Finding Inputs that Reach a Target Expression

Matthew Naylor and Colin Runciman
Department of Computer Science
University of York
Heslington, York, YO10 5DD
{mfn,colin}@cs.york.ac.uk

Abstract

We present an automated program analysis, called Reach, to compute program inputs that cause evaluation of explicitly-marked target expressions. Reach has a range of applications including property refutation, assertion breaking, program crashing, program covering, program understanding, and the development of customised data generators. Reach is based on lazy narrowing, a symbolic evaluation strategy from functional-logic programming.

We use Reach to analyse a range of programs, and find it to be a useful tool with clear performance benefits over a method based on exhaustive input generation. We also explore different methods for bounding the search space, the selective use of breadth-first search to find the first solution quickly, and techniques to avoid evaluation that is unnecessary to reach a target.

1. Introduction

A desirable goal when testing programs is *total program coverage*, meaning that every expression in the program has contributed to at least one correct test run. While automatic approaches to testing, such as random and exhaustive testing, are often able to cover large portions of a program, certain intricate program paths can be problematic. For example, consider the following program fragment whose behaviour depends on whether two input lists `x` and `y` are both sorted or not.

```
if   sorted x && sorted y
then ...
else ...
```

The problem is that there are *many* values of `x` and `y` that cover the else-branch of the conditional expression, and relatively few that cover the then-branch. Using the analysis that we develop in this paper, called Reach, we can insert a special keyword `target` into the program as follows.

```
if   sorted x && sorted y
then target (...)
else ...
```

By applying the Reach analysis to this modified program, we obtain a set of values for `x` and `y` that cause evaluation of the target, and hence evaluation of the then-branch. The analysis is based on *lazy narrowing*, a symbolic evaluation strategy from functional-logic programming [5].

The problem of finding inputs that cause evaluation of marked target expressions in a program is quite a general one. For example, Reach can be used to automatically refute or satisfy arbitrary boolean expressions occurring in a program, such as program properties and assertions. And by placing targets at positions in the program where behaviour is undefined, such as in non-exhaustive case expressions, Reach can be used to find inputs that cause programs to crash. Furthermore, Reach can be used to describe powerful input data generators simply by placing targets at suitable positions in the source program.

The source language that Reach operates on is a core functional language with only a small number of syntactic constructs. This lets us present and implement Reach without getting bogged down in the technical details of any specific language. However, the core language that we use can be directly produced by the York Haskell Compiler [4], so Reach can be applied to standard Haskell [11] programs with only modest restrictions on a few primitive data types.

1.1. Road-map

In Section 2 we introduce and motivate Reach by applying it to part of a binary search-tree implementation, and discuss how Reach fares on this example in comparison to random and exhaustive testing. In Section 3, we precisely define how the Reach analysis works along with the core language that it operates on. In Section 4 we evaluate Reach by applying it to several programs, and we quantify its performance benefit in comparison to an approach based on exhaustive testing. In Section 5 we explore some extensions to

```
del a Empty   = Empty
del a (Node b t0 t1)
  | a < b     = Node b (del a t0) t1
  | a > b     = Node b t0 (del a t1)
  | otherwise = ext t0 t1

ext Empty t   = t
ext (Node a t0 t1) t
              = Node a t0 (ext t1 t)
```

Figure 1. Deletion from a binary search-tree

the analysis, in particular: (1) different methods of bounding the search space; (2) using breadth-first search to find the first solution sooner; and (3) methods to avoid analysing expressions that can't lead to the evaluation of a target. In Section 6 we compare Reach with related work, and finally, in Section 7, summarise our conclusions and discuss future work.

2. An example application of Reach

Suppose that we wish to define the deletion operation of a binary search-tree data structure, and then verify that it is correct. In the functional language Haskell, a data type for binary search trees can be defined as

```
data Tree a = Empty
            | Node a (Tree a) (Tree a)
```

This declaration defines a tree of elements of type `a` to be either the empty tree, or a node containing an element of type `a` and two subtrees, each containing elements also of type `a`. The deletion operation is defined by case-analysis on the input tree, as shown in Figure 1. It behaves as follows:

- If the given tree is empty then `del` simply returns an empty tree.
- If the element to be deleted is less than or greater than the value at the root of the tree, then it is recursively deleted from the tree's left or right subtree respectively.
- If the element to be deleted is equal to the value at the root, then the left subtree is returned, with its rightmost `Empty` constructor replaced with the right subtree.

An important property of deletion is that it preserves the ordering of the input tree, i.e. assuming that the input tree is ordered, then so too is the output tree.

```
prop_ordDel a t =
  ord t ==> ord (del a t)
```

```
every p Empty          = True
every p (Node a t0 t1) =
  p a && every p t0 && every p t1

ord Empty          = True
ord (Node a t0 t1) =
  every (<= a) t0 && every (>= a) t1
```

Figure 2. Ordering predicate over trees

The operator `==>` represents standard boolean implication. The `ord` function checks that the given tree is ordered, as defined in Figure 2. It is defined in terms of the helper function `every` which checks that some predicate `p` holds on every value in the given tree.

