Property Directed Generation of First-Order Test Data

Fredrik Lindblad

Chalmers University of Technology / Göteborg University

Abstract

Random testing is a powerful method for verifying program properties. However, as the complexity of the program and properties increases, writing customized input data generators quickly becomes necessary. We present a method for systematic generation of input data by lazy instantiation using meta variables and parallel partial evaluation of properties. This is applied on specification based program verification. We claim that some program verification problems can be handled by systematic generation without the need of writing custom generators, and that some problems, for which writing generators is not a solution and random testing fails, are still simple enough for systematic generation. The system we present is related to functional logic programming.

1 INTRODUCTION

One way to test a program is to write formal specifications for some aspects of its intended bevahiour. This enables automatic instead of manual testing, since the specifications can be used to assess whether a test succeeds or not. QuickCheck[2] is a popular tool for random testing of Haskell programs. Here, the program properties are expressed as boolean valued functions in Haskell itself.

As an example, imagine that we are representing binary search trees (BST) by ordinary binary trees and that we have implemented an operation, `merge`, which merges two BST:s. The precondition of `merge` should be that the two trees are really BST:s, i.e. the order of the elements are correct. We therefore implement a boolean function, `isBST`, which checks whether a tree is a BST. Next, we can state the property that, given two BST:s, the result of `merge` is again a BST:

```
prop t1 t2 = (isBST t1 && isBST t2) `implies`
               isBST (merge t1 t2)
```

If non-specialised random testing is applied on this property, then most of the generated input trees will not be BST:s, since a less restrictive representation is used, namely general binary trees. Each time this happens, the implication is rendered effectless. A lot of fruitless tests are thus performed. This problem can be overcome in QuickCheck by writing custom data generators, which narrow the generation and makes the precondition fulfilled more frequently.

What we propose is to instead make more use of the precondition which has already been written, in this case `isBST`. This can be achieved by generating input data step-by-step while evaluating the functional description of the precondition.

To gain the benefits from this, the search should be systematic rather than random. This way of searching for data is in itself not new, see the section on related work. But, to our knowledge, we make some contributions both in terms of the presentation of the system and in terms of its area of application.

In order to allow for a property directed generation of data, we start from a small functional language with lazy evaluation and add *meta variables* (or logical variables). These are used as place holders for unknown parts of the data. During the search, the data is instantiated one step at a time, in the order in which it is required by the evaluation of the property. We call this mechanism *lazy instantiation*.

Another addition w.r.t. a standard functional language is a possibility to perform *parallel evaluation*[1]. This is useful in the presence of meta variables. In particular, consider a predicate which is a conjunction of two other predicates, `p x = p1 x && p2 x`. Both `p1` and `p2` put restrictions on `x`. If `(&&)` is defined in the normal way, i.e. by looking at the arguments one at a time from left to right, we may have a situation where `x` is partially instantiated in such a way that, at this point, the value of `p2 x` is known to be `False`, whereas the value of `p1 x` is still unknown. Thus, by looking at both arguments in parallel, we might know something about the value of a function at an earlier stage and thereby reduce search space. Of course, replacing the normal `(&&)` by a parallel version requires precaution in the presence of partial functions.

A key question is whether this property directed approach has an impact on the size of the search space substiantal enough to be noticed in practice. Later on we will claim that this is the case by presenting some examples.

2 RELATED WORK

2.1 Automatic Random Testing

In the area of functional programming, *QuickCheck*[2] is a popular tool for testing. It is easy to adopt, since properties are written as normal boolean functions. A disadvantage of QuickCheck is that the properties are not taken into account while constructing input data. It was argued in the introduction that this sometimes makes the testing inefficient.

There is a another tool for testing haskell code, called *SmallCheck* and developed by Colin Runciman. Like our approach, it is based on replacing random by systematic testing. But, like QuickCheck, it blindly generates data and then tests it. The main idea behind SmallCheck is that counter examples many times are small and that systematic generation can be better for various reasons, e.g. becuase it can find the smallest counter example first. Our approach shares many characteristics of SmallCheck, but adds some machinery to allow a more efficient systematic search.

