Failing Faster: Overlapping Patterns for Property-Based Testing

Jonathan Fowler and Graham Hutton

School of Computer Science
University of Nottingham, UK

Abstract. In property-based testing, a key problem is generating input data that satisfies the precondition of a property. One approach is to attempt to do so automatically, from the definition of the precondition itself. This idea has been realised using the technique of needed narrowing, as in the Lazy SmallCheck system, however in practice this method often leads to excessive backtracking resulting in poor efficiency. To reduce the amount of backtracking, we develop an extension to needed narrowing that allows preconditions to fail faster based on the use of overlapping patterns. We formalise our extension, show how it can be implemented, and demonstrate that it improves efficiency in many cases.

1 Introduction

Property-based testing, popularised by QuickCheck [4], is an automated approach to testing in which a program is validated against a formal 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. In order to generate the input data it is often required to write a custom generator. For example, consider the following simple property of a sorting function:

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

This property has two arguments, given by a number n and a list of numbers l. The property itself states that if the list l is a permutation of the numbers from 0 to $n-1$, then sorting this list will give the expected result. However, while the above definition captures a valid property, it suffers from a practical problem. In particular, if we use a standard generator for a list of numbers then the precondition *perm n l* will rarely be met, making it difficult to generate enough test cases to adequately test the *sort* function.

To overcome this problem, a custom generator is often used. For example, the QuickCheck system [4] provides a range of type-classes and combinators for building custom generators, using which we can define a generator for properties such as *propSort*. Nevertheless, writing custom generators is time consuming and for more complex examples, such as generating well-typed terms, is the subject of research work [12, 18]. Furthermore, it is difficult to combine such generators, in the sense that two generators that are efficient in isolation may no longer be efficient when they are combined together in some way.

Another approach is to attempt to derive an efficient generator from the definition of the precondition, in our example the property *perm*. One realisation of this approach is to use the technique of *needed narrowing* [1, 10] from functional logic programming. For example, Lazy Smallcheck [20] adopts a narrowing strategy and EasyCheck [2] directly uses the needed narrowing language Curry. Using this approach, a program is evaluated in a speculative manner. Beginning with a free input variable, the variable is refined by choosing a constructor when the value is required to proceed with evaluation. If evaluation ends negatively then the process backtracks, while if it ends positively then we have generated a value satisfying the condition. However, a naturally written property often does not make an efficient generator. In particular, the generator may be forced to backtrack excessively if it finds itself in a branch of the program for which the constraints are never satisfied, as we shall see in the next section.

In this paper we explore a new technique to help reduce the amount of backtracking that is required in a needed-narrowing approach to property-based testing. The technique, which is a generalisation of the parallel conjunction operator used in several tools [13, 20, 3], allows evaluation of multiple branches of the program simultaneously, potentially allowing a result to be derived at an earlier stage of refinement. Particularly for operators such as conjunction, disjunction and addition, both arguments can be evaluated in tandem. In the case of conjunction, failure on either side results in failure of the whole expression. To achieve this, we use a form of *overlapping pattern matching*. The pattern matching is resolved in an order-independent fashion and overlapping patterns are allowed. More precisely, in this paper we:

- Motivate and introduce the use of overlapping patterns for needed narrowing property-based testing using our permutation example (section 2).
- Define our source language and formalise the notion of overlapping patterns within this language. In particular, we extend the notion of overlapping patterns with a needed narrowing semantics (section 3).
- Explain how our language can be implemented (section 4) and explore the benefits of overlapping patterns in two case studies (section 5).
- Compare with related work (section 6) and conclude (section 7).

The paper is aimed at a general functional programming audience with some experience of property-based testing systems such as QuickCheck, but no specialist knowledge of needed narrowing is assumed. For the purposes of examples, we use a Haskell-like syntax and semantics. Although we only apply our technique to property-based testing, we believe it could also be used to improve the efficiency of other tools and languages based on needed narrowing.

