Validating Definitional Interpreters Using Property-Based Testing

Automated Validation of Definitional Interpreters RP Q4 2020 / 2021

Alexandru Radu Moraru Casper Bach Poulsen Cas van der Rest

Delft University of Technology
Research Project - Q4 2020 / 2021

ABSTRACT

This paper presents an evaluation of different generation methods of input expressions to definitional interpreters. We compare three different ways of generating expressions of a specified algebraic data type. The approaches that we describe are QuickCheck, SmallCheck and a uniform generation technique, as laid out in their original papers. Subsequently, we illustrate some of the advantages and pitfalls that ought to be considered when using each of the aforementioned approaches. Experimental evaluation shows that the uniform generation method can provide promising results at the expense of time. Compared with this, the QuickCheck and SmallCheck off-the-shelf generation methods are time-efficient, yet lack precision when it comes to generating well-typed terms.

1 INTRODUCTION

The aim of this research project is to tackle the issue of automatically validating whether two definitional interpreters are equivalent [1]. To provide a better understanding of the topic, consider the following interpreters written in Haskell:

```haskell
data Expr = Num Int | Mul Expr Expr

eval :: Expr → Int
eval (Num i)     = i
eval (Mul e1 e2) = interp e1 * interp e2

evil :: Expr → Int
evil (Num i)     = i
evil (Mul e1 e2) = interp e1 * interp e1
```

Listing 1: *Two non-equivalent interpreters*

Now, a simple counter-example to prove that the evaluators from Listing 1 are not equivalent is the case when the expressions `e1` and `e2` are not identical. While finding such a counter-example is trivial for these interpreters, doing this for any two arbitrary programs is known to be an undecidable problem [2].

This research project does not aim to find a way to decide the functional equivalence problem, but rather it tries to explore the effectiveness of alternative solutions that can realistically check interpreter equivalence in a reasonable time. Finding such alternative solutions would be worthwhile because of the sheer amount of fields where they could be applied. One such example would be in the context of the Concepts of Programming Languages second-year course at TU Delft. In this course, the students have weekly assignments in which they implement various such interpreters. Currently, to test the students' solutions, the course staff employs the use of unit tests that explicitly check the desired functionality. However, writing such tests is tedious and requires extensive manual work, therefore automating this process would significantly reduce the time needed to maintain the test suite.

The main research question that we tackle in this paper is:

How effective are property-based testing frameworks, such as QuickCheck [3] *or SmallCheck* [4] *, for automatically generating input expressions to definitional interpreters?*

Consequently, we present an exploration of how to create valid[1] useful input expressions for definitional interpreters. To this end, the work of Claessen et al. [5], and the work of Pałka et al. [6] provide a good starting point for generating useful test data. Nonetheless, to ensure that generating such terms amounts to good results, a thor-

[1]In this context, *valid* entails well-typed input expressions.

ough evaluation needs to be performed. As a consequence, we consider various metrics when measuring effectiveness. One heuristic that we will use is simply the number of faults found. In addition, the time and space efficiency of the aforementioned generation methods will also be evaluated. Alternatively, one could measure the number of tests needed to find a known fault, yet this also poses the problem of randomness in test data which requires a discussion of its own.

1.1 Contributions

The main contributions of this paper are:

- A comparative study between the efficiency of QuickCheck's (Section 3.1) and SmallCheck's (Section 3.2) off-the-shelf approaches for generating input expressions to definitional interpreters.
- An evaluation and implementation of the Claessen et al. approach [5] for generating well-typed terms, which satisfy a given predicate and that follow a uniform distribution (Sections 3.3 and 4).
- A comparison between the effectiveness of the aforementioned approaches by taking into account time and space measurements together with other evaluation metrics (Section 4).
- A public repository which includes the codebase that we used to perform the evaluation of the different generation methods, the subsequent results that we obtained and the corresponding log files.

1.2 Road-map

The paper starts by surveying the relevant concepts used in this research in Section 2 and by describing the methodology in Section 3. Section 4 describes the experimental setup and showcases the found results, whereas Section 5 contains relevant discussions and proposes ideas for future work. Section 6 will address some of the ethical implications of this project while also describing reproducibility aspects. Finally, Section 7 contains an overview of the paper and concludes the research.