2.2 Functional Logic Programming

The search algorithm we present is closely related to that of *functional logic programming*. One of the most popular such language is *Curry*[3]. The search in Curry has two main components; residuation and narrowing. In our approach the search strategy closely corresponds to that of narrowing, which exploits lazy evaluation to achieve a demand-driven search in a similar way[5]. Also, Curry has a construction for *concurrent conjunction*, more or less corresponding to our parallel evaluation. The semantics of concurrent conjunction is however not quite precisely presented[4] and among the various implementations of the language it does not seem to be formally supported.

3 THE SYSTEM

Program verification as presented in the introduction readily generalises to the problem of finding members of a decidable predicate expressed as a boolean function, `p :: X -> Bool`. In other words, we search for values `x` of type `X` such that `p x` computes to `True`. The presented system is restricted in such a way, that `X` must be a first-order algebraic datatype. Including higher-order types would require a more involved machinery.

We start from a functional programming language with first-order non-recursive algebraic datatypes and case-expressions. To this we add meta variables, denoted by indexed question-marks, $?_i$, which allow step-by-step construction of values during search. A second extension is that, instead of an ordinary case expression, the language has a construction for *select-case* terms. This is simply a collection of normal case expressions which are evaluated in parallel. As soon as it is clear which branch to choose for one of the case expressions in a select-case term, it is selected for further evaluation and all the other case expressions are discarded. The feature of parallel case expressions is intended only to allow symmetrical definitions of operators such as conjunction, as described in the introduction. A function is supposed to be well-defined, even though it is defined by parallel case expressions. Therefore, it is harmless to discard all alternatives but one in a select-case term.

Recursive datatypes and functions are not part of the presentation. They can be thought of as infinite structures or as defined using meta-level fix-points. Termination issues are not addressed.

After a more precise presentation of the syntax and the partial reduction of terms, we describe a simple search strategy for the problem stated above. This is followed by some notes on the correctness of the search strategy as well as on our implementation.

$$
\begin{array}{rcl@{\qquad}rcl}
\tau & ::= & \overline{c\ \overline{\tau}} & \Gamma & ::= & \varnothing \\
 & & & & \mid & \Gamma[?_i : \tau] \\
t,u,v & ::= & x & & & \\
 & \mid & \texttt{select}\ \overline{\texttt{case}\ t\ \texttt{of}\ \overline{br}} & \Delta & ::= & \varnothing \\
 & \mid & c\ \overline{t} & & \mid & \Delta[?_i = c\ \overline{t}] \\
 & \mid & ?_i & & & \\
 & \mid & \texttt{blkd}_{\mathcal{M}}\ t & br & ::= & c\ \overline{x} \rightarrow t
\end{array}
$$

FIGURE 1. Types, terms, environments and closures

3.1 Syntax

Figure 1 presents the syntax. A bar over an entity denotes a list. Variables are denoted by x and constructor names by c. Types are denoted by τ. A type is a list of constructors where each constructor is followed by a list of its arguments' types. Let $\mathcal{B}$ denote the type of booleans, $\mathcal{B} \equiv \{\texttt{true}, \texttt{false}\}$.

A term, denoted by t, u, v, is either a variable occurrence, a select-case expression, a constructor application or a meta variable occurrence. Select-case expressions consists of a list of ordinary case expressions. Each case expression has a list of branches, denoted by br. For the sake of simplicity, the patterns in the branches are restricted to the form $c\ \overline{x}$, i.e. a constructor followed by a list of variables. A constructor application is a constructor followed by a list of argument terms. The last term construction, $\texttt{blkd}_{\mathcal{M}}\ t$, is used internally by the algorithm and represents a term, t, whose further reduction is blocked by a set of meta variables, $\mathcal{M}$.

Environments, Γ, are either empty or another environment extended by a typing judgement for a meta variable. Closures, Δ, are either empty or another closure extended by an equality judgement for a meta variable. The right hand side of the equality judgement is restricted to be a constructor application.

3.2 Type Rules

We omit a formal presentation of type rules for terms in the language. They basically contain no surprises. However, an important detail is that we demand case expressions to have exactly one branch for each constructor in the data type. A reference implementation of the language and search algorithm has been written[6]. This also contains a type-checking algorithm.

