only the reachable states, and then re-reversing the arrows to build
the NFA for *Board*

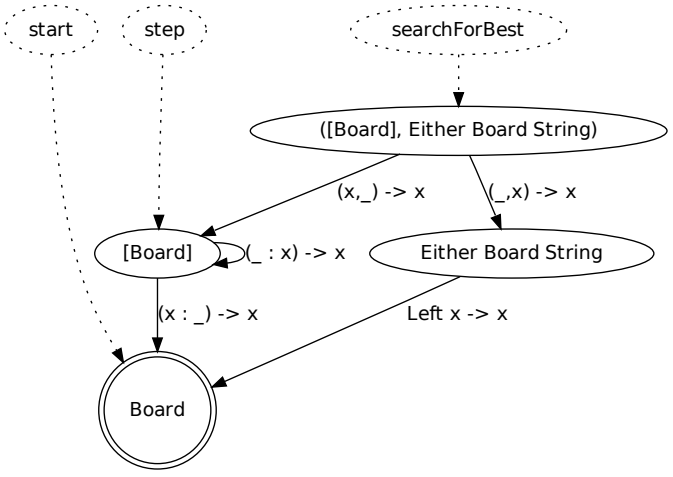


Figure 4.8 The syntax of IRULAN's *test expressions*

```
expr ::= root arg* subst

root ::= identifier
       | constant
       | case expr of Ctr _* x _* -> (x :: type)

arg ::= expr
      | ?i :: type
      | '*' :: tyvar

constant ::= int | char | float | double | integer
```

IRULAN's test expressions (shown in Figure 4.8) are an explicitly typed subset of Haskell expressions, although the type information is usually elided when presenting test expressions to the user. All test expressions, `expr`, start with some `root` applied to zero or more arguments, `arg`. The test expressions also have a substitution, `subst`, that maps free type variables to their instantiated value (if they have one).

The `root`s are either Haskell identifiers (which carry their own type information internally), `constant`s (*Int*s, *Double*s, *Char*s, etc) or `case` expressions. `case` expressions always look for a single element in exactly one single constructor, returning that and recording the (instantiated) type of the resulting variable.

Test expression arguments can recursively be expressions, to-be-refined `?` arguments that carry their index (`i`) and type, or an *unconstrained* variable (`*`), which we discuss further in the Polymorphism part of Section 4.2.3 below.

Table 4.2 Some example expressions and their Haskell representation

Test Expression	Type	As Haskell
`(False :: Bool)` $\emptyset$	Bool	*False*
`(id :: a -> a) (False :: Bool` $\emptyset$`)` $a/Bool$	Bool	*id False*
`case (foo :: Maybe Int` $\emptyset$ `) of` `  Just x -> (x :: Int)` $\emptyset$	Int	**case** *foo* **of** *Just* $x \rightarrow x$

`(fromJust :: Maybe a -> a)`
`  ((snd :: (b,c) -> c)`
`  (foo :: (Int,Maybe Bool)` $\emptyset$ `)`
$b/Int, c/Maybe\ Bool$ `)`
$a/Bool$
	Bool	*fromJust* (*snd foo*)

When presenting test expressions we may elide type information, as IRULAN does to create presented test expressions that are (nearly always) Haskell expressions. Since we assume a rank one type system, Haskell's type inference will be powerful enough to understand applications of identifiers to other sub expressions. The two constructs that do not coincide directly with Haskell are the `?i` argument form (which can be replaced by $\perp$ or *error i*), and `*` (which can be replaced by ()). As a practical issue, occasionally GHC's internal type class dictionaries will manifest themselves as identifiers used by IRULAN, which could also be elided. We show the relationship between some full test expressions and their elided form in Table 4.2.

The grammar for test expressions is designed to make some meaningless expressions impossible to write.

Figure 4.9 The *Plan* data structure

```
data Plan
  = Step { testExpression :: Expr
         , onOk           :: Plan
         , onDataCon      :: Map DataCon Plan
         , onBottom       :: Map BottomId Plan }
  | Split { children      :: [Plan] }
```

For example, expressions cannot start with a `?i` at their root, and a case statement cannot be of the form `case ?i of ...` because the evaluation of such expressions would first force the `?i` (and require it to be refined into a more meaningful expression) before anything else.

4.2.2 The Plan

With a support set in place, IRULAN then builds a *Plan*, a novel, lazy data structure that represents the (possibly infinite) number of ways in which IRULAN could build expressions to test the functions to be tested. Figure 4.9 gives a simplified presentation of the *Plan* data type.

The *Step* constructor specifies a test expression to run, and subsequent *Plan*s to follow based on the result of evaluating the *testExpression* to WHNF. The *Split* constructor represents explicitly non-deterministic choice in the *Plan* (and gives rise to a search space). We will consider both cases by looking at an example *Plan*. *Split* with no children is used to encode a *Plan* that cannot continue as it has no descendants.

We will often talk of a single function being tested, and our *Plan*s will start with a *Step* rooted in testing that function / identifier (initially with no arguments as our algorithm will describe). To test all the functions in a module, the *Plan* for each is made, and then a *Split* is used to combine the list of *Plan*s (one for each function) into a single *Plan* for use later in IRULAN.

Figure 4.11 shows part of the *Plan* dynamically generated by IRULAN while testing the *insert* function in Figure 4.10 (which is the same as Program 3.1 from the overview). The *Plan* consists of a series of *Step*s: each *Step* is denoted by an oval containing the expression to be evaluated. Where there are several ways to generate test data, a diamond is used to represent *Split*s in 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:

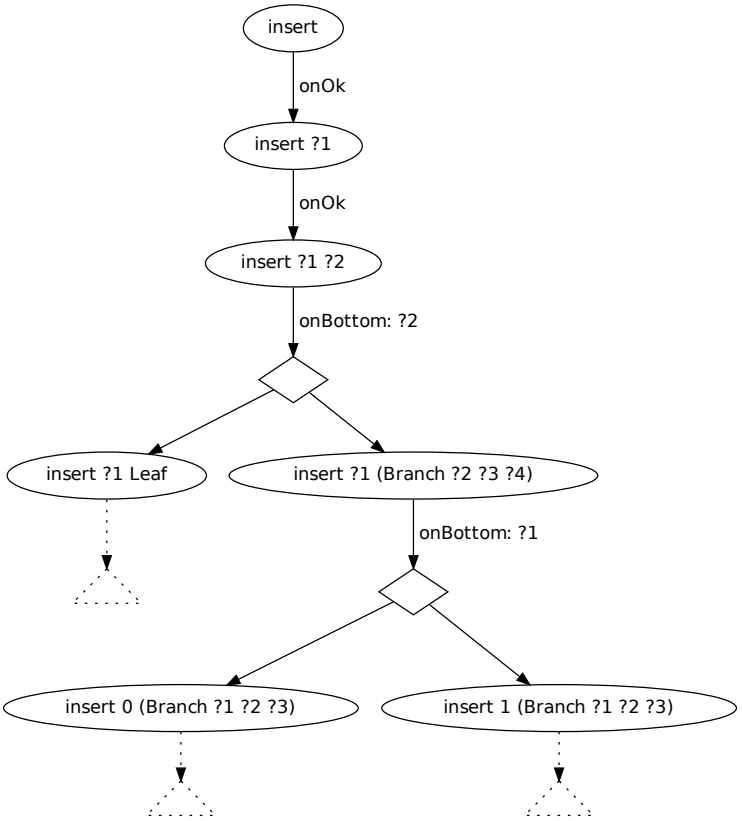
1. `onOk`: This is followed when IRULAN discovers that e successfully ran to WHNF. When this happens, IRULAN is given a *Plan* to follow unconditionally. This new *Plan* either increases the number of arguments passed to e, or, if e is used to build an argument to f, it instantiates that argument to e and continues f's evaluation.

2. `onDataCon`: This case is followed 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

Figure 4.10 Simple Haskell *IntTreeExample* module.

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

Figure 4.11 Part of the *Plan* followed by IRULAN while testing the *insert* function in Program 4.10, which illustrates the `onOk` and `onBottom` cases.



publicly exported, IRULAN will go on to inspect (using case expressions) the arguments passed to d by f. To do so, the *Plan* for inspecting each argument can be retrieved by looking up d in the map from data constructors to *Plan*s.

Since both `onOk` and `onDataCon` can be triggered due to an expression running to WHNF, a non-deterministic choice by a search strategy (§4.4) is used to choose which is followed first.

3. `onBottom`: This case is followed when e requires the evaluation of one of its `?` arguments, which is looked up in the map to find the *Plan* detailing possible ways of instantiating that argument.

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

The first *Step* in the plan of Figure 4.11 tests whether *insert* accepts any of its arguments. This triggers the `onOk` case, because running *insert* with no arguments returns successfully. Following `onOk` 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 `onBottom` case. At this point, IRULAN returns a *Plan* with a non-deterministic choice (denoted in Figure 4.11 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 `onBottom` 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 Section 4.4.

4.2.3 The Plan Algorithm

The core algorithm for generating a *Plan* is fairly simple. We have then built upon this core algorithm to add more features to IRULAN. We proceed as follows:

After introducing and explaining the basic algorithm, we are going to add case statements to destruct a final result. This will enable us to see if the lazy computations inside returned data constructors are hiding exceptions that would otherwise be unobserved. We will then show that some duplicate test expressions can be generated by IRULAN, and how carefully threading a cache through the *Plan* can prevent some of these being generated. Finally we make use of the constructor graph from Section 4.1.4 to provide extra ways of creating arguments to test functions.

The Basic Algorithm

In Figure 4.12 we give a Haskell-like pseudo code for creating a simple *Plan* data structure, ignoring case expressions, and assuming polymorphic substitution is handled correctly by the helper functions.

Figure 4.12 Pseudo Code for creating a simple plan data type

```
type Instantiate = Expr → Plan
gen :: Expr → Type → Instantiate → Plan
gen e targetType instantiate
  = Step { testExpression = e
         , onOk           = onOk
         , onBottom       = onBottom
         }
  where
    onOk
      | typeOf e `unifiesWith` targetType = instantiate e
      | otherwise = gen (addBottomArgumentTo e) targetType instantiate
    onBottom = [?i ↦ Split [gen providerId (typeOf ?i)
                                (λe' → gen ([e'/?i] e) targetType instantiate)
                           | providerId ← querySSIdForType (typeOf ?i)
                           ]
                |?i ← getBottomArguments e
               ]
createPlan :: Identifier → Plan
createPlan x = gen x (resultType x) (\_ → Split [])
```

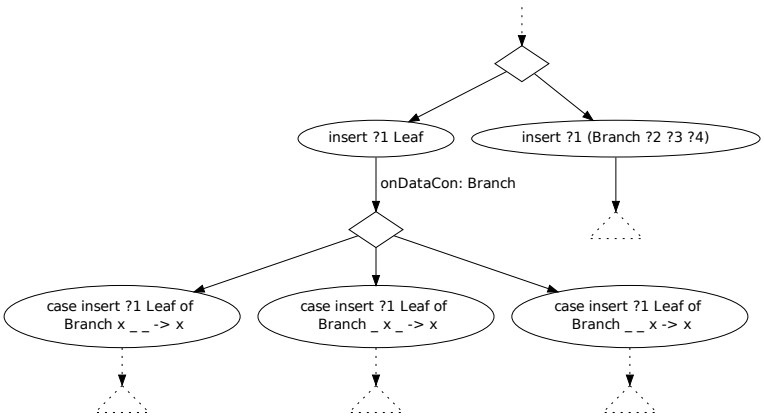
The core generation of the plan takes place in the function *gen*. This function is responsible for making a plan *Step* for a given test expression e. The eventual aim is to create *Step*s that build up an expression of type *targetType*, which will be reached by successfully applying e to zero or more arguments. Once e has been built up to that type a continuation *Plan* is provided through the *instantiate* argument.

The use of the *targetType* means that *gen* can generate higher order functions as test data if necessary, by specifying the higher order type (e.g. $Int \to Bool$) in *targetType*. Without some way of expressing how many arguments to apply e to, the algorithm could only saturate e, which may not always be desired.

The two helper functions *onOk* and *onBottom* that are used in the constructed *Step* contain the logic for the respective following plans.

- In *onOk* we can assume that the original e has successfully run to WHNF. The first guard checks whether e has reached the *targetType*, if so it uses the *instantiate* argument to get the continuing *Plan*. Otherwise, we apply e to a new `?` argument, and recurse into *gen*.
- In *onBottom*, we can assume that e has been run and required the evaluation of a `?` argument. *onBottom* therefore returns a map, where the keys are the `?` arguments in e, and the values are the possible *Plan*s for instantiating that `?` with a more concrete expression. Identifiers that can provide expressions with the same type as the `?` argument are queried from the support set ($querySSIdsForType :: Type \to [Id]$) and for each of them, *gen* is used to recursively build an expression with the type of the respective `?` argument.

Figure 4.13 Part of the *Plan* followed by IRULAN while testing the *insert* function in Program 4.10, illustrating the `onDataCon` case.



The instantiate argument to *gen* is used to replace the `?` argument with the recursively built expression e', which is then passed back into *gen* to test the newly substituted expression.

In order to create a simple plan to test some arbitrary identifier, as in *createPlan*, we use *gen* with the identifier as the initial expression. The *targetType* is the result type of the identifier, so that any arguments needed to make the identifier return the required result type will be created. If the Plan can successfully execute the identifier then there is nothing further to do, hence the empty continuation.

Destructing Returned Values

The first extension to the above scheme is a change that enables case statements to be used to force the resulting value at the end of testing, which can find errors that may otherwise be hidden due to laziness. For example, when testing the *insert* function from Figure 4.10, the *Plan* created by *createPlan* on *insert* would stop after successfully executing *insert* `?1` *Empty* to a WHNF. However we would like the *Plan* to continue in the manner shown in Figure 4.13; i.e. that if *insert* `?1` *Empty* returns a value created by a *Branch* constructor, there are 3 ways to continue testing, by looking at the three children of that *Branch* constructor.

In Figure 4.14 we detail an extended version of the algorithm that will use case expressions where possible to destruct the final value at the end of testing. The new functionality to support this is mostly confined to *createDestructPlan* which is the top level interface, that uses a new *Instantiate* continuation called *tryToDestructPlan* to actually generate the case expressions. However in order to support this new functionality the type of *Instantiate* has had to change which has altered several other parts of the original algorithm.

In the basic algorithm (Figure 4.12), the *Instantiate* type was used to represent the continuation for when an expression successfully ran to WHNF. The *Plan* returned by *Instantiate* would end up in the *onOk* field of some *Step*. However now we need to use case expressions, which means that when an expression runs to WHNF we

Figure 4.14 Pseudo Code for creating a simple plan data type that can destruct values

```
type Instantiate = Expr → (Plan, Map DataCon Plan)
emptyPlan = Split []
gen :: Expr → Type → Instantiate → Plan
gen e targetType instantiate
  = Step { testExpression = e
         , onOk           = onOk
         , onBottom       = onBottom
         , onDataCon      = onDataCon
         }
  where
    (onOk, onDataCon)
      | typeOf e `unifiesWith` targetType = instantiate e
      | otherwise = (gen (addBottomArgumentTo e) targetType instantiate, emptyMap)
    onBottom
      = [?i ↦ Split [gen providerId (typeOf ?i)
                                    (λe' → (gen ([e'/?i] e) targetType instantiate, emptyMap))
                     | providerId ← querySSIdForType (typeOf ?i)
                    ]
        |?i ← getBottomArguments e
       ]
createPlan :: Identifier → Plan
createPlan x = gen x (resultType x) (\_ → (emptyPlan, emptyMap))
createDestructPlan :: Identifier → Plan
createDestructPlan x = gen x (resultType x) tryToDestruct
tryToDestruct :: Instantiate
tryToDestruct e
  | Just dataCons ← querySSForPublicDataCons (typeOf e)
  = (emptyPlan, [dataCon ↦ Split [gen (case e of dataCon x0 .. xn → xi) childType tryToDestruct
                                  | (i, childType) ← children dataCon
                                 ]
                 | dataCon ← dataCons
                 , let n = maxChildIndex dataCon
                 ]
    )
  | hasFunctionType e = (gen e (resultType e) tryToDestruct, emptyMap)
  | otherwise = (emptyPlan, emptyMap)
```

may want to continue with some plan unconditionally (through *onOk*) or we will want to use a case expression over its constructor (through *onDataCon*). Later, when we can use case expressions to produce arguments, it is possible we will want to do both. The changes to the original algorithm to facilitate this are as follows:

- *Instantiate*'s type now returns both a *Plan* intended for the *onOk* field of a *Step*, and also a *Map DataCon Plan* for the *onDataCon* field of the same *Step*.
- To make implementing the rest of this change easier, we have aliased the plan with no elements (*Step* []) to *emptyStep*.
- In *gen*, both *onOk* and *onDataCon* come from *instantiate* when e has been applied to the right number of arguments. When an extra `?` argument needs adding to e, the *onOk* case stays the same as before, and, since there will be no data constructors to consider (as e would have been of function type), an empty map is used for the *onDataCon* map.
- In *onBottom*, the *emptyMap* returned in response to a sub expression e' being found is due to us not (yet) considering how to refine missing arguments with case expressions.

As mentioned above, the actual destruction of result values with case expressions happens in *tryToDestruct*, which is used by *createDestructPlan*.

The behaviour of *tryToDestruct* is determined by the type of the incoming expression e.

- **It is a possibly destructible value (non-function, non-constant) type:** The first guard of *tryToDestruct* encodes this check, $querySSForPublicDataCons :: Type \rightarrow Maybe\ [DataCon]$ will return *Nothing* if the type cannot possibly have any associated data constructors, and *Just dataCons*, where *dataCons* will be a collection of the publicly exported data constructors for the queried type.

  For each of the *dataCons*, we build a *Split* plan, where each element in the plan carries on testing a different element in the respective *dataCon*.
- **It is a function type:** The second guard of *tryToDestruct* would be reachable if a previous case expression extracted a child that had a function type. Since *gen* is designed to add new arguments to expressions, we use *gen* to add the requisite number of arguments before trying to destruct again.
- **It is a type variable, application of type variables, or a constant:** In this case there is nothing we are able to do. Note that it shouldn't be possible to reach this case with a type variable as the type, as it would imply that there is a non-error value with type a for any a.

Removing redundant test expressions

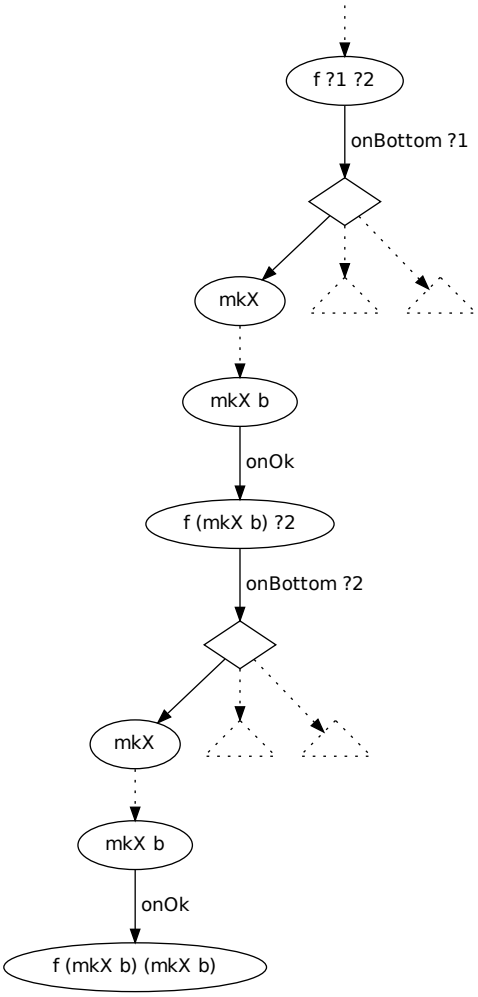
For a particular *Plan*, it is possible that some test expressions may be constructed more than once in different places. Within the *Plan* generation algorithm, it is possible to remove some of this duplication by remembering

Figure 4.15 An example to motivate removing redundant test expressions

(a) An example library

$$f :: X \to X \to A$$
$$mkX :: B \to X$$
$$b :: B$$

(b) A *Plan* for the example library



which expressions have already been generated between the current location and the root, and what branch was followed (i.e. what the result of the expression was) that led to the current location.