To arrange for `prop_ordDel` to be verified by Reach, we need to introduce the following function.

```
refute True  = True
refute False = target False
```

It is just the identity function on booleans, but has the `False` branch marked as the target. We underline the `target` function here because it is specially recognised by Reach. Now we just need to wrap `prop_ordDel` with a call to `refute`.

```
main     :: Int -> Tree Int -> Bool
main a t =  refute (prop_ordDel a t)
```

Notice that we have given an explicit type signature for `main`. This is because `prop_ordDel` is polymorphic over the contents of the given tree, and we want `main` to have concrete type so that we can actually run it. Applying Reach to this program, we get a series of function applications that reach the target. The first is:

```
main 0 (Node 0 Empty (Node 3 Empty
                 (Node 2 Empty Empty)))
```

So, as the reader may have already noticed, our example program is incorrect! The problem is that we forgot to make `ord` recursively call itself on each subtree of the root node.

Before moving on to presenting Reach in full, we take a brief diversion to see how well random and enumerative testing are able to refute `prop_ordDel`.

2.1. Randomly testing the example

Random testing can be used to refute program properties in Haskell using the library QuickCheck [2]. However, because random generation of recursive data types such as trees can lead to very large values, even infinite ones, we

first need to bound the size of the generated trees. This can be achieved by defining a custom generator function called `tree`. Now `main` can suitably redefined for QuickCheck testing as

```
main a =  forAll tree $ \t ->
            prop_ordDel a t
```

At an interactive Haskell prompt (denoted by >) we can test the property.

```
> quickCheck main
OK, passed 100 tests.
```

In this particular test run a counter example has not been found. In fact, the first time that we tried to refute the property using QuickCheck, we required 14 runs of 100 tests before finding a counter example. With a little thought, we can see why random testing can be problematic, even on such a small program:

- A counter-example must contain a tree where every element of the left subtree is less than the root, and every element of the right subtree is greater than the root (as required by the faulty `ord` predicate). Many trees do not satisfy this predicate.
- Furthermore, the element to delete from the tree, also being generated at random, independently of the tree, must occur in the tree.
- Finally, deletion must result in a tree that no longer satisfies the faulty `ord` predicate. There are many deletions that will not break the `ord` invariant.

The performance of QuickCheck on this example can be improved by writing a more sophisticated `tree` generator. For example, it could be modified to generate *ordered* random trees, but then we have an additional obligation to show that the new generator can yield all and only trees satisfying `ord`. The motivation for Reach is to remove the need for writing such generators, and to try and cover intricate execution paths *automatically*.

2.2. Exhaustively testing the example

In related work we have developed our own library for program testing, called *SmallCheck* [12]. SmallCheck is inspired by QuickCheck, but enumerates program inputs *systematically* in order of increasing size, by iterative deepening, up to a specified depth bound. Using SmallCheck, there is no need to explicitly bound the size of the input trees, so the `tree` generator is now easier to define. At an interactive Haskell prompt we can test `prop_ordDel`:

```
> depthCheck 3 main
Depth 3: Failed test no. 851.
  -2
  Node (-2) Empty (Node (-1)
         (Node 0 Empty Empty) Empty)
```

So exhaustive testing is able to reveal a counter example after 851 tests. In Section 4 we will see that Reach is able to cover the same input spaces as an exhaustive checker, but often much more rapidly.

3. The Reach analysis

In this section, we present the Reach analysis. Given a program with any number of marked target expressions, the analysis computes inputs to a top-level *source function*, as specified by the programmer, such that *some* target expression gets evaluated. The analysis can be easily extended to find inputs such that *all* targets get evaluated. This is achieved by sequentially composing the analysis accross several instances of the program where each instance contains only one target from the original.

We define how Reach operates on a *first-order* core functional language. In practice, Reach operates on a higher-order language with the restriction that it cannot synthesise *function values* as top-level program inputs. With this restriction, dealing with higher-order functions is straightforward, so the details are omitted here.

The core language contains standard functional programming constructs: (possibly recursive) function definitions; function application; let-expressions for sharing the results of computations; data constructors for creating tree-shaped values; and case-expressions for inspecting the root and obtaining the children of such values. Full Haskell programs can be translated to the core language, with the exception of *built-in* data types and functions. Currently, Reach does not support such primitives, but it deals with integers by unary-encoding them using normal data constructors.

The analysis respects *lazy evaluation*, meaning that any input it generates will reach the target if the program is evaluated *lazily*. Lazy evaluation means that expressions in the program are evaluated *at most once* and *only if necessary* to compute the program's result.

