Towards a Theory of Reach

Jonathan Fowler and Graham Huttom

School of Computer Science
University of Nottingham, UK

Abstract. When testing a program, there are usually some parts that are rarely executed and hence more difficult to test. Finding inputs that guarantee that such parts are executed is an example of a *reach problem*, which in general seeks to ensure that targetted parts of a program are always executed. In previous work, Naylor and Runciman have developed a reachability solver for Haskell, based upon the use of lazy narrrowing from functional logic programming. Their work was focused on practical issues concerning implementation and performance. In this paper, we lay the groundwork for an underlying theory of such a system, by formally establishing the correctness of a simple reach solver.

1 Introduction

A desirable goal of software testing is for every reachable expression within a program to contribute to at least one test execution of the program. The testing then exhibits program coverage. Random property testing systems such as Quickcheck [2] often cover most of a program, but particularly hard to reach expressions may remain untested. The *Reach* system [11] was developed to address this problem, by generating inputs that evaluate a particular target expression. By using the Haskell Program Coverage (HPC) tool [4] to find expressions which are not tested by Quickcheck, and Reach to generate inputs that evaluate these expressions, the goal of program coverage can be achieved.

Work to date on the Reach system by Naylor and Runciman [11, 12] has focused on the implementation and performance of various underlying solvers. In this paper, we investigate a formal definition for the *reach problem*, and how the forward solver defined in their original paper [11] can be shown to be correct. Having such a theory is important to check the correctness of more complex solvers, such as backwards solver described in Naylor's thesis [12]. The act of formalisation also opens up new potential avenues for further research into alternate evaluation strategies, as discussed in section 7.

The forward reach solver uses a *lazy narrowing* evaluation strategy adapted from functional logic programming. Lazy narrowing can be thought of as an extension of a non-strict language with free variables. The basic idea is that when the value of a free variable is required for a case analysis to proceed, we bind the free variable to each possible alternative form that it may have. To focus on the essence of the problem, we consider a minimal language (section 3) that includes only Peano encoded natural numbers, a target expression, 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 express and understand the properties of the reach problem and lazy narrowing. Within the context of this minimal language we:

- Extend the language with free variables, and give a precise definition for the 'reach problem' in this setting (section 4);
- Define a lazy narrowing semantics for the extended language and use the semantics to define a forward reach solver (section 5);
- Show that the lazy narrowing semantics is sound and complete with respect to the original semantics, and that our reach solver is correct (section 6);
- Provide a mechanical verification of our results in Agda (section 6).

We present proofs for our main results based on a number of lemmas, but for brevity do not provide proofs for the lemmas and refer the interested reader to the accompanying Agda code for the details [3]. The intended audience for the article is functional programmers with a basic knowledge of semantics. No prior knowledge of Reach is assumed; an introduction is given in section 2.

2 The Reach Problem

Reach is a tool for Haskell that can be used help achieve program coverage. A *reach problem* is a Haskell program with a marked target expression and source function. The goal is to find an input to the source function that entails evaluation of the target expression. The target is typically placed in a rarely evaluated expression within the program. The inputs generated from the running of the Reach solver can then be used as test cases for these expressions.

As an example, consider a simplified version of a *balance* function from the standard library *Data.Map*. The *balance* function takes a binary tree and redistributes the tree when one sub-tree contains substantially more elements than the other, in this case four times as many:

$$
\begin{aligned}
&\mathit{balance} :: \mathit{Tree}\ a \to \mathit{Tree}\ a \\
&\mathit{balance}\ (\mathit{Leaf}\ a) = \mathit{Leaf}\ a \\
&\mathit{balance}\ (\mathit{Node}\ lt\ rt) \\
&\quad \mid \mathit{size}\ rt \geqslant 4 * \mathit{size}\ lt = \mathit{balanceToL}\ lt\ rt \\
&\quad \mid \mathit{size}\ lt \geqslant 4 * \mathit{size}\ rt = \mathit{balanceToR}\ lt\ rt \\
&\quad \mid \mathit{otherwise} \qquad\quad = \mathit{Node}\ lt\ rt
\end{aligned}
$$