2 BACKGROUND

In order for two programs to be equivalent, on any given input both ought to produce the same output. In most programs, the space of possible input expressions is infinite, therefore it is infeasible to check all possible variants. Moreover, a program might never halt on a given input. Both of the aforementioned intricacies are, in short, the reasons why the functional equivalence problem is undecidable. Consequently, the focus of this paper shifts to testing two programs on a finite set of possible input expressions that can exercise most execution paths of the interpreters under test.

2.1 Property-based Testing

An important aspect of this research project is property-based testing. This type of testing has become increasingly popular, especially in the functional programming world [7]. This is, to some extent, mostly due to QuickCheck, which is greatly utilized within the Haskell community, being the most intensively used testing package and among the top 10 packages of any kind on hackage[2] [8]. As the name of *property-based testing* implies, it makes use of properties that can hold for the program under test. For example, the addition laws for integers can be expressed as properties, which, in turn, can be verified with QuickCheck or SmallCheck. To this end, Listing 2 showcases the properties for the commutativity and associativity laws of integers.

```
prop_commutativity :: Int → Int → Bool
prop_commutativity a b =
    a + b == b + a

prop_associativity :: Int → Int → Int
    → Bool
prop_associativity a b c =
    a + (b + c) = (a + b) + c
```

Listing 2: *Addition laws expressed as properties*

When testing these properties, QuickCheck will automatically generate random `Int` values for each test that it creates and repeatedly check whether the properties hold. SmallCheck, on the other hand, will exhaustively generate `Int` values, gradually increasing the complexity of the respective inputs until reaching the size specified by the programmer. In both cases, if any counter-example is found that

[2] `https://hackage.haskell.org/package/QuickCheck`

invalidates a property, further testing will stop and the problematic input will be displayed. Otherwise, a message will be shown which indicates that the properties held true for a specific number of tests. By default, the number of tests is 100, yet the programmer can modify this manually.

While QuickCheck and SmallCheck provide built-in code for generating most of Haskell's predefined types, when it comes to user-defined types, the programmer is tasked with doing this by using library functions. As a consequence, the first step that needs to be taken to tackle this research project is to define simple grammars, as shown in Listing 3. In addition, this also means writing out the required functions that are needed to generate expressions of the needed type. QuickCheck requires ADTs[3] to be an instance of `Arbitrary`, whereas SmallCheck requires them to be an instance of `Serial`.

```haskell
data Expr = Val Int
  | Add (Expr, Expr)
  | Sub (Expr, Expr)
  | Mul (Expr, Expr)
  | Div (Expr, Expr)

data Expr = Val Bool
  | And (Expr, Expr)
  | Or  (Expr, Expr)
  | Not Expr
```

Listing 3: *Arithmetic & booleans ADTs*

Afterwards, the next step is to define a couple of correct equivalent interpreters for each of the aforementioned grammars. Moreover, in both cases, faulty interpreters which contain small errors are also needed. Following this, properties for each grammar are defined that can check the output of the two known equivalent interpreters and the output between one correct and one incorrect interpreter. These properties can be written as such:

```haskell
prop_interp :: Expr → Bool
prop_interp e = interp1 e == interp2 e
```

Listing 4: *Property for checking interpreter equivalence*

With the above-defined grammars, there is no need to worry about expressions being of a particular type since they contain only one, namely `Int` in the case of arithmetic expressions and `Bool` in the case of boolean expressions. Consequently, all generated expressions are type-correct and would not produce an error when being interpreted. Unfortunately, the same cannot be said for more complex grammars, such as the ones which contain if-then-else expressions, lambdas, function applications and much more. In such instances, brute-forcing random generation to obtain well-typed terms skews the distribution towards trivial test cases. This, in turn, means that the approach which we described so far to generate user-defined data needs to be modified to produce well-typed terms.

2.2 Generating Well-Typed Terms

A possible technique that can help with generating well-typed terms was presented in the work by Claessen et al. [5]. This paper describes an approach for deriving test data generators for a specified algebraic data type. Furthermore, the test data is uniformly distributed over the required values of a specified size. An interesting feature of this paper is that the test data which is generated also satisfies a predicate specified by the programmer. To this end, some predicates that are useful for the problem of determining program equivalence are to check that a given input is well-typed or to ensure that a term is of a specific type.