2 Motivation and Basic Idea

To motivate the need for overlapping pattern matching we take a deeper look at the example from the introduction. We show that naively evaluating the *perm* precondition with a needed narrowing strategy results in excessive backtracking, and how the use of an overlapping conjunction operator mitigates the problem. We begin by defining the *perm* condition as follows:

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

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. Because needed narrowing is more effective on algebraic data types than on primitive data types [16], we assume an inductive type *Nat* of natural numbers built up from basic constructors *Zero* and *Suc*.

By running a needed narrowing evaluation on the above definition for *perm*, we can generate values satisfying the constraints. Needed narrowing is based upon the idea of extending normal evaluation to include *free variables*, with the value of such variables being refined as evaluation proceeds. We given a brief overview to the technique below; for a more in-depth introduction, see [9, 15].

At the start of evaluation, a free variable is chosen for each argument of the program and the program is reduced until the value of a variable is required to continue. The program is then *suspended* on the variable. To allow further progress, a constructor is chosen for the suspended variable and the program is then reduced further in the same way. If evaluation fails to succeed we backtrack, chosing alternative constructors for each variable.

For example, if we consider the first constraint *length* $l \equiv n$ in *perm* for the case when $n = 4$, needed narrowing will yield the following solution:

$$l = [x_0, x_1, x_2, x_3]$$

Along the way we would discard lists such as $[x_0, x_1, x_2]$ which fail to satisfy the constraint. The variables $x_0..x_3$ are free in the solution, in the sense they can be substituted for any value while still satisfying the constraint. Continuing with evaluation of the second constraint, *all* $({<}4)$ l, we further refine the variables. We consider a *partial* solution, in which the constraint is not yet satisfied, to illustrate a situation in which excessive backtracking will occur:

$$l = [1, 1, x_2, x_3]$$

This list begins with two ones and according to the current constraint, *all* $({<}4)$ l, we have neither failed nor succeeded and as such we should carry on refining x_2 and x_3. However, when we arrive at the final constraint, *allDiff* l, this partial solution will fail, and we have to backtrack over all combinations of x_2 and x_3 before we can continue again. Moreover, as we consider generating longer permutations, the amount of backtracking increases exponentially.

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

```
perm n l = length l ≡ n  ∧  allDiff l  ∧  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 *allDiff l* constraint, however it will fail the *all* (<4) *l* constraint but only after the remaining variables $x_1..x_3$ are refined while evaluating *allDiff*. In both cases, the backtracking is caused because a partial solution fails to satisfy a later constraint 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 constraints 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 needed narrowing all 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 pragma to indicate the change in intended semantics:

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

The definition has two new clauses, given by simply commuting 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 ∧ *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 needed narrowing mechanism must be modified to evaluate both arguments of the pattern match before it refines variables.

In *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 section 4, in the time it takes to generate one hundred valid permutations of length nine for the *perm* constraint defined using the traditional conjunction operator, we can generate one hundred valid permutations of length forty using the overlapping version.

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 _     False = True
danger True  True  = True
```

Using this definition, *danger False False* can reduce to either *False* or *True*, depending on whether the first or second clause is used, and is therefore non-deterministic. To counter this, we require confluence laws which guarantee that evaluation is deterministic if all expressions are terminating.

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. For example it is straightfoward to define overlapping versions of the addition and maximum operators on natural numbers, and for the applicative operator <*> on the *Maybe* type [14]. As illustrated by the latter example, overlapping definitions are not restricted to commutative operators. The *maximum* function is defined and used in section 5, and a range of other useful overlapping definitions are provided in our implementation [8].

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

We use a simple functional programming core language with definitions, constructors, variables, lambda expressions and application. The language only allows pattern matching at the top-level of a function definition, which is in the style of Agda [17] and simplifies the theory. However, other styles of pattern matching can readily be rewritten into this form. For example, a **case** expression can be abstracted into a definition with each alternative forming a clause of the definition. The syntax of the language is formally defined as follows:

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

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 use an overline to represent a list of elements, e.g. $\overline{Patt}$ is a potentially empty list of patterns. Expressions and definitions are parameterised by a set of free variables X, which is only used in the needed narrowing semantics. The language has 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. Each variable should only appear 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.

