Lazy Generation of Canonical Test Programs

Jason S. Reich, Matthew Naylor and Colin Runciman

Department of Computer Science, University of York
{jason,mfn,colin}@cs.york.ac.uk

Abstract Property-based testing can be a highly effective form of lightweight verification, but it relies critically on the method used to generate test cases. If we wish to test properties of compilers and related tools we need a generator for source programs as test cases.
We describe experiments[1] generating functional programs in a core first-order language with algebraic data types. Candidate programs are generated freely over a syntactic representation with positional names. Static conditions for program validity and canonical representatives of large equivalence classes are defined separately. The technique is used to investigate the correctness properties of a program optimisation and two language implementations.

Keywords: automated testing, SmallCheck, lightweight verification, compiler correctness, search-based software engineering.

1 Introduction

Testing, when used effectively, can reveal a wide variety of programming errors. For the time invested in the implementation of testing, it can return a large improvement in confidence of software correctness. For example, a suite of programs is often used (Partain, 1993; Dietz, 2008) for verifying a compiler's correct behaviour.

In property-based testing, program properties are defined in the host language as functions returning a Boolean. A property-based testing library then instantiates the arguments of these functions, searching for negative results. The QuickCheck (Claessen and Hughes, 2000) property-based testing library for Haskell uses generators that randomly select test values of the appropriate types. This approach relies on failure cases occurring frequently enough to appear in a random sample of a few hundred or a few thousand tests. SmallCheck and Lazy SmallCheck (Runciman et al., 2008) instead fully explore a bounded enumeration of all possible test values up to some size. This approach appeals to the *Small Scope hypothesis* (Jackson, 2012, page 15) that programming errors will appear for small data values.

[1] Source code available at `https://github.com/jasonreich/ProgGen`.

```
data Pro    = Pro (Seq1 Nat) Exp (Seq RedDef)
data RedDef = Lam  Nat   Bod
data Bod    = Solo Exp
            | Case Exp   (Seq1 Alt)
data Exp    = Var  VarId
            | App  DecId (Seq Exp)
data VarId  = Arg  Nat
            | Pat  Nat
data DecId  = Con  Nat
            | Red  Nat
data Alt    = Nat :→: Exp
```

Figure 1. Initial definition of our core language.

In this paper, we use Lazy SmallCheck to enumerate all small test programs that are *valid* (well-formed, well-scoped and well-typed, §2), *canonical* (of a regular form detailed in §3) and *terminating* (also §3).

Rather than directly constructing programs that satisfy these conditions, we freely generate abstract syntax trees and filter out those for which some required condition does not hold. With careful representation choices for the freely generated abstract syntax, a lazy and condition-driven approach to test generation can efficiently and effectively prune large classes of unwanted test programs.

2 Generating Valid Programs of a Core Language

Our Core Language We choose to work with a first-order core functional language with algebraic data types. In order to generate programs in this language, we first define a datatype for its abstract syntax, as in Figure 1.

A program *Pro cds e rds* consists of a single datatype definition represented as a sequence *cds* of one or more constructor arities, a main expression *e* to be evaluated and zero or more top-level value definitions *rds* whose applications are reducible. A top-level value definition *Lam ar b* is a lambda abstraction of arity *ar*. The body may be just a single applicative expression *Solo e* or it may be a case expression *Case e as* with alternatives for different constructions of the subject *e*.

Expressions are, as usual, recursively composed applications with either variables or zero-arity applications as leaves. Variable references are explicitly tagged: *Arg* for argument variables and *Pat* for pattern variables in alternatives. Applied references are also tagged: *Con* for constructors and *Red* for top-level names whose applications are reducible. These are all referenced by the natural-number positions of their definitions.

Free Generation Using SmallCheck (Runciman et al., 2008) we can now define functions to enumerate all values of these AST datatypes bounded by a given depth of construction. The *Serial* instances are defined in Figure 2.