This approach focuses on uniformly generating data of a specific size, which in this case is understood as the number of constructors found in a term. For example, considering terms of the previously mentioned arithmetic ADT, `Val 0` has size 2, whereas `Add (Val 1, Val 2)` has size 8. To facilitate this, the authors of the paper build upon some of the work by Duregård et al. [9] by utilizing an indexing function which maps integers to values of a specific size and type. More formally, this can be described in the following mathematical notation:

$$\text{index}_{S,k} : \{0...|S_k| - 1\} \to S_k$$

where S is the data type that we are trying to generate terms in, k is the size of a term that we are seeking, $|S_k|$ means the cardinality of that set, and S_k represents the set of values of type S and size k. Furthermore, the authors define a Haskell generalized algebraic data type, as shown in Listing 5. This GADT serves as a uniform representation of a *space* of values of a specified type from which we can sample random terms.

[3]Throughout this paper, we refer to ADT as short for an algebraic data type.

```
data Space a where
  Empty :: Space a
  Pure  :: a → Space a
  (:+:) :: Space a  → Space a → Space a
  (:*:) :: Space a  → Space b → Space (
    a, b)
  Pay   :: Space a  → Space a
  (:$:) :: (a → b) → Space a → Space b
```

Listing 5: *The Space GADT*

Besides this, the paper also lays out most of the implementation details needed to generate random uniform data. For example, one of the most important functions is:

```
uniform :: (a → Bool) → Space a → Int →
     Gen a
```

which takes a predicate, an algebraic representation of a subset of values of type `a` and an `Int`. This function is useful because it returns values of the specified type and size that satisfy the given predicate. Moreover, to speed up computations, whenever a value that does not satisfy the predicate is encountered, the space of possible values is further reduced to narrow down *correct* terms.

One of the main advantages of this approach is that one can start generating terms almost right away with just a little effort required by the programmer. More specifically, the programmer needs only to define the GADT spaces for the required ADTs before being able to generate expressions. Unfortunately, since this method closely follows the topic of functional enumeration, as laid out in previous papers [4], [9], time-efficiency becomes a concern when generating terms of larger sizes. Since some of the interpreters require more complex expressions (which implies terms of larger sizes) in order to find faults, this issue can severely hinder the performance of the generation method. Additionally, a limitation to this approach would arise when choosing a predicate that is rarely or never satisfied. In the latter case, there are no terms that can be found, therefore the functions that are searching for correct expressions will never terminate.

2.3 Conclusions

While the above-mentioned approaches have their own merits, an issue that arises is to determine which one is the most suitable for the given problem. Consequently, we will later explore how these methods can be used to generate well-typed input expressions for definitional interpreters. However, while using *valid* test data is the most intuitive approach when it comes to testing, ill-typed input expressions can also reveal faults in the interpreters. A choice with regards to the percentage of test cases that are type-correct needs to be made.

3 METHODOLOGY

In this section, we present a detailed explanation of the method that we followed to explore the relative merit of each generation method. For the general setup, several ADTs were defined for which we wrote both correct and faulty interpreters. Following this, we wrote properties for each interpreter. The property that was always used was to check that the output between a given pair of interpreters is the same (see Listing 4). However, for brevity, we mainly show detailed implementation steps and examples for the arithmetic ADT presented in Section 3 and leave the interested reader to explore the rest of the source code in the public GitHub repository.

3.1 Using QuickCheck

The main task after writing the QuickCheck generator for the arithmetic ADT, as defined by the `arbNaiveExpr` function (Appendix A, Listing 11) is to compare the interpreters by using QuickCheck properties. When doing this to check program equivalence between two equivalent interpreters, we are prompted with the following message:

```
Main> quickCheck prop_correct_interp
+++ OK, passed 100 tests.
```

However, when comparing a correct interepreter with a faulty one, QuickCheck can[4] present the following message:

```
Main> quickCheck prop_faulty_interp
Falsified (after 5 tests):
    (-2) - (-1) * 4 + 3 * 1
```

Indeed, when running this example manually on both interpreters, they produce different outputs. More specifically, the correct one gives 5, whereas

[4]The word *can* is used because of randomness since it is not guaranteed that the same failing test will be shown.