3.1. Syntax

The syntax of expressions in the core language is defined in Figure 3. The meta-variable v ranges over variable names, f over function names, and c over data constructor names. Sequences of meta-variables are denoted with an overhead arrow, e.g. $\vec{e}$ denotes a sequence of expressions. In the analysis, we assume the availability of a relation $\in$

$$
\begin{array}{rcll}
a & ::= & c\,\vec{v} \mapsto e & \text{(case alternative)} \\
b & ::= & v = e & \text{(let binding)} \\
e & ::= & v & \text{(variable)} \\
 & | & f\ \vec{e} & \text{(function application)} \\
 & | & c\ \vec{e} & \text{(data construction)} \\
 & | & \texttt{case}\ e\ \texttt{of}\ \vec{a} & \text{(case expression)} \\
 & | & \texttt{let}\ \vec{b}\ \texttt{in}\ e & \text{(let expression)} \\
 & | & \{\!|e|\!\} & \text{(target)}
\end{array}
$$

Figure 3. Syntax of expressions

$$
\begin{array}{rcll}
d & ::= & f\,\vec{v} = e & \text{(function definition)} \\
p & ::= & \vec{d} & \text{(program)}
\end{array}
$$

Figure 4. Syntax of programs

such that $x \in \vec{x}$ holds if x is a member of the sequence $\vec{x}$. Finally, note that the $\{\!|e|\!\}$ construction in the syntax definition represents a *target expression*.

The syntax of core programs is defined in Figure 4. Regarding such programs, we assume the availability of a function $fresh$ such that $fresh(f)$ returns a new function definition, defined exactly like f, except that all variables it contains are renamed to be *unique* within the context of the entire program. Knowing that all variables in the program are unique simplifies the presentation as scoping issues such as variable capture can be ignored.

3.2. Overview of the analysis

The analysis is based on a symbolic evaluation strategy from functional-logic programming called *lazy narrowing*. Analysis proceeds by evaluating the source function with *unbound variables* as inputs. If case-inspection is performed on an unbound variable, then the evaluator non-deterministically *forks* into several branches, one for each alternative in the case expression. In each branch, the variable is bound to the pattern of the corresponding case alternative. If the analysis encounters a target expression then it *succeeds* by returning it, and forks off an analysis of the expression inside the target, since it may lead to evaluation of another target.

Forking is implemented by *backtracking* in Reach, although we discuss how breadth-first disjunction can be used selectively to good effect in Section 5.2. Furthermore, symbolic evaluation of expressions with recursive function calls can easily lead to non-termination, so in Sections 3.6 and 5.1 we discuss methods for *bounding* the analysis.

3.3. The analysis relation

Operationally, the analysis works by repeated applications of a *unit-evaluator*. The unit-evaluator attempts to reduce an expression by the smallest perceivable amount. It either results in a data constructor with unevaluated children, representing the root of the expression result, or the target if its value is demanded for evaluation to continue, or an unbound variable.

We define this unit-evaluator as a big-step transition relation, $\rightarrow$, between initial and final states. Initial and final states are of the same type: a pair of the form $\langle s, e\rangle$ containing an expression e and a *substitution function* s that partially maps variables to expressions. So the value of a variable v that is contained in e is $s\ v$, provided that v is actually bound, i.e. $v \in dom(s)$. We use the notation $s[v = e]$ to extend the substitution s with the mapping of v to e.

We now define the transition relation as a number of separate rules, each dealing with a different syntactic construct. Since the unit evaluator stops once it has reduced an expression to a constructor, we have:

$$\langle s, c\,\vec{e}\rangle \ \rightarrow\ \langle s, c\,\vec{e}\rangle \qquad \text{(Constr)}$$

The unit-evaluator also stops upon demanding the value of a target expression:

$$\langle s, \{\!|e|\!\}\rangle \ \rightarrow\ \langle s, \{\!|e|\!\}\rangle \qquad (\text{Targ}_1)$$

In addition, analysis continues on the expression contained in the target, since its value may be required to evaluate another target:

$$\frac{\langle s, e\rangle \ \rightarrow\ \langle s', e'\rangle}{\langle s, \{\!|e|\!\}\rangle \ \rightarrow\ \langle s', e'\rangle} \qquad (\text{Targ}_2)$$

Notice that the previous two rules overlap, capturing the non-deterministic nature of the analysis. The third construct on which the evaluator stops is an unbound variable:

$$\langle s, v\rangle \ \rightarrow\ \langle s, v\rangle \ \underline{\text{if}}\ v \notin dom(s) \qquad (\text{Var}_1)$$

Alternatively, if a variable is bound then its value is retrieved from the substitution, and evaluated:

$$\frac{\langle s, s\ v\rangle \ \rightarrow\ \langle s', e\rangle}{\langle s, v\rangle \ \rightarrow\ \langle s'[v = e], e\rangle} \ \underline{\text{if}}\ v \in dom(s) \qquad (\text{Var}_2)$$

Notice in the final substitution that the variable is mapped to the result of evaluating the expression it is bound to, thus avoiding repeated computation. The rules for function application and let-expressions are quite straightforward and are defined in Figure 5.