The type *Seq a* is synonymous with the list type [*a*] but the depth-bound is the same for all elements of the list — a *Seq a* list bounded by depth *d* has at most *d* items, each of which has depth at most $d - 1$. The *Seq1* variant is for lists with at least one element.

It is often convenient not to count simple tags or tupling structures when determining the depth of a construction. The compositions with *depth* 0 are for that purpose.

Let's run the *Pro* series generator with increasing depth bounds, and count the number of programs generated.

```
> [length (series i :: [Pro]) | i ← [0 . .]]
 [0, 4, 3504, 27700575980220, ...
```

What do the four *Pro* values at depth one look like? These can be rendered as follows,[2] with the convention that arguments are renamed x,y,..., pattern variables p,q,..., constructors A,B,... and top-level functions f,g,....

```
data D = A          data D = A          data D = A          data D = A
> x                 > p                 > A                 > f
```

As these programs are freely generated from the abstract syntax type, they are as yet unconstrained by any static semantics. Only one of them is valid — the third one. At depth two there are already thousands of similar-looking *Pro* values, hardly any of which are valid. Beyond depth two our machines are overwhelmed by the task of enumeration.

Validity Test Only one of the programs generated at depth one was valid. The other three referred to *undefined* variables or functions. At greater depths another form of invalidity can occur: there may be *arity disagreement* between uses and definitions. We must avoid, or cut short, the work of generating such invalid programs.

We can define a predicate *valid*, as in Figure 3. The auxiliary functions *validr*, *valide* and *valida* test the validity of reducibles, expressions and applications respectively. The first two arguments to *valide* are the enclosing scopes for argument and pattern variables.

If we test the property $\lambda p \rightarrow \mathit{valid}\ p \implies \mathit{True}$, of the 3,504 syntactically generated programs at depth two, just 160 are found to be valid. So even this simple validity check greatly reduces the number of programs to be tested. But as things stand, we still have to generate a large number of invalid programs, only to reject them as test cases.

[2] Text in sans-serif indicates Haskell (host language) code. Code in the `typewriter` typeface represents the target *core* language.

```
instance Serial Pro where
  series = cons3 Pro ∘ depth 0
instance Serial RedDef where
  series = cons2 Lam ∘ depth 0
instance Serial Bod where
  series = (cons1 Solo ∪ cons2 Case) ∘ depth 0
instance Serial Exp where
  series = cons1 Var ∪ cons2 App
instance Serial VarId where
  series = (cons1 Arg ∪ cons1 Pat) ∘ depth 0
instance Serial DecId where
  series = (cons1 Con ∪ cons1 Red) ∘ depth 0
instance Serial Alt where
  series = cons2 (:→:) ∘ depth 0
```

Figure 2. The series generators for our initial syntactic representation.

```
  -- Test a program is well-scoped and arity consistent.
valid :: Pro → Bool
valid (Pro (Seq1 cons) m (Seq eqns)) = valide 0 0 m ∧ all validr eqns
  where
      -- Test a reducable is well-scoped and arity consistent.
    validr (Lam a (Solo e))               = valide a 0 e
    validr (Lam a (Case s (Seq1 alts))) = valide a 0 s ∧
      and [indexThen c cons (λp → valide a p e) | (c :→: e) ← alts]
      -- Test an expression is well-scoped and arity consistent, in context.
    valide a _ (Var (Arg v))      = a ≢ 0 ∧ v < a
    valide _ p (Var (Pat v))      = p ≢ 0 ∧ v < p
    valide a p (App d (Seq es)) = valida d (N $ length es) ∧ all (valide a p) es
      -- Test an application is well-scoped and arity correct.
    valida (Con c) n = indexThen c cons (λn'              → n ≡ n')
    valida (Red f) n = indexThen f eqns (λ(Lam n' _) → n ≡ n')
  -- Index an element from a list and apply predicate. Default to False.
indexThen :: Nat → [a] → (a → Bool) → Bool
indexThen (N i) xs f = (¬ ∘ null) xs' ∧ head xs'
  where xs' = map f (drop i xs)
```

Figure 3. Validity of positional programs.

