Narrowing in on Property-Based Testing

Jonathan Fowler

Thesis submitted to the University of Nottingham

for the degree of Doctor of Philosophy, 2018

Abstract

Narrowing is one of the primary methods for implementing functional logic programming languages. Property-based testing is an automatic approach to assuring the correctness of software systems. In recent years, a number of systems have been developed that seek to apply the benefits of narrowing in the area of property-based testing. This thesis considers two limitations with these systems. First of all, most of the existing narrowing-based testing tools have focused on practical issues, and lack supporting theory. And secondly, these tools typically only perform well on properties that have particular forms. We address these limitations by developing an approach to narrowing that is both practical and principled, and demonstrate how this can be used to expand the range of properties that can be automatically tested using a narrowing-based approach.

Acknowledgements

This thesis has been a long time coming. Both in my life, as I have wanted to do a PhD for many years but was unsure of the topic for a long time, and in its final conception as I have always struggled with writing and creating this thesis has been an arduous, although often rewarding, task. I have relied on the help and support of many people during that time, and I would like to thank a few of them here.

To start, I would like to express my gratitude to my supervisor Graham Hutton for first taking me as a student and then his invaluable advice during my PhD. He has always been supportive and helpful even when I haven't been the most diligent student. Both his enthusiasm for functional programming and his constant desire to capture the essence of a problem had a big impact on me, which I hope is appreciable in this thesis.

I thank Colin Runciman for introducing me to the use of narrowing in property-based testing and then later for his insightful feedback examining this thesis. Many thanks to my other examiner Venanzio Capretta, also for his feedback, and additionally for inviting me to contribute to his research which was an enjoyable aside to the work in this PhD.

Thanks to everyone in the FP lab at Nottingham for making my time there interesting and teaching me a lot about functional programming and type theory. They are Thorsten Altenkirch, Henrik Nilsson, Nicolai Kraus, Paolo Capriotti, Gabe Dijkstra, Ivan Perez Dominguez, Jan Bracker, Jennifer Hackett, Ambrus Kaposi, Jakob von Raumer, Martin Handley, Guerric Chupin and Colm Baston,

I thank my parents, Elizabeth and Mark Fowler, for their constant encouragement and the opportunities which made doing this PhD possible.

Finally I would like to express my love and gratitude to my partner Tamara Fitters. You have always been patient and supporting, even when my writing period extended beyond the birth of our son Nate. I could not have done it without you.

Contents

Chapter 1

Introduction

Narrowing is a commonly used evaluation strategy in functional-logic programming languages which combines the functional approach to programming with logic programming's ability to compute with incomplete information. In logic programming, incomplete information within a program is represented by free variables and logical features can then be used to subject the result of a program to a constraint. The problem is then to find bindings for the free variables in the program which satisfy the constraint. Narrowing is a strategy to achieve this, which evaluates the program while delaying the instantiation of free variables until they are required to continue. This strategy has proved successful, performing far better than the basic strategy in which free variables are guessed blindly [6], and has been used in many functional-logic languages.

Property-based testing, popularised by systems such as QuickCheck [16], is an automated approach to testing in which a program is validated against a specification. In most tools, the specification consists of properties written as programs outputting Boolean values. Input data is generated randomly or systematically, and the program is executed in an attempt to find a counterexample. Property-based testing can offer significant advantages over traditional testing methods, such as unit testing. Particularly, a programmer is often able to achieve more thorough testing with less effort as test cases can be automatically generated and evaluated.

The application of narrowing to property-based testing is a natural fit. In property-based testing we are attempting to find a counterexample to a property which can be written directly as a functional-logic program. Furthermore, traditional property-based testing can be viewed as taking a blind guess – an approach which narrowing had already been shown to improve upon. It is no surprise then that research into testing with narrowing inspired strategies is an active area of research with tools being developed to tackle a range of problems related to property-based testing. These problems include the generation of test cases [15, 32, 33], end-to-end property-based testing [12, 13, 48] and achieving program coverage in testing [23, 38].

This thesis builds on research on property-based testing using narrowing on two frontiers. First of all, we address the lack of a formal basis for most tools, which have generally focused on the practical aspects of implementation and benchmarking. And secondly, we expand the scope of properties which can be tested both effectively and directly with narrowing. More precisely, the thesis makes the following contributions:

- We formalise narrowing as an extension to a functional programming language. The use of an underlying functional language simplifies the formalisation and is particularly suited for the formalisation of narrowing property-based testing tools, many of which are based on functional languages.

- We develop a narrowing evaluation strategy based on the concept of overlapping patterns, which increases the scope of properties which are testable in an effective manner by narrowing. This work builds on the formalisation noted above.

- We implement a narrowing property-based testing tool for a functional programming language that is a subset of Haskell. The tool supports random and enumerative testing along with different narrowing strategies including using overlapping patterns.

- We evaluate the implementation using a number of case studies. We compare a basic property-based testing strategy with traditional narrowing and overlapping narrowing strategies, and measure several metrics to give insight into the differences in performance.

Parts of the thesis are based on two published papers, with the author of the thesis as the lead author for each paper:

- Jonathan Fowler and Graham Hutton, "Towards a theory of Reach", in the *Proceedings of the 16th International Symposium on trends in Functional Programming*, 2015. [26]

- Jonathan Fowler and Graham Hutton, "Failing Faster: Overlapping Patterns for Property-Based Testing", in the *Proceedings of the 19th International Symposium on Practical Aspects of Declarative Languages*, 2017. [27]

The thesis is structured as follows:

Chapter 2 provides the necessary background to the theory used in the thesis. We introduce narrowing, property-based testing, existing narrowing property-based testing tools, and operational semantics.

Chapter 3 lays the foundation for a theory of narrowing as an extension to a functional programming language. We use a minimal language that isn't suitable for actual programming but provides enough features to demonstrate the essence of narrowing. We demonstrate the practical application of our theory in the area of property-based testing.

Chapter 4 develops an implementation of a property-based testing tool using narrowing. Two variants of the narrowing tool are benchmarked against a basic tool on a variety of properties, and we identify some of the features of properties that can influence performance.

Chapter 5 builds on the formalisation of narrowing in chapter 3 by adding overlapping patterns to the language. This new feature improves performance when defining conditions using conjunctions, and supports the definition of bespoke size constraints, which we illustrate in two case studies.

Chapter 6 extends the implementation from chapter 4 with overlapping patterns and evaluates the performance against the original narrowing tool. The new implementation gives improved performance and a better distributuion of test cases on certain forms of properties.

The thesis is aimed at a reader who is familiar with the basics of functional programming in a language such as Haskell (in particular, the use of strong typing, recursive datatypes and functions, and inductive proofs), but we don't assume any specialised knowledge in areas such as logic programming, narrowing, property-based testing, or program semantics.

Chapter 2

Background

In this chapter we provide some background on a range of different topics that are central to the thesis, namely functional-logic programming (section 2.1), property-based testing (section 2.2), research into combining these two notions (section 2.3), and finally, operational semantics (section 2.4). As noted at the end of the previous chapter, the reader is assumed to be familiar with the basics of functional programming.

2.1 Functional-Logic Programming

In this section we introduce functional-logic programming and give a simple example in one of the principal languages, Curry [29]. We use this example to demonstrate the idea of narrowing and then discuss a number of additional topics from the research literature.

Functional-logic programming aims to combine the features and advantages of the two declarative programming paradigms from which its name derives. There have been two main approaches to this union: beginning with a logic language and adding functional programming features, as in the Mercury programming language [50]; or conversely, beginning with a functional language and adding logic programming features, as in the Curry programming language [29]. This thesis takes inspiration mainly from the latter approach.

The Curry language offers a functional programming language with Haskell-like syntax which also includes logic programming features, the most important of which are two small but semantically powerful additions. The first and most significant addition is *free variables*, more commonly called logic variables in logic programming, which can be instantiated to any value of their type. The second is inspired by the observation that most of the functionality deriving from the Horn clauses of logic programming can be encoded using *partial pattern matching* (along with free variables). Therefore, Curry adds partial pattern matching as a language feature.

We demonstrate these language features with an example. We consider a datatype of people, and a partial function which defines the mother of some of the people:

$$\begin{aligned}
&\textbf{data}\ \mathit{Person} = \mathit{Alice} \mid \mathit{James} \mid \mathit{Liz} \mid \mathit{Eve}\\
&\mathit{mother} :: \mathit{Person} \to \mathit{Person}\\
&\mathit{mother}\ \mathit{Alice} = \mathit{Liz}\\
&\mathit{mother}\ \mathit{James} = \mathit{Eve}\\
&\mathit{mother}\ \mathit{Liz} = \mathit{Eve}
\end{aligned}$$

Suppose that we wish to compute who the children of *Eve* are, i.e. any x which satisfies $\mathit{mother}\ x \equiv \mathit{Eve}$. We can do this in a simple manner using functional-logic programming. First, we define a helper function *when*:

$$\begin{aligned}
&\mathit{when} :: \mathit{Bool} \to a \to a\\
&\mathit{when}\ \texttt{True}\ \ a = a
\end{aligned}$$

The partial function *when* takes a condition and a value and returns the value only when the condition is true. Note that when the condition is false, rather then giving a runtime error as in a functional language such as Haskell, in Curry failure of a pattern match results in backtracking. Now utilising a free variable we can define the *children* of a mother:

$$\begin{aligned}&\mathit{children} :: \mathit{Person} \to \mathit{Person}\\&\mathit{children}\ a = \mathbf{let}\ x\ \mathit{free}\ \mathbf{in}\ \mathit{when}\ (\mathit{mother}\ x \equiv a)\ x\end{aligned}$$

For example, *children Eve* will produce all the children of *Eve*. In particular, the function will create a free variable x, and only return a value when this free variable is bound to a person whose mother is *Eve*. As there are two possible solutions, *James* or *Liz*, the function is non-deterministic. To compute these solutions Curry uses a narrowing strategy.

Narrowing [4] is an evaluation strategy which refines the values of free variables as and when they are needed. A substitution is used to store the values of these variables and is extended each time a new free variable is introduced. We demonstrate how this works on our example. Beginning with the empty substitution $\{\,\}$ with no variables, we can expand the expression *children Eve* (without affecting the substitution) by simply inlining the body of the definition of *children*:

$$\begin{aligned}1)\quad &\{\,\}\\&\mathit{children}\ \mathit{Eve}\\\to\ &\mathbf{let}\ x\ \mathit{free}\ \mathbf{in}\ \mathit{when}\ (\mathit{mother}\ x \equiv \mathit{Eve})\ x\end{aligned}$$

The next step is to remove the let expression by introducing a free variable x into the substitution (which is initially bound to itself):

$$\begin{aligned}2)\quad &\{x \mapsto x\}\\&\mathit{when}\ (\mathit{mother}\ x \equiv \mathit{Eve})\ x\end{aligned}$$

Evaluating the *when* function requires a pattern match on its first argument. In order to evaluate the argument, $mother\ x \equiv a$, we need the value of x. First we refine the value of x to *Alice* and reduce:

$$\begin{aligned}
3\ a)\quad & \{x \mapsto Alice\} \\
& when\ (mother\ Alice \equiv Eve)\ Alice \\
\to\ & when\ (Liz \equiv Eve)\ Alice \\
\to\ & when\ \texttt{False}\ Alice \\
\to\ & \bot
\end{aligned}$$

Alice is not a child of *Liz* and consequently the pattern match for *when* fails, which results in a failed state $\bot$. The computation backtracks and an alternative binding is considered. This time x is refined to *James*:

$$\begin{aligned}
3\ b)\quad & \{x \mapsto James\} \\
& when\ (mother\ James \equiv Eve)\ James \\
\to\ & when\ \texttt{True}\ James \\
\to\ & James
\end{aligned}$$

James is a child of *Liz* and the computation ends successfully. If we were enumerating solutions we could go on by trying the other possible bindings for x and find *Liz* as another child of *Eve*. This concludes the example.

We note that although the presentation of narrowing given is consistent with how Curry operates, the style is representative of the approach in this thesis as opposed to a more traditional interpretation [28]. A further example of narrowing is given in chapter 3, where we show the evaluation of a property-based testing example and demonstrate how it is beneficial in this setting.

2.1.1 Further Functional-Logic Topics

Below we briefly discuss some additional features and evaluation strategies that are commonly found in functional logic languages.

Residuation

Residuation [28] is an alternative evaluation strategy for functional-logic programs that can either be used on its own or in combination with narrowing. With narrowing, when we encounter a free variable it is instantiated with a possible value in order to make progress. In contrast, with residuation when we encounter a free variable we suspend evaluation of the current expression; evaluation then continues with the next expression to be considered, with the suspended expression being resumed as and when its free variable becomes bound during subsequent evaluations.

This approach can be beneficial as it delays the binding of a free variable, which may allow more evaluation to be shared between different instantiations of the variable. However, it suffers from two related drawbacks. First of all, suspended evaluations are not guaranteed to be restarted, and hence evaluation may fail to produce a result. And secondly, to work effectively it generally requires additional annotations from the programmer to control where residuation should be used, to ensure that free variables eventually become bound. For these reasons, in this thesis we restricted our attention to a narrowing-based evaluation strategy.

Equational Constraints

Equational constraints [6, 29] are an additional feature in languages such as Curry, which can be considered as an optimisation of the standard equality operator in programming languages. By way of example, consider the list equality $[Alice, x] \equiv [y, z]$, in which each of the variables x, y and z is free. Such an equality can be solved by instantiating the variable y to *Alice*, and the variables x and z to any identical value of the *Person* type, which gives four possible solutions. In contrast, the equational constraint $[Alice, x] =:= [y, z]$ can be solved by instantiating y to *Alice* as before, and simply binding x to z (or equivalently, vice versa), reducing the number of possible solutions to just one by delaying the instantiation of x and z.

This facility can sometimes be useful. For example, pattern matching on functions can be desugared into equational constraints [6]. However, the use of such constraints necessitates moving into the realms of logic programming, whereas our focus is on the use of logic programming techniques to test properties expressed in the functional paradigm. Nonetheless, it would be interesting to explore the addition of equational constraints to our theoretical and practical developments.

Non-Deterministic Pattern Matching

In some functional logic languages, including Curry, pattern matching can also be non-deterministic. That is, it may produce different result values for the same argument value. For example, using this idea, the *children* function from the previous section could also be expressed as:

$$\begin{aligned}
&children :: Person \to Person\\
&children\ Liz = Alice\\
&children\ Eve = James\\
&children\ Eve = Liz
\end{aligned}$$

Note that there are two alternatives for matching *Eve*. In Curry, pattern matching is evaluated independently of the order in which the alternatives appear, so either choice is possible. We do not consider this form of pattern matching in the thesis for two reasons. Firstly, the syntax that is used conflicts with the form of pattern matching developed in chapter 5, where we consider overlapping pattern matching that is deterministic. And secondly, the functionality of non-deterministic pattern matching can already be obtained using free variables and traditional pattern matching [3] (and also the converse [5], free variables can be modelled using non-deterministic pattern matching).

2.2 Property-Based Testing

In this section, we introduce property-based testing through a QuickCheck example, and then discuss further aspects including properties with preconditions, the distribution of test cases, and number of related research areas.

Any testing tool must have a criterion by which it determines whether a particular test has passed or failed. In property-based testing, this criterion is given by a *specification* – a set of properties the program should satisfy. As an example, we consider a specification of the *reverse* function, which we have taken from the original QuickCheck paper [16]:

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

Any *reverse* function must satisfy all three of these properties. In order to use a property as an automated criterion for testing, we must convert it into a program which determines whether the property fails or succeeds on a given input. In QuickCheck, properties are written in Haskell, and the specification above can be written as follows:

$$\begin{aligned}
&\mathit{propRevOne}\ x &&= \mathit{reverse}\ [x] \equiv [x] \\
&\mathit{propRevApp}\ \mathit{xs}\ \mathit{ys} &&= \mathit{reverse}\ (\mathit{xs} +\!\!+\ \mathit{ys}) \equiv \mathit{reverse}\ \mathit{ys} +\!\!+\ \mathit{reverse}\ \mathit{xs} \\
&\mathit{propRevRev}\ \mathit{xs} &&= \mathit{reverse}\ (\mathit{reverse}\ \mathit{xs}) \equiv \mathit{xs}
\end{aligned}$$

For example, the last property states that for any input list *xs*, reversing the list twice should return the original list. Each property can be tested by simply supplying an input and then evaluating the property. Therefore, testing becomes a problem of generating inputs. In this case, for the top property we have to generate an element — for example, we could choose an integer — and for the other two properties we need to generate lists of elements.

QuickCheck generates inputs randomly and provides a library of combinators to aid in doing so. For example, a generator for a list of integers can be defined in QuickCheck as follows:

```
generateLists :: Gen [Int]
generateLists = frequency [
  (1, return []),
  (4, (:) <$> elements [1 .. 10] <*> generateLists)]
```

The *frequency* and *elements* combinators choose a random value from a list. The *frequency* combinator does so according to the weights given in the list, and so this generator creates an empty list 20% of the time and 80% of the time adds an element, randomly chosen from 1 to 10 using the *elements* combinator, before being called recursively.

We now have all the parts to run tests on each of the properties. For example, running the following QuickCheck test validates *propRevRev*:

$> quickCheck\ (forAll\ generateLists\ propRevRev)$

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

QuickCheck generates one hundred random lists and finds each satisfies the property. The tool can test many examples using only a specification and a generator. In fact, in this case we do not even need to define the generator and can instead use QuickCheck's standard generator for a list of integers:

$> quickCheck\ (propRevRev :: [Int] \to Bool)$

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

2.2.1 Preconditions

Many properties only hold if the input is of a certain form. For example, consider a binary search tree, the datatype of which could be represented as follows:

$\mathbf{data}\ Tree\ a = Leaf \mid Node\ Tree\ a\ Tree$

There is an implicit assumption that the elements of such a tree are ordered, otherwise most functions defined on the tree will not work correctly. For example, a *member* function which determines whether a given element is in a tree should satisfy something akin to the following property:

$propMember :: Int \to Tree\ Int \to Bool$

$propMember\ a\ t\ = \ member\ a\ t \equiv any\ (\equiv a)\ t$

This property states that an integer is a *member* of a tree if *any* element in the tree is equal to that integer. However, a typical definition of a *member* function will not satisfy this property, because for efficiency the function will typically exploit the ordering of the tree. Hence, we need to add a precondition that the tree is *ordered*:

$$\mathit{propMember}\ a\ t \;=\; \mathit{ordered}\ t \implies \mathit{member}\ a\ t \equiv \mathit{any}\ (\equiv a)\ t$$

The implication operator used here is not the traditional one. Importantly, it separates those inputs which fail the precondition, which are invalid test cases, from those that satisfy the precondition and the property (whereas with a traditional implication these both evaluate to true). When reporting the results of a test only the valid test cases are counted.

However, testing the above property with a standard generator for trees still won't be effective. This is because filtering the test cases alters the distribution – randomly generated large trees are much less likely to be ordered, and therefore there is skew towards smaller trees. One approach to resolving this issue is to write a custom generator for ordered trees. In this thesis we will explore an alternative approach, based upon generating ordered trees directly from the *ordered* precondition.

2.2.2 Distribution of Test Cases

One of the major decisions in property-based testing is choosing how test cases are distributed. Most tools either generate test cases by enumeration or randomly. We discuss the variations, advantages and disadvantages of each in terms of the resulting distributions:

Enumeration

In enumerative testing, all inputs are tested up to a size limit. Runciman et al. [48] motivate this approach with the observation that if a program fails its specification then "it almost always fails in some simple case". And furthermore, they also observe the contraposition “if a program does not fail in any simple case, it hardly ever fails in any case”. Therefore, enumerating and testing the simple cases should offer a reasonable assurance that the property is satisfied.

This approach also has the benefit that if a counterexample is found, it will find the smallest counterexample as all inputs up to a certain depth are considered. Preconditions are also of less concern than in random testing, as invalid test cases can be filtered out without impacting the distribution. The primary disadvantage of enumeration is that the number of test cases often increases exponentially with the size limit, meaning that it is often difficult to enumerate beyond a relatively small size of input. This problem is often apparent with the traditional method of limiting the depth of constructors, however recent research [1] has shown that limiting the number of constructors is generally more effective and works well in practice. Enumeration is used in a variety of property-based testing tools [33, 38, 48, 21].

Random

In random testing, as the name suggests the basic idea is that inputs are generated according to a random distribution. The types of distribution can be broadly broken down into two categories: uniform and non-uniform. In uniform distributions, test cases are selected with equal probability from a bounded selection. In practice, however, this can be difficult to achieve in an efficient manner, because even calculating the number of test cases within the given bound can be problematic, which in turn makes calculating the frequency of a test case difficult.

In non-uniform distributions, test cases are selected based on user-defined probabilities. For instance, in the *generateLists* example from earlier in this chapter, we assigned different weights to the two constructors for lists, resulting in a non-uniform distribution of test cases. The primary advantage of this approach is that it is straightforward to implement. The disadvantage is that it can be difficult to select appropriate weights to ensure that the resulting distribution gives a suitable test coverage.

2.2.3 Related Areas

In this section, we give a brief overview of two further research areas that are not directly used in the thesis but may be useful to readers who are interested in applying our results in their own work.

Shrinking

After finding a counterexample to a property, the program in question will need debugging. The size of the counterexample will affect the ease of this process – the trace of a program run on a small counterexample will generally be small and the error should be easy to spot. However, property-based testing, particularly with random generation, can produce large counterexamples. Shrinking is a process to reduce the size of these counterexamples in order to simplify debugging. In QuickCheck, shrinking can be achieved by defining a shrinking function [30, 14], which given an input, produces a list of similar but smaller inputs. For example, a shrinking function for a list of integers could be defined as follows:

$$\begin{aligned}
&\mathit{shrinkList} :: [\mathit{Int}] \to [[\mathit{Int}]]\\
&\mathit{shrinkList}\ [\,] = [\,]\\
&\mathit{shrinkList}\ (a : l) = [l]\\
&\qquad\qquad\qquad +\!\!+\ [a : l' \mid l' \leftarrow \mathit{shrinkList}\ l]\\
&\qquad\qquad\qquad +\!\!+\ [a' : l \mid a' \leftarrow \mathit{shrinkInt}\ a]
\end{aligned}$$

That is, if the list is empty, then the input is already minimal. Otherwise, for a non-empty list there are three possibilities: remove the head of the list, retain the head and shrink the tail of the list, or shrink the head of the list (using a function to shrink integers) and retain the tail.

Given such a shrink function, when QuickCheck finds a counterexample it also checks all shrinkings of this value, and only reports the counterexample when it cannot be shrunk further. Note that the result is not necessarily the minimal counterexample, but in practice this is often the case.

Generating and simplifying specifications

Property-based testing requires a programmer to create specifications for their programs. To aid a programmer in this task, a number of tools have been constructed to assist in the automated discovery of specifications. For example, QuickSpec [18] and Speculate [10] attempt to discover specifications by enumerating possible properties they may satisfy, and then using a property-based testing methodology to either discard or accept the properties. HipSpec [17] takes this process further by trying to construct formal proofs of the properties that are generated.

Another related area is attempting to simplify specifications, and checking their completness. For example, in the *reverse* specification given on page 11, the third property is redundant as it is implied by the first two properties, and furthermore, the first two properties uniquely characterise *reverse*, i.e. any function which satisfies these two properties is equivalent to the *reverse* function. The FitSpec [9] tool can be used to test a specification for completeness and suggest possible redundancies.

2.3 Narrowing In Property-Based Testing

The application of functional-logic programming techniques to property-based testing is a natural idea, as can be seen by the ease in which the latter can be embedded in the former. That is, the essence of property-based testing — to find a counterexample that refutes a property — can be realised succinctly as a functional-logic program:

$$\begin{aligned}&\mathit{refute} :: (a \to \mathit{Bool}) \to a\\&\mathit{refute}\ p = \mathbf{let}\ x\ \mathit{free}\ \mathbf{in}\ \mathit{when}\ (\neg\ p\ x)\ x\end{aligned}$$

This definition expresses that we can refute a property by first creating a free variable, and then selecting all possible values for this variable that do not satisfy the given property. However, while *refute* has the desired behaviour, the approach requires the use of a functional-logic language, whereas in this thesis we are interested in testing using a functional language. Hence, we need to consider alternative approaches.

Even within the context of a functional-logic language, there are challenges that remain with the use of *refute* for property-based testing. For example, a common pattern in properties is to have a precondition consisting of the conjunction of multiple constraints and on this form of property the standard narrowing based evaluation strategy is often ineffective. We give a simple example of such a property at the beginning of chapter 5, along with explanation of why standard narrowing is not effective when testing the property.

To resolve these tensions, in this thesis we develop an approach to narrowing for functional languages that is specifically designed to support property-based testing, and show how this can be extended to increase the scope of properties that can be effectively tested. We will discuss other narrowing-based approaches and tools for property-based testing in the related work sections of subsequent chapters.

2.4 Operational Semantics

The semantics of a programming language describes its behaviour, thereby giving each program a meaning. In this thesis we use the *operational* approach, in which the behaviour of a program is described as a series of computational steps. Operational semantics can be classifed into two styles, *small-step* and *big-step*, both of which are used in this thesis.

To illustrate the two styles, in this section we consider a simple example language comprising logical values and conditional expressions:

$$\mathit{Expr} ::= \textbf{val}\ \ \mathit{Bool} \mid \textbf{if}\ \ \mathit{Expr}\ \mathit{Expr}\ \mathit{Expr}$$

The expression $\textbf{if}\ e\ e'\ e''$ is an *if*-expression, in which the *subject* is e, the *then* branch is e' and the required *else* branch is e''.

2.4.1 Small-Step Semantics

Small-step semantics, also known as structured operational semantics [46], describes the individual steps of a computation. These steps can then be chained together into a sequence of reductions. In this thesis, we use a style of small step semantics known as *reduction semantics* [22]. In this style, local reduction rules are defined along with the contexts in which these rules can be applied. For our example language, the local reduction rules are given by a relation $\to_R \subseteq \mathit{Expr} \times \mathit{Expr}$ defined as follows:

$$\frac{}{\textbf{if}\ (\textbf{val}\ \texttt{True})\ e\ e' \to_R\ e} \qquad \frac{}{\textbf{if}\ (\textbf{val}\ \texttt{False})\ e\ e' \to_R\ e'}$$

The first rule states that if the subject of an *if*-expression is `True` then the expression can be reduced to the *then* branch. The second rule covers the case when the subject is `False`.

We then define the contexts in which these rules can be applied. Informally, a context is an expression with a singular hole, denoted by $\bullet$, and a substitution $\mathbf{C}[e]$ replaces the hole in a context $\mathbf{C}$ with the expression e. For our language, we define the notion of contexts as follows:

$$\frac{}{\bullet \;\; \text{context}} \qquad\qquad \frac{\mathbf{C} \;\; \text{context}}{(\textbf{if} \;\; \mathbf{C} \;\; e \; e') \;\; \text{context}}$$

The above definition expresses that a local reduction rule can either be applied directly, or in the subject of an *if*-expression. This form of context defines a reduction strategy in which the subject of an *if*-expression must be reduced before the branches are considered. It is also possible to define a *full reduction* semantics in which the hole in the context can also appear in the branches of an *if*-expression and therefore a reduction can happen anywhere in the expression. This idea will be used later on in the thesis when we come to consider overlapping patterns.

A small-step semantics for expressions, $\rightarrow \subseteq \mathit{Expr} \times \mathit{Expr}$, is then given simply by applying a local reduction rule in a context:

$$\frac{e \rightarrow_R e' \qquad \mathbf{C} \;\; \text{context}}{\mathbf{C}[e] \rightarrow \mathbf{C}[e']}$$

This rule represent a single reduction step. We can chain these reductions together by taking the *reflexive-transitive closure* of the relation:

$$\frac{}{e \rightarrow^* e} \qquad \frac{e \rightarrow e' \qquad e' \rightarrow^* e''}{e \rightarrow^* e''}$$

In this thesis we use small-step semantics in chapters 3 and 5. As we shall see, using a small-step approach gives a natural means of extending a functional semantics to a narrowing semantics, and is also well-suited to defining the semantics of overlapping patterns.

2.4.2 Big-Step Semantics

Big-step semantics, also known as natural semantics [31], describes the complete reduction of an expression to its final result. For our example language, a big-step semantics $\Downarrow \subseteq \mathit{Expr} \times \mathit{Bool}$ can be defined as follows:

$$\frac{}{\textbf{val}\ \ b \Downarrow b} \qquad \frac{e \Downarrow \texttt{True} \qquad e' \Downarrow b}{\textbf{if}\ e\ e'\ e'' \Downarrow b} \qquad \frac{e \Downarrow \texttt{False} \qquad e'' \Downarrow b}{\textbf{if}\ e\ e'\ e'' \Downarrow b}$$

The first rule expresses that if the initial expression is already a logical value, then this value is the result of the evaluation. The second rule states that when evaluating an *if*-expression, if the subject evaluates to `True`, then the expression evaluates to the result of the *then* branch. The third gives a similar rule for when the subject evaluates to `False`.

In this thesis we use big-step semantics in chapters 4 and 6, to define a call-by-need narrowing semantics. Big-step semantics are well-suited for this purpose, as they have a close relation to our actual implementation, which is coded as an abstract machine in Haskell. For example, the above big-step semantics can be realised in Haskell as follows:

```
evalBig :: Monad m ⇒ Expr → m Bool
evalBig (val  b) = return b
evalBig (if  e e′ e″) = do
  b ← evalBig e
  case b of
    True → evalBig e′
    False → evalBig e″
```

We have used a monadic style for the code, even though it is not strictly necessary here, for consistency with the rest of the thesis in which the use of monads plays an important role. The first case of the function implements the first rule of the semantics. The second case implements the second and third rules. We could also have combined these two rules in the semantics if we wished, but we preferred to keep the case analsysis at the rule level. This aside, the main difference between the semantics and the implementation is that the implementation makes the order of evaluation explicit, whereas in the semantics the order is only implicit.

2.5 Conclusion

In this chapter we have set the scene for the rest of the thesis, by reviewing the basic ideas of functional-logic programming, property-based testing, and the operational approach to semantics.

Chapter 3

Narrowing Theory

In this chapter we develop a narrowing theory that can be used as justification for a narrowing tool such as the one found in chapter 4, and which we use as a foundation for our more complex overlapping theory (chapter 5). It is based on the paper *Towards a Theory of Reach* [26].

3.1 Introduction

Narrowing-inspired evaluation strategies have been used by many tools for the purpose of property-based testing a functional programming language [15, 33, 38, 48]. This research has generally focused on the practical issues concerning implementation and performance. In this chapter, we lay the groundwork for proving the correctness of such tools. We do so by considering narrowing as an extension to a semantics for a functional programming language and then relating the extension back to the original semantics by giving a soundness and a completeness theorem.

To focus on the essence of the problem, we initially consider a minimal language with a standard non-strict semantics (section 3.3). The language only includes Booleans and lists along with if-expressions and case-expressions. Abstracting away from the details of a real language such as Haskell we keep the presentation neat and concise but still include enough detail to understand the properties of narrowing.

To define the narrowing semantics we first add free variables. In the semantics (section 3.4) these variables are refined, bound to a constructor, when they are required for evaluation to continue. These refinements are stored in a substitution which is accrued during evaluation.

The relation between the extended and original semantics is formalised by a soundness and a completeness theorem (section 3.5), of which the proofs have been formally verified in Agda (section 3.6). Here we present proofs of the main results based on a number of lemmas, but for brevity we have not provided proofs of the lemmas, which can be found in the accompanying Agda code [24]. We then describe how the language can be extended with a number of additional features and extend the Agda formalisation accordingly (section 3.7). Finally, we discuss related work and draw conclusions (sections 3.8 and 3.9).

Although property-based testing is only one use of such a narrowing semantics we adopt it to give the chapter context. First, in a step-by-step example of narrowing (section 3.2), which we subsequently use to discuss the advantages of narrowing in property-based testing. Then throughout we discuss the refutation of a property as an application of our theory. This special case of the theory shows that if the narrowing semantics derives a refutation then this is a refutation in the original language (soundness), and that if a refutation exists then it exists in the narrowing semantics (completeness).

3.2 Narrowing Step-by-Step

In this section we give a demonstration of narrowing on a property-based testing example and in doing so we explain why narrowing is beneficial to testing. We consider the following property of the *union* function [1]:

$$\begin{aligned}&\mathit{propUnion} :: \mathit{Ord}\ a \Rightarrow [\,a\,] \rightarrow [\,a\,] \rightarrow \mathit{Result}\\ &\mathit{propUnion}\ x\ y = \mathit{set}\ x\ \wedge\ \mathit{set}\ y\ \implies\ \mathit{set}\ (\mathit{union}\ x\ y)\end{aligned}$$

The *propUnion* property asserts the result of the *union* is a finite set, when its two inputs are finite sets. We represent finite sets as strictly ordered lists and the condition *set* determines whether a list is a valid set. Such a property is difficult to test directly using property-based testing. This is because most standardly generated lists will not satisfy the set condition, and as such lists cannot be used to test the union function, they are not valid test cases. QuickCheck solves this problem by allowing users to define custom generators [16], however by using narrowing we can avoid the need to do so. In fact, sets are generated effectively by a narrowing evaluation of the *set* condition with a free variable input and we can test the entire property effectively by evaluating it with two free variables as input. We now demonstrate how this idea works in practice.

For our example of a narrowing evaluation we focus on the *set* condition which will be the first sub-expression of *propUnion* to be evaluated. We use the following definition of the set condition:

[1]Note, we have used Haskell as opposed to our minimal language in order to give a meaningful example.

```
$set :: Ord\ a \Rightarrow [a] \to Bool$
$set\ l = $ **case** $l$ **of**
  $[\,] \to$ True
  $(a : l') \to set'\ a\ l'$
$set'\ a\ l = $ **case** $l$ **of**
  $[\,] \to$ True
  $(a' : l') \to a < a'\ \wedge\ set'\ a'\ l'$
```

We evaluate the expression *set* with a free variable applied and bind a free variable whenever its value is required to proceed with evaluation, i.e. when the free variable is the subject of the case expression being evaluated. Each state during the evaluation is given by an expression and a substitution, a mapping which is an accumulation of the free variable bindings up to the current point of evaluation. For our example, the initial expression is $set\ l$, in which l is free, and the initial substitution is the trivial mapping $\{l \mapsto l\}$:

1) $\{l \mapsto l\}$
   $set\ l$

Starting with the trivial mapping rather than the traditional empty mapping helps with the formalisation, as discussed further in section 3.4.1. The first step of evaluation is to inline the definition for $set\ l$:

```
2) $\{l \mapsto l\}$
   (**case** $l$ **of**
     $[\,] \to$ True
     $(a : l') \to ...)$
```

In order for evaluation to continue the value of the free variable l is now required, which necessitates a *refinement*. The variable is of list type, and therefore can be refined to either the empty list or a cons constructor. To begin with, we bind it to the empty list, represented by the substitution $l \mapsto [\,]$, and then continue evaluating the expression.

3) $\{\, l \mapsto [\,] \,\}$

   `True`

The expression has evaluated to `True` and therefore we have generated a *set* which is our binding for l the empty list $[\,]$. Although this is a *set* it is not a particularly interesting one. We now look at another scenario to see where narrowing is beneficial. We start part way through the narrowing with the substitution $\{\, l \mapsto 0 : a : l' \,\}$:

4) $\{\, l \mapsto 0 : a : l' \,\}$

   $0 < a \;\wedge\; set'\ a\ l'$

The value of a is required to continue. Consider if we bind it to 0:

5 *a*) $\{\, l \mapsto 0 : 0 : l' \,\}$

   `False`

The expression is evaluated to `False` as a set should only have one of each element (as encoded with a strict order). The variable l' is left free in the substitution and we can conclude that no binding of l' will ever give a set. Narrowing allows us to immediately discard any list of this form and therefore stops the generation of many invalid test cases.

