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 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 [6] was developed to address this problem, by generating inputs that evaluate a particular target expression. By using the Haskell Program Coverage (HPC) tool [3] 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 [6, 7] 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 [6] 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 [7]. The act of formalisation also opens up new potential avenues for further research into alternate evaluation strategies, as discussed in section 8.

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 7).

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. 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:

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

When testing this function randomly, for example using a *standard generator* for a Quickcheck property [5], the trivial case when the tree is already balanced is tested far more often then the interesting case when the tree needs balancing. By placing a target expression, which we indicate by the symbol $\bullet$, 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:

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

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

$$\mathit{Node}\ (\mathit{Leaf}\ 0)\ (\mathit{Node}\ (\mathit{Node}\ (\mathit{Leaf}\ 1)\ (\mathit{Leaf}\ 2))\ (\mathit{Node}\ (\mathit{Leaf}\ 5)\ (\mathit{Leaf}\ 2)))$$

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 [6, 7]. Forward Reach uses *lazy narrowing* in order to generate inputs efficiently. Lazy narrowing is a concept from functional logic programming [1, 4] 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 simply 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 a value for 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 $x \mapsto \mathit{Leaf}\ x$, and updating the expression being evaluated accordingly:

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

We can now reduce the case expression:

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

Evaluation of this execution path terminates with the value $\mathit{Leaf}\ x$. In this case, the target has not been evaluated so the input $\mathit{Leaf}\ x$ 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 $\mathit{Leaf}\ x$. 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

In order 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 [6, 7], 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 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 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. In particular, a *substitution* of type $X \to Y$ is a mapping from a set of free variables X to partial values that contain free variables of type 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. show there is no 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 [6] 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 first argument 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 such a number can have.

To formalise this idea, we begin by writing $x \,/\, a$ for the one-point substiution 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, the minimal narrowing set is *advancing*, in the sense 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:

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

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:

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

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.*

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

5.2 Semantics

We now have all the ingredients required to define the 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 the 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:

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

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.

*** GRAHAM HAS REVISED UP TO HERE ***

6 Correctness of Forward Reach Semantics

To prove that the Forward Reach semantics satisfies the criteria of reachability we first show that the lazy narrowing semantics are correct. That is, the lazy narrowing semantics are sound (§6.1) and complete (§6.2) with respect to the original semantics.

6.1 Soundness

Theorem 1. *For every reduction in the lazy narrowing semantics there is a corresponding reduction in the original semantics.*

$$e \rightsquigarrow^{+} \langle e',\ \tau \rangle \implies e[\tau] \rightarrow^{*} e'$$

Before proving soundness we introduce a lemma.

Lemma 2. *A small step reduction in the original language can be lifted over a substitution.*

$$\forall\, \sigma : Sub_{X \to Y}, \quad e \rightarrow_X e' \implies e[\sigma] \rightarrow_Y e'[\sigma]$$

Proof of soundness. The proof proceeds by rule induction. There are three cases to consider:

Case 1 The base case, the lazy narrowing reduction is a *fill* rule:

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

The goal is just a direct instance of the *refl* rule of transitive closure.

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

Case 2 We consider two cases for the *seq* rule. Firstly, when the first reduction is a narrowing reduction:

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

In this case the proof almost a direct instance of the inductive hypothesis, $e[\sigma] \rightsquigarrow^{+} \langle e',\ \tau \rangle \implies e[\sigma][\tau] \rightarrow^{*} e'$. The narrowing relation is composed with the application.

$$\frac{\dfrac{e[\sigma] \rightsquigarrow^{+} \langle e',\ \tau \rangle}{e[\sigma][\tau] \rightarrow^{*} e'}\ \textsc{sound}}{e[\sigma \gg\!\!> \tau] \rightarrow^{*} e'}\ \textsc{lemma-1}$$

Case 3 The second case for the *seq* rule occurs when the first reduction is a promoted reduction from the original language.

$$\textsc{narr}\ \frac{\dfrac{e \rightarrow_X e'}{e \rightsquigarrow \langle e',\ return \rangle} \qquad e' \rightsquigarrow^{+} \langle e'',\ \tau \rangle}{e \rightsquigarrow^{+} \langle e'',\ return \gg\!\!> \tau \rangle}\ \textsc{seq}$$

The promoted reduction from the original language is lifted by the substitution and then sequenced with the inductive hypothesis of the remaining reduction.