We will use the notation $[x_1 : \tau_1, \ldots, x_n : \tau_n] \vdash t : \tau$ to denote the judgement that t is type correct and of type τ in the context where $x_1 : \tau_1, \ldots, x_n : \tau_n$.

3.3 Term reduction

We now turn to head reduction of terms, $t \twoheadrightarrow_{\Delta} t'$, in the presence of meta variables, presented in figure 2. The term which is reduced is assumed to be closed, i.e. all

$$\frac{t_i \twoheadrightarrow_\Delta c\ \overline{u} \quad \texttt{pick}\ c\ \overline{u}\ \overline{br_i} = v \quad v \twoheadrightarrow_\Delta v'}{\texttt{select}\ \overline{\texttt{case}\ t\ \texttt{of}\ \overline{br}} \twoheadrightarrow_\Delta v'}$$

$$\frac{\forall i.t_i \twoheadrightarrow_\Delta \texttt{blkd}_{\mathcal{M}_i}\ t_i'}{\texttt{select}\ \overline{\texttt{case}\ t\ \texttt{of}\ \overline{br}} \twoheadrightarrow_\Delta \texttt{blkd}_{\bigcup_i \mathcal{M}_i}\ \texttt{select}\ \overline{\texttt{case}\ t'\ \texttt{of}\ \overline{br}}}$$

$$\frac{}{c\ \overline{t} \twoheadrightarrow_\Delta c\ \overline{t}} \qquad \frac{\Delta \vdash ?_i = t}{?_i \twoheadrightarrow_\Delta t} \qquad \frac{\Delta \nvdash ?_i}{?_i \twoheadrightarrow_\Delta \texttt{blkd}_{\{?_i\}}\ ?_i}$$

FIGURE 2. Term reduction

variable occurrences are bound by a variable on the left hand side of a surrounding case branch. It may on the other hand of course contain *meta* variable occurrences. A reduction is indexed by the closure in which it takes place. The idea is to reduce a term outside-in until a meta variable is encountered whose value is not known, i.e. is not bound by an equality judgement in the closure. The reduction of a term always produces either a constructor application or a blocked term.

When reducing a select-case expression the presence of several alternative definitions is used to pick any one of them which allows the reduction to continue further. If the scrutinee term of one of the case expressions reduces to a constructor application, then, due to the restriction we imposed on the patterns, we know that the left hand side of one of its branches will match the term. The function `pick` is not described formally. It is assumed to, given a constructor, a list of terms and a list of branches, find the branch whose head on the left hand side matches the constructor and return the branches' body with the argument terms correctly substituted into it. The resulting term is then reduced. If, on the other hand, the scrutinees of all the case expressions reduce to a blocked term, then further reduction is postponed, and the case expression itself is marked blocked by the union of the blocking meta variables.

A meta variable occurrence is reduced to its value if it has an equality constraint in the closure. Otherwise, the term is marked blocked by the meta variable itself.

We will now illustrate the two non-standard aspects of the reduction beginning with meta variables and blocked terms. Let us define a macro for boolean conjunction.

$$x \wedge y \equiv \texttt{select case}\ x\ \texttt{of}\ \{\texttt{true} \to y,\ \texttt{false} \to \texttt{false}\}$$

Next, assume we want to reduce $?_1 \wedge ?_2$ in a closure, Δ, not containing any equality judgements for $?_1$ or $?_2$. Expanding the macro we see that the scrutinee of the only case expression is $?_1$, and $?_1 \twoheadrightarrow_\Delta \texttt{blkd}_{\{?_1\}}\ ?_1$. Since the scrutinee of the case expression reduces to a blocked term, we have $?_1 \wedge ?_2 \twoheadrightarrow_\Delta \texttt{blkd}_{\{?_1\}}\ ?_1 \wedge ?_2$. Hence, the term was only partially reduced and blocked by $?_1$ to indicate that the lack of knowledge of this meta variable's value stopped the reduction from proceeding further. Moreover, if the term consists of nested case expressions, the blocking