the incorrect one gives 6. While this approach illustrates a failing test case, it is fairly difficult for the programmer to immediately infer the place in which the error was introduced. In this instance, the error rests in the `Add` case of evaluating subexpressions (as seen in Listing 6), yet the counter-example illustrated by QuickCheck includes all possible operations within the arithmetic ADT.

```
interp (Add left right) =
  case interp left of
    Left err   → Left err
    Right valL →
      case interp left of  -- error here
        Left err   → Left err
        Right valR → Right (valL + valR)
```

Listing 6: *Faulty addition case in arithmetic ADT*

Incidentally, when using QuickCheck to generate terms within a more complex grammar (as shown in Listing 13, Appendix B), the percentage of well-typed terms is considerably low. This makes it extremely inefficient and furthermore difficult to actually find faults in the interpreters. When using QuickCheck's built-in functions to monitor test data and control the tests that will be used by providing a pre-condition (namely that of an expression to be well-typed), we notice that only 19 are type-correct, whereas 1000 were discarded. Moreover, a breakdown of the depth[5] of well-typed expressions shows that only a small fraction of terms have more complex structures.

```
Main> quickCheck prop_interp
*** Gave up! Passed only 19 tests;
1000 discarded tests:
74% 1
16% 2
 5% 3
 5% 4
```

Listing 7: *Example of a possible distribution of QuickCheck generated well-typed test cases per depth*

The results shown in Listing 7 are one of the main reasons for the need to explore different frameworks or methods for generating input expressions other than just QuickCheck. For reference, the term `If (Gt (EInt 17, EInt 42), (EInt 1, EInt 2))` has depth 3. As we hinted before, even such simple expressions are rarely ever generated, therefore errors hidden within such cases have a low probability of being found.

3.2 Using SmallCheck

Similar to QuickCheck, SmallCheck also requires the programmer to define a set of functions for it to generate data of a specified type (Appendix A, Listing 12 illustrates the SmallCheck generator for the arithmetic ADT). Once that is done, defining properties is identical to the way of defining properties in QuickCheck. When considering the same ADT as in the previous subsection, SmallCheck gives the following message when checking program equivalence between a correct interpreter and a faulty one:

```
Main> smallCheck 4 prop_faulty_interp
Failed test no. 12.
there exists 1 + 0 such that
  condition is false
```

Listing 8: *Checking arithmetic interpreter equivalence with a SmallCheck property of depth 4*

Compared to the counter-example reported by QuickCheck, SmallCheck provides more readable test cases. Consequently, in this instance the programmer can easily conclude that the problem rests within the `Add` case of interpreting arithmetic expressions. This is, to some extent, a direct consequence of SmallCheck's main design motivation: *"If a program does not fail in any simple case, it hardly ever fails in any case. A successful test-run using our tools can give exactly this assurance: specified properties do not fail in any simple case"* [4].

Nonetheless, the above-mentioned quote should not always be taken as an absolute certainty. For example, checking the same property as before, yet with a depth of 3, gives a false sense of security:

```
Main> smallCheck 3 prop_faulty_interp
Completed 9 tests without failure.
```

Listing 9: *Checking arithmetic interpreter equivalence with a SmallCheck property of depth 3*

This demonstrates that careful consideration is needed when choosing the depth parameter for the `smallCheck` function. A depth that is too low can lead to few test cases being generated, which can amount to false negatives. On the other hand, a depth that is too big can generate considerably many test cases, which can result in long runtimes.

[5]By *depth*, here we refer to the maximum size of any branch within the expression. For example, `Val 7` has depth 1, whereas `Add (Val 1) (Add (Val 2) (Val 3))` has depth 3.