$$\textsc{lemma-2}\ \dfrac{\dfrac{\dfrac{e \to_X e'}{e[\tau] \to e'[\tau]} \qquad \dfrac{e' \rightsquigarrow^+ \langle e'',\ \tau\rangle}{e'[\tau] \to^* e''}\ \textsc{sound}}{e[\tau] \to^* e''}\ \textsc{seq}}{e[\mathit{return} \gg\!= \tau] \to^* e''}\ \textsc{ret}$$

□

6.2 Completeness

Lemma 3. *If the base expression of a reduction is not suspended then the reduction can be "unlifted".*

$$e[\tau] \to e' \wedge (\forall x,\ \neg(e \multimap x)) \implies \exists e'_\tau,\quad e \to e'_\tau \wedge e'_\tau[\tau] \equiv e'$$

Definition 1. We define an ordering on substitutions. A substitution is a prefix of another if their exists a substitution joining them.

$$\sigma \sqsubseteq \sigma' \iff \exists \upsilon,\quad \sigma \gg\!= \upsilon \equiv \sigma' \wedge$$

A narrowing set, set of substitutions, is complete if for every input their is a substitution that is a prefix of the input.

$$N \subseteq \mathit{Sub}_{X \to Y},\quad \mathit{complete}(N) \iff \forall \tau : \mathit{Inp}_X\ .\ \exists \sigma \in N\ .\ \sigma \sqsubseteq \tau$$

Lemma 4. *The lazy narrowing set is complete.*

$$\forall x \in X,\quad \mathit{complete}(\mathit{Narr}_X(x))$$

Lemma 5. *The lazy narrowing set is advancing.*

$$\forall x \in X,\quad \sigma \in \mathit{Narr}_X(x),\quad \mathit{return} \sqsubset \sigma \tag{1}$$

Theorem 2. *For every reduction 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. We consider three cases, the original reduction is: a *refl* rule, a *seq* rule in which the base expression is not suspended and a *seq* rule where the base expression is suspended.

Case 1 The *refl* rule.

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

Directly yields the *fill* rule.

$$\dfrac{}{e \rightsquigarrow^+ \langle e[\tau],\ \tau\rangle}\ \textsc{fill}$$

Case 2 The *seq* rule along with the condition that the base expression is not suspended, $\neg\, Susp(e)$:

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

As the base expression is not suspended we can use lemma 3, the "unlift" lemma. This gives us a small step reduction from the base expression:

$$e \to e'_\tau \tag{unlift}$$

Along with an equality:

$$e'_\tau[\tau] \equiv e' \tag{eq}$$

The proof follows:

$$\dfrac{\dfrac{\dfrac{\text{UNLIFT}\ \dfrac{}{e \to e'_\tau}}{e \rightsquigarrow^+ \langle e'_\tau,\ return\rangle}\ \text{COMP} \qquad \dfrac{\dfrac{e' \to^* e''}{e'_\tau[\tau] \to^* e''}\ \text{EQ}}{e'_\tau \rightsquigarrow^+ \langle e'',\ \tau\rangle}\ \text{COMP}}{e \rightsquigarrow^+ \langle e'',\ return \gg\!\Rightarrow \tau\rangle}\ \text{SEQ}}{e \rightsquigarrow^+ \langle e'',\ \tau\rangle}\ \text{RET}$$

Case 3 Finally, any reduction, $e[\tau] \to^* e'$, with a suspended base expression, $e \multimap x$.

As the narrowing set, $Narr_(x)$, is complete we can find a substitution within it which contains the input, τ.

We have, $\sigma \in Narr_(x)$, and input τ' such that:

$$\tau \equiv \sigma \gg\!\Rightarrow \tau' \tag{split}$$

Therefore:

$$\dfrac{\dfrac{\dfrac{\text{NARR}\ \dfrac{e \multimap x \qquad \sigma \in Narr_(x)}{e \rightsquigarrow \langle e[\sigma],\ \sigma\rangle} \qquad \dfrac{\dfrac{e[\tau] \to^* e'}{e[\sigma][\tau'] \to^* e'}\ \text{SPLIT \& LEM-1}}{e[\sigma] \rightsquigarrow^+ \langle e',\ \tau'\rangle}\ \text{COMP}}{e \rightsquigarrow^+ \langle e',\ \sigma \gg\!\Rightarrow \tau'\rangle}\ \text{SEQ}}{e \rightsquigarrow^+ \langle e',\ \tau\rangle}\ \text{SPLIT}$$

Well Founded What is not clear is that case 3 is that case 3 is a sound induction. To do so we show that the number of narrowing steps performed in a lazy narrowing reduction must be finite; this shows the induction is well founded.