In order to continue generating a *set* we have to backtrack, undoing our last binding of $a \mapsto 0$. We might try $a \mapsto 1$ instead:

5 *b*) $\{\, l \mapsto 0 : 1 : l' \,\}$

   $set'\ 1\ l'$

Now if we bind l' to $[\,]$, then another solution is formed with the final substitution being $l \mapsto 0 : 1 : [\,]$.

At the beginning of this section we claimed that narrowing was able to generate solutions effectively. The natural question then is what do we mean by effectively? In the next chapter, we test an implementation of a narrowing tool on this property and find that it is able to generate random test cases with a well-defined distributed and reasonable performance. As the definition of reasonable performance is somewhat subjective we also discuss what drives the performance. Particularly, performance is reasonable for this property because the maximum amount of backtracking required increases linearly with the depth limit imposed (section 4.4).

3.2.1 Benefits of Narrowing

The primary benefit of narrowing occurs when a result is deduced for a range of inputs as free variables remain in the substitution. In our example we saw the substitution $\{\, l \mapsto 0 : 0 : l' \}$ in which l' remains free, returned `False` when applied to the set condition. This allowed us to conclude any test case beginning with two zeroes is not valid and therefore removes the need for them to be evaluated individually. For property-based testing particularly, narrowing could conclude a range of inputs is invalid (as in the example), could conclude a range of inputs all satisfy a property or could conclude that a range of inputs refutes a property.

A second benefit often associated with narrowing is shared evaluation. In many implementations of narrowing, the evaluation between different inputs is shared up to the point where their differences cause execution to take separate branches. Shared evaluation is typically a feature of languages with dedicated narrowing evaluation, such as Curry [29]. However sharing generally necessitates specific compiler support whereas narrowing without sharing can sometimes be implemented with only limited compiler support. For example, the tool Lazy Smallcheck [48] utilises the already existing error-handling functionality in the Glasgow Haskell Compiler to implement a narrowing inspired evaluation which does not have sharing. In the next chapter, we compare the performance of narrowing evaluation with and without sharing against a basic evaluation strategy.

3.3 Language and Semantics

In this section we introduce the language that we use to give a theory of narrowing. The language is not suitable for actual programming, but does provide enough structure to describe the key mechanisms of narrowing. To this end the language has only two types, Booleans and lists, which provides enough to describe properties and demonstrate the key properties of narrowing. The grammar for expressions of the language is defined as follows:

$$\begin{aligned}
\mathit{Expr} ::= &\ \ \texttt{False} \ \mid\ \texttt{True} \ \mid\ \textbf{if}\ \mathit{Expr}\ \mathit{Expr}\ \mathit{Expr}\\
&\mid\ [\,]\ \mid\ \mathit{Expr} : \mathit{Expr}\ \mid\ \textbf{case}\ \mathit{Expr}\ \mathit{Expr}\ \mathit{Alt}\\
&\mid\ \texttt{var}\ \mathit{Var}\\
\mathit{Alt}\ \ ::= &\ (\mathit{Var} : \mathit{Var}) \to \mathit{Expr}\\
\mathit{Val}\ \ ::= &\ \ \texttt{False} \ \mid\ \texttt{True} \ \mid\ [\,]\ \mid\ \mathit{Val} : \mathit{Val}\\
\mathit{Type} ::= &\ \mathit{Bool} \mid [\,\mathit{Type}]
\end{aligned}$$

That is an expression is either a boolean, a list, an if-statement, a case expression on lists or a variable. Case expressions have the form **case** $e\ e_{[]}\ f$, where e is the subject, $e_{[]}$ is the first alternative for the empty list and f is the second alternative for the cons constructor. Expressions are assumed to be closed – variables only appear within the case expression in which they are bound. The values of the language are Booleans and lists. All the expressions are assumed to be well-typed under a standard set of typing rules.

Note that the language does not contain functions or recursion, as these are not required to study the 'essence' of narrowing. We do however provide an additional Agda formalisation that incorporates these features, as discussed in section 3.7.

The behaviour of expressions is defined as a small-step operational semantics. First we define the *redex* reductions, $\rightarrow_R \subseteq \mathit{Expr} \times \mathit{Expr}$, which are the local reduction rules:

$$\frac{}{\textbf{if}\ \texttt{True}\ e\ \_ \rightarrow_R e}\ \textsc{if-1} \qquad \frac{}{\textbf{if}\ \texttt{False}\ \_\ e \rightarrow_R e}\ \textsc{if-2}$$

$$\frac{}{\textbf{case}\ []\ e\ \_ \rightarrow_R e}\ \textsc{case-1} \qquad \frac{f = (u : v) \rightarrow e}{\textbf{case}\ (a : l)\ \_\ f \rightarrow_R e[u/a, v/l]}\ \textsc{case-2}$$

Using a small-step semantics will allow for a natural extension to narrowing later. The semantics for the if-statements and case expressions are standard, where $e[u/a]$ denotes the substitution of variable u by the expression a in the expression e in a capture avoiding manner. Next we define the contexts in which a redex can applied:

$$\frac{}{\bullet\ \text{context}} \qquad \frac{\mathbf{C}\ \text{context}}{(\textbf{if}\ \mathbf{C}\ e\ e')\ \text{context}} \qquad \frac{\mathbf{C}\ \text{context}}{(\textbf{case}\ \mathbf{C}\ e\ f)\ \text{context}}$$

A context is an expression with a hole, •, within it which represents where a reduction can be performed. In our semantics we can either apply a reduction to the top-level expression or within the subject of case and if expressions. The replacement of the hole in a context with an expression e is denoted $\mathbf{C}[e]$.

We can now define the operational semantics of the language. The reduction rule, $\rightarrow \subseteq \mathit{Expr} \times \mathit{Expr}$, is given by:

$$\frac{e \rightarrow_R e' \qquad \mathbf{C}\ \text{context}}{\mathbf{C}[e] \rightarrow \mathbf{C}[e']}$$

When applying the semantics in practice, we often use the reflexive transitive closure, $\rightarrow^*$, which is defined in the normal manner:

$$\frac{e \rightarrow e' \qquad e' \rightarrow^* e''}{e \rightarrow^* e''}\ \text{SEQ} \qquad\qquad \frac{}{e \rightarrow^* e}\ \text{REFL}$$

The semantics can be shown by standard methods to be normalising (always terminates in a finite number of steps) and deterministic (always produces a single possible result). However, neither property is a requirement for the extension to narrowing or the correctness result which follows.

3.4 Narrowing Semantics

In this section we define a *narrowing* semantics for our minimal language extended with free variables. As illustrated in section 3.2, the basic idea of narrowing is that when evaluation of an expression is suspended on the value of a free variable, we allow evaluation to proceed by performing a *refinement*, in which each partial value that the variable could have is considered in turn. As evaluation proceeds a substitution is gradually built up which tracks the instantiation of free variables. Finally, we consider how the narrowing semantics can be used in property-based testing.

3.4.1 Adding Free Variables

Before we define the narrowing semantics we need to extend our language with free variables. One possible encoding of free variables is to simply allow our expressions to be open, letting the existing variables be free. Although this is the approach taken by others, such as the Reach work [38, 39], we choose to syntactically separate the free variables as an extension of the language. Our reason for making this choice is that free variables are independent of the normal variables of a language. For example, it is easy to make a similar extension to a language that does not have any form of variables.

The extended grammar for expressions is defined below, in which each rule is now parameterised by a set X of free variables and their types, and expressions and values are extended with free variables of the form `fvar` X. Note that we do not require the set of variables for an expression to be minimal, i.e. the set may contain variables that are not used in the expression.

$$\begin{aligned}
\mathit{Expr}_X &::= \texttt{False} \mid \texttt{True} \mid \textbf{if}\ \mathit{Expr}_X\ \mathit{Expr}_X\ \mathit{Expr}_X \\
&\quad\mid [\,] \mid \mathit{Expr}_X : \mathit{Expr}_X \mid \textbf{case}\ \mathit{Expr}_X\ \mathit{Expr}_X\ \mathit{Alt}_X \\
&\quad\mid \texttt{var}\ \mathit{Var} \mid \texttt{fvar}\ X \\
\mathit{Alt}_X &::= (\mathit{Var} : \mathit{Var}) \rightarrow \mathit{Expr}_X \\
\mathit{Val}_X &::= \texttt{False} \mid \texttt{True} \mid [\,] \mid \mathit{Val}_X : \mathit{Val}_X \mid \texttt{fvar}\ X
\end{aligned}$$

We will view values of type Val_X as *partial values*, in the sense that they may contain undefined components represented by the free variables. We can also view the original grammars as special cases of the free variable versions in which the free variable sets are empty, i.e. $Expr \equiv Expr_\emptyset$, $Alt \equiv Alt_\emptyset$ and $Val \equiv Val_\emptyset$. We write $x :: t \in X$ if x is a free variable in X with type t. When the type isn't important we shorten this to $x \in X$

Substitutions

Substitutions are used to update the free variables in an expression by providing a mapping from each free variable to a partial value. Formally, a *substitution* of type $X \to Y$ is a mapping from the set of free variables X to partial values that contain free variables from the set Y:

$$Sub_{X \to Y} \quad = \quad X \to Val_Y$$

Defining substitutions in this manner rather than as a partial mapping from an infinite set of variables results in a simpler formalisation in Agda. In particular, incorporating the set of variables for the domain and range directly into the type removes the need to add the variable sets as constraints later on. A second benefit of this approach is that it yields a monadic interpretation of the composition of substitutions. Given this representation the traditional empty map becomes the trivial map in which each variable is mapped to itself.

A special case of a substitution occurs when the resulting free variable set is empty and therefore binds each variable to a value. We call such a substitution an *input*:

$$Inp_X \quad = \quad Sub_{X \to \emptyset}$$

$$
\begin{array}{ll}
\_\,[\_] \;::\; \mathit{Expr}_X \rightarrow \mathit{Sub}_{X \rightarrow Y} \rightarrow \mathit{Expr}_Y & \\
\texttt{False}\,[\sigma] & = \texttt{False} \\
\texttt{True}\,[\sigma] & = \texttt{True} \\
\textbf{if}\ e\ e'\ e''\,[\sigma] & = \textbf{if}\ (e\,[\sigma])\ (e'\,[\sigma])\ (e''\,[\sigma]) \\
[\,]\,[\sigma] & = [\,] \\
(a : l)\,[\sigma] & = a\,[\sigma] : l\,[\sigma] \\
\textbf{case}\ e\ e'\ ((v : v') \rightarrow e'')\,[\sigma] & = \textbf{case}\ (e\,[\sigma])\ (e'\,[\sigma])\ ((v : v') \rightarrow e''\,[\sigma]) \\
\texttt{var}\,v\,[\sigma] & = \texttt{var}\,v \\
\texttt{fvar}\ x\,[\sigma] & = \sigma\ x
\end{array}
$$

Figure 3.1: The application of a substitution to an expression.

We denote substitutions by σ and inputs by τ. The process of applying a substitution is defined recursively in the normal way (Figure 3.1). Note that applying an input to an expression results in an expression in our original language.

3.4.2 Preliminaries

We define a number of extra concepts that are used in our formalisation of narrowing, in the form of suspended expressions, minimal narrowing sets, and the composition of substitutions.

Suspended expressions

An expression e is *suspended* on a free variable x, denoted by $e \multimap x$, if the value of the variable is required for evaluation of the expression to proceed any further. For our language, an expression is suspended on a free variable if it appears in the evaluation context:

$$\frac{\mathbf{C}\ \text{context}}{\mathbf{C}[x] \multimap x}$$

Expressions that are suspended can make no further transitions in our small-step operational semantics. However, the converse is not true. In particular, values cannot make further transitions, but are not suspended.

Minimal narrowing set

When an expression is suspended there is a set of possible refinements that can be performed. A refinement is a substitution that should be *minimal*, in the sense that it should only instantiate a free variable just enough to allow evaluation to continue, and no further. In our language, when the free variable has Boolean type it is refined to `True` or `False` and when it has a list type it is refined to $[\,]$ or $(x : x')$ where x and x' are new free variables.

To formalise this idea, we begin by writing $x \ / \ a$ for the one-point substitution that maps the free variable $x \in X$ to the partial value $a \in \mathit{Val}_Y$ and leaves all other variables in X unchanged, defined as follows:

$$\begin{aligned}
&(/) && :: (x \ \in \ X) \to \mathit{Val}_Y \to \mathit{Sub}_{X \to X[x/Y]} \\
&(x \ / \ a) \ x' \mid x \equiv x' && = a \\
&\qquad\qquad\mid \mathit{otherwise} && = \texttt{fvar} \ x'
\end{aligned}$$

The return type of the substitution is given by $X\ [x \ / \ Y] = (X - \{x\}) \ \cup \ Y$, in which the element $x \in X$ is replaced by the set Y. Note that the type of $(/)$ depends on the name of the variable x, i.e. the operator has a *dependent* type. Being precise in this manner helps to simplify our

Agda formalisation. Using this operator we can now define the *minimal narrowing set*, $\mathit{Narr}_X(x)$, of a free variable $x \in X$. In the case where $x :: \mathit{Bool}$, we have:

$$\mathit{Narr}_X(x) \;=\; \{x/\,\texttt{False},\;\; x/\,\texttt{True}\} \qquad \textsf{where } x :: \mathit{Bool} \in X$$

A Boolean free variable can either be replaced by `True` or `False`. In both cases the resulting variable set will be $X[x/\emptyset]$. In the case where the free variable is list:

$$\begin{aligned}\mathit{Narr}_X(x) \;=\; \{x/[\,],\;\; x/(y : y')\} \qquad &\textsf{where}\;\; x :: [a] \in X \\ &y,\; y' \notin X,\; y :: a,\; y' :: [a]\end{aligned}$$

The free variable can either be replaced by the empty list or the cons constructor. In the empty list case there are no fields and the resulting variable set is again $X[x/\emptyset]$. The cons case is more interesting as there are two fields. Each field is replaced by a new free variable, y and y', of the correct type and the new variable set includes these variables: $X[x/\{y :: a,\; y' :: [a]\}]$.

The narrowing set has two properties that play an important role in *completeness* of the lazy narrowing semantics. Firstly, the minimal narrowing set itself obeys a notion of completeness, in the sense that for every input that is possible before the narrowing there exists a substitution in which the input remains possible. And secondly, each substitution in the minimal narrowing set is *advancing*, in that it always instantiates a variable. These properties are formalised in section 3.5.2.

Composition of Substitutions

As evaluation proceeds under narrowing, we will construct a substitution in a compositional manner from the refinements. In order to define a composition operator for substitutions, we first note that *Val* forms a monad under the following definitions:

$$
\begin{array}{l}
\mathit{return} :: X \to \mathit{Val}_X \\
\mathit{return} = \texttt{fvar} \\
(\gg\!=) \;::\; \mathit{Val}_X \to (X \to \mathit{Val}_Y) \to \mathit{Val}_Y \\
\begin{array}{lcl}
\texttt{False} & \gg\!= \sigma = & \texttt{False} \\
\texttt{True} & \gg\!= \sigma = & \texttt{True} \\
[\,] & \gg\!= \sigma = & [\,] \\
(e : e') & \gg\!= \sigma = & (e \gg\!= \sigma) : (e' \gg\!= \sigma) \\
\texttt{fvar}\ x & \gg\!= \sigma = & \sigma\ x
\end{array}
\end{array}
$$

We note in passing that this is the *free monad* of the underlying functor for partial values. Using the $\gg\!=$ operator for this monad it is then straightforward to define the composition operator for substitutions:

$$
\begin{array}{l}
(>\!\!=\!\!>) \quad :: \; \mathit{Sub}_{X \to Y} \to \mathit{Sub}_{Y \to Z} \to \mathit{Sub}_{X \to Z} \\
\mathit{sa} >\!\!=\!\!> \mathit{sa}' = \lambda a \to \mathit{sa}\ a \gg\!= \mathit{sa}'
\end{array}
$$

Moreover, expanding out the definition of *Sub* in the type for the $>\!\!=\!\!>$ operator gives $(X \to \mathit{Val}\ Y) \to (Y \to \mathit{Val}\ Z) \to (X \to \mathit{Val}\ Z)$, which corresponds to the standard notion of *Kleisli composition* for the *Val* monad.

Along with the monad laws we require one more law, relating the composition of substitutions to the application of a substitution.

Lemma 1. *The sequential application of substitutions to an expression is* *equivalent to the application of the composed substitutions to the expression:*

$$e[\sigma][\sigma'] \;\equiv\; e[\sigma \ggg \sigma']$$

3.4.3 Semantics

We now have all the ingredients required to define a narrowing semantics for our minimal language. A step in the new semantics is either:

- a single step in the original semantics; or
- a refinement, if the expression is suspended.

To keep track of the substitutions that are applied during narrowing, we write $e \rightsquigarrow \langle e', \sigma \rangle$ to mean that expression e can make the transition to expression e' in a single step, where σ is the substitution that has been applied when a refinement is made. In the case of a step in the original semantics, we simply return the identity substitution, which is given by the *return* operator of the *Val* monad. More formally, we define a transition relation $\rightsquigarrow \subseteq \mathit{Expr}_X \times (\mathit{Expr}_Y \times \mathit{Sub}_{X \to Y})$ for narrowing by the following two inference rules:

$$\frac{e \rightarrow_X e'}{e \rightsquigarrow \langle e',\ \mathit{return} \rangle}\,\textsc{prom} \qquad \frac{e \multimap x \qquad \sigma \in \mathit{Narr}_X(x)}{e \rightsquigarrow \langle e[\sigma],\ \sigma \rangle}\,\textsc{ref}$$

The first rule promotes transitions from the original semantics to the new semantics, where $\rightarrow_X \subseteq \mathit{Expr}_X \times \mathit{Expr}_X$ is the trivial lifting of the transition relation $\rightarrow \subseteq \mathit{Expr} \times \mathit{Expr}$ to operate on expressions with free variables in the set X, for which the inference rules remain syntactically the same as previously except that they now operate on expressions of a more general form. The second rule applies a minimal narrowing step to a suspended expression.

The definition of how to sequence steps in our extended semantics, which takes into account the additional presence of substitutions, is given by a relation $\rightsquigarrow^*$ that is defined by the following two rules:

$$\frac{e \rightsquigarrow \langle e',\ \sigma\rangle \qquad e' \rightsquigarrow^* \langle e'',\ \sigma'\rangle}{e \rightsquigarrow^* \langle e'',\ \sigma \ggg \sigma'\rangle}\ \textsc{seq} \qquad \frac{e \in \mathit{Expr}_X \qquad \tau \in \mathit{Inp}_X}{e \rightsquigarrow^* \langle e[\tau],\ \tau\rangle}\ \textsc{fill}$$

The first rule simply composes the substitutions from the two component reductions. The second rule performs a final substitution that instantiates any remaining free variables and in doing so makes formalising the relation to the original semantics simpler.

3.4.4 Property-Based Testing

Property-based testing attempts to find an input which refutes a property. The set of possible refutations to a property, $\mathit{refute}(e) \subseteq \mathit{Inp}_X$, can be given by:

$$\tau \in \mathit{refute}(e) \quad \Longleftrightarrow \quad e[\tau] \rightarrow^* \texttt{False}$$

That is, an input τ that provides values for the free variables refutes the property e *iff* the input applied to the property evaluates to `False`.

We can give an alternative definition of refutation by using our narrowing semantics. Given a property, $e \in \mathit{Expr}_X$, the set of inputs $\mathit{refute}_N(e) \in$

Inp_X that refute the condition are defined as follows:

$$\tau \in \mathit{refute}_N(e) \quad \Longleftrightarrow \quad e \rightsquigarrow^* \langle \texttt{False},\ \tau \rangle$$

That is, an input τ refutes a property *iff* there is a narrowing reduction sequence that evaluates to `False` and whose ending substitution can be further refined to the input τ. The key difference with our original definition of *refute* is that the narrowing semantics constructs an input substitution during the reduction sequence, whereas the original semantics requires that we are given an input so that it can be applied prior to starting the reduction process. In the next section we show that these two notions of refutation coincide.

3.5 Correctness of the Narrowing Semantics

We formalise the relationship between our narrowing semantics and the original semantics. This relationship is characterised by two properties, *soundness* and *completeness*, which are proved using a number of lemmas. The proofs of the lemmas themselves are provided in the associated Agda formalisation.

3.5.1 Soundness

Lemma 2. *A transition in the original semantics can be lifted through a substitution. Given a substitution $\sigma \in Sub_{X \to Y}$, we have:*

$$e \rightarrow_X e' \implies e[\sigma] \rightarrow_Y e'[\sigma]$$

Theorem 3.1 (Soundness)**.** *For every reduction sequence in the narrowing semantics there is a corresponding sequence in the original semantics:*

$$e \rightsquigarrow^* \langle e',\ \tau\rangle \quad\Longrightarrow\quad e[\tau] \rightarrow^* e'$$

Proof. The proof proceeds by rule induction on the definition for the narrowing relation $\rightsquigarrow^*$, for which there are three cases to consider.

Case 1 In the base case when the narrowing is a simple application of

$$\frac{}{e \rightsquigarrow^* \langle e[\tau],\ \tau\rangle}\ \textsc{fill}$$

the goal follows immediately from the reflexivity of $\rightarrow^*$:

$$\frac{}{e[\tau] \rightarrow^* e[\tau]}\ \textsc{refl}$$

Case 2 There are two inductive cases to consider, depending on the nature of the first reduction in a narrowing sequence. We first consider the case when the reduction is a refinement, constructed as follows:

$$\frac{\textsc{ref}\ \dfrac{e \multimap x \qquad \sigma \in \mathit{Narr}_X(x)}{e \rightsquigarrow \langle e[\sigma],\ \sigma\rangle} \qquad e[\sigma] \rightsquigarrow^* \langle e',\ \tau\rangle}{e \rightsquigarrow^* \langle e',\ \sigma \ggg \tau\rangle}\ \textsc{seq}$$

We are now free to use the three assumptions $e \multimap x$, $\sigma \in \mathit{Narr}_X(x)$ and $e[\sigma] \rightsquigarrow^* \langle e',\ \tau\rangle$ in our proof. In this case, we only require the third of these assumptions in order to verify our goal, by first using the induction hypothesis (IH) $e[\sigma] \rightsquigarrow^* \langle e',\ \tau\rangle \implies e[\sigma][\tau] \rightarrow^* e'$, and then applying lemma 1:

$$\dfrac{\dfrac{e[\sigma] \rightsquigarrow^* \langle e',\ \tau\rangle}{e[\sigma][\tau] \rightarrow^* e'}\ \textsc{ih}}{e[\sigma \mathrel{>\!\!=\!\!>} \tau] \rightarrow^* e'}\ \textsc{lemma}\ 1$$

Case 3 We now consider the case when the first reduction is a promoted reduction from the original language, constructed as follows:

$$\dfrac{\textsc{prom}\ \dfrac{e \rightarrow_X e'}{e \rightsquigarrow \langle e',\ \mathit{return}\rangle} \qquad e' \rightsquigarrow^* \langle e'',\ \tau\rangle}{e \rightsquigarrow^* \langle e'',\ \mathit{return} \mathrel{>\!\!=\!\!>} \tau\rangle}\ \textsc{seq}$$

In this case our goal can then be verified by lifting the reduction from the original language through the input substitution using lemma 2, sequencing with the result of applying the induction hypothesis to the remaining reduction sequence, and finally applying an identity law for Kleisli composition:

$$\dfrac{\dfrac{\textsc{lemma}\ 2\ \dfrac{e \rightarrow_X e'}{e[\tau] \rightarrow e'[\tau]} \qquad \dfrac{e' \rightsquigarrow^* \langle e'',\ \tau\rangle}{e'[\tau] \rightarrow^* e''}\ \textsc{ih}}{e[\tau] \rightarrow^* e''}\ \textsc{seq}}{e[\mathit{return} \mathrel{>\!\!=\!\!>} \tau] \rightarrow^* e''}\ \textsc{id}$$

□

Although the above proof was presented specifically for the narrowing semantics given in section 3.4, it is not dependent on the properties of the narrowing set or the condition for applying a refinement (in our case suspension of the variable). Therefore the proof is also valid for any narrowing set and any applicability condition.

3.5.2 Completeness

Definition 3.1. We exploit two pre-orderings on substitutions, which respectively capture the idea of one substitution being a *prefix* or *suffix* of another:

$$\sigma_1 \sqsubseteq \sigma_2 \iff \exists \sigma'.\ \sigma_1 \mathrel{>\!\!\!=\!\!\!>} \sigma' \equiv \sigma_2$$

$$\sigma_1 \leqslant \sigma_2 \iff \exists \sigma'.\ \sigma' \mathrel{>\!\!\!=\!\!\!>} \sigma_1 \equiv \sigma_2$$

Lemma 3. *If the source expression of a transition in the original semantics is not suspended then the transition can be 'unlifted'. Given a substitution $\sigma \in Sub_{X \to Y}$ and a transition $e[\sigma] \to_Y e'$ for which $e \not\multimap x$, we have:*

$$\exists e''.\ e \to_X e'' \ \wedge\ e''[\sigma] \equiv e'$$

Lemma 4. *The narrowing set is complete. For every input there is a substitution in the narrowing set that is a prefix of the input:*

$$\forall x \in X, \tau \in Inp_X\,.\ \exists \sigma \in Narr_X(x).\ \sigma \sqsubseteq \tau$$

Lemma 5. *The narrowing set is advancing. The identity substitution is a strict prefix of every substitution in the narrowing set:*

$$\forall x \in X,\ \sigma \in Narr_X(x).\ return \sqsubset \sigma$$

Lemma 6. *The suffix relation $<$ is well-founded. For any finite substitution τ_0, there only exists finite chains of substitutions τ_i such that:*

$$\tau_n < \ldots < \tau_1 < \tau_0$$

Lemma 7. *A suffix formed by an advancing prefix is strict.*

$$\sigma \gg\!\!\Rightarrow \sigma_1 \equiv \sigma_2 \;\wedge\; \mathit{return} \sqsubset \sigma \quad \Longrightarrow \quad \sigma_1 < \sigma_2$$

Theorem 3.2 (Completeness)**.** *For every reduction sequence in the original semantics there is a corresponding reduction in the narrowing semantics:*

$$e[\tau] \rightarrow^* e' \quad \Longrightarrow \quad e \rightsquigarrow^* \langle e',\ \tau\rangle$$

Assuming that τ is a finite substitution.

Proof. The proof proceeds by double induction. First on the length of the reduction sequence $e[\tau] \rightarrow^* e'$ and then on the size of the input τ.

Case 1 In the base case when the evaluation is just reflexivity

$$\frac{}{e[\tau] \rightarrow^* e[\tau]}\ \textsc{refl}$$

the goal follows immediately by instantiating free variables:

$$\frac{}{e \rightsquigarrow^* \langle e[\tau],\ \tau\rangle}\ \textsc{fill}$$

Case 2 There are two inductive cases to consider, depending on whether or not the expression e is suspended when the sequencing rule is applied:

$$\frac{e[\tau] \rightarrow e' \qquad e' \rightarrow^* e''}{e[\tau] \rightarrow^* e''}\ \textsc{seq}$$

In the case when e is not suspended our goal can be verified as follows, in which the two branches of the proof tree exploit the two conclusions from lemma 3:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{e \to e'_\tau}\,\textsc{lemma 3}}{e \rightsquigarrow \langle e'_\tau,\ \mathit{return}\rangle}\,\textsc{prom} \qquad \dfrac{\dfrac{e' \to^* e''}{e'_\tau[\tau] \to^* e''}\,\textsc{lemma 3}}{e'_\tau \rightsquigarrow^* \langle e'',\ \tau\rangle}\,\textsc{ih}}{e \rightsquigarrow^* \langle e'',\ \mathit{return} \ggg\!\!\!\Rightarrow \tau\rangle}\,\textsc{seq}}{e \rightsquigarrow^* \langle e'',\ \tau\rangle}\,\textsc{id}$$

Case 3 Finally, the we consider the case when e is suspended on x. As the narrowing set $\mathit{Narr}(x)$ is complete (lemma 4), there exists a valid refinement that is a prefix of the input τ i.e. a substitution $\sigma \in \mathit{Narr}(x)$ and input τ' for which $\tau \equiv \sigma \ggg\!\!\!\Rightarrow \tau'$. Based upon this observation our goal can then be verified as follows:

$$\dfrac{\dfrac{\dfrac{e \multimap x \qquad \sigma \in \mathit{Narr}(x)}{e \rightsquigarrow \langle e[\sigma],\ \sigma\rangle}\,\textsc{ref} \qquad \dfrac{\dfrac{e[\tau] \to^* e'}{e[\sigma][\tau'] \to^* e'}\,\textsc{lemma 1}}{e[\sigma] \rightsquigarrow^* \langle e',\ \tau'\rangle}\,\textsc{ih}}{e \rightsquigarrow^* \langle e',\ \sigma \ggg\!\!\!\Rightarrow \tau'\rangle}\,\textsc{seq}}{e \rightsquigarrow^* \langle e',\ \tau\rangle}\,\textsc{lemma 4}$$

In this case, the length of the reduction sequence of the induction hypothesis is the same as the length of the reduction sequence in the original statement. In this case we rely on our second inductive principle, the size of the input. Via lemma 5 and 7 we have input τ' is a strict suffix of τ, that is $\tau' < \tau$. Together with lemma 6 this guarantees the well-foundedness of the proof.

□

Whereas the soundness proof was independent of the properties of the narrowing set and the condition for its applicability, the completeness proof relies on the fact that the narrowing set is complete and advancing, and that a refinement can always be applied when an expression is suspended.

3.5.3 Correctness

Using the soundness and completeness results, it is now straightforward to prove that our two notions of refutation are equivalent:

Theorem 3.3 (Correctness)**.** *For all expressions $e \in Expr_X$:*

$$refute_N(e) \;\equiv\; refute(e)$$

Proof.

$$\begin{aligned}
\tau \in refute_N(e) &\iff e \leadsto^* \langle \texttt{False},\ \tau\rangle && \text{(by definition)}\\
&\iff e[\tau] \rightarrow^* \texttt{False} && \text{(theorems 3.1 and 3.2)}\\
&\iff \tau \in refute(e) && \text{(by definition)}
\end{aligned}$$

□

3.6 Agda Formalisation

The correctness result has also been formalised in the Agda [41]. The underlying minimal language of the Agda formalisation has minor differences, with a natural number type instead of Boolean and list types. This is because it was formalised for the paper, *Towards a Theory of Reach* [24], which is the basis of this chapter. The changes only result in small differences in the proofs of lemmas but the soundness and completeness proofs remain identical.

Apart from this the Agda formalisation follows the presentation given in the chapter closely: the language grammar and semantic rules convert directly to inductive datatypes, and rule induction translates to recursive dependent functions. A proof of the main result and all associated lemmas is available online from:

`http://tinyurl.com/reachtheory`

Using Agda brings a number of important benefits. First of all, it provides a guarantee that the results are correct. Secondly, it helped guide the development of the theory and proofs, resulting in a number of simplifications. For example, when translating our original formalisation into Agda we found that it contained a subtle error. The process of correcting the error also pointed towards a neater theory. In particular, our original narrowing formulation kept the substitution as an environment, only replacing free variables when they were needed. The most natural way to fix the error was to apply the substitution to the current expression immediately, removing the need to keep the substitution as an environment. This also removed an unnecessary distinction in the formalisation: in the original formulation the expression/environment pair $\langle e,\ \sigma \rangle$ behaved equivalently to the pair $\langle e[\sigma],\ \sigma \rangle$, yet the two were distinct. And finally, the use of Agda had a positive effect on the formulation of the representation of substitutions. In order to ensure totality in the Agda we had to parameterise substitutions with the set of variables used in their domain and result. Far from being a hindrance, this led to the monadic formulation of composition.

3.7 Extending the Language

In this chapter we focused on a minimal language to emphasise the key elements of the process of narrowing. However, our results also scale up to a more realistic language that includes function application, lambda abstraction and fixed points [24]. This section briefly describes the changes that are required to the Agda formalisation.

First of all, the expression grammar is extended to include the three new constructors: function application, lambda abstraction and fixed points. The small step semantics is extended to account for the new language constructs.

Our formalisation of the narrowing semantics for the extended language restricts free variables, and by extension narrowing, to first-order datatypes (Boolean and List types). Although this is certainly a limitation, it is standard in the narrowing literature, where a narrowing theory is generally described for first-order data initially, and then potentially extended to the higher-order case in subsequent work. With this restriction, the alteration to the narrowing semantics and correctness proof is minor. The suspension predicate, $e \multimap x$, has to be updated as an expression can now be suspended within a function application or a fixpoint expression. We defined the narrowing semantics by lifting the original semantics, and this definition remains unchanged except that we now lift the extended semantics. Finally, the lemmas, particularly the lift and unlift lemmas (2 and 3), need updating to account for the additional cases. The proof of soundness and completeness remain identical under the updated lemmas.

3.8 Related Work

There is a large body of work on the theory of narrowing in functional logic programming. We introduce and compare two particularly relevant theories to ours. In their seminal work, Antoy et al. [4] established the soundness and completeness of the related notion of needed narrowing, and the optimality of needed narrowing within a restricted domain. However, whereas our formalisation is based on extending a small-step semantics, theirs is based on classical rewrite systems. As a result, our approach is easier to mechanically verify, which we have done, as the semantics of our language has a direct representation in proof assistants. In fact, to the best of our knowledge, this is a first time that a lazy narrowing formalisation has had such a verification.

A formulation of narrowing which is more closely related to ours is given by Albert et al. [2] in which a "natural" big-step semantics is defined before an implementation driven small-step semantics is introduced. Both semantics are call-by-need, implement sharing, and are proved to be equivalent. They go on to extend the small-step semantics with additional features such as equational constraints and external functions. There is a difference in motive in comparison to our work, as they establish narrowing as a programming language feature whereas we are interested in using narrowing to analyse the operation of a program. The difference manifests itself in the theories: they relate their small-step semantics back to their defining big-step semantics, whereas we relate our lazy narrowing semantics back to the underlying functional semantics.

3.9 Conclusion

We have established the correctness of a narrowing semantics as an extension of a semantics for a minimal language. Our final formulation of the semantics is the result of several iterations and improvements, and captures the main ideas of narrowing in a simple and concise manner. In particular, the use of an underlying small-step semantics was instrumental in simplifying the theory. The simplicity along with the use of precise types enables a direct translation of our result to the Agda system [24].

In chapter 5 we develop the theory by defining a narrowing semantics for a more complex language which includes the novel notion of an overlapping pattern. There are a number of other interesting directions in which the theory could be extended, which we discuss at the end of that that chapter.

Chapter 4

Implementation and Evaluation

In the previous chapter we developed a theory of narrowing for a simple call-by-name functional language. In this chapter, we show how this idea can be realised in practice for a call-by-need language. We begin by defining our language and its big-step operational semantics. We then show how this can be implemented in Haskell, and how the resulting system can be used for property-based testing. Finally, we evaluate the performance of the system on a number of case studies.