3.1 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 needed narrowing strategy.

First we define a *local* semantics $\rightarrow_R \subseteq\ Expr_X \times Expr_X$ that performs basic reduction steps on expressions, which is then lifted into an evaluation context. As usual, a *substitution* is a mapping from variables to expressions, and 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 $f\ \overline{p} = e\ \in\ \text{defn}(f)$ 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.

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

The predicate `Matches` used above captures the idea of a successful match of expressions against patterns, where `Match` gives the definition for a single pattern, `Matches` for a list of patterns, and s is the resulting substitution:

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

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

In 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 grammer 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.

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

Definition 1. The *full reduction* semantics, $\rightarrow \subseteq Expr_X \times Expr_X$, is given by:

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

Definition 2. $\rightarrow^*$ is the reflexive/transitive closure of $\rightarrow$.

To ensure that our language is deterministic and avoid examples such as *danger False False* from section 2 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 3. Two expressions are *definitionally equivalent*, written $e \equiv e'$, if there exists 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 exists 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 4. The *most general unifier* is defined by the inference rules below. $\texttt{Unify}(p,\ q,\ s_1,\ s_2)$ denotes the unification of patterns p and q by substitutions s_1 and s_2, and similarly for a list of patterns with $\texttt{Unifies}$. Note we are using the assumption that every variable appears only once in each pattern here.

$$\frac{\texttt{Unifies}(\overline{p},\ \overline{q},\ s_1,\ s_2)}{\texttt{Unify}(c\ \overline{p},\ c\ \overline{q},\ s_1,\ s_2)} \qquad \frac{}{\texttt{Unify}(v,\ p,\ \{v \mapsto p\},\ \emptyset)}$$

$$\frac{}{\texttt{Unify}(p,\ v,\ \emptyset,\ \{v \mapsto p\})}$$

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

This definition has the expected behaviour, that is:

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

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. For the terminating subset of the language we can check whether a definition satisfies the confluence property automatically by generating the unifiers pairwise and normalising each clause.

Definition 5. 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 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 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. We return to the issue of expressions with infinite reduction sequences in the concluding section.

3.2 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 needed 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*.

Our notion of evaluation context is call-by-name, and hence only evaluates the left-hand side of an application. When the left-most expression is a definition, we evaluate the arguments until a pattern match is possible. For efficiency, we only reduce arguments that could lead to a pattern match. Sometimes more than one argument needs to be reduced to allow a pattern match, in which case we reduce the arguments in a left-biased order. The rules are defined formally below, where $\overline{\mathbf{C}}$ is a list of expressions with one context.

$$\frac{}{\bullet\ \text{evalcxt}}\ \textsc{hole} \qquad \frac{\mathbf{C}\ \text{evalcxt}}{(\mathbf{C}\ \ e)\ \text{evalcxt}}\ \textsc{app-l} \qquad \frac{\texttt{Subjects}(\overline{\mathbf{C}},\ f)}{(f\ \overline{\mathbf{C}})\ \text{evalcxt}}\ \textsc{args}$$

The `Subject` predicates specify the parts of the arguments that should be reduced. The contexts which form the subjects have a clause of the definition for which they are the first sub-expression blocking the pattern match. All expressions to the left of the subject should already match their respective patterns in the clause. The predicates are defined by the following rules:

$$\frac{\texttt{Subject}(\mathbf{C},\ p) \qquad \texttt{Matches}(\overline{p},\ \overline{e_0},\ \_) \qquad f\ \overline{p_0}\ p\ \overline{p_1} = \_ \in \mathrm{defn}(f)}{\texttt{Subjects}(\overline{e_0}\ \mathbf{C}\ \overline{e_1},\ f)}$$

$$\frac{\mathbf{C}\ \text{evalcxt}}{\texttt{Subject}(\mathbf{C},\ c\ \overline{p})} \qquad\qquad \frac{\texttt{Subject}(\mathbf{C},\ p) \qquad \texttt{Matches}(\overline{p_0},\ \overline{e_0},\ \_)}{\texttt{Subject}(c\ \overline{e_0}\ \mathbf{C}\ \overline{e_1},\ c\ \overline{p_0}\ p\ \overline{p_1})}$$