For example, consider the excerpt of a module shown in Figure 4.15a, and part of the *Plan* for testing it in Figure 4.15b. During the testing of function f the expression $mkX\ b$ could be used to produce test data for f's first argument. After following the *onOk* arc, it is known that $mkX\ b$ will evaluate to a WHNF. Later, while generating test data for f's second argument, the expression $mkX\ b$ will be re-executed, even though we know that its evaluation will result in a WHNF.

Note that this knowledge only propagates down through the *Plan*, so only descendants of a *Step* know what the outcome of that *Step* was.

To take advantage of this knowledge, a cache mapping expressions to outcomes can be threaded through the algorithm; so for example *gen* and *Instantiate* now accept this cache as an argument. Recursive calls to *gen* or an *Instantiate* argument in *onOk* and *onBottom* add knowledge of the test expression's outcome (reducing to WHNF in *onOk*, or the particular `?` argument in *onBottom*) to the cache. Before *gen* creates a new *Step* for a given test expression e, it consults the cache and if it finds e is already in it, it returns the *Plan* that would be followed given e's known result (e.g returning the *Plan* built by *onOk* as opposed to the *Step* with e in it).

Using case expressions for arguments

The final addition to the basic *Plan* algorithm is the ability to use case expressions in order to generate arguments for functions that need them.

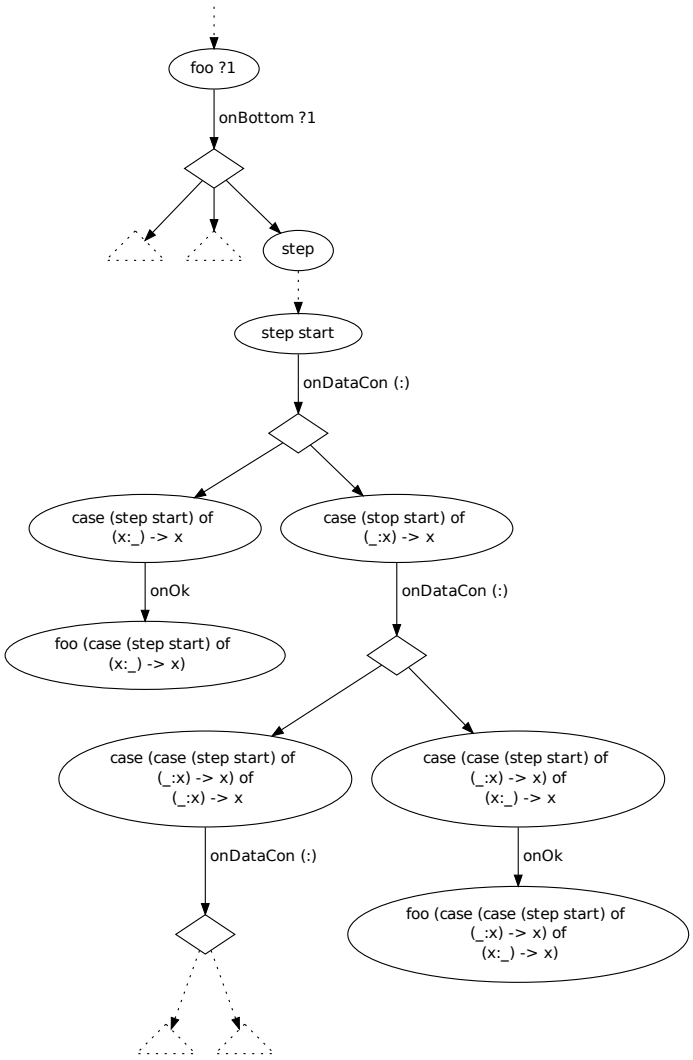
Figure 4.16 The API the Constructor Graph (§4.1.4) presents to the *Plan*

$$
\begin{aligned}
&\mathit{queryConstructorGraph} :: \mathit{Type} \to [(\mathit{Id}, \mathit{NFAState})]\\
&\textbf{data}\ \mathit{NFAState} = \mathit{NFAState}\ \{\ \mathit{accepting} :: \mathit{Bool}\\
&\qquad\qquad\qquad\qquad\quad ,\ \mathit{nextSteps} :: \mathit{Map}\ \mathit{DataCon}\ (\mathit{Map}\ \mathit{Int}\ \mathit{NFAState})\\
&\qquad\qquad\qquad\qquad\quad \}
\end{aligned}
$$

We make use of the Constructor Graph (§4.1.4), through the abstract interface presented in Figure 4.16. The *queryConstructorGraph* function returns a list of *Id*entifiers that (once saturated with arguments) may be wrapped in case expressions to provide the *Type* queried. The associated *NFAState* with each *Id* represents a state in the Constructor Graph. Each state implicitly represents a type and is associated with a test expression; for example the initial states will represent the types of their associated *Id*s saturated with arguments. Accepting states are ones that represent the originally queried type. The transitions between states are in the *nextSteps* field. In order to transition from one state to another, the runtime value of the states' associated expression must be looked up in the *DataCon* part of the *nextSteps* field. The resulting *Int* map gives possible continuations. The *Int*s are the indexes of children in the *DataCon*, and thus each *Int* in that map encodes a case expression over the state's expression that leads to a new state.

In Figure 4.17 we show part of a *Plan* that uses case expressions to produce test arguments. The *Plan* is for

Figure 4.17 Part of a *Plan* that uses case expressions to produce arguments. Based on the Constructor Graph in Figure 4.6b.



testing a function *foo* :: *Board* → *Bool*, and uses the constructor graph from Figure 4.6b. The plan shown for instantiating *foo*'s `?1` argument is rooted at *step*, which is one of the roots of the constructor graph. Once *step* has been saturated with arguments producing the test expression *step start*, the edges in the constructor graph NFA guide *onDataCon* continuations. In this case, the (:) data constructor leads to two new states, either taking the head element of *step start*'s list (through the **case** (*step start*) **of** (x: _) → x), or taking the tail of the list (through the **case** (*step start*) **of** (_ : x) → x).

In the former case, the resulting case expression is of type *Board*, so if it evaluates to a WHNF then the *onOk* *Plan* can be followed, which instantiates *foo*'s `?1` argument to the case expression.

In the latter case, the resulting case expression is of type [*Board*], so if it is a value with a (:) data constructor at the root, then the same choice between taking the head or tail of that list can be made.

Figure 4.18 The additions to the *Plan* algorithm to allow case expressions to produce test arguments

```
gen :: Expr → Type → Instantiate → Plan
gen e targetType instantiate
  ...
  where
    onBottom
      = [?i ↦ Split ([...as before...] ++
                     [gen x (resultType x) (caseGen nfa instantiate)
                      | (x, nfa) ← queryConstructorGraph (typeOf ?i)
                     ])
         |?i ← getBottomArguments e
        ]
caseGen :: NFAState → Instantiate → Instantiate
caseGen nfa instantiate e
  = (accOk, accDC `mapUnion` stepDC)
  where
    (accOk, accDC)
      | accepting nfa = instantiate e
      | otherwise = (emptyPlan, emptyMap)
    stepDC = [dc ↦ Split [gen e' (resultType e') (caseGen nfa' instantiate)
                          | (i, nfa') ← dcArgs
                          , let e' = ⟦ case e of dc x0 .. xn → xi ⟧
                          ]
              | (dc, dcArg) ← nextSteps nfa
              , let n = maxChildIndex dc
              ]
```

In Figure 4.18 we detail the changes to the *Plan* algorithm (as of Figure 4.14) needed to enable case expressions as arguments to functions. We do not consider the threading through of the cache to remove locally duplicate expressions as it adds noise to the presentation.

The first observation is that the need to construct case expression based arguments is triggered by the need to refine a `?` argument to a value. This refinement happens in the *onBottom* helper of *gen*, which is where the first changes are made. Here we extend the possible *Plan*s inside the *Split* with ones that try to use case expressions.

The call to *queryConstructorGraph* will return root identifiers, x and *NFAState*s, *nfa* that can be used to

generate these case expressions. In Figure 4.17, one such x would be the *step* inside the first *Split*. The actual *Plan* to use case expressions is initially generated by *gen*, in order to saturate the x to the right number of arguments. In Figure 4.17, the dotted line between *step* and *step start* indicates the work done by *gen*.

The continuation to *gen* to use once x has been saturated is a new helper function *caseGen*, which uses the *nfa* to build appropriate case expressions, and the original *instantiate* for when it reaches *accepting* states in the *nfa*.

The *caseGen* function has to take two pieces of information into account when creating the *onOk* and *onDataCon* tuple that it returns. The first is whether the current *nfa* state is an *accepting* state, the second is the outgoing edges from the current *nfa* state.

The $accOk :: Plan$ and $accDC :: Map\ DataCon\ Plan$ helper variables are based on the *accepting* state of the *nfa*. When the *nfa* is accepting, they are the result of calling the *instantiate* that uses the case expression built up as an argument. For example in Figure 4.17, the two *onOk* arcs would have passed through *accOk*. In these cases, the *accDC* map would be empty. However if a case expression itself forced the evaluation of a `?` argument then *accDC* would have elements. Consider **case** (*True*:`?1`) **of** $(\_ : x) \rightarrow x$ as a contrived example, then the continuation after instantiating `?1` would have *onDataCon* (and thus *accDC*) mappings for the (:) that the instantiated expression could return. When the *nfa* state is not accepting, these are empty.

The outgoing edges from the current *nfa* state provide entries for the *onDataCon* part of *caseGen*; these are built up in *stepDC*. Each *DataCon*, *dc*, that comes from *nextSteps nfa* may give rise to several *Plan*s, where each of these *Plan*s extracts a different child from *dc* using a case expression. The children to extract (the *i*s) are specified by the keys in $dcArg :: Map\ Int\ NFAState$. Each child also gives rise to a new *NFAState*, nfa', which is used in the recursive use of *gen* with *caseGen* to carry on generating case expressions until the correct type is reached.

One subtlety in the implementation of *caseGen* is that the returned $Map\ DataCon\ Plan$ in the tuple has to combine both *accDC* and *stepDC*. When both *Map*s contain the same key, this means their values (i.e. the *Plan*s) must be combined in a meaning-preserving way. For simplicity this can be done by creating a new *Split* with both sub-*Plan*s as elements.

Polymorphism

The presentation so far has assumed that polymorphic functions are processed correctly. We now attempt to make this notion more precise, and outline some further changes to the algorithm presented that are needed to make this work.

When the support set is queried for (non constructor) identifiers that can provide a type, a lookup backed by a *TypeMap* (§4.1.3) occurs. This lookup will return *Id*entifiers, and also a substitution. This substitution can easily be applied to the identifiers returned by the lookup through the `subst` mapping in the raw syntax for test expressions (Figure 4.8).

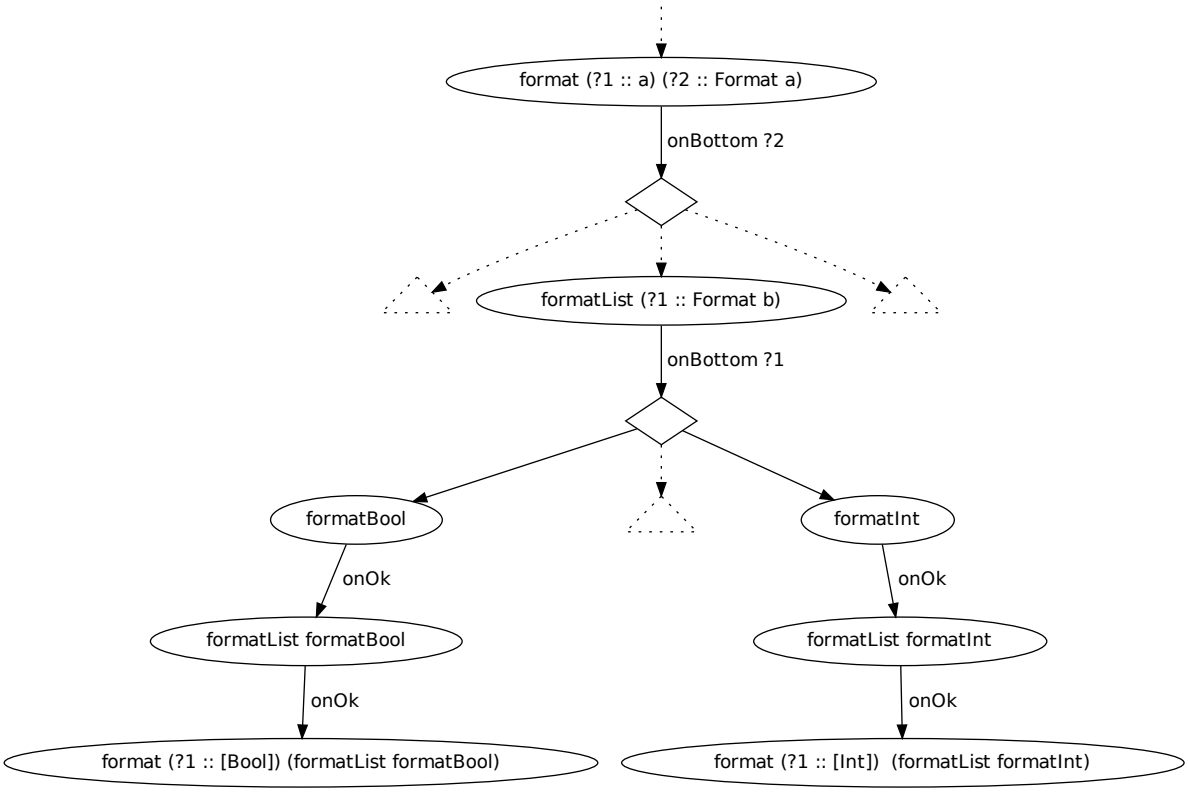
However the algorithm presented can lose some substitutions. This is due to the *Instantiate* parameter being used to allow programming in a continuation passing style. In *gen*, the unification between e and the *targetType* (thus far presented as the function *unifiesWith* which returns a *Bool*) should also return a substitution that explains how the two unify. This substitution should then be passed through *instantiate* with e, so that any new constraints can be applied to the expression *instantiate* builds by replacing a ? argument with e.

Figure 4.19 A module describing simple *Format*ters, to motivate some polymorphic issues

```
data Format a = ...
formatInt :: Format Int
formatBool :: Format Bool
formatList :: Format a → Format [a]
format :: a → Format a → String
```

For an example of one of these new constraints, consider Figure 4.19, which contains a small module for describing *Format*ters. There are two primitive *Format*ters, for formatting *Int*s and *Bool*s, and a composite *Format*ter that given a primitive *Format*ter, can format a list of that primitive. Finally, the function *format* takes a *Format*ter of some type, and a value of that type, and returns the formatted *String* for the value using the formatter provided.

Figure 4.20 An example *Plan* where substitutions should be propagated across *Instantiate*



In Figure 4.20 we detail some parts of the *Plan* from testing the *format* function. We have made explicit the

type of some of the `?` arguments in this *Plan*. Both the left and right branches (*formatInt* or *formatBool*) represent two recursive calls to *gen*, followed by two successive uses of the *instantiate* argument when following the *onOk* arcs in each branch.

What is important to note is that both recursive calls to *gen* in any branch constrain further the type of *format*'s `?1` argument, even though *gen* is being used to generate test data for the `?2` argument. The first call, which produces *formatList* should refine the `?1 :: a` to `?1 :: [b]`. The second calls, which produce `formatBool` and `formatInt` must ground *format*'s argument to `?1 :: [Bool]` and `?1 :: [Int]`, respectively.

Figure 4.21 The alterations to the *Plan* algorithm to allow substitutions of type variables to bubble up through *Instantiate* callbacks

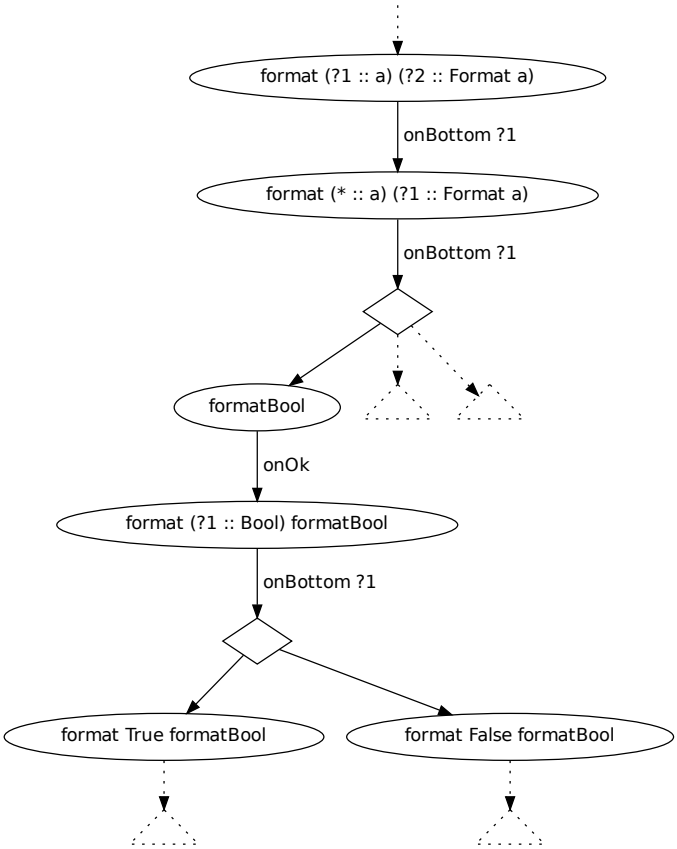
```
type Instantiate = Expr → Subst → (Plan, Map DataCon Plan)
gen :: Expr → Type → Instantiate → Plan
gen e targetType instantiate
    ...
  where
    (onOk, onDataCon)
      | Just σ ← typeOf e `unify` targetType = instantiate e σ
      ...
    onBottom
      = [?i ↦ Split [gen providerId (typeOf ?i)
                                    (λe' σ → (gen (σ ([e'/?i] e)) targetType instantiate, emptyMap))
                     | providerId ← querySSIdForType (typeOf ?i)
                     ] ++ ...
        |?i ← getBottomArguments e
        ]
tryToDestruct :: Instantiate
tryToDestruct e' σ
    ...
  where
    e = σ (e')
caseGen :: NFAState → Instantiate → Instantiate
caseGen nfa instantiate e σ
    ...
  where
    (accOk, accDC)
      | accepting nfa = instantiate e σ
      ...
```

A solution to this problem, as shown in Figure 4.21, is to make *Instantiate* also accept the substitution that the unification in *onOk* should generate, and then apply it to the replaced expression before continuing. There is also a small modification made to *tryToDestruct* and *caseGen* to make them match up with the new type of *Instantiate*.

There is also an optimisation to do with polymorphism that can, in a pathological case, prevent a needless explosion in the branching of the *Plan*.

In Figure 4.22 we show a different part of the *Plan* for the support set from Figure 4.19. We now consider the case when format is strict in its first argument before it is strict in its second. This is unusual because the

Figure 4.22 A *Plan* that introduces unconstrained polymorphic values



first argument has type `forall a . a`, i.e. it could be a value of *any* type. If the support set were queried for identifiers that could provide such a type, it would return all of them. However, the only meaningful thing a function can do with a value of any type is force it to WHNF via *seq*. Because of this, we have added an 'unconstrained type variable' form, `*`, to the syntax of our expressions that represents a `?` argument of a type variable type that has been forced to WHNF. At runtime, the `*` can be replaced by any value that has a WHNF (IRULAN uses unit, `()`).

When substitutions are applied and the type of a `*` should change from a type variable to something else (e.g. after the *onOk* arc in Figure 4.22), then the substitution replaces the `*` with a `?` argument of the new type. This will then cause the `?` argument to be evaluated to WHNF and require instantiating (the threaded cache scheme described above should prune this out). However it will now have a more refined type that is appropriate for using to generate test data.

Figure 4.23 The changes to the algorithm to enable the unconstrained variables (`*`) optimisation

```
gen :: Expr → Type → Instantiate → Plan
gen e targetType instantiate
    ...
  where
    onBottom
        =   [?i ↦ mkBottomArgument ?i
            |?i ← getBottomArguments e
            ]
      where
        mkBottomArgument ?i
          | isTyVar (typeOf ?i) = gen ([*/?i] e) targetType instantiate
          | otherwise = Split ([...] ++ [...])
```

The changes in the algorithm required to enable this optimisation are small, and shown in Figure 4.23. *onBottom* now checks whether the type of the `?` argument it is creating a *Plan* for is a type variable or not. If it is, it replaces that `?` argument with a `*`, otherwise it builds the *Split Plan* it would have before. The removal of `*` arguments happens as part of applying a substitution σ:

$$
\begin{aligned}
&\sigma\ (* :: a)\\
&\quad \mid (\sigma\ a) \not\equiv a = ? :: (\sigma\ a)\\
&\quad \mid otherwise = (* :: a)
\end{aligned}
$$

The complete algorithm (excluding threading of the cache) is shown in Figure 4.24.