The final construct to consider is the case-expression. To evaluate a case expression as a whole it is first necessary to evaluate the case subject:

$$\frac{\langle s[\vec{v} = \vec{e}], e\rangle \;\rightarrow\; \langle s', e'\rangle}{\langle s, f\ \vec{e}\rangle \;\rightarrow\; \langle s', e'\rangle} \qquad \text{(App)}$$

$$\underline{\text{Where}}\ (f'\ \vec{v} = e) = fresh(f)$$

$$\frac{\langle s[\vec{b}], e\rangle \;\rightarrow\; \langle s', e'\rangle}{\langle s, \texttt{let}\ \vec{b}\ \texttt{in}\ e\rangle \;\rightarrow\; \langle s', e'\rangle} \qquad \text{(Let)}$$

Figure 5. Application and Let Rules

$$\frac{(c\ \vec{v} \mapsto e) \in \vec{a}, \quad \langle s[\vec{v} = \vec{e}], e\rangle \;\rightarrow\; \langle s', e'\rangle}{\langle s, c\ \vec{e}, \vec{a}\rangle \;\Rightarrow\; \langle s', e'\rangle} \qquad \text{(Match)}$$

$$\frac{(c\ \vec{v} \mapsto e) \in \vec{a}, \quad \langle s[v = c\ \vec{v}], e\rangle \;\rightarrow\; \langle s', e'\rangle}{\langle s, v, \vec{a}\rangle \;\Rightarrow\; \langle s', e'\rangle} \qquad \text{(Narrow)}$$

Figure 6. Match and Narrow Rules

$$\frac{\langle s, e\rangle \;\rightarrow\; \langle s_0, e_0\rangle, \;\; \langle s_0, e_0, \vec{a}\rangle \;\Rightarrow \langle s_1, e_1\rangle}{\langle s, \texttt{case}\ e\ \texttt{of}\ \vec{a}\rangle \;\rightarrow\; \langle s_1, e_1\rangle} \qquad \text{(Case)}$$

Depending on the result of evaluating the case subject, a number of different transitions can be made. These transitions are defined by a new relation, $\Rightarrow$, similar to $\rightarrow$, except that the initial state additionally contains a list of case alternatives.

There are three cases for $\Rightarrow$ to consider. First, if the case subject evaluates to a target expression, then $\Rightarrow$ defines that target to be the result of evaluating the whole case expression, since evaluation now demands the target's value to proceed. This rule can be thought of as "propagating" the target:

$$\langle s, \{\!|e|\!\}, \vec{a}\rangle \;\Rightarrow\; \langle s, \{\!|e|\!\}\rangle \qquad \text{(TargProp)}$$

Secondly, if the case subject evaluates to a constructor, then evaluation continues on the case alternative that matches that constructor. The Match rule is defined in Figure 6.

Thirdly, if the case subject evaluates to an unbound variable, then evaluation non-deterministically forks for each case alternative. In each branch, the unbound variable is bound to the pattern of the corresponding case alternative. The Narrow rule is also defined in Figure 6. Notice, in the Narrow rule, that the non-determinism is captured by the assertion $(c\ \vec{v} \mapsto e) \in \vec{a}$ where c is an *unconstrained* constructor. In contrast, in the Match rule, the constructor c *is* constrained so evaluation is deterministic.

3.4. Fully-demanding analysis

The $\rightarrow$ relation analyses the program by demanding evaluation of the source function to a value that is just one constructor deep (the root of the tree-shaped result value). To demand full evaluation of the source function, the $\rightarrow$ relation is applied recursively to the root's children in a left-to-right, depth-first order, corresponding with standard lazy evaluation.

3.5. A note on failure

One common built-in primitive that Reach does support, and which we have not mentioned above, is `fail`. Under normal execution, `fail` causes a crash, perhaps with an error message. In Reach, executing `fail` fits naturally alongside non-determinism, and corresponds to triggering backtracking. Calls to `fail` are often introduced in translating Haskell programs to core programs because case expressions in Haskell programs are allowed to be non-exhaustive. Indeed, Reach can be used to crash programs simply by marking all calls to `fail` as targets.

By fully supporting recursive let expressions, our analysis also allows definition of cyclic (conceptually infinite) data structures. However recursive lets can also cause a program to crash, e.g. `let x = x in x`. Such expressions are called black holes. With a very minor modification of the Var_2 rule, Reach can detect black holes and treat them in the same way as `fail`. The other means of program failure, non-termination, famously can't be detected, but it can be avoided, as discussed in the Section 5.1.