Definition 6. The *evaluation reduction* semantics, $\to_E$, is now defined by:

$$\frac{\mathbf{C}\ \text{evalcxt} \qquad e \to_R e'}{\mathbf{C}[e] \to_E \mathbf{C}[e']}$$

3.3 Needed Narrowing

The needed narrowing semantics reduces an expression until all the evaluation contexts are suspended on a free variable. At this point we refine the left-most suspending variable to a new value, with the resulting refinements to the free variables being stored in an accompanying substitution. We call this type of substitution a *refinement* to disambiguate it from the general notion.

Formally, a refinement σ of type $X \mapsto Y$ is a function from the free variable set X to *partial values* with free variables Y, where a partial value is a term build up from constructors and variables. Composition of refinements, which we denote by $\ggg$, is defined in the standard way. The null refinement, $\mathit{return} \in X \mapsto X$, corresponds to the trivial substitution that maps each free variable to itself.

Definition 7. The *narrowing set* of a expression, $\mathit{narrowing}(e)$, is the set of refinements that replace the left-most suspended variable with a constructor of the correct type with new free variables for the fields. The narrowing set should be complete, in the sense that it contains every constructor of the type.

For example, the narrowing set for an expression suspended on a variable x of type *Nat* is (where x/c is the point refinement replacing x with c):

$$\{x/\,\texttt{Zero}, x/\,\texttt{Suc}\,y\} \qquad\qquad (y \text{ is a fresh variable})$$

We can now define the needed narrowing reduction as follows:

Definition 8. The *needed narrowing* reduction, $\rightsquigarrow\ \subseteq \mathit{Expr}_X \times (\mathit{Expr}_Y \times X \mapsto Y)$, is defined by the following two inference rules:

$$\frac{e \to_E e'}{e \rightsquigarrow \langle e',\ \mathit{return}\rangle} \qquad\qquad \frac{e \not\to_E \qquad \sigma \in \mathit{narrowing}(e)}{e \rightsquigarrow \langle e[\sigma],\ \sigma\rangle}$$

The first rule states that any evaluation reduction is also a needed narrowing reduction, with no refinement necessary. The second states that if no such reduction is possible then a refinement from the narrowing set should be used.

Definition 9. The natural extension to the reflexive/transitive closure of the needed narrowing reduction is given by composing the resulting refinements:

$$\frac{}{e \rightsquigarrow^* \langle e[\sigma],\ \sigma\rangle} \qquad\qquad \frac{e \rightsquigarrow \langle e,\ \sigma\rangle \qquad e' \rightsquigarrow^* \langle e'',\ \sigma'\rangle}{e \rightsquigarrow^* \langle e'',\ \sigma \mathbin{>\!\!=\!\!\!>} \sigma'\rangle}$$

Note that in the reflexive case we use an arbitrary substituton σ rather than the null refinement *return*, as this simplifies the formulation of the completeness result for our new semantics, which we now present along with soundness.

Theorem 2. *(Needed narrowing is sound.) For every needed narrowing reduction sequence there exists a corresponding reduction in the original semantics:*

$$e \rightsquigarrow^* \langle e',\ \sigma\rangle \implies e[\sigma] \to_E^* e'$$

Proof. By induction on the needed narrowing reduction chain. □

To ensure that the corresponding completeness theorem is valid, we restrict our attention to expressions that strongly normalise under any refinement, which we denote using the predicate `Norm`.