4.1 Language and Semantics

The semantics for our prototype implementation differs in several ways from the semantics in the previous chapter. The biggest change is the shift to *call-by-need* evaluation which we explain shortly. We also use a big-step style of presentation as this style leads to a close relation with our implementation in Haskell. Finally, we move to a complete functional-language which includes algebraic datatypes and parametric polymorphism, which is suitable for expressing our examples.

Call-by-need For our formalisation in the previous chapter we used a *call-by-name semantics*, in which variable substitution occurs in place, i.e. by replacing any occurrences of the variable with the expression being substituted. Whereas this is convenient for formalisation it is often inefficient in practice as the substituted expression may have to be evaluated multiple times if it has been substituted in multiple places. To avoid this inefficiency while retaining a non-strict semantics in this chapter we use a *call-by-need* semantics. In call-by-need evaluation, substitutions are stored in an environment called a *heap*, and when a variable's value is required it is taken from the heap and evaluated. The variable's value is then stored back on the heap so it can be reused.

Evaluation sharing It is important to note that we discuss two different forms of evaluation sharing in this thesis – one relating to evaluation sharing between narrowing evaluations and the other via the *heap* in call-by-need evaluation. As our main concern is narrowing, we reserve the term *evaluation sharing* for the sharing between narrowing evaluations and refer to the sharing in *call-by-need* as evaluation reuse.

4.1.1 Language

The basis of our implementation is a core functional language (figure 4.1) which is similar to the core language of Haskell[36]. The top level of the language is given by function definitions which consist of a function identifier *Fun*, a list of arguments and an expression that forms the body of

$$
\begin{array}{lcl}
\mathit{Defn}_X & ::= & \mathit{Fun}\ \overline{\mathit{Var}} = \mathit{Expr}_X \\
\mathit{Expr}_X & ::= & \mathit{Fun}\ \overline{\mathit{Expr}_X} \ \mid\ \mathit{Con}\ \overline{\mathit{Expr}_X} \ \mid\ \texttt{var}\ \mathit{Var} \ \mid\ \texttt{fvar}\ X \\
 & \mid & \textbf{let}\ \mathit{Var} = \mathit{Expr}_X\ \textbf{in}\ \mathit{Expr}_X \\
 & \mid & \textbf{case}\ \mathit{Expr}_X\ \textbf{of}\ \overline{\mathit{Alt}_X} \ \mid\ \bot \\
\mathit{Val}_X & ::= & \mathit{Con}\ \overline{\mathit{Val}_X} \ \mid\ \texttt{fvar}\ X \ \mid\ \bot \\
\mathit{Alt}_X & ::= & \mathit{Con}\ \overline{\mathit{Var}} \to \mathit{Expr}_X
\end{array}
$$

Figure 4.1: The core language for our tool

the function. The expression can be: an application of a function, the application of a constructor, a variable, a free variable, a let binding, a case expression or bottom. Expressions should be well-typed under the standard typing rules, with each type having an associated set of constructors (the typing rules have been omitted as they are standard). Case expressions should have a complete set of alternatives. To simplify the semantics, we assume that function and constructor applications are complete however this is not a restriction in our implementation and the semantics of partial applications can be added using standard techniques [44]. Only function definitions in the language are allowed to be recursive, in particular we do not consider recursive **let** expressions. The bottom expression is primarily used to allow size limits on the inputs but is analogous to the Haskell equivalent.

We typically denote definitions by f, expressions by e, constructors by $\mathbf{c}$, closed variables by u and v, and free variables by x and y.

Narrowing

Similarly to section 3.4.1, we define refinements as a function from a set of typed variables to partial values of the given type: $\mathit{Sub}_{X \to Y} = X \to \mathit{Val}_Y$. The minimal narrowing set is a set of refinements each of which replaces a single variable with a constructor, and is given by

$$\mathit{Narr}_X(x^t) \;=\; \{x \,/\, \mathbf{c}\ \overline{y} \mid \mathbf{c} \in \mathit{cons}(t)\} \qquad\qquad \overline{y} \notin X$$

where x^t denotes x of type t and $\mathit{cons}(t)$ is the set of constructors of type t and the constructor application, $\mathbf{c}\ \overline{y}$, is complete with each variable having the correct type. The refinements form a monad, as defined in section 3.4.1.

4.1.2 Semantics

We define the call-by-need narrowing evaluation of our language using a big-step semantics. Each state in the evaluation is represented by an environment which consists of an expression and a heap, which is a mapping from variables to expressions used to avoid repeated evaluation of the same expression. We also record the substitution accumulated by narrowing steps. Formally, we have the following definitions:

$$\begin{aligned} \mathit{Heap}_X &\subseteq \mathit{Var} \times \mathit{Expr}_X \\ \mathit{Env}_X &= \mathit{Expr}_X \times \mathit{Heap}_X \\ \Downarrow &\subseteq \mathit{Env}_X \times (\mathit{Env}_Y \times \mathit{Sub}_{X \to Y}) \end{aligned}$$

We define evaluation to be complete when the expression is in *weak head normal form*, which occurs when the top-level expression is a constructor. Note, because we have restricted the language to complete applications there is no other type of expression in weak head normal form. This can be written as a "reflexive" rule:

$$\frac{}{\langle \mathbf{c}\ \overline{e},\ s\rangle \Downarrow \langle \mathbf{c}\ \overline{e},\ s,\ \mathit{return}\rangle}$$

This forms our base rule and states that any expression which is a constructor evaluates to itself with no update to the heap or substitution.

If the expression to be evaluated is a free variable, then a narrowing step is applied:

$$\frac{\sigma \in \mathit{Narr}_X(x) \qquad \langle \texttt{fvar}\ x[\sigma],\ s[\sigma]\rangle \Downarrow \langle e,\ s',\ \sigma'\rangle}{\langle \texttt{fvar}\ x,\ s\rangle \Downarrow \langle e,\ s',\ \sigma \ggg \sigma'\rangle}$$

A refinement is taken from the narrowing set and applied to both the current expression and the expressions on the heap. Note the semantics does not define how the refinement should be chosen from the narrowing set; this is left for the implementation. In this chapter we implement two possible methods: enumerating the refinements and random choice.

Variables are introduced to the heap during the evaluation of let expressions and function application:

$$\frac{\langle e',\ \{v \mapsto e\} \cup s\rangle \Downarrow \langle e'',\ s',\ \sigma\rangle}{\langle \textbf{let}\ v = e\ \textbf{in}\ e',\ s\rangle \Downarrow \langle e'',\ s',\ \sigma\rangle}$$

$$\frac{\langle e,\ \{\overline{v} \mapsto \overline{e}\} \cup s\rangle \Downarrow \langle e',\ s',\ \sigma\rangle \qquad (e, \overline{v}) = \mathit{fresh}(f)}{\langle f\ \overline{e},\ s\rangle \Downarrow \langle e',\ s',\ \sigma\rangle}$$

In which the term *fresh*(f) produces a new instantiation of f with fresh variables and $\overline{v} \mapsto \overline{e}$ is the mapping of each variable in $\overline{v}$ to its corresponding expression in $\overline{e}$.

The rule for evaluating a variable is:

$$\frac{\langle e,\ s\rangle \Downarrow \langle e',\ s',\ \sigma\rangle \qquad v \mapsto e \in s}{\langle \texttt{var}\ v,\ s\rangle \Downarrow \langle e',\ s'[v/e'],\ \sigma\rangle}$$

The variable's binding is taken from the heap, evaluated and then the heap is updated with the result.

The semantics of case expressions are defined in the standard way:

$$\frac{\langle e,\ s\rangle \Downarrow \langle \mathbf{c}\ \overline{e},\ s',\ \sigma\rangle \qquad \langle e'[\overline{v}/\overline{e}],\ s'\rangle \Downarrow \langle e'',\ s'',\ \sigma'\rangle \qquad \mathbf{c}\ \overline{v} \mapsto e' \in \overline{alt}}{\langle \mathbf{case}\ e\ \mathbf{of}\ \overline{alt},\ s\rangle \Downarrow \langle e'',\ s'',\ \sigma \ggg\!\!\Rightarrow \sigma'\rangle}$$

Note that the evaluation uses direct substitution and does not add variables to the heap. Instead we maximise evaluation reuse by converting expressions to an atomic form which utilises let expressions, which we describe after giving the semantic rules for $\bot$, which are as follows:

$$\frac{}{\langle \bot,\ s\rangle \Downarrow \langle \bot,\ s,\ \mathit{return}\rangle} \qquad \frac{\langle e,\ s\rangle \Downarrow \langle \bot,\ s',\ \sigma\rangle}{\langle \mathbf{case}\ e\ \mathbf{of}\ \overline{alt},\ s\rangle \Downarrow \langle \bot,\ s',\ \sigma\rangle}$$

The semantics are those of a standard error type: if $\bot$ is required for evaluation then it is propagated to the result.

Atomic Form

As part of our compilation we convert all constructors in our expressions into an atomic form by binding their non trivial fields to variables. A constructor is in atomic form if it has the following structure:

$$\mathit{Atom}_X ::= \mathit{Con}\ \overline{\mathit{Atom}_X} \ \mid\ \mathit{Var}\ \mid\ X$$

This atomic form is similar to that used in the Glasgow Haskell Compiler [44]. To convert an expression to this form we can bind the fields of constructors using let expressions, for example:

$$\mathbf{c}_1\,(\mathbf{c}_2\,e)\,e' \rightarrow \mathbf{let}\ v = e\ \mathbf{in}\ \mathbf{let}\ v' = e'\ \mathbf{in}\ \mathbf{c}_1\,(\mathbf{c}_2\ v)\,v' \qquad (e{,}e'\ \text{not atomic})$$

We use an atomic form rather than introducing variables from alternatives onto the heap in order to ensure evaluation reuse. In particular, if we have an expression of the following form,

> **case** v **of**
>
> $\quad \mathbf{c}\ \ v' \to \_$
>
> $\quad ..$

and v evaluates to $\mathbf{c}\, e$ then just adding $v' \mapsto e$ to the heap will not ensure evaluation reuse as v will still be bound to $\mathbf{c}\ e$ (and therefore e maybe re-evaluated without sharing). However if e is atomic then it is either a constructor, in which case there is no evaluation to be done, or a variable, in which case the evaluation will be shared.

4.2 Implementation

Our implementation of the semantics is an abstract machine written in Haskell. In this section, we give an overview by describing the surface syntax, providing an example of the conversion from semantics to a program and show how we handle the non-determinism of narrowing. The implementation is freely available online from:

`https://github.com/jonfowler/narrowcheck`

4.2.1 Syntax

The syntax we use for our language is a subset of Haskell which is desugared into the core language described in the previous section. Figure 4.2 shows the set function from section 3.2 converted from Haskell to our implementation. Several changes have been made. The *set′ where* clause has been

$$\textbf{data}\ \mathit{List}\ a = \mathit{Cons}\ a\ (\mathit{List}\ a) \mid \mathit{Empty}$$

$$\mathit{set} :: \mathit{List}\ \mathit{Nat} \to \mathit{Bool}$$

$$\mathit{set}\ \mathit{Empty} = \texttt{True}$$

$$\mathit{set}\ (\mathit{Cons}\ a\ l) = \mathit{set}'\ a\ l$$

$$\mathit{set}'\ a\ \mathit{Empty} = \texttt{True}$$

$$\mathit{set}'\ a\ (\mathit{Cons}\ a'\ l) = a < a' \ \wedge\ \mathit{set}'\ a'\ l$$

Figure 4.2: The *set* function from section 3.2 in our implementation.

replaced with a top-level definition, the function has been made monomorphic to remove the need for the *Ord* typeclass, the syntactic sugar for lists has been replaced and the guards have been removed. Apart from these changes the function remains the same. In examples we still use Haskell for convenience, however all the examples can be converted into our language and the code can be found in the examples folder of the repository [25].

To obtain the core language, nested pattern matches are desugared to individual case expressions using the method described by Wadler [43, Chapter 5]. Additionally we convert to atomic form as described in the previous section.

4.2.2 Semantics to Implementation

The semantics from the previous section are converted into an evaluation function. The function utilises a monad, *Narrow*, which is a combination of a state monad, which stores the evaluation environment including the heap, and a non-deterministic monad, *Refine*, which tracks the refinements made during narrowing and is explained in the next section.

As an example we show the part of the *eval* function which evaluates a variable by retrieving its value from the heap, evaluating it and updating the heap with the result. The function utilises two helper functions $getVar :: Var \to Narrow\ Expr$, which gets the value of a variable from the heap; and $updateVar :: Var \to Expr \to Narrow\ ()$, which updates the value of a variable on the heap.

```
type Narrow = StateT Env Refine

eval :: Expr → Narrow (Expr, Sub)
eval ...
eval (var v) = do
  e ← getVar v
  (e′, σ) ← eval e
  updateVar v e′
  return (e′, σ)
```

4.2.3 Search Tree

To capture the non-determinism of narrowing we use a search tree to represent the result of evaluation. The branches of the trees are possible narrowing refinements and the leaves contain the result of evaluation. Different search strategies can be defined by different traversals of the tree. The data type and monad instance are as follows:

```
data Refinement = Refinement {refinement :: Sub, freq :: Int}

data Refine a = Branch [(Refinement, Refine a)]
              | Leaf a
```

```
instance Monad Refine where
  return = Leaf
  Leaf a >>= f = f a
  Branch as >>= f = Branch $ (map ∘ fmap) f as
```

The branches of the tree are labelled with the refinement they represent and the frequency with which the refinement should occur in a random search strategy (section 4.3.3). Similarly to composing refinements (section 3.4.2), we note that this is a free monad on the underlying functor **data** *BranchF a* = *BranchF* [(*Refinement*, *a*)]. We can now lift the branching logic into a monad operator *branch*, which hides the tree structure:

```
branch :: [Refinement] → Narrow Refinement
branch rs = StateT $ λs → Branch (map (λr → (r, Leaf (r, s))) rs)
```

Finally, we can implement the narrowing rule as part of the *eval* function:

```
eval :: Expr → Narrow (Expr, Sub)
eval . .
eval (fvar x) = do
  σ ← branch $ narrowingSet x
  modifyHeap (fmap (subst σ))   -- update the heap
  (e, σ′) ← eval (subst σ (fvar x))
  pure (e, σ >=> σ′)
```

The definition uses three helper functions: *narrowingSet* :: *FreeVar* → [*Refinement*], which gives the narrowing set for a typed free variables; *modifyHeap* :: (*Heap* → *Heap*) → *Narrow* (), which modifies the heap; and *subst*, which substitutes a refinement into an expression.

The above definition updates the heap as soon as refinement is made, while this reflects the semantics it is somewhat expensive in practice as the heap is repeatedly traversed. To avoid this, in the actual implementation we store the refinements as part of the environment and only apply the refinement when a free variable is encountered.

4.2.4 Running a narrowing evaluation

Running a narrowing evaluation is now quite simple. We wrap the *eval* function in a simple helper, by means of the following definition:

$$\begin{aligned}&\mathit{narrow} :: \mathit{Expr} \to \mathit{Env} \to \mathit{Refine}\ (\mathit{Expr}, \mathit{Sub})\\ &\mathit{narrow}\ e\ \mathit{env} = \mathit{evalStateT}\ (\mathit{eval}\ e)\ \mathit{env}\end{aligned}$$

The *narrow* function takes an expression and an evaluation environment and produces a tree of results. For example we can use it to evaluate the *union* example from section 3.2 as follows

$$\mathit{narrow}\ \{\mathit{seqN}\ 2\ \$\ \mathit{union}\ (\texttt{fvar}\, x)\ (\texttt{fvar}\, y)\}\ \mathit{unionEnv}$$

where the term inside the curly braces is an expression in our language with *seqN* a function that forces the first n values of a list, and the *unionEnv* is an evaluation environment with an empty heap and contains function definitions for *union*, *seqN* and their dependencies. This evaluates to a tree containing results and the refinements on the leaves, such as the following:

$$0\ :\ 1\ :\ l \qquad\qquad \{x \mapsto [0],\ y \mapsto 1 : l\}$$

On the left we have the result of the union of x and y, in which the first two elements have been forced and l remains *free*, and on the right is the substitution created by the narrowing evaluation.

4.3 Property-Based Testing

This section describes how the implementation that we have developed above can be used for property-based testing.

4.3.1 Properties

Properties in our system are functions with return type *Result*, which represents three possible outcomes: a failed precondition, in which case the test case is invalid; a successful result, where the test case satisfies the property; or a failure, where the test case is a counterexample:

$$\textbf{data}\ \mathit{Result} = \mathit{Invalid} \mid \mathit{Success} \mid \mathit{Failure}$$

Properties are typically defined using a specialised implication operator $(\Longrightarrow) :: \mathit{Bool} \rightarrow \mathit{Bool} \rightarrow \mathit{Result}$, defined as follows:

$$\begin{aligned}
&(\Longrightarrow) :: \mathit{Bool} \rightarrow \mathit{Bool} \rightarrow \mathit{Result}\\
&\texttt{False} \Longrightarrow \_ = \mathit{Invalid}\\
&\texttt{True} \Longrightarrow b = \mathit{property}\ b
\end{aligned}$$

$$\begin{aligned}
&\mathit{property} :: \mathit{Bool} \rightarrow \mathit{Result}\\
&\mathit{property}\ \texttt{False} = \mathit{Failure}\\
&\mathit{property}\ \texttt{True} = \mathit{Success}
\end{aligned}$$

Property-based testing can then be run on any typed, first-order, monomorphic definition with a return type of *Result*. For example, we can test the *propUnion* property from section 3.2, as follows:

```
$ narrowcheck -p propUnion Union.hs
Property refuted with input:
[0]
[0]
```

This example runs a random test, of up to one hundred test cases, of the property *propUnion* specified by the `-p` flag. A narrowing evaluation strategy is used on the property with generated free variable arguments, i.e. *propUnion* (`fvar` x) (`fvar` y). This particular test run records a failure with the two arguments, [0] and [0], using the faulty implementation of *union* which does not remove duplicate elements.

4.3.2 Size Limit

In enumerative and random testing it is often important to limit the size of test cases. In enumerative testing a size limit ensures the pool of test cases is finite, and in random testing a size limit stops test cases getting too big and possibly even growing infinitely (it is possible to have a distribution in which a test case is finite with probability less than 1). To achieve this, we enforce a maximum constructor depth on each test case. We define the depth over values without free variables but with bottoms $\mathit{Val}_\bot = \mathit{Val}_\emptyset$. This allows us to use the same definition for both a narrowing evaluation strategy and the basic evaluation strategy which we compare it against. We define define $\mathit{depth}_n(t) \subseteq \mathit{Val}_\bot$ as the set of values with depth equal to or less than n of type t by the following rules,

$$\frac{\forall\, \mathbf{c} \in t.\ \neg \mathit{fieldless}(\mathbf{c})}{\bot \in \mathit{depth}_0(t)} \qquad \frac{\mathit{fieldless}(\mathbf{c})}{\mathbf{c}\ \emptyset \in \mathit{depth}_0(t)} \qquad \frac{\forall a \in \overline{a}.\ a \in \mathit{depth}_n(t)}{\mathbf{c}\ \overline{a} \in \mathit{depth}_{n+1}(t)}$$

where *fieldless* is a predicate asserting a constructor has no fields. Fieldless constructors are given zero depth, and therefore any type with a trivial constructor has a value at any depth. For types without a fieldless constructor we include $\bot$ in the zero depth set. It should be noted that $\bot$ will only appear in fields of zero depth whereas fieldless constructors can be lifted to any depth, e.g. $\mathit{Left}\ \bot \in \mathit{depth}_1(\mathit{Either\ Bool\ Bool})$ but $\bot \notin \mathit{depth}_1(\mathit{Either\ Bool\ Bool})$ whereas $\mathit{False} \in \mathit{depth}_n(\mathit{Bool})$ for all n.

This definition of *depth* has been chosen carefully to give a fair comparison between the basic evaluation strategy, which will evaluate each test case in a set individually, and narrowing, which might reach a conclusion for a group of test cases. The use of $\bot$ ensures that the basic method considers all the same test cases but as $\bot$ only appears in fields where there is no other valid value there are no superfluous test cases. If a test evaluates to $\bot$ the result is taken to be invalid.

We can add a size limit to a narrowing evaluation by updating our definition of the narrowing set,

$$\begin{array}{rcll}
\mathit{Narr}_X(x^t_0) & = & \{\mathbf{c}\,\emptyset \mid \mathbf{c} \in \mathit{cons}(t), \mathit{fieldless}(\mathbf{c})\} & \exists\, \mathbf{c} \in t.\ \mathit{fieldless}(\mathbf{c}) \\
\mathit{Narr}_X(x^t_0) & = & \{\bot\} & \forall\, \mathbf{c} \in t.\ \neg\mathit{fieldless}(\mathbf{c}) \\
\mathit{Narr}_X(x^t_{n+1}) & = & \{x \,/\, \mathbf{c}\ \overline{y_n} \mid \mathbf{c} \in \mathit{cons}(t)\} & \overline{y_n} \notin X
\end{array}$$

where x_n denotes a free variable decorated with a maximum depth of n,

4.3.3 Search Strategies

We consider two enumeration-based search strategies and one random search strategy. For enumeration, we use one strategy which implements narrowing with shared evaluation and one which implements narrowing without shared evaluation, which allows us to compare the performance impacts of narrowing and sharing evaluation separately.

Enumeration

The full narrowing enumerative strategy simply explores the tree from left to right, and returns all the results as a list:

$$\begin{aligned}&\mathit{enumerate} :: \mathit{Refine}\ a \to [\,a\,]\\&\mathit{enumerate}\ (\mathit{Leaf}\ \ a) = [\,a\,]\\&\mathit{enumerate}\ (\mathit{Branch}\ \mathit{bs}) = \mathit{concatMap}\ (\mathit{enumerate} \circ \mathit{snd})\ \mathit{bs}\end{aligned}$$

This definition of *enumerate* will benefit from evaluation sharing when the search tree is produced from a narrowing evaluation, as at each branch the evaluation performed so far will be reused in each sub-tree.

To implement evaluation without sharing, we need to restart evaluation for each test case. To do so, we keep track of the next test case to be performed with a path through the search tree and regenerate the search tree each time. The implementation is shown in Figure 4.3.

Random Search

For our random search we give each constructor a weighting which determines the frequency with which it is chosen in the random search. To declare the weights in the implementation we use a pragma called DIST (short for distribution). For example, we can give weights to the leaf and node constructors of a binary search tree as follows:

$$\begin{aligned}&\textbf{data}\ \mathit{BST}\ \ a = \mathit{Node}\ (\mathit{BST}\ \ a)\ a\ (\mathit{BST}\ \ a) \mid \mathit{End}\end{aligned}$$

```
{-# DIST Node 4 #-}
{-# DIST End 1 #-}
```

```
  -- A path is an infinite stream of Ints, each representing the next
  -- branch of the tree to evaluate:
type Path = [Int]
  -- Evaluate the branch corresponding to a path and return the
  -- next path if one still exists:
evaluatePath :: Refine a → Path → (Maybe Path, a)
evaluatePath (Leaf a) _ = (Nothing, a)
evaluatePath (Branch bs) (i : p)
  = case evaluatePath (bs !! i) p of
      (Nothing, a)
        | (i + 1) ≡ length bs → (Nothing, a)
        | otherwise → (Just (i + 1 : repeat 0), a)
      (Just p, a) → (Just (i : p), a)
  -- Enumerate paths in order, we accept a function as an
  -- argument so Haskell does not store the evaluation tree:
noShareEnumerate :: (b → Refine a) → b → [a]
noShareEnumerate f b = go f b (Just (repeat 0))
  where go _ _ Nothing = []
        go f b (Just p)
          = let (p', a) = evaluatePath (f b) p
            in a : go f b p'
```

Figure 4.3: An enumeration strategy without evaluation sharing.

Our random search implements backtracking. That is, when it arrives at an invalid leaf or empty branch the last step is undone and a different branch is tried. It is desirable to limit this process as otherwise the search might get stuck in a locale of the search tree in which there are no possible solutions. There are two simple ways to limit backtracking. One is to limit the total number of backtracking steps, such as the approach taken by Claessen at al. [15]. The approach we take is to limit the number of backtracking steps taken from any leaf. This makes it easier to reason about the effect of backtracking because this form of backtracking has no global state and therefore can be reasoned about locally.

There is a trade off between the size of the backtrack limit and the fidelity of the distribution. Without a backtrack limit, the frequency of a refinement in the distribution will be proportional to its weight, so long as there is a valid test case which includes that refinement. With a backtrack limit this is not the case as generation may fail even if there is a valid test case which uses refinement. The lower the backtrack limit the higher the probability of failure resulting in a greater skew of the distribution. We explore the effects of different backtrack limits in the case studies (§4.4).

A basic implementation of a random generator can be found in Figure 4.4. It is a slight simplification of our actual generator which also uses a strategy similar to that of non-sharing enumeration to stop evaluation being shared between runs. For random testing this is sensible as it avoids high memory use[1] and as the evaluation of random test cases often deviate early, that is they take a different branch of the program, sharing gives little performance benefit. It should be noted that shared evaluation still

[1]Note that for enumeration we do not have high memory use because we only need to store the current evaluation path, as once part of the evaluation tree has been enumerated it is not reused and so can be garbage collected

```
import System.Random
  -- Select a random element from a non-empty list and also
  -- return the rest of the list
takeRand :: [a] → State StdGen (a, [a])
takeRand l = do
  i ← state $ randomR (0 . . length l − 1)
  let (l1, a : l2) = splitAt i l
  return (a, l1 ++ l2)
randomSearch :: Int → Refine (Maybe a) → Rand (Maybe a)
randomSearch backtrack t = go t [ ] where
  go (Leaf (Just a)) _ = return (Just a)
  go (Branch bs) backups | ¬ (null bs) = do
    (t′, bs′) ← takeRand bs
    go t′ (take backtrack (bs′ : backups))
  go _ (bs : backups) = go (Branch bs) backups
  go _ _ = return Nothing
```

Figure 4.4: Implementation of a random, backtracking search

occurs within a run through the backtracking process.

4.4 Case Studies

We explore the performance of our narrowing based tool in a number of case studies. In each case we test a property of a program by executing the property with our narrowing based evaluation. We compare a selection of metrics to a basic strategy in which test data is generated generically and then applied to the property and executed traditionally. The code for the case studies can be found in appendix A and on Github [25].

We look at three different programs which highlight different features a property might have. The aim is to provide a discussion of the main factors which affect the performance of testing a property. A more exhaustive benchmark with a larger number of properties is provided in chapter 6 where we compare the narrowing tool to an extended overlapping narrowing tool. Here and there we have only compared the performance of our tool against different variants of itself. In an ideal world we would have also compared the performance to existing property-based testing solutions, we have not done so because making such a comparison meaningful is difficult as for convenience our tool runs in an abstract machine and that carries a large performance penalty.

One important discussion is the effect of the backtrack limit on the performance of random testing and the distribution of generated test cases. For our main comparison we use a conservative limit of 3. At this level backtracking has only a small impact on performance which allows a direct comparison to the basic method.

4.4.1 Basic Strategy

The basic strategy is to generate an input and then apply it to the property. We use the same construction depth limit and in random testing generate constructors with the same weights as we do with the narrowing tool. The tests run in an equivalent abstract machine for direct comparison.

4.4.2 Evaluation

For enumerative testing we repeat each experiment ten times with a time limit of twelve minutes. For random testing we run one thousand test cases and repeat each experiment forty times with a time limit of four minutes. For both we increase the construction depth incrementally until an experiment exceeds the time limit. The limits were chosen after experimentation to give us a range of results within a reasonable time while allowing for repetition.

All results reported were obtained using a quad-core Intel i5 running at 3.2GHz, with 16GB RAM, under 64-bit Ubuntu 16.04 LTS with kernel 4.4.0. The program is written without the use of parallel features and so the number of cores is expected to only have a limited effect on performance.

4.4.3 Metrics

Along with the time taken we measure a number of other metrics in order to give us an insight into the drivers of performance. For enumerative testing we measure the number of generated and invalid tests considered. For random testing we measure the success rate of generating values and the average size of values.

Number of test cases and invalid tests cases

We count the number of generated test cases (which might have either satisfied or refuted the property) along with the number of invalid possibilities tried. The sum of these two metrics is equal to the number of leaves in the search tree. These metrics are the same for narrowing with and without sharing but differ in the basic strategy. We refer to them as *tests* and *invalid* respectively in results tables.

Success Rate

For random testing we measure the percent of test cases which satisfy the precondition and thereby perform a valid test. We call this the success rate and it is an important measurement as a low success rate may indicate a bad distribution of test cases.

Average Size

In order to assess the distribution of randomly generated values we measure their average size. Unlike other tests we use a problem-specific measurement of size. This is usually the size of the spine structure, e.g. the length of a list or the number of nodes in a tree.

4.4.4 Union of Sets

We begin by evaluating the union property on which we originally demonstrated narrowing (§3.2). The property, which we specialise for natural numbers, is defined as follows:

$$\mathbf{data}\ \mathit{Nat} = Z \mid S\ \mathit{Nat}$$

$$\mathit{propUnion} :: [\mathit{Nat}] \to [\mathit{Nat}] \to \mathit{Bool}$$

$$\mathit{propUnion}\ x\ y = \mathit{set}\ x\ \wedge\ \mathit{set}\ y\ \implies\ \mathit{set}\ (\mathit{union}\ x\ y)$$

The property asserts that if the two input lists satisfy the *set* condition then their union should also. We recall that the sets are represented by strictly increasing lists and test a faulty implementation of union (Fig. 4.5).

The performance results for testing the union property are given in Figure 4.6. The tables, 4.6a and 4.6b, show the metrics for enumerative

$$
\begin{array}{l}
set :: [\,Nat\,] \to Bool \\
set\ [\,] = \texttt{True} \\
set\ (a : l) = set'\ a\ l \\
\quad \textbf{where}\ set'\ a\ [\,] = \texttt{True} \\
\qquad\qquad set'\ a\ (a' : l) = a < a' \ \wedge\ set'\ a'\ l \\
union :: [\,Nat\,] \to [\,Nat\,] \to Bool \\
union\ [\,]\ l = l \\
union\ l\ [\,] = l \\
union\ (a : l)\ (a' : l')\ \mid a < a' \qquad = a : union\ l\ (a' : l') \\
\qquad\qquad\qquad\qquad\quad \mid otherwise = a' : union\ (a : l)\ l'
\end{array}
$$

Figure 4.5: The set condition and a faulty implementation of union

and random search at a selection of construction depths (including the maximum construction depth for each problem). The two graphs, 4.6c and 4.6d, show the time taken across the construction depths. Note that we give graphs for enumerative testing a logarithmic scale, owing to the typically exponential performance, whereas the graphs for random testing have a linear scale.

Enumeration

The results for the enumerative evaluation can be found in the graph 4.6c and the table 4.6a.

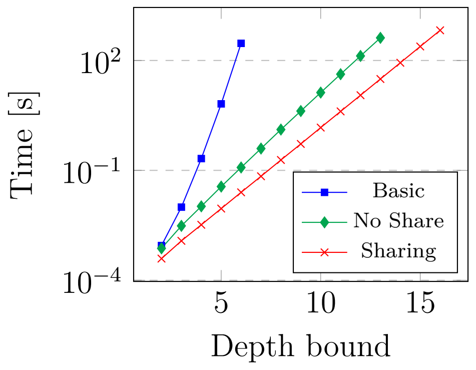
Observation 1. The performance of the narrowing with sharing tool is several orders of magnitudes faster than the basic method. The narrowing tool can produce 710,000 test cases with a maximum depth of 15 within the time limit whereas the basic method is only able to produce 441 of size

Strat	**Metric**	**5**	**6**	**7**	**..**	**12**	**..**	**15**
Basic	time	6.54s	294s	-	..	-	..	-
	tests	169	441	-	..	-	..	-
	invalid	1.1E5	3.8E6	-	..	-	..	-
Narr	no share	36.1ms	120ms	394ms	..	133s	..	-
	sharing	9.05ms	25.1ms	68.9ms	..	11.3s	..	241s
	tests	104	248	596	..	4.9E4	..	7.1E5
	invalid	105	300	870	..	1.7E5	..	3.8E6

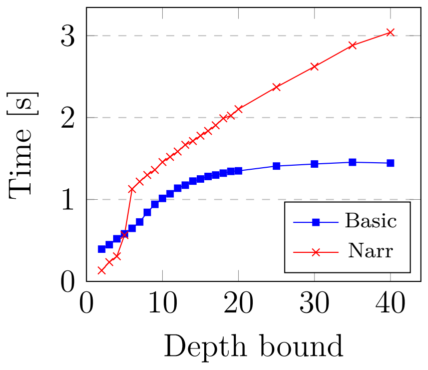
(a) Benchmark results for testing the union property by enumeration to given construction depth.

Strat	**Metric**	**5**	**10**	**15**	**20**	∞
Basic	time	581ms	1.01s	1.25s	1.35s	1.46s
	success	12.0%	12.2%	12.2%	12.2%	12.2%
	size	0.65	0.66	0.66	0.66	0.66
Narr	time	566ms	1.46s	1.78s	2.10s	4.21s
	success	50.7%	36.2%	49.7%	60.9%	100.0%
	size	1.75	1.70	2.23	2.70	5.01

(b) Benchmark results for testing the union property by random generation with maximum construction depth. The *size* metric is the average list length of all the inputs.



(c) Time taken to test the union property by enumeration.



(d) Testing union problem on 1000 randomly generated inputs.

Figure 4.6: Benchmark results for testing the **union** property

6.

This can be explained by the narrowing tool's ability to determine the result of partially bound inputs, as shown step-by-step in section 3.2. For example, if the partial input begins with two zeroes, $(0 : 0 : l)$, the tool already determines it is *Invalid* and no further inputs of this form need to be tried. In contrast the basic method has to try all inputs of this form within the construction depth. In the table 4.6a we can see at construction depth 6 there is only 441 combinations where both lists satisfy the set condition but over 3,000,000 where they don't.

The ability to determine partially bound inputs also explains the discrepancy in the number of test cases generated between the two methods, with the narrowing tool generating fewer test cases. This is because it is possible for a partially bound input to be a successful test case. For example, the partial input consisting of $[\,]$ and $[n]$ already satisfies the property for any value of n.

Observation 2. The majority of the performance increase happens due to narrowing evaluation but shared evaluation also provides a performance benefit. At depth 6, narrowing with sharing is 4.8 times faster than narrowing without sharing but narrowing without sharing is 2453 times faster than basic evaluation.

Random

For the random search we use a weighting of 1 for the empty constructor and 5 for the cons constructor. The zero and successor constructor are given equal weightings. The usual backtrack limit of 3 is used for the narrowing tool. The results can be found in the graph 4.6d and the table 4.6b.

Observation 3. Although the basic strategy performs quickly it actually fails to produce anything but trivial values. This can be observed as the average length of the lists remains at 0.66 with the strategy only producing a successful value 12.2% of the time.

This is unsurprising considering at depth 5 the basic method is choosing between over 100,000 test cases of which only 169 are valid (note that the comparatively high success rate occurs as the distribution is not uniform). Of the 12.2% success rate, 9.3% are accounted for test cases where both inputs have a length of either one or less and are therefore trivially ordered. Therefore testing with this method is not very effective.

Observation 4. Using a narrowing strategy and with the depth limit above 10, the success rate and average size of the test cases generated increases with the depth limit. Without a depth limit the success rate is 100% and the average size is 5.

The reason the success rate increases with depth is due to properties of the *set* function. In particular, generating the final value of a set is only possible if the depth limit allows the value to be strictly greater than the previous one. As the depth limit increases it is more likely that there is a greater value and without a depth limit there will always be.