3.6. Bounding the search space

In the presence of recursive functions, our analysis is quite likely to get stuck down one particular path of execution, never managing to backtrack far enough to explore alternative paths. For a backtracking (depth-first) analysis to be useful, the search space must somehow be bounded. We limit the tree-depth of input data that can be generated, and fail if the limit is exceeded. This method has the attractive property that the programmer can clearly identify the portion of a program's input space that Reach's results are valid for. For example, if Reach returns no results, then it is a fact that no inputs exist that reach a target, up to the specified depth.

4. Evaluating Reach

In this section, we use Reach to analyse several example programs and compare it with an approach to target-finding based on *exhaustive* test-data generation, which we refer to

as BlindReach. Both approaches bound the maximum tree-depth of input data that can be generated.

We first introduce the example programs, and then apply Reach and BlindReach to each. All of the example programs, and implementations of Reach and BlindReach, are available on the web at `http://www.cs.york.ac.uk/fp/scam07/`.

A binary search-tree library The first example program is a binary search-tree library written by Hinze [6]. This library contains functions very similar to those defined in Section 2, and we have added our faulty `ord` predicate from Figure 2. Reach is used to check two properties of the library. First, `BST.prop1`, which is the same property as `prop_ordDel` from Section 2. Second, `BST.prop3`, which is defined as:

```
prop3 x t = ord t && not (member x t)
              ==> member x (insert x t)
```

We also try to generate inputs that cause the equality branch of the deletion function to be reached (`BST.ReachDel`), by placing a target in the suitable position.

A Countdown solver The second program, written by Hutton [7], solves instances of the "Countdown" problem. Given a integer n and a sequence of integers $\vec{n}$, the Countdown problem is to find an arithmetic expression which refers to each number in $\vec{n}$ at most once, and which evaluates to n. Hutton proves that several desirable properties of his program hold. For example, `CD.lemma7` states that Hutton's optimised algorithm computes the same result as a simple brute-force implementation:

```
lemma7 ns n =
  solutions ns n == solutions' ns n
```

We also use Reach to find countdown problem instances that have at least one valid solution (`CD.gen`):

```
gen ns n =
  refute (null (solutions ns n))
```

Finally we try to find inputs that cause Hutton's program to divide by zero by using a specially-placed target expression in Hutton's `apply` function:

```
apply Div x y =
  targetWhen (y == 0) (x `div` y)
```

Here, `targetWhen` is a useful abstraction on `target`, defined as:

```
targetWhen True  x = target x
targetWhen False x = x
```

A library of digital circuits The third example program is taken from the circuit library of the Reduceron project [9]. The Reduceron is an FPGA-based reduction machine for executing Haskell programs, written in Haskell. We apply Reach to three properties of the circuit library. One property, `Circs.prop2`, relates the addition of bit-vectors (`/+/`) with standard Haskell addition (`+`) using a conversion function (`num`) from bit-vectors to Haskell numbers:

```
prop2 a b =
  num (a /+/ b) == num a + num b
```

This property does not hold in general, as `/+/` throws away its final carry output. The second property, `Circs.prop3`, relates bit-vector equality with number equality in a similar style to `prop2`:

```
prop3 a b = length a == length b
  ==> (a /=/ b) == (num a == num b)
```

The final property relates a binary multiplexor (`binMux`) with the list indexing operator (`!!`), again using `num` conversion:

```
prop1 s xs =
  length xs == 2^length s && rect xs
    ==> (binMux s xs == xs !! num s)
```

The antecedent of this property is very constraining: the length of input list `xs` must be two to the power of the length of the bit-vector `s`, and further, all the bit-vectors in `xs` must have the same length.

A checkmate finder The fourth example program, taken from [10], takes chess end-game scenarios as input and generates a sequence of moves that forces checkmate, if such a sequence exists. Using the program's `isCheckmate` predicate that takes a chess board and a colour representing who is to move next, we use Reach to generate chess boards in which one side is checkmated:

```
gen c b =
  isValidBoard b && isCheckmate c b
```

The `isValidBoard` predicate introduces several constraints, such as "each side must have one king" and "no more than one piece is allowed on each square". Furthermore to increase difficulty, we add the constraint that no pieces are allowed to be placed on the rim of the board.