When testing this function randomly, for example using a *standard generator* for a Quickcheck property [8], the case when the tree is already balanced according to the above definition is tested far more often then the interesting case when the tree needs balancing. By placing a target expression, indicated by $\bullet$, on the branch of the guard requiring the tree to be right-heavy we create a Reach problem which will generate input trees that require balancing:

$$
\begin{array}{l}
balance :: Tree\ a \to Tree\ a \\
balance\ (Leaf\ a) = Leaf\ a \\
balance\ (Node\ lt\ rt) \\
\quad \mid size\ rt \geqslant 4 * size\ lt = \bullet \\
\quad \mid size\ lt \geqslant 4 * size\ rt = balanceToR\ lt\ rt \\
\quad \mid otherwise \qquad\quad = Node\ lt\ rt
\end{array}
$$

A solution to the Reach problem with *balance* as the input function is a tree which satisfies the first guard, such as the following:

$$Node\ (Leaf\ 0)\ (Node\ (Node\ (Leaf\ 1)\ (Leaf\ 2))\ (Node\ (Leaf\ 5)\ (Leaf\ 2)))$$

This tree can then be used as an input to the original *balance* function to ensure that the auxiliary function *balanceToL* is executed as part of testing. In a similar manner, we can move the target expression to the second branch of the guard to find a tree which ensures that *balanceToR* is executed.

2.1 Forward Reach

In this section we introduce the primary reach solver, Forward Reach, defined by Naylor and Runciman [11, 12]. Forward Reach uses *lazy narrowing* in order to generate inputs efficiently. Lazy narrowing is a concept from functional logic programming [1, 6] and can be described as the natural extension of a non-strict semantics to a language with free variables. Free variables are only bound when their value is required for evaluation to proceed.

To illustrate, we give an example of a lazy narrowing reach solver in action. We show the first steps of an analysis of the *balance* function from the previous section. Each state during 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 *balance* x and the initial substitution is the trivial mapping $x \mapsto x$ from x to itself:

1) $\{x \mapsto x\}$
   balance x

The first step of evaluation is to inline the definition for *balance* x:

2) $\{x \mapsto x\}$
   case x **of**
   Leaf $a \to$ *Leaf* a
   Node $lt\ a\ rt \to ...$

In order for evaluation to continue the value of the free variable x is now required, which necessitates a *narrowing* step. To begin with, the variable is bound to the leaf constructor for trees by refining the substitution to $x \mapsto Leaf\ x$, and updating the expression being evaluated accordingly:

3) $\{x \mapsto \mathit{Leaf}\ x\}$
case *Leaf* x **of**
Leaf $a \rightarrow$ *Leaf* a
Node $lt\ a\ rt \rightarrow ...$

We can now reduce the case expression:

4) $\{x \mapsto \mathit{Leaf}\ x\}$
Leaf x

Evaluation of this execution path terminates with the value *Leaf* x. In this case, the target has not been evaluated so the input *Leaf* x is not a solution to the reach problem, independent of any value substituted for x. Evaluation now backtracks and x is bound to the node constructor for trees. After the narrowing step and following reduction of the case expression we have:

5) $\{x \mapsto \mathit{Node}\ x_{\mathbf{l}}\ x_{\mathbf{r}}\}$
if $\mathit{size}\ x_{\mathbf{l}} \geqslant 4 * \mathit{size}\ x_{\mathbf{r}}$ **then** $\bullet$ **else** ...

Analysis will continue with evaluation of the expression $\mathit{size}\ rt \geqslant 4 * \mathit{size}\ lt$. Inputs that evaluate to the target will be collected and evaluation will continue until a set number of solutions is found or a given termination condition is reached, e.g. the input has been enumerated to a particular depth.

Lazy narrowing has two key efficiency benefits over the 'naive' approach in which possible inputs are enumerated and evaluated from the beginning each time. First of all, and most importantly, it allows for portions of the input domain to be either discarded or accepted if the evaluation concludes while there are still free variables in the substitution, as the same conclusion can be drawn for any input formed by replacing these free variables. This can greatly reduce the search space. For example, above we were discard any input of the form *Leaf* x. Secondly, some evaluation is shared between different inputs if they have common structure. In particular, their evaluation is shared up to the point where their differences cause execution to take separate branches.