Lazy Free Generation The problem of generating test cases that satisfy conditions was a large part of the motivation for *Lazy* SmallCheck (Runciman et al., 2008). This tool applies conditions to *partially defined* values. If a test value is sufficiently defined to allow a condition to be evaluated to True (or to False), then it is known from this single evaluation that *all possible refinements* of this test value will also satisfy (or fail to satisfy) the condition. If the partiality of a test value makes the condition undefined, the test value is refined at exactly the place needed for evaluation of the condition to proceed further.

In principle, Lazy SmallCheck might run *more* test cases than SmallCheck for the same condition — since it tests partial values as well as total ones. But in practice, where there is a structural condition that most tests do not satisfy, Lazy SmallCheck uses many fewer tests.

If we again test the property $\lambda p \rightarrow \mathit{valid}\ p \implies \mathit{True}$, but this time using Lazy SmallCheck, just 187 tests are needed to obtain the same 160 programs at depth two. That is just under 5% of the tests required under SmallCheck.

3 Canonicity

If we could only test a compiler using just two source programs, it would be a better test if the two programs really were quite distinct, not just insignificant variations of each other. The same argument applies even if we can use a large number of test programs. Resources are always limited. So we don't want to waste them by testing umpteen versions of essentially the same program.

We shall use several principles to define *canonical* programs. Each of these programs is a unique representative of a whole class of essentially equivalent programs. The principles of canonicity are discovered through the analysis of programs being generated.

3.1 Principles of Ordering and Complete Reference

The two programs below perform the same computation under the obvious isomorphism between their datatypes. The only difference between them is the *ordering* of constructors, function definitions and case alternatives.

```
data D  =  A | B D D               data D  =  A D D | B
f x y   =  case x of               f x     =  case x of
           A     -> y                         A p q -> p
           B p q -> B p (g q y)    g x y   =  case x of
g x     =  case x of                          A p q -> A p (g q y)
           B p q -> p                         B     -> y
> g (f A (B A A))                  > f (g B (A B B))
```

A canonical representative of both programs respects an ordering for each of these things. Assuming the standard, automatically derived instances of *Ord*

```
canonicalOrder (Pro (Seq1 cons) _ (Seq eqns)) =
     -- Non-strict ordering of constructor arities
   orderedBy (⩽) cons ∧par
     -- Strict lexicographic ordering of equations
   orderedBy (<) eqns ∧par
     -- Strict ordering of case-alternatives by constructor
   and [ orderedBy (<) [c | (c :→: _) ← alts]
       | (Lam _ (Case _ (Seq1 alts))) ← eqns]
orderedBy :: (a → a → Bool) → [a] → Bool
orderedBy f (x : y : zs) = f x y ∧ orderedBy f (y : zs)
orderedBy _ _            = True
```

Figure 4. Predicate for the ordering of constructors, equations and alternatives.

for our AST datatypes, a canonical ordering predicate for programs is given in Figure 4.

The orderings over equations and alternatives are irreflexive; we forbid duplicate definitions. The ordering over constructor arities is not; we permit more than one constructor of the same arity.

The following programs are also in direct correspondence. There is a duality so far as the roles of the constructors `A` and `B` are concerned, and the arguments of function `f` are flipped.

```
data D = A | B                    data D = A | B
f x y = case x of                 f x y = case y of
        A -> A                            A -> x
        B -> y                            B -> B
> f B B                           > f A A
```

So here is a further ordering requirement in canonical programs. Constructors of equal arity must be *first used* in the program in the same order as they are declared in the datatype. And function arguments must be *first used* in the function body in the order given by their argument positions.

Further, for any program that declares *unused* constructors, arguments or pattern variables there is a simpler equivalent program without them. In a canonical program, all constructors and arguments are used.

Finally, a program with *unused function definitions* also has a smaller equivalent without them. In a canonical program, all functions can be reached by a static call-chain from the main expression. See §3.6 for further discussion of dead code.