The average size of 5, which occurs when testing without a depth limit, is the expected average as the length of the list forms a geometrical distribution.

Observation 5. Increasing the backtrack limit to 30, or removing it entirely, increased the success rate of generating a value to 100% while improving performance (table 4.7). This also resulted in an increase in the

Strat	**Metric**	**5**	**10**	**15**	**20**	∞
3	time	566ms	1.46s	1.78s	2.10s	4.21s
	success	50.7%	36.2%	49.7%	60.9%	100.0%
	size	1.75	1.70	2.23	2.70	5.01
30	time	386ms	764ms	1.16s	1.55s	4.19s
	success	100.0%	100.0%	100.0%	100.0%	100.0%
	size	1.81	2.61	3.23	3.69	5.00
∞	time	388ms	762ms	1.16s	1.55s	4.14s
	success	100.0%	100.0%	100.0%	100.0%	100.0%
	size	1.82	2.61	3.22	3.68	4.98

Figure 4.7: Performance results for different backtrack limits testing the union property randomly with the narrowing tool.

average length of generated test cases in the tests where a construction depth limit was imposed.

Typically, increasing the backtrack limit will result in worse performance as the tool will have to search harder for a solution, often with diminishing returns. For the *union* property this is not the case. This is because when trying to find a final value in a set it is also possible to backtrack further, ending the set rather than generating the value. At a backtrack depth of 3 this is rarely possible however with backtrack limit of 30 it is almost always possible for our distribution.

4.4.5 Reverse

We consider the staple property-based testing example of appending reversed lists:

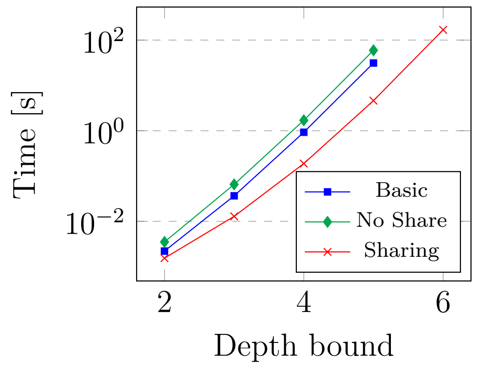
$$\begin{aligned}&\mathit{propReverse} :: [\mathit{Nat}] \to [\mathit{Nat}] \to \mathit{Result}\\&\mathit{propReverse}\ \mathit{as}\ \mathit{bs} = \mathit{property}\ \$\\&\qquad \mathit{reverse}\ (\mathit{as} \mathbin{+\!\!+} \mathit{bs}) \equiv (\mathit{reverse}\ \mathit{bs} \mathbin{+\!\!+} \mathit{reverse}\ \mathit{as})\end{aligned}$$

Strat	**Metric**	**3**	**4**	**5**	**6**
Basic	time	36.6ms	925ms	31.2s	-
	tests	256	4225	1.1E5	-
	invalid	0	0	0	-
Narr	no share	65.4ms	1.70s	59.5s	-
	sharing	12.9ms	189ms	4.61s	168s
	tests	256	4225	1.1E5	3.8E6
	invalid	0	0	0	0

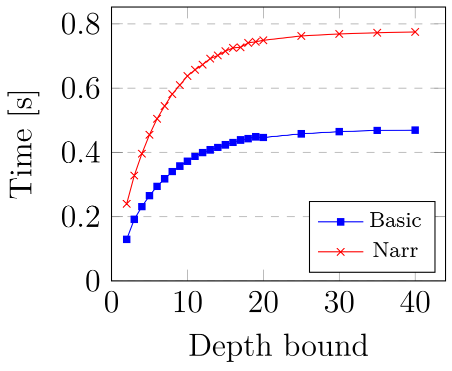
(a) Benchmark results for testing the reverse property by enumeration to given construction depth

Strat	**Metric**	**5**	**10**	**15**	**20**	∞
Basic	time	265ms	373ms	424ms	446ms	470ms
	success	100.0%	100.0%	100.0%	100.0%	100.0%
	size	2.98	4.17	4.66	4.85	5.01
Narr	time	455ms	638ms	715ms	749ms	776ms
	success	100.0%	100.0%	100.0%	100.0%	100.0%
	size	2.99	4.20	4.68	4.86	5.00

(b) Benchmark results for testing the reverse property by random generation with maximum construction depth



(c) Time taken to test the reverse property by enumeration



(d) Testing reverse problem on 1000 randomly generated inputs

Figure 4.8: Benchmark results for testing the **reverse** property

For this property there is no precondition and therefore all test cases are valid. We use the following correct implementation of reverse:

```
reverse :: [a] → [a]
reverse l = go [] l
  where go acc [] = acc
        go acc (a : l) = go (a : acc) l
```

Enumeration

The results for the enumerative evaluation can be found in the table 4.8a and the graph 4.8c.

Observation 6. The full narrowing tool has better performance than the basic evaluation strategy. For example, at depth five the narrowing strategy completes on average around seven times faster than the basic method.

Observation 7. Narrowing without sharing is the worst performing strategy, taking around twice as long as the basic strategy at depth five.

These two observations together suggest that shared evaluation is beneficial for performance but narrowing does not improve performance. To explain this we note that *propReverse* is *hyper-strict*, that is it requires the entire input to be evaluated to produce a result. This is because the equality needs to check that every element of the list is the same in order to conclude the two lists are equal and as this is a correct implementation this always happens. This can be seen experimentally as both the basic and narrowing strategies evaluate the same number of test cases. As the property is *hyper-strict* narrowing never produces an answer for a range of inputs and therefore is not beneficial.

Sharing evaluation however does produce a significant benefit as large parts of the evaluation do not need to be repeated. For example, the structure of the reversed list will be evaluated first and so this evaluation will be shared between any inputs with the same list structure. Narrowing without sharing gives the worst performance as it has the overhead of having to perform substitution in narrowing steps but does not benefit from sharing.

Random

For the random search we use a weighting of 1 for the empty constructor and 5 for the cons constructor for the list type and a equal weightings to the zero and successor constructor. We chose these weightings as experimentation showed they gave a reasonable distribution of test cases.

Observation 8. Both strategies have a 100% success rate as there is no precondition and the test sets do contain any $\perp$ terms.

Observation 9. The narrowing strategy is slower. Without a depth limit it takes 776ms whereas the basic strategy takes 460ms, which is 1.65 times faster.

As there is no precondition or invalid test cases there is no requirement to backtrack and therefore the random strategy does not benefit from any form of evaluation sharing. This result is consistent with the non-sharing result for enumeration. As no backtracking is necessary the backtrack limit does not affect performance and therefore no comparison is provided.

4.4.6 Ordered Trees

Next we consider an implementation of a delete function for an ordered tree. The property we check asserts that the tree remains ordered after an element is deleted from it:

$$\textbf{data } \mathit{Tree} = \mathit{Node}\ \mathit{Tree}\ \mathit{Nat}\ \mathit{Tree} \mid \mathit{Leaf}$$

$$\mathit{propDel}\ a\ t = \mathit{ord}\ t \implies \mathit{ord}\ (\mathit{del}\ a\ t)$$

The definitions for the ordered condition and delete function can be found in figure 4.9. It should be noted that there are more efficient implementations of the ordered condition. In particular linear time implementations exist whereas this implementation has a worst case of quadratic time. We use this inefficient version as it demonstrates a key issue with narrowing and random testing.

Enumeration

The results for the enumerative evaluation can be found in the table 4.10a and the graph 4.10c.

Observation 10. Both methodologies can enumerate all the test cases up to construction depth 4, with the full narrowing tool being approximately 100 times faster than the basic method.

This perfomance difference can once again be explained by the narrowing tool's ability to prune the search space by evaluating partial inputs. There are 238,145 trees at construction depth 4, of which only 2004 satisfy the *ord* constraint. With the basic method of evaluation each tree has to be tested separately in combination with the 5 possible values for the second argument of *propDel*, the element being deleted. The narrowing evaluation benefits by discarding many partial unordered trees before they are completed, testing under 10,000 partial inputs in total.

The size of the search space increases exponentially and there are over

$$
\begin{aligned}
&ord :: Tree \to Bool\\
&ord\ Leaf = \texttt{True}\\
&ord\ (Node\ t1\ \ a\ t2) \ = \ allT\ (\leqslant a)\ t1 \ \wedge\ ord\ t1\\
&\qquad\qquad\qquad\qquad \wedge\ allT\ (\geqslant a)\ t2 \ \wedge\ ord\ t2\\
&allT :: (Nat \to Tree) \to Tree \to Bool\\
&allT\ p\ Leaf = \texttt{True}\\
&allT\ p\ (Node\ t1\ \ a\ t2) = p\ a\ \wedge\ allT\ t1\ \wedge\ allT\ t2\\
&del :: Nat \to Tree \to Tree\\
&del\ \_\ Leaf = Leaf\\
&del\ a\ (Node\ t1\ \ a'\ t2)\mid a < a' = Node\ (del\ a\ t1)\ a'\ t2\\
&\qquad\qquad\qquad\qquad\mid a > a' = Node\ t1\ \ a'\ (del\ a\ t2)\\
&\qquad\qquad\qquad\qquad\mid otherwise = append\ t1\ \ t2\\
&append :: Tree \to Tree \to Tree\\
&append\ Leaf\ \ t = t\\
&append\ (Node\ t1\ \ a\ t2)\ t3 = Node\ t1\ \ a\ (append\ t2\ t3)
\end{aligned}
$$

Figure 4.9: Implementation of the ordered condition, *ord*, and delete function, *del*

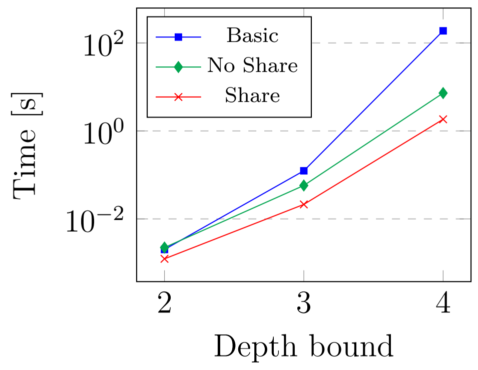
Strat	**Metric**	**2**	**3**	**4**	**5**
Basic	time	2.02ms	124ms	189s	-
	tests	21	228	1.0E4	-
	invalid	6	748	1.2E6	-
Narr	no share	2.23ms	57.6ms	7.26s	-
	sharing	1.24ms	21.4ms	1.84s	-
	tests	13	122	4593	-
	invalid	2	41	4186	-

(a) Benchmark results for testing the ordered tree property by enumeration to given construction depth

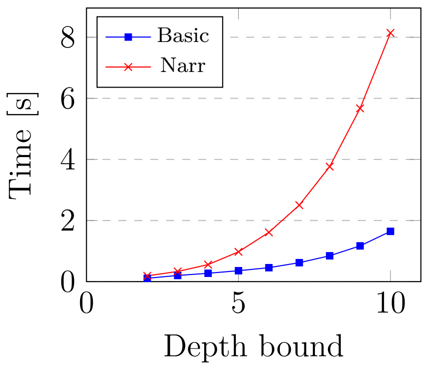
Strat	**Metric**	**4**	**5**	**6**	**7**	**8**	**9**
Basic	time	273ms	357ms	456ms	620ms	845ms	1.17s*
	success	50.2%	45.5%	44.7%	44.1%	43.9%	43.9%
	size	0.97	0.62	0.45	0.38	0.36	0.35
Narr	time	557ms	974ms	1.61s	2.50s	3.76s	5.67s*
	success	67.9%	54.1%	49.1%	47.7%	47.6%	47.0%
	size	1.85	1.28	0.88	0.70	0.61	0.58

* Experiment discontinued due to declining size rather than time limit.

(b) Benchmark results for testing the ordered tree property by random generation with maximum construction depth. The size metric is the average number of nodes in generated tree.



(c) Time taken to test the ordered tree property by enumeration



(d) Testing ordered tree property on 1000 randomly generated inputs

Figure 4.10: Benchmark results for testing the **ordered tree** property

200 billion trees of construction depth 6. Hence it is not surprising that neither method is able to enumerate the test cases at this depth.

Observation 11. Evaluation sharing consistently performs better with a four times speedup at depth 4.

Random

For the random testing, we use a weighting of two for the nodes and one for the leaves. The usual backtracking limit of 3 is used for the narrowing tool. The size metric is the average number of nodes in the tree. The results can be found in the table 4.10b and the graph 4.10d.

Observation 12. Neither method is effective at random testing, with the average size of tree declining as the construction depth increases.

The reason the size of generated trees decreases is due to the relation between the size of a tree and the probability that it is ordered. For large trees, with lots of elements, it is highly likely that some of the elements are unordered and therefore such trees are unlikely to be valid test cases. Correspondingly, small trees have a higher probability of being ordered and therefore are more likely to make successful test cases, reducing the average size.

It may still be surprising that the average size decreases rather than staying roughly the same. This happens because of the construction limit which causes a proportionally higher number of medium suchThat trees to be generated when the limit is low. As before, these are more likely to be ordered than the larger trees generated at higher construction depth.

The above explanation suffices to explain why the basic method generates smaller trees but does not explain why the narrowing tool also has

Strat	**Metric**	**4**	**5**	**6**	**7**	**8**	**9**
3	time	557ms	974ms	1.61s	2.50s	3.76s	5.67s
	success	67.9%	54.1%	49.1%	47.7%	47.6%	47.0%
	size	1.85	1.28	0.88	0.70	0.61	0.58
30	time	967ms	20.6s	60.5s	122s	-	-
	success	100.0%	79.6%	66.4%	61.7%	-	-
	size	3.10	3.10	3.07	3.26	-	-
∞	time	971ms	-	-	-	-	-
	success	100.0%	-	-	-	-	-
	size	3.13	-	-	-	-	-

Figure 4.11: Performance results for different backtrack limits testing the ordered tree property randomly with the narrowing tool.

this flaw. To explain, we consider the partial tree, *Node t1 a t2*, for which the following condition should be satisfied:

$$allT\ (\leqslant a)\ t1 \ \wedge\ ord\ t1 \ \wedge\ allT\ (\geqslant a)\ t2 \ \wedge\ ord\ t2$$

The first two conditions, that all elements in the left sub-tree are less than the node element and that the left sub-tree is ordered, are not independent as they are both conditions on the left sub-tree. As narrowing the sub-tree to satisfy the first condition only sometimes yields a tree which is ordered, we often end up in situation where there is no solution within the backtrack limit. Similarly to previously, a large sub-tree is less likely to be ordered and therefore smaller trees are favoured. The latter two conditions, on the right sub-tree, also suffer from a similar relationship. Interdependent conditions turn out to be a common problem in the evaluation of properties and are the motivation for the overlapping narrowing language explored in the next chapter.

We test the narrowing tool with an increased backtrack limit of 30 and without a backtrack limit. The results can be found in figure 4.11.

Observation 13. Increasing the backtrack limit increased the success rate of generated values and increased the average size of trees produced at the cost of performance.

At depth 4, increasing the backtrack limit to 30 results in a 100% success rate and increases the average size from around 1.9 to 3.1. Increasing the construction depth still does not result in an increase in average size, with only construction depth 7 resulting in a slightly increased average size of 3.26. Without a construction depth limit, construction depth 4 was the maximum completed in a time limit; however, as it is possible to enumerate this depth in reasonable time we conclude that there is little benefit to random testing over enumeration using narrowing.

4.4.7 Huffman Compression

Finally, we consider an implementation of Huffman compression similar to that implemented by Bird [8]. The implementation supplies an *encode* function which compresses a string based on a table of codes, a *decode* function which reverses the compression, and a *mkHuff* function which generates an optimal table of codes for a given string. We consider a property that asserts that encoding and decoding a string should result in the same string:

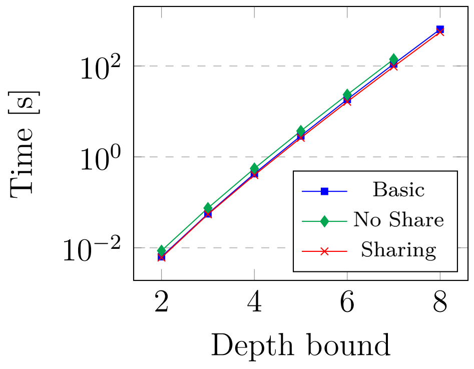
```
data Letter = A | B | C | D | E
propHuff :: [Letter] → Result
propHuff s =  ¬ (null s)  ⟹  decode t (encode t s) ≡ s
  where t = mkHuff s
```

Strat	**Metric**	**4**	**5**	**6**	**7**	**8**
Basic	time	428ms	2.89s	18.4s	111s	641s
	tests	780	3905	2.0E4	9.8E4	4.9E5
	invalid	1	1	1	1	1
Narr	no share	555ms	3.72s	23.4s	141s	-
	sharing	397ms	2.62s	16.3s	96.8s	556s
	tests	780	3905	2.0E4	9.8E4	4.9E5
	invalid	1	1	1	1	1

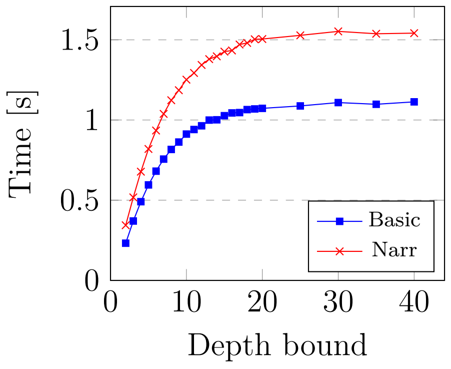
(a) Benchmark results for testing the Huffman property by enumeration to given construction depth

Strat	**Metric**	**5**	**10**	**15**	**20**	∞
Basic	time	595ms	912ms	1.03s	1.07s	1.11s
	success	83.2%	83.6%	83.6%	83.6%	83.4%
	size	3.60	5.06	5.62	5.83	6.03
Narr	time	819ms	1.25s	1.43s	1.50s	1.55s
	success	100.0%	100.0%	100.0%	100.0%	100.0%
	size	3.60	5.02	5.61	5.86	6.00

(b) Benchmark results for testing the Huffman property by random generation with maximum construction depth



(c) Time taken to test the Huffman property by enumeration



(d) Testing Huffman problem on 1000 randomly generated inputs

Figure 4.12: Benchmark results for testing the **Huffman** property

Enumeration

The results for the enumerative evaluation can be found in the graph 4.12c and the table 4.12a.

Observation 14. Narrowing with sharing performs the fastest, followed by the basic strategy, and narrowing without sharing being the worst. The performance differences are small in comparison with our other case studies. At depth 7, the basic strategy is 15% slower than the sharing strategy and narrowing without sharing is a further 27% slower.

This property follows a similar pattern to the reverse property. The *encode* function is hyper-strict, requiring its full argument before it returns any result, and as the precondition is also trivial narrowing offers little benefit. Once again evaluation sharing provides some benefit.

Random

We use a weighting of 1 for the empty constructor, 5 for the cons constructor and equal weightings for the letters. The usual backtrack limit of 3 is used for the narrowing tool. The results can be found in the graph 4.12d and the table 4.12b.

Observation 15. The basic method performs better than the narrowing method, which is 40% slower with no construction depth limit.

Again similar to the reverse property, the basic method performs better as backtracking is not necessary, apart from for the trivial precondition, and therefore the main performance impact comes from the overhead of narrowing. The precondition, reduces the success rate of the basic method to 83.6% but this does not skew the distribution as it only discards the empty list. Both strategies adhere, within a small margin of error, to the expected average size of 6 when no construction depth is imposed.

4.4.8 Summary

We summarise the findings from our case studies.

Shared Evaluation in Enumeration

Shared evaluation offers a consistent performance increase over narrowing without sharing with all the case studies benefiting to various degrees (observations 2, 6, 11, and 14). In two case studies, the union property and reverse, sharing resulted in over a ten times performance benefit at the maximum comparable depth.

Although our results show that it is possible for shared evaluation to give a substantial performance benefit, it is important to note that the performance effect will vary dependent on implementation. As our implementation evaluates in an abstract machine which utilises Haskell's runtime system it is particularly suited to shared evaluation due to Haskell's call-by-need evaluation system. More research is required to see whether these performance benefits can be realised in a compiled system.

Narrowing in Enumeration

Narrowing improved performance by multiple magnitudes in the enumeration of the union and ordered tree problems at their maximum comparable depths (observations 1 and 10). Whereas the performance increase from shared evaluation was roughly constant the performance increase from narrowing grew with the construction depth for these two problems. For these case studies the performance increase was driven primarily by the narrowing of preconditions which determined that ranges of test cases were invalid.

In the reverse and Huffman encoding case studies, narrowing performed slightly worse. This was because both properties are hyper-strict, requiring the whole input in order to produce a result and so narrowing was not beneficial and incurred an overhead from handling substitutions.

Random Testing and Backtrack Limit

Narrowing had a positive impact on random testing for the union and ordered tree case studies, but carried a small performance penalty on the reverse and Huffman case studies. This aligns with the impact of narrowing for enumerative testing. We focus discussion on the union and ordered tree case studies.

In the union case study we found the basic method produced a distribution heavily skewed towards small trivial test cases whereas narrowing was able to produce a reasonable distribution. This situation was further improved by increasing the backtrack limit, which also increased the performance. In the ordered tree case study we found both the basic method and the narrowing method produced heavily skewed results, albeit somewhat less so in the narrowing case. Increasing the backtrack limit improved the distribution somewhat however the results were still clearly skewed and much slower. We identified interdependent conditions as the reason for this slowdown which we address in subsequent chapters.

4.5 Related Work

In this section we discuss related work, focusing on the implementations of property-based testing tools which utilise narrowing.

Reach The work in the chapter is inspired by Naylor and Runciman's work on Reach [38, 39]. Whereas our work is focused on property-based testing, their aim was finding test cases which evaluate a *target* placed a program, for which they considered two different forms of analysis. Initially, they developed a 'forward' analysis that attempts to solve this problem by enumerating test cases using a narrowing evaluation strategy similar to ours. They then went on to develop a 'backward' anaylser which solves the problem by beginning at the target and evaluating the program in the opposite direction.

Lazy SmallCheck Around the same time, Runciman and Naylor also developed Lazy SmallCheck [48], a property-based testing system for Haskell. The system was based on an enumerative search strategy to cover all test cases up to a given depth, and implemented a form of narrowing by using the error-handling functionality in the underlying Haskell compiler. However, this implementation does not share evaluation between test cases, and so is comparable to the wide version of the evaluation strategy that we considered in this chapter albeit using efficient compiled Haskell as opposed to an abstract machine. The tool also introduces parallel conjunction, an operator which helps alleviate interdependent conditions and that we generalise in the next chapters.

First-order data generation Lindblad developed a tool for the generation of first-order data [33]. Similarly to Lazy SmallCheck, it is based on the idea of enumerating test cases up to a given depth. The tool provides a language for expressing properties of data, and exploits narrowing and a parallel version of the conjunction operation to generate data satisfying these properties.

FEAT Duregard et al. develop an algebra for producing enumerations of algebraic data-types in their paper *FEAT: Functional Enumeration of Algebraic Types* [21]. The algebra allows efficient random access to the enumerated values and they evaluate their library by using random testing with backtracking on a property-based testing case study.

EasyCheck Christiansen and Fischer developed the property-based testing tool EasyCheck [13], for the functional-logic language Curry. The tool utilises the language's narrowing evaluation strategy, and supports a novel form of random testing that aims to maximise the amount of evaluation sharing between test cases. However, this form of random testing has the drawback of producing a lot of similar test cases, which often necessitates running the tool multiple times to produce a wide range of test cases.

Lucky Lampropoulos et al. developed Lucky [32], a domain-specific language for generating test cases for property-based testing. The language combines narrowing with the use of a custom constraint solver to avoid instantiating free variables too early. They provided a formalisation along with a prototype interpreter which they demonstrated on a number of case studies.

4.6 Conclusion and Further Work

In this chapter, we developed and evaluated a narrowing-based tool for property-based testing. We gave a formal narrowing semantics for the language which the tool is based on and described how this could be converted into an implementation. In evaluation, we found that both evaluation sharing and narrowing can provide performance benefits, with the former providing a consistent improvement and the latter providing substantial improvements but only on a subset of problems.

There is plenty of scope for this research to be extended. We do so in the following chapter, by adding overlapping patterns to the language which helps address the problem of interdependent conditions that we found in this chapter. A few other potential avenues are discussed below.

In this work we have considered a narrowing implementation as an abstract machine. However to truly establish performance benefits it would be necessary to compare compiled programs. Work on compiled narrowing is still an area of active research [11, 7], and property-based testing is an interesting application for a narrowing compiler.

In our case studies evaluation sharing showed consistent performance benefits and there is potential to expand the scope of this feature. In our tool, evaluation sharing ends as soon as the evaluation of the inputs diverges, however the inputs might have other common parts. For example, evaluation of $\mathit{reverse}\ l'$ within the expression $\mathit{reverse}\ l +\!\!+ \mathit{reverse}\ l'$ will only be shared between evaluations with the same input l, but evaluation of this expression could be shared irrespective of the input l. Expanding evaluation sharing will be a trade off between additional overheads such as memory use and the benefits of storing the evaluation so careful analysis would probably required to determine where it would be effective.

Chapter 5

Overlapping Patterns

In this chapter we develop and formalise the notion of overlapping pattern matching in order to increase the effectiveness of narrowing. Overlapping patterns address the issue of co-dependent conditions, as seen in the previous chapter, and allow properties to be encoded with bespoke size constraints.

We begin the chapter by motivating and introducing overlapping patterns through an example. Then, building on the work of chapter 3, we formalise a narrowing semantics that includes overlapping patterns. We then examine two property-based testing case studies to see how overlapping patterns can be used in practice, before reviewing related work and concluding.

This chapter is based on the paper *Failing Faster: Overlapping Patterns for Property-Based Testing* [27]. In this next chapter we extend our implementation with overlapping patterns and evaluate the performance.

5.1 Motivation and Basic Idea

To motivate the need for overlapping patterns we take a look at a property that epitomizes the issue of co-dependent conditions. The property,

$$propSort\ n\ l = perm\ n\ l \implies sort\ l \equiv [0\,..\,(n-1)]$$

states that if a list l is a permutation of the numbers from 0 to $n-1$, then sorting this list will give the expected result. We take the *perm* condition to be defined as follows:

$$\begin{aligned} &perm &&:: Nat \to [Nat] \to Bool \\ &perm\ n\ l &&= length\ l \equiv n\ \wedge\ all\ ({<}n)\ l\ \wedge\ allDiff\ l \end{aligned}$$

That is, a list l is a permutation of the natural numbers below a given limit n if three conditions are satisfied: the list has the correct length, all the numbers in the list are below the limit, and all the numbers in the list are different. Preconditions defined as a conjunction of constraints in this manner are a common pattern in properties; for example, the ordered tree property we considered also had this pattern (section 4.4). The two last constraints, *all* $({<}n)$ and *allDiff*, are co-dependent as they both constrain the elements of the list. Note that although the first constraint is also on the list, we don't consider this to be co-dependent with the other two as it is a condition on the structure of the list while the others can be satisfied independent of the structure. We use an inductive definition of natural numbers, as narrowing is more effective on algebraic data types than primitive data types [39], with values built from the constructors `Zero` and `Suc`.

First, we consider using a traditional narrowing evaluation to satisfy the *perm* condition and generate test cases for the property. We generate permutations of size 4 by evaluating $\mathit{perm}\ 4\ x$ in which x is free variable. Evaluating to satisfy the first condition, $\mathit{length}\ x \equiv 4$, will refine the free variable by applying the following substitution,

$$x \mapsto [x_0, x_1, x_2, x_3]$$

which represents all possible lists of length 4 as each x_i is a free variable. Continuing by narrowing on the second constraint, $\mathit{all}\ ({<}4)\ l$, we consider the following state, part way through the evaluation:

$$x \mapsto [1, 1, x_2, x_3]$$

The first two elements of the list have each been refined to 1 and according to the current constraint, $\mathit{all}\ ({<}4)\ x$, we have neither failed nor succeeded and as such we continue evaluation by refining x_2 and x_3. However, no input of this form satisfies the final condition $\mathit{allDiff}\ x$. The evaluation will have to backtrack, considering all combinations of x_2 and x_3, without success. Moreover, as we consider generating longer permutations, the number of inputs to consider increases exponentially.

Note that the problem is not resolved by reordering the conditions. For example, suppose that we swapped the order of the last two conditions:

$$\mathit{perm}\ n\ l = \mathit{length}\ l \equiv n\ \wedge\ \mathit{allDiff}\ l\ \wedge\ \mathit{all}\ ({<}n)\ l$$

Then we quickly run into a similar issue. For example, the partial solution, $l = [4, x_1, x_2, x_3]$ does not fail the $\mathit{allDiff}\ l$ constraint, however it will fail the $\mathit{all}\ ({<}4)\ l$ constraint but only after the remaining variables x_1, x_2, x_3 are refined while evaluating $\mathit{allDiff}$. In both cases, the backtracking is caused because a partially refined input fails to satisfy a later condition but this is not evident at the time due to the evaluation order. A natural way to avoid this problem is to evaluate all the conditions simultaneously, rather than sequentially.

To realise this behaviour, we replace traditional pattern matching in our language with *overlapping* pattern matching. Pattern matching in the language is then order-independent, and in each iteration of narrowing all relevant arguments to a pattern match are normalised.

By way of example, consider the logical conjunction operator, which is traditionally defined as follows:

```
False ∧ _ = False
True  ∧ x = x
```

Using this definition, progress can only be made by evaluating the first argument of a conjunction, because each clause of the definition depends on the value of the first argument. Instead, we re-define the operator using overlapping patterns, using a special-purpose pragma to indicate the change in intended semantics:

```
{-# OVERLAP (∧) #-}
False ∧ _     = False
True  ∧ x     = x
_     ∧ False = False
x     ∧ True  = x
```

The definition has two new clauses, given by simply swapping the order of the arguments in the original definition. The idea is that a pattern match can succeed on any of the four clauses, independent of the order that they are stated in. Using this definition, progress can be made by evaluating either argument of the conjunction as the new clauses are no longer dependent on the first argument. For example, we can now reduce $x \wedge$ `False` to `False` for any expression x, which is not the case with the original definition. To take advantage of this additional power, the underlying narrowing mechanism must be modified to evaluate both arguments of the pattern match before it refines variables.

In the *perm* example above, we considered the list $l = [1, 1, x_2, x_3]$ and found that it required a large amount of backtracking. In particular, the constraint *allDiff l* only failed once we had considered all combinations of x_2 and x_3. The new overlapping conjunction operator avoids this problem because it is not biased to the left-argument, allowing *allDiff l* to fail immediately for this example list without the need to further refine the remaining variables.

This additional efficiency is also borne out in practice. For example, using the implementation that we describe in the next chapter, in the time that it takes to generate one hundred valid permutations of length eight for the *perm* constraint defined using the traditional conjunction operator, we can generate one hundred valid permutations of length thirty using the overlapping version. It is also important to note, in this example and in general we only have to change the property to use overlapping patterns and not the program itself to get the desired performance.

However, we have to be careful when using overlapping pattern matching not to introduce non-determinism. Consider the following dangerous function:

```
{-# OVERLAP danger #-}
danger False _     = False
danger True  _     = True
danger _     False = True
danger _     True  = True
```

Using this definition, *danger* `False False` can reduce to either `False` or `True`, depending on whether the first or third clause is used, and is therefore non-deterministic. To counter this, we require overlapping definitions to satisfy confluence laws which guarantee that evaluation is deterministic if all expressions are terminating. The law states that overlapping clauses of a definition should produce the same result and is given formally in section 5.2.1. In principle we could check this condition statically however we currently require the user ensure this law is obeyed.

Other logical operators such as disjunction and implication can be defined using overlapping patterns in a similar manner to conjunction, and will benefit from similar improvements in efficiency. The mechanism can also be used with other data types. A few examples can be found in Figure 5.1, where overlapping definitions of the addition and maximum operators on natural numbers, and for the applicative operator [37] on the *Maybe* type are given. As illustrated by the latter example, overlapping definitions are not restricted to commutative operators.

5.2 Generalizing and Formalizing

In this section we define the syntax and semantics of our language of overlapping patterns. We consider the normalising subset of the language and

```
{-# OVERLAP (+) #-}
(+) :: Nat → Nat → Nat
Zero   + y      = y
Suc x  + y      = Suc (x + y)
x      + Zero   = x
x      + Suc y  = Suc (x + y)

{-# OVERLAP max #-}
max :: Nat → Nat → Nat
max Zero    y       = y
max (Suc x) y       = Suc (max x (pred y))
max x       Zero    = x
max x       (Suc y) = Suc (max (pred x) y)
  where
    pred Zero    = Zero
    pred (Suc x) = x

{-# OVERLAP (<*>) #-}
(<*>) :: Maybe (a → b) → Maybe a → Maybe b
Nothing <*> _       = Nothing
Just f  <*> a       = fmap f a
_       <*> Nothing = Nothing
f       <*> Just a  = fmap ( $ a) f
```

Figure 5.1: Example overlapping function definitions.

show that a confluence restriction on definitions is sufficient to guarantee that the language is deterministic. We then extend the semantics with narrowing, and show that the new semantics is sound and complete with respect to the original version of the semantics.

We use a small functional programming core language with definitions, constructors, variables, lambda expressions and application. To simplify the theory, the language only allows one form of pattern matching: at the top-level of a function definition, interpreted in an overlapping, order-independent manner. However, other forms of pattern matching, such as **case** expressions and non-overlapping patterns, can readily be rewritten into this form.

The syntax of the language is formally defined as follows:

$$\begin{aligned}
\mathit{Defn}_X &::= \overline{\mathit{Var}\ \mathit{Patts} = \mathit{Expr}_X} \\
\mathit{Expr}_X &::= \mathit{Con} \mid \mathit{Var} \mid X \mid \mathit{Expr}_X\ \mathit{Expr}_X \mid \lambda \mathit{Var}\ .\ \mathit{Expr}_X \\
\mathit{Patts} &::= \overline{\mathit{Var}}\ (\mathit{Con}\ \overline{\mathit{Var}})\ \overline{\mathit{Var}}
\end{aligned}$$

That is, a definition is made up of a list of clauses, with a pattern for each argument on the left and an expression on the right. We discuss the exact form of definitions after covering the rest of the language. Expressions and definitions are parameterised by a set of free variables X, which is only used in the narrowing semantics. The language has a standard set of typing rules, which we omit for brevity. Each type has a set of constructors and the patterns used in definitions should form a covering of these constructors. We assume, each variable only appears once in a pattern, and the only free variables in an expression should be those that appear in the set X.

We often use f for definitions, e for expressions, c for constructors, u and v for closed variables, x and y for free variables, and p and q for patterns. We take $f\,\overline{e}$ to mean the definition f completely applied to a sequence of arguments $\overline{e}$, i.e. the length of the sequence of arguments is the same as the arity of the definition.

Overlapping definitions

The definitions in the language are functions with precisely one overlapping pattern match. This is represented in the left hand side of a clause, in which each argument has a pattern and precisely one of these patterns is an un-nested constructor pattern and the rest are variables. We explain the reasoning behind this form of pattern matching via three questions:

Why does pattern matching occur within a definition? Or conversely, why not define pattern matching using a case expression as we did in chapter 3? The answer is to allow for normalising, recursive programs in the context of a full reduction semantics. Using case expressions and recursion will often result in there being infinite reduction sequences as reduction can be performed within the alternatives. By including pattern matching in definitions we can force the evaluation of an argument before a definition is applied and its right-hand side comes into scope.