4.3 Runtime 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 `?i` argument, or a WHNF value.

Figure 4.24 The Complete Algorithm

```
type Instantiate = Expr → Subst → (Plan, Map DataCon Plan)
emptyPlan = Split []
gen :: Expr → Type → Instantiate → Plan
gen e targetType instantiate
  = Step  {testExpression = e
          , onOk          = onOk
          , onBottom      = onBottom
          , onDataCon     = onDataCon
          }
  where
    (onOk, onDataCon)
      | Just σ ← typeOf e `unify` targetType = instantiate e σ
      | otherwise = (gen (addBottomArgumentTo e) targetType instantiate, emptyMap)
    onBottom
      = [?i ↦ mkBottomArgument ?i
        |?i ← getBottomArguments e
        ]
      where
        mkQArgument ?i
          | isTyVar (typeOf ?i) = gen ([∗/?i] e) targetType instantiate
          | otherwise = Split [gen providerId (typeOf ?i)
                                 (λe' σ → (gen (σ ([e'/?i] e)) targetType instantiate, emptyMap))
                              | providerId ← querySSIdForType (typeOf ?i)
                              ] ++
                              [gen x (resultType x) (caseGen nfa instantiate)
                              | (x, nfa) ← queryConstructorGraph (typeOf ?i)
                              ]
createPlan :: Identifier → Plan
createPlan x = gen x (resultType x) (\_ → (emptyPlan, emptyMap))
createDestructPlan :: Identifier → Plan
createDestructPlan x = gen x (resultType x) tryToDestruct
tryToDestruct :: Instantiate
tryToDestruct e' σ
  | Just dataCons ← querySSForPublicDataCons (typeOf e)
  = (emptyPlan, [dataCon ↦ Split [gen (case e of dataCon x0 .. xn → xi) childType tryToDestruct
                                  | (i, childType) ← children dataCon
                                  ]
                 | dataCon ← dataCons
                 , let n = maxChildIndex dataCon
                 ]
    )
  | hasFunctionType e = (gen e (resultType e) tryToDestruct, emptyMap)
  | otherwise         = (emptyPlan, emptyMap)
  where
    e = σ (e')
caseGen :: NFAState → Instantiate → Instantiate
caseGen nfa instantiate e σ
  = (accOk, accDC `mapUnion` stepDC)
  where
    (accOk, accDC)
      | accepting nfa = instantiate e σ
      | otherwise = (emptyPlan, emptyMap)
    stepDC = [dc ↦ Split [gen e' (resultType e') (caseGen nfa' instantiate)
                          | (i, nfa') ← dcArgs
                          , let e' = ⟦ case e of dc x0 .. xn → xi ⟧
                          ]
              | (dc, dcArg) ← nextSteps nfa
              , let n = maxChildIndex dc
              ]
```

IRULAN pre-compiles test modules to binary form using GHC, so that execution of tests happens on compiled and not interpreted code. Test expressions are built by using GHC to look up identifier symbols as dynamic *HValue*s, and coercing them to be functions to apply them to each other or exceptions where necessary. case expressions are implemented by explicitly inspecting the representation of a *HValue* in the heap, and returning the appropriate child directly. `?i` arguments are compiled by making them *throw* a custom error that specifies which `i` the `?i` argument was.

Once an expression has been converted into an executable entity (which will have type *HValue*), 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.

  If the exploration strategy (§4.4) wants to follow a *Plan* in the *onDataCon* map then the data constructor at the root of the WHNF will be required. The *Engine* can find this by directly inspecting the closure in the heap, in a similar way to the way case expressions are built.

- **An exception is thrown:** This could be due either to the evaluation of a `?` argument, or a general user exception. To distinguish between the two cases, IRULAN inspects the caught exception.

  However, special care must be taken with nested exceptions. For example, we found that the following was a fairly common pattern in some of our benchmarks:

  $$panic :: String \to a$$
  $$panic\ x = error\ (\texttt{"Panic!: "} +\!\!+ x +\!\!+ \texttt{" - please report this bug!"})$$

  If a test expression such as *panic* `?1` is evaluated, the act of printing out the `"Panic!: "` error message from the first exception will cause the strict argument exception (`?1`) to be thrown. To avoid this, the *Engine* will force the full string representation of the error message, and if a nested exception is found, replaces it with `"<nested exception thrown>"`.

- **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.

  Unfortunately the allocations check is overly conservative and does not take into account deallocations due to the garbage collector (there is no mechanism for seeing actual memory usage), as it is possible for an expression to allocate and have the garbage collector chase it, so the net allocations are not increasing. In practice this hasn't been an issue.

4.4 Exploration Strategies

As discussed in Section 4.2, the *Plan* has a series of non-deterministic choice points. The choices are between following an *onOk* or *onDataCon Plan* when an experiment reaches a WHNF, or which child *Plan* to follow inside a *Split*. These choice points naturally give rise to a search space, where some paths through the space will find errors, and others may continue indefinitely without raising any. The choice of how to explore this space therefore is important.

In order to experiment with different types of search strategy, IRULAN uses Haskell's lazy evaluation to decouple exploration of the *Plan* from its construction. Actual execution (or looking up in caches) of test expressions for their runtime results is also abstracted into an API provided by IRULAN which means it is easy to use different search strategies with IRULAN. We discuss this API further in Section 4.5

We have experimented with several kinds of exploration strategies in IRULAN. Most of these strategies require a bound to limit exploration, to e.g. stop the search getting stuck going down a single path.

The natural bound would be the number of *Step* or *Split*s that have been traversed from the root of the *Plan*. There can be pathological *Plan*s that have infinite chains of *Split*s in them so at least one bound must take into account the number of *Split*s seen. However, this type of depth bound is not intuitive. Predicting the "depth" at which certain test expressions will appear is not straightforward, and can be complicated by optimisations (e.g. the threaded through cache).

Standard bounds used in comparable work ([CH00], [RNL08]) are based on the syntactic structure of test expressions; commonly a function of the expression size (e.g. number of terminals or the sum of the terminals and non-terminals in the expression), or the nesting depth of the expression (e.g. the maximum number of non-terminals between the root of the expression and any non-terminal).

In order to allow comparison with related work, some of the search strategies in IRULAN can be parameterised with some of these syntactic properties. The natural depth bound still needs to be used, but if a test expression is seen that exceeds the extra specified bounds then the strategies can prune that particular branch.

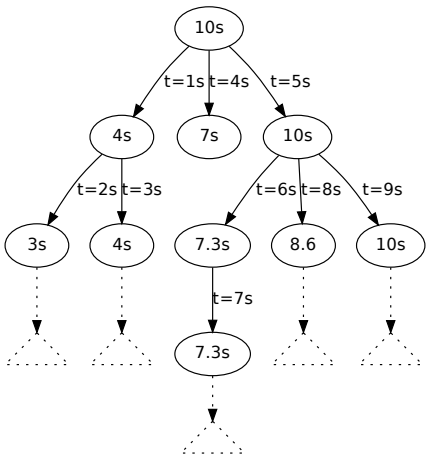
We implemented 4 search strategies in IRULAN:

- **Depth First**. Requires a depth bound to place a limit on the number of *Plan Step*s and *Split*s visited down any branch. This plan can also use the syntactic properties to prune branches early.

  The example in Trace 3.2 uses this strategy.

  The main advantage of this strategy is that it keeps memory consumption low, linear in the depth of the exploration tree. However, for a fixed amount of time, DFS will require a depth bound that is finely tuned. If the depth bound specified is too low then not all of the search space that could be covered will be as it will finish early and spare time will be left; if it is too large then the search space will not be evenly covered (only an early part will be), and there will not be enough time to reach later parts of the space.

Figure 4.25 An example search space using the Depth First, Time Split search strategy.



- **Iterative Deepening Depth First**.

  This strategy explores using repeated use of the depth first strategy. The underlying DFS is first invoked with a depth of 1, and then this depth is iteratively increased in increments of 3. Other, syntactic bounds can also be used, but they are not incremented. This strategy can be told to run for a user-specified amount of time.

  After experimentation (discussed in Section 5.4), we have determined that the iterative deepening strategy appears to be the most effective strategy for IRULAN and so is the strategy used in our evaluation.

- **Depth First, Time Split**.

  This is a variation on the depth first strategy. In addition to the depth and syntactic bounds placed, this strategy also requires a fixed amount of time to run for. Each node in the tree is allocated an amount of time it is allowed to run for, where the immediate children of a node are allocated an equal share of their parents time. If any node is visited after their allocated time is expired then that branch is pruned.

  For an example of this strategy, consider the annotated, abstract search space presented in Figure 4.25. Here nodes represent some computation (e.g. executing an expression) that (for the sake of illustration) takes 1 second to compute. Nodes are annotated with the time by which they must be completed. The algorithm is initialised to run for 10 seconds, and starts at $t = 0$ seconds. All solid nodes are executed at some point in the run, but dashed nodes represent branches that are pruned and not explored due to the time limit for that node being expired. The node edges are annotated with the time at which the algorithm explores that edge, by following the increasing edge times it should be possible to see that the algorithm proceeds in a left-to-right depth first manner. One interesting thing to note is that any "spare" time means yet to be explored branches get more time to be explored (biasing against branches that have already been visited). For example, the middle first child (labelled to finish by 7 seconds) actually finishes being fully explored at $t = 5$ seconds as it has no children. The third child of the root then gets its allocated 3 seconds, plus the left over 2 seconds from the middle, while the first child of the root has

already been explored and gains no extra time.

The ideas behind this technique are not novel (time is being used here as a *fuel* to limit execution down some branches, as an alternative measurement of depth [CGH00]), however it is an idea held in the folklore of the speculative and model checking programming communities and not deliberately described in the research literature.

- **Random Restarting** this is a random strategy that starts at the root of the *Plan* and follows a random path down as far as it can go (or until it hits a bound). It then restarts back at the root, but the *Plan* is reset first so it is possible (but unlikely) that it will follow the same route through the *Plan* again. The resetting is done to stop the strategy forcing the entire *Plan*, which would require large amounts of memory for any large search space.

We compare the effectiveness of the different search strategies in Section 5.4.

4.5 Caching

One important feature of a purely functional programming language like Haskell is that given the same arguments, a function should always return the same result. This property means that executing the same test expression (which consist of a root function applied to some arguments) multiple times should always do the same thing.

We have already seen one application of applying this optimisation in-line in the branches of the *Plan*, using a threaded cache to skip execution of expressions when their result will already have been known. However, IRULAN will often generate the same test expressions in multiple different branches. The *Plan* itself cannot optimise away this case, as the search strategy will dictate which branch is explored first, and thus which test expression will be a duplicate of another.

However, at runtime, (GHC) Haskell's implementation does not really use expressions, but values, where multiple different expressions can map to the same value. This means that test data created using two different test expressions could yield identical values and thus the function being tested using the test data would be tested in an identical way twice.

For a contrived example of this, consider Figure 4.26, which contains a simple model of people with names. *processName* is the function we want to test. Assume that *processName* is strict in its first argument, i.e. we need to generate test data of type *Name*. We could use *johnsName*, and then test *processName johnsName*. We could also use *getName john* and then test *processName* (*getName john*). However both *getName john* and *johnsName* would have exactly the same runtime value, so the two tests of *processName* will behave identically. It is possible to see whether two values are the same by seeing if both values are pointers to the same closure in the runtime heap, which is a feature encapsulated by GHCs *StableName* API.

Figure 4.26 Caching example

```
module Person where
data Name = ...
data Person = Person Name
getName :: Person → Name
getName (Person n) = n
johnsName :: Name
john :: Person
john = Person johnsName
processName :: Name → String
```

This suggests an optimisation such that if *johnsName* has been executed, and when *getName john* is executed we detect it has the same runtime value as *johnsName*, we can then prune the entire branch after *johnsName*, as it will behave identically to the branch following *johnsName*.

We experimented with adding a cache to the runtime API. The cache is tied to the function that actually runs a test expression, to keep it apart from the traversal scheme in use. In Figure 4.27 we present a Haskell-like pseudo code of the runtime component.

There is a primitive for actually evaluating (compiling and executing) an expression, *evalExpression*, which either returns *EvalOk* if the expression has a WHNF, or *EvalException* (with the exception) if the expression throws an exception. If the expression reaches a `?` argument then the exception will reflect that, similarly the exception will also encode out of time or exceeded allocation limits. In the *EvalOk* case, the *RuntimeValue* contains both the dynamic value created (the *HValue*), and a means of doing a pointer equality check with other *HValue*s via the *StableName*.

The function that interacts with the cache and uses *evalExpression* is *runExpression*. It converts an *Expression* into one that has been run at some point (either immediately or, if found in the cache, at some time in the past). Both immediately run (*NewResult*s) or previously run (*CachedResult*) have the result of running the experiment, and, if the experiment had a WHNF the *RuntimeValue* corresponding to the result.

The result of running an expression is richer than the simple *EvalResult*. The *Ok* case is for a successful WHNF, *Errors* for when a runtime error (or time-out / memory allocation limit) has occurred and *Bottoms* for when a `?` argument was reached. *Property* is for when the test expression represents a Haskell property that evaluated to a *Bool*. The *Bool* argument is whether the property returned *True* or *False* (passed or failed respectively). The other case, *Redundant* is for use with the cache. If the runtime value of the expression is (or was) detected to be the same value as that of some other expressions, then the result is *Redundant*, and the known list of expressions it was equivalent to is returned.

We assume the cache itself is accessible through the following abstract interface:

```
lookupExpressionResult :: Expression → IO (Maybe (ExpressionResult, Maybe RuntimeValue))
storeExpressionResult  :: Expression → ExpressionResult → IO ()
storeExpressionHValue :: Expression → RuntimeValue → IO (Maybe [Expression])
```

Figure 4.27 Outline of the use of the cache in the Runtime API

```
evalExpression :: Expression → IO EvalResult

data EvalResult
  = EvalOk RuntimeValue
  | EvalException SomeException

data RuntimeValue = RV { toHValue :: HValue
                       , toStableName :: (StableName HValue)
                       }

runExpression :: Expression → IO RanExpression

data RanExpression
  = NewResult    { getExpressionResult :: ExpressionResult
                 , expressionRTValue :: (Maybe RuntimeValue)
                 }
  | CachedResult { getExpressionResult :: ExpressionResult
                 , expressionRTValue :: (Maybe RuntimeValue)
                 }

data ExpressionResult
  = Redundant [Expression]
  | Ok
  | Errors     SomeException
  | Bottoms    BottomId
  | Property   Bool

runExpression expression = do
  mExistingResult ← lookupExpressionResult expression
  case mExistingResult of
    Just (existingResult, existingRTV) → do
      return $ CachedResult existingResult existingRTV
    Nothing → do
      evalResult ← evalExpression expression
      let storeAndReturn expResult mRtVal = do
        storeExpressionResult expression expResult
        return (NewResult expResult mRtVal)
      case evalResult of
        EvalException (asBottomArgument → Just i) → storeAndReturn (Bottoms i) Nothing
        EvalException e → storeAndReturn (Errors e) Nothing
        EvalOk rtVal → do
          if expressionIsProperty expression
            then storeAndReturn (Property (cast (toHValue rtVal))) (Just rtVal)
            else do
              mEquivalent ← storeExpressionHValue expression rtVal
              case mEquivalent of
                Just equivExprs → return (NewResult (Redundant equivExprs) (Just rtVal))
                Nothing → return (NewResult Ok (Just rtVal))
```

lookupExpressionResult takes an *Expression* and, if the *Expression* is already in the cache then its result (and if it had a WHNF, its runtime value) is returned, otherwise *Nothing* is returned. There are two ways of storing a result, depending on whether the expression has a useful runtime value or not. Those that don't (errors, `?` arguments or properties that returned either *True* or *False*) just associate the expression with its result in the cache (*storeExpressionResult*). However those expressions that do have a runtime value (i.e. the *ExpressionResult* for them is *Ok*) can attempt to associate the expression with *Ok* and its *RuntimeValue* using *storeExpressionHValue*. *storeExpressionHValue* also checks to see if other expressions have got the same runtime value, and if they do, it associates that expression with the others, and returns *Just* the other expressions that also gave that value.

Returning to Figure 4.27, the implementation of *runExpression* first sees if the expression is known to the cache. If it is, then a *CachedResult* can just be returned. Otherwise, the expression needs to be executed using *evalExpression*. The result of evaluating the expression, *evalResult*, will need to be turned into the richer *ExpressionResult*. In addition, the cache will need updating with the new mapping from *expression* to the *ExpressionResult*. Since many cases of this conversion will involve the same logic of updating the cache (without storing a WHNF value) and returning a *NewResult*, we encapsulate this into the helper function *storeAndReturn*.

The analysis of *evalResult* then establishes if an exception thrown represented a `?` argument (via *asBottomArgument*), or was a general exception. If the expression did evaluate to a WHNF, then we check if it represents testing a property and *storeAndReturn* if so in that case. We do not worry about storing the value *True* or *False* as test expressions of type *Bool* will always be formed using their constructors. Finally, if the evaluation of *expression* led to a WHNF that didn't represent a property being tested, then we are in the interesting case. We attempt to associate the expression with its runtime value in the cache, and see if any equivalent expressions are known for that value. If there are, then the test expression is *Redundant*, otherwise it is *Ok*.

IRULAN provides three different cache implementations.

- **No Cache** This implementation has no cache at all, so all lookups return *Nothing* and storing is a no-op.
- **Unlimited Cache** This implementation is based around three maps. The first uses a trie structure to map expressions that don't have an associated value to their *ExpressionResult*s. The second uses a trie to map expressions that do have an associated value to their *RuntimeValue*s. The third maps *RuntimeValue*s to the list of *Expression*s that share that value. *lookupExpressionResult* consults the first and second maps to see if the expression is known (expressions that have an associated value are implicitly *Ok* and don't need to be stored in the first map). *storeExpressionResult* only needs to update the first map. *storeExpressionHValue* has to perform a lookup in the third map, and possibly update the second and third map. There are no limits imposed on the sizes of the maps, so this cache will consume memory as testing proceeds.
- **FIFO Cache** This implementation is a variation of the unlimited cache that also stores a queue of expressions that have been stored in the cache. Once the queue reaches a certain size, expressions at the

start of the queue are popped off and removed as new ones are stored and pushed on, keeping the number of expressions in the cache a fixed constant number.

There is another optimisation that a cache of runtime values enables. The *Plan* algorithm will evaluate an argument of a function to a WHNF before applying it to the function. The *Plan* arranges this by running the expression e to a WHNF, and then applies e to some function f, building the and executing the composite expression $f\ e$. If e was successfully executed and stored in the cache, then during the compilation and execution of $f\ e$, instead of compiling the expression e, IRULAN can bind the runtime value of e to a variable, x, and compile and run $f\ x$.

In practice, we found that in general the overheads of looking up and storing expressions in a cache outweigh the cost of just executing them again. We discuss this point further in our evaluation of the cache in Section 5.3.

4.6 Discussion

In this chapter we have outlined the core algorithms and data structures that underpin the implementation of IRULAN. Many of these algorithms have gone through several iterations and evolutions as the work has progressed, and there are alternative approaches to some of the problems we have tackled in the existing research literature.

The early iterations of this work did not consider case expressions in test expressions and assumed the type system was monomorphic, and so the support set was principally about building up a database of functions and providing a query function that could find all those in the database that matched with the target type. This database was originally implemented as a list of identifiers, which meant that lookup took time linear in the length of the list (as each identifier would have all the types it provided checked to see if any matched with the queried type). We implemented the *TypeMap* in order to improve the complexity of the lookup function to something that was hopefully parameterised by the size of the type being looked up instead of the size of the search space.

IRULAN's decision to build a polymorphic support set, and then test with polymorphic data, (only instantiating types to monomorphic ones when necessary) is a novelty over the existing Haskell testing tools. The decision to support this flexibility led to the invention of the specialised lookup algorithm for the *TypeMap*.