After we impose all these ordering and complete-reference conditions, we have just two programs at depth two, generated by Lazy SmallCheck as a result of 109 tests. And at depth 3, instead of an overwhelming number of programs, just 4,413 programs are produced as a result of 24,373,980 tests.

Unorderable Equations Consider the following programs that *do not* satisfy the equation-ordering condition.

```
data D = A | B A                   data D = A | B A
f x = B (g x)                      f x = B (g x)
g x = B (f x)                      g x = B (f x)
> f A                              > g A
```

In the current positionally-referenced representation, these programs have *no canonical form*. Reversing the equation ordering simply gives the other program. Our solution for now is to *limit the number of top-level definitions to two* and change the referencing scheme as follows. Within a top-level definition reference is either recursive or else it references the other top-level definition: *Self* and *Other*. Within the main expression, we keep positional naming. i.e. 0 and 1.

As both of these reference models can be implemented with Boolean values, the *Red* constructor is changed to hold *Bool* instead of *Nat*. The definition of *valid* also needs to be changed to account for the new referencing scheme. The ordering predicates work without modification.

3.2 Principle of Depth Balance

To reach a rich space of small test programs, we need to generate function bodies at around depth four or five. But we do *not* need datatypes with four or five constructors, each with four or five arguments! Nor do we need multiple high-arity function declarations.

At depth n, the default syntactic generators give between one and n constructors. The constructors and functions each have $arity \leq n$. Not only is this signature space far richer than we need to express interesting programs — LISP has taught us that — but also the depth limits largely prevent *uses* of these declarations from being generated anyway.

Therefore, mirroring the top-level *two function limit*, the number of constructor declarations and the arities of declarations are capped at two. This could be implemented using a further condition but another approach will be outlined in §3.4.

3.3 Principle of Caller/Callee Demarcation

Wherever there is an application of a defined function, there may be different ways to split work between caller and callee. A canonical program should make this split only in standardised ways.

Both caller and callee should do *something*. The caller must do something: it cannot just be the application of the callee to some of the caller's arguments (or else any application of the caller could more simply be an application of the callee). The principle of complete reference excludes many cases, but we also exclude as a body any application of a function to exactly the same arguments.

The callee must also do something: a function body cannot simply be one of the arguments (or else any application could be replaced by a subexpression). Again the principle of complete reference already excludes most cases, but we also exclude the identity.

Even in our original program representation, we had a form of caller/callee constraint: case expressions can only occur outermost in a function body. So the callee does the case distinction. In canonical programs, the caller computes the case subject: that is, a case subject is just an argument variable, and by the ordering principle, it must be the first argument.

This too could be implemented by a further condition, but we use another approach, as the following section explains.

3.4 Principle of Nonredundant Representation

It is pleasing that Lazy SmallCheck can prune away the 3,502 invalid or non-canonical programs of depth at most two by running only 109 tests, finally delivering for us the two interesting test programs. But the very high proportion of *Pro* values that fail the conditions does prompt a question: would a further change of representation enable us to generate fewer invalid or non-canonical programs in the first place?

We have already established that canonical case subjects are first arguments. So in our new representation the case subject can be omitted.

For a program to be valid, all uses of constructors or functions must match declared datatype and function arities. In a canonical program with complete reference, it follows that the datatype can be determined from the other parts of the program, and the arity of each function can be determined from its body. So instead of generating a datatype definition and function arities, and testing for valid and complete uses, we need only generate a main expression and function bodies.

The cap of two on the number of constructors and functions can also be encoded in the sequence representation types in programs, and in *Bool* index types for declarations. With function arities bounded by two a *Bool* index also suffices for argument variables. Figure 5 details the new representation.

Case-alternative patterns now reference constructors according to their position, doing away with the need for a separate ordering condition for alternatives.