A silicon compiler The fifth and final example program, taken from [3], is a silicon compiler (and simulator) for an Esterel-like [1] programming language called Flash. We use Reach to investigate the validity of the following property.

```
prop1 p q =
  sameDuration (p :>> q) (q :>> p)
```

Given two programs, `p` and `q`, this property asserts that the time take to execute `p` followed by `q` is the same as the time to execute `q` followed by `p`. We also use Reach to generate Flash programs containing while loops that execute their bodies at least once:

```
circuit (While cond p) start =
  targetWhen (or (cond <&> start)) (..)
```

Here, we pass the condition and the start signal (which triggers execution of the while loop) through a conceptual and-gate, and require that the output of this gate is true on at least one clock cycle. We were motivated to try this example after noticing that randomly generated Flash programs often contain while loops that never execute. This is because the clock cycles on which the condition is true must coincide with those on which the start signal is true. It is therefore pleasing to see how such a data generator can be described so easily using Reach. Furthermore, by strengthening the condition passed to `targetWhen`, we can also easily constrain the while loops to be *terminating*.

4.1. Results

The times taken to find all program inputs that reach the target, at varying data-depth bounds, using both Reach and BlindReach, are shown in Tables 1 and 2.

The parenthesised numbers beside the timings represent how many inputs were found. Note that sometimes the parenthesised number will be smaller when using Reach because Reach-generated inputs may be partially instantiated, representing several concrete inputs as one.

Comparison of Reach and BlindReach on the Flash compiler has not been carried out because it is a potentially non-terminating program, even for depth-bounded inputs. We overcome this problem in the Section 5.1.

4.2. Conclusions

In all cases Reach explores the space of possible inputs in shorter time than BlindReach does. In some cases, such as `Circs.prop1` and `BST.prop3`, Reach can cover the input space 2-3 orders of magnitude faster than BlindReach. And at larger depth-bounds one can expect the benefit of Reach to be even greater. The more rapidly that we can explore the input space, the more chance of finding a target-reaching input such as a counter-example to a property. The `Mate.gen` example illustrates this point very clearly: using Reach we can find a target-reaching input in reasonable time (14s), but using BlindReach we gave up (after 1800s) without finding any inputs.

The results suggest that Reach's benefit is greatest when stringent constraints are placed on the input data very early in evaluation, such as in the antecedent (see `Circs.prop1`) or conjunct (see `Mate.gen`) of a property. However, Reach's benefit is less significant on congruence properties such as `Circs.prop2` and `CD.lemma7`. The likely reason for this is that standard Haskell equality (==) between terms that depend on the top-level inputs will cause the inputs to become fully instantiated, eliminating the possibility that parts of the input space can be pruned. Of course, Reach will not perform the repeated computation that a testing-based approach such as BlindReach will, but this does appear to have as significant an impact on search time.

5. Extensions to the basic analysis

In this section we explore some consequences of using a *recursion bound* instead of a data bound, and also the selective use of *breadth-first search* to find the first input sooner. We also discuss ongoing work that aims to avoid evaluation of expressions that is unnecessary to reach a target.

5.1. Recursion Bound

Instead of bounding the depth of input data, Reach could bound the recursion depth of each function call, with the advantage that the analysis always terminates, even on possibly non-terminating input programs. However, when using such a recursion bound, the input-space that is actually covered by Reach is no longer very clear, and there is no guarantee of a simple relationship between the inputs explored and the recursion depth.

Furthermore it seems that the covered input space can sometimes be quite different, depending on which bounding method is used. For example, Table 1 shows that the first solution found by `Mate.gen` under a data depth bound is at depth 9 and contains 3 chess pieces, whereas under a recursion bound, is at depth 4 and contains 5 chess pieces. The solution with 5 pieces has a data depth greater than 9, so lies outside the set of inputs covered when using the smallest data depth bound that contains a solution.

It is therefore interesting to know if one bounding method is ever preferable to the other, and if so, under what circumstances. During our experiments we found one example where the chosen bounding method seems very important. When using Reach to check `prop1` of the Flash compiler, 544 counter examples are found in 14.8s at recursion bound 2. The smallest of these counter examples has a data depth of 3:

```
(Ifte (True:False:_) Skip Wait) Wait
```

Table 1. Comparing Reach with BlindReach.

		Data Depth		
Problem	Method	3	4	5
BST.prop1	Reach	0.7s (31)	272s (9268)	
	BlindReach	1.5s (31)	2523s (9268)	
BST.prop3	Reach	0.5s (0)	78s (0)	
	BlindReach	1.3s (0)	2340s (0)	
BST.ReachDel	Reach	0.1s (4)	0.1s (6)	
	BlindReach	0.8s (456)	>1800s	
CD.lemma7	Reach	0.4s (0)	14.7s (0)	
	BlindReach	0.5s (0)	52.7s (0)	
CD.DivBy0	Reach	0.2s (0)	2.4s (0)	173s (0)
	BlindReach	0.2s (0)	9.6s (0)	872s (0)
CD.gen	Reach	0.2s (3)	1.6s (26)	88s (215)
	BlindReach	0.2s (3)	8.7s (47)	758s (715)
Circs.prop1	Reach	0.2s (0)	0.2s (0)	
	BlindReach	2.4s (0)	107s (0)	
Circs.prop2	Reach	0.5s (93)	2.6s (451)	17s (1977)
	BlindReach	0.5s (93)	3.3s (451)	21s (1977)
Circs.prop3	Reach	0.2s (0)	0.6s (0)	3s (0)
	BlindReach	0.3s (0)	1.3s (0)	7.2s (0)

Table 2. Comparing Reach with BlindReach, and data bound with recursion bound.