The algorithm the *TypeMap* represents is a very special case of a more general line of research ([AP98], [Rit89], [RT89]) into algorithms for looking up functions similar to a particular type (for e.g. documentation assistants for libraries such as the Haskell API search engine Hoogle [Mit08]). In [Rit89] the author is interested in finding functions that have a type that is isomorphic (up to Currying and argument swapping) to the one queried. For example, you may want a combinator to fold over a list, and know that it should have a type like $((a, b) \to b) \to [a] \to b \to b$[3]. Querying that type should find, for example $foldl :: (b \to a \to b) \to b \to [a] \to b$. The

[3]The author of [Rit89] notes that in 5 different functional languages, there are 10 different names for fold functions, and five different (but isomorphic) types given to them.

approach to tackle this is to rewrite types to a normal form, where types with equal normal forms are isomorphic up to Currying and argument swapping. However the approach outlined does not deal with polymorphic type variables efficiently, and so the authors resulting query function is implemented as a linear scan over the database of normalised types. We note that it could be interesting to add a preprocessing step to IRULAN to normalise types before they are added to the *TypeMap*, and to then use the trie lookup algorithm to find unifying types. This means that a little bit of extra work would be needed to make the APIs of requested and found types match up (e.g. if IRULAN queries for a type $Int \to Bool \to Char$ to use as a higher order function, and lookup returns an identifier of type $(Int, Bool) \to Char$, then that identifier would need to be wrapped in *curry* to make it usable as a result of the query).

It would also be interesting to look quantitatively at the support sets, and the types of queries that happen against the *TypeMap* in practice, in order to guide other optimisations. In [RT89], the authors suggest several ideas for this more general type of query, and quantitatively look at a corpus of identifiers and their types to see what kind of optimisations make the most sense given the properties of the corpus.

To the best of our knowledge, making use of case expressions in Haskell to generate test data is novel to this work. The state machine representation for generating case expressions in the constructor graph is analogous (and was inspired by) the *path graphs* representation used in [CGE08]. The authors use their path graphs to represent a potentially infinite set of field access expressions in an object oriented setting (in order to infer locks to protect them). Our use is similar, the state machine represents the potentially infinite set of case expressions (which are similar to field accesses) that can build expressions of certain types. We believe we are the first to apply this idea in a functional setting for generating test data.

There are also some optimisation opportunities that would be interesting to explore in the constructor graph, particularly if there is a depth bound being followed by the search strategy using it. For example, consider the traversal of the reversed constructor graph to find all reachable identifiers. During that traversal, some paths from the target type to a root identifier may have a length longer than the current depth bound used by the search strategy. If the depth information was available, these paths (and thus some test expressions) could be pruned. Also, if edges in the returned NFA also remembered the shortest length between them and the target node, search strategies would know not to follow certain edges as they create expressions that will never be used for their intended purpose.

The decision to implement the *Plan* as a lazy data structure is one that has created several benefits, but also several problems. Laziness allows the *Plan* building algorithm to just express the search space naturally in the data structure, and not have to worry about callbacks to another library or threading of the current expression state through, which would have been alternative implementation approaches. The full implementation in IRULAN is essentially the algorithm presented, but the type of functions like *gen* uses a *Reader* monad to pass around the support set (which we assumed was a static constant during the presentation) and thread through the expression removing cache.

The laziness of the *Plan* did cause some issues. The search strategies have to be careful to not hold onto the absolute root of the current *Plan* for the duration of their run, otherwise the garbage collector would not be

able to collect visited nodes (as they would still be reachable from the root), and the explored structure would quickly consume all memory. For depth first search (and time split), this is straightforward to arrange, after a test expression has been run, the possible children that follow are pushed onto a work queue. This means the current *Step* can be forgotten and reclaimed by the garbage collector. The next *Plan* to look at is then popped off the work queue. However, the iterative deepening and random strategies require being able to restart exploration from the root of the *Plan*. At the implementation level, this required some rather awkward code to give these strategies a function that could produce new (identical) *Plan*s when applied to an argument. It was awkward as GHC would often optimise this function to remember the *Plan* if there wasn't an artificial dependency on the argument introduced to produce it.

Comparable tools, such as QuickCheck [CH00], SmallCheck, Lazy SmallCheck [RNL08] tie the search strategy into the generation of test data and execution. For example, in QuickCheck the specification of test data includes direction about the relative distribution of data to use, and evaluating the generator yields the test data in the appropriate random order. SmallCheck and Lazy SmallCheck are implemented using a depth first traversal, and iterative deepening comes from re-invoking the main algorithm with increasing limits. EasyCheck [CF08], a lightweight testing application for the functional logic programming language Curry does make the space of test data explicit, through the use of a primitive in the language that lazily reifies all the possible values a value could take in the tree. This abstraction means the authors can explore two variations of search strategy, using level diagonalisation and branch randomisation.

IRULAN did briefly contain two other search strategies; a breadth first traversal and a random strategy that remembered test expressions it had executed and thus didn't need to execute them again. Both strategies would quickly exhaust the available heap memory as they fully explored the search space, remembering so many expressions and their results, and thus were not practical.

During implementation of the *Plan*, there were some design decisions made in IRULAN that, while independent of the core algorithm, did have some impact on the implementation. For example, the `?` arguments in IRULAN are implemented as thrown exceptions that contain a unique id. In Lazy SmallCheck a similar approach is used, and the unique identifier corresponds to a path through the expression that is used to identify the `?` and refine it. This means that, in Lazy SmallCheck, `?` arguments in the original expression do not need to be renamed, and new `?` arguments in the term being substituted in have to have the old path as a prefix. In IRULAN, there is a supply of `?` argument identifiers for each branch of the *Plan* that gets incremented whenever a new `?` argument is created (as part of *addBottomArgument*). Substituting a new sub-expression for a `?` argument in an old term does require traversing the old term to find the instance of the `?` argument, but the relative size of these terms is small. For presentation purposes, we normalise the `?` arguments shown to always start at index 0. The structural path approach adopted by Lazy SmallCheck would be complicated by our approach of testing the instantiation for the `?` value explicitly before substituting it in, as the paths would change depending on the context of the parent expression.

There is a small optimisation that IRULAN makes (but we have not made explicit here) with respect to `?` arguments, in that constructors e.g. $Branch :: IntTree \to Int \to IntTree \to IntTree$ are known to be lazy in their

arguments, so there is no point testing the chain of *Branch*, *Branch* `?1` and *Branch* `?1 ?2`, as they will always retrun a WHNF. So we can (if we want to make test data of type *IntTree*) just jump straight to *Branch* `?1 ?2 ?3`. However, for arbitrary functions, the decision not to saturate them with `?` arguments means we retain a tiny amount of extra precision when reporting the causes of errors.

The implementation of IRULAN is around 6000 lines of Haskell (including comments and white space) spread between 41 modules. There are roughly a further 4000 lines of unit and integration tests to ensure the correctness of IRULAN. This does not include size of the packages from the wider Haskell community that IRULAN makes use of, but that were critical for ease of development.

The code base may seem surprisingly large given the modest several hundred lines of code used to implement the Check family of tools. However IRULAN has made explicit many data structures (for example the *Plan*) which other tools leave implicit, to allow different exploration strategies to be easily implemented. In addition, there are many experiments within IRULAN, for example caching (again with different strategies) that can be enabled and disabled, which all require extra code to manage. IRULAN has primarily been a research platform, and while it is a completely usable tool, it would certainly be possible to streamline some of the implementation to make it more modest in size.

Chapter 5

Experimental Evaluation

In this chapter, we evaluate IRULAN's effectiveness by running it on various benchmarks.

We start with two large experiments. The first looks at the nofib benchmark suite, and shows what kind of errors IRULAN can find "for free". The second experiment compares IRULAN with the existing property checking tools Lazy SmallCheck, SmallCheck and QuickCheck using the benchmark from [RNL08].

Following the large experiments are two smaller tests to explain and justify some of the configuration choices made in the larger ones. The first focuses on the performance of IRULAN's cache, to motivate why it is by default disabled. We then compare the different search strategies, showing how important depth bound selection can be for depth first search, and motivate why we selected iterative deepening as our search strategy for the larger experiments.

Finally we discuss some case studies drawn from using IRULAN's regression testing functionality and show that IRULAN has been used to find real bugs in third party libraries.

5.1 Nofib

Haskell has an established set of benchmarking programs, called *nofib* [Par93]. nofib consists of three suites of increasing complexity: *imaginary*, *spectral* and *real*:

- The **imaginary** suite consists of Haskell programs that represent pathological cases designed to stress compilers (and their writers). These are mostly tiny Haskell programs with only a *main* function exported.
- The **spectral** suite, consisting of the algorithmic cores of real programs, such as a chess end-game solver and a minimal expert system implementation.
- The **real** suite, consisting of real Haskell programs. This includes implementations of a Prolog interpreter, a grep utility and an LZW compression tool.

Table 5.1 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 - 10000:	19	4
10000 - 100000:	2	1

Irulan is designed to test the API of pure library code. However, some of the nofib benchmark programs export only a *main* function of type *IO()* (i.e., they are programs with no library component). Irulan cannot perform any useful testing on these programs, as values of type *IO ()* do not have a WHNF that (typically) involves any real computation taking place. We have excluded these programs from our experiments. These programs include the entire imaginary suite (which consists entirely of tiny programs exporting only a *main* function), 23 programs from the spectral suite, and 3 programs from the real suite. In addition, we have created and use filtered versions of the real and spectral suites that do not include modules which only export a *main :: IO()*.

While it would have been possible to alter the source programs to export their library component as-well as *main*, we wanted to investigate Irulan running on "real sources" as much as possible. Our simple filtering rule means we can hopefully focus more on library than application code. Unfortunately, as we will discuss, there are some cases where programs export *main* and its helper functions that are also of an *IO* () type. Another limitation that affects coverage is Irulan's inability to test type class instance declarations. Many modules declare, for example, *Eq* instances for their data types. Some of these declarations may get exercised during execution as they are relied upon by the library code, however some do not. While we could have edited out such unused instances, again we wanted to see how Irulan performs on "real" code.

After this filtering, we are left with a total of 51 programs, 24 in the spectral suite and 27 in the real suite. To work around some bugs in the HPC tool chain which we use to report code coverage information, we had to convert the nofib `.lhs` files to `.hs` files, and remove some non-Unicode encoded comments from some files. We made no other changes to the source code of the nofib programs beyond this.

Table 5.1 shows the approximate sizes of the 51 programs tested in these suites—in terms of number of expressions, as reported by HPC [GR07]—after filtering out the modules that only export a *main :: IO()* function. [1] In Section 5.1.1, 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.

[1] 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.

For both suites we ran with the use of case expressions enabled and disabled, where each module in each program was run for 1, 10, 60, and 300 seconds. We focus on just a few of these runs here, but present the full graphs of these results in Appendix A.

5.1.1 Coverage Results

Figure 5.1 Code coverage as a percentage of expressions executed for the spectral suite, showing the effect of case expressions

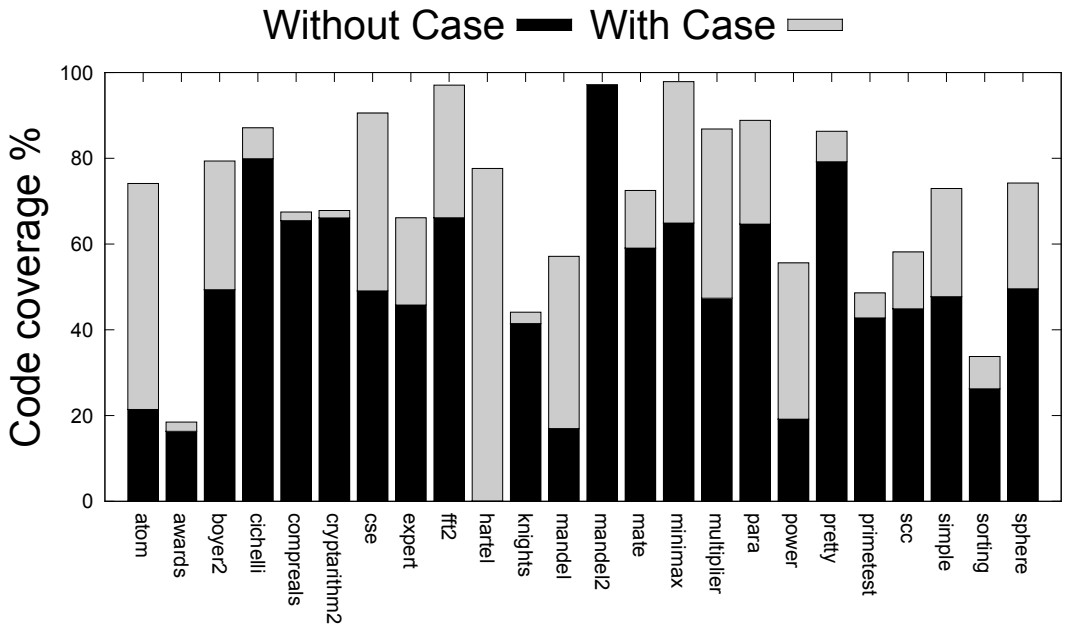


Figure 5.1 gives the code coverage for the spectral suite, where each module in each program was tested for 5 minutes using iterative deepening. The average coverage per program achieved by IRULAN is 70.83%, with a minimum of 18.48% (for `awards`) and a maximum of 97.86% (for `minimax`).

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 *Int* arguments, where our constants (-1, 0 and 1) are enough to trigger the different conditional cases. IRULAN completely explores every exported function in `fft2`, the only unexecuted code being an unexported and unreachable function that could be removed as dead code.

There are also cases where IRULAN does not achieve such high coverage. For example, the `awards` benchmark features a quicksort library that needs polymorphic values of type a with either an *Ord* a type class instance or higher order function of type $a \to a \to Bool$ to achieve better coverage. Unfortunately in this case IRULAN's support set does not contain any such values. Although IRULAN's support set was pre-seeded with suitable constants to use as the value a, (e.g. *Int*s) by default it does not include any type class instances for these constants. Currently for IRULAN to find the type class instance, the algorithm building up the support set would need to explore the module declaring it as part of chasing some other constraint. The same applies for finding the higher order function. Note that the user could explicitly add to IRULAN's command line a module that contains suitable identifiers, however for this experiment we wanted to see what IRULAN could do without

tuning it on a per-program basis.

Figure 5.1 also shows the effect that generating case expressions has on coverage. With case expressions 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 expressions prevents testing almost entirely: e.g., for `hartel` IRULAN achieves only 0.09% coverage, compared to 77.64% when case expressions are used. This is due to `hartel` consisting of constant definitions which include large data structures containing lists of values. Without case expressions, none of these composite values are decomposed. In addition, the increase in code coverage due to using case expressions is also matched by the discovery of more errors, as we will discuss in Section 5.1.2.

Figure 5.2 Code coverage as a percentage of expressions executed for the real suite, showing the effect of a longer runtime

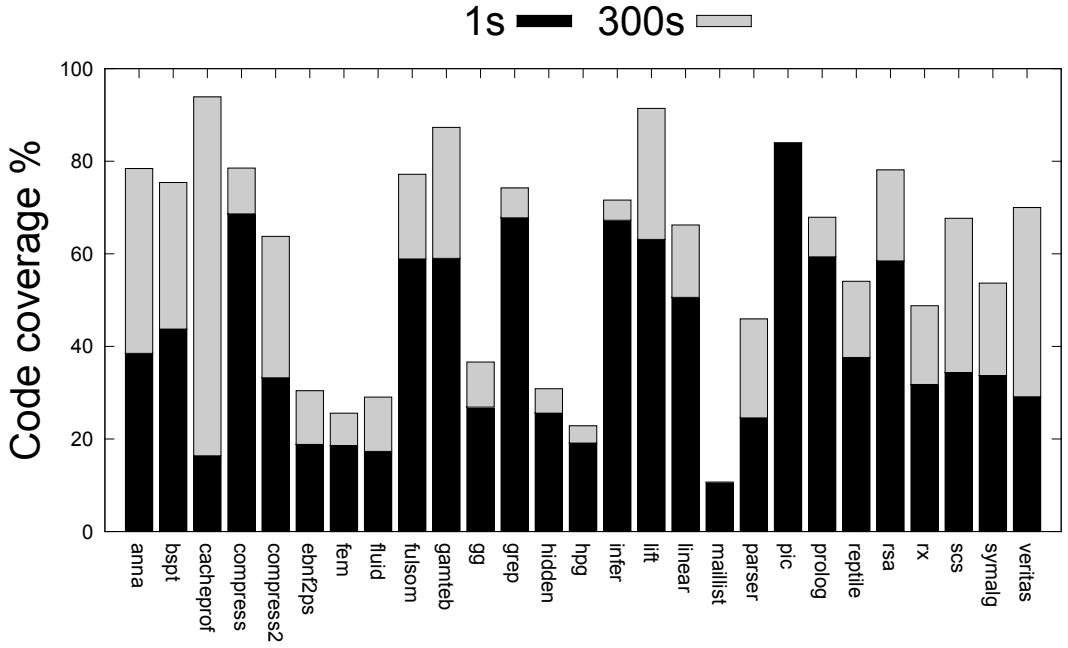


Figure 5.2 gives the code coverage for the real suite after 1 second and 5 minutes of testing of each module in each program. The average coverage per program achieved by IRULAN after 5 minutes is 59.78%, with a minimum of 10.65% (for `maillist`) and maximum of 93.91% (for `cacheprof`). The `maillist` 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 one second achieves useful results, getting two thirds the coverage achieved after five minutes (40.62%). After one minute of testing each module, the average coverage is much closer to the five minute result, reaching 57.10%. This indicates that the coverage results are converging, and smaller gains will be achieved from even longer runs. In general it would not be possible to gain 100% coverage on these suites, as there are some language artefacts IRULAN cannot explore. For example unexported and unreferenced functions that should be removed as dead code, type class instances that are exported but not used and *IO* based helper functions.

5.1.2 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 try 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 expression 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 [APJE09]; looking for more precise ways of identifying errors would be interesting future work.)

The use of case expressions allows IRULAN to discover errors that would otherwise be left undetected. Of the 37 unique locations mentioned above, 7 of them were identified only by expressions with case expressions 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 led 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 the precondition 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. While this error does involve a precondition violation, it also demonstrates a difficulty with working with Haskell, where laziness often causes errors to manifest themselves far away from their root cause. Note that the error message references a function (*empty*) that is not mentioned in the test expression and comes from a different source file to the one being tested. If *searchTree* did check its precondition and throw an exception, IRULAN would still report it. However in this case, it would be easy to see if the exception accurately and helpfully described what went wrong and to filter it out of future reports.

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:

$$
\begin{aligned}
&\mathit{max\_list} :: [\mathit{Double}] \to \mathit{Double} \\
&\mathit{max\_list}\ \mathit{list} \\
&\quad = \mathit{reduce\_list}\ \mathit{max}\ (\mathit{head}\ (\mathit{list} :: [\mathit{Double}]))\ \mathit{list}
\end{aligned}
$$

As an additional more complicated example, IRULAN found a similar error in the `expert` program from the spectral suite:

$$
\begin{aligned}
&\textbf{data}\ \mathit{Phrase} = \mathit{Term}\ \mathit{String}\ [\mathit{Phrase}] \mid ... \\
&\mathit{goal}\ \mathit{ws} \mid ... = \mathit{relation}\ \mathit{ws} \\
&\mathit{relation}\ \mathit{ws} = \mathit{split}\ \mathit{ws}\ \mathit{noun}\ \mathit{verb}\ \mathit{noun} \\
&\quad \textbf{where}\ \mathit{verb} = \mathit{head}\ [w \mid w \leftarrow \mathit{ws}, ...] \\
&\mathit{split}\ \mathit{ws}\ f\ \mathit{op}\ g = \mathit{Term}\ \mathit{op}\ [f\ \mathit{lhs}, g\ \mathit{rhs}]
\end{aligned}
$$

If the *goal* function is invoked with an empty list as an argument, it will eventually produce a *Term* value where the *String* in the *Term* will be a `Prelude.head: empty list` error. In *relation*, the definition of *verb* uses a call to *head* which is the cause of the error when *ws* is an empty list []. *verb* then gets passed to *split* as its third argument (*op*) which becomes the first argument in the returned *Term*. IRULAN reports the following error expression, which requires the use of case expressions to take apart the resulting data constructor:

```
case goal ([]) of
  Term x _ -> x ==> !  Prelude.head: empty list
```

Infinite loops, memory and stack overflow errors: While IRULAN cannot detect infinite loops per se, cases in which the evaluation of an expression exceeds the resources allocated by IRULAN are often indicative of pathological cases caused by bugs in the program.