$$\tau' \leq \tau \iff \exists \sigma, \quad \tau \equiv \sigma \gg\!\Rightarrow \tau'$$

$$\tau >_\sigma \tau' \iff \tau \geq_\sigma \tau' \wedge \tau' \not\geq \tau$$

There is only a finite sequence of τ_i such that:

$$\tau_0 > \tau_1 > ... > \tau_n$$

$$\tau \geq_\sigma \tau' \wedge return \sqsubset \sigma \iff \tau >_\sigma \tau'$$

6.3 Correctness

7 Agda Proof

- Dependently typed language
- Confidence in proof, use of rule induction
- Neat setting for proofs on semantics: inductive types model rules directly, pattern matching makes rule induction easy.

8 Conclusions and Future Work

In this article we explored the process of formalising Reach for a minimal language. Particularly we looked at a forward reach solver based on a lazy narrowing semantics for the language. We showed that the soundness and completeness of the lazy narrowing semantics implied that the solver was correct. We are currently working on an Agda formalisation of this result. Below we discuss limitations of our current work and avenues of further research.

Needed narrowing The traditional formulation of lazy narrowing is given by Antoy et al. in their seminal paper on needed narrowing [1]. Our approach differs from theirs in several ways. Primarily, our lazy narrowing semantics is defined over a functional semantics with a clear order of evaluation whereas they deduce a sensible order from a rewrite system with no order of evaluation; the laziness of our semantics is inherited from the laziness of the underlying language whereas they construct this laziness. Therefore our strategy is simpler and more appropriate for the analysis of a functional language in which the evaluation order is set. Secondly, we have formally verified our proof of correctness.

Extended language features In this article we have focused on a minimal language avoid complications. Although it should be straightforward to extend our language to handle algebraic data types using generics, there are many more interesting domains to explore reachability in. The most obvious omission is the lack of functions in our language. First order functions could be introduced to lazy narrowing using a trie representation, such as in the improved Lazy Smallcheck [9].

Another area for extension is primitive data types. Native representations of integers and characters do not fit into lazy narrowing neatly as they are large flat data types with strict semantics. In the original Reach integers are replaced with Peano natural numbers. It is clear that Peano natural numbers are only

suitable for problems which only require small numbers and further work on lazy representations is needed to cover all problems effectively.

A final interesting area to explore is dependent type theory. Lazy narrowing has frequently been used for automated property based testing and dependent type theory seems a natural coupling as it offers an inbuilt language for specification. Also there is potential for interesting comparison to existing work such as automated proof search [8].

Parallel evaluation Tools such as Lindblad's data generator and Lazy SmallCheck define disjunctive operators in which both expressions are evaluated in parallel. This can result in a major improvement in the performance of lazy narrowing; the result can be deduced if either expression evaluates to true for the current substitution state. We could easily add a similar operator to our language. However our formulation suggests a generalisation to this idea, we evaluating branches in parallel and utilising equational reasoning on case expressions.

Efficiency We showed that the lazy narrowing definition of reachability for our language adheres to the original specification. However we have not made any formal argument regarding the efficiency, either against an alternative narrowing semantics or a naive approach. Such an argument could be made on the number of reduction steps required to find the reach solutions in a given, finite set of inputs.

References

[1] Antoy, S., Echahed, R., Hanus, M.: A Needed Narrowing Strategy. In: Proc. 21st ACM Symposium on Principles of Programming Languages. vol. 47, pp. 776–822 (2000)

[2] Claessen, K., Hughes, J.: QuickCheck: a Lightweight Tool for Random Testing of Haskell Programs. ACM Sigplan Notices 35, 268–279 (2000)

[3] Gill, A., Runciman, C.: Haskell Program Coverage. Proceedings of the ACM SIGPLAN workshop on Haskell workshop - Haskell '07 p. 1 (2007)

[4] 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. No. 0 in POPL '97, ACM Press (1997)

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

[6] Naylor, M., Runciman, C.: Finding Inputs that Reach a Target Expression. In: SCAM 2007 - Proceedings 7th IEEE International Working Conference on Source Code Analysis and Manipulation. pp. 133–142 (2007)

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

[8] Norell, U.: Towards a practical programming language based on dependent type theory. PhD thesis Department of Computer Science and Engineering Chalmers University of Technology pp. 13264–13269 (2007)

[9] Reich, J.S., Naylor, M., Runciman, C.: Advances in Lazy SmallCheck. In: Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). vol. 8241 LNCS, pp. 53–70 (2013)