The arity of functions can be deduced by finding the maximum argument in the function body. The datatype definition can be inferred by combining information about program constructor applications and the maximum pattern variable in constructor alternatives. Conditions are still used to prune away non-canonical programs that are not precluded by the nonredundant representation.

The change to a nonredundant representation dramatically reduces the number of tests required at each depth. At depth 3 (analogous to the previous representation's depth 4), only 25,393 tests are required to reduce a space of 2,371,256

```
data ProR   = ProR ExpR (Seq0'2 BodR)
data BodR   = SoloR ExpR  | CaseR (AltR, AltR)
data ExpR   = VarR VarIdR | AppR DeclIdR (Seq0'2 ExpR)
data VarIdR = ArgR Bool   | PatR Bool
data DeclIdR = ConR Bool  | RedR Bool
data AltR   = NoAltR      | AltR ExpR
```

Figure 5. Nonredundant representation of our core language.

programs to 11 canonical representatives. At depth 4, analogous to the previously unattainable depth 5, it takes 28,311,473 tests to find 423,582 canonical programs.

3.5 Principle of Live Computation

Most interesting functional programs are recursive. But some recursively defined functions can unproductively fail to terminate. For example, here are two programs generated at depth 4.

```
data D  =  A
f       =  g A
g x     =  case x of
           A -> f
> f
```

```
data D  =  A | B D
f       =  B f
g x     =  case x of
           A   -> x
           B p -> g p
> g f
```

To exclude programs such as the one on the left, we add the condition that any recursive applications are either beneath a constructor, or else descend into the construction of a recursive argument. At depth 3, this simple termination condition does not reduce the number of programs produced but it does reduce the number of tests required to 19,099. At depth 4, only 74,414 canonical programs are now produced after 20,550,413 tests.

This still leaves some non-terminating programs such as the one on the right. (View `D` as Peano numerals, `f` as infinity and `g` as a semi-test for finite numbers.) A far more sophisticated condition (e.g. Abel, 1998) would be needed to eliminate such programs yet allow useful recursion.

For now, we have decided to accept that some unproductive programs will remain. A more sophisticated condition would require significant extra machinery and adversely affect lazy pruning performance. However, property testing must allow for the possibility of an unproductive program.

3.6 Principle of Live Code

The following programs are among those generated at depth 3. They are indistinguishable in their execution as the B case alternatives are never used. Some form of data-flow analysis is needed to detect dead code.

```
data D = A | B         data D = A | B         data D = A | B
f x y = case x of      f x y = case x of      f x y = case x of
        A -> y                 A -> y                 A -> y
> f A B                        B -> x                 B -> A
                       > f A B                > f A B
```

Dynamic evaluation of candidate test programs, followed by a simple reachability analysis, detects dead code more accurately than reachability analysis alone. We must avoid unbounded computation arising from recursive applications, but to avoid unfolding all recursive calls would limit results too much. Our solution is *single-shot recursion*: on any call path we evaluate at most two applications of the same function.

The bounded evaluation traverses the abstract syntax tree in *normal order*, contrasting with the other *in-order* conditions. Validity checks can therefore be bypassed due to the use of Lazy SmallCheck's *parallel conjunction* operator. As validity is required for evaluation, a partial validity checker is integrated into the dead code checker.

Although the live-code condition supersedes the function-reachability and constructor-use of §3.1, it is still worth applying all these conditions. The combination of different traversal orders may prune failures sooner.

Eliminating programs with dead code results in another dramatic fall in tests; depth 3 requiring only 2,731 tests and depth 4 only 445,791 tests. Now just four canonical programs remain at depth 3. These are the constant A program and the following:

```
data D = A | B         data D = A             data D = A | B
f x = case x of        f x y = case x of      f x y = case x of
      A -> B                   A -> y                 A -> y
> f A                  > f A A                > f A B
```

The leftmost program could be interpreted as partial inversion with D as the Boolean type. Both other programs are partial conjunction, where A is True and B is False, with different inputs. Alternatively, these could be viewed as partial disjunction where A is False and B is True.