3 A Minimal Language

In this section we introduce the minimal language that we will use for the rest of paper. The language is not suitable for actual programming, but does provide enough structure to describe the key mechanisms of lazy narrowing. To this end the language has only one type, Peano natural numbers, which provides the simplest example type for showing the recursive mechanics of narrowing. The grammar for expressions of the language is defined as follows:

Exp ::= `Zero`
| `Suc` *Exp*
| $\bullet$

$$
\begin{array}{rcl}
 & | & \textbf{case}\ \mathit{Exp}\ \textbf{of}\ \mathit{Exp}\ \mathit{Alt} \\
 & | & \texttt{var}\ \mathit{Var} \\
\mathit{Alt} & ::= & \texttt{Suc}\ \mathit{Var} \to \mathit{Exp} \\
\mathit{Val} & ::= & \texttt{Zero} \mid \texttt{Suc}\ \mathit{Val}
\end{array}
$$

That is, an expression is either a natural number, a target expression $\bullet$, a case expression, or a variable from some given set *Var* of names. Case expressions have the form **case** e **of** $e_0\ f$, where the first alternative is the `Zero` branch and the second alternative is the `Suc` branch, which can depend on its argument variable. Expressions are assumed to be closed; variables only appear within the case expression in which they are bound. The values of the language are simply the natural numbers. We do not regard the target expression itself as a value, because our intended interpretation is that the values are 'normal' results.

Note that the language does not contain function application or recursion, as these are easily added and have no bearing on lazy narrowing. One might also ask why the target expression is included in the original expression language, rather than the extension we make in the next section to define the Reach problem. The reason is simply for convenience: if the target was excluded we would need to extend both the syntax and semantics in the next section, whereas including it here means that we only need to extend the syntax.

The behaviour of expressions is defined a small-step operational semantics, $\to \subseteq \mathit{Exp} \times \mathit{Exp}$, by means of the following inference rules:

$$
\frac{}{\textbf{case}\ \bullet\ \textbf{of}\ e_0\ f \to \bullet}\ \text{TARGET} \qquad \frac{}{\textbf{case}\ \texttt{Zero}\ \textbf{of}\ e_0\ f \to e_0}\ \text{CASE-Z}
$$

$$
\frac{}{\textbf{case}\ (\texttt{Suc}\ e)\ \textbf{of}\ e_0\ (\texttt{Suc}\ v \to e') \to e'[v := e]}\ \text{CASE-SUC}
$$

$$
\frac{e \to e'}{\textbf{case}\ e\ \textbf{of}\ e_0\ f \to \textbf{case}\ e'\ \textbf{of}\ e_0\ f}\ \text{SUBJ}
$$

Using a small-step semantics enforces a clear order of evaluation, and supports a natural extension to lazy narrowing. If the case subject is a `Zero` or `Suc` then the semantics are standard, where $e'[v := e]$ denotes the substitution of variable v by the expression e in the expression e' in a capture avoiding manner. The target expression behaves in the same way as an error value, i.e. it is always propogated through a case expression to the top level, on the basis that once we have found a target no further evaluation is required.

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

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

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 definition of the Reach problem or the correctness result which follows.

4 Adding Free Variables

To specify the Reach problem we require a notion of *free* variables. One possibility is to simply allow our expressions to be open, letting the existing variables be free. Although this is the approach taken in the original Reach work [11, 12], 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 regular 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 as follows, in which each rule is now parameterised by a set X of free variables, and expressions and values are extended with free variables of the form `fvar` X:

$$
\begin{array}{lcl}
Exp_X & ::= & \texttt{Zero} \\
 & | & \texttt{Suc}\ Exp_X \\
 & | & \bullet \\
 & | & \textbf{case}\ Exp_X\ \textbf{of}\ Exp_X\ \ Alt_X \\
 & | & \texttt{var}\ Var \\
 & | & \texttt{fvar}\ X \\
Alt_X & ::= & \texttt{Suc}\ Var \to Exp_X \\
Val_X & ::= & \texttt{Zero} \mid \texttt{Suc}\ Val_X \mid \texttt{fvar}\ X
\end{array}
$$

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. $Exp \equiv Exp_\emptyset$, $Alt \equiv Alt_\emptyset$ and $Val \equiv Val_\emptyset$.