Theorem 3. *(Needed narrowing is complete.) For every reduction of a normalising expression there is a corresponding needed narrowing reduction:*

$$\mathtt{Norm}(e_0) \wedge e_0[\sigma] \to_E^* e_1 \implies \exists\ e_1'.\ e_0 \rightsquigarrow^* \langle e_1',\ \sigma\rangle \wedge e_1 \equiv e_1'$$

Proof. By induction on the length of reduction sequences, which are guaranteed to be finite by the normalisation precondition. In order to complete the proof, we require a slightly generalised induction hypothesis:

$$\begin{aligned} &\mathtt{Norm}(e_0)\ \wedge\ \mathtt{Norm}(e_0')\ \wedge\ e_0[\sigma] \to_E^* e_1\ \wedge\ e_0 \equiv e_0' \\ &\qquad\qquad \implies \exists\ e_1'.\ e_0' \rightsquigarrow^* \langle e_1',\ \sigma\rangle\ \wedge\ e_1 \equiv e_1' \end{aligned}$$

□

4 Implementation

In this section we give a short account of the prototype implementation of the property based testing language. We describe the language, the needed narrowing run-time and how we use property based testing. The implementation can be found at:

`https://github.com/JonFowler/overlapCheck`

The language is a core functional programming language with a Haskell like syntax. It includes algebraic data-types and definition with both overlapping and order-dependent pattern matching. For examples in this paper we have used Haskell syntax which are translated to case-less, monomorphic code in our implementation.

The implementation realises the needed narrowing evaluation in a virtual machine encoded in Haskell. The result of evaluation is given by a lazy search tree. Each node contains a free variable with the branches of the node corresponding to constructor bindings for the variable. Normal-form results are at the leaves. Different search strategies correspond to different methods of traversing the tree.

Properties in our implementation are functions a the return type *Result*. The *Result* type represents the three possible outcomes: a failed precondition, in which case the test case is invalid; a successful result, where the test case satisfies the property; and a failure, where the test case is a counter-example to the property. Generally, we form properties by utilising a specialised implication, $(\implies) :: \mathit{Bool} \to \mathit{Bool} \to \mathit{Result}$.

For the purposes of this paper we use a random search strategy. At each node we select a random constructor according to a defined distribution, until we arrive at a result. If the precondition failed then we backtrack to try and find a valid result. It is sensible to limit the amount backtracking as sometimes we might arrive at a state with no nearby solutions. We do this by limiting how many variables we can reverse, randomly enumerating all possible constructors at each variable in an attempt to find a continuation e.g. If the backtrack limit is three and a failure occurs at a node which is twelve deep in the tree, we will backtrack to a minimum of depth nine in search of a solution.

5 Case Studies

In this section we look at two examples using our overlapping property-based tester. The first case study involves the generation of ordered trees and demonstrates how overlapping patterns can be used to encode bespoke size constraints. The second generates typed expressions for a simple language in which we demonstrate a useful technique for writing efficient narrowing generators. In both examples we focus on the generation of data satisfying a constraint.

Our aim in each case is to find a definition of the constraint that eliminates the need for backtracking (apart from the rebinding of a single constructor). We say that such a constraint *fails fast*. Formally, a constraint fails fast if when testing any partial solution against the constraint it either directly fails or there exists a refinement of the solution which succeeds. The needed narrowing generator formed by a constraint which fails fast is generally efficient.

5.1 Ordered Trees

Consider a type of binary trees with natural numbers stored in the nodes, with the additional constraint that numbers within the tree are ordered:

```
data Tree = Leaf | Node Tree Nat Tree
ordered                  :: Tree → Bool
ordered Leaf             = True
ordered (Node t0 a t1) = allTree (⩽ a) t0  ∧  allTree (⩾ a) t1
                            ∧  ordered t0  ∧  ordered t1
```

Note that *ordered* uses the overlapping version of conjunction to ensure that it can be tested in an efficient manner, and is defined using an auxiliary function *allTree* that checks if every element in a tree satisfies a given condition.