$$
\begin{array}{lcl}
\texttt{init}\ t\ \tau & = & (\varnothing[?_1:\tau], \varnothing, t') \\
 & & \quad \texttt{where}\ \ t[x:=?_1] \twoheadrightarrow_{\varnothing} t' \\
 & & \\
\texttt{step}\ ?_i\ c_j\ (\Gamma,\Delta,t) & = & (\Gamma[\overline{?_k}:\overline{\tau}_j], \Delta', t') \\
 & & \quad \texttt{where}\ \ \Gamma \vdash ?_i : \overline{c\ \overline{\tau}} \\
 & & \qquad\quad \Gamma \nvdash \overline{?_k} \\
 & & \qquad\quad \Delta' = \Delta[?_i = c_j\ \overline{?_k}] \\
 & & \qquad\quad t \twoheadrightarrow_{\Delta'} t' \\
 & & \\
\texttt{search}\ (\Gamma,\Delta,\texttt{true}) & = & [\texttt{compose}\ \Delta\ ?_1] \\
\texttt{search}\ (\Gamma,\Delta,\texttt{false}) & = & [] \\
\texttt{search}\ (\Gamma,\Delta,\texttt{blkd}_{\mathcal{M}}\ t) & = & \texttt{concatMap}\ (\backslash c_j \rightarrow \texttt{search}\ (\texttt{step}\ ?_i\ c_j\ (\Gamma,\Delta,t)))\ \overline{c} \\
 & & \quad \texttt{where}\ \ ?_i \in \mathcal{M} \\
 & & \qquad\quad \Gamma \vdash ?_i : \overline{c\ \overline{\tau}}
\end{array}
$$

$$
\begin{array}{lcl}
\texttt{compose}\ \Delta\ ?_i \ \mid\ \Delta \vdash ?_i = c\ \bar{t} & = & c\ (\texttt{map}\ (\texttt{compose}\ \Delta)\ \bar{t}) \\
\texttt{compose}\ \Delta\ ?_i \ \mid\ \Delta \nvdash ?_i & = & ?_i
\end{array}
$$

FIGURE 3. Definitions of init, step, search and compose

information is passed on to the top of the resulting term. For instance, applying the reduction rules we get $(?_1 \wedge ?_2) \wedge ?_3 \twoheadrightarrow_{\Delta} \texttt{blkd}_{\{?_1\}}\ (?_1 \wedge ?_2) \wedge ?_3$.

The other anomality is the parallel evaluation following from the presence of the select-case construction. We can define a parallel, symmetrical version of the conjunction.

$$
\begin{array}{l}
x \wedge_p y \equiv \texttt{select}\ \{\texttt{case}\ x\ \texttt{of}\ \{\texttt{true} \rightarrow y,\ \texttt{false} \rightarrow \texttt{false}\}, \\
\qquad\qquad\qquad\qquad \texttt{case}\ y\ \texttt{of}\ \{\texttt{true} \rightarrow x,\ \texttt{false} \rightarrow \texttt{false}\}\}
\end{array}
$$

Given the term $?_1 \wedge_p ?_2$, we see that the scrutinees of the two parallel case expression are $?_1$ and $?_2$ respectively. Both these reduce to blocked terms, $\texttt{blkd}_{\{?_1\}}\ ?_1$ and $\texttt{blkd}_{\{?_2\}}\ ?_2$, and the select-case term is consequently blocked by the union of the blocking meta variables. Thus $?_1 \wedge_p ?_2 \twoheadrightarrow_{\Delta} \texttt{blkd}_{\{?_1,\ ?_2\}}\ ?_1 \wedge_p ?_2$. In the case of parallel conjunction, if either of $?_1$ and $?_2$ are instantiated to `false`, then $?_1 \wedge_p ?_2$ will reduce to `false`.

3.4 Search Algorithm

Now assume that we have a term, t, and a type, τ, such that $[x:\tau] \vdash t : \mathcal{B}$. For this predicate over τ we want to find all elements, i.e. constructor terms, u, such that $t[x := u] \twoheadrightarrow \texttt{true}$.