4.1 Substitutions

An input to an expression is a mapping from its free variables to values. In order to define this formally, we first make a slight detour to introduce the more general notation of a substitution, which will be used later in lazy narrowing. 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:

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

Using this notion, an *input* to an expression can then be viewed as a special case when the set of free variables in the result is empty:

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

We denote substitutions by σ and inputs by τ. The process of applying a substitution is defined recursively on the structure of an expression:

$$
\begin{array}{ll}
\_\,[\_] & :: \mathit{Exp}_X \to \mathit{Sub}_{X \to Y} \to \mathit{Exp}_Y \\
\texttt{Zero}\ [\sigma] & = \texttt{Zero} \\
\texttt{Suc}\ e\ [\sigma] & = \texttt{Suc}\ (e\ [\sigma]) \\
\bullet\ [\sigma] & = \bullet \\
\textbf{case}\ e\ \textbf{of}\ e_0\ (\texttt{Suc}\ v \to e')\ [\sigma] & = \textbf{case}\ e\ [\sigma]\ \textbf{of}\ e_0\ [\sigma]\ (\texttt{Suc}\ v \to e'\ [\sigma]) \\
\texttt{var}\ v\ [\sigma] & = \texttt{var}\ v \\
\texttt{fvar}\ x\ [\sigma] & = \sigma\ x
\end{array}
$$

4.2 Reachability

We can now specify the meaning of *reachability* within our framework. Given an expression $e \in \mathit{Exp}_X$ with free variables X, the set of inputs $\mathit{reach}(e) \subseteq \mathit{Inp}_X$ that reach the target expression is defined as follows:

$$\tau \in \mathit{reach}(e) \quad \Longleftrightarrow \quad e[\tau] \to^* \bullet$$

That is, an input τ that provides values for the free variables in expression e satisfies the reachability condition *iff* the input applied to the expression evaluates to the target. This equivalence describes what it means for a given input to reach the target, but does not describe a specific reach problem. An example such problem might be to find a specific input that satisfies reachability, or to show that none exists. In most languages, but not in our minimal language, the problem is undecidable and therefore an additional termination criterion is included, e.g. find a solution up to a given search depth.

A naive approach to implementing a reach solver is to search for a solution by brute force enumeration and evaluation of all possible inputs. Clearly, however, this is not very efficient. Instead, Naylor and Runciman [11] implement an approach based on lazy narrowing which proves far more effective. This approach shares evaluation, where possible, across the input domain.

5 Lazy Narrowing Semantics

In this section we define a semantics for our minimal language extended with free variables, based upon the notion of *lazy narrowing*, a symbolic evaluation strategy from functional logic programming. As illustrated in section 2, the basic idea of lazy 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 *narrowing step*, 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.

5.1 Preliminaries

We begin by defining a number of concepts that are used in our formalisation of the notion of lazy 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, the relation $\multimap \subseteq Exp_X \times X$ can be defined by the following two inference rules:

$$\frac{}{\texttt{fvar}\, x \multimap x}\ \text{SUSP} \qquad \frac{e \multimap x}{\textbf{case}\ e\ \textbf{of}\ e_0\ f \multimap x}\ \text{SUBJ-SUSP}$$

That is, free variables are themselves suspended, and a case expression is suspended if its subject expression is suspended. Expressions that are suspended can make no further transitions in our small-step operational semantics from section 3. However, the converse is not true. In particular, values and the target expression cannot make further transitions, but are not suspended.

Minimal narrowing set When an expression is suspended there is a set of possible narrowing steps that can be performed. However, in order to maximise laziness, each of the steps that are considered should be *minimal*, in the sense that it should only instantiate the free variable just enough to allow evaluation to continue, and no further. For our language, in which the only values are natural numbers, this means replacing a free variable x by either `Zero` or `Suc` (`fvar` x), the two possible forms that a natural number can have.