		Data Depth			Recursion Depth		
Problem	Method	5	6	9	2	3	4
Mate.gen	Reach	0.1s (0)	0.1s (0)	13.9s (1^{st})	0.1s (0)	32.1s (0)	44.2s (1^{st})
	BlindReach	0.2s (0)	1140s (0)	>1800s (0)			

Table 3. Comparing data bound with recursion bound.

		Data Depth		Recursion Depth	
Problem	Method	2	3	1	2
Flash.prop1	Reach	9.5s (0)	>300s (0)	0.1s (0)	14.8s (544)

Table 4. The effect of breadth-first conjunction (1).

		Conjunct Ordering					
Problem	Method	1-2-3	2-1-3	2-3-1	3-2-1	3-1-2	1-3-2
BST.prop4	Standard Reach	29s	0.05s	2.2s	>300s	>300s	26s
BST.prop4	Using Breadth-First Conjuncts	0.14s	0.3s	0.31s	0.3s	0.35s	0.17s

Table 5. The effect of breadth-first conjunction (2).

		Conjunct Ordering	
Problem	Method	1-2	2-1
Mate.gen	Standard Reach	14s	>300s
Mate.gen	Using Breadth-First Conjuncts	27s	27s

However, finding a counter example in reasonable time, using a data-depth bound of 3, seems difficult, as suggested by Table 3 (note that for the purpose of this example, Flash has been modified to always terminate). The reason for this difficulty is that there are over 180 million Flash programs at data depth 3, whereas there are under 64 thousand at recursion depth 2. The above counter example can be found at recursion depth 2 because different functions are constructing different parts of the input. So the choice of bounding method certainly matters, and is an important avenue for future experimentation.

5.2. Breadth-first search

In Haskell, evaluation of the expression `p && q` will sequentially evaluate `p` followed if necessary by `q`. This has the consequence for Reach that the order of conjuncts in a boolean expression affects the order in which the search space is explored, and thus can have a significant impact on the time taken to find a target-reaching input. This is slightly unsatisfying since the programmer should not be concerned about such operational issues.

To illustrate, Table 5 shows that flipping the two conjuncts in the function `Mate.gen` causes Reach to take much longer to find an example input. Another example is a property from the binary search-tree library:

```
prop4 a t0 t1 =
  ord t0 && ord t1 && member a t0
    ==> member a (join t0 t1)
```

Table 4 shows the wide variation in times to first solution for different conjunct orders. In an attempt to overcome this problem, we added a breadth-first disjunction operator, `|||`, to Reach. The idea is that `a ||| b` causes Reach to fairly evaluate `a` and `b` in parallel, completely independently of each other. Using this operator, breadth-first conjunction can be defined as follows:

```
a |&| b = (a && b) ||| (b && a)
```

Now, by using `|&|` instead of `&&` in the above examples, the time taken to find the first solution becomes proportional to the minimum time taken by all the permutations of the conjuncts. This illustrated in Tables 4 and 5.

5.3. Avoiding unnecessary evaluation

In some cases it is possible to determine that evaluation of a particular expression cannot cause evaluation of a target. Let us call such an expression *dead*, and define it to be an expression that doesn't contain a target or call a function that is on the call-path to a target-containing function. The possibility now arises to *discard* dead expressions in order to avoid unnecessary evaluation.

However, avoiding the evaluation of any expression may consequently avoid failing, hence Reach could generate inputs which actually cause the program to crash rather than reach a target. A promising solution to this problem is to save discarded expressions and evaluate them only if a target is actually found. One suitable place in the analysis where dead expressions can be discarded in this way is in the fully-demanding evaluator (see Section 3.4). However, work in this area is ongoing and we are not yet ready to present results.

We are also working on a backwards analysis that starts from the target and repeatedly lifts it up through the program, introducing equational constraints along the way, until it reaches the top-level source function. The equational constraints are solved by symbolic evaluation, like in the current analysis. In this approach, parts of the program that are irrelevant to reaching the target are never considered. We have a prototype implementation, but our experiments are still ongoing.

6. Related work

Libraries such as QuickCheck [2] and SmallCheck [12] allow arbitrary functions in a program to be tested automatically, vastly alleviating the problem of having to write test-data generators by hand. The big advantage of these approaches is their simplicity: no source-code analysis is required, and hence they work seamlessly with experimental language features and advanced libraries. However, they generate data *blindly*, so the programmer must often resort to writing custom data generators to obtain the desired coverage of the program.