In our experiments, the execution 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 time-out, and 4,122 times for the 128 MB allocation limit. A related error was also the discovery of Haskell `stack overflow` exceptions, of which there were 145. 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:

$$
\begin{aligned}
&\mathit{log2} :: \mathit{Integer} \to \mathit{Integer} \\
&\mathit{log2} = \mathit{genericLength} \circ \mathit{chop}\ 2 \\
&\mathit{chop} :: \mathit{Integer} \to \mathit{Integer} \to [\mathit{Integer}] \\
&\mathit{chop}\ b = \mathit{chop}'\ [\,] \\
&\quad \textbf{where}\ \mathit{chop}'\ a\ n = \textbf{if}\ n \equiv 0\ \textbf{then}\ a
\end{aligned}
$$

$$\textbf{else } chop'\ (r : a)\ q$$
$$\textbf{where } (q, r) = n\ `divMod`\ b$$

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.

5.1.3 Discussion

At a first glance, it may appear that many of the errors that IRULAN has found in the nofib suite can be stated as implicit precondition errors. For example, it is perfectly reasonable that taking the maximum of an empty list is an error, and that log base 2 is undefined for negative numbers.

However, the library writer has made these preconditions implicit, and, as IRULAN shows, the error messages they provide (assuming the function doesn't just crash) are not necessarily descriptive of the fault made by the library user. If these exceptions were thrown as part of a larger code base, then the debugging process has to start with a cryptic error message in a library before being traced backwards up into user code. If violations of the implicit precondition threw descriptive error messages, then some of this debugging process would be alleviated.

In addition, throwing descriptive error messages would document in the library that the failing behaviour was deliberate. The programmer who faces a `Prelude.head` exception in an external library and has to establish if it's his input violating a precondition, or a fault in the library itself is not in an enviable position (particularly if they have not established a minimal test case).

IRULAN however provides small test cases that precisely identify the error conditions. Given the small test cases it can be straightforward to see what the cause of the error was. Anecdotally, it was very easy to take any of the test cases presented above and establish why the functions in question failed on them, because the expressions in error only contain inputs relevant to tripping the exception.

Of course, because IRULAN has no explicitly stated preconditions attached to functions, it will report false positives. If the programmer adds defensive checks and throws descriptive error messages, IRULAN will still report these. As noted, when using the iterative deepening scheme, many, many error causing expressions may be found and shown to the user, however using a filter program (such as sort / uniq) can remove the trivially duplicate cases. We do not believe IRULAN should be used standalone, but as part of a system that allows the user to work through IRULAN's output and have the system remember expressions that are acceptable and those that are genuine errors. Later runs can then hide the accepted test expressions if they appear again, and only highlight the remaining genuine errors. Although this may seem like a lot of work (and would provide a lot of initial work if applied to an established project), if such a system was used incrementally from the start of development, it would be manageable, and provide another layer of checking at each milestone in development.

A rudimentary form of this strategy has been used for the integration tests of IRULAN itself. For each integration test, a Haskell module to be executed by IRULAN was written. The integration test framework would then run IRULAN in full trace mode on the module and record its output. This output was (the first time) manually inspected, to see if IRULAN had behaved as expected. If it was accepted (i.e. the output was as desired), it was saved alongside the test module to be picked up by the test framework. Future runs of the test would diff IRULAN's output with the already accepted version and bring attention to only the new test cases or any changes in behaviour that were noticed. Sometimes the accepted version of the output would need to be revised due to changes in IRULAN's behaviour, in others bugs had been introduced in IRULAN so the accepted version was providing a correct reference point to know when the bugs had been fixed.

If this technique were adopted by a user of IRULAN, it would be the modules in test framework (as opposed to IRULAN itself) that would be evolving, and the framework, through its use of IRULAN and diff would be able to highlight automatically any changes in behaviour (in terms of errors thrown). A user would likely also wish to run IRULAN's output through a filter (e.g. sort) before sending it to diff, so that similar error conditions (e.g. relating to the same root function) were grouped together, as the iterative deepening exploration strategy would make them appear separately otherwise. There would be some extra work needed to be undertaken by the programmer to validate the initial outputs, but in an incremental setting where this strategy is employed from the outset, the programmer will only ever be validating new or changed input-output pairs, and never looking at the same expression more than once (if it is correct).

This form of testing, where IRULAN displays both the input to the function you've written, and its current output (in the form of the exception it throws) is interesting. These techniques have since been generalised to not just showing expressions and the exceptions they throw, but to showing expressions and the value they produce by using the *show* function, allowing IRULAN to perform regression testing. This strategy is in contrast to unit testing, where the user has to explicitly state the inputs and the expected output. Here the user is presented input and output pairs, and just has to check (ideally once) that they match. Property testing falls somewhere in the middle of this continuum, as the user has to state explicitly the general property that holds, but it is up to the testing tool to provide the inputs and then show the inputs that don't obey the property.

5.2 Property Testing Comparison

In this section we evaluate the performance of IRULAN as a property testing tool, by comparing it to some of the existing established Haskell testing tools, QuickCheck, Smallcheck and Lazy SmallCheck. We use as our benchmark the same set of programs used to previously compare SmallCheck and LazySmallcheck, from [RNL08].

This benchmark consists of 16 properties based on the 10 following programs:

- Okasaki's Red-Black tree implementation (with fault injected)
- Bird's Huffman codec

- Turner's abstraction algorithm
- Cryptarithmetic solver (Claessen et al. AFP 2003)
- Hutton's Countdown solver
- Mitchell's Catch lemma
- Runciman's Mate-in-N finder
- Set as ordered list
- SAD circuit
- Mux circuit

One problem that immediately strikes us is how to configure the tools, and what a meaningful comparison is. QuickCheck's random technique can be configured to terminate after a number of tests have run, SmallCheck and Lazy SmallCheck work on a notion of depth, that while shared between the two tools is markedly different to the default notion of depth in IRULAN.

However, the quantity of code coverage each tool achieves on a particular benchmark, whether the tool finds the bugs in the erroneous benchmarks, and how much wall-clock time they spend searching are reasonable comparisons to make. So in order to compare them we set each to use an iterative deepening traversal scheme. For Quickcheck, we iterate using a similar formula to the one used in the original benchmark, this means running n batches of 1000 successful tests with QuickCheck's maximum expression size *bound* parameter set to $n * 2 / 3$, for increasing values of n.

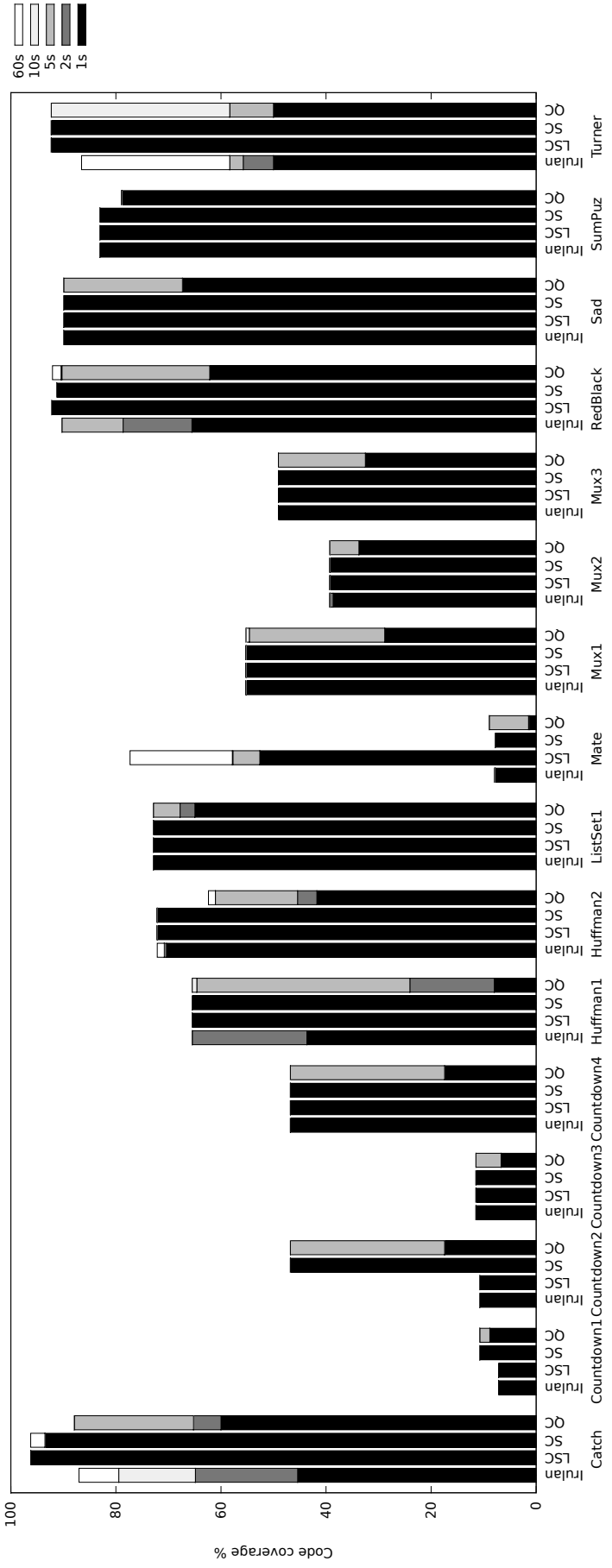
Each tool was run on each property for increasing amounts of time (1, 2, 5, 10 and 60 seconds) and the expression code coverage (as reported by HPC) recorded at the end. For each time and tool, the runs were repeated four times and the results averaged. We used a quad-core, 2.66Ghz Intel Core 2, Debian Linux, 2.6.32, 64 bit kernel with 4 GB RAM for the experiments.

Figure 5.3 presents the code coverage results for the different properties. It is important to note that absolute coverage is not important (in many cases the properties tested simply do not aim to cover 100% of the code), but the relative coverage between the tools is what is interesting.

For example, looking at the `Catch` property, we can see that in 1 second IRULAN managed to obtain 45% coverage, at 2 seconds this had improved to 65%. Running IRULAN for 5 seconds didn't make any further improvements, however after 10 seconds the coverage had increased again to 79%, and after a minute covered up to 87%. In comparison, Lazy SmallCheck achieved 96% coverage in under a second, and didn't improve with a longer runtime.

The coverage obtained by QuickCheck did not always increase as the runtime increased, this is due to QuickCheck's random exploration scheme sometimes getting "lucky" during the shorter runs. When plotting the results, we made the results at higher times be the maximum of the result at lower times and the current result.

Figure 5.3 Property testing, comparison of code coverage achieved by different tools on different benchmark properties



Some of the benchmarks require built in values (e.g. *Int*s or *Char*s) as part of testing. While the other tools all have ways of relating the current depth to the range of values to enumerate, IRULAN explicitly requires the primitive constants to be added to its command line. With the exception of *Mate* and *RedBlack* as discussed below, properties that needed *Int* values (`Countdown1,2,3,4`) had $-1, 0, 1$ and 2 added, and properties that needed *Char*s (`Huffman1,2` and `SumPuz`) had `'a'`, `'b'` and `'c'` added.

The graph shows some general trends, for example on half of the properties tried, all tools eventually achieve equal coverage, and SmallCheck or Lazy SmallCheck are always one of the best performers in these tests.

As described in [RNL08], the `RedBlack` and `Mate` properties do have counterexamples available to be found. Usually the Check tools will terminate when a counterexample is found, so in order to accurately measure code coverage for the entire run, we modified them to carry on executing after reporting the counterexample.

The `RedBlack` property (a test of a Red-Black tree backed implementation of set) tests code with a deliberate fault injected. Lazy SmallCheck is the only tool that finds the counterexample within 60 seconds on our test. (It finds a counterexample within 1 second). IRULAN, when given constants $-2, -1, 0$ and left to run in iterative deepening mode, will not find the counterexample within 20 minutes of testing. This is due to a mixture of the difference in the search spaces explored by IRULAN and Lazy SmallCheck (for example LSC does not consider tuples to increase the notion of depth whereas IRULAN does, which makes some difference), but also due to the overheads IRULAN has with having to compute and compile arguments. It is possible to constrain IRULAN's search space beyond just a depth bound (e.g. by pruning the search when the number of non-terminals and terminals in any expression exceeds some limit), and with that constraint and a tuned depth limit (i.e. not iterative deepening), IRULAN can find a counterexample in under 3 minutes.

The `Mate` property (a test of a mate-in-N chess problem solver) also features a deliberately injected bug, however only Lazy SmallCheck finds a counterexample (within 60 seconds in our test).

The `Turner` program and property was the most interesting in the benchmark suite. The program compiles and optimises lambda expressions using Turner's combinators [Tur79]. The implementation is based on the description from Chapter 16 of [Pey87]. The core property comes from [Tur79], that using the combinators to abstract a variable from an expression, and then applying (using :@) the variable to the result should yield the original expression; i.e.

$$prop\_abstr\ (v, e) = simplify\ (abstr\ v\ e\ :@V\ v) \equiv e$$

Where variables (v), expressions and combinators are defined thus:

$$\textbf{data}\ Var = V0 \mid V1$$
$$\textbf{data}\ Exp = Exp\ :@Exp \mid L\ Var\ Exp \mid V\ \ Var \mid F\ Comb$$
$$\textbf{data}\ Comb = I \mid K \mid B \mid C \mid S \mid C' \mid B' \mid S'$$

When IRULAN was run on this module, it immediately found counterexamples, such as:

$$prop\_abstr\ (?, ((F\ I)\ :@(F?)))$$

while the other tools found nothing.

The F constructor in *Exp* wraps the target combinators. If the original expression being abstracted and simplified contains combinators (i.e. uses the F constructor) then the property doesn't necessarily hold. This is because the original expression may not be in some normal form (i.e. $simplify\ e \not\equiv e$), so the *simplify* after re-abstracting may end up with a simpler expression than was started with. This inequality between before and after expressions would then cause the test to fail.

The instances for *Serial* and *Arbitrary* used by the Check tools (presumably) deliberately omitted the F constructor, so that no expressions with combinators in them would be generated. IRULAN however just saw the publicly exported constructors attempted to use all and thus found the counterexamples. In effect, IRULAN discovered an invariant of the original property that was encoded (but not explicitly documented) in the generation instances for the Check tools.

There are two possible fixes, one is to add an explicit export list to the module to make sure the F constructor can't be used (and thus stop IRULAN from trying to use it), or to weaken the property and make the Check instances use all the data constructors. We chose the latter, and altered the property to the weaker:

$$prop\_abstr\ (v, e) = simplify\ (abstr\ v\ e :@V\ v) \equiv simplify\ e$$

The results shown are for running against this version. No tools found any counter examples.

With the exception noted above for the `Turner` benchmark, IRULAN required no changes making to the tests, and did not require any specialised export lists to enable property testing. The other tools do of course require the instances for *Arbitrary* or *Serial* providing, although we note that using the reflection technology that IRULAN uses it could be fairly straightforward to automatically generate such instances without requiring the user to write any boilerplate code.

5.2.1 Performance Comparison

We have also looked in detail at the relative performance of Lazy SmallCheck and IRULAN on the TestListSet1 benchmark. This benchmark features a set implementation backed by an ordered list, and the property checks that the *insert* function maintains set ordering:

$$\textbf{type}\ Set\ a = [a]$$
$$prop\_insertSet :: (Char, Set\ Char) \to Bool$$
$$prop\_insertSet\ (c, s) = ordered\ s \longrightarrow ordered\ (insert\ c\ s)$$

Lazy SmallCheck and IRULAN will be required to enumerate lists of *Char*s as the second (uncurried) argument to the function. Due to laziness, many of these lists which are not ordered will not require their tails being generated.

For this performance comparison, we modified Lazy SmallCheck to print out each test argument as it is tested. If an unrefined value is part of the expression being tested (analogous to an IRULAN `?` argument), then Lazy SmallCheck will instantiate it with the first value from the possible instantiations before printing it out.

A typical run of this property for 60 seconds using iterative deepening caused 10,202,802 expressions to be executed, which corresponded to 4,824,357 unique expressions being tested. IRULAN by comparison executed 102,058 expressions, which corresponded to 13,775 unique expressions. This indicates that Lazy SmallCheck is at least two orders of magnitude faster than IRULAN at building and executing test expressions. Of course this includes the overheads of converting each expression to a string and printing it out, without which Lazy SmallCheck is likely even faster.

However it is important to note that the two tools managed to explore different search spaces for these properties during the 60 seconds of iterative deepening. In-fact, neither tool subsumes the other in the test expressions generated. IRULAN managed to produce some larger test inputs, for example: (`'a'`, `"aaaaaaaaaaaaaaaaaaa"`) (these accounted for 3,946 extra unique cases Lazy SmallCheck didn't reach). On the other hand, Lazy SmallCheck expands the range of *Char* constants it will use as the depth increases, unlike IRULAN which uses a fixed set (`'a'`, `'b'`, `'c'` in this case) specified on the command line. This means Lazy SmallCheck produces many test cases (e.g. (`'c'`, `"blllll"`)), that IRULAN never will. In this example, Lazy SmallCheck produces 4,819,694 unique test cases that IRULAN did not.

In order to better understand the two orders of magnitude performance difference between the two tools, we undertook to profile Lazy SmallCheck and IRULAN. We used GHC's built in profiling support [SPJ95] to analyze Lazy SmallCheck running on the TestListSet1 benchmark. Unfortunately IRULAN's use of the GHC API to construct bytecode backed expressions means that IRULAN cannot be successfully profiled using GHC's native profiling tools. We instead integrated a simple profiling mechanism into IRULAN that allows us to record the execution time for certain expressions (taking into account the execution time for their children). Although this shouldn't be relied on for authoritative information, it does clearly indicate that handling ? errors in the Plan, and building the expressions to execute are the most time consuming parts of IRULAN's runtime.

Profiling was performed against a 60 seconds iterative deepening run of both tools. In both cases, relatively little runtime was devoted to actually running the property; for Lazy SmallCheck *ordered* and *insert* accounted for only 3.3% (1.98s) of the total runtime. IRULAN only spent approximately 2.5 seconds actually executing test expressions (but this includes the overheads of setting up threads and communication via MVars to watch for excessive allocations and time usage).

Most of the execution time of Lazy SmallCheck was spent in the children of the two functions *refute* (30.72s) and *run* (26.34%). *refute* contains the logic to apply the test function to its arguments, performing refinement if necessary; *run* constructs the test function and arguments that are passed to refute. The most expensive single site is the function *total*, which took 13.91 seconds. *total* is used to turn partial values into total ones, and has been heavily used as we were printing out every expression tested during the run.

IRULAN's biggest hotspot was the control logic for establishing what *Plan* step to execute following a ? argument,

where approximately 16 of the 60 seconds were spent. The second largest hotspot (13 seconds) was in the construction of the runtime values from abstract test expressions.

5.2.2 Discussion

This benchmark, despite at no point showing IRULAN to be the best tool for a particular property, does validate that IRULAN is a usable property testing tool that does explore a similar search space to the other tools.

Unfortunately, the cost of reflecting, building, executing and analysing test expressions does affect IRULAN's performance and means it can take longer to explore the same space as the other tools. In this benchmark, IRULAN's overheads compared to the other tools for building and testing a single test expression are particularly exaggerated, as all of the test data is built by the simple application of data constructors to one another. This means that while the other tools build a value out of the constructors and can use the *Show* instance of those constructors to natively format them to show the user what test expression was just executed, IRULAN maintains its own meta representation of the expressions it is executing to show the user, as well as building the value out of data constructors to pass to the property function being tested.

IRULAN has these overheads because it has been designed for a more general automatic test expression generation. For the user of a property testing application, some of these features could provide benefits that are not witnessed in this benchmark. For example, IRULAN's ability to take test data from its support set and use it to test a polymorphic function through unification means that testing a polymorphic property at various types can be slightly more naturally expressed than creating copies or aliases of the property with the explicitly monomorphic types.