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:

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

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, which also manifests itself in our Agda formalisation. Using this operator we can now define the *mimimal narrowing set* $\mathit{Narr}_X(x)$ of a free variable $x \in X$ by simply replacing x by the two possible forms that it may have:

$$\mathit{Narr}_X(x) \;=\; \{x/\,\texttt{Zero},\ \ x/\,\texttt{Suc}\ (\texttt{fvar}\ x)\}$$

This 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 6.2.

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

$$
\begin{array}{ll}
\mathit{return} & :: X \to \mathit{Val}_X \\
\mathit{return} & = \texttt{fvar} \\
(\gg\!=) & :: \ \mathit{Val}_X \to (X \to \mathit{Val}_Y) \to \mathit{Val}_Y \\
\texttt{Zero} \gg\!= \sigma & = \texttt{Zero} \\
\texttt{Suc}\ e \gg\!= \sigma & = \texttt{Suc}\ (e \gg\!= \sigma) \\
\texttt{fvar}\ x \gg\!= \sigma & = \sigma\ x
\end{array}
$$

In fact, *Val* is the *free monad* of the underlying functor for the natural numbers. Using the $\gg\!=$ operator for this monad it is then straightforward to define the *Kleisli composition* operator for substitutions:

$$
\begin{array}{ll}
(\succ\!\!\succ) & :: \ \mathit{Sub}_{X \to Y} \to \mathit{Sub}_{Y \to Z} \to \mathit{Sub}_{X \to Z} \\
\sigma \succ\!\!\succ \sigma' & = \lambda a \to \sigma\ a \gg\!= \sigma'
\end{array}
$$

Along with the monad laws there is one more important 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:*

$$e[\sigma][\sigma'] \ \equiv \ e[\sigma \succ\!\!\succ \sigma']$$

5.2 Semantics

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

- a single step in the original semantics; or
- a minimal narrowing step, 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 in the case of a narrowing step. 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 $\rightsquigarrow \subseteq Exp_X \times (Exp_Y \times Sub_{X \to Y})$ for lazy narrowing by the following two inference rules:

$$\frac{e \to_X e'}{e \rightsquigarrow \langle e',\ return \rangle}\ \text{PROM} \qquad \frac{e \multimap x \qquad \sigma \in Narr_X(x)}{e \rightsquigarrow \langle e[\sigma],\ \sigma \rangle}\ \text{NARR}$$

The first rule promotes transitions from the original semantics to the new semantics, where $\to_X \subseteq Exp_X \times Exp_X$ is the trivial lifting of the transition relation $\to \subseteq Exp \times Exp$ 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.

When sequencing steps in our new semantics together, we need to take account of the presense of substitutions, for which purposes we define a closure operator $\rightsquigarrow^+$ by the following two rules:

$$\frac{e \rightsquigarrow \langle e',\ \sigma \rangle \qquad e' \rightsquigarrow^+ \langle e'',\ \tau \rangle}{e \rightsquigarrow^+ \langle e'',\ \sigma \gg\!\!> \tau \rangle}\ \text{SEQ} \qquad \frac{e \in Exp_X \qquad \tau \in Inp_X}{e \rightsquigarrow^+ \langle e[\tau],\ \tau \rangle}\ \text{FILL}$$

The first rule simply composes the substitutions from the two component reductions. The second rule adds a final narrowing step to the end of a reduction sequence that instantiates any remaining free variables. The reason for including a final narrowing step is twofold. First of all, it makes the definition of reachability in the next section very direct. The second reason relates to the generality of our completeness result, and we will return to this point later on.