This problem has been addressed very recently by Lindblad [8] who describes a method for taking program properties written in a functional language and generating test data, *guided by the definition of the property*. Like Reach, Lindblad's method uses techniques similar to functional-logic programming. In addition, Lindblad introduces a parallel conjunction operator `>&<` whereby an expression of the form `p x >&< q x` evaluates `p x` and `q x` in parallel. So, if at any stage a partially instantiated `x` falsifies `p` *or* `q` then the conjunction immediately evaluates to false. Lindblad shows that parallel conjunction reduces the need to tweak the order of conjuncts in properties. So the goal is similar to our breadth-first conjunction operator, but the means is quite different: Lindblad's operator evaluates both conjuncts under the same environment, whereas ours evaluates them independently. Whereas Lindblad's method is only helpful when the conjuncts are dependent on each other, ours can always help, but has the disadvantage of duplicating work.

Lindblad also compares his method against exhaustive test data generation, giving measured timings over a number of benchmarks. The results seem to be quite similar to ours, showing in some cases a speed improvement of 2-3 orders of magnitude. Indeed, this should not be too surprising as there there are many similarities between Lindblad's method and ours, although the two have been developed independently. One of the main differences, however, is that Reach solves a more general problem, of which property-directed data generation is just a special case. Lindblad states that not having to explicitly write custom test-data generators is one of the big benefits of his approach. Instead the programmer can simply state the property and have relevant data generated automatically. We have seen that, in Reach, the potential benefit is even greater: powerful data generators can be developed without even writing a property, simply by placing targets at suitable points in the existing program (for example, the Flash program generator from Section 4). Furthermore, Reach seems to have a wider variety of applications, such as assertion breaking, program covering, program crashing and program understanding, although we have yet to fully explore these possibilities.

7. Conclusions and Future Work

We have presented an analysis, called Reach, that solves the problem of finding inputs that reach any or all of a set of marked target expressions in a program. Reach has been applied to a range of programs and found to be a useful tool with clear performance benefits over a naive solution based on exhaustive input generation. We explored two methods of bounding the search space, one by a data depth bound and the other by a recursion depth bound, and found that the choice of which to use can sometimes be vital to finding a solution in reasonable time. We also found the selective use of breadth-first search useful to avoid the need to tweak the order of conjuncts in program properties.

We are currently exploring how Reach can be extended to avoid evaluation of expressions that cannot lead to a target, including the use of our backwards analysis method, as discussed in 5.3. In future work we would like to explore other possible applications of Reach, to investigate the use of a basic integer constraint solver (so that larger numbers can be handled), and to synthesise function values as inputs.

Apart from the very recent work by Lindblad, we are not aware of other applications of functional-logic programming techniques to the analysis of functional programs. Reach suggests that this combination could be a very fruitful avenue for future research.

8. Acknowledgements

This first author is supported by an award from the Engineering and Physical Sciences Research Council of the United Kingdom. We thank Neil Mitchell for his work on generating simple core functional programs from the York Haskell Compiler, and Emil Axelsson for his comments on an earlier draft.

References

[1] G. Berry and G. Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[2] K. Claessen and J. Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In *Proc. of International Conference on Functional Programming (ICFP)*. ACM SIGPLAN, 2000.

[3] K. Claessen and G. Pace. An Embedded Language Framework for Hardware Compilation. In *Participants Proceedings of the 2002 Workshop on Designing Correct Circuits*.

[4] D. Golubovsky, N. Mitchell, and M. Naylor. Yhc.Core: from Haskell to Core. *The Monad.Reader Issue 7*. `http://www.haskell.org/haskellwiki/The_Monad.Reader`.

[5] M. Hanus and H. Kuchen. Integration of Functional and Logic Programming. *ACM Computing Surveys*, 28(2):306–308, 1996.

[6] R. Hinze. A fresh look at binary search trees. *Journal of Functional Programming*, 12(6):601–607, 2002.

[7] G. Hutton. The Countdown Problem. *Journal of Functional Programming*, 12(6):609–616, Nov. 2002.

[8] F. Lindblad. Property directed generation of first-order test data. The Eigth Symposium on Trends in Functional Programming, New York, 2007, to appear.

[9] M. Naylor. The Reduceron: An FPGA Machine for Executing Haskell. `http://www.cs.york.ac.uk/~mfn/reduceron/`.

[10] W. Partain et al. The `nofib` Benchmark Suite of Haskell Programs. `http://darcs.haskell.org/nofib/`, 2007.

[11] S. Peyton Jones, editor. *Haskell 98 Language and Libraries – The Revised Report*. Cambridge University Press, Cambridge, England, 2003.

[12] C. Runciman. The SmallCheck distribution. `http://www.cs.york.ac.uk/fp/smallcheck0.2.tar`.