A simple search algorithm is defined by the functions in figure 3. Given a predicate and its domain, i.e. a t and a τ, `init` produces the initial deduction state. The state consists of an environment, a closure and a partially reduced term. The initial

term is based on t where the free variable is replaced by $?_1$, and the initial environment declares $?_1$ to be of type τ. The `step` function takes a meta variable, a constructor and a state. For that state, the closure is extended by an equality judgement binding the meta variable to an application of the desired constructor with fresh meta variables as arguments. The environment is extended by type judgements for the new meta variables and the term is re-reduced in the new closure. So, the search algorithm is based on elementary steps which refine the data by one constructor at a time.

Given an initial state, the `search` function does a depth first search by applying the step function recursively. Before each step, the term of the state is examined. If it is equal to `true`, a solution has been found and the value is constructed by calling the `compose` function. The constructed term may however contain uninstantiated meta variables, corresponding to parts of the data which the reduction did not depend on. Hence, in general, a solution is a term containing constructor applications and (uninstantiated) meta variables. We will call such a term a partial constructor term (PCT).

If the term is instead `false`, a non-solution has been reached and the search is back-tracked. Finally, if the term is of the form $\texttt{blkd}_{\mathcal{M}}\ t$, then one of the blocking meta variables in $\mathcal{M}$, say $?_i$, is picked for refinement and the search is forked by one branch for each constructor of the meta variable's type. Thus, for a pair, t and τ, the call `search` $(\texttt{init}\ t\ \tau)$ will return a list of PCT:s making the predicate true.

The presented algorithm is not deterministic in how to choose a meta variable to refine when there are several of them blocking the term. Changing the strategy for picking a blocking meta variable can certainly have a large impact on the size of the search space. A natural choice is to keep a collection of meta variables for refinement and add new ones for each new reduction of the term. The collection of pending meta variables can e.g. be either a queue or a stack. One could also think of using a priority queue and trying to design some heuristics for assigning priorities to meta variables. However, coming up with good and general such heuristics seems difficult. Using the stack approach is vastly superior for some special applications, but the queue seems better in most cases. It also has the advantage that it treats the meta variables in a fair way. This is important when dealing with recursive functions and datatypes. Since, when using a queue, all blocking meta variables will be refined sooner or later, some problems will generate a finite search space rather than infinite.

3.5 Correctness of the Search Algorithm

Without thinking too much, one could find it natural to simply state the following regarding the correctness of the presented search algorithm. The search algorithm is sound if, for all constructor terms u in $\texttt{search}\ t\ \tau$, we have $t[x := u] \twoheadrightarrow_{\varnothing} \texttt{true}$. It is complete if the reverse holds. But, due to the presence of select-case expressions, the result of an evaluation might be different depending on which case expression is selected. One way to handle this indeterminism is to replace the reduction of

terms presented above with a version which takes every possible evaluation path into account. Let `mhn` t denote the set of terms that result when reducing t following all possible paths, i.e. all alternatives in the select-case expressions. We leave out a precise definition of `mhn`, hoping that the intention is clear.

Another detail which needs our attention is the fact that `search` produces a list of PCT:s, i.e. constructor terms that may contain meta variables. When a solution of the search contains a meta variable, say $?_i : \tau_i$, we should be able to replace $?_i$ by any constructor term $u : \tau_i$ and still have a solution. Hence, a single solution can represent a whole class of non-partial constructor terms. To reflect this, we introduce the notation $u \subseteq v$ to denote that, for the PCT:s u and v, the represented non-partial constructor terms of u are a (non-strict) subset of those of v, i.e. u is a specialisation of v.

Using `mhn` and $\subseteq$ we can now make a more precise formulation of the correctness. Let t and τ be any term and type such that $[x : \tau] \vdash t : \mathcal{B}$. The algorithm is sound if, for all PCT:s $u \in$ `search` $t\ \tau$, it holds that `true` $\in$ `mhn` $t[x := u]$. It is complete if, for all PCT:s u such that $(\texttt{mhn}\ t[x := u]) = \{\text{true}\}$, it holds that there is a PCT $v \in$ `search` $t\ \tau$ such that $u \subseteq v$.

We have not proved the soundness and completeness, but believe that the algorithm is correct in this sense. For predicates which can compute to both `true` and `false`, the soundness and completeness do not express anything. But, as stated before, the indeterminism in this language is only intended to give alternative definitions of well defined functions, such as a symmetric conjunction. It is up to the user to make sure that all alternatives in a term always result in the same value. This should however not cause much problem, since the parallel evaluation is supposed to be used in only a few functions, which can be written and proved well defined once and for all.