3.3 Using Well-Typed Random Data

Using the approach outlined in Section 2.2 for generating expressions in the arithmetic ADT produces similar results to SmallCheck, in the sense that similar counter-examples are reported to the programmer. For reference, the spaces for expressions that were needed for the generator are depicted in Listing 10.

```
spExpr = Pay (spAdd :+: spSub :+:  spMul
    :+: spDiv :+: spVal)
spAdd  = Add :$: (spExpr :*: spExpr)
spSub  = Sub :$: (spExpr :*: spExpr)
spMul  = Mul :$: (spExpr :*: spExpr)
spDiv  = Div :$: (spExpr :*: spExpr)
spVal  = Val :$: spInt

spInt :: Space Int
spInt = Pay (Pure 0 :+: (succ :$: spInt))
```

Listing 10: *Defined spaces for the arithmetic ADT*

Since the arithmetic ADT requires integers in the leaf constructor (`Val`) to actually evaluate different operations (and to also ensure creation of finite expressions), a space for `Int` values also needs to be defined in order to generate terms. Moreover, to ensure that the algorithm terminates, `Int` values should also be attributed a cost. The choice that we make in this paper is to attribute a cost of 1 for each successor of `0`. For example, `0` has cost 1, whereas `1` has cost 2. However, supposing that the arithmetic ADT would have special operations solely on negative numbers, the above-mentioned choice for `Int` spaces would fail to catch any errors whatsoever. While solvable, this issue proves that choices for the implementation of spaces need to be backed up with a thorough understanding of the ADT that one wants to generate terms in.

4 RESULTS

This section showcases different results that we obtained when evaluating the generation methods mentioned within the previous sections. More specifically, we present a breakdown of the number of tests needed to find faults for each method and further provide more insight into the time and space measurements of the uniform generation approach. Throughout this section, we refer to a ”test run” as the execution of 100 tests (for QuickCheck and uniform generators). Moreover, consider the following notations when reading the results:

`QC` $\longrightarrow$ QuickCheck generation of well-typed terms

`SC` $\longrightarrow$ SmallCheck generation

`UG` $\longrightarrow$ Uniform generation of well-typed terms

4.1 Fault Detection

Tables 1 and 2 report the average number of tests (from ten test runs) needed to find a fault in each pattern match case of the interpreters for the boolean and arithemetic ADTs, respectively. Not surprisingly, all methods managed to find the faults within a reasonable time in each test run. The difference in the number of tests needed for each method directly stems from their respective implementation details. For more results regarding fault detection with `UG` on the previously mentioned ADTs, we ask the interested reader to consult Appendix C.

Method	`And`	`Or`	`Not`
`QC`	5	7	5
`SC`	9	16	9
`UG`	3	2	2

Table 1: *The average number of tests needed to find a fault in each pattern match case for the boolean ADT.*

Method	`Add`	`Sub`	`Mul`	`Div`
`QC`	3	3	4	3
`SC`	12	19	33	37
`UG`	4	5	6	3

Table 2: *The average number of tests needed to find a fault in each pattern match case for the arithmetic ADT.*

For the conditional ADT (Listing 13, Appendix B), we used the following test suite to evaluate how effective the three generation methods were when it came to detecting faults:

a. Substraction instead of addition for `Add`

b. Addition instead of multiplication for `Mul`

c. No negation applied for the `Not` operation

d. Apply `And` instead of `Or`

e. Apply `Or` instead of `And`

f. `If` always returns the `True` branch

g. `If` branches are swapped

h. `Eq` operator is always `True`

i. Apply ”<” instead of ”>” for `Gt`

Subsequently, we present the results for the conditional ADT by illustrating in tables 3 and 4 the average number of tests each generation method took to find a fault and by indicating the number of test runs that eventually found a fault. For this analysis, we used depth 5 for SmallCheck and utilized terms of size 5 for the uniform generation method. We explicitly omit results for faulty interpretation of `App` and `Lambda` constructors since it was not even possible to generate such terms with SmallCheck or the uniform generator when using strings for variable names.

	a	b	c	d	e	f	g	h	i
QC	-	-	4	5	-	-	-	2	3
SC	20	25	23	355	699	2570	1179	235	859
UG	10	12	7	3	-	-	-	-	-

Table 3: *#Tests to find faults in the conditional ADT*

	a	b	c	d	e	f	g	h	i
QC	-	-	7	1	-	-	-	1	3
SC	10	10	10	10	10	10	10	10	10
UG	10	10	10	10	-	-	-	-	-

Table 4: *#Test runs that detected faults in the conditional ADT*

The results for the conditional ADT in the tables above illustrate interesting behaviours. Firstly, QuickCheck was not able to detect faults in all ten runs. Secondly, the uniform generation method failed to detect any faults for the (e-i) errors. This is because terms that contain complex `If`, `Gt`, `Or` or `Eq` structures have a size of at least 7, and generating just one term of size 6 takes more than 80 seconds (as shown in Figure 1). For reference, Table 5 showcases examples of terms of sizes 2 through 6 that were generated. This, together with the previous results, point out the limitations of using the uniform generation method and indicate that improvements are needed for this to be realistically applied to grammars of various complexities.

Size	Example 1	Example 2	Example 3
1	-	-	-
2	True	False	1
3	!True	!False	2
4	!(!True)	!(!False)	3
5	1 * 1	!(!(!False))	False || True
6	!(!(!(!True)))	1 < 2	!(1 == 1)

Table 5: *Examples of terms of different sizes for the conditional ADT.*

For experimental purposes, we also performed an analysis on a generator that combined the QuickCheck and the uniform generator by using relative weights to control which generator to be chosen with each test case. In doing this, we used the same test suite as before but experimented with different generator weights. Since the results did not improve, we chose to omit them here, but refer the interested reader to Appendix D.

4.2 Time & Space Needed to Generate Terms

To see exactly how much the term size affects the overall run time and space consumption, we evaluated these metrics on the conditional ADT and reported the findings in figures 1 and 2.

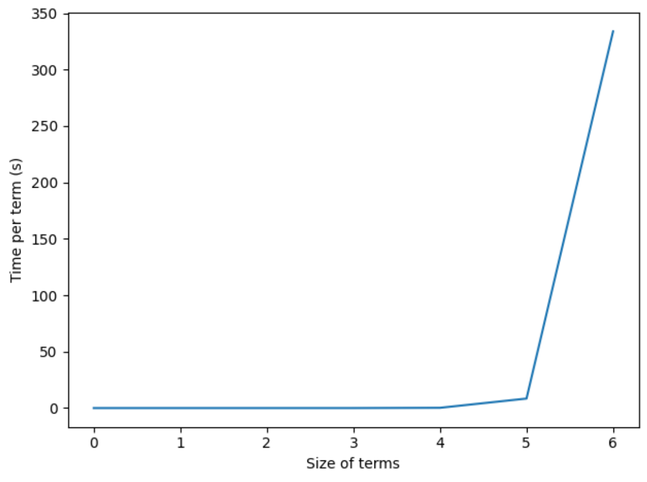

Figure 1: *Time needed to generate terms of different sizes using `UG` for the conditional ADT*

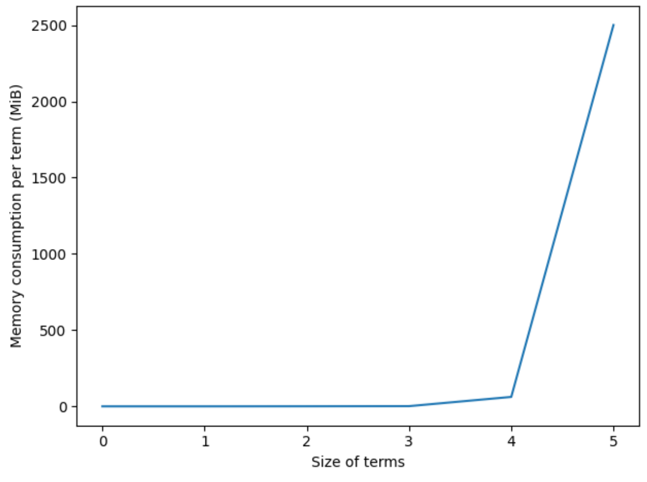

Figure 2: *Space needed to generate terms of different sizes using `UG` for the conditional ADT*

Sadly, the results are not that promising, in the sense that practical use of this method would be infeasible. Fortunately, the time results for a simply-typed lambda calculus (STLC) with De Bruijn indices give us a more hopeful outcome (see Figure 3). With this, we were able to mitigate two of the previously mentioned issues. Firstly, using this "*new*" STLC ADT (as defined in Listing 14, Appendix B) allowed us to generate lambda terms, as opposed to before when we were using string identifiers. Secondly, since the search space of valid expressions for this ADT is smaller, we could *realistically* generate terms of larger sizes. On the other hand, the QuickCheck and SmallCheck generators managed to generate more than 1000 tests for the conditional ADT in less than a second and used less than 20 MiB (note that the majority of terms were ill-typed).

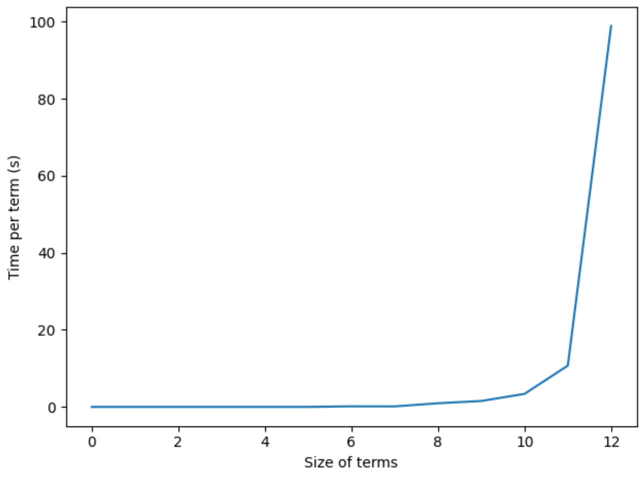

Figure 3: *Time needed to generate lambda terms of type* `Int` $\rightarrow$ `Int` *of different sizes using* `UG` *for the STLC*

5 DISCUSSION & FUTURE WORK

Following a dedicated approach to creating generators for user-defined data is considerably less prone to errors and less time consuming than manually writing test data generators. For instance, a QuickCheck generator works recursively, in the sense that at every recursion level, a random constructor is chosen, the weight of which is specified by the programmer to control the distribution [5].

As a consequence, two related questions arise: *What is a good distribution of test data such that constructors are not underrepresented? To what extent can programmers be confident that they have not introduced faults within their generator?* With a dedicated approach, such as the uniform generation method, we need to worry only about writing the spaces for the required ADT. However, while following such systematic approaches has its benefits, it is clear that some constraints or limitations have to be considered. For example, we have shown that increasing the size of the terms in the uniform generators amounts to an exponential increase in the time needed to generate expressions.

Additionally, another issue that we identified is that the uniform generation method cannot simply be applied to any ADT. For example, due to the nature of its implementation, generating terms for the simply-typed lambda calculus with string variables would be infeasible. For efficiency reasons, we recommend using a simply-typed lambda calculus with De Bruijn indices for a smaller overall search space.

Nevertheless, before applying these methods within an educational context, further research and improvements ought to be made. For instance, a time improvement is greatly needed for the uniform generation approach. More specifically, the use of memoization would significantly improve the time needed to perform indexing and cardinalities on large sets. A comparison between the authors' results for generating well-typed lambda terms and our results shows that faster indexing would amount to considerably better results. Due to time constraints and because the original paper did not contain the implementation of this feature, we were not able to include it within this research, thus we suggest this for future work.

Generally, we believe that the methods outlined in this paper could also be used to validate type-checkers. Having a unified view of the effectiveness of the uniform generation method for validating both type-checkers and definitional interpreters would increase the level of confidence in this technique. Additionally, using the Pałka et al. approach [5] as a complementary tool would also be interesting to approach in further research.

Finally, we suggest looking into additional methods of determining functional equivalence. To that end, some examples include symbolic executers [10], checking program equivalence [11] and concolic testing [12], [13]. Alternatively, fuzzing [14] could also be used as another approach for generating input expressions.

6 RESPONSIBLE RESEARCH

6.1 Ethical Implications

As mentioned in Section 1, one of the domains in which the concepts explored in this paper can be used is in the Concepts of Programming Languages course at TU Delft. As a consequence, the students' interpreters will be tested against random data, which poses the question of fair grading. Suppose two students have almost the same interpreter defined. One student may be graded maximum points because of passing tests, whereas the other student may be graded slightly lower points because their code was tested against completely different data due to randomness.

One possible approach to mitigate this issue from a fairness perspective could be to only generate random data once and subsequently use that to grade each student. Another alternative could be to perform multiple test runs before giving a verdict with regards to a student's grade. Unfortunately, both options once again pose the same issue.

A viable option could be to use the approaches mentioned in this paper as a complementary tool to aid grading rather than fully discard the tests that were already used by the course staff. In doing so, these "*new*" methods can provide more insight into the performance of the students' interpreters and assist the course staff with grading, while taking into account the equity aspect. Overall, randomness can introduce ample issues when it comes to fairness and objectivity in grading.

6.2 Threats to Validity

There are several things to address about the validity of the experimental results. Firstly, while confident in the correctness of the generators, there still is a possibility that we introduced errors. For example, our implementation of the Claessen et al. [5] paper could have introduced bugs or un-optimized code, due to possible misinterpretation.

Regarding representativeness, a possible threat to validity might be the choice of simple grammars. Throughout Section 4, we indicated that practical use of this method would require optimizations. To that end, we still illustrated how the methods fair with a more complex grammar, yet refrained from making any general statements.

Lastly, we attribute some possible noise in the time and space measurements to external background processes that could not be closed. Additionally, the time measurements for all three methods cannot have a one to one comparison, in the sense that each generator has a different notion of size, depth or number of generated tests.

6.3 Reproducibility

As discussed previously, this research project sometimes utilizes random data, therefore reproducibility is harder to achieve when considering the chosen methods. Nonetheless, the repository[6] in which all results from the previous sections were shown, is publicly available for further investigation where needed. Besides the code, this repository also includes all the relevant logs of generated input expressions that were used to perform the evaluations in Section 4. In doing so, interested readers can obtain the same results that were shown.

7 CONCLUSION

The purpose of this report was to outline and evaluate the effectiveness of different methods for generating input expressions to definitional interpreters. To this end, we explored the relative merit of three approaches from previously written research papers.

In terms of evaluation, each generation method was judged against the number of tests needed to find known faults. While the SmallCheck and uniform generators provided similar results in terms of the reported counter-examples for non-equivalent interpreters, SmallCheck required more tests to discover faults. Compared to these, QuickCheck could not detect any non-trivial faults in the interpreters due to mainly generating ill-typed expressions. In addition, we showed that the uniform generation approach can provide meaningful test data at the expense of time. In future work, we suggest an exploration of the memoization aspect to see how much the time measurements will improve.

Finally, it is our belief that a unified view of the research done in this paper and the research of validating type-checkers is needed in order to have a more complete opinion on the effectiveness of using property-based testing frameworks.

[6] `https://github.com/alemoraru/input-expression-generators`

REFERENCES

[1] C. B. Poulsen, *Automated validation of definitional interpreters*, [Online]. Available: `http://casperbp.net/store/automated-test-generation-bep-2021.pdf`, [Accessed Apr. 19, 2021].

[2] M. Sipser, *Introduction to the Theory of Computation*, Third. Boston, MA: Course Technology, 2013, pp. 219–220, ISBN: 113318779X.

[3] K. Claessen and J. Hughes, "QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs," in *ICFP '00*, (Montréal, Canada), M. Odersky and P. Wadler, Eds., ACM, 2000, pp. 268–279, ISBN: 1-58113-202-6. DOI: `10.1145/351240.351266`.

[4] C. Runciman, M. Naylor and F. Lindblad, "Smallcheck and lazy smallcheck: Automatic exhaustive testing for small values," *SIGPLAN Not.*, vol. 44, no. 2, pp. 37–48, Sep. 2008, ISSN: 0362-1340. DOI: `10.1145/1543134.1411292`. [Online]. Available: `https://doi.org/10.1145/1543134.1411292`.

[5] K. Claessen, J. Duregård and M. H. Pałka, "Generating constrained random data with uniform distribution," in *Functional and Logic Programming*, M. Codish and E. Sumii, Eds., Cham: Springer International Publishing, 2014, pp. 18–34, ISBN: 978-3-319-07151-0.

[6] M. H. Pałka, K. Claessen, A. Russo and J. Hughes, "Testing an optimising compiler by generating random lambda terms," in *Proceedings of the 6th International Workshop on Automation of Software Test*, ser. AST '11, Waikiki, Honolulu, HI, USA: Association for Computing Machinery, 2011, pp. 91–97, ISBN: 9781450305921. DOI: `10.1145/1982595.1982615`. [Online]. Available: `https://doi.org/10.1145/1982595.1982615`.

[7] N. Dubien, *Introduction to property based testing*