At depth 4, we have just 64 programs that satisfy all these principles of canonicity and validity.

4 Performance

So far, we have discussed performance abstractly, with regard to the number of tests to reach a set of desirable programs. In this section, we shall also consider

Table 1. Performance of non-redundant representation at depth 3.

Conditions	Execution time	Tests required	Remaining programs
Validity	2643ms	138,617	855
+ Ordering + Use	690ms	34,745	124
+ Caller/Callee	580ms	25,393	11
+ Live Computation	437ms	19,099	11
+ Live Code	72ms	2,731	4

execution time. All figures were obtained using GHC 7.0.3 on 2GHz dual-core PC with 4GB of RAM.

Table 1 shows performance figures when applying the various conditions at depth 3 of the non-redundant representation. The initial freely generated space contains 2,043,136 'programs'. Execution times are measured using the *Criterion* (O'Sullivan, 2011) benchmarking library, averaging 100 measurements and ensuring a 0.95 confidence interval. As each additional condition is applied, the number of tests required to reach a set of desirable programs falls. This trend is mostly mirrored by a fall in execution time. However, execution time does not fall quite as rapidly as the number of tests performed. The time per test lengthens as the number of conditions increases. In fact, the *mean execution time per test* increases by 38% from validity to the full suite of conditions for canonicity.

Enumerating all canonical programs at depth 4 takes approximately 15 seconds. At depth 5, it takes around 3 hours to produce the 310,003 canonical programs.

5 Applications

We use these canonical programs to investigate the correctness properties of language implementations and program optimisations. The first example produces a small program that exposes the differences between static binding and dynamic binding. The second investigates some correctness properties of compiler optimisations both in terms of semantic preservation and performance improvement.

5.1 Static vs. Dynamic Binding

Suppose we implement different semantics for our source language. One version uses *static binding*, evaluating arguments in the environment of the application call. The other uses *dynamic binding* where arguments are evaluated in the environment of the argument reference.

The generated programs are evaluated under each semantics up to a given maximum derivation-tree depth and the results are compared under equality. This property is defined as *prop_bind* in Figure 6. Testing discovers a small example program at depth 4, for which static binding and dynamic binding produce different results.

```
prop_bind :: Pro → Bool
prop_bind e = isJust static ∧ isJust dynamic  ⟹  static ≡ dynamic
  where static  = evalFor 1000 False e ≫= return ∘ forceResult 5
        dynamic = evalFor 1000 True e ≫= return ∘ forceResult 5
```

Figure 6. A mistaken equivalence between static and dynamic binding.

```
data D = A | B D
f = g A A
g x y = case x of
        A   -> B y
        B p -> g p x
> g f f
```

Under static binding, the program returns `B (B A)` as we would usually expect. However, under the dynamic binding semantics, the program returns `B A`. In the recursive call to `g`, the environment contains $\{x \mapsto p, y \mapsto x, p \mapsto A\}$ when variable `y` is referenced.

5.2 Optimisations on a Sestoft Abstract Machine

Sestoft (1997) details the derivation of several abstract machines of improving efficiency. These abstract machines evaluate expressions written in a core higher-order functional language. A simple transformation converts our core first-order language into a form that can be executed by the Sestoft Mark 2 abstract machine.

Our goal is to verify a simple program transformation that non-recursively inlines function applications. In this case, we wish to ensure not only *semantic equivalence* but also *optimisation of reduction steps*. These are formally defined as *prop_inline_sem* and *prop_inline_opt* respectively in Figure 7.