Now consider a function to delete a number from a tree while still maintaining the ordering invariant, as captured by the following property [15, 16]:

```
propDelete      :: Nat → Tree → Bool
propDelete a t = ordered t  ⟹  ordered (delete a t)
```

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. To resolve this problem, narrowing-based testers usually limit the size of the solution by depth or by the number of constructors [15, 16, 20]. However, these metrics are often too simple to avoid backtracking.

To avoid bactracking in our example we need to limit only the depth of the tree and not the depth of elements (limiting the depth of elements limits their value but they have a minimum value dictated by the preceding elements.) Overlapping patterns allow us to limit the size with bespoke constraints. A function to calculate the *depth* of a tree can be defined as follows:

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

In turn, our property can then be refined to include a depth limit:

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

The use of overlapping patterns is crucial in two ways for this kind of example. Firstly, in the new definition of *propDelete*, the sizing constraint relies on overlapping conjunction to be visible while the ordering constraint is being tested. Secondly, and more interestingly, the definition for *depthTree* relies on an overlapping version of the maximum function for natural numbers:

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

The auxiliary function *pred* decrements a natural number, stopping at zero. A traditional maximum function would be left-biased and so the right branch of the tree could become arbitrarily large without triggering the size limit.

The constraint $\mathit{ordered}\ t\ \wedge\ \mathit{depthTree}\ t \leqslant n$ fails fast for any n. With this constraint we are able to produce trees of up to depth ten at the same rate it took to produce tree limited to depth five without the use of overlapping patterns.

TODO: *Add some performance results and concluding remarks.*

5.2 Well-Typed Expressions

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

```
data Expr = Add Expr Expr | If Expr Expr Expr | N Nat | B Bool
```

An 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:

$$\mathit{propEval}\ n\ e = \mathit{typed}\ e\ \wedge\ \mathit{depthExpr}\ e \leqslant n\ \Longrightarrow\ \mathit{notError}\ (\mathit{eval}\ e)$$

We will focus on the *typed* condition. This condition has a trivial 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:

```
data Type = Nat | Bool
typeof               :: Expr → Maybe Type
typeof (Add e e′)    = case (typeof e, typeof e′) of
                         (Just Nat, Just Nat) → Just Nat
                         _                    → Nothing
typeof (If e e′ e″)  = case (typeof e, typeof e′, typeof e″) of
                         (Just Bool, Just t′, Just t″) | t′ ≡ t″ → Just t′
                         _                                       → Nothing
typeof (N _)         = Just Nat
typeof (B _)         = Just Bool
```

However, the function *typeof* has an inefficient narrowing semantics. In particular, for a compound expression to be identified as being ill-typed using the above definition we must first compute the type of all its subexpressions, whereas in many cases this is not necessary. For example, the expression $\mathit{If}\ (\mathit{Add}\ u\ v)\ w\ x$ is ill-typed for any u, v, w, x, but a version of *typed* defined using *typeof* would not be able to deduce this until specific expressions had been filled in for these four variables. In other words, the *typed* condition does not fail fast.

TODO: *Fix the explanation here, which is incorrect.*

To solve this problem we define an alternative constraint, $\mathit{hastype} :: \mathit{Expr} \rightarrow \mathit{Type} \rightarrow \mathit{Bool}$, 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.

```
hastype (Add e e′) Nat = hastype e Nat ∧ hastype e′ Nat
hastype (If e e′ e″) t = hastype e Bool ∧ hastype e′ t ∧ hastype e″ t
```

$$\begin{array}{ll} \mathit{hastype}\ (N\ \_)\ \mathit{Nat} & = \mathit{True} \\ \mathit{hastype}\ (B\ \_)\ \mathit{Bool} & = \mathit{True} \\ \mathit{hastype}\ \_\ \_ & = \mathit{False} \end{array}$$

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:

$$\begin{array}{l} \mathit{hastype}\ (\mathit{If}\ (\mathit{Add}\ u\ v)\ w\ x)\ t \\ \ = \mathit{hastype}\ (\mathit{Add}\ u\ v)\ \mathit{Bool}\ \wedge\ \mathit{hastype}\ w\ t\ \wedge\ \mathit{hastype}\ x\ t \\ \ = \mathit{False}\ \wedge\ \mathit{hastype}\ w\ t\ \wedge\ \mathit{hastype}\ x\ t \\ \ = \mathit{False} \end{array}$$