IRULAN's ability to build and remember the test expressions associated with a test value can also be useful in some cases. For example if a data type is not publicly exported, but instead only exports smart constructors (i.e. functions) then it is likely that the show instance for that data type will abstract away from the order of calls to the smart constructors. For example, popular *Set* libraries for Haskell may build up a *Set* using a combination of *insert* and *delete* calls (e.g. $insert\ 3 \circ delete\ 4 \circ insert\ 2 \circ insert\ 4\ \$\ empty$), but will render the structure as *fromList* $[2, 3]$. Even if the Check tools are told to use the smart constructors *insert* and *empty*, they will report test data using *fromList*, as that is what the *Show* instance uses. If there was a fault in *insert* then some extra work must be done to track the chain of calls that occured.

In conclusion, IRULAN can make testing some kinds of properties easier for a developer due to its automatic inference schemes and remembering of the expression generated as a test argument. However, if you can express and maintain the test generation code in a way supported by the Check library approaches, then your performance is likely to be better due to the much lower overheads these libraries enjoy.

5.3 Evaluating the Cache

The runtime cache, described in Section 4.5 was assumed to provide benefits by requiring fewer duplicate expressions to be run, and pruning large duplicate branches, at the expense of more memory usage. However, the benefits of the cache also have to outweigh the overheads of insertions and lookups into it.

During early experimentation and development with IRULAN, we noticed that the overheads were only sometimes amortized by the cache. However, at that time, IRULAN executed test expressions by building GHC core expressions, and then getting GHC to compile and link them before running them. We then realised that IRULAN could look up identifiers directly from the linker and cast (*unsafeCoerce*) them to apply them, as it does now. By cutting out the GHC core compilation step, compilation and execution times became significantly smaller and the cache overheads became more noticeable.

As an informal evaluation of the cache in the latest version of IRULAN, we ran IRULAN in several configurations on two programs from the spectral suite of the nofib benchmark. The programs were chosen arbitrarily from the set of programs that have a long (greater than 1 minute) runtime for DFS at depth 25. `mandel2` is a Mandelbrot set generator, and `simple` is a numerical application looking at energy in a fluid simulation.

These experiments were run on a Intel(R) Core(TM) i5 CPU 650, 3.20GHz (dual-core with 2 hyper threads per core), Linux 2.6.35.6 #2 SMP x86_64 GNU/Linux, 8GB RAM. 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. IRULAN was run against these two programs using DFS with depth limits of 20 and 25. At both depths for both programs, we ran IRULAN with various runtime cache configurations:

- FIFO Cache, with a maximum size of 10 expressions
- FIFO Cache, with a maximum size of 100 expressions
- FIFO Cache, with a maximum size of 1000 expressions
- Caching Disabled
- Unlimited Cache

In Figure 5.4 we show graphically how long IRULAN took to cover the respective search space for each of these experiments. In both programs at the higher depth (that required a longer runtime), the overheads of the cache in any form is clearly in excess of the benefits it brings.

In Table 5.2 and Table 5.3 we present some statistics about the outcomes of expressions executed during the runs at depth 25. For each configuration we give the number of test expressions that were:

- **cached** - These test expressions had previously been run by IRULAN, and the result for the expression was found in the cache, so it wasn't necessary to execute it again.

Figure 5.4 A Cache experiment example

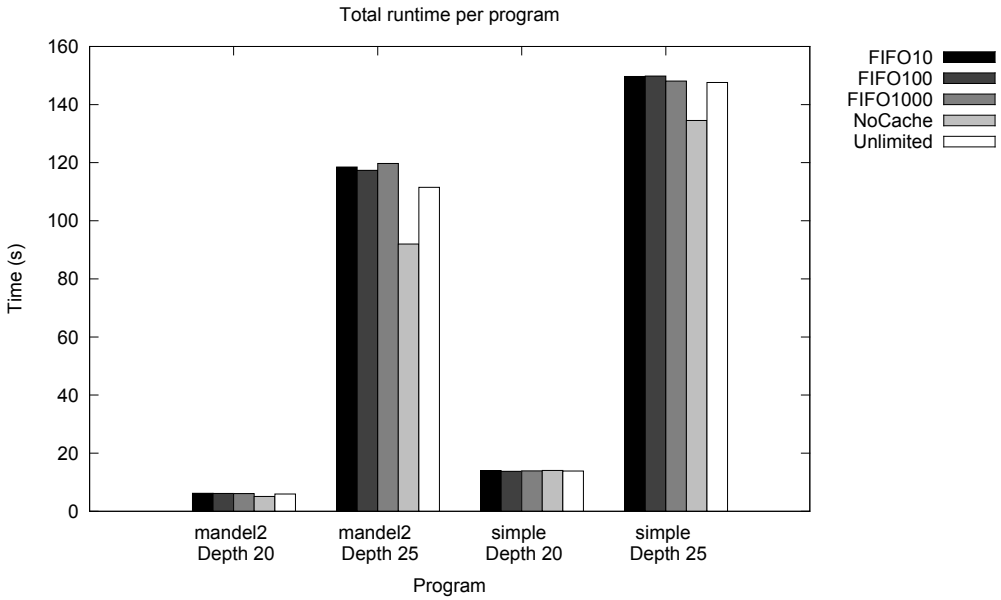


Table 5.2 `simple` execution at depth 25

Configuration	# cached	# executed	# redundant	# pruned
FIFO 10	212	97316	224	30
FIFO 100	571	96739	229	248
FIFO 1000	571	96739	234	248
Unlimited	571	96739	241	248
No Cache	-	97558	-	-

Table 5.3 `mandel2` execution at depth 25

Configuration	# cached	# executed	# redundant	# pruned
FIFO 10	0	150735	2516	0
FIFO 100	0	150735	3550	0
FIFO 1000	0	150735	3616	0
Unlimited	0	150735	3623	0
No Cache	-	150735	-	-

- **executed** - These test expressions did not have an entry in the cache, so they had to be executed.
- **redundant** - These are test expressions that were executed, but their runtime value was discovered to be the same to some previous expression's value, and so the branch following this expression would have been pruned.
- **pruned** - This is the number of test expressions that didn't need to be considered at all due to the pruning following redundant expressions. This number is derived by subtracting the number of cached and number of executed expressions from the number of executed expressions in the No Cache configuration.

From Table 5.2, we see that for `simple`, less than 1% of all expressions are found in the cache. There are also redundant expressions that lead to some pruning of the tree, but the amount of pruning achieved is (relatively) minuscule and, as Figure 5.4 shows, provides no overall benefit.

`mandel2`, in Table 5.3 is the canonical worst case - there are no cache hits at all in any configuration. There are, however, many redundant expressions discovered (about 2% of all expressions have a runtime value that has been seen before), and as the cache grows in size, more of these are detected. However, their detection does not cause any expressions to be pruned. This means the redundant expressions were detected at the natural end of a plan, where the value being generated is not going to be reused later. The lookups and stores in this case are simply extra overheads that have not provided any benefit.

In general we have found the overheads of using the cache (extra time for lookups and insertions) and memory use, plus the added variable of the size of the cache to use (for longer runs of IRULAN, the unlimited cache will quickly exhaust system resources) mean that for most cases, the cache is not useful. In future, it may be interesting to see if it could be optimised (perhaps with some extra information from the search strategy and the *Plan*).

There may be other programs for which IRULAN would benefit from the current cache , but this will require test expressions that take consistently longer to execute than we have found in our experimental benchmarks.

5.4 Evaluating the Search Strategies

When running IRULAN, the user has a choice of several search strategies. For our larger benchmarks we have used the iterative deepening strategy, and we attempt to motivate that choice here. The underlying idea behind this experiment is that when selecting a search strategy, the user won't necessarily know which one to use in general. Then, once the user has selected a search strategy to use, many of the strategies need a depth parameter, expressing a limit of how deep into the *Plan* they should explore. A user however is likely to only know one piece of information; how long they are willing to wait for some preliminary information. Given just the constraint of how long they should wait, what search strategy is most effective?

We present the results from running IRULAN on three programs from the spectral suit of the nofib benchmark. These programs were chosen arbitrarily from the set of programs that kept increasing code coverage due to longer

runtime in our initial nofib experiment. This means that the choice and parametrisation of the search strategy used to explore these programs is important, as for the fixed amount of time presented there is a genuine range of code coverage that could be achieved. `boyer2` is a Haskell implementation of the Boyer program from the Gabriel Lisp benchmarks ([Gab85]). The original Boyer program performs rule-directed rewriting. The Haskell re-implementation features a module that has a data structure to represent Lisp expressions, and combinators to bridge between normal Haskell expressions and Lisp ones (as the comment in the module claims "Lisp-like functions which allow easy hand translation from Lisp to Hope+"). `mate` is a checkmate-in-n solving application and `minimax` is a solver for tic-tac-toe based on the mini-max algorithm.

These experiments were run on a Intel(R) Core(TM) i5 CPU 650, 3.20GHz (dual-core with 2 hyper threads per core), Linux 2.6.35.6 #2 SMP x86_64 GNU/Linux, 8GB RAM. 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. IRULAN was run against each module in these three programs with the cache disabled, with a time limit of 60 seconds per module.

With this configuration, we varied the search strategy used to explore the *Plan*. We tested DFS in four variations, each running to a different depth. The depth limits used were 30, 35, 40 and 45. If the 60 seconds of testing time for the module finished before the search space was fully explored, then IRULAN would stop and record the code coverage achieved thus-far. We also ran our time split variation on DFS, our random restarting strategy and iterative deepening.

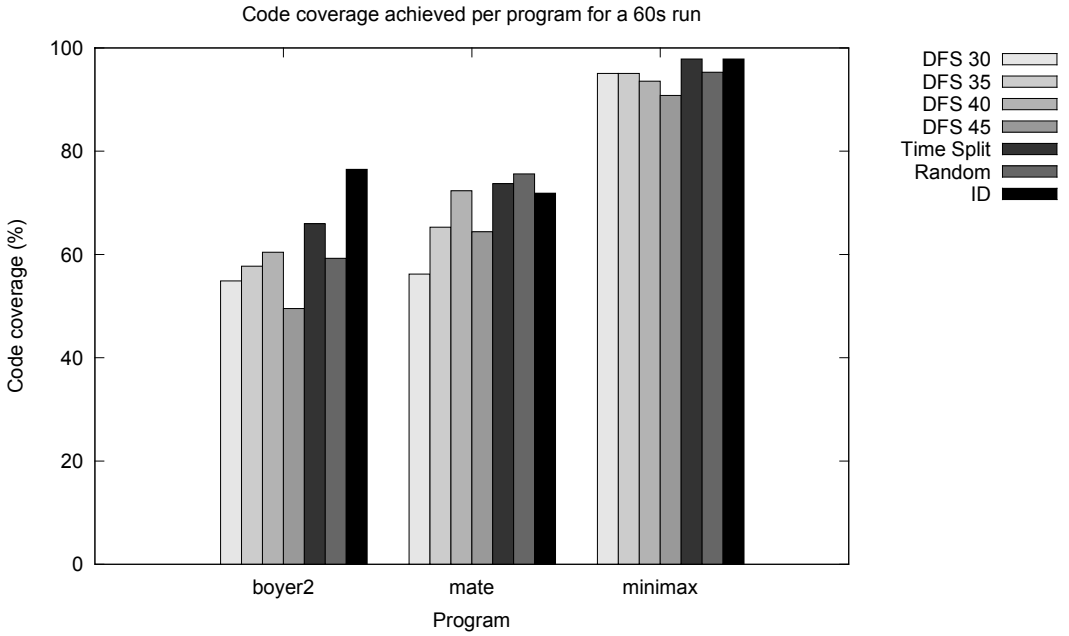
A slight wart in the implementation of IRULAN means that the time split and random strategies do require a depth bound to be set. In this experiment we used a bound of 100, which should be large enough to be as though no limit was set. All the strategies could have had extra bounds placed based to prune their searches based on syntactic properties of the test expressions seen. However we assume a user would not want to use these in the first instance, only if they were interesting in shaping the search space in a very specific way.

In Figure 5.5 we show graphically the code coverage achieved for each configuration for each program.

The first four grey bars for each program are the DFS runs. The `boyer2` and `mate` graphs show how important the right choice of depth limit for DFS is if there is only a fixed amount of time permitted for a run. Between depths 40 and 45 for these two programs the code coverage suddenly drops, demonstrating that at depth 45 the search space was not fully explored, and a large proportion of branches that would have been explored at a lower depth were just not reached in time.

The next three darker bars are for (in order left to right) time split, random and iterative deepening. Random search (the middle bar of the three) for a given amount of time can either be lucky (e.g. in `mate` where it performs best) or unlucky (e.g. in `boyer2` where it performs worse than DFS). Time split performs consistently well, outperforming DFS, but can still be unlucky with the branches that it has to prune due to time constraints, which cost it a lot of potential code coverage in `boyer2`. Iterative deepening also performs consistently well. In `boyer2` it managed to iterate up to a sweet spot of depth between 40 and 45 that meant it was able to gain more coverage than the other traversal schemes.

Figure 5.5 Code coverage achieved for the different search strategy configurations



However in `mate` we see iterative deepening achieve less coverage than DFS (at depth 40). This is an example where choosing the highest depth bound possible to reach in the time allocated and just exploring that (as DFS does) will likely outperform iterative deepening (which will have to iterate up to that depth). Of course, this requires an oracle to know what the right depth was (if the oracle chooses too high or too low, as per the other DFS runs in `mate`, and iterative deepening is a superior choice again).

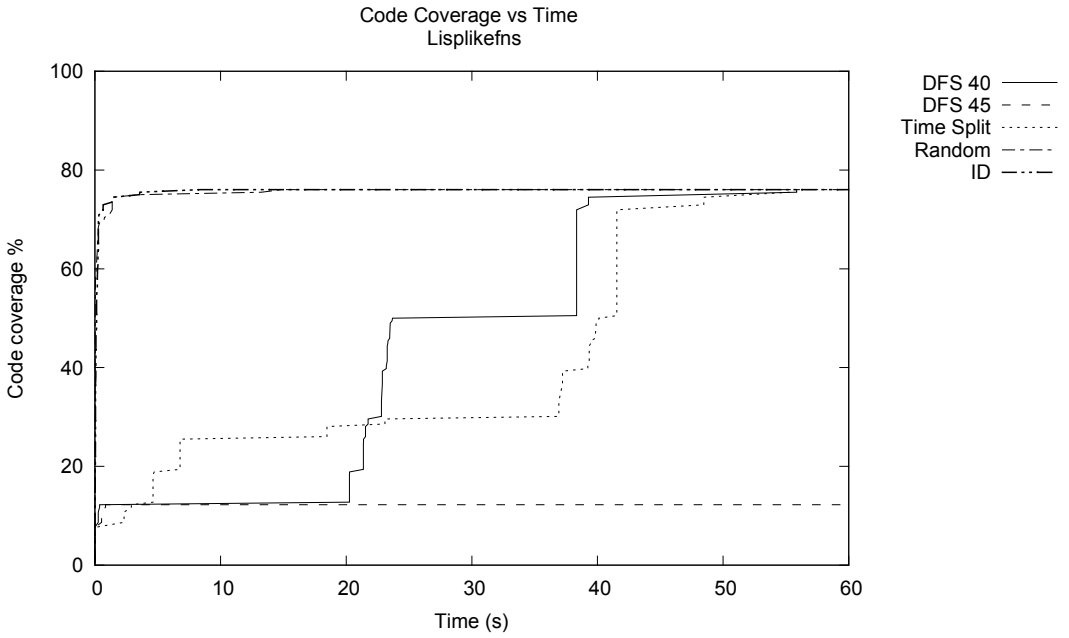
In Figure 5.6 we show how the code coverage changed as time proceeded during the testing of one module. We chose the *Lisplikefns* module from `boyer2` (the module of Haskell to Lisp combinators described above). Iterative Deepening and Random very quickly reach the ceiling of the code coverage they can achieve, indicating that (in this case) the user could have run for 10 or 20 seconds as opposed to the full 60. Time split (the dotted line increasing throughout the entire minute) explores the search space evenly across the time allotted, and thus takes longer to reach its ceiling. DFS 40, which in this case explores its search space in just under 60 seconds behaves similarly to time split. Finally, the higher DFS 45 gets stuck in an early branch and flat lines at about 15% code coverage for the entire 60 seconds.

5.5 Regression Testing

A small modification to Irulan has allowed us to find high-level regression bugs by cross-checking different versions of the same application. To compare two different versions, we use Irulan to generate a test suite for each, and then compare the two test suites to detect changes in behaviour. 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.

In order to build a test suite, it is necessary to build a set of input, output pairs which cover a fragment of the

Figure 5.6 A breakdown of code coverage against time for the *Lisplikefns* module in `boyer2`



behaviour of the function under test. Thus far, we have seen how IRULAN can generate inputs for a function during its testing routine. However, IRULAN observes only whether a test reduced to WHNF successfully, or the contents of the error produced if not. For regression testing, a richer form of test output expectation was necessary.

Frequently a programmer will want to be able to visualise values, for example by printing them to a console. In Haskell values of a particular type can be converted into *String*s for printing to the console or serialising to disk through the type class *Show*. Conventionally, programmers will instruct the compiler to automatically derive the instance of *Show* for them. We extended IRULAN with the ability to use this type class when available to present the actual value produced during testing, instead of simply "`.`". These *String* result values now provide test cases with richer expected outputs than previously. When IRULAN is run on a module, it can save a set of test cases that produce either exceptions or string values; building up a test suite specifying the behaviour of that module.

When run on a different implementation of the same module, another test suite can be built. We investigated how these two test suites could be compared, to show how the behaviour of a module has changed between two implementations.

Given two test suites, *T1* and *T2*, we try to match input-output pairs in *T1* with corresponding pairs from *T2*. Considering a pair $(i1, o1)$ from *T1* we would want to present the corresponding test inputs and outputs from *T2*, and check that the behaviour of the implementation hasn't changed. However, due to laziness made explicit through the use of ? arguments, There are several cases to consider:

1. There is exactly one test case $(i2, o2)$ in *T2* where *i2* is identical to *i1*. We then report a change in behaviour if *o1* and *o2* differ.

2. If *i1* contains ?s, then there may be many test cases $(i2, o2)$ in *T2* such that *i1* is more general than *i2*. By more general, we mean that *i1* is identical to *i2*, except that where *i1* features a ?, *i2* 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 *T2* than in *T1*.

   We match the single more general test expression *i1* with all of its more specific instances in the second suite. We report an error if any of the specific output instances *o2* have a different result value from the output *o1*.

3. The symmetric case, where *i1* is one of several test inputs in *T1* that are more specific than a single test input *i2* in *T2* has also to be considered.

4. *i1* may have no corresponding test input in the other suite, i.e. there are no test inputs *i2* in *T2* such that *i2* is more general, equal to or more specific than *i1*. We report *i1* 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 *T1*, but *foo* Y ? ==>*False* in *T2*. This could also occur if IRULAN was exploring a different search space (perhaps iterative deepening was used and the implementation managed to get further into the search space in one implementation than another).

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 only including a set of test cases that successfully throw an error (not including ? argument errors) or return values that can be pretty printed to *String* values via Haskell's *show* convention.

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

Undergraduate Programming Exam

The first year undergraduate Computing students at Imperial sit a 3 hour practical Haskell programming exam during the Spring term. Students are given a written description of the algorithm to implement, broken down into functions to be implemented. With the task description, the students are given a skeleton Haskell module which features stubs for functions that the students must implement, and some test cases that can be used by the students to check they are doing the right thing. A sample answer is also produced, to aid in creating a mark scheme and for the manual creation of a test suite by a teaching associate for automated testing of the student's solutions.

The 2011 exam problem was to implement a type inference and checking algorithm for a small core functional language. This involved implementing a function for unifying two types, and polymorphic type inference for each of the constructs in the language.

Before being given to the students, the exam was trialled on two PhD students in the department. Both of the PhD student implementations fully passed the test suite in the skeleton file. However running IRULAN on one of the PhD student solutions revealed immediately an infinite loop condition in the implementation of unification that hadn't been spotted[2].

Once this bug had been fixed, test suites corresponding to the PhD student implementations and the sample answer were created using IRULAN, and then pairwise compared. The comparison highlighted several subtle differences in behaviour between the model answer and both PhD student solutions. Upon closer inspection, these revealed some subtle bugs to do with the propagation of type errors through recursive calls during the type inference implementation in the model answer. Neither the small built in test suite, nor the later larger teaching associate test suite highlighted this bug.

Hackage libraries

Hackage[3] is a public collection of released Haskell libraries and applications. Authors can upload their Haskell source to the Hackage database, and there is an established tool chain for users to download and install software uploaded to Hackage. 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.