At depth 5, the semantic equivalence property is satisfied by all 310,003 canonical programs. However, the following counterexample is found for the optimisation property. If no inlining is performed then this program takes 44 steps to reduce to normal form. But if inlining is applied it takes 46 steps.

```
f x   = case x of
        A   -> B x
        B p -> g x p
g x y = case x of
        A -> f y
        B p -> x
> f (g A A)
```

```
prop_inline_sem :: ProR → Bool
prop_inline_sem p = isJust (haltState r0)  ⟹  haltState r0 ≡ haltState r1
  where r0 = (traceFor 1000 ∘ translate) p
        r1 = (traceFor 1000 ∘ translate ∘ opt_inline) p

prop_inline_opt :: ProR → Bool
prop_inline_opt p =  isJust (haltState r0)  ⟹  length r0 ⩾ length r1
  where r0 = (traceFor 1000 ∘ translate) p
        r1 = (traceFor 1000 ∘ translate ∘ opt_inline) p

translate :: ProR → SestExpr
traceFor :: Int → SestExpr → [SestState]
haltState :: [SestState] → Maybe SestExpr
```

Figure 7. Predicates for testing inlining transformation.

The reason is as follows. In the original, `g A A` is only evaluated once but after inlining it is evaluated twice. The shared evaluation of `x` in the body of `f` is lost.

6 Further Work

Verifying Canonicalisation We should like to verify that every interesting test program has a canonical equivalent. The program generating framework itself could be used to test the existence of canonical representatives for each *reasonable* program. Assuming that every program is represented by a canonical variant, we could write a function that transforms any given program into a canonical representative. We could check that the function satisfies this specification.

Increasing Coverage Each canonical test program represents a class of programs performing equivalent computations but with different naming, ordering or abstraction boundaries, or with redundant parts. Every valid core program has an equivalent representative, and in that sense every core-program computation is represented in generated tests. This technique has proved very successful in reducing the exhaustive space of test programs.

But what if some desired property of a compiler, or other program-processor under test, fails only when a program is in some way non-canonical? If only canonical programs are tested, such potential failures will go undetected. One solution is to attach a post-processor to the canonical program generator. Given each canonical program, the post-processor picks an equivalent at random, not forgetting the possibility of picking the canonical program itself.

Extending the Core Language The core language used in this paper lacks features found in other core representations of functional languages. For example, both *GHC External Core* (Tolmach et al., 2009) and *F-lite* (Naylor and Runciman, 2010) include primitive values and operations, (recursive) local definitions and higher-order functions.

The abstract syntax datatype, generator and validity checker could be extended to include these features. However, the search-space of generated programs would be greatly enlarged. Some further principles of canonicity would be essential for practical purposes.

Generalising the Framework Although we have explained principles of canonicity in terms of our core language, the ideas are quite generic. In almost every programming language, or other complex structural representation, there are choices of names or positions, orderings and divisions between units, that do not fundamentally alter the computations or structures being described. There is also the possibility of parts that are in some sense redundant. So similar techniques might be applied successfully to generate test examples in quite different formalisms.

7 Related Work

The automatic generation of compiler test cases has long been an area of interest. A survey from the late 1990s (Boujarwah and Saleh, 1997) discusses and classifies a range of techniques. The papers cited generally use advanced generating grammars to ensure that only "semantically correct" (valid) programs are produced. A few authors generate test programs freely over context-free or EBNF grammars but with the stated aim of testing a compiler's syntax checker.

For testing functional programs, the QuickCheck work, starting with the award-winning paper at ICFP 2000 (Claessen and Hughes, 2000) has been hugely influential. QuickCheck is a library for property based testing based on the definition of type-based generators for *random* test values. A recent paper (Palka et al., 2011) describes the use of QuickCheck to generate random lambda terms for compiler testing. De Bruijn (1972) indexing is used to avoid problems of equivalence up to renaming. Aside from the use of random lambda terms, as opposed to exhaustively enumerated small equational programs, another significant difference from the approach reported here is that Palka et al. rely on a generating context including the signatures of pre-defined functions.

Other functional-programming researchers have looked into program enumeration. For example, Katayama (2007) enumerates typed lambda terms. The motivation is to provide exhaustive search for appropriately typed expressions during program synthesis. Katayama highlights the advantages of a de Bruijn representation, and the importance of excluding "equivalent expressions which cause redundancy in the search space and multiple counting". In this work too, the generator generates terms applying a library of pre-defined functions, and one of the equivalance-avoiding techniques is to apply known simplification laws for these functions. But the discussion notes a need to do more to eliminate duplicate or equivalent solutions.