Further analysis shows that *hastype* fails fast, which once again relies on the overlapping definition of conjunction. Using this new condition, 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:

$$\mathit{propEval}\ n\ t\ e = \mathit{hastype}\ e\ t\ \wedge\ \mathit{depthExpr}\ t \leqslant n \implies \mathit{noError}\ (\mathit{eval}\ e)$$

Note that the typing variable has no effect on the validity of the property, and is only used for the purposes of making the narrowing process more efficient. Analysis will show the combination of *hastype* and *depthExpr* fails fast. For this example there is little performance benefit to the use of overlapping patterns, although they do still play a crucial role in the size constraint. This is because there is only a single typing constraint and the components are largely independent. However in extending the expressions to a more realistic language, such as simply typed lambda calculus, it is convenient to use inter-dependent constraints.

5.3 Discussion

We have also investigated some more in-depth examples, the aforementioned simply typed lambda calculus and red-black trees. In both cases we were able to create generators with a limited, easily definable amount of backtracking while maintaining a compositional style - separating each constraint into constituent parts e.g. the red-black tree constraint is composed of a red constraint, a black constraint, a ordered constraint and a depth limiting constraint. Our red-black implementation is similar to that of Naylor [15], the efficiency afforded by overlapping patterns results in the consistent finding of a bug which Naylor's Reach struggles to find.

The use of narrowing can lead to subtle performance issues, as we saw in section 5.2 where the *typeof* constraint is inefficient. To avoid these performance issues close attention has to be taken to the possible sources of backtracking. Overlapping patterns help by making easy to write constraints with limited backtracking but they are no silver bullet, we still have to be careful to avoid bactracking.

Whereas the use of overlapping conjunction is ubiquitous and performance critical in our examples it is not clear whether the more general form of overlapping patterns is necessary. Particularly, it is possible in our examples to replace the use of other overlapping functions, such as *max*, with additional narrowing variables however the resulting function will normally be less general than its overlapping counterpart.

6 Related Work

Curry On first inspection the functional logic language Curry [11], which implements needed narrowing, appears to have a similar feature to our overlapping patterns - Curry also allows overlapping patterns in definitions. However whereas our overlapping patterns are deterministic with evaluation proceeding along a single branch, Curry's are non-deterministic with evaluation considering every matching branch.

Overlapping patterns The overlapping patterns we use are similar to those of Cockx [5, 6]. Cockx develops overlapping pattern matching for dependent type theory, extending the Agda programming language with them. We both use the additional power of overlapping patterns to draw conclusions faster but Cockx uses them to simplify proofs by eliminating boilerplate while we use them to fail faster in a needed narrowing semantics.

Parallel Conjunction Parallel conjunction has been used in a variety of narrowing based testing tools. It originates in Lindblad's work on data generation [13] and Lazy Smallcheck [20], both of which use an enumerative style of testing. Subsequently parallel conjunction has been used by Claessen et al. [3] to randomly generate data with uniform distribution. The parallel conjunction operator is equivalent to the overlapping conjunction operator but whereas previous focus has been practical we have given a precise narrowing semantics for the more general class of overlapping definitions. The research of Claessen et al. is most similar to ours in that they use a narrowing style for random testing however their aim of uniform distribution, via the use of Feat [7], makes backtracking hard to avoid for many problems.

TODO: *Is this correct (avoiding backtracking hard)?*

7 Conclusion and Future Work

In this article we have motivated and formalised an extension to needed narrowing allowing overlapping patterns. We use the needed narrowing evaluation to generate data satisfying a constraint from a program specifying a constraint. Overlapping patterns allows us to do this efficiently using composable constraints. There are many areas in which this work could be extended and improved. Two interesting ones are:

Other language features The interaction of other language features and overlapping patterns along with needed narrowing is interesting area for further research. Adding first-order functions to the narrowing semantics is one area for which the trie representation used in the extended Lazy Smallcheck [19] offers a starting direction. To establish our completeness theorem we require a strong normalisation condition, a unintended side effect of this restriction is the loss of co-inductive data-types the addition of which is potential area of further research.