Why does each clause only have a single pattern match? Using a single pattern match simplifies aspects of the semantics and the implementation in the next chapter. Just as standard nested pattern matching can be deconstructed into individual case expressions, it is possible to deconstruct nested overlapping patterns into a sequence of overlapping definitions with a single pattern match. It should be noted that unlike nested traditional pattern matching, which often results in simpler definitions, we have not found any useful examples of nested overlapping patterns as most functions benefiting from an overlapping definition are basic operations.

Why do definitions overlap on every argument? As we can form standard pattern matching by definitions with one argument and other functions by using lambda expressions, it is enough to consider definitions to be overlapping on all arguments.

5.2.1 Semantics

We give a standard small-step operational semantics to the language in a contextual style. We start by defining a full reduction semantics, in which any reducible term in an expression can be reduced. This allows us to define notions of equivalence and establish confluence properties. We then define a call-by-name evaluation strategy by limiting the form of contexts that can be used, which is then used to define the narrowing strategy.

First we define a *local* semantics $\rightarrow_R \subseteq \mathit{Expr}_X \times \mathit{Expr}_X$ that performs basic reduction steps on expressions, which is then lifted into an evaluation context. For the semantics, we require substitutions binding standard variables to expressions. To avoid confusion, in this chapter we refer to narrowing substitutions exclusively as *refinements* and reserve the term substitution for mappings from standard variables, which we typically denote using s and t. We write $e[s]$ for the application of a substitution to each variable in an expression, $\emptyset$ for the identity substitution that maps each variable to itself, and $s;t$ for the composition of substitutions.

The first local rule is the standard β-rule:

$$\frac{}{(\lambda v.e)\ e' \rightarrow_R e[v \mapsto e']}\ \textsc{sub}$$

The second rule states that we can reduce a definition if the pattern of any of its clauses matches the arguments, where $f\ \overline{p} = e\ \in\ \mathrm{defn}(f)$ means that the clause $f\ \overline{p} = e$ is part of the definition for the function f. In contrast to traditional pattern matching, the clauses of a definition may be applied in any order.

$$\frac{f\ \overline{p} = e'\ \in\ \mathrm{defn}(f) \qquad \texttt{Matches}(\overline{p},\ \overline{e},\ s)}{f\ \overline{e} \rightarrow_R e'[s]}\ \textsc{match}$$

The predicate `Matches` used above captures the idea of a successful match of expressions against patterns with s the resulting substitution. It is defined as follows,

$$\frac{}{\texttt{Match}(v,\ e,\ \{v \mapsto e\})} \qquad \frac{}{\texttt{Match}(\mathbf{c}\ \overline{v},\ \mathbf{c}\ \overline{e},\ \{\overline{v} \mapsto \overline{e}\})}$$

$$\frac{}{\texttt{Matches}(\epsilon,\ \epsilon,\ \emptyset)} \qquad \frac{\texttt{Match}(p,\ e,\ s) \qquad \texttt{Matches}(\overline{p},\ \overline{e},\ t)}{\texttt{Matches}(p\ \overline{p},\ e\ \overline{e},\ s;t)}$$

in which `Match` gives the definition for a single pattern, `Matches` for a list of patterns and where $\overline{v} \mapsto \overline{e}$ is the binding of each of the variables in $\overline{v}$ to its corresponding expression in $\overline{e}$.

In turn, a *context* is an expression with a singular hole in any location, as defined by the following set of inference rules:

$$\frac{}{[]\ \text{context}}\ \textsc{hole} \qquad \frac{\mathbf{C}\ \text{context}}{(\lambda v.\, \mathbf{C})\ \text{context}}\ \textsc{lam}$$

$$\frac{\mathbf{C}\ \text{context}}{(\mathbf{C}\ \ e)\ \text{context}}\ \textsc{app-l} \qquad \frac{\mathbf{C}\ \text{context}}{(e\ \ \mathbf{C})\ \text{context}}\ \textsc{app-r}$$

We use inference rules above rather than a grammar because the extra generality of this notation is used when contexts are revised later on. As usual, we write $\mathbf{C}[e]$ for the result of replacing the hole in $\mathbf{C}$ with the expression e. Note that the semantics will reduce under lambda (the `LAM` rule), this is to establish an equivalence relation between terms and allow a fully general confluence law but does occur in actual evaluation as defined in section 5.2.2.

Using the local semantics and the notion of contexts we can now define the full reduction semantics for expressions in our language.

Definition 5.1. The *full reduction* semantics, $\rightarrow \subseteq\ \mathit{Expr}_X \times \mathit{Expr}_X$, is defined below:

$$\frac{e \rightarrow_R e' \qquad \mathbf{C}\ \text{context}}{\mathbf{C}[e] \rightarrow \mathbf{C}[e']}$$

Definition 5.2. $\rightarrow^*$ is the reflexive/transitive closure of $\rightarrow$.

A reduction to normal form, which we use in this chapter to establish the relation to the narrowing semantics, is given by:

Definition 5.3. A *normalising reduction sequence* is a reduction sequence to an expression which can no longer be reduced.

$$e \downarrow e' \iff e \rightarrow^* e' \;\wedge\; e' \not\rightarrow$$

To ensure that our language is deterministic and avoid examples such as *danger* `False False` from section 5.1 that have more than one normal form, we require all definitions to satisfy a confluence property. To formalise this property we first define the notions of definitional equivalence and unification.

Definition 5.4. Two expressions are *definitionally equivalent*, written $e \equiv e'$, if there are reduction sequences from e and e' to the same expression:

$$e \equiv e' \iff \exists e''.\ e \rightarrow^* e'' \wedge e' \rightarrow^* e''$$

Informally, two patterns are unifiable if there is an expression which matches both the patterns. We can formalise this by giving a pair of substitutions which when applied to each pattern yield the common expression.

Definition 5.5. The *most general unifier* is defined by the inference rules below. `Unify`$(p,\ q,\ s_1,\ s_2)$ denotes the unification of patterns p and q by substitutions s_1 and s_2, and similarly for a list of patterns with `Unifies`. Note we are using the assumption that every variable appears only once in each pattern here.

$$\frac{}{\texttt{Unify}(v,\ p,\ \{v \mapsto p\},\ \emptyset)} \qquad \frac{}{\texttt{Unify}(\mathbf{c}\ \overline{v},\ \mathbf{c}\ \overline{v'},\ \{\overline{v} \mapsto \overline{v'}\},\ \emptyset)}$$

$$\frac{}{\texttt{Unify}(\mathbf{c}\ \overline{v},\ v,\ \emptyset,\ \{v \mapsto \mathbf{c}\ \overline{v}\})}$$

$$\frac{}{\texttt{Unifies}(\epsilon,\ \epsilon,\ \emptyset,\ \emptyset)} \qquad \frac{\texttt{Unify}(p,\ q,\ s_1,\ s_2) \qquad \texttt{Unifies}(\overline{p},\ \overline{q},\ s_1',\ s_2')}{\texttt{Unifies}(p\ \overline{p},\ q\ \overline{q},\ s_1; s_1',\ s_2; s_2')}$$

This definition has the expected behaviour, that is:

$$\begin{aligned}
&\texttt{Unify}(p,\ q,\ s_1,\ s_2) \\
&\qquad \implies p[s_1] = q[s_2] && \text{(unifier)} \\
&\qquad \wedge\ \forall t_1 t_2.\ p[t_1] = q[t_2].\ \exists r.\ s_1; r = t_1 \wedge s_2; r = t_2 && \text{(most general)}
\end{aligned}$$

If the patterns of two clauses of a definition are unifiable then, given a suitable context, it is possible for two different MATCH reductions in our semantics to be applied. In order to maintain determinism for such clauses a confluence restriction is required. The confluence restriction states that the right-hand sides of each pair of clauses must be definitionally equivalent under their unifying substitution if one exists. In principle for the terminating subset of the language we could check whether a definition satisfies the confluence property automatically by generating the unifiers pairwise and normalising each clause.

Definition 5.6. A definition satisfies the *confluence restriction* if for any pair of clauses, $f\ \overline{p} = e$ and $f\ \overline{q} = e'$, we have the following property:

$$\texttt{Unifies}(\overline{p},\ \overline{q},\ s_1,\ s_2) \implies e[s_1] \equiv e'[s_2]$$

Theorem 5.1. *The relation $\rightarrow^*$ is confluent if all the definitions satisfy the confluence restriction, i.e. for any reductions $e \rightarrow^* e_1$, $e \rightarrow^* e_2$, there exists an expression e' such that $e_1 \rightarrow^* e'$ and $e_2 \rightarrow^* e'$.*

Proof. By parallel reduction [45, 40] with special consideration for overlapping patterns. □

It follows in the standard way from the above confluence property that any expression that only has finite reduction sequences has precisely one normal form. Hence, our semantics is deterministic for such expressions.

5.2.2 Evaluation Order

Our current semantics allows reduction rules to be applied in any context and in any order. This is convenient for defining the behavioural properties of the semantics, but in order to define the narrowing semantics and give an efficient implementation, we need to restrict where reduction rules are applied. To do this we define a subset of contexts called *evaluation contexts*.

The notion of evaluation context is call-by-name, and hence only evaluates the left-hand side of an application. When the expression is a definition applied to some arguments then the arguments are reduced until one is a constructor and therefore a pattern match is possible. We do not assert in what order the arguments should be reduced but in our implementation the reduction occurs from left to right. The rules are defined formally as follows,

$$\frac{}{\bullet \ \text{evalcxt}}\ \textsc{hole} \qquad \frac{\mathbf{C}\ \ \text{evalcxt}}{(\mathbf{C}\ \ e)\ \ \text{evalcxt}}\ \textsc{app-l}$$

$$\frac{\overline{\mathbf{C}} = \overline{\mathrm{e}}\,\mathbf{C}\,\overline{e'} \qquad \mathbf{C}\ \text{evalcxt} \qquad \forall e \in \overline{\mathrm{e}}\,\overline{e'}.\ \neg whnf(e)}{(f\ \overline{\mathbf{C}})\ \ \text{evalcxt}}\ \textsc{args}$$

in which $\overline{\mathbf{C}}$ is a list of expressions with one context. The predicate $\neg whnf$ asserts that an expression is not in weak head normal form. In this case that means that none of the expressions is a constructors and therefore it is not possible to reduce the definition using a pattern match. Note, we can always make progress as every argument in a definition must perform a pattern match (arguments which do not perform a pattern match are introduced using a lambda).

Definition 5.7. The *evaluation reduction* semantics, $\rightarrow_E$, is defined by:

$$\frac{\mathbf{C}\ \ \text{evalcxt} \qquad e \rightarrow_R e'}{\mathbf{C}[e] \rightarrow_E \mathbf{C}[e']}$$

5.2.3 Narrowing

The narrowing semantics follows a similar format to our previous formalisation (section 3.4). We reiterate the definitions both for convenience and to update them to our current language.

Refinement

We define the partial values of the language, as follows:

$$Val_X = Con\ \overline{Val_X} \mid X$$

A refinement of type $X \mapsto Y$ is a function from the free variable set X to Val_Y. Composition of refinements, which we denote by $\ggg$, is defined in the standard way. The null refinement, $return \in X \mapsto X$, corresponds to the trivial substitution that maps each free variable to itself. The refinements adhere to lemma 1:

$$e[\sigma][\sigma'] \equiv e[\sigma \ggg \sigma']$$

That is, the sequential application of refinements is equivalent to the application of composed refinements.

We define inputs as refinements mapping variables to values i.e. of type $X \mapsto Val_\emptyset$. We denote refinements with σ and inputs with τ.

Narrowing Set

The narrowing set is defined as previously as the complete, advancing and minimal set of refinements which replace a single variable. The narrowing set for free variable x of type t is given as follows,

$$Narr_X(x^t) = \{x \,/\, \mathbf{c}\ \overline{y} \mid \mathbf{c} \in cons(t)\} \qquad \overline{y} \notin X$$

where $cons(t)$ are the constructors of type t. Formal definitions of complete and advancing are given in section 3.5, lemmas 4 and 5. Completeness ensures that every constructor is represented in the narrowing set, and advancing that every refinement in the set is not trivial which is important for the well-foundedness of the completeness theorem.

Semantics

An expression is suspended on a free variable if it is an evaluation context and there are no other possible evaluation reductions to make. This can be defined as follows:

$$\frac{\mathbf{C}[x] \not\to_E \qquad \mathbf{C}\ \text{evalcxt}}{\mathbf{C}[x] \multimap x}$$

Note an expression might be suspended on multiple different free variables when an overlapping definition is being evaluated; refining any of them will allow progress to be made. This definition differs from that in section 3.4 by the requirement that there are no possible reductions. In the previous semantics this requirement was implicit as there was only one possible reduction. Here it ensures that the expression is only suspended if no progress can be made.

We can now define narrowing reduction as follows:

Definition 5.8. The *narrowing* reduction, $\rightsquigarrow \subseteq \mathit{Expr}_X \times (\mathit{Expr}_Y \times (X \mapsto Y))$, is defined by the following two inference rules:

$$\frac{e \to_E e'}{e \rightsquigarrow \langle e',\ \mathit{return}\rangle} \qquad\qquad \frac{e \multimap x \qquad \sigma \in \mathit{Narr}_X(x)}{e \rightsquigarrow \langle e[\sigma],\ \sigma\rangle}$$

The first rule states that any evaluation reduction is also a narrowing reduction, with no refinement necessary. The second states that if evaluation is suspended then a narrowing step should be taken.

Definition 5.9. A narrowing evaluation to normal form is given by:

$$\frac{e[\tau] \downarrow e' \qquad e \not\twoheadrightarrow_E \qquad e \not\multimap}{e \lightning \langle e',\ \tau\rangle} \qquad\qquad \frac{e \rightsquigarrow \langle e,\ \sigma\rangle \qquad e' \lightning \langle e'',\ \sigma'\rangle}{e \lightning \langle e'',\ \sigma \mathbin{>\!\!=\!\!\!>} \sigma'\rangle}$$

The second rule, composing reductions, is standard. The first rule, which corresponds to a base rule, refines the remaining free variables by applying an arbitrary input and then normalises the resulting expression. The application of an input simplifies the formulation of the completeness result. The two conditions for the base rule, $e \not\twoheadrightarrow_E$ and $e \not\multimap$, are equivalent to the notion of the expression being in weak head normal form. Therefore, this definition corresponds to a narrowing evaluation to weak head normal form followed by a normalisation in the normal language. Alternatively, we could have opted to apply the narrowing strategy recursively once reaching weak head normal form, but we decided not to do this as the current formulation is simpler and sufficient for our purposes.

Soundness

Before providing the definition and proof of soundness we remind the reader of lemma 2, which is also satisfied by the overlapping semantics. The lemma states that for any refinement σ we have:

$$e \to e' \implies e[\sigma] \to e'[\sigma]$$

Theorem 5.2. *(narrowing is sound.) For every normalising narrowing* *reduction there exists a corresponding reduction in the original semantics:*

$$e \mathrel{\rotatebox[origin=c]{-90}{$\rightsquigarrow$}} \langle e',\ \tau\rangle \implies e[\tau] \downarrow e'$$

Proof. The proof proceeds by rule induction on the definition for the narrowing relation $\mathrel{\rotatebox[origin=c]{-90}{$\rightsquigarrow$}}$, for which there are three cases to consider.

Case 1 In the base case when the narrowing reduction is complete and the expression is normalised in the original language,

$$\frac{e[\tau] \downarrow e' \qquad e \not\rightarrow \qquad e \not\multimap}{e \mathrel{\rotatebox[origin=c]{-90}{$\rightsquigarrow$}} \langle e',\ \tau\rangle}$$

then the result appears as the first assumption of the rule, $e[\tau] \downarrow e'$.

Case 2 There are two inductive cases to consider, depending on the nature of the first reduction in the narrowing reduction. We first consider the case when the reduction is a refinement, constructed as follows:

$$\frac{\dfrac{e \multimap x \qquad \sigma \in \mathit{Narr}_X(x)}{e \rightsquigarrow \langle e[\sigma],\ \sigma\rangle} \qquad e[\sigma] \mathrel{\rotatebox[origin=c]{-90}{$\rightsquigarrow$}} \langle e',\ \tau\rangle}{e \mathrel{\rotatebox[origin=c]{-90}{$\rightsquigarrow$}} \langle e',\ \sigma \mathrel{>\!\!=\!\!>} \tau\rangle}$$

The proof follows from applying the inductive hypothesis on the final assumption, (IH) $e[\sigma] \mathrel{\rotatebox[origin=c]{-90}{$\rightsquigarrow$}} \langle e',\ \tau\rangle \implies e[\sigma][\tau] \downarrow e'$, and then applying lemma 1:

$$\cfrac{\cfrac{e[\sigma] \mathrel{\rotatebox[origin=c]{-90}{$\rightsquigarrow$}} \langle e',\ \tau\rangle}{e[\sigma][\tau] \downarrow e'}\ \textsc{ih}}{e[\sigma \mathrel{>\!\!=\!\!>} \tau] \downarrow e'}\ \textsc{lemma}\ 1$$

Case 3 We now consider the case when the first step is a reduction from the evaluation reduction semantics, constructed as follows:

$$\frac{\dfrac{e \rightarrow_E e'}{e \rightsquigarrow \langle e',\ \mathit{return} \rangle} \qquad e' \lightning \langle e'',\ \tau \rangle}{e \lightning \langle e'',\ \mathit{return} \mathbin{>\!\!=\!\!>} \tau \rangle}$$

In this case we use $\rightarrow_E \subseteq \rightarrow$ and lemma 2, to lift the reduction into the original semantics and sequence it with the evaluation formed from the inductive hypothesis:

$$\cfrac{\cfrac{\text{LEMMA 2}\ \dfrac{e \rightarrow_E e'}{e[\tau] \rightarrow e'[\tau]} \qquad \dfrac{e' \lightning \langle e'',\ \tau \rangle}{e'[\tau] \downarrow e''}\ \text{IH}}{e[\tau] \downarrow e''}}{e[\mathit{return} \mathbin{>\!\!=\!\!>} \tau] \downarrow e''}\ \text{ID}$$

□

Completeness

To ensure that the corresponding completeness theorem is valid, we restrict our attention to expressions that strongly normalise under any refinement. We begin by defining the set of expressions that normalise.

Definition 5.10. The set of normalising expressions, `Norm`, can be defined by the inductive rule:

$$\frac{\forall e'.\ e \rightarrow e' \implies e' \in \texttt{Norm}}{e \in \texttt{Norm}}$$

Definition 5.11. An expression *always normalises*, $e \in \mathtt{Norm}^{\square}$, if it normalises under any refinement:

$$e \in \mathtt{Norm}^{\square} \iff \forall \sigma.\ e[\sigma] \in \mathtt{Norm}$$

It should be noted that $e \in \mathtt{Norm}^{\square}$ does not imply that there are no infinite narrowing reductions. For example, if we consider the *even* function,

$$\begin{aligned}&\mathit{even}\ \mathtt{Zero} = \mathtt{True}\\ &\mathit{even}\ (\mathtt{Suc}\ \ n) = \neg\ (\mathit{even}\ n)\end{aligned}$$

then the expression *even* x has an infinite narrowing sequence in which we repeatedly apply the `Suc` refinement. We state one lemma concerning $\mathtt{Norm}^{\square}$ before proceeding with the completeness theorem.

Lemma 8. $\mathtt{Norm}^{\square}$ *is closed under* $\rightarrow$

$$e \in \mathtt{Norm}^{\square} \wedge e \rightarrow e' \implies e' \in \mathtt{Norm}^{\square}$$

Proof. For each σ, use lemma 2 to lift the reduction to $e[\sigma] \rightarrow e'[\sigma]$, then $e'[\sigma] \in \mathtt{Norm}$ follows as $e[\sigma] \in \mathtt{Norm}$. ☐

Theorem 5.3. *(narrowing is complete.) For every reduction of a normalising expression there is a corresponding narrowing reduction:*

$$e \in \mathtt{Norm}^{\square}\ \wedge\ e[\tau] \downarrow e' \implies\ e \rightsquigarrow \langle e',\ \tau\rangle$$

Proof. In order to complete the proof, we need to prove a slightly generalised statement:

$$e_1 \in \mathtt{Norm}^{\square}\ \wedge e_0[\tau] \downarrow e'\ \wedge\ e_0[\tau] \rightarrow^{*} e_1[\tau] \implies e_1 \rightsquigarrow \langle e',\ \tau\rangle$$

Generalising the statement in this way weakens the assumptions and allows us to use the statement inductively. We consider three possible inductive cases and prove formally that the inductive hypothesis used in each case is well-founded. The basis of this well-foundedness is a strict lexicographical order formed from either a decrease in the size of the input, τ, or a step in a normalising evaluation.

Case 1 When there are no more reductions, $e_1 \not\rightarrow$, and e_1 is not suspended on any free variable, we have $e_1 \not\multimap$, i.e. e_1 is in weak head normal form. Then we can form the solution using our assumptions and confluence:

$$\dfrac{\text{CONFLUENCE}\ \dfrac{e_0[\tau] \downarrow e' \qquad e_0[\tau] \rightarrow^* e_1[\tau]}{e_1[\tau] \downarrow e'} \qquad e_1 \not\rightarrow \qquad e_1 \not\multimap}{e_1 \leadsto \langle e',\ \tau \rangle}$$

Case 2 If there exists $e_1 \rightarrow_E e_1'$ then we can perform a reduction. We can form the assumptions to use the statement inductively,

$$\text{IH}\ \dfrac{\dfrac{e_1 \in \mathtt{Norm}^{[]} \qquad e_1 \rightarrow_E e_1'}{e_1' \in \mathtt{Norm}^{[]}} \qquad e_0[\tau] \downarrow e' \qquad \dfrac{e_0[\tau] \rightarrow^* e_1[\tau] \qquad \dfrac{e_1 \rightarrow_E e_1'}{e_1[\tau] \rightarrow e_1'[\tau]}}{e_0[\tau] \rightarrow^* e_1'[\tau]}}{e_1' \leadsto \langle e',\ \tau \rangle}$$

where the left branch utilises lemma 8 and the right branch lemma 2. The well-foundedness of this application of the inductive hypothesis comes from progress towards normalising e_1. Note that the input τ remains the same. The proof follows by lifting the reduction into the narrowing language and then sequencing with the inductive hypothesis:

$$\dfrac{\dfrac{e_1 \rightarrow_E e_1'}{e_1 \rightsquigarrow \langle e_1',\ \mathit{return}\rangle} \qquad \dfrac{\vdots}{e_1' \downzigzagarrow \langle e',\ \tau\rangle}\,\text{IH}}{e_1 \downzigzagarrow \langle e',\ \tau\rangle}$$

Case 3 Finally, if the expression is suspended, $e_1 \multimap x$, then we need to take a narrowing step. By the completeness of the narrowing set, lemma 4, we have $\sigma \in \mathit{Narr}_x$ such that $\sigma \mathrel{>\!=\!\!>} \tau' = \tau$. Then we can use the statement inductively,

$$\dfrac{\dfrac{e_1 \in \mathtt{Norm}^{\square}}{e_1[\sigma] \in \mathtt{Norm}^{\square}} \qquad \dfrac{e_0[\tau] \rightarrow e'}{(e_0[\sigma])[\tau'] \rightarrow e'} \qquad \dfrac{e_0[\tau] \rightarrow^* e_1[\tau]}{(e_0[\sigma])[\tau'] \rightarrow^* (e_1[\sigma])[\tau']}}{e_1[\sigma] \downzigzagarrow \langle e',\ \tau'\rangle}\,\text{IH}$$

where the left branch makes use of the definition of $\mathtt{Norm}^{\square}$, and the remaining two branches both use the simple property of composed substitutions (lemma 1). The well-foundedness is given by a decreasing input size, $\tau' < \tau$, which is guarenteed by the advancing property of the narrowing set (lemma 5). The proof then follows by appending a narrowing step to our inductive hypothesis:

$$\dfrac{\dfrac{e_1 \multimap x \qquad \sigma \in \mathit{Narr}_X(x)}{e_1 \rightsquigarrow \langle e_1[\sigma],\ \sigma\rangle} \qquad \dfrac{\vdots}{e_1[\sigma] \downzigzagarrow \langle e',\ \tau'\rangle}\,\text{IH}}{e_1 \downzigzagarrow \langle e',\ \sigma \mathrel{>\!=\!\!>} \tau'\rangle}$$

Well Foundedness The proof is well-founded based on the combination of two well-founded partial orders, the ordering of inputs on strict suffixes and the ordering of normalising expressions under $\rightarrow$. The suffix order and its well-founded nature were established in section 3.5.2 and required for a refinement, $X \mapsto Y$, the domain and range to be finite and each of the variables in Y to appear at least once in the result. These restrictions have no impact for our use. We combine the orders lexicographically, with the suffix order taking priority, as follows:

$$\frac{\tau' < \tau}{(\tau', e') < (\tau, e)} \qquad\qquad \frac{e \rightarrow e' \qquad e \in \texttt{Norm}}{(\tau, e') < (\tau, e)}$$

It is well known that lexicographical product of two well-founded orders is also well-founded. In case 2 of our proof the inductive hypothesis is valid under the second rule, that is the input refinement stays the same and the normalising expression is reduced. In case 3 the inductive hypothesis is valid under the first rule; the input refinement is a strict suffix of the original refinement because the refinements in our narrowing set are advancing. □

In contrast to the soundness proof which proceeds similarly to that of the previous formulation, this proof of completeness differs significantly. In particular, the proof of completeness in the previous formulation relied on an almost direct correspondence between the narrowing semantics and the original semantics (lemma 3), whereas here we utilise confluence and normalisation to establish the relationship.

A weaker confluence?

It could be argued that the definition of confluence is too strong for the purpose of narrowing because it disallows some definitions which behave identically under narrowing. A pertinent example is multiplication. We might expect the following definition to be valid:

$$\begin{array}{lcll}
\texttt{Zero} & * & v & = \texttt{Zero} \\
\texttt{Suc}\ u & * & v & = u * v + v \\
u & * & \texttt{Zero} & = \texttt{Zero} \\
u & * & \texttt{Suc}\ v & = u + u * v
\end{array}$$

However, if we consider $(\texttt{Suc}\ \ u) * (\texttt{Suc}\ \ v)$ then we have two possible reduction sequences:

$$\begin{array}{lclcl}
\texttt{Suc}\ \ u * \texttt{Suc}\ \ v & \to & (u * (\texttt{Suc}\ \ v)) + \texttt{Suc}\ \ v & \to & \texttt{Suc}\ ((u + u * v) + v) \\
\texttt{Suc}\ \ u * \texttt{Suc}\ \ v & \to & \texttt{Suc}\ \ u + (\texttt{Suc}\ \ u * v) & \to & \texttt{Suc}\ (u + (u * v + v))
\end{array}$$

The two results, which are in normal form, are not definitionally equal as the + operators are associated differently. However, from a perspective of narrowing we might consider these two expressions to behave the same. That is, under any refinement the observable result of $x * y$ is the same, in which we only consider constructors to be observable.

While it seems possible to formalise the above notion by basing confluence on a equality based on observation as opposed to definitional equality, doing so would add to the complexity of the formalisation. As all the functions we use obey the definitional notion of confluence we have not explored this avenue further at the present time.

We could also consider extending the domain of valid functions by directly extending the reduction rules of $\to$ and the notion of definitional equality. For example, we could imagine adding the reduction rule,

$$(u + v) + w \ \to\ u + (v + w)$$

which would equate the two alternative multiplication reductions and has the advantage of being directly applicable to our theory. However, without a general method for adding such reduction rules, we would have to check confluence is obeyed every time a rule is added.

5.3 Property-based testing examples

In this section we consider two examples of using overlapping patterns to aid property-based testing. We focus on the generation of data for testing and demonstrate two programming techniques which we use frequently in the case studies in the next chapter. The first example demonstrates how overlapping patterns can be used to encode bespoke size constraints in the generation of ordered trees. The second demonstrates how additional free variables can aid in writing efficient narrowing through the example of typed expressions in a simple language.

Our aim in each case is to find a definition of the precondition that eliminates the need for backtracking (apart from rebinding of a single constructor). We say that such a constraint *fails fast*. Formally, a constraint fails fast if when testing any partial value against the constraint it either directly fails or there is a refinement of the value that succeeds. The needed narrowing generator formed by a constraint which fails fast is generally efficient. In this section we focus on the qualitative experience of programming with overlapping patterns. Performance results and detailed description of the implementation can be found in the next chapter.

5.3.1 Ordered Trees

We recall our definition of ordered trees as binary trees with natural numbers stored in ascending order within the nodes:

```
data Tree = Leaf | Node Tree Nat Tree
```

$$\begin{aligned}
&\mathit{ord} :: \mathit{Tree} \to \mathit{Bool}\\
&\mathit{ord}\ \mathit{Leaf} = \texttt{True}\\
&\mathit{ord}\ (\mathit{Node}\ \mathit{t1}\ \ a\ \ \mathit{t2}) \ = \ \mathit{allT}\ (\leqslant a)\ \mathit{t1}\ \wedge\ \mathit{ord}\ \mathit{t1}\\
&\qquad\qquad\qquad\qquad\ \ \wedge\ \mathit{allT}\ (\geqslant a)\ \mathit{t2}\ \wedge\ \mathit{ord}\ \mathit{t2}
\end{aligned}$$

Note that this definition now uses overlapping conjunction and in doing so addresses the issue of co-dependent conditions we found in the previous chapter. If evaluating the first constraint, $\mathit{allT}\ (\leqslant a)\ \mathit{t1}$, causes the second constraint, the recursive call to *ord*, to fail then overlapping conjunction will ensure this failure is realised immediately.

We consider again the order preserving characteristic of a delete function that removes a given element from a tree:

$$\begin{aligned}
&\mathit{propDelete} \qquad :: \mathit{Nat} \to \mathit{Tree} \to \mathit{Bool}\\
&\mathit{propDelete}\ a\ t = \mathit{ord}\ t\ \implies\ \mathit{ord}\ (\mathit{del}\ a\ t)
\end{aligned}$$

Unfortunately, if we test this property in its current form it will often fail to halt, because randomly generated values of recursively defined types such as trees are often infinite. Before we resolved this problem by setting a global limit on the depth of constructors (section 4.3.2), however this limit can result in backtracking as to satisfy *ord* a natural number might be required that breaks the limit. For random testing it is sufficient to limit the number or depth of nodes without restricting the elements, and doing so avoids backtracking. We can use overlapping patterns to achieve this.

First of all, we define a *suchThat* function, which can be used to add size constraints to a property:

```
{-# OVERLAP suchThat #-}
suchThat :: Result → Bool → Result
suchThat Invalid _       = Invalid
suchThat Success x       = if x then Success else Invalid
suchThat Failure x       = if x then Failure else Invalid
suchThat _       False   = Invalid
suchThat x       True    = x
```

We can then use this function to update our property,

```
propDelete n a t =
  (ordered t ⟹ ordered (delete a t))
  `suchThat` (depthTree t ⩽ n)
```

in which the *depthTree* constraint is given by:

```
depthTree :: Tree → Nat
depthTree Leaf = Zero
depthTree (Node t1 _ t2) = Suc (max (depthTree t1) (depthTree t2))
```

The use of overlapping patterns is crucial in two ways. Firstly, in the suchThat constraint the overlapping patterns ensure that the constraint is always evaluated. The left-biased nature of our implementation means this *suchThat* constraint will not impact the evaluation of the property in any other way. Secondly, the definition of *depthTree* relies on the overlapping version of the maximum function. This is important as a traditional maximum function only evaluates its right side once it has completed evaluation of its left side and so during narrowing the right branch of the tree could become arbitrarily large without triggering the size limit.

To ensure termination in enumerative testing, we need to limit the size of elements. We also opt to limit the number of the nodes as opposed to the depth of nodes as Duregaard showed in his thesis [1] that limiting the size of terms in this way is generally more effective for enumerative testing. The property then becomes

$$\begin{aligned}&\mathit{propDeleteEnum}\ n\ a\ t\\&\quad = (\mathit{ordered}\ t \implies \mathit{ordered}\ (\mathit{delete}\ a\ t))\\&\quad `\mathit{suchThat}`\ (\mathit{sizeTree}\ t \leqslant n\ \wedge\ \mathit{allT}\ (\leqslant n)\ t)\end{aligned}$$

in which the *sizeTree* is given by:

$$\begin{aligned}&\mathit{sizeTree} :: \mathit{Tree} \to \mathit{Nat}\\&\mathit{sizeTree}\ \mathit{Leaf} = \texttt{Zero}\\&\mathit{sizeTree}\ (\mathit{Node}\ \mathit{t1}\ \_\ \mathit{t1}) = \texttt{Suc}\ (\mathit{sizeTree}\ \mathit{t1} + \mathit{sizeTree}\ \mathit{t2})\end{aligned}$$

Again, this definition uses an overlapping function + in order to ensure that the size limit is always adhered to. We know of no way of defining such a size limit without the use of overlapping addition and therefore don't believe this size limit can be defined in already existing tools which only implement parallel conjunction. In contrast the depth limit condition can be defined only using of overlapping conjunction as so:

$$\begin{aligned}&\mathit{depthLessThan} :: \mathit{Tree} \to \mathit{Nat} \to \mathit{Bool}\\&\mathit{depthLessThan}\ \mathit{Leaf}\ \_ = \texttt{True}\\&\mathit{depthLessThan}\ (\mathit{Node}\ \_\ \_\ \_)\ \texttt{Zero} = \texttt{False}\\&\mathit{depthLessThan}\ (\mathit{Node}\ \mathit{t1}\ \_\ \mathit{t2})\ (\texttt{Suc}\ \ n) = \mathit{t1}\ `\mathit{depthLessThan}`\ n\ \wedge\ \mathit{t2}\ `\mathit{depthLessTh}\end{aligned}$$

In general, bespoke size limits are useful both to avoid backtracking, which makes property-based testing more efficient, and to allow greater flexibility in terms of distribution. The above properties are evaluated in the next chapter, where we see in practice that evaluating these properties with overlapping patterns does indeed only require trivial backtracking, and also that the use of overlapping patterns offers a substantial performance improvement over traditional narrowing.