3.6 Implementation

We have made an implementation of the presented system. It consists of a compiler which accepts a fragment of haskell and a run-time system executing the search algorithm. This fragment to much extent corresponds to the language presented above, but with recursive functions and datatypes added. In the presented rules for term reduction, we chose to let the rule which blocks a select-case term not leave the blocking information in its subexpressions. This was in order to keep the presentation simple. But if this information is preserved, re-reduction of a term can be more economic. In the implementation, when a meta variable has been refined, only the parts of the term which are indeed blocked by that meta variable are re-evaluated.

The blocking meta variables are stored in a queue, as discussed above. Also, the search is parameterized by a search depth, which is in general needed when dealing with recursive datatypes. Iterated deepening is used to find small solutions before larger ones.

Adding more standard features of haskell such as higher-order functions and

gp*n*	finite algebraic groups, *n* is size of group
hp*n*	leftist, ranked, heaps, *n* is size of heap
gr*n*	directed, single edge graphs with paths from all to all edges, *n* is number of edges plus vertices
re*n*	regular expressions, checking an incorrect property, *n* is total number of constructors in generated data
ri*n*	valid pairs of type and term for reference implementation, *n* is number of constructors in type and term

TABLE 1. Abbreviations and descriptions of examples

classes is not problematic as long as the type of the data which is being constructed is first-order. We have recently developed a new compiler which reads the ghc external core language. The obvious advantage of this is that classes, patterns, guards and many other features come for free. We have however not rewritten the examples given in [6], so the code may appear a bit circumstantial.

4 EXAMPLES

We will now present a few examples where the system has been used to systematically generate data. Table 1 gives a short presentation of the examples which are referred to in the figures of this section. The full examples can be found in [6].

In the first three examples, the measurement of the size of the data is customized rather than based on the depth of the search (number of constructors). This was done by adding a size predicate to the property. We represent the size by natural numbers implemented as a recursive datatype. For non-linear data, such as trees, one needs to add the sizes of several subterms. Using an ordinary addition function defined by recursion on one of the arguments can lead to a generation of much larger data than intended. We work around this by again using the feature of parallel evaluation and defining a parallel version of addition. Using this, as soon as any part of a tree grows, the increase of the size is propagated to the top.

In order to make the examples a bit more vivid to the reader, we will give some more details of a couple of them. The example `hp` is about leftist, ranked heaps. The heaps are represented by ranked trees with natural numbers both as ranks and elements in the nodes.

```
data RTree = Empty
           | Node Nat Nat RTree RTree
```

The predicate `isHeap` decides whether a ranked tree is a leftist, ranked heap. It is defined in terms of three subproperties.

```
isHeap :: RTree -> Bool
isHeap x = wellRanked x >&< leftist x >&< hop x
```

The last conjunct is the heap order property. All three subproperties are defined recursively over the tree. The operator `(>&<)` is the parallel version of `(&&)`. In order to control the size of the heaps, we use the following property to generate them.

```
prop :: Nat -> RTree -> Bool
prop size x = sizeRTree x <= size >&< eltSizeOk size x
              >&< isHeap x
```

The number of nodes is calculated by `sizeRTree` using the parallel addition mentioned above, while `eltSizeOk` makes sure the element in each node is smaller than the given size.

The example `re` uses the following datatype for regular expressions:

```
data RE = Sym Nat
        | Or RE RE
        | Seq RE RE
        | And RE RE
        | Star RE
        | Empty
```

Symbols are represented by natural numbers. The function `accepts` checks whether a regular expression matches a list of symbols.

```
accepts :: RE -> [Nat] -> Bool
```

We add the auxiliary function `repInt` such that `repInt n k re` accepts between `n` and `k` sequences of symbols, where each sequence is accepted by `re`.

```
repInt :: Nat -> Nat -> RE -> RE
```

The search was done on the negation of the following property:

```
prop :: (Nat, Nat, RE, RE, [Nat]) -> Bool
prop (n,k,p,q,s) =
  (accepts (And (repInt n k p) (repInt n k q)) s)
  <==> (accepts (repInt n k (And p q)) s)
```

A counter example to this property is found after about one second using iterated deepening.