5.3 Forward Reachability

Finally, we can now give an alternative characterisation of reachability using our lazy narrowing semantics. Given an expression $e \in Exp_X$, the set of inputs $reach_F(e) \in Inp_X$ that reach the target expression is defined as follows:

$$\tau \in reach_F(e) \quad \Longleftrightarrow \quad e \rightsquigarrow^+ \langle \bullet,\ \tau \rangle$$

That is, an input τ satisfies the forward reachability condition *iff* there is a lazy narrowing reduction sequence that ends with the target and the given input. The key difference with our original definition of reachablity in section 4.2 is that our new semantics *constructs* an input substitution during the reduction sequence, whereas the original semantics requires that we are *given* a substitution so that it can be applied prior to starting the reduction process. In the next section we show that these two notions of reachability coincide.

6 Correctness of the Narrowing Semantics

To prove that forward reachability is equivalent to the original definition, we first formalise the relationship between the lazy 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.

6.1 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:*

$$e \to_X e' \implies e[\sigma] \to_Y e'[\sigma]$$

Theorem 1 (Soundness). *For every reduction sequence in the lazy narrowing semantics there is a corresponding sequence in the original semantics:*

$$e \rightsquigarrow^+ \langle e',\ \tau \rangle \quad \implies \quad e[\tau] \to^* e'$$

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 $\to^*$:

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

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 narrowing step, constructed as follows:

$$\textsc{narr}\ \frac{\dfrac{e \multimap x \qquad \sigma \in Narr_X(x)}{e \rightsquigarrow \langle e[\sigma],\ \sigma \rangle} \qquad e[\sigma] \rightsquigarrow^+ \langle e',\ \tau \rangle}{e \rightsquigarrow^+ \langle e',\ \sigma \gg\!\!= \tau \rangle}\ \textsc{seq}$$

We are now free to use the three assumptions $e \multimap x$, $\sigma \in Narr_X(x)$ 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) $e[\sigma] \rightsquigarrow^+ \langle e',\ \tau \rangle \implies e[\sigma][\tau] \to^* e'$, and then applying lemma 1:

$$\frac{\dfrac{e[\sigma] \rightsquigarrow^+ \langle e',\ \tau \rangle}{e[\sigma][\tau] \to^* e'}\ \textsc{ih}}{e[\sigma \gg\!\!= \tau] \to^* e'}\ \textsc{lemma 1}$$

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

$$\textsc{prom}\ \frac{\dfrac{e \to_X e'}{e \rightsquigarrow \langle e',\ \mathit{return} \rangle} \qquad e' \rightsquigarrow^+ \langle e'',\ \tau \rangle}{e \rightsquigarrow^+ \langle e'',\ \mathit{return} \gg\!\!= \tau \rangle}\ \textsc{seq}$$

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:

$$\dfrac{\dfrac{\dfrac{\text{LEMMA 2}\ \dfrac{e \to_X e'}{e[\tau] \to e'[\tau]} \qquad \dfrac{e' \rightsquigarrow^+ \langle e'',\ \tau \rangle}{e'[\tau] \to^* e''}\ \text{IH}}{e[\tau] \to^* e''}\ \text{SEQ}}{e[\mathit{return} \gg\!= \tau] \to^* e''}\ \text{ID}}$$

□

Although the above proof was presented specifically for the specific case of lazy narrowing semantics, it is not dependent on the properties of the narrowing set or the condition for applying a narrowing step. Therefore the proof is also valid for any narrowing set and any applicability condition.

6.2 Completeness

Definition 1. We exploit two pre-orderings on substitutions, which respectively capture the idea of one substitution being a *prefix* or *suffix* of another:

$$\sigma_1 \sqsubseteq \sigma_2 \iff \exists \sigma'.\ \sigma_1 \gg\!= \sigma' \equiv \sigma_2$$

$$\sigma_1 \leqslant \sigma_2 \iff \exists \sigma'.\ \sigma' \gg\!= \sigma_1 \equiv \sigma_2$$

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* $\sigma \in \mathit{Sub}_{X \to Y}$ *and a transition* $e[\sigma] \to_Y e'$ *for which* $e \not\to x$*, we have:*

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

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

$$\forall x \in X, \tau \in \mathit{Inp}_X\ .\ \exists \sigma \in \mathit{Narr}_X(x).\ \sigma \sqsubseteq \tau$$

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

$$\forall x \in X,\ \sigma \in \mathit{Narr}_X(x).\ \mathit{return} \sqsubset \sigma$$