5.3.2 Well-Typed Expressions

In this example we generate typed expressions for a simple language. We use this example to demonstrate a technique for building constraints that fail fast that combines well with the use of overlapping patterns. The language has addition, conditional expressions, natural numbers, and logical values:

$$\textbf{data}\ \mathit{Expr} = \mathit{Add}\ \mathit{Expr}\ \mathit{Expr} \mid \mathit{If}\ \mathit{Expr}\ \mathit{Expr}\ \mathit{Expr} \mid N\ \mathit{Nat} \mid B\ \mathit{Bool}$$

A useful property for this language states that for any well-typed expression up to a given depth, evaluating the expression will not produce an error:

$$\begin{aligned}&\mathit{propEval}\ n\ e\\&\quad = (\mathit{typed}\ e \implies \mathit{notError}\ (\mathit{eval}\ e))\\&\quad \text{`}\mathit{suchThat}\text{`}\ (\mathit{depthExpr}\ e \leqslant n)\end{aligned}$$

We will focus on the *typed* condition. This condition has a simple definition in terms of a more general function *typeOf* that attempts to determine the type of an expression, which may be either *Nat* or *Bool*, with the *Maybe* mechanism being used handle the possibility that an expression may be ill-typed:

$$\textbf{data}\ \mathit{Type} = \mathit{Nat} \mid \mathit{Bool}$$

$$\begin{aligned}&\mathit{typeOf} && :: \mathit{Expr} \to \mathit{Maybe}\ \mathit{Type}\\&\mathit{typeOf}\ (\mathit{Add}\ e\ e') && = \textbf{case}\ (\mathit{typeOf}\ e, \mathit{typeOf}\ e')\ \textbf{of}\\& && \qquad (\mathit{Just}\ \mathit{Nat}, \mathit{Just}\ \mathit{Nat}) \to \mathit{Just}\ \mathit{Nat}\\& && \qquad \_ \qquad\qquad\qquad\qquad\quad\ \to \mathit{Nothing}\end{aligned}$$

$$\begin{aligned}
&\mathit{typeOf}\ (\mathit{If}\ e\ e'\ e'') = \mathbf{case}\ (\mathit{typeOf}\ e, \mathit{typeOf}\ e', \mathit{typeOf}\ e'')\ \mathbf{of} \\
&\qquad\qquad (\mathit{Just\ Bool}, \mathit{Just}\ t', \mathit{Just}\ t'') \mid t' \equiv t'' \to \mathit{Just}\ t' \\
&\qquad\qquad \_ \qquad\qquad\qquad\qquad\qquad\qquad\quad \to \mathit{Nothing} \\
&\mathit{typeOf}\ (N\ \_) \qquad = \mathit{Just\ Nat} \\
&\mathit{typeOf}\ (B\ \_) \qquad = \mathit{Just\ Bool}
\end{aligned}$$

However, the function *typeOf* has an inefficient narrowing semantics. For example, an expression of the form $\mathit{If}\ (\mathit{Add}\ u\ v)\ w\ x$ is ill-typed for any u, v, w, x, because it is already evident that the first argument is not a logical value, but a version of *typed* defined using the function *typeOf* would not be able to deduce this until specific expressions had been filled in for the variables u and v. In other words, the *typed* condition does not fail fast.

To solve this problem we define an alternative constraint, $\mathit{hasType} :: \mathit{Expr} \to \mathit{Type} \to \mathit{Bool}$, in which the type of the expression is taken as an argument rather then returned as a result. In this manner, the type is refined during the narrowing process alongside the expression itself.

$$\begin{aligned}
&\mathit{hasType}\ (\mathit{Add}\ e\ e')\ \mathit{Nat} = \mathit{hasType}\ e\ \mathit{Nat} \ \wedge\ \mathit{hasType}\ e'\ \mathit{Nat} \\
&\mathit{hasType}\ (\mathit{If}\ e\ e'\ e'')\ t \quad = \mathit{hasType}\ e\ \mathit{Bool} \ \wedge\ \mathit{hasType}\ e'\ t \ \wedge\ \mathit{hasType}\ e''\ t \\
&\mathit{hasType}\ (N\ \_)\ \mathit{Nat} \qquad = \mathtt{True} \\
&\mathit{hasType}\ (B\ \_)\ \mathit{Bool} \qquad = \mathtt{True} \\
&\mathit{hasType}\ \_\ \_ \qquad\qquad\quad = \mathtt{False}
\end{aligned}$$

If we reconsider our example expression, $\mathit{If}\ (\mathit{Add}\ u\ v)\ w\ x$, then we can see our new typing constraint identifies this as being ill-typed:

$$\begin{aligned}
&\mathit{hasType}\ (\mathit{If}\ (\mathit{Add}\ u\ v)\ w\ x)\ t \\
&= \mathit{hasType}\ (\mathit{Add}\ u\ v)\ \mathit{Bool}\ \wedge\ \mathit{hasType}\ w\ t\ \wedge\ \mathit{hasType}\ x\ t \\
&= \mathtt{False}\ \wedge \mathit{hasType}\ w\ t\ \wedge\ \mathit{hasType}\ x\ t \\
&= \mathtt{False}
\end{aligned}$$

The *hasType* program does not fail fast but satisfies a similar weaker condition: any partial value formed by evaluating the constraint with free arguments either directly fails when applied to the constraint, or there is a refinement of the value that succeeds.

Using the *hasType* constraint, our original property concerning well-typed expressions up to a given depth can now be reformulated to include the type of the expression as an additional narrowing variable:

$$\begin{aligned}
&\mathit{propEval}\ n\ t\ e \\
&\quad = (\mathit{hasType}\ e\ t \implies \mathit{noError}\ (\mathit{eval}\ e)) \\
&\quad \text{`}\mathit{suchThat}\text{`}\ (\mathit{depthExpr}\ e \leqslant n)
\end{aligned}$$

Here we have used an additional narrowing variable to help enforce a global constraint, that the expression is typed. The essential idea is to realise the constraint as a datatype, in this case that datatype is simply the type of an expression, and then write a condition which relates this datatype to the data being generated. In this way sub-terms can be provided a shared context in a manner which is narrowable. Another example, is to generate perfectly balanced trees, in which all branches have the same depth. We need to share the depth across the sub-trees in a narrowable fashion and can do so with the following function,

$$\mathit{balanced} :: \mathit{Tree} \to \mathit{Nat} \to \mathit{Bool}$$
$$\mathit{balanced}\ \mathit{Leaf}\ x = x \equiv \texttt{Zero}$$
$$\mathit{balanced}\ (\mathit{Node}\ \_\ \_\ \_)\ \texttt{Zero} = \texttt{False}$$
$$\mathit{balanced}\ (\mathit{Node}\ t1\ \_\ t2)\ (\texttt{Suc}\ n) = \mathit{balanced}\ t1\ n\ \wedge\ \mathit{balanced}\ t2\ n$$

in which *balanced* t n is satisfied if all the branches of t have depth n. A property which takes n as a narrowing variable will generate balanced trees. We use a very similar function to enforce the balance condition of red-black trees in the next chapter.

5.4 Related Work

The functional logic language Curry [29] implements needed narrowing, and supports the use of overlapping patterns in definitions. However, the semantics is different to our system and in effect the two are unrelated. In particular, overlapping patterns in Curry are non-deterministic and provide a convenient syntax to use the inherent non-determinism of narrowing easily. Overlapping patterns in our language are deterministic and allow additional reductions in the narrowing semantics. Furthermore unlike Curry overlapping patterns they cannot be encoded using other narrowing constructs [5].

The form of overlapping patterns that we use in our system is similar to that proposed by Cockx [19, 20], who develops the idea in the context of dependent type theory and the Agda programming language. However, the intended purpose is different, with our aim being to improve the performance of property-based testing under a narrowing semantics, and Cockx seeking to simplify the development of proofs in a dependently-typed setting.

A number of narrowing-based testing tools use the notion of parallel conjunction. The idea originates in Lindblad's work on data generation [33] and Lazy Smallcheck [48], both of which use an enumerative style of testing. Subsequently, parallel conjunction has been used by Claessen et al. [15] to randomly generate data with a uniform distribution. Parallel conjunction is equivalent to overlapping conjunction, but whereas previous testing work using this operator has been more practically focused, we have given a precise narrowing semantics for a general form of overlapping definitions. The research of Claessen et al. is the most similar to our work, in that they also use a narrowing-style for random testing. However, their aim of producing a uniform distribution, using a variant of Feat [21], makes backtracking hard to avoid for many problems.

5.5 Conclusion and Future Work

In this chapter we have motivated and formalised an extension to our narrowing semantics to allow overlapping patterns in definitions. We saw two benefits of overlapping patterns in property-based testing. Firstly, in the evaluation of co-dependent conditions, and secondly, by enabling the encoding of size constraints. In the next chapter we give an account of the implementation and evaluate the performance benefits of overlapping patterns. Below we reflect on our new semantics and suggest some possible directions for further work.

While overlapping patterns can improve the performance of property-based testing, the use of narrowing can lead to subtle performance issues, as we saw in section 5.3 with the *typeOf* constraint. To avoid performance issues close attention must be paid to possible sources of backtracking.

Overlapping patterns help by making it easier to define constraints with limited backtracking, but they are no silver bullet, and further research is required to establish appropriate methodologies for identifying and limiting sources of backtracking.

In our original paper [27] we were unsure of the necessity of general overlapping definitions, stating that in most cases it might suffice to use overlapping conjunction along with the additional narrowing variables. Since then we have found multiple uses which seemingly cannot be encoded with the above scheme. For example, the author knows of no way of achieving the behaviour of *suchThat* function, and no way to enforce a limit on the number of nodes in tree, with only narrowing and overlapping conjunction. Although these uses are not strictly critical to property-based testing they certainly are useful in giving the programmer flexibility and in easing the creation of fast failing preconditions.

We discussed one possible area of future research in finding a less restrictive definition of confluence (section 5.2.3). The theory could also be extended by the addition of other language features, and how these interact with narrowing and overlapping patterns would require further research. Adding the capability to refine and narrow first and higher-order functions is one possible extension for which the trie representation of partial functions used in the extended Lazy Smallcheck [47] offers a starting direction. Another interesting area is the addition of locally bound narrowing variables, which are difficult to add directly to our theory as they have no analogy in traditional functional language.

It would also be interesting to investigate the analysis of programs which use overlapping patterns. For example, identifying cases where a condition does not fail fast would be useful. It would be very challenging to produce a static analysis which achieves this but a run-time analysis which determine when a condition is not failing fast could be practically useful.

Chapter 6

Implementation and Evaluation

In this chapter we develop an implementation of narrowing with overlapping patterns by extending the tool that was developed in chapter 4. We begin by defining the language and operational semantics, paying particular attention to how the semantics is designed to handle the combination of overlapping patterns and narrowing effectively. We then evaluate the performance of the tool on a number of case studies by comparing narrowing with overlapping patterns against the traditional form of narrowing.

6.1 Language and Semantics

In this section we give the core language and semantics of our implementation with overlapping patterns. Whereas the semantics in the previous chapter was designed to give an intuitive, theoretical account of overlapping patterns, the semantics of this chapter reflects the implementation closely. In particular, the semantics is call-by-need, is implemented with continuations for efficiency purposes, and is explicit in the order of evaluation and narrowing.

The core language is defined by the following grammar:

$$\begin{aligned}
&\mathit{Defn}_X ::= \mathit{Fun}\ \overline{\mathit{Var}} = \mathit{Expr}_X \\
&\mathit{Expr}_X ::= \mathit{Fun}\ \overline{\mathit{Expr}_X} \ \mid\ \mathit{Con}\ \overline{\mathit{Expr}_X} \ \mid\ \mathit{Expr}_X\ \mathit{Expr}_X \ \mid\ \texttt{var}\ \mathit{Var} \\
&\qquad\quad \mid\ \texttt{fvar}\ X \ \mid\ \textbf{let}\ \mathit{Var} = \mathit{Expr}_X\ \textbf{in}\ \mathit{Expr}_X \\
&\qquad\quad \mid\ \mathit{Cases}_X \\
&\mathit{Cases}_X ::= (\mathit{Case}_X \parallel \mathit{Cases}_X) \ \mid\ \mathit{Case}_X \\
&\mathit{Case}_X = \textbf{case}\ \mathit{Var}\ \textbf{of}\ \overline{\mathit{Con}\ \overline{\mathit{Var}} \to \mathit{Expr}_X} \\
&\mathit{Val}_X \ ::= \mathit{Con}\ \overline{\mathit{Val}_X} \ \mid\ \texttt{fvar}\ X
\end{aligned}$$

The language consists of definitions, functions, constructors applied to list of expressions, applications, variables, free variables, let expressions and overlapping case expressions. Unlike in chapter 4 we consider partially applied functions as they play an important role in the semantics. Constructor applications are still assumed to be complete and the language is assumed to be typed under the standard rules.

The syntax for overlapping patterns differs somewhat from chapter 5 to simplify the definition of the semantics. In particular, we have taken pattern matching out of definitions and instead represent overlapping patterns as a series of case expressions. For example, the overlapping definition of the *max* function is given by:

$$\begin{aligned}
max\ x\ y = \ & \textbf{case}\ x\ \textbf{of} \\
& \quad \texttt{Zero} \to y \\
& \quad (\texttt{Suc}\ x') \to \texttt{Suc}\ (max\ x'\ (pred\ y)) \\
\parallel\ & \textbf{case}\ y\ \textbf{of} \\
& \quad \texttt{Zero} \to x \\
& \quad (\texttt{Suc}\ y') \to \texttt{Suc}\ (max\ (pred\ x)\ y')
\end{aligned}$$

Here, the *max* function is represented by two case expressions, either of which can be reduced to proceed with evaluation. Once one of the case expressions has been matched the other case expressions can be dropped. The subjects of case expressions are restricted to variables in order to maximise sharing. With this restriction it is easy to convert an overlapping set of case expressions into an overlapping definition from chapter 5, by creating a function with arguments equal to the number of case expressions.

We typically denote definitions by f, expressions by e, constructors by $\mathbf{c}$, closed variables by u and v, and free variables by x and y. We denote a function of arity n as f_n and similarly a list with its length as $\overline{e}_n$.

Continuation semantics

The use of overlapping patterns places special requirements on the evaluation. Whereas for traditional narrowing we could apply a narrowing step immediately on encountering a free variable in the evaluation context, with overlapping patterns evaluation could still make progress on a separate branch. Therefore, an expression is only suspended when every overlapping branch is suspended. To represent this the semantics is given by interspersing a big-step reduction, which evaluates every branch until it is suspended, with narrowing steps.

As we have to traverse every branch of an expression after each narrowing step it is important to minimise the cost of doing so. Therefore, we utilise local stacks to store continuations, which reduces the need to traverse the expression and puts the focus on the evaluation context. A stack is defined as follows:

$$\mathit{Stack}_X ::= \emptyset \;\mid\; \bullet\; \mathit{Expr}_X; \mathit{Stack}_X \;\mid\; \texttt{var}\; \mathit{Var}; \mathit{Stack}_X$$

That is, the stack is either empty, has an application on top or has a variable on top.

Next, we define the big step semantics which reduces an expression to either a suspended form or a result. We do so by first defining a single-step reduction relation $\rightarrow$, which represents the standard, non-overlapping, part of the semantics and is a reduction of an environment consisting of an expression, a stack and a heap. In this setting, the heap is a mapping from variables to an expression *and* a stack, with the addition of a stack required to store progress when an expression is suspended.

First, we define the rules for applications and functions:

$$\frac{}{\langle e\, e',\ \kappa,\ s\rangle \rightarrow \langle e,\ \bullet e'; \kappa,\ s\rangle} \qquad \frac{m > n}{\langle f_m\, \overline{e}_n,\ \bullet e; \kappa,\ s\rangle \rightarrow \langle f_m\, \overline{e}e,\ \kappa,\ s\rangle}$$

$$\frac{(e, \overline{v}) = \mathit{fresh}(f)}{\langle f_m\, \overline{e}_m,\ \kappa,\ s\rangle \rightarrow \langle e,\ \kappa,\ s \cup \{v_i \mapsto (e_i, \emptyset) | v_i \in \overline{v}, e_i \in \overline{e}\}\rangle}$$

That is, we evaluate an application by pushing it to the stack. If we are evaluating an incomplete function application, then we pop an argument off the stack. Finally, if we are evaluating a complete function application then we inline the definition and add the arguments to the heap.

Next, we define the rules for let expressions and variables:

$$\frac{}{\langle \textbf{let } v = e \textbf{ in } e',\ \kappa,\ s\rangle \to \langle e',\ \kappa,\ s \cup \{v \mapsto (e, \emptyset)\}\rangle}$$

$$\frac{v \mapsto (e, \kappa') \in s}{\langle v,\ \kappa,\ s\rangle \to \langle e,\ \kappa'; \texttt{var}\ v; \kappa,\ s\rangle}$$

$$\frac{\mathit{whnf}(e)}{\langle e,\ \texttt{var}\, v; \kappa,\ s\rangle \to \langle e,\ \kappa,\ s[v \mapsto (e, \emptyset)]\rangle}$$

That is, to evaluate a let expression we add the expression to the heap with an empty stack. When a variable is to be evaluated, we retrieve its value from the heap and push the variable and its local stack onto the environment stack. When a variable is on top of a stack, and the expression is in weak head normal form we save this result onto the heap. An expression is in weak head normal form if it is a constructor or a partially applied function, and is defined formally as such:

$$\frac{}{\mathit{whnf}(\mathbf{c}\ \overline{e})} \qquad \frac{m > n}{\mathit{whnf}(f_m\ \overline{e}_n)}$$

We can now lift the sequence of local reductions into a big step semantics,

$$\frac{\mathit{whnf}(e)}{\langle e,\ \emptyset,\ s\rangle \Downarrow \langle e,\ \emptyset,\ s\rangle} \qquad \frac{\mathit{env} \to \mathit{env}' \qquad \mathit{env}' \Downarrow \mathit{env}''}{\mathit{env} \Downarrow \mathit{env}''}$$

which, in its current form is a standard functional semantics for the terms on which it is defined. We now need to define the semantics for an expression suspended on a free variable and for overlapping case expressions.

Suspending semantics

When an expression becomes suspended on a free variable we must store any progress made in evaluating the variables. We introduce a new state to our environment for when the expression is suspended, writing $\langle e \multimap x,\ \kappa,\ s\rangle$ for the expression e being suspended on x. We have the following rules,

$$\frac{}{\langle \texttt{fvar}\ x,\ \kappa,\ s\rangle \to \langle \texttt{fvar}\ x \multimap x,\ \kappa,\ s\rangle}$$

$$\frac{\forall u.\ \texttt{var}\ u \notin \kappa}{\langle e \multimap x,\ \kappa;\texttt{var}\,v;\kappa',\ s\rangle \to \langle \texttt{var}\ v \multimap x,\ \kappa',\ s[v \mapsto (e,\kappa)]\rangle}$$

where the first rule defines the transition to a suspended state, which occurs when a free variable is in focus. The second rule stores the current evaluation state into the first variable on the stack, and updates the variable with any progress which has been made in its evaluation.

We add another final state to the big-step semantics,

$$\frac{\forall u.\ \texttt{var}\ u \notin \kappa}{\langle e \multimap x,\ \kappa,\ s\rangle \Downarrow \langle e \multimap x,\ \kappa,\ s\rangle}$$

which states that a suspended expression without any more variables to update has completed evaluation.

Overlapping semantics

The most complex part of the semantics is the evaluation of overlapping patterns. We evaluate the case expressions in turn. If a case expression successfully matches, evaluation proceeds along that branch. If a case expression is suspended then we continue with the next case expression, unless there are no more in which case the overlapping pattern is suspended. To encode this we define a special reduction $\Downarrow^*$ which is a relation from a partial evaluated overlapping pattern to a resulting environment. The partially evaluated environment has the form $\langle cs \multimap x,\ cs',\ \kappa,\ s\rangle$, in which cs is the already evaluated component of the overlapping pattern which is suspended on x, cs' is the part to be evaluated, and κ, s are the stack and heap as normal.

The evaluation of an overlapping pattern is defined in terms of this relation as follows,

$$\frac{cs = e_0 \parallel .. \parallel e_n \qquad \langle \emptyset \multimap \bullet,\ cs,\ \kappa,\ s\rangle \Downarrow^* \langle e,\ \kappa',\ s'\rangle}{\langle cs,\ \kappa,\ s\rangle \Downarrow \langle e,\ \kappa',\ s'\rangle}$$

where $\bullet$ represents the absence of a free variable, and will be filled in subsequently. Note, the resulting expression e may or may not be suspended.

When the evaluation of the case subject succeeds we have the following rule:

$$\frac{\begin{array}{c}\langle \mathtt{var}\, v,\ \emptyset,\ s\rangle \Downarrow \langle \mathbf{c}\ \overline{e},\ \emptyset,\ s'\rangle \\ \langle e[\overline{v}/\overline{e}],\ \kappa,\ s'\rangle \Downarrow \langle e',\ \kappa',\ s''\rangle \qquad \mathbf{c}\ \overline{v} \mapsto e \in \overline{alt}\end{array}}{\langle cs \multimap x,\ \textbf{case}\ v\ \textbf{of}\ \overline{alt} \parallel cs',\ \kappa,\ s\rangle \Downarrow^* \langle e',\ \kappa',\ s''\rangle}$$

That is, the result of the case subject is matched with its alternative and evaluation continues along this branch.

When the case expression is suspended on its subject we have the following rule:

$$\frac{\langle \texttt{var}\ v,\ \emptyset,\ s\rangle \Downarrow \langle \texttt{var}\ v \multimap y,\ \emptyset,\ s'\rangle \qquad \langle (cs \parallel \mathbf{case}\ v\ \mathbf{of}\ \overline{alt}) \multimap (x \mathbin{<\!|\!>} y),\ cs',\ \kappa,\ s'\rangle \Downarrow^{*} \langle e',\ \kappa',\ s''\rangle}{\langle cs \multimap x,\ \mathbf{case}\ v\ \mathbf{of}\ \overline{alt} \parallel cs',\ \kappa,\ s\rangle \Downarrow^{*} \langle e',\ \kappa',\ s''\rangle}$$

That is, we proceed with evaluation by trying to evaluate the next case expression, and add the suspended case expression to those already evaluated. If an overlapping pattern is suspended we consider it to be suspended on the first case expression. Therefore, we define the function $<\!|\!>$ as follows:

- $\mathbin{<\!|\!>} y = y$

$x \ \mathbin{<\!|\!>} \_ = x$

Considering an overlapping pattern to be suspended on the first case expression falls short of the ideal of pattern matching being entirely order independent however it turns out to be helpful in practice. For example, we use *suchThat* to add size constraints to the end of properties and doing so ensures that we don't refine a test case prematurely just to satisfy a size constraint.

Finally, if we have exhausted the case expressions then the overlapping pattern is suspended:

$$\frac{}{\langle cs \multimap x,\ \emptyset,\ \kappa,\ s\rangle \Downarrow^{*} \langle cs \multimap x,\ \kappa,\ s\rangle}$$

This finishes the definition of $\Downarrow$ which reduces an expression either to a result or until it is suspended on all branches.

Narrowing semantics

We can define the narrowing semantics now by interspersing narrowing steps between evaluations. The $\rightsquigarrow$ reduction is defined as follows:

$$\frac{\langle e,\ \kappa,\ s\rangle \Downarrow \langle e',\ \emptyset,\ s'\rangle \qquad \mathit{whnf}(e')}{\langle e,\ \kappa,\ s\rangle \rightsquigarrow \langle e',\ s',\ \mathit{return}\rangle}$$

$$\frac{\sigma \in \mathit{Narr}_X(x) \qquad \langle e,\ \kappa,\ s\rangle \Downarrow \langle e' \multimap x,\ \kappa',\ s'\rangle \qquad \langle e'[\sigma],\ \kappa'[\sigma],\ s'[\sigma]\rangle \rightsquigarrow \langle e'',\ s'',\ \sigma'\rangle}{\langle e \multimap x,\ \kappa,\ s\rangle \rightsquigarrow \langle e'',\ s'',\ \sigma \mathrel{>\!\!=\!\!>} \sigma'\rangle}$$

That is, if the expression is in weak head normal form then evaluation terminates. Otherwise, the expression is suspended and a refinement is chosen from the narrowing set $\mathit{Narr}_X(x)$, which was defined in section 5.2, and applied to the environment before the expression re-evaluated.

6.2 Implementation

The implementation is an abstract machine written in Haskell, which extends the implementation of chapter 4. The implementation and the case studies in this chapter can be found in the following repository:

`https://github.com/jonfowler/narrowcheck`

The language used is the same subset of Haskell as in chapter 4, with the addition of functions defined using overlapping patterns. This language is desugared into the core language given in section 6.1. Overlapping functions are denoted using a pragma. For example,

$$
\begin{array}{l}
\{\text{-\# OVERLAP } \mathit{max} \text{ \#-}\} \\
\mathit{max}\ \texttt{Zero}\ y \quad = y \\
\mathit{max}\ (\texttt{Suc}\ x)\ y = \texttt{Suc}\ (\mathit{max}\ x\ (\mathit{pred}\ y)) \\
\mathit{max}\ x\ \texttt{Zero} \quad = x \\
\mathit{max}\ x\ (\texttt{Suc}\ y) = \texttt{Suc}\ (\mathit{max}\ (\mathit{pred}\ x)\ y)
\end{array}
$$

is an overlapping definition of *max* and will be desugared to the function given in the previous section. The implementation has similar features as the previous implementation. That is, it converts expressions to an atomic form (section 4.1), implements narrowing using the same search tree (section 4.2) and uses the same search strategies (section 4.3).

The main difference in the implementation is a new *eval* function which reflects the new semantics. As in the semantics, we define the evaluation by first defining a function that reduces an expression until it is suspended on a free variable, and then defining *eval* by interspersing these reductions with narrowing steps.

The type of *evalToSuspend*, which corresponds to $\Downarrow$, is as follows:

$$
\begin{array}{l}
\textbf{type}\ \mathit{Env} = (\mathit{Stack}, \mathit{Heap}) \\
\mathit{evalToSuspend} :: \mathit{Monad}\ m \Rightarrow \\
\qquad \mathit{Expr} \rightarrow \mathit{StateT}\ \mathit{Env}\ m\ (\mathit{Expr}, \mathit{Maybe}\ \mathit{FreeVar})
\end{array}
$$

That is, *evalToSuspend* takes an expression and produces a new expression, along with either a free variable when the new expression is suspended, or *Nothing* in which case the expression will be in weak head normal form. During the reduction, updates will be made to the environment, which is formed of a heap and a stack, but no narrowing steps will be performed. The implementation of *evalToSuspend* follows $\Downarrow$ closely.

The *evalToSuspend* function is then utilised to realise the full overlapping narrowing semantics $\rightsquigarrow$ in the following *eval* function:

```
type Narrow = StateT Env Refine

eval :: Expr → Narrow (Expr, Sub)
eval e = do
  (e', suspendedOn) ← evalToSuspend e
  case suspendedOn of
    Nothing → return (e', subReturn)
    Just x → do
      σ ← branch $ narrowingSet x
      refineEnv σ
      (e'', σ') ← eval (subst σ e')
      return (e'', σ >=> σ')
```

The function uses the *Refine* monad defined in section 4.2, which handles the non-determinism caused from the choice of narrowing steps. The evaluation uses one additional helper function, *refineEnv* :: *Sub* → *Narrow* (), which applies a substitution to the environment. The *return* substitution is denoted *subReturn*. The definition follows $\rightsquigarrow$ closely, with the two branches of the case expression corresponding to the two rules.

6.2.1 Optimisations

The implementation contains two optimisations not covered by the semantics given for the core language, which we describe briefly below.

1. If an expression becomes suspended in an environment with no overlapping case expressions, then a narrowing step can be performed directly, and by doing so we avoid having to traverse the expression

tree. This can be implemented easily by keeping a Boolean in the environment representing whether we are in a unique non-overlapping branch.

2. To avoid having to traverse the entire tree after each narrowing step, at each overlapping branch we keep track of the free variables the branch is suspended on. When we perform a narrowing step, we consider the refined free variable the *active variable*, and on the subsequent evaluation we only evaluate the branches which are suspended on this variable.

6.3 Case Studies

In this section, we compare the performance of evaluating properties with and without the use of overlapping patterns. We consider testing by enumerating test cases and random choice, and we measure the metrics for those that we defined in section 4.4. We consider the union and ordered tree properties from chapter 4, the permutation and well-typed expression properties from chapter 5, and two new case studies. We do not consider the reverse and Huffman properties as these are not impacted by narrowing for the reasons explained in chapter 4, and therefore overlapping patterns will also not impact their evaluation. The code for the case studies can be found in appendix A and on Github [25].

6.3.1 Size Limits and Distribution

The use of overlapping patterns to define bespoke size limits gives an important performance benefit for some problems, but also makes direct comparison to a traditional narrowing evaluation difficult. This is because if we use a bespoke size limit for the overlapping version and traditional size limit for the traditional version, the distribution of the test cases differ substantially. For this reason, we opt to use overlapping patterns in traditional properties but only to encode bespoke size limits.

We can give a size parameter to the tool using the `-s` flag:

```
$ narrowcheck -s 8 -p propSort Perm.hs
+++ Ok, successfully passed 100 tests in 0.59s
```

Properties with bespoke size limits are represented by a function in which the first argument is the size parameter and is of type *Nat*.

6.3.2 Evaluation

We evaluate the case studies in the same fashion as in section 4.4, which we recall for convenience. For enumerative testing we repeated each experiment ten times with a time limit of twelve minutes. For random testing we ran one thousand test cases and repeated each experiment forty times with a time limit of four minutes. For both we increased the size limit incrementally until an experiment exceeds the time limit. All results reported were obtained using a quad-core Intel i5 running at 3.2GHz, with 16GB RAM, under 64-bit Ubuntu 16.04 LTS with kernel 4.4.0.

We use a backtrack limit of 30 for random testing, as we found this to be a reasonable compromise in the case studies of section 4.4. When testing by enumeration we use the evaluation sharing definition (section 4.3), as this proved previously to give a performance benefit.

6.3.3 Sorted Permutations

First we consider the motivational example for overlapping patterns given in chapter 5. The property asserts that sorting a permutation should give an ascending sequence and is given by:

$$\begin{aligned} &\mathit{propSort} :: \mathit{Nat} \to [\mathit{Nat}] \to \mathit{Bool} \\ &\mathit{propSort}\ n\ l \quad = \quad \mathit{length}\ l \equiv n\ \wedge\ \mathit{all}\ ({<}n)\ l\ \wedge\ \mathit{allDiff}\ l \\ &\qquad\qquad\quad \Longrightarrow\ \mathit{sort}\ l \equiv [0\mathinner{..}(n-1)] \end{aligned}$$

The property requires no bespoke size limit, as the first the two components of the precondition restrain the size of a test case.

The performance results for testing the permutation property are given in Figure 6.1. Table 6.1a shows the time taken to enumerate test cases at different size limits, and also shows the number of successful and invalid test cases at each limit. Graph 6.1c is a plot of the time taken at these size limits. Note that the scale here, and for other enumeration graphs, is logarithmic as the time taken to enumerate tests cases typically grows exponential with the size limit.

Table 6.1b shows the time taken to produce 1000 random permutations of the given size and also includes the success rate for creating these permutations. As the size of a permutation is preordained we do not measure an average size for this property. Graph 6.1d is a plot of time taken at these size limits. This graph, along with the other graphs for random performance, has a linear scale.

Enumeration

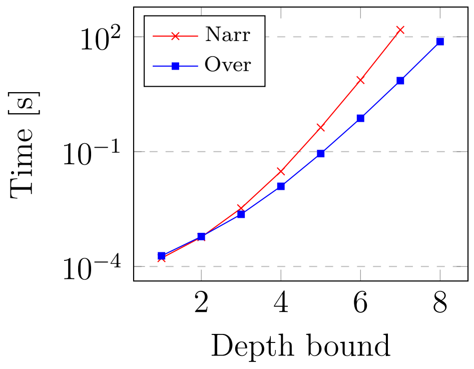
The results for the testing *propSort* by enumeration can be found in the table 6.1a and the graph 6.1c.

Strat	**Metric**	**3**	**4**	**5**	**6**	**7**	**8**
Narr	time	3.32ms	30.5ms	427ms	7.40s	151s	-
	tests	6	24	120	720	5040	-
	invalid	38.0	322	3792	5.5E4	9.6E5	-
Over	time	2.31ms	12.4ms	89.0ms	744ms	7.15s	74.9s
	tests	6	24	120	720	5040	4.0E4
	invalid	29.0	146	917	6710	5.6E4	5.1E5

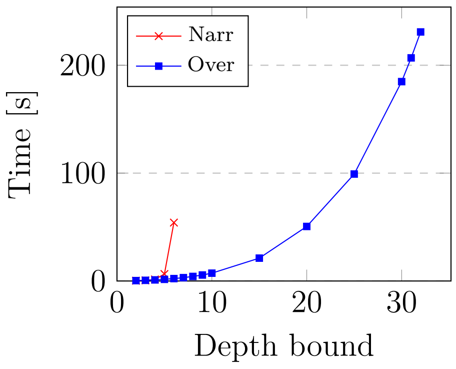
(a) Benchmark results for enumerating all tests of the permutation property of a given size

Strat	**Metric**	**5**	**6**	**10**	**20**	**32**
Narr	time	6.48s	54.2s	-	-	-
	success	100.0%	100.0%	-	-	-
Over	time	1.54s	2.23s	7.26s	50.6s	231s
	success	100.0%	100.0%	100.0%	100.0%	100.0%

(b) Benchmark results for testing 1000 permutations of a given size



(c) Time taken to test the permutation property by enumeration



(d) Testing permutation problem on 1000 randomly generated inputs

Figure 6.1: Benchmark results for testing the **permutation** property

Observation 1. Evaluating the property utilising overlapping patterns is significantly faster. Enumerating values of depth seven takes over twenty times as long without overlapping patterns.

This performance result is driven by the differing number of test cases which need to be considered. We can see the traditional narrowing evaluation evaluates almost a million test cases, almost all invalid, at depth seven with the overlapping version only needing to check around 60 thousand. This is due to the co-dependent conditions which cause the traditional narrowing strategy to backtrack over a large number of invalid test cases. For example, as we saw in section 5.1, traditional narrowing cannot determine that $[1, 1, x_2, x_3]$ can never be a valid permutation and will have to backtrack over all combinations of x_2, x_3. In contrast, overlapping patterns will determine this is not valid immediately and will do so for any invalid prefix. It should be noted that the overlapping pattern version still does not satisfy our fails fast condition (defined in section 5.3), because sometimes multiple backtracking steps are required to find a suitable element of the permutation.

Random

All constructors are given equal weight and a backtrack limit of 30 is used. The results can be found in the table 6.1b and the graph 6.1d. As all permutations are the same "size" we don't include a size metric.

Observation 2. Random testing is more effective with overlapping patterns. In the allocated time overlapping evaluation can generate 1000 values of depth 32 whereas the maximum depth achieved by traditional evaluation was 6.

The difference in performance is again explained by the co-dependent patterns, with traditional evaluation having to perform large amounts of backtracking. There is no benefit to random testing for traditional evaluation as it is unable to test 1000 random permutation of size seven in the given time, but was able to enumerate all of these permutations within the same time period. Random testing with overlapping patterns however is beneficial as 1000 tests of size 32 can be performed whereas a maximum of size 8 was reached by enumeration.

6.3.4 Union of Sets

We evaluate the performance of the union property given in section 4.4. The property checks whether a *union* function produces a valid set when given two valid sets:

$$\begin{array}{l} \mathit{propUnion} :: \mathit{Nat} \to [\mathit{Nat}] \to [\mathit{Nat}] \to \mathit{Result} \\ \mathit{propUnion}\ n\ x\ y = \mathit{suchThat} \\ \quad (\mathit{set}\ x\ \wedge\ \mathit{set}\ y\ \Longrightarrow\ \mathit{set}\ (\mathit{union}\ x\ y)) \\ \quad (\mathit{length}\ x \leqslant n\ \curlywedge\ \mathit{length}\ y \leqslant n\ \curlywedge\ \mathit{all}\ (\leqslant n)\ x\ \curlywedge\ \mathit{all}\ (\leqslant n)\ y) \end{array}$$

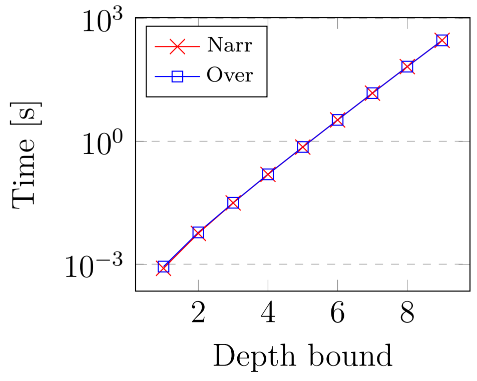
A *set* is represented by an ordered list. We constrain the length of the list and each element of the list to be less than or equal to the given size. We use the same size constraint for both the narrowing and overlapping versions of the tests (section 6.3.1). Note, this size constraint utilises overlapping conjunction, as does the definition of *suchThat*. To represent this we use the $\curlywedge$ operator to denote conjunction that is overlapping even when using traditional evaluation.