8 Conclusions

Our aim has been to enumerate valid and canonical programs for the purposes of compiler verification. We have shown that large spaces of freely generated terms can be pruned effectively to yield 'interesting' programs. Exploration of the search space indicates that Boolean programs such as partial inversion, conjunction and disjunction appear at depth 3. Canonical programs involving Peano numerals (e.g. addition) and lists (e.g. append) emerge at depth 6. This paper roughly mirrors the process by which the principles were discovered.

First, an algebraic data type for the abstract syntax is defined and a free generator is created using (Lazy) SmallCheck combinators. Through the observation of the resulting programs, conditions are defined to eliminate invalid and non-canonical programs. The representation is reconsidered to eliminate the redundancy that allows the invalid and non-canonical terms to arise. And so the procedure repeats. Implementation details are occasionally reevaluated to account for the interactions of the different conditions.

We have discovered several principles of canonicity for our first-order language and dramatically reduced the problem size. We expect that further investigation of the currently generated programs will reveal new principles of canonicity or more restrictive variations of existing conditions.

We applied our testing technique to investigate several properties relating to evaluation, compilation and optimisation. The results obtained are encouraging. However, more complex applications motivated our work: we wish to investigate the correctness and improvement properties of supercompilers. It remains to be seen what further refinements of our technique will be needed to succeed in this goal.

Acknowledgements The authors would like to thank Michael Banks, Emma Maksymowicz and Chris Poskitt for their invaluable proof reading. They also extend their gratitude to the programme committee for their constructive feedback on earlier drafts. This research was supported, in part, by the UK's Engineering and Physical Sciences Research Council through the Large-Scale Complex IT Systems project, EP/F001096/1.

Bibliography

Abel, A.: foetus — termination checker for simple functional programs. URL: `http://www2.tcs.ifi.lmu.de/~abel/foetus.pdf` (1998)

Boujarwah, A.S., Saleh, K.: Compiler test case generation methods: a survey and assessment. Information & Software Technology 39, 617–625 (1997)

de Bruijn, N.G.: Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. Indagationes Mathematicae 75(5), 381–392 (1972)

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. pp. 268–279. ICFP '00, ACM (2000)

Dietz, P.F.: The GCL ANSI Common Lisp test suite. URL: `http://en.scientificcommons.org/42309664` (2008)

Jackson, D.: Software Abstractions: Logic, Language and Analysis. MIT Press, Revised edn. (2012)

Katayama, S.: Systematic search for lambda expressions. In: Trends in Functional Programming Volume 6, pp. 111–126. TFP2005, Intellect Books (2007)

Naylor, M., Runciman, C.: The Reduceron reconfigured. In: Proceedings of the 15th ACM SIGPLAN international conference on Functional Programming. pp. 75–86. ICFP '10, ACM (2010)

O'Sullivan, B.: The criterion package, v0.5.1.1. URL: `http://hackage.haskell.org/package/criterion` (2011)

Palka, M.H., Claessen, K., Russo, A., Hughes, J.: Testing an optimising compiler by generating random lambda terms. In: Proceedings of the sixth IEEE/ACM Workshop on Automation of Software Test. pp. 91–97. AST '11 (2011)

Partain, W.: The nofib benchmark suite of Haskell programs. In: Functional Programming, Glasgow 1992. pp. 195–202. Workshops in Computing, Springer-Verlag (1993)

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. Haskell '08, ACM (2008)

Sestoft, P.: Deriving a lazy abstract machine. Journal of Functional Programming 7, 231–264 (1997)

Tolmach, A., Chevalier, T., The GHC Team: An external representation for the GHC Core Language (for GHC 6.10). URL: `http://www.haskell.org/ghc/docs/6.10.4/html/ext-core/core.pdf` (2009)