In the Containers and Diff libraries IRULAN automatically highlights changes in behaviour that were intentional; the first being an optimisation, and the second a bugfix. In the TreeStructures library IRULAN identified a change in behaviour that indicates a bug, and the test cases made it easy for us to locate the change which introduced the bug (a contributed patch not by the original author). Finally in Presburger IRULAN identified several bugs introduced in the unreleased version control head, as well as changes in behaviour. Also, even without performing regression testing IRULAN's normal error finding behaviour found newly introduced bugs in the unreleased library version.

IRULAN was configured to run for 60 seconds per module with an iterative deepening exploration scheme. In addition we explicitly added the *Int* constants 0, 1, 2, 3. However for Presburger we used a different pool of constants: -1, 0, 1, 2, 101. We originally tested Presburger in the original configuration, and discovered some errors and changes in behaviour (due to the 0 constant being in that configuration). We then looked over the documentation to the library and inferred that -1 might also be a useful value to test, which then revealed more errors. So without a deep knowledge of the source code of the library, our IRULAN based testing was able to inform further, useful IRULAN based testing.

[2] Full disclosure: we were one of these PhD students, and, embarrassingly, the bug was found in our implementation.
[3] `http://www.hackage.haskell.org`

We present our findings from some of these libraries.

Containers

The Containers library features efficient implementations of many general purpose container data types[4]. IRULAN was able to detect a change in behaviour between versions 0.2.0.0 and 0.3.0.0 of Data.IntSet (an efficient set for *Int*s):

```
Test case: fromAscList [0,0,0,1,0]
v 0.2.0.0: fromList [0,1]
v 0.3.0.0: fromList [0,0,1]
```

The input list, $[0, 0, 0, 1, 0]$ is not an ascending list and therefore fails the precondition of *fromAscList*. The test case produced by IRULAN shows that previously that precondition wasn't assumed, whereas version 0.3.0.0 of the library assumes the precondition (presumably to provide a more efficient implementation). It should be noted that the documentation of the library was also changed between the two versions to state that the precondition is now not checked.

Diff

The Diff library provides an implementation of the standard diff algorithm[5]. Regression testing of released versions 0.1 and 0.1.1 of the Diff package highlighted a change in behaviour. Looking at the examples provided by IRULAN it is easy to see that a bug had been fixed.

Consider the following test case produced by IRULAN (neatened for presentation):

```
Test case: getDiff [0,1] [1,3]
    v 0.1:  [(S,1),(S,3),(F,0),(F,1)]
  v 0.1.1:  [(F,0),(B,1),(S,3)]
```

It demonstrates that calculating the difference of two lists with a shared element (the element 1 in both arguments) previously would not identify the element as coming from both (B), and instead duplicate it by saying it came from the second list (S) and then from the first (F).

Given the precise test cases it was easy to then look at the source code and discover the change that had fixed the bug, which was a tiny change of two characters in a single line of the code. We contacted the author of the Diff library to confirm we had found a fixed bug, and to enquire about the testing methodology used to find and fix it, and received the following reply:

> I haven't touched this library in years, but that was indeed a bug that was fixed. Diff was one of my first Haskell projects, and I remember very little about it. I think I found the bug because

[4] http://hackage.haskell.org/package/containers
[5] http://hackage.haskell.org/package/Diff

> somebody pointed it out to me on IRC, since I remember it being pointed out, and a search in my archives doesn't reveal an email about it. At the time I wasn't yet conversant with quickcheck & co, otherwise I would have been able to catch it using them. Sorry I don't have more to add.

TreeStructures: The TreeStructures library provides implementations of various heap and tree data structures[6].

The comparison of the test suites revealed that the building of a binary heap from a list of elements had changed between the 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 yielded the following reply:

> I'm a bit embarrassed that I let that bug slip in! When I incorporated [redacted]'s patch, I didn't look too closely, kicked off the (minimal) QuickCheck tests and applied it. I haven't had a chance to look further than your examples and the patch, but it does indeed look like it's building a linked list, rather than a well formed heap.

Testing the TreeStructures module also revealed a weakness in IRULAN's automatic matching up of test inputs. Between the two released implementations, the argument order to several functions changed. This meant that we could not automatically compare test inputs involving these changed functions; improving matching to detect such changes would be a useful future work.

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 [Coo72], 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. In addition IRULAN identified that the evaluation order of arguments across implication had changed between the two versions. According to the author of the library:

> it's been a while since a looked at this code, but I think what happened is that I was trying to implement some additional optimizations and then I broke something and never got around to fixing it. Ah, open source development :-) Based on your examples, these look very much like real bugs, so I should certainly take a look at the code again. At this stage of the project, I am a lot less concerned with the lazyness issues because there is a lot one can do to improve the performance

[6] `http://hackage.haskell.org/package/TreeStructures`

(both by making things more strict to avoid memory leaks, and more lazy, to avoid unnecessary checks). In general, being able to spot these difference seems like useful functionality though.

Is your tool available to try out somewhere? It certainly seems quite useful!

5.5.1 Discussion

This experiment, applying the test generation algorithms presented so far to regression testing (as opposed to error finding or property testing), has proven successful and suggests some interesting avenues for further work. The most immediate issue, as highlighted by the TreeStructures library, is that changes in an API, such as an argument reordering (or a function rename) would break our simplistic matching algorithm. This then makes it hard to automatically present the input, output pairs where behaviours have changed. There are several solutions to this problem, from the fully manual: requiring the programmer specify the mapping between old inputs and new inputs, to more automated (perhaps using edit-distance like algorithms and type information to work out the most likely candidate for rename or argument swapped APIs).

One possible weakness of this approach is that it relies on *Show* instances to be available for the result values of the functions being tested. Further than that, the approach also relies on the *Show* instance accurately representing the internal state of the value, and that string inequality between shown values means the values are meaningfully different. In general, (and for the examples we have tried) this has been the case, but this experiment is still early work and so it may be that in larger systems these assumptions are not safe to make.

Of course we could have generated 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 not provide the rich information that cross-checking the test suites provides.

Chapter 6

Conclusion

In this thesis we have shown that:

- Appropriate sets of identifiers to use to create test data can be obtained automatically by only inspecting the types of the functions to test. Our algorithm, based on looking in the module declaring the type of interest, works well in practice. In addition, our visible constructor optimisation means that when a data type is not opaque, we keep the number of identifiers in our sets low by not including functions that can also build that type.
- The black box generation of Haskell expressions can yield test suites that achieve high code coverage for Haskell applications. By only relying on the API of the library being tested, and not its full source code, we have developed a testing strategy that can work with arbitrary libraries, be they transparent user developed or closed external binaries.
- Greater code coverage can be achieved by adding case expressions to the syntax of expressions generated. Many Haskell functions return composite data structures such as lists or tuples. Using case expressions to peek inside them, we can check that these lazy data structures don't hide bugs. In addition, case expressions provide extra ways to generate test data, by allowing the extraction of wrapped values from composite data structures.
- Control flow that leads to exceptions being thrown can be triggered by our black box testing technique, and the expressions that trigger exceptions can aid in understanding the bug. Our implementation will check what the runtime value of an expression is, and will report to the user if an exception was thrown (or if it took longer than a limit to execute), along with the expression. We have found that understanding why a particular expression caused an exception is generally straightforward, as the expression usually represents a minimal test case for the bug.
- When testing is for a fixed amount of time, iterative deepening is the most effective strategy for exploring the huge search space of expressions that could be generated. An experimental alternative approach, time

split, also appears to be promising. If the user knows the depth to which they wish to explore the search space, then depth bounded is the most efficient, but may take an arbitrary amount of time.

- The overheads of using a cache to prevent execution of duplicate expressions and to prune some control flow branches outweighs just re-executing Haskell expressions, at the moment.
- It is possible to test meaningfully a user's library for erroneous edge cases without requiring them to make any source level changes to their Haskell application. In addition, if a user has written property predicates it is possible to test them without requiring any further source level changes, only provision of a small set of constants and time for an iterative deepening exploration to work. We believe that ideal tools work for a user, and not require the user to make extra changes to their application, tying it to the tool.

6.1 Summary of Technical Achievements

In this work we have explored one facet of automated black box testing in the context of Haskell. Our exploration has detailed several algorithms and data structures that have been useful for our goals. These have then been realised in a tool, IRULAN, which we have evaluated to demonstrate the effectiveness of our ideas. We now briefly summarise the achievements of our work, highlighting the different algorithms, and subsequent evaluation performed.

- **The automatic inference of a Support Set**. We have explained how we automatically infer appropriate identifiers and constructors from the functions to be tested. The user can also augment this support set with primitive constants and identifiers from other modules.

  As part of the support set we also discussed:

  - **The *TypeMap*** which allows lookup of values keyed by types that unify with a queried type.
  - **The Constructor Graph** which allows construction of NFAs that outline how to use case expressions to build an expression of a certain type.

- **The *Plan* algorithm**. We have given a pseudo-code implementation of the creation of a data structure that encodes testing Haskell functions. It is based on a needed narrowing or lazy instantiation approach, and extended with the ability to use case expressions to build expressions.

  It also featured:

  - **A threaded cache** which removes some redundant expressions from being considered for testing.
  - **Polymorphism** handled by keeping types at their most general for as long as possible to prevent premature commitment and thus unnecessary expansion of the search space.

- **nofib**. We ran IRULAN against programs from the nofib benchmark suite, and showed the code coverage that it can automatically achieve and presented some of the errors that it found.

- **Property Testing**. We compared IRULAN with the existing Haskell property testing tools QuickCheck, SmallCheck and Lazy SmallCheck and showed that it is usable as a property testing tool.
- **Runtime Caches**. We hypothesised that referential transparency meant that caches could prevent some duplicate test expressions being executed, and allow the pruning of some branches. Unfortunately some experimental evaluation showed that the cache overheads were too great to be useful in general.
- **Search Strategies**. We explored several different search strategies (DFS, random, time split and iterative deepening) over the *Plan* structure and provide evidence for iterative deepening and time split being the most effective for testing within a fixed time budget.

6.2 What IRULAN adds to a Haskell Programmer's Toolbox

Haskell programmers have many tools in their toolbox. As we discussed in Chapter 2, there are a variety of specialised tools designed to test programs, from the ubiquitous trio of property testing libraries (QuickCheck, SmallCheck and Lazy SmallCheck), to the symbolic executor Reach, and static analysis tools such as Catch and ESC/Haskell.

The property testing tools generate test data to apply to property functions to see if they can be made to return false. IRULAN generalises this idea to arbitrary functions, to see if they can be made to throw exceptions. In order to generate test data, IRULAN favours automation in the discovery of data sources (with the exception of constants) over the manual specification of test data generators required by QuickCheck or type-class expressed generators of SmallCheck and Lazy SmallCheck. While more sophisticated users can benefit from the ability to manually specify test data generators in the Check tools (e.g. the depth of test expressions can be controlled at a very fine grained level, or generation can be guided to e.g. only include sorted lists for a sorted list property), automatically inferring test data like IRULAN does is an advantage for the user in terms of ease of use for less sophisticated users, and imposes a lower cost for the testing of existing code. In the future it would be great to have a "best of both worlds" tool for dynamic crash testing and property checking that can use automatic inference, but also be fine tuned if necessary.

IRULAN also presents a new twist on more traditional unit testing tools through its regression testing extension. We have not seen any existing tools for Haskell that attempt to automatically snapshot the functionality of a library and then provide a way to automatically compare snapshots. IRULAN's way of doing this is particularly interesting as different snapshots can present inputs to a function that generalise each other (due to lazy evaluation). This form of testing moves beyond looking at a piece of code in isolation, but rather checking behaviour as the code evolves. It would be exciting to see other tools take this idea further.

Side effecting functions that have the *IO* type are effectively not tested by IRULAN, as reducing them to WHNF does not cause the side effect to happen. In contrast, the static analysis performed by Catch can handle *IO* by conservatively assuming an IO based function can return any value of its type. However there is a difference in philosophy between Catch and IRULAN: IRULAN attempts to see if there is a way in which the exported

functions from a library could be called that causes an error, Catch attempts to see if there is an input that could be provided to a program (through *IO* function return values) that create an exception.

Another tool with a different testing philosophy to Irulan is ESC/Haskell. Here, if the user specifies pre and post conditions on their Haskell functions, then the tool will check for each function that, assuming the precondition of a function is met, that it meets the obligations of the preconditions of the functions it calls, and assuming those functions' post conditions hold that the current function satisfies its own postcondition. Irulan does not feature a specific way to specify pre or post conditions, and so will likely generate test inputs that violate any implicit preconditions a function may have. However if the user decides to make these preconditions explicit through checks in the code that throw known exceptions, then they can make filtering out precondition failures easier by just discarding those counter-examples. This has the advantage that in regression testing mode those counter-examples can still be kept and if any change in behaviour occurs (e.g. test cases now hit precondition failures when they previously returned results) then Irulan can highlight that as something for the programmer to be aware of.

6.3 Applications and Future Work

6.3.1 More Tools for Haskell Programmers

The core algorithms in Irulan could be extended in many ways to make useful tools for Haskell programmers. At its core, Irulan is a test expression generator, and as we have shown, this has applications beyond error finding to regression testing. Another application would be to use Irulan as a unit test generator. Presenting users with inputs and what their function currently outputs, the user could lock the outputs that are correct (to be checked they are still the same in future runs) and then work on making the function do the right thing for inputs that are currently incorrect.

The ability to index a map by unifying types, as per the *TypeMap* could also provide useful functionality for Haskell IDEs. For example, when presenting context-sensitive completions, a *TypeMap* could be used, pretty much as it is now in Irulan, to provide a list of suggestions of identifiers that can provide the current context's type.

The automatic inference of a support set in Irulan could also have other applications. For example, several Haskell libraries require boiler-plate type class instances to be written to apply them to several types. Concretely, Irulan's inferred support set could be used as another way to generate the *Arbitrary* and *Serial* type class instances needed by the Check family of tools.

6.3.2 Code Coverage

In the overview we outlined an experimental feature that records HPC statistics after each test expression is executed, and establishes a minimized set of test expressions that achieves the same coverage as the full

suite. This technique could potentially prune many redundant test expressions and give the user just the important cases to consider. It could also be used to group and inform filtering of expressions that cause errors. Unfortunately this approach currently has some limitations. For example, HPC information is only available for modules that are compiled with HPC enabled, so different control flow branches in dependant libraries may not get exercised. Whether this is or is not important is unknown, so devising an experiment to explore this space would be a good next step.

Taking the code coverage further, it would be interesting to explore the opportunities for improvement if code coverage information was used as part of the exploration strategy. This would mean moving from black box to grey box testing (analysing code coverage in the testing loop gives some information as to the implementation of functions, but not full source details). Of course, moving to full white box testing with knowledge of expression structure would give further opportunities for more precise testing.

6.3.3 Evaluation Metrics

During the evaluation section we have used Haskell Program Coverage (i.e. subexpression based code coverage) as a quantifiable evaluation point. While we have also used other metrics (such as number and types of errors found and how long and if counterexamples were found in the property benchmark), there are other techniques that could have been used.

One common strategy for investigating error detection in existing software is to perturb the source code, and seeing if the perturbations are detected by the tool being used. For example, conditional |if/then/else| branches could be swapped, or (for Haskell) the order of guards swapped, or identifiers exchanged for faulty versions. This technique, often called mutant generation, usually transforms a single piece of source code into several hundred or thousand mutants; mutants then detected by the tool (e.g. by finding introduced crashes) are then said to be "killed". One problem with mutant generation is that a lot of care must go into creating the mutants, to be aware what the mutants are really testing. For example in object oriented languages it is not appropriate to create mutants by negating boolean values on conditional constructs in order to test control flow coverage, as control flow in object oriented languages is usually encoded in dynamic dispatch on message receivers. For Haskell, higher order functions and polymorphic combinators may make meaningful mutant creation not straightforward.

We are currently unaware of any research or tools for Haskell to automatically create mutant test suites with well established properties, although it would be a very interesting area of research to pursue.

6.3.4 Support Set Inference

There are some limitations during the support set creation phase. For example, with type classes Irulan currently relies on GHC's dictionary passing transform to remove type classes from the set of concepts it has to understand. Unfortunately some type class specific information does remain and that can prove problematic. Orphan instances are not found by Irulan unless its support set happened to analyse a module that contains

them while chasing other dependencies. Solving this is mostly a software engineering issue, but may bring useful ideas for other work in this space.

There is another software engineering issue at work within the Haskell community. The Haskell ecosystem has been rapidly growing, and the notion of a *package* is emerging for the convenient grouping of modules that make up a library. As the definition of a package stabilises, it would be interesting to try and test the interface of a library at a slightly coarser grain than the level of identifiers exported from a module. Extending IRULAN with the knowledge of visible modules and hidden modules (an idiom introduced by the introduction of Haskell packages) and only presenting test cases that use visible identifiers from visible modules would be useful, but would also present interesting challenges for determining the appropriate identifiers to use.

Additionally, IRULAN currently allows the user to provide a constant pool for adding to the support set. This is useful for specifying specific built in values, such as *Int*s, *Char*s, that the set of support discovery mechanism cannot find as they are not inductively defined. We have some very experimental work for analysing the source code of any presented modules to identify constants automatically by looking at the constants used in the source code. However it would be interesting to take this work further, looking at deeper forms of source code analysis to work out constants that are important to reach all branches. This would require knowledge of theories (e.g. addition or solving inequalities) underlying *Int* or *Double* values, for example.

There is also the issue of generating higher order functions (HOFs). IRULAN can use existing functions in its support set as higher order functions. However other testing tools, such as SmallCheck and QuickCheck, actually synthesise HOFs by mapping from possible inputs to outputs. The ability to synthesise higher order functions would be valuable for testing a library before it is released to users (as a user could provide arbitrary functions), whereas IRULAN's ability to use existing functions is suitable for testing more closed systems where the HOF will typically be a provided function.

6.3.5 Referential Transparency

The idea of using referential transparency to avoid needing to re-execute some expressions, and to prune branches, seems very appealing. Our initial work with runtime caches to exploit this did not prove successful however, so some further focused work there would appear to be useful. Our experiments suggested that there are some simple optimisations that may help, for example not checking for redundant values when at the end of a *Plan* branch. There may be further ways for the *Plan* to give hints to the cache as to what types of expression to look up in certain places. There is also the design space of the implementation of the caches themselves that may mean some overheads can be reduced.

An issue relating to referential transparency is the ability to test functions that do *IO*. IRULAN assumes that all functions it tests are referentially transparent (the threaded cache in the *Plan* and the runtime caches exploit this). Since IRULAN is generating arbitrary test expressions and executing them, we decided not to try and add the ability to run *IO* expressions; IRULAN can and will try and evaluate values of type *IO* to a WHNF, but that does not mean the action gets executed, just that there is an action described by a WHNF. Adding the

ability to execute and test *IO* functions presents several challenges. For example, a sandbox or white/blacklist may need creating to ensure functions such as *deleteFile* aren't able to wipe data from the user's hard disk. Another issue would be whether *IO* values could be used as arguments to pure functions, and ensuring that the resulting expressions correctly capture that the *IO* action needs to be run before it can be passed into a pure function. There is also issues with *IO* computations spawning new threads or processes which would need to be monitored and correctly handled.

6.3.6 Haskell features

Throughout this work, we have assumed the Haskell functions being tested conformed to a rank one type system. We have also assumed the absence of many GHC extensions to the Haskell language (e.g. type and data families, GADTs, existential types). These assumptions were made to keep the core as simple and easy to develop as possible while still allowing an expressive testing tool to be developed. Extending Irulan's understanding of types to the full richness of GHC Core's would be an interesting exercise, however it would also pose challenges for mapping the more exotic generated expressions back to Haskell expressions that could be entered into a GHCi prompt.

There are a couple of less exotic Haskell features that we decided to ignore as features that would be nice to have in a polished tool, but add little to a research prototype. For example, data constructors can be declared using Haskell's record syntax. This allows the fields in a record to be named, and that name also acts as a Haskell function which extracts that field from the data constructor. Instead of creating explicit case expressions for data constructors which export their field accessors, Irulan could use those accessors instead. However, since case expressions work over any exported data constructor, we decided against adding extra complexity while the benefits of using selectors was being researched.

Another simple limitation in Irulan is that types declared using Haskell's **newtype** syntax cannot have case expressions constructed over them (as there is nothing to scrutinise at runtime). To do so would require extra checking to see if the case expression were over a **newtype**'d type, and if so during compilation not actually to construct the case expression. This would then lead into the interesting space of potential optimisation where an identical operation may get executed twice, even though it's represented by syntactically different expressions.