Strat	**Metric**	**4**	**5**	**6**	**7**	**8**	**9**
Narr	time	157ms	728ms	3.34s	15.1s	66.9s	293s
	tests	961	3969	1.6E4	6.5E4	2.6E5	1.0E6
	invalid	2021	9533	4.3E4	1.9E5	8.5E5	3.7E6
Over	time	157ms	731ms	3.35s	15.1s	66.9s	293s
	tests	961	3969	1.6E4	6.5E4	2.6E5	1.0E6
	invalid	2021	9533	4.3E4	1.9E5	8.5E5	3.7E6

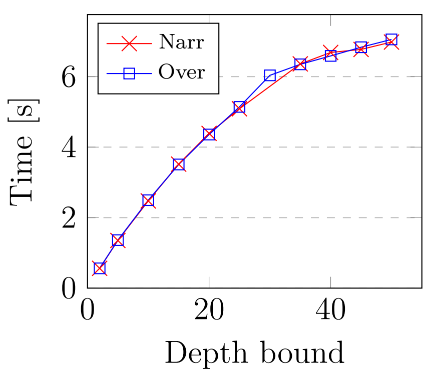
(a) Benchmark results for enumerating tests of the union property with maximum construction depth

Strat	**Metric**	**10**	**20**	**30**	**40**	∞
Narr	time	2.47s	4.39s	5.99s	6.68s	7.69s
	success	100.0%	100.0%	91.3%	96.0%	100.0%
	size	3.40	4.35	4.23	4.58	5.04
Over	time	2.49s	4.36s	6.04s	6.59s	7.52s
	success	100.0%	100.0%	91.0%	96.3%	100.0%
	size	3.41	4.32	4.21	4.57	4.99

(b) Benchmark results for 1000 random tests of the union property with maximum construction depth



(c) Time taken to test the union property by enumeration



(d) Testing the union property on 1000 randomly generated inputs

Figure 6.2: Benchmark results for testing the **union** property

Enumeration

The results for testing *propUnion* by enumeration can be found in the table 6.2a and the graph 6.2c.

Observation 3. There is no significant performance difference between evaluation with and without overlapping patterns.

Although the definition of *set*,

$$set\ [\,] = \texttt{True}$$
$$set\ (a : l) = set'\ a\ l$$
$$set'\ a\ [\,] = \texttt{True}$$
$$set'\ a\ (a' : l) = a < a' \ \wedge\ set'\ a'\ l$$

contains a conjunction, whether it is overlapping has little impact on performance. Although both sides of the conjunction depend on a', progress on the right side can only be made once l has been refined and therefore overlapping patterns have no impact.

Random

All constructors are given equal weight and a backtrack limit of 30 is used. The results can be found in the table 6.2b and the graph 6.2d. The size metric is given by the average number of elements in the sets.

Observation 4. Once again, the two modes of evaluation have no significant performance difference.

There is little performance difference for the same reason as when testing by enumeration. That is, although *set* uses an overlapping conjunction, the order the free variables are refined means the left side of the conjunction is completely evaluated before any progress is made on the right side.

6.3.5 Ordered Trees

Next we assess the ordered tree property which we used as a case study for overlapping patterns. The property is given by,

$$\begin{aligned}
&\mathit{propDelete} :: \mathit{Nat} \to \mathit{Nat} \to \mathit{Tree} \to \mathit{Result} \\
&\mathit{propDelete}\ n\ a\ t = \mathit{suchThat} \\
&\quad (\mathit{ordered}\ t \implies \mathit{ordered}\ (\mathit{delete}\ a\ t)) \\
&\quad (\mathit{sizeLimit}\ n\ a\ t)
\end{aligned}$$

in which we use different definitions of *sizeLimit* for enumeration and random testing.

Enumeration

The results for testing *propDelete* by enumeration can be found in the table 6.3a and the graph 6.3c. For enumeration we limit the total number of nodes in the tree and the size of each element:

$$\mathit{sizeLimit}\ n\ a\ t = a \leqslant n\ \curlywedge\ \mathit{countNodes}\ t \leqslant n\ \curlywedge\ \mathit{maxNode}\ t \leqslant n$$

The definition of *countNodes* and *maxNode* use the overlapping version of + and *max* respectively. For tree-based structures it is often useful to limit the number of nodes rather than the depth, as it is typically only possible to enumerate to a low max depth as the number of test cases grows extremely quickly.

Observation 5. Enumerating with overlapping patterns is significantly faster than without. At depth six the overlapping version is over four times as fast.

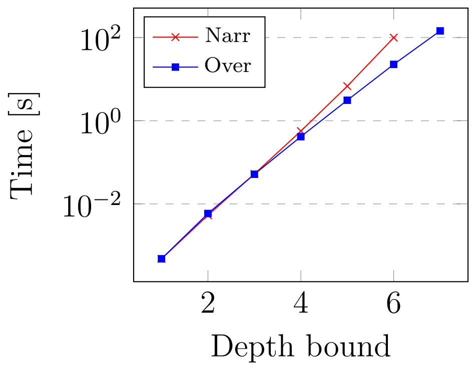
Once again, we find this performance difference is largely driven by the difference in the number of potential test cases considered. At depth six,

Strat	**Metric**	**3**	**4**	**5**	**6**	**7**
Narr	time	52.4ms	563ms	6.77s	99.3s	-
	tests	211	1191	6483	3.4E4	-
	invalid	431	4346	5.1E4	6.8E5	-
Over	time	51.5ms	413ms	3.10s	22.6s	144s
	tests	211	1191	6483	3.4E4	1.8E5
	invalid	361	2536	1.6E4	1.0E5	5.9E5

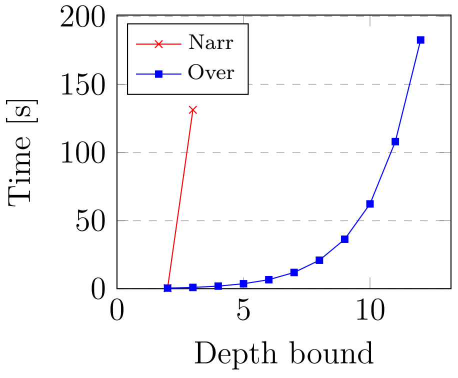
(a) Benchmark results for enumerating all ordered trees with up to a given number of nodes

Strat	**Metric**	**2**	**3**	**4**	**10**	**12**
Narr	time	343ms	131s	-	-	-
	success	100.0%	90.1%	-	-	-
	size	1.55	2.49	-	-	-
Over	time	417ms	911ms	1.87s	62.3s	183s
	success	100.0%	100.0%	100.0%	100.0%	100.0%
	size	1.56	2.71	4.31	33.6	60.5

(b) Benchmark results for testing 1000 ordered trees with given maximum depth



(c) Time taken to test the ordered tree property by enumeration



(d) Testing ordered tree property on 1000 randomly generated inputs

Figure 6.3: Benchmark results for testing the **ordered tree** property

overlapping patterns has to evaluate less than a fifth of the test cases. This is because of co-dependent patterns in the *ordered* constraint,

$$\begin{aligned} &\mathit{ordered}\ \mathit{Leaf} = \texttt{True} \\ &\mathit{ordered}\ (\mathit{Node}\ t1\ \ a\ t2) \ = \ \mathit{all}\ (\leqslant a)\ t1 \ \wedge\ \mathit{ord}\ t1 \\ &\qquad\qquad\qquad\qquad\quad \wedge\ \mathit{all}\ (\geqslant a)\ t2 \ \wedge\ \mathit{ord}\ t2 \end{aligned}$$

in which the first and last pair of constraints are co-dependent.

Random

As in section 4.4, the node constructor is given a weight of two and the leaf constructor a weight of one. The backtrack limit of 30 is used. The results can be found in the table 6.3b and the graph 6.3d. The average size metric is given by the number of nodes in the graph. We use a depth limit to restrain the size:

$$\mathit{depthLimit}\ n\ t = \mathit{maxDepth}\ t \leqslant n$$

We do not have to limit the size of the elements as they follow a geometric distribution and therefore their size is limited probabilistically.

Observation 6. Evaluating with overlapping patterns is far more effective in random testing. The maximum depth obtained within the time limit without overlapping patterns was three with an average size of two and a half. The maximum depth achieved with overlapping patterns was twelve with an average size of over sixty.

The performance difference is far more pronounced than in enumerative testing. This is likely because when backtracking in enumerative testing the cost is amortised across successful test cases, whereas in random testing a large amount of backtracking may be required to generate a single value.

We can see in the overlapping case that the generation of test cases is 100% successful, as we noted in the previous chapter the ordered tree precondition satisfies the fails fast condition and therefore only ever backtracks a single time. We can confirm this experimentally by setting the backtrack limit to 1 and seeing that the success rate is still 100%.

Overall, random testing is effective with overlapping patterns in the sense that it allows the testing of far bigger trees than using enumeration. The average size of sixty in the final test run is far greater than the max enumerated size of seven. With traditional narrowing this is not the case, with the average size being less than three.

6.3.6 Well-Typed Expressions

Next, we look at the well-typed expression property which we used to demonstrate the technique of using narrowing variables to enforce constraints (section 4.4). The property

$$\begin{aligned}
&\mathit{propEval} :: \mathit{Nat} \to \mathit{Type} \to \mathit{Expr} \to \mathit{Result}\\
&\mathit{propEval}\ n\ t\ e = \mathit{suchThat}\\
&\quad (\mathit{hasType}\ e\ t \implies \mathit{noError}\ (\mathit{eval}\ e))\\
&\quad (\mathit{sizeLimit}\ n\ e)
\end{aligned}$$

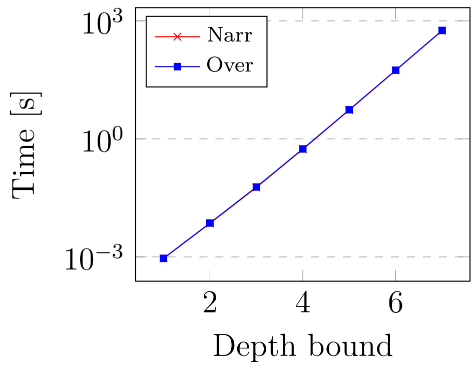
asserts that a well-typed expression evaluates without error. The use of an additional narrowing variable for the type ensures the property has efficient narrowing semantics. We use different size limits for enumeration and random testing.

Strat	**Metric**	**2**	**3**	**4**	**5**	**6**	**7**
Narr	time	7.11ms	58.4ms	546ms	5.42s	55.0s	567s
	tests	45	332	2791	2.5E4	2.3E5	2.2E6
	invalid	196	1422	1.2E4	1.1E5	1.0E6	1.0E7
Over	time	7.25ms	59.7ms	555ms	5.50s	55.7s	571s
	tests	45	332	2791	2.5E4	2.3E5	2.2E6
	invalid	196	1422	1.2E4	1.1E5	1.0E6	1.0E7

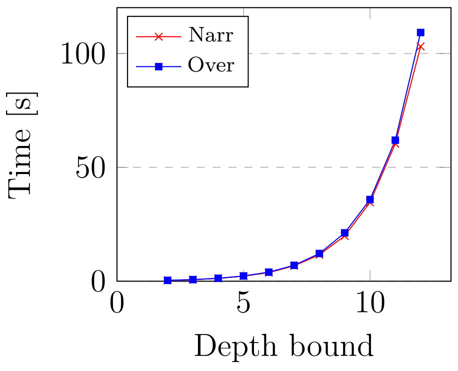
(a) Benchmark results for enumerating all expressions up to maximum depth

Strat	**Metric**	**4**	**6**	**8**	**10**	**12**
Narr	time	1.25s	3.78s	11.6s	34.6s	103s
	success	100.0%	100.0%	100.0%	100.0%	100.0%
	size	12.0	31.2	77.7	189	439
Over	time	1.28s	3.98s	12.2s	35.9s	109s
	success	100.0%	100.0%	100.0%	100.0%	100.0%
	size	11.9	30.8	78.2	185	438

(b) Benchmark results for testing 1000 expressions with given maximum depth



(c) Time taken to test the well-typed expression property by enumeration



(d) Testing well-typed expression property on 1000 randomly generated inputs

Figure 6.4: Benchmark results for testing the **well-typed expression** property

Enumeration

The results for testing *propEval* by enumeration can be found in the table 6.4a and the graph 6.4c. The size restraint,

$$
\begin{aligned}
&\mathit{countExpr}\ (B\ \_) = \texttt{Zero}\\
&\mathit{countExpr}\ (N\ n) = n\\
&\mathit{countExpr}\ (\mathit{Add}\ e\ e') = \texttt{Suc}\ (\mathit{countExpr}\ e + \mathit{countExpr}\ e')\\
&\mathit{countExpr}\ (\mathit{If}\ e\ e'\ e'')\\
&\quad = \texttt{Suc}\ (\mathit{countExpr}\ e + \mathit{countExpr}\ e' + \mathit{countExpr}\ e'')\\
&\mathit{sizeLimit}\ n\ e = \mathit{countExpr}\ e \leqslant n
\end{aligned}
$$

places a limit on the number of nodes. We give Booleans and the zero natural the weight of zero. This ensures that there is always a valid refinement that can be made during evaluation, which is not the case with a traditional size limit. Ensuring size constraints always allow a valid refinement is good principle and is generally achievable by identifying bases cases which can be given “zero” size.

Observation 7. Performance with and without overlapping patterns is very similar.

Once again, the *hasType* condition contains no co-dependent constraints and therefore there is no substantial performance difference. The overlapping version is consistently slightly slower, which is likely due to the overhead of traversing the overlapping conjunction in *hasType*.

Random

All constructors are given equal weight and a backtrack limit of 30 is used. The results can be found in the table 6.4b and the graph 6.4d. The size limit is given by the depth of an expression,

$$\mathit{sizeLimit}\ n\ e = \mathit{depthExpr}\ e \leqslant n$$

where *depthExpr* is equivalent to *countExpr* with addition replaced by the maximum function. The average size metric is calculated by counting the number of $\mathit{Add}, \mathit{If}, N, B$ constructors in the expression.

Observation 8. The overlapping version takes around 5% longer to evaluate than the non-overlapping version.

Again, this difference is likely caused by the additional overhead of traversing the overlapping conjunction in *hasType*. We find both evaluation strategies able to test large well-typed expressions effectively, with each being able to test 1000 expressions with an average of over 400 nodes within the time limit.

6.3.7 N-Queens Constraint Problem

Here we change our focus to a constraint problem and consider the classic puzzle of finding a layout of n queens on a $n \times n$ chess board such that no queens threatens another. For this problem we use our tool in generate mode for which it generates solutions to a given predicate. For example we can execute the command,

```
$ narrowcheck -g -e -s 4 -p nQueens Perm.hs
+++ Ok, enumerated 2 solutions in 18.2ms
[2,4,1,3]
[3,1,4,2]
```

where flag `-g` indicates we are generating solutions, and flag `-e` that we are enumerating. The example gives the solutions to the 4-queens problem in which board is represented by a list of naturals, each representing the position of a queen on a row. The constraint problem is given by:

$$\begin{aligned}
&\mathit{nQueens} :: \mathit{Nat} \to [\mathit{Nat}] \to \mathit{Bool}\\
&\mathit{nQueens}\ n\ x\\
&\quad = \ \mathit{length}\ x \equiv n \ \wedge\ \mathit{all}\ ({<}n)\ x \ \wedge\ \mathit{allDiff}\ x\\
&\quad \wedge\ \mathit{diagonalAsc}\ x \ \wedge\ \mathit{diagonalDesc}\ x
\end{aligned}$$

where the first two constraints check the board is of size n, the next constraint checks the columns contain only one queen, and the final two constraints check that no diagonal contains more than one queen.

Enumeration

The results for enumerating solutions for a given n-queen problem can be found in the table 6.5a and the graph 6.5c.

Observation 9. The use of overlapping patterns improves performance significantly. Within the allocated time the overlapping version is able to enumerate the 2680 solutions to the 11-queens problem whereas the non-overlapping version is only able to enumerate the forty solutions to the 7-queens problem.

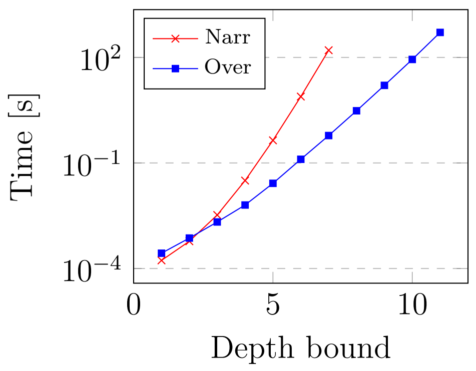
This problem is similar to the *permutation* example, as the constraint is the same as the *permutation* condition with the addition of two extra constraints on the diagonals. We find that evaluation without overlapping patterns performs similarly to that problem, with both enumerating size 7 lists in around 150 seconds. The overlapping version however can enumerate to greater depths on this problem, enumerating size 11 lists in the allocated time, compared to 8 in the permutation example. This is because the additional constraints allow many partial values to be discarded early

Strat	**Metric**	**6**	**7**	**8**	**9**	**10**	**11**
Narr	time	7.72s	160s	-	-	-	-
	solutions	4	40	-	-	-	-
	failures	5.6E4	9.6E5	-	-	-	-
Over	time	127ms	602ms	3.05s	16.1s	87.8s	514s
	solutions	4	40	92	352	724	2680
	failures	898	3553	1.6E4	7.2E4	3.5E5	1.8E6

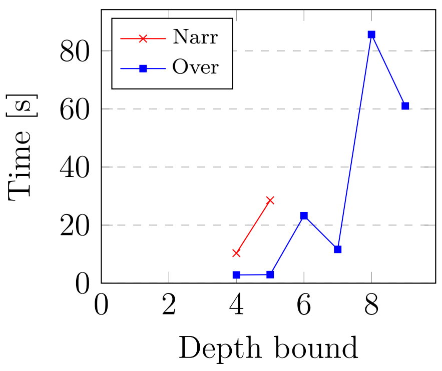
(a) Benchmark results for enumerating the solutions to the n-queens problem

Strat	**Metric**	**4**	**5**	**6**	**7**	**8**	**9**
Narr	time	10.3s	28.5s	-	-	-	-
	success	100.0%	100.0%	-	-	-	-
Over	time	2.85s	2.94s	23.2s	11.6s	85.6s	61.0s
	success	100.0%	100.0%	100.0%	100.0%	44.4%	47.6%

(b) Benchmark results for generating 1000 solutions to the n-queens problem



(c) Time taken to enumerate solutions to the n-queens problem



(d) Generating 1000 solutions to the n-queens constraint problem

Figure 6.5: Benchmark results for solving the **n-queens problem**

– the total number of partial values considered at depth 8 is around 1600 in this problem, whereas it is over 50,000 when evaluating the permutation property.

Random

All constructors are given equal weight and a backtrack limit of 30 is used. The results can be found in the table 6.5b and the graph 6.5d.

Observation 10. Again, the use of overlapping patterns significantly improves performance. Using overlapping patterns we can generate 1000 solutions up to size 9 within the allocated time, whereas without it was only possible to generate up to size 5.

We find that both evaluation strategies are slower at generating solutions than in the permutation example. This isn't surprising as the additional constraints make it harder to find a solution. With overlapping patterns this difference is very large. In the permutation example we were able to test up to size 32 whereas we were only able to generate up to size 9 here. This is because when generating permutations it will always be possible to find a valid element as long as the initial segment is valid. This limits the amount of backtracking required. However in the n-queens problem it might not be possible to place a queen in a row, even though the initial rows obey the constraints. For example, $[1, 3, x]$, has no possible placement for x even though the first two queens do not threaten each other. This means that sometimes large amounts of backtracking are required. We can see this experimentally as at sizes 8 and 9 the tool fails to find a solution around 50% of the time with a backtrack limit of 30.

6.3.8 Red-Black Trees

Finally, we consider a balanced tree implementation in the form of ordered red-black trees, as given by Okasaki [42]. The tree is made of nodes, each coloured either red or black. Every branch of the tree has to have the same number of black nodes and no red node can have a red child. These two constraints together imply that branches differ in length by a factor of two at most, which is the sense in which a red-black tree is balanced. We define the data-type for red-black trees as follows:

```
data RedBlack
  = L                                    -- Leaves
  | N Colour RedBlack Nat RedBlack       -- Coloured nodes
data Colour
  = R   -- Red
  | B   -- Black
```

That is, a red-black tree is a binary tree in which the nodes are coloured and contain an element in the form of a natural number. As usual, these elements should be ordered in a valid tree. Note, we have used short names as it is convenient in the definition of *insert* (Fig. 6.6).

We define the condition of tree being red-black by

```
redBlack :: Nat → RedBlack → Bool
redBlack b t = rootBlack t ∧ black b t ∧ red t ∧ ordered t
```

in which the only constraint which we have not already stated is *rootBlack*, which asserts that the root node is black. The *black* condition is the most interesting. This condition takes an argument b, which will be used as a narrowing variable, and asserts that each branch has b black nodes. This definition has efficient narrowing semantics, and similarly to *hasType*, the narrowing variable is used to share context between multiple branches while allowing the context to be refinable.

The property we test is on a faulty implementation of an insert function, shown in Figure 6.6, which was considered by Naylor [38]. The fault is especially interesting as it is in a rarely evaluated branch of the program, and therefore can be difficult to detect. The property is:

$$
\begin{aligned}
&\mathit{propInsert} :: \mathit{Nat} \to \mathit{Nat} \to \mathit{Nat} \to \mathit{Tree} \to \mathit{Result}\\
&\mathit{propInsert}\ n\ b\ a\ t = \mathit{suchThat}\\
&\quad (\mathit{redBlack}\ b\ t \implies \mathit{redBlack}'\ (\mathit{insert}\ a\ t))\\
&\quad (\mathit{sizeLimit}\ n\ b\ a\ t)
\end{aligned}
$$

The property asserts that a red-black tree should still be red-black after it has a new element inserted. On the right side we use a second version of the red-black condition, *redBlack′*, which is the same as *redBlack* but does not take a narrowing variable. We do so because the depth of black nodes in the resulting tree might differ from the original tree. Ideally, we would be able to write something akin to $\mathit{exists}\ b' \circ \mathit{redBlack}\ b'\ (\mathit{insert}\ a\ t\ t)$ in which a new narrowing variable is introduced with existential quantification i.e. to satisfy the condition we need to find a value b' that satisfies condition and to refute it we need to show none do. Adding such a feature could be an avenue of future research.

Enumeration

The results of testing *propInsert* by enumeration can be found in the table 6.7a and the graph 6.7c. We constrain the total number of nodes and the magnitude of each element by the size limit. This can be encoded as:

```
data RedBlack
  = L                                    -- Leaves
  | N Colour RedBlack Nat RedBlack       -- Coloured nodes
data Colour
  = R   -- Red
  | B   -- Black
insert :: Nat → RedBlack → RedBlack
insert x L = N R L x L
insert x (N col a y b)
  | x < y = balance col (insert x a) y b
  | x > y = balance col a y (insert x b)
  | otherwise = N col a y b
balance :: Colour → Tree → Nat → Tree → Tree
balance B (N R (N R a x b) y c) z d = N R (N B a x b) y (N B c z d)
balance B (N R a x (N R b y c)) z d = N R (N B a x b) y (N B c z d)
balance B a x (N R (N R b y c) z d) = N R (N B a x c) y (N B b z d)
balance B a x (N R b y (N R c z d)) = N R (N B a x b) y (N B c z d)
balance col a x b = N col a x b
```

Figure 6.6: A faulty implementation of an *insert* function. The error occurs on the third line of balance in which subtrees b and c are swapped.

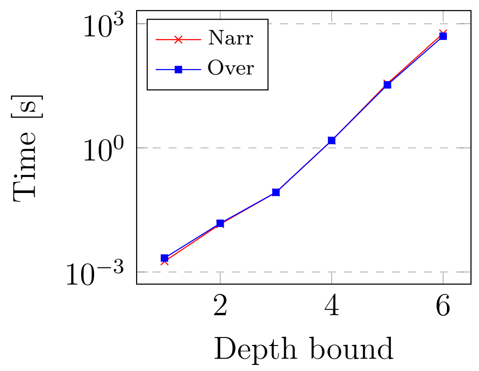
Strat	**Metric**	**2**	**3**	**4**	**5**	**6**
Narr	time	14.3ms	83.9ms	1.50s	35.7s	584s
	tests	48	260	3190	5.4E4*	6.8E5*
	invalid	63	291	4240	1.1E5	2.4E6
Over	time	15.0ms	84.1ms	1.51s	33.3s	499s
	tests	48	260	3190	5.4E4*	6.8E5*
	invalid	62	262	3316	5.8E4	7.2E5

* counter-example was found

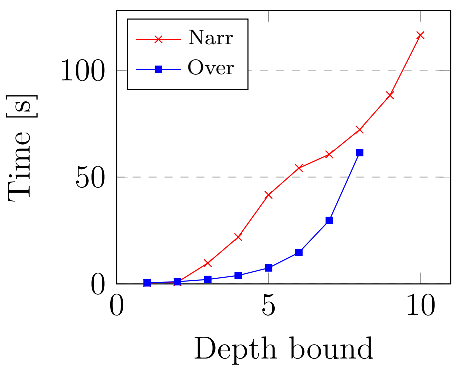
(a) Benchmark results for enumerating all red-black trees with a set number of nodes

Strat	**Metric**	**3**	**4**	**5**	**6**	**7**	**8**
Narr	time	9.76s	21.9s	41.7s	54.2s	60.7s	72.2s
	success	80.5%	63.5%	48.1%	43.8%	43.8%	43.6%
	size	2.06	1.24	0.51	0.32	0.32	0.32
	found	0%	62.5%	12.5%	0%	0%	0%
Over	time	2.09s	3.94s	7.48s	14.7s	29.7s	61.5s
	success	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%
	size	2.95	4.74	7.16	10.7	15.5	21.7
	found	0%	97.5%	97.5%	100%	100%	100%

(b) Benchmark results for testing 1000 red-black trees with a given depth of black nodes. The *found* field gives the percent of test runs in which a counter-example was found.



(c) Time taken to test the red-black tree property by enumeration



(d) Testing red-black tree property on 1000 randomly generated inputs

Figure 6.7: Benchmark results for testing the **red-black tree** property

$$\begin{aligned}
&\mathit{sizeLimit} :: \mathit{Nat} \to \mathit{Nat} \to \mathit{Nat} \to \mathit{Tree} \to \mathit{Result}\\
&\mathit{sizeLimit}\ n\ b\ a\ t\\
&\quad = \ (\mathit{countReds}\ t + 2\,\hat{}\ b - 1) \leqslant n\\
&\quad \wedge\ \ \mathit{all}\ (\leqslant)\ n\ t\\
&\quad \wedge\ \ a \leqslant n
\end{aligned}$$

In the first condition we calculate the number of elements in the tree by adding the current number of red nodes to the projected number of black nodes, $2\,\hat{}\, b - 1$. We define a narrowing version of the exponential function as:

$$\begin{aligned}
&(\hat{}\,) :: \mathit{Nat} \to \mathit{Nat} \to \mathit{Nat}\\
&\_\,\hat{}\ \texttt{Zero} = 1\\
&x\,\hat{}\ \texttt{Suc}\ \ n = x + x * \mathit{pred}\ (x\,\hat{}\ n)
\end{aligned}$$

Unlike a traditional definition of the exponential function, this version can reduce the expression $2\,\hat{}\ \texttt{Suc}\ \ n$ to $\texttt{Suc}\ (\texttt{Suc}\ (2 * \mathit{pred}\ (2\,\hat{}\ n))$, which encodes the fact that $2^{1+n} \geqslant 2$. For our purposes, this means that at all points in the evaluation, the projected count of black nodes will be equal to the number of black nodes required if the black node depth is not increased further.

Observation 11. The overlapping version performs better, enumerating all trees with of six or fewer nodes around 15% faster than the traditional version.

The performance difference is again explained by the differing number of test cases which need to be considered. In total, the overlapping version considers around 1.4 million cases whereas the narrowing version considers over 3 million. In contrast to the other case studies considered, the speedup

Strat	**Metric**	**2**	**3**	**4**	**5**
Narr	time	3.7ms	57.9ms	8.33s	432s*
	tests	17	133	5671	383*
	invalid	13	308	1.0E5	1.1E7*
Over	time	4.0ms	59.2ms	6.42s	0.27s*
	tests	17	133	5671	383*
	invalid	13	245	3.5E4	929*

* metrics for finding the first counter-example, as full enumeration was not completed

Figure 6.8: Benchmark results for enumerating red-black trees with the depth limit defined in section 4.3

of around 15% is substantially less than the over 50% reduction in the number of test cases. This is likely because the red-black constraint has many conditions and therefore traversing these conditions between each narrowing step will incur a significant performance penalty.

Observation 12. At depth five 480 failing test cases are found and at depth six 14,080 failing test cases are found.

Observation 13. Using the non-bespoke depth limit from section 4.3, the tool could find a counter-example at depth five but failed to complete the enumeration (Fig. 6.8).

This result seems to indicate that the use of a bespoke size limit was beneficial. With the bespoke size limit counterexamples were found at two depths in which the enumeration was completed, but without the bespoke limit we are only able to find a counter-example part way through an enumeration which doesn't finish in reasonable time. However, the bespoke size limit does take longer to find the first counter-example, taking over 10 seconds whereas using overlapping patterns with a traditional limit one was found in under half a second. This time difference is likely because the bespoke size limit has to consider trees with much larger elements and so each red-black configuration will have many more test cases (as this particular bug is predominantly caused by the colouring and shape of the tree this evaluation is wasted). It should also be noted that the time to find the first counter-example is not necessarily reliable, in the sense that it is likely to be highly dependent on the order of refinement and evaluation.

The results for the traditional size limit also show a far greater difference in performance between overlapping and narrowing evaluation. The first counter-example is found in under half a second with overlapping patterns, but takes over seven minutes without. This is likely because much bigger trees are considered at this depth in which co-dependent constraints have a much larger impact. This can be seen in the huge difference in invalid test cases considered, over 10 million without overlapping evaluation compared to around 1000 with.

Random

As in the ordered tree example, we give nodes a weight of two and leaves a weight of one. The normal backtrack limit of 30 is used. The size limit enforces a limit on the depth of the nodes in the tree but does not limit the size of elements:

$$\begin{aligned}
&\mathit{sizeLimit}\ n\ \_\ \_\ t = \mathit{maxDepth}\ t \leqslant n\\
&\mathit{maxDepth}\ L = \texttt{Zero}\\
&\mathit{maxDepth}\ (N\ \_\ t1\ \_\ t2) = \texttt{Suc}\ (\mathit{max}\ (\mathit{maxDepth}\ t1)\ (\mathit{maxDepth}\ t2))
\end{aligned}$$

The results can be found in the table 6.7b and the graph 6.7d. The size metric is the average number of nodes in the tree.

Observation 14. The traditional version is ineffective at random testing the red-black property. As the size limit increases the success rate of generating test cases and the average size of the trees generated declines.

This result is not surprising, as we saw previously that the traditional version fails to produce ordered trees effectively and this problem has the additional constraint of generating trees satisfying the red-black condition. The condition consists of two components, the *red* and *black* constraints, which are co-dependent as both constraints restrict the colours of the tree nodes. With the chosen constructor weightings, the empty tree and one node tree account for 40.7% of the test cases, which is almost all the 43.6% of the successful tests at size 8.

Observation 15. The overlapping version is effective at random testing, generating 1000 random test cases with average size of more than twenty nodes within the allocated time. The backtrack limit was never reached at any depth, and therefore the tool successfully generated a test case at every attempt.

When evaluated in an overlapping fashion the precondition will fail as soon as either the *red* or the *black* constraint becomes unsolvable. The amount of backtracking required is generally small and if we instead limit the depth of black nodes, $\mathit{sizeLimit}\ n\ b\ \_\ \_ = b \leqslant n$, a backtrack limit of 3 will always suffice (if we are at the limit of black nodes we might have to backtrack over the black node variable, the colour of the node and then we will always be able to replace the node with a leaf). We chose the actual limit in order to get greater granularity in the size limit and give more detailed results.

Observation 16. From depth four onward, the tool consistently finds the bug in the program. At depth 4 an average test run completes within 4 seconds.

Random testing is arguably more effective than enumerative testing in this case study as it generally finds a counter-example much faster and can do so at a depth limit which is much less than the maximum limit completed, suggesting that random testing could comfortably find an even rarer bug. Whereas enumerative testing generally has an exponentially increasing search space and ends up testing many similar test cases, random testing only samples from the increasing search space and typically has few similar test cases[1]. The exponential nature of enumerative testing also impacts the ease of testing. In this study we used a carefully designed bespoke limit for enumerative testing to try and minimise the exponential increase in search space but random testing is effective with a simple depth limit.

6.3.9 Discussion

We review the impact of overlapping patterns on performance before discussing the overall experience of testing with narrowing and overlapping patterns.

Performance

Overall, we found overlapping patterns to improve the performance of property-based testing. On four of the six case studies we looked at, the overlapping version was substantially faster at evaluating test cases. On the other two there was little performance difference between the two with

[1]Random testing does repeat trivial test cases many times, however these test cases account for only a negligible amount of computation time and, if desired, the repetition could easily be avoided by remembering which small test cases have been performed.

only a very minor performance penalty for overlapping patterns in some cases.

We found that overlapping patterns had the biggest impact in random testing. In all five property-based testing case studies we were able to test much larger test cases by random testing than by enumeration. In contrast, the traditional method was only able to consistently generate larger test cases in two of the case studies and was completely ineffective for the remaining three (sorted permutations, ordered trees, and red-black trees).

We have used bespoke size limits extensively in the case studies, and for some of the case studies they reduce the need for backtracking (ordered trees, union and red-black trees). More research is needed to establish whether they have a significant impact on the effectiveness of testing. This type of comparison is harder to evaluate as changing the size limit impacts the distribution, and therefore the comparison cannot be made based on time taken. Other metrics, such as the ability to find counter-examples or code coverage would have to be used.

Experience

The use of overlapping patterns allows us to write property with a precondition formed of a conjunction of constraints with a reasonable expectation that the property can be tested effectively. Furthermore, we can use them to define a size constraint which is tailored to the data-type being generated – not only giving the user greater control over the distributions of tests, but also helping to eliminate backtracking and therefore making testing more effective. It should be noted that while we have sometimes resorted to quite complex size limits in order to eliminate backtracking, for example the size limit in the red-black property, a simple size limit is generally sufficient to achieve good results. For enumerative testing, Duregaard [1] has shown that limiting the number of constructors is effective, and for random testing limiting the depth of the spine of a data-type generally works well.

As we noted in the previous chapter, narrowing can lead to subtle performance bugs. This is apparent in red-black property where we used both an additional narrowing variable to restrain the black nodes, and a non-standard exponent function. Whereas the exponent function was only used in a size constraint and not strictly necessary, it is difficult to write a version of the black constraint with efficient semantics without using an additional narrowing variable. These optimisations are likely to be surprising to a functional programmer who is not familiar with narrowing, and so experience is needed to be fully proficient at property-based testing with narrowing. However, experience is also needed in order to write bespoke generators à la QuickCheck, and the combination of overlapping patterns and narrowing will work well on a large proportion of properties immediately. A programmer can always try this as a low effort approach, and fallback on a different approach if it fails.

6.4 Related Work

In this section we review related work, discussing techniques related to overlapping patterns.

Parallel Conjunction Several property-based testing tools have a special parallel conjunction operator which is equivalent to the overlapping definition of conjunction [33, 48, 15]. The implementation used by these tools utilises the Glasgow Haskell Compiler's exception handling, which necessitates re-evaluating the program after every refinement. In comparison, we define a semantics which both covers the more general concept of overlapping patterns and also stores the progress made in the evaluation.

Residuation Residuation is an alternative strategy for the evaluation of functional-logic languages used in many implementations [34, 35, 49, 28]. In a similar manner to our semantics, expressions are evaluated deterministically and suspended when a variable is required to continue. Expressions are combined with an operator synonymous with parallel conjunction. Unlike our semantics, refinements or instantiations can only be made by the use of explicit predicates. This has the advantage of giving the programmer greater control but has two significant disadvantages: a refinement may happen on a variable which is not currently impeding evaluation; and the process is incomplete, in the sense that it is unable to compute solutions if insufficient instantiations are made.

6.5 Conclusion and Further Work

In this chapter, we developed and evaluated our property-based testing tool extending it with an implementation of overlapping patterns. We concluded that the use of overlapping patterns is beneficial in the testing of properties and allows a greater variety of properties to be tested effectively in an automated fashion. This benefit stems from two advantages: the ability to evaluate a combination of constraints effectively using overlapping conjunction, and the use of bespoke size limits. There are many ways this research could be extended, a few of which we discuss below.

The combination of overlapping patterns and narrowing requires a novel evaluation strategy as every branch of a term has to be reduced, which generally necessitates traversing the entire term, before a narrowing step can be undertaken. There are many interesting avenues to explore to make this process more efficient, such as the compilation of programs or exploring different novel evaluation strategies. For example, we could consider beginning evaluation from the leaves of an expression, which has the potential advantage of avoiding traversing the expression tree each time a narrowing step is made.

Measuring program coverage is a natural way of judging the effectiveness of testing, and using narrowing to help improve the code coverage of testing has been explored [38, 23]. However, the application of heuristics to direct evaluation is often hampered by the need to resolve a precondition before the program being scrutinised is evaluated. Using overlapping patterns we can easily avert this problem, simply by changing the evaluation order of implication in the following way:

```
{-# OVERLAP ( ⟸ ) #-}
( ⟸ ) :: Result → Bool → Result
Success  ⟸ True   = Success
Failure  ⟸ True   = Failure
_        ⟸ False  = Invalid
```

Such a definition allows the tested program to be evaluated immediately, and therefore could allow for more sophisticated heuristics.

Chapter 7

Conclusion

In this final chapter we draw some conclusions on the work of the thesis. In particular, we provide a retrospective summary of the main achievements of each chapter, reflect on our original objectives in light of what we have achieved, and discuss some potential avenues for further work.

7.1 Summary

- In chapter 3 we established a theory of narrowing as an extension to a functional programming language. The main result of the chapter was a soundness and completeness theorem that related a narrowing semantics to a functional semantics for a minimal language.

- In chapter 4 we developed a narrowing tool for the purpose of property-based testing. The tool was evaluated against a basic tool, which does not utilise narrowing, and confirmed previous research that narrowing improves the performance on certain property-based testing problems. We tested two different narrowing evaluations, with and without shared evaluation, allowing us to apportion the performance benefits of narrowing between shared evaluation and wide evaluation.

- In chapter 5 we expanded our formalisation to include the notion of overlapping patterns. Overlapping patterns allow evaluation of multiple branches of a program simultaneously, potentially deriving a result with fewer instantiated free variables. We showed two advantages of this extension for property-based testing: allowing effective combining of constraints and allowing for the definition of bespoke size constraints.

- In chapter 6 we extended our narrowing tool to incorporate overlapping patterns. We benchmarked the tool, using the original narrowing tool as comparison, and saw that overlapping patterns extend the range of properties that can be effectively tested in an automatic fashion.

7.2 Reflection

The starting point for the research undertaken in this thesis was the following proposition: that narrowing is a useful tool for property-based testing in functional programming languages such as Haskell.

To validate this proposition, we have:

- *Developed a theory of narrowing for functional languages.* Previous work in this area has focused on practical issues concerned with adding narrowing to a functional language. To the best of our knowledge, this thesis presents the first supporting theory for such an extension. Rather than developing the theory from scratch, we built upon existing theories of functional programming in order to reuse ideas and results, which simplified our development.

- *Produced a theory of overlapping patterns for functional languages.* This work formalises and generalises the notion of parallel conjunction, which has previously been used to improve the performance of several testing tools. The use of overlapping patterns in a narrowing evaluation can delay the instantiation of free variables, which can bring significant performance benefits in property-based testing.

- *Implemented a property-based testing system based on our theories.* We developed a prototype implementation in Haskell, which realises all of the ideas from the thesis. The system implements a narrowing evaluation strategy, supports overlapping patterns, and provides functionality for both enumerative and random testing of properties.

- *Demonstrated the practical utility of our system on a range of examples.* We considered some classic examples from the literature on property-based testing, and showed how the combination of the use of narrowing and overlapping patterns can both improve performance, and expand the range of properties that can be effectively tested in an automatic manner without the use of a custom generator. Other properties could be tested effectively with alterations, such as adding a bespoke size limit or introducing narrowing variables, which reduce the amount of backtracking.

7.3 Further work

Based on the results of our research during the last four years, we feel that the use of narrowing is beneficial for property-based testing and is an interesting avenue for future research. At the end of each chapter, we suggested potential areas for future research specific to the chapter in question. Here we summarise a few of these potential direction of research:

- *Investigating the impact of narrowing on space usage.* The main benchmark which we have used in this thesis is the time taken to complete a task, however it would also be interesting to investigate the space usage. There is often a trade off between these two performance measures, and we expect to find this is true for the techniques of evaluation sharing and overlapping patterns, both of which generally speed up evaluation but have additional space requirements.

- *Eliminating backtracking* To use narrowing effectively for testing we found it is necessary to minimise or eliminate backtracking. Techniques such as overlapping patterns help achieve this goal, however they come with no guarantee. This motivates two potential areas of research. Firstly, investigating effective means of proving when a narrowing evaluation will not backtrack. And secondly, designing narrowing languages in which the lack of backtracking is guaranteed by construction.

- *Implementing a compiled version of narrowing.* The direct compilation of narrowing to machine-code or a low level intermediate representation is relatively new topic in functional-logic languages, with most implementations either compiling to a different high-level language or running in abstract machines (as in this thesis). It would be interesting to see whether our approach to formalising narrowing as an extension to a functional language could also be applied to compilation by building a narrowing compiler as an extension to a compiler for a functional language.

Appendix A

Case study code

This appendix includes the code for the case studies in chapters 4 and 6. For brevity the code provided here is presented in Haskell, the actual code consumed by our tool is a subset of Haskell, without some of the syntactic sugar, and can be found on Github [25].

A.1 Naturals

The code for naturals which are used in many of the examples.

module *Nat* **where**

import *Prelude hiding* ((^))

data *Nat* = `Zero` | `Suc` *Nat* **deriving** *Show*

$(\hat{}) :: Nat \rightarrow Nat \rightarrow Nat$

$\_\,\hat{}\; \texttt{Zero} = 1$

$a\,\hat{}\,(\texttt{Suc}\ x) = a + (a * pred\ (a\,\hat{}\,x))$

$lengthNat :: [a] \rightarrow Nat$

$lengthNat = \mathbf{foldr}_{(const\ \texttt{Suc}),\texttt{Zero}}\ .$

```
instance Enum Nat where
  toEnum = fromIntegral
  fromEnum Zero = 0
  fromEnum (Suc n) = 1 + fromEnum n
instance Eq Nat where
  Zero ≡ Zero = True
  Zero ≡ Suc _ = False
  Suc _ ≡ Zero = False
  Suc x ≡ Suc y = x ≡ y
instance Ord Nat where
  Zero ⩽ _ = True
  Suc _ ⩽ Zero = False
  Suc x ⩽ Suc y = x ⩽ y
  Zero >_ = False
  Suc x > Suc y = x > y
  Suc _ > Zero = True
  max Zero y = y
  max x Zero = x
  max (Suc x) y = Suc (max x (pred y))
instance Num Nat where
  Zero +y = y
  Suc x + y = Suc (x + y)
  x − Zero = x
  Zero −_ = Zero
  Suc x − Suc y = x − y
  Zero ∗_ = Zero
```

```
    (Suc x) * y = y + (x * y)
    fromInteger 0 = Zero
    fromInteger n = Suc (fromInteger (n - 1))
    abs n = n
    signum _ = Suc Zero
```

A.2 Union of Sets

The union of sets case study, sections 4.4.4 and 6.3.4.

```
module Union where
import Nat
import Prelude
import Property
 {-# Dist [] 1 #-}
 {-# Dist (:) 5 #-}
check :: [Nat] → [Nat] → Result
check l l′ = set l ∧ set l′ ⟹ set (union l l′)
set :: [Nat] → Bool
set [ ] = True
set (a : l) = go a l
  where
  go _ [ ] = True
  go b (c : l′) = (b < c) ∧ go c l′
union :: [Nat] → [Nat] → [Nat]
union [ ] l = l
union l [ ] = l
union (a : l) (a' : l')
    | a < a' = a : union l (a' : l')
    | otherwise = a' : union (a : l) l'
```

A.3 Reverse

The reverse case study, section 4.4.5.

```
module Reverse where

import Prelude hiding (Char)
import Property

 {-# DIST [] 1 #-}
 {-# DIST (:) 5 #-}

data Char = U | V | W | X | Y deriving (Eq, Enum, Ord, Show)

checkBasic :: [Char] → [Char] → Result
checkBasic l l' = post $ reverse (l ++ l') ≡ (reverse l' ++ reverse l)
```

A.4 Huffman Compression

The Huffman compression case study, section 4.4.7.

```
module Huffman where

import Data.List
import Data.Maybe
import Nat
import Property
import Prelude hiding (Char)

data Tree = Leaf Char | Fork Tree Tree deriving (Eq, Ord, Show)

data Char = U | V | W | X | Y deriving (Eq, Enum, Ord, Show)

 {-# DIST [] 1 #-}

 {-# DIST (:) 5 #-}

checkBasic :: [Char] → Result

checkBasic l = ¬ (null l) ⟹ l ≡ encodeDecode l

encodeDecode :: [Char] → [Char]

encodeDecode l = let t = huffTree l in decode t (encode t l)

decode :: Tree → [Bool] → [Char]

decode _ [] = []

decode t bs = dec t bs

  where

  dec (Leaf x) bs′ = x : decode t bs′

  dec (Fork t′ _) (False : bs′) = dec t′ bs′

  dec (Fork _ t′) (True : bs′) = dec t′ bs′

encode :: Tree → [Char] → [Bool]

encode t =

  let table = codeTable t

  in concatMap (fromJust ∘ (‘lookup‘ table))

collate :: [Char] → [(Nat, Tree)]

collate [] = []

collate (c : cs) = let

  (n, cs′) = countRemove c (c : cs)

  in insert (n, Leaf c) (collate cs′)

countRemove :: Char → [Char] → (Nat, [Char])
```

```
$countRemove\ \_\ [\,] = (0, [\,])$
$countRemove\ x\ (y : ys) = \mathbf{let}$
  $(count, rest) = countRemove\ x\ ys$
  $\mathbf{in\ if}\ x \equiv y\ \mathbf{then}\ (count + 1, rest)\ \mathbf{else}\ (count, y : rest)$
$huffTree :: [Char] \to Tree$
$huffTree\ cs = mkHuff\ (collate\ cs)$
$mkHuff :: [(Nat, Tree)] \to Tree$
$mkHuff\ [(\_, t)] = Fork\ t\ (Leaf\ Y)$   -- Tree must have two elements
$mkHuff\ l = go\ l$
  $\mathbf{where}$
  $go\ [(\_, t)] = t$
  $go\ ((n0, t0) : (n1, t1) : wts) =$
    $go\ (insert\ (n0 + n1, Fork\ t0\ t1)\ wts)$
$codeTable :: Tree \to [(Char, [Bool])]$
$codeTable\ t = go\ [\,]\ t$
  $\mathbf{where}$
  $go\ p\ (Leaf\ x) = [(x, p)]$
  $go\ p\ (Fork\ xt\ yt) = go\ (p +\!\!+ [\texttt{False}])\ xt +\!\!+ go\ (p +\!\!+ [\texttt{True}])\ yt$
```

A.5 Overlapping Prelude

The overlapping “Prelude” replaces logical operators with their overlapping counterparts.

```
$\mathbf{module}\ OverlapPrelude$
  $(\mathbf{module}\ Prelude$
  $, (\wedge)$
```

```
   , (∨)
   ) where
import Prelude hiding (( ∧ ), (∨))
{-# OVERLAP ( ∧ ) #-}
( ∧ ) :: Bool → Bool → Bool
False ∧ _ = False
_ ∧ False = False
True ∧ y = y
x ∧ True = x
{-# OVERLAP (∨) #-}
(∨) :: Bool → Bool → Bool
False ∨ y = y
x ∨ False = x
True ∨ _ = True
_ ∨ True = True
```

A.6 Overlapping Naturals

An overlapping implementations of the natural module.

```
module OverlapNat where
import Prelude hiding ((ˆ))
data Nat = Zero | Suc Nat deriving Show
(ˆ) :: Nat → Nat → Nat
_ˆ Zero = 1
aˆ(Suc x) = a + (a ∗ pred (aˆx))
instance Enum Nat where
```

```
    toEnum = fromIntegral
    fromEnum Zero = 0
    fromEnum (Suc n) = 1 + fromEnum n
instance Eq Nat where
    Zero ≡ Zero = True
    Zero ≡ Suc _ = False
    Suc _ ≡ Zero = False
    Suc x ≡ Suc y = x ≡ y
instance Ord Nat where
    Zero ⩽ _ = True
    Suc _ ⩽ Zero = False
    Suc x ⩽ Suc y = x ⩽ y
    Zero >_ = False
    Suc x > Suc y = x > y
    Suc _ > Zero = True
     {-# OVERLAP max #-}
    max Zero y = y
    max x Zero = x
    max (Suc x) y = Suc (max x (pred y))
    max x (Suc y) = Suc (max (pred x) y)
instance Num Nat where
     {-# OVERLAP (+) #-}
    Zero +y = y
    Suc x + y = Suc (x + y)
    x + Zero = x
    x + Suc y = Suc (x + y)
```

$x - \texttt{Zero} = x$

$\texttt{Zero} - \_ = \texttt{Zero}$

$\texttt{Suc}\ x - \texttt{Suc}\ y = x - y$

$\texttt{Zero} * \_ = \texttt{Zero}$

$(\texttt{Suc}\ x) * y = y + (x * y)$

$\mathit{fromInteger}\ 0 = \texttt{Zero}$

$\mathit{fromInteger}\ n = \texttt{Suc}\ (\mathit{fromInteger}\ (n - 1))$

$\mathit{abs}\ n = n$

$\mathit{signum}\ \_ = \texttt{Suc Zero}$

A.7 Sorted Permutations

The permutation case study, section 6.3.3. For this and following examples the overlapping versions can be used by providing the $-DOVERLAP$ flag. This swaps the ‘Nat‘ and ‘Prelude‘ libraries for their overlapping counterparts.

```
{-# LANGUAGE CPP #-}
{-# LANGUAGE NoImplicitPrelude #-}
module Perm where
#ifdef OVERLAP
import OverlapPrelude hiding ((^))
import OverlapNat
#else
import Prelude hiding ((^))
import Nat
#endif
```

```
import Data.List
import Property
```

$check :: Nat \to [Nat] \to Result$

$check\ n\ l = perm\ n\ l \implies [0\mathinner{\ldotp\ldotp} n-1] \equiv sort\ l$

$perm :: Nat \to [Nat] \to Bool$

$perm\ n\ l = (n \equiv lengthNat\ l) \ \wedge\ all\ ({<}n)\ l \ \wedge\ allDiff\ l$

$allDiff :: [Nat] \to Bool$

$allDiff\ [\,] = \texttt{True}$

$allDiff\ (n : l) = notElem\ n\ l \ \wedge\ allDiff\ l$

A.8 Ordered Trees

The ordered tree case study, sections 4.4.6 and 6.3.5. Three properties are provided *check* for the results in chapter 4, *checkEnum* with a size limit for enumerative testing and *checkRand* with a size limit for random testing. The bespoke size limits are defined in the *OrderedTreeType* module which always uses overlapping functions.

```
{-# LANGUAGE CPP #-}
{-# LANGUAGE NoImplicitPrelude #-}
{-# LANGUAGE DeriveFunctor #-}
{-# LANGUAGE DeriveFoldable #-}
module OrderedTreeType where
import OverlapPrelude hiding ((^))
import OverlapNat
data Tree a = Leaf | Node (Tree a) a (Tree a)
  deriving (Eq, Show, Functor, Foldable)
```

```
{-# DIST Leaf 1 #-}
{-# DIST Node 2 #-}
countTree :: Tree a → Nat
countTree Leaf = Zero
countTree (Node t1 _ t2) = Suc (countTree t1 + countTree t2)
depthTree :: Tree a → Nat
depthTree Leaf = Zero
depthTree (Node t1 _ t2) = Suc (max (depthTree t1) (depthTree t2))
depthElem :: Tree Nat → Nat
depthElem Leaf = Zero
depthElem (Node t1 a t2) =
  maximum [a, depthElem t1, depthElem t2]
enumSize :: Nat → Tree Nat → Bool
enumSize i t = (countTree t ⩽ i) ∧ all (⩽ 4) t
```

```
{-# LANGUAGE CPP #-}
{-# LANGUAGE NoImplicitPrelude #-}
module OrderedTree where
#ifdef OVERLAP
import OverlapPrelude hiding ((ˆ))
import OverlapNat
#else
import Prelude hiding ((ˆ))
import Nat
#endif
```

```
import OrderedTreeType

import Property

check :: Nat → Tree Nat → Result

check n t = ordered t ⟹ ordered (del n t)

checkRand :: Nat → Nat → Tree Nat → Result

checkRand i n t = check n t ‘suchThat‘ depthTree t ⩽ i

checkEnum :: Nat → Tree Nat → Result

checkEnum i t

  = check 1 t ‘suchThat‘ enumSize i t

del :: Ord a ⇒ a → Tree a → Tree a

del _ Leaf = Leaf

del n (Node t1 a t2)

  | a < n = Node t1 a (del n t2)

  | n > a = Node (del n t1) a t2

  | otherwise = ext t1 t2

  where

  ext Leaf t = t

  ext (Node t11 b t12) t = Node t11 b (ext t12 t)

ordered :: Ord a ⇒ Tree a → Bool

ordered Leaf = True

ordered (Node t1 a t2)

  =  all (⩽ a) t1  ∧  ordered t1

   ∧  all (⩾ a) t2  ∧  ordered t2
```

A.9 Well-Typed Expressions

The well-typed expression case study, section 6.3.6

```
{-# NoImplicitPrelude #-}

module ExprType where

import OverlapNat

data Expr
  = Add Expr Expr
  | If Expr Expr Expr
  | Natural Nat
  | Boolean Bool

countExpr :: Expr → Nat
countExpr (Natural v) = v
countExpr (Boolean _) = Zero
countExpr (If e e′ e″) =
  Suc (countExpr e + countExpr e′ + countExpr e″)
countExpr (Add e e′) = Suc (countExpr e + countExpr e′)

depthExpr :: Expr → Nat
depthExpr (Natural v) = v
depthExpr (Boolean _) = Zero
depthExpr (If e e′ e″) = Suc $
  maximum [depthExpr e, depthExpr e′, depthExpr e″]
depthExpr (Add e e′) = Suc (max (depthExpr e) (depthExpr e′))
```

```
{-# LANGUAGE CPP #-}
{-# LANGUAGE NoImplicitPrelude #-}
```

```
module Expr where

#ifdef OVERLAP
import OverlapPrelude hiding ((^))
import OverlapNat
#else
import Prelude hiding ((^))
import Nat
#endif

import ExprType
import Property
import Control.Monad
import Data.Maybe

data Type = Nat | Bool | NoType

check :: Type → Expr → Result
check t e = hasType e t ⟹ isJust (evalExpr e)

checkRand :: Nat → Type → Expr → Result
checkRand n t e
  = check t e `suchThat` depthExpr e ⩽ n

checkEnum :: Nat → Type → Expr → Result
checkEnum n t e
  = check t e `suchThat` countExpr e ⩽ n

hasType :: Expr → Type → Bool
hasType (Natural _) Nat = True
hasType (Boolean _) Bool = True
hasType (If e e′ e″) t =
  hasType e Bool ∧ hasType e′ t ∧ hasType e″ t
hasType (Add e e') Nat = hasType e Nat ∧ hasType e' Nat
hasType _ _ = False

evalExpr :: Expr → Maybe Expr
evalExpr (Natural n) = Just $ Natural n
evalExpr (Boolean b) = Just $ Boolean b
evalExpr (Add e e') = join $
  evalAdd <$> evalExpr e <*> evalExpr e'
evalExpr (If e e' e'') = join $
  evalIf <$> evalExpr e <*> evalExpr e' <*> evalExpr e''

evalAdd :: Expr → Expr → Maybe Expr
evalAdd (Natural n) (Natural m) = Just $ Natural (n + m)
evalAdd _ _ = Nothing

evalIf :: Expr → Expr → Expr → Maybe Expr
evalIf (Boolean True) p _ = Just p
evalIf (Boolean False) _ q = Just q
evalIf _ _ _ = Nothing
```

A.10 N-Queens Constraint Problem

The n-queens constraint problem, section 6.3.7

```
 {-# LANGUAGE CPP #-}
 {-# LANGUAGE NoImplicitPrelude #-}

module NQueens where

#ifdef OVERLAP
import OverlapPrelude hiding ((ˆ))
import OverlapNat
```

```
#else
import Prelude hiding ((^))
import Nat
#endif
import Property
nQueens :: Nat → [Nat] → Result
nQueens n l
  = (n ≡ lengthNat l)
   ∧ all (<n) l
   ∧ allDiff l
   ∧ checkDiagonals (map (n−) l)
   ∧ checkDiagonals l ⟹ True
checkDiagonals :: [Nat] → Bool
checkDiagonals [] = True
checkDiagonals (n : l) = checkDiag n l ∧ checkDiagonals l
  where
  checkDiag _ [] = True
  checkDiag Zero _ = True
  checkDiag (Suc n′) (a : l′) = (n′ ≢ a) ∧ checkDiag n′ l′
allDiff :: [Nat] → Bool
allDiff [] = True
allDiff (n : l) = notElem n l ∧ allDiff l
```

A.11 Red-Black Trees

The red-black tree case study, section 6.3.8

```
 {-# LANGUAGE CPP #-}
 {-# LANGUAGE NoImplicitPrelude #-}
 {-# LANGUAGE DeriveFunctor #-}
 {-# LANGUAGE DeriveFoldable #-}
module RedBlackType where
import OverlapPrelude hiding ((ˆ))
import OverlapNat
data Colour = R | B
  deriving (Eq, Show)
data Tree a
   = L | N Colour (Tree a) a (Tree a)
  deriving (Eq, Foldable, Functor, Show)
type SizeNat = Nat
 {-# DIST L 1 #-}
 {-# DIST N 2 #-}
maxElem :: Tree Nat → Nat
maxElem L = Zero
maxElem (N _ t0 a t1) = maximum [a, maxElem t0, maxElem t1]
enumSize :: Nat → Nat → Nat → Tree Nat → Bool
enumSize n k a t
   = (countReds t + pred (2ˆk) ⩽ n)
   ∧ all (⩽ n) t
   ∧ (a ⩽ n)
countReds :: Tree a → Nat
countReds (N R t1 _ t2) = 1 + countReds t1 + countReds t2
countReds (N B t1 _ t2) = countReds t1 + countReds t2
countReds L = Zero

treeDepth :: Tree a → Nat
treeDepth L = Zero
treeDepth (N _ t1 _ t2) = Suc (max (treeDepth t1) (treeDepth t2))
```

```
{-# LANGUAGE CPP #-}
{-# LANGUAGE NoImplicitPrelude #-}
module RedBlack where
#ifdef OVERLAP
import OverlapPrelude hiding ((^))
import OverlapNat
#else
import Prelude hiding ((^))
import Nat
#endif
import Property
import RedBlackType

checkRand :: Nat → Nat → Nat → Tree Nat → Result
checkRand n k a t
  = redBlackN k t ⟹ redBlack (insert a t)
  `suchThat` treeDepth t ⩽ n

checkEnum :: Nat → Nat → Nat → Tree Nat → Result
checkEnum n k a t
  = redBlackN k t ⟹ redBlack (insert a t)
  `suchThat` enumSize n k a t
```

```
redBlackN :: Ord a ⇒ Nat → Tree a → Bool
redBlackN k t = blackRoot t ∧ blackN t k ∧ red t ∧ ord t

redBlack :: Ord a ⇒ Tree a → Bool
redBlack t = blackRoot t ∧ black t ∧ red t ∧ ord t

insert :: Ord a ⇒ a → Tree a → Tree a
insert x s = makeBlack (ins s)
  where
  ins L = N R L x L
  ins (N col a y b)
    | x < y = balance col (ins a) y b
    | x > y = balance col a y (ins b)
    | otherwise = N col a y b
  makeBlack (N _ a y b) = N B a y b

balance :: Colour → Tree a → a → Tree a → Tree a
balance B (N R (N R a x b) y c) z d = N R (N B a x b) y (N B c z d)
balance B (N R a x (N R b y c)) z d = N R (N B a x b) y (N B c z d)
balance B a x (N R (N R c y b) z d) = N R (N B a x b) y (N B c z d)
balance B a x (N R b y (N R c z d)) = N R (N B a x b) y (N B c z d)
balance col a x b = N col a x b

blackRoot :: Tree a → Bool
blackRoot L = True
blackRoot (N B _ _ _) = True
blackRoot _ = False

  -- INVARIANT 1. No red node has a red parent.
red :: Tree a → Bool
red L = True
red (N col a _ b) = (¬ (isRed col) ∨ (blackRoot a ∧ blackRoot b))
     ∧ red a ∧ red b
isRed :: Colour → Bool
isRed R = True
isRed B = False
  -- INVARIANT 2. Every path from the root to an empty node
  -- contains the same number of black nodes.
black :: Tree a → Bool
black = fst ∘ go
  where
  go L = (True, Zero)
  go (N c t1 _ t2) = let
    (b1, d1) = go t1
    (b2, d2) = go t2
    in (b1 ∧ b2 ∧ (d1 ≡ d2)
      , if isRed c then max d1 d2 else Suc (max d1 d2)
      )
  -- Is a fixed black depth
blackN :: Tree a → Nat → Bool
blackN L Zero = True
blackN (N R t1 _ t2) n = blackN t1 n ∧ blackN t2 n
blackN (N B t1 _ t2) (Suc n) = blackN t1 n ∧ blackN t2 n
blackN _ _ = False
  -- INVARIANT 3. Trees are ordered.
ord :: Ord a ⇒ Tree a → Bool
ord L = True
```

$$\mathit{ord}\ (N\ \_\ \mathit{t0}\ \ a\ \mathit{t1}) = \mathit{all}\ (\leqslant a)\ \mathit{t0}\ \ \wedge\ \ \mathit{all}\ (\geqslant a)\ \mathit{t1}\ \ \wedge\ \ \mathit{ord}\ \mathit{t0}\ \ \wedge\ \ \mathit{ord}\ \mathit{t1}$$

Bibliography

[1] PhD thesis.

[2] Elvira Albert, Michael Hanus, Huch Frank, Javier Oliver, and Vidal Germán. Operational semantics for declarative multi-paradigm languages. *Journal of Symbolic Computation*, 40(1), 2005.

[3] Sergio Antoy. Constructor-based conditional narrowing. In *Proceedings of the 3rd ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 199–206. ACM, 2001.

[4] Sergio Antoy, Rachid Echahed, and Michael Hanus. A needed narrowing strategy. *Journal of the ACM*, 47(4), 2000.

[5] Sergio Antoy and Michael Hanus. Overlapping rules and logic variables in functional logic programs. In *International Conference on Logic Programming*, pages 87–101. Springer, 2006.

[6] Sergio Antoy and Michael Hanus. Functional logic programming. *Communications of the ACM*, 53(4):74–85, 2010.

[7] Sergio Antoy and Andy Jost. A new functional-logic compiler for Curry: Sprite. In *International Symposium on Logic-Based Program Synthesis and Transformation*, pages 97–113. Springer, 2016.

[8] Richard Bird. *Introduction to Functional Programming using Haskell.* Prentice Hall Series in Computer Science. Prentice Hall, 1998.

[9] Rudy Braquehais and Colin Runciman. Fitspec: refining property sets for functional testing. In *Proceedings of the 9th International Symposium on Haskell*, pages 1–12. ACM, 2016.

[10] Rudy Braquehais and Colin Runciman. Speculate: discovering conditional equations and inequalities about black-box functions by reasoning from test results. In *Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell*, pages 40–51. ACM, 2017.

[11] Bernd Braßel, Michael Hanus, Björn Peemöller, and Fabian Reck. Kics2: A new compiler from Curry to Haskell. In *International Workshop on Functional and Constraint Logic Programming*, pages 1–18. Springer, 2011.

[12] Lukas Bulwahn. The new quickcheck for Isabelle. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, volume 12, pages 92–108. Springer, 2012.

[13] Jan Christiansen and Sebastian Fischer. Easycheck – test data for free. In *International Symposium on Functional and Logic Programming*, pages 322–336. Springer, 2008.

[14] Koen Claessen. Shrinking and showing functions: (functional pearl). In *Proceedings of the 2012 Haskell Symposium*, pages 73–80. ACM, 2012.

[15] Koen Claessen, Jonas Duregård, and Michał H Pałka. Generating constrained random data with uniform distribution. In *International Sym-*

posium on Functional and Logic Programming, pages 18–34. Springer, 2014.

[16] 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*, pages 268–279, 2000.

[17] Koen Claessen, Moa Johansson, Dan Rosén, and Nicholas Smallbone. Automating inductive proofs using theory exploration. In *International Conference on Automated Deduction*, pages 392–406. Springer, 2013.

[18] Koen Claessen, Nicholas Smallbone, and John Hughes. QuickSpec: Guessing formal specifications using testing. In *International Conference on Tests and Proofs*, pages 6–21. Springer, 2010.

[19] Jesper Cockx. *Overlapping and Order-Independent Patterns in Type Theory*. PhD thesis, Master thesis, KU Leuven, 2013.

[20] Jesper Cockx, Frank Piessens, and Dominique Devriese. Overlapping and order-independent patterns. In *European Symposium on Programming Languages and Systems*, pages 87–106. Springer, 2014.

[21] Jonas Duregård, Patrik Jansson, and Meng Wang. Feat: Functional Enumeration of Algebraic Types. *Proceedings of the 2012 Haskell Symposium*, pages 61–72, 2012.

[22] Matthias Felleisen and Robert Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical computer science*, 103(2):235–271, 1992.

[23] Sebastian Fischer and Herbert Kuchen. Data-flow testing of declarative programs. In *Proceedings of the 13th ACM SIGPLAN international conference on Functional programming*, pages 201–212. ACM, 2008.

[24] Jonathan Fowler. Towards a Theory of Reach - Agda Proof, 2015. Available at: https://github.com/JonFowler/theoryofreach.

[25] Jonathan Fowler. The NarrowCheck system for property-based testing, 2016. Available at: https://github.com/JonFowler/NarrowCheck.

[26] Jonathan Fowler and Graham Huttom. Towards a theory of Reach. In *International Symposium on Trends in Functional Programming*, pages 22–39. Springer, 2015.

[27] Jonathan Fowler and Graham Hutton. Failing faster: overlapping patterns for property-based testing. In *International Symposium on Practical Aspects of Declarative Languages*, pages 103–119. Springer, 2017.

[28] Michael Hanus. A unified computation model for functional and logic programming. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 80–93. ACM, 1997.

[29] Michael Hanus et al. Curry - an integrated functional logic language. Technical report, 2016. Version 0.9.0.

[30] John Hughes. Quickcheck testing for fun and profit. In *International Symposium on Practical Aspects of Declarative Languages*, pages 1–32. Springer, 2007.

[31] Gilles Kahn. Natural semantics. In *Annual Symposium on Theoretical Aspects of Computer Science*, pages 22–39. Springer, 1987.

[32] Leonidas Lampropoulos, Diane Gallois-Wong, Cătălin Hriţcu, John Hughes, Benjamin C Pierce, and Li-yao Xia. Beginner's luck: a language for property-based generators. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 114–129. ACM, 2017.

[33] Fredrik Lindblad. Property directed generation of first-order test data. In *Trends in Functional Programming*, pages 105–123. Citeseer, 2007.

[34] John W Lloyd. Combining functional and logic programming languages. In *Proceedings of the 1994 International Symposium on Logic programming*, pages 43–57. Mit Press, 1994.

[35] John W Lloyd. Declarative programming in Escher. Technical report, 1995.

[36] Simon Marlow, Simon Peyton Jones, et al. The Glasgow Haskell Compiler. Technical report, 2004.

[37] Conor McBride and Ross Paterson. Applicative programming with effects. *Journal of functional programming*, 18(1):1–13, 2008.

[38] Matthew Naylor and Colin Runciman. Finding inputs that Reach a target expression. In *International Conference on Source Code Analysis and Manipulation*, pages 133–142. IEEE, 2007.

[39] Matthew Francis Naylor. *Hardware-Assisted and Target-Directed Evaluation of Functional Programs.* PhD thesis, University of York, 2008.

[40] Tobias Nipkow. More Church–Rosser proofs. *Journal of Automated Reasoning*, 26(1):51–66, 2001.

[41] Ulf Norell. *Towards a Practical Programming Language Based on Dependent Type Theory*. PhD thesis, Goteborg University, 2007.

[42] Chris Okasaki. Red-black trees in a functional setting. *Journal of functional programming*, 9(4):471–477, 1999.

[43] Simon Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice Hall Series in Computer Science. Prentice Hall, 1987.

[44] Simon Peyton Jones, Will Partain, and André Santos. Let–floating: moving bindings to give faster programs. In *Proceedings of the first ACM SIGPLAN international conference on Functional programming*, pages 1–12. ACM, 1996.

[45] Frank Pfenning. A proof of the Church-Rosser theorem and its representation in a logical framework. Technical report, Carneige-Mellon University, 1992.

[46] Gordon D Plotkin. A structural approach to operational semantics. 1981.

[47] Jason S Reich, Matthew Naylor, and Colin Runciman. Advances in Lazy SmallCheck. In *Symposium on Implementation and Application of Functional Languages*, pages 53–70. Springer, 2012.

[48] 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. ACM, 2008.

[49] Gert Smolka. The Oz programming model. In *Computer science today*, pages 324–343. Springer, 1995.

[50] Zoltan Somogyi, Fergus J Henderson, and Thomas Charles Conway. Mercury, an efficient purely declarative logic programming language. *Australian Computer Science Communications*, 17:499–512, 1995.