Embedding in established tools In this paper we gave a implementation of overlapping patterns in a prototype language. It would be interesting to add overlapping patterns to a more established tool, either a property based testing library such Lazy Smallcheck [20] or a functional logic language such as Curry [11].

References

[1] Antoy, S., Echahed, R., Hanus, M.: A Needed Narrowing Strategy. Journal of the ACM 47(4), pp. 776–822 (2000)

[2] Christiansen, J., Fischer, S.: EasyCheck - Test Data for Free. In: Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). vol. 4989 LNCS, pp. 322–336 (2008)

[3] Claessen, K., Duregård, J., Pałka, M.H.: Generating Constrained Random Data with Uniform Distribution. In: International Symposium on Functional and Logic Programming. pp. 18–34. Springer (2014)

[4] Claessen, K., Hughes, J.: QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In: Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (2000)

[5] Cockx, J.: Overlapping and order-independent patterns in type theory. Ph.D. thesis, Master thesis, KU Leuven (2013)

[6] Cockx, J., Piessens, F., Devriese, D.: Overlapping and order-independent patterns. In: European Symposium on Programming Languages and Systems. pp. 87–106. Springer (2014)

[7] Duregård, J., Jansson, P., Wang, M.: Feat: Functional Enumeration of Algebraic Types. ACM SIGPLAN Notices 47(12), 61–72 (2013)

[8] Fowler, J.: overlapCheck implementation (2016), `https://github.com/JonFowler/overlapCheck`

[9] Fowler, J., Hutton, G.: Towards a Theory of Reach. International Symposium on Trends in Functional Programming pp. 22–39 (2015)

[10] Hanus, M.: A Unified Computation Model for Functional and Logic Programming. In: Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (1997)

[11] Hanus, M.: Curry - An Integrated Functional Logic Language. Tech. rep. (2016)

[12] Hritcu, C., Hughes, J., Pierce, B.C., Spector-Zabusky, A., Vytiniotis, D., Azevedo de Amorim, A., Lampropoulos, L.: Testing noninterference, quickly. In: ACM SIGPLAN Notices. vol. 48, pp. 455–468. ACM (2013)

[13] Lindblad, F.: Property Directed Generation of First-Order Test Data. In: Proceedings of the Eighth Symposium on the Trends in Functional Programming (2007)

[14] McBride, C., Paterson, R.: Applicative Programming With Effects. Journal of Functional Programming 18(1) (2008)

[15] Naylor, M., Runciman, C.: Finding Inputs that Reach a Target Expression. In: Proceedings of the 7th IEEE International Working Conference on Source Code Analysis and Manipulation (2007)

[16] Naylor, M.F.: Hardware-Assisted and Target-Directed Evaluation of Functional Programs. Ph.D. thesis, University of York (2008)

[17] Norell, U.: Towards a Practical Programming Language Based on Dependent Type Theory. Ph.D. thesis, Goteborg University (2007)

[18] Pałka, M.H., Claessen, K., Russo, A., Hughes, J.: Testing an optimising compiler by generating random lambda terms. In: Proceedings of the 6th International Workshop on Automation of Software Test. pp. 91–97. ACM (2011)

[19] Reich, J.S., Naylor, M., Runciman, C.: Advances in Lazy SmallCheck. In: Proceedings of the 24th Symposium on the Implementation and Application of Functional Languages (2013)

[20] Runciman, C., Naylor, M., Lindblad, F.: SmallCheck and Lazy SmallCheck Automatic Exhaustive Testing for Small Values. In: Proceedings of the First ACM SIGPLAN Symposium on Haskell (2008)