4.1 Lazy Instantiation

Figure 4 shows a comparison between using the system, i.e. the property directed lazy instantiation, versus a blind enumeration, both in order to generate all possible test data up to the size indicated by the suffixed number in the chart. The logarithm of the running times is used to fit several orders of magnitude into the

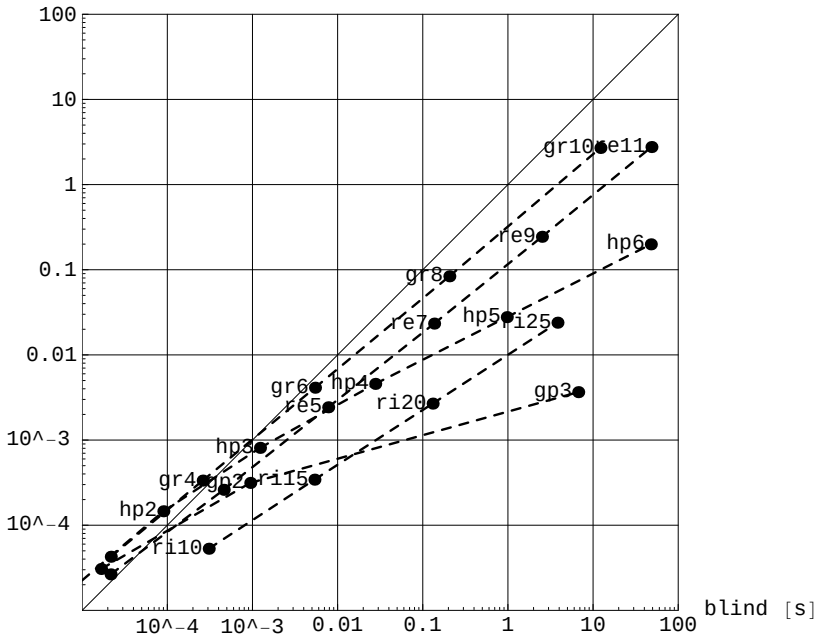

FIGURE 4. Lazy instantiation vs. blind enumeration

same plot. If a point is to the right of the diagonal, lazy instantiation is faster than blind enumeration.

For some of the examples, lazy instantiation becomes noticably faster as the size increases. The most successful ones are two to three orders of magnitude faster at the largest sizes included in the plot. For the less successful, the ratio grows rather slowly with size. It is obvious that some applications are more suitable than others for the lazy instantiation.

The main idea of our approach is to evaluate properties step-by-step while constructing the data. Hopefully this allows knowing the output of the predicate (`True` or `False`) when the data is still to large extent uninstantiated. In such situations it is clear that all further instantiations will result in the same output, since the evaluation does no depend on the uninstantiated parts. If the output is known at an early stage often enough, the resulting pruning of the search tree will be substantial and the speed gain of the enumeration will be noticable.

4.2 Parallel Evaluation

In figure 5, the comparison is between using and not using parallel evaluation. In this plot there are additional lines, consisting of shorter dashes. These lines connect runs which only differ by the order of certain conjuncts in the properties.