6.3.7 Parallelism and Concurrency

Irulan does not currently exploit parallelism to enable testing multiple expressions at the same time, however it would be easy to see how such an extension could be made to work. The explicit internal |Plan|, and the flexibility with which search strategies can be specified should make it straightforward to allow strategies that divide the |Plan| between multiple cores. Of course, getting the trade off between dividing work between cores and having enough work to make the overheads of managing multiple cores worthwhile would be an interesting space of optimisation, but for large enough spaces it should be easy to make work.

Haskell also makes it very easy to write pure, parallel programs (i.e. programs that run in parallel without using |IO|, though e.g. the |par| combinator). While we have not directly tested this, because the construct is pure, it should not cause a problem for IRULAN to test code that uses it, as the external interface of such code (i.e. what IRULAN sees), will still be pure.

However at the moment IRULAN has no support for testing explicitly concurrent programs. If support for testing *IO*-based expressions were added to IRULAN, then supporting *IO*-based concurrency would be a natural extension. In supporting this work, we could draw lessons from the published experiences of using QuickCheck like techniques in Erlang (from e.g. [CPS+09]). Here the authors note that the testing library should also be able to interact with the scheduler of the program, in order to try and force more of the unlikely thread interactions that would not normally occur. With the scheduler under control of the testing tool, traces and repeatable test cases can also be produced.

6.4 Final Words

As long as humans are allowed to program, they will make mistakes, and there will be bugs in software. To combat this, many techniques (testing, verification, peer-review) employ abstraction, repetition, and sanity checking to help programmers make sure that the code they write is the code they mean, and that what programs do is what the programmer intends.

Pure functional programming lends itself towards automation of some of these techniques due to its simple underlying core; but it can still be complicated due to the richness of the abstractions available. In this work we have investigated testing in a pure functional language, and focusing on producing test cases with their results (in the form of errors they throw) that we can get (from the programmer's point of view) almost for free. Our results are encouraging, and we have many ideas for how to turn our techniques into even more useful tools for Haskell programmers.

6.5 Finding the Lazy Programmer's Bugs

The full sources for the prototype implementation of our techniques, IRULAN, together with our experimental evaluations, is freely available, at `http://www.doc.ic.ac.uk/~tora/irulan`.

Appendix A

Full nofib Results

In Figure A.1 we present all code coverage results from the 8 different configurations of IRULAN we ran on the programs in the spectral suite. In Figure A.2 we present all code coverage results from the 8 different configurations of IRULAN we ran on the programs in the real suite. For both graphs, the results where case expressions were enabled are prefixed with a 'C', and disabled 'N'. The times correspond to how long IRULAN was run, using iterative deepening, on each of the modules in the program.

Figure A.1 All coverage results for spectral. 'C' bars have case expressions enabled, 'N', disabled. The times are runtime per module per program.

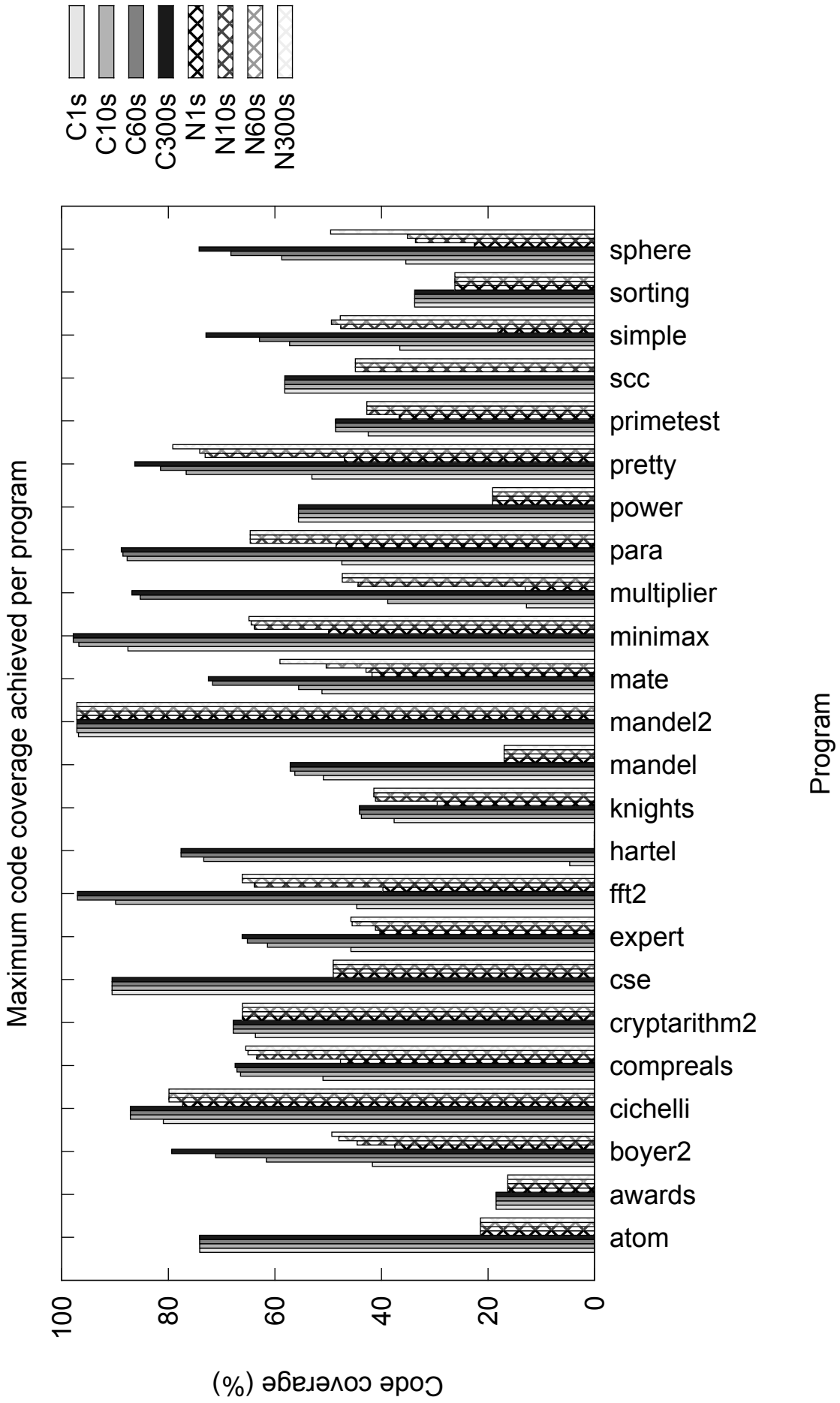
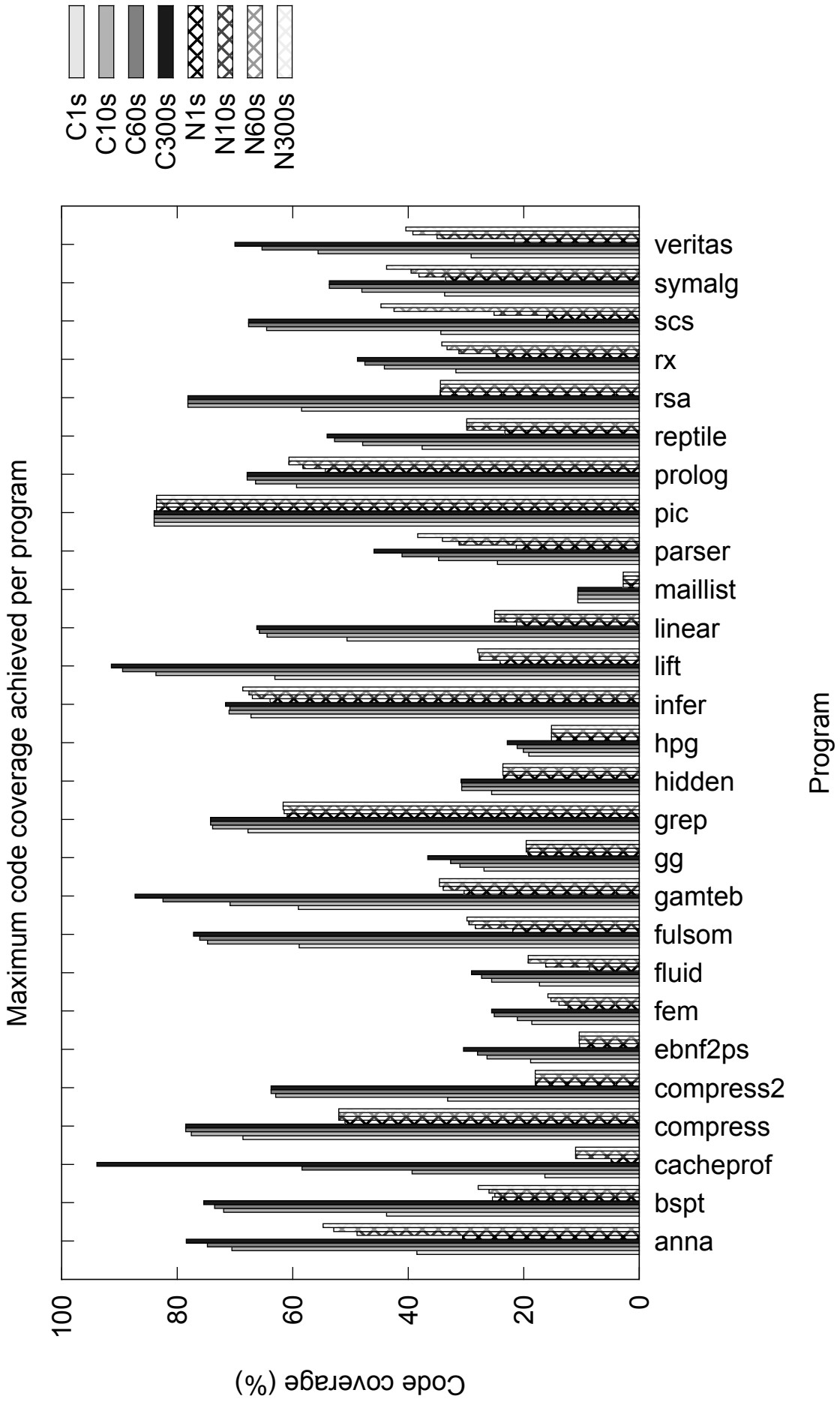


Figure A.2 All coverage results for real. 'C' bars have case expressions enabled, 'N', disabled. The times are runtime per module per program.



Bibliography

[ACE] Tristan Allwood, Cristian Cadar, and Susan Eisenbach. Irulan source download and raw results. `http://www.doc.ic.ac.uk/~tora/irulan/`.

[AE] Tristan Allwood and Susan Eisenbach. JavaFeather, code coverage output and FJ implementations. `http://www.doc.ic.ac.uk/~tora/JavaFeather/`.

[AE09] Tristan Allwood and Susan Eisenbach. Tickling Java with a Feather. *Electronic Notes in Theoretical Computer Science*, 238:3–16, October 2009.

[AEH00] Sergio Antoy, Rachid Echahed, and Michael Hanus. A needed narrowing strategy. *JACM: Journal of the ACM*, 47:776–822, July 2000.

[AP98] Nancy An and Young Park. A structured approach to retrieving functions by types. In *Proceedings of the third ACM SIGPLAN international conference on Functional programming*, ICFP '98, pages 344–, New York, NY, USA, 1998. ACM.

[APJE09] Tristan Allwood, Simon Peyton Jones, and Susan Eisenbach. Finding the needle: stack traces for GHC. In *Proceedings of the 2nd ACM SIGPLAN symposium on Haskell*, Haskell '09, pages 129–140, New York, NY, USA, 2009. ACM.

[BKM02] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. Korat: automated testing based on Java predicates. In *Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, ISSTA '02, pages 123–133, New York, NY, USA, 2002. ACM.

[CDE08] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association.

[CF08] Jan Christiansen and Sebastian Fischer. EasyCheck: test data for free. In *Proceedings of the 9th international conference on Functional and logic programming*, FLOPS'08, pages 322–336, Berlin, Heidelberg, 2008. Springer-Verlag.

[CGE08] Dave Cunningham, Khilan Gudka, and Susan Eisenbach. Keep off the grass: locking the right path for atomicity. In *Proceedings of the Joint European Conferences on Theory and Practice of Software 17th international conference on Compiler construction*, CC'08/ETAPS'08, pages 276–290, Berlin, Heidelberg, 2008. Springer-Verlag.

[CGH00] Michelle Cope, Ian P. Gent, and Kevin Hammond. Parallel heuristic search in haskell. In *Scottish Functional Programming Workshop*, pages 65–76, 2000.

[CH00] Koen Claessen and John Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, ICFP '00, pages 268–279, New York, NY, USA, 2000. ACM.

[CM95] Richard H. Connelly and F. Lockwood Morris. A generalization of the trie data structure. *Mathematical Structures in Computer Science*, 5(03):381–418, 1995.

[Coo72] D. C. Cooper. Theorem proving in arithmetic without multiplication. In *Machine Intelligence*, volume 7, pages 91–99, New York, 1972. American Elsevier.

[CPS+09] Koen Claessen, Michal Palka, Nicholas Smallbone, John Hughes, Hans Svensson, Thomas Arts, and Ulf Wiger. Finding race conditions in Erlang with QuickCheck and PULSE. In *Proceedings of the 14th ACM SIGPLAN international conference on Functional programming*, ICFP '09, pages 149–160, New York, NY, USA, 2009. ACM.

[CRW01] Olaf Chitil, Colin Runciman, and Malcolm Wallace. Freja, Hat and Hood - A Comparative Evaluation of Three Systems for Tracing and Debugging Lazy Functional Programs. In *Selected Papers from the 12th International Workshop on Implementation of Functional Languages*, IFL '00, pages 176–193, London, UK, 2001. Springer-Verlag.

[DMB08] Leonardo De Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *Proceedings of the Theory and practice of software, 14th international conference on Tools and algorithms for the construction and analysis of systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[FK07] Sebastian Fischer and Herbert Kuchen. Systematic generation of glass-box test cases for functional logic programs. In *Proceedings of the 9th ACM SIGPLAN international conference on Principles and practice of declarative programming*, PPDP '07, pages 63–74, New York, NY, USA, 2007. ACM.

[Gab85] Richard P. Gabriel. *Performance and evaluation of LISP systems.* Massachusetts Institute of Technology, Cambridge, MA, USA, 1985.

[Gil01] Andy Gill. Debugging Haskell by Observing Intermediate Data Structures. *Electronic Notes in Theoretical Computer Science*, 41(1):1 – 1, 2001. 2000 ACM SIGPLAN Haskell Workshop (Satellite Event of PLI 2000).

[GLM08] Patrice Godefroid, Michael Y. Levin, and David A Molnar. Automated Whitebox Fuzz Testing. In *Network Distributed Security Symposium (NDSS)*. Internet Society, 2008.

[GR07] Andy Gill and Colin Runciman. Haskell program coverage. In *Proceedings of the ACM SIGPLAN workshop on Haskell*, Haskell '07, pages 1–12, New York, NY, USA, 2007. ACM.

[HHPJW96] Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones, and Philip L. Wadler. Type classes in Haskell. *ACM transactions on Programming languages and systems (TOPLAS)*, 18:109–138, March 1996.

[IPW99] Atshushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight Java: a minimal core calculus for Java and GJ. In *Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '99, pages 132–146, New York, NY, USA, 1999. ACM.

[Jac06] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis.* MIT Press, 2006.

[JW91] Simon Peyton Jones and Philip Wadler. A static semantics for haskell. Draft paper, Glasgow, 91.

[Kin76] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19:385–394, July 1976.

[Kor96] Bogdan Korel. Automated test data generation for programs with procedures. In *Proceedings of the 1996 ACM SIGSOFT international symposium on Software testing and analysis*, ISSTA '96, pages 209–215, New York, NY, USA, 1996. ACM.

[Lin07] Fredrik Lindblad. Property Directed Generation of First-Order Test Data. In *The Eigth Symposium on Trends in Functional Programming*, pages 105–123, 2007.

[MFS90] Barton P. Miller, Louis Fredriksen, and Bryan So. An empirical study of the reliability of UNIX utilities. *Communications of the Association for Computing Machinery (CACM)*, 33:32–44, December 1990.

[Mic] Sun Microsystems. OpenJDK. `openjdk.java.net`.

[MIPG07] Simon Marlow, José Iborra, Bernard Pope, and Andy Gill. A lightweight interactive debugger for haskell. In *Proceedings of the ACM SIGPLAN workshop on Haskell*, Haskell '07, pages 13–24, New York, NY, USA, 2007. ACM.

[Mit08] Neil Mitchell. Hoogle overview. *The Monad.Reader*, (12):27–35, November 2008.

[MR08] Neil Mitchell and Colin Runciman. Not all patterns, but enough: an automatic verifier for partial but sufficient pattern matching. In *Proceedings of the first ACM SIGPLAN symposium on Haskell*, Haskell '08, pages 49–60, New York, NY, USA, 2008. ACM.

[Nil98] Henrik Nilsson. *Declarative Debugging for Lazy Functional Languages.* PhD thesis, January 26 1998.

[NR07] Matthew Naylor and Colin Runciman. Finding Inputs that Reach a Target Expression. In *Proceedings of the Seventh IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 133–142, Washington, DC, USA, 2007. IEEE Computer Society.

[Par93] Will Partain. The nofib Benchmark Suite of Haskell Programs. In *Proceedings of the 1992 Glasgow Workshop on Functional Programming*, pages 195–202, London, UK, 1993. Springer-Verlag.

[Pey87] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice–Hall, 1987.

[Rit89] Mikael Rittri. Using types as search keys in function libraries. In *Proceedings of the fourth international conference on Functional programming languages and computer architecture*, FPCA '89, pages 174–183, New York, NY, USA, 1989. ACM.

[RNL08] Colin Runciman, Matthew Naylor, and Fredrik Lindblad. SmallCheck and Lazy SmallCheck: Automatic exhaustive testing for small values. In *Proceedings of the first ACM SIGPLAN Symposium on Haskell*, pages 37–48, 2008.

[RT89] Colin Runciman and Ian Toyn. Retrieving re-usable software components by polymorphic type. In *Proceedings of the fourth international conference on Functional programming languages and computer architecture*, FPCA '89, pages 166–173, New York, NY, USA, 1989. ACM.

[SCJD07] Martin Sulzmann, Manuel M. T. Chakravarty, Simon Peyton Jones, and Kevin Donnelly. System f with type equality coercions. In *Proceedings of the 2007 ACM SIGPLAN international workshop on Types in languages design and implementation*, TLDI '07, pages 53–66, New York, NY, USA, 2007. ACM.

[Sil07] Josep Silva. A comparative study of algorithmic debugging strategies. In *Proceedings of the 16th international conference on Logic-based program synthesis and transformation*, LOPSTR'06, pages 143–159, Berlin, Heidelberg, 2007. Springer-Verlag.

[SPJ95] Patrick M. Sansom and Simon L. Peyton Jones. Time and space profiling for non-strict, higher-order functional languages. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '95, pages 355–366, New York, NY, USA, 1995. ACM.

[TDH08] Nikolai Tillmann and Jonathan De Halleux. Pex: white box test generation for .net. In *Proceedings of the 2nd international conference on Tests and proofs*, TAP'08, pages 134–153, Berlin, Heidelberg, 2008. Springer-Verlag.

[Tur79] D. A. Turner. A New Implementation Technique for Applicative Languages. *Software: Practice and Experience*, 9:31–49, 1979.

[VPK04] Willem Visser, Corina S. Păsăreanu, and Sarfraz Khurshid. Test input generation with java pathfinder. In *Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, ISSTA '04, pages 97–107, New York, NY, USA, 2004. ACM.

[WCBR01] Malcolm Wallace, Olaf Chitil, Thorsten Brehm, and Colin Runciman. Multiple-view tracing for Haskell: a new Hat. In Ralf Hinze, editor, *Preliminary Proceedings of the 2001 ACM SIGPLAN Haskell Workshop*, pages 151–170, Firenze, Italy, September 2001. Universiteit Utrecht UU-CS-2001-23. Final proceedings to appear in ENTCS 59(2).

[Xu06] Dana N. Xu. Extended static checking for haskell. In *Proceedings of the 2006 ACM SIGPLAN workshop on Haskell*, Haskell '06, pages 48–59, New York, NY, USA, 2006. ACM.