Theorem 2 (Completeness). *For every reduction sequence in the original semantics there is a corresponding reduction in the lazy narrowing semantics:*

$$e[\tau] \to^* e' \implies e \rightsquigarrow^+ \langle e',\ \tau \rangle$$

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

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

$$\frac{}{e[\tau] \to^* e[\tau]}\ \text{REFL}$$

the goal follows immediately by instantiating free variables:

$$\frac{}{e \rightsquigarrow^+ \langle e[\tau],\ \tau \rangle}\ \text{FILL}$$

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:

$$\frac{e[\tau] \to e' \qquad e' \to^* e''}{e[\tau] \to^* e''}\ \text{SEQ}$$

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:

$$\dfrac{\dfrac{\text{PROM}\ \dfrac{\text{LEMMA 3}\ \dfrac{}{e \to e'_\tau}}{e \rightsquigarrow \langle e'_\tau,\ \mathit{return} \rangle} \qquad \dfrac{\dfrac{e' \to^* e''}{e'_\tau[\tau] \to^* e''}\ \text{LEMMA 3}}{e'_\tau \rightsquigarrow^+ \langle e'',\ \tau \rangle}\ \text{IH}}{e \rightsquigarrow^+ \langle e'',\ \mathit{return} \gg\!\Rightarrow \tau \rangle}\ \text{SEQ}}{e \rightsquigarrow^+ \langle e'',\ \tau \rangle}\ \text{ID}$$

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

$$\dfrac{\dfrac{\text{NARR}\ \dfrac{e \multimap x \qquad \sigma \in \mathit{Narr}(x)}{e \rightsquigarrow \langle e[\sigma],\ \sigma \rangle} \qquad \dfrac{\dfrac{e[\tau] \to^* e'}{e[\sigma][\tau'] \to^* e'}\ \text{LEMMA 1}}{e[\sigma] \rightsquigarrow^+ \langle e',\ \tau' \rangle}\ \text{IH}}{e \rightsquigarrow^+ \langle e',\ \sigma \gg\!\Rightarrow \tau' \rangle}\ \text{SEQ}}{e \rightsquigarrow^+ \langle e',\ \tau \rangle}\ \text{LEMMA 4}$$

Well-foundedness In the third case above, we need to explicitly verify that the induction is well-founded, as in this case this the induction hypothesis is not trivially smaller. Instead, with each iteration the *input* gets smaller. To formalise this well-foundedness neatly and generally, we restrict our notion of substitutions $\mathit{Sub}_{X \to Y}$ to the case when the free variable sets X and Y are finite, and every variable in Y appears in the result of the substitution. For our purposes this leads to no loss of generality and all of our definitions satisfy these restrictions. With these in place, we then have the following two results, which together with lemma 5 ensures that the use of induction in the third case is well-founded.

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

$$\tau_n < ... < \tau_1 < \tau_0$$

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

$$\sigma \gg\!\!> \sigma_1 \equiv \sigma_2 \ \wedge \ \mathit{return} \sqsubset \sigma \implies \sigma_1 < \sigma_2$$

□

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 narrowing steps can always be applied when an expression is suspended.

6.3 Correctness

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

Theorem 3 (Correctness). *For all expressions $e \in Exp_X$:*

$$\mathit{reach}_F(e) \equiv \mathit{reach}(e)$$

Proof.

$$\begin{aligned}
\tau \in \mathit{reach}_F(e) &\iff e \rightsquigarrow^+ \langle \bullet, \tau \rangle && \text{(by definition)} \\
&\iff e[\tau] \to^* \bullet && \text{(theorems 1 and 2)} \\
&\iff \tau \in \mathit{reach}(e) && \text{(by definition)}
\end{aligned}$$

6.4 Formalisation

Our results have also been formalised in Agda [13], and mechanically verified proofs for the main correctness theorem and all associated lemmas are available online [3]. The formalisation follows the above presentation closely, and provides a machine-checked guarantee that our results are correct. In addition, the use of Agda was beneficial for helping to guide the development of our theory and proofs, and led to a number of simplifications of the original versions.