We can see that for some examples there is a permutation of the conjuncts, for

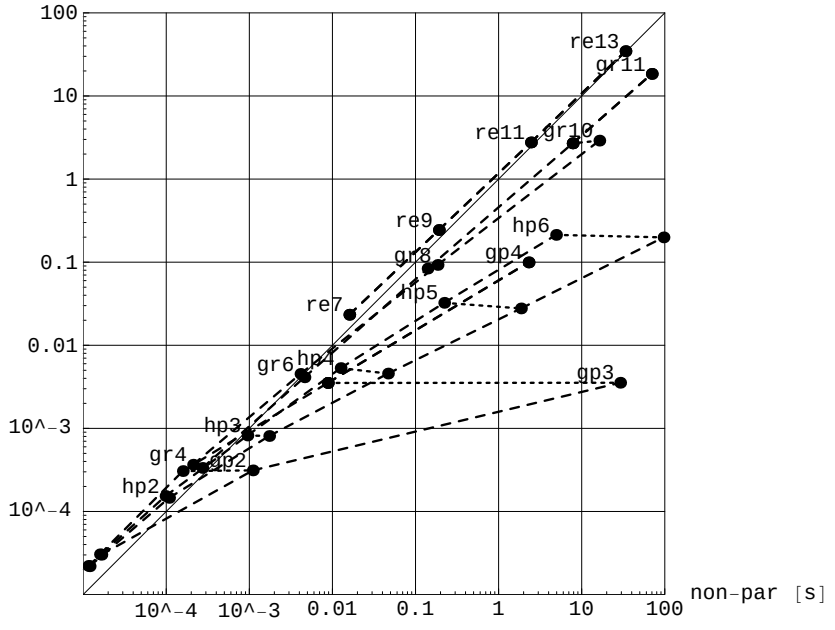

FIGURE 5. Non-parallel vs. parallel evaluation

which the non-parallel search works equally good, or even slightly better than the parallel search. On the other hand, for some of them there are also permutations where non-parallel search is substantially worse than parallel. The fact that the short dashed lines are almost horizontal however shows that the parallel search is not sensitive to the order of conjuncts, as expected. Hence, the main advantage of parallel evaluation seems to be its robustness – when used, one is not obliged to tweak the order of conjuncts in the property.

5 CONCLUSIONS AND FUTURE WORK

The presented examples indicate that using lazy instantation rather than blind generate -and-test may improve the speed of enumerating constrained data considerably. We have not yet conducted a thourough comparison between our approach and a random testing tool, such as QuickCheck. However, what we have seen so far indicate that many examples which can be solved using our approach are not tractable with random testing unless custom generators are added. Also, for some problems, like the regular expressions example, writing generators is too complicated and random testing simply fails. This aside, not having to write custom generators is in our opinion already an improvement.

In a typical program verification situation, such as the one presented in the introduction, we have a choice. Either we generate input data meeting the precondition and, for each such data, test the consequent of the implication. Or, we

try generating data which directly disproves the property, i.e. by involving the entire property in the search. One advantage of the latter is that this provides more guidance for the search. However, our experiments show that the effect of this extra guidance is very modest, at best increasing speed a few times. On the other hand, the former choice makes it possible to test programs written in another language, functional or non-functional. As long as the precondition is expressed in our system, a search can be performed that enumerates valid input data which can be exported to another language and used for testing.

Applying random testing directly on lazy instantiation does not make sense. A systematic search is necessary in order to benefit from the guidance. However, we have done some experiments on combining systematic and random search. The results show that this can sometimes lead to better performance and it would be interesting to look at this more closely.

One interesting improvement of the system could be to make the algorithm at any point detect if the current problem seperates into several independent ones. Consider e.g. a term such as $P(?_1) \wedge_p Q(?_2)$. Since the first and second conjunct depend on disjoint sets of meta variables, the problem could be separated in two. This way the instantiation of $?_1$ and $?_2$ would not be interleaved and the search space could shrink drastically. We have an experimental version of the system implementing this idea, but cannot yet report any empirical results.

The presented system does not cover generation of data containing primitives, such as strings or integers. We have done some experiments with instantiating primitive values via equality constraints. However, in all the presented examples we use recursively defined natural numbers instead of integers. This seems to suffice in many cases since we often look for solutions containing small numbers.

REFERENCES

[1] Sergio Antoy, Rachid Echahed, and Michael Hanus. Parallel evaluation strategies for functional logic languages. In *International Conference on Logic Programming*, pages 138–152, 1997.

[2] Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. In *ICFP '00: Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, pages 268–279, New York, NY, USA, 2000. ACM Press.

[3] M. Hanus. A unified computation model for functional and logic programming. In *Proc. 24st ACM Symposium on Principles of Programming Languages (POPL'97)*, pages 80–93, 1997.

[4] M. Hanus et al. Curry, an integrated functional logic language. Report, 2006.

[5] M. Hanus and P. Réty. Demand-driven search in functional logic programs. Research report rr-lifo-98-08, Univ. Orléans, 1998.

[6] F. Lindblad. Source code for examples including reference implementation. `http://www.cs.chalmers.se/~frelindb/tfp_suppl.tar`