7 Conclusions and Future Work

In this article we established the correctness of a reach solver for a minimal language, based upon a soundness and completeness result for a lazy narrowing semantics. Our final formulation of the semantics is the result of several iterations and improvements, and captures the essential ideas of lazy narrowing in a simple and concise manner. To the best of our knowledge, this is the first formalisation

of lazy narrowing to take advantage of an underlying functional semantics in order to simplify the theory. By way of comparison, the seminal work of Antoy et al [1] established a soundness and completeness result for the related notion of *needed narrowing*. However, their work was formulated within the domain of logic programming, whereas ours takes place in a functional setting. In particular, we are then able to take advantage of simple techniques such as small-step semantics and rule induction to streamline our definitions and proofs, and to facilitate a direct formalisation of our results in the Agda system.

There are number of interesting directions in which the theory developed in this article could be extended and improved, which are summarised below.

Other reach solvers The work in this article lays the ground for attempting to formalise alternative and more general reach solvers, such as the Backward Reach solver defined in Naylor's thesis [12]. In addition, tools such as Lindblad's data generator [9] and Lazy SmallCheck [15] define logical *or* operators in which both expressions are evaluated in parallel, which could signifiantly improve the performance of lazy narrowing as *or* expressions can be reduced to true if either of their arguments reduce to true in the current substitution state. We could easily add such an operator to our language. However, our formulation suggests a generalisation to this idea, in the form of evaluating branches in parallel and utilising equational reasoning on case expressions.

Other language features We used a minimal language for simplicity, but it is important to consider how our approach generalises to other language features. For algebraic datatypes, we expect it should be straightforward to extend our theory using ideas from generic programming as in the work of Hinze [7], while first-order functions could be handled by representing functions using tries as in the improved Lazy Smallcheck [14]. Another interesting area to explore is dependent type theory. Lazy narrowing is often used in automated property based testing and dependent type theory seems a natural coupling as it offers an inbuilt language for specifications. In this area there is also potential for interesting comparison to related work such as automated proof search [13].

Efficiency We showed that the lazy narrowing definition of reachability for our language is correct with respect to the original specification of reachability. However we have not made any formal argument regarding the efficiency of the lazy narrowing approach, either against an alternative narrowing semantics or a naive approach based on brute force search. Such an argument could be made on the basis of simply counting the number of reduction steps required, or adopt a more sophisticated approach, for example using the idea of *improvement theory* [10], which has recently been used to prove that a general purpose optimisation technique for lazy languages never makes programs worse [5].

Acknowledgements

We would like to thank members of the Functional Programming Lab in Nottingham for useful comments and suggestions regarding this work.

References

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

[2] 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 - ICFP '00. vol. 35, pp. 268–279 (2000)

[3] Fowler, J.: Towards a Theory of Reach - Agda Proof (2015), https://github.com/JonFowler/theoryofreach

[4] Gill, A., Runciman, C.: Haskell Program Coverage. In: Proceedings of the ACM SIGPLAN Workshop on Haskell - Haskell '07. pp. 1–12 (2007)

[5] Hackett, J., Hutton, G.: Worker/Wrapper/Makes It/Faster. In: Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming (2014)

[6] 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 - POPL '97. pp. 80–93 (1997)

[7] Hinze, R.: A New Approach to Generic Functional Programming. In: Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages - POPL'00. pp. 119–132 (2000)

[8] Hughes, J.: QuickCheck: An Automatic Testing Tool for Haskell (QuickCheck manual), http://www.cse.chalmers.se/ rjmh/QuickCheck/manual.html

[9] Lindblad, F.: Property Directed Generation of First-Order Test Data. In: Proceedings of the Eighth Symposium on the Trends in Functional Programming - TFP '07 (2007)

[10] Moran, A., Sands, D.: Improvement in a Lazy Context: An Operational Theory for Call-by-need. In: Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (1999)

[11] 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 - SCAM '07. pp. 133–142 (2007)

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

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

[14] 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 - IFL' 12. pp. 53–70 (2013)

[15] 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. pp. 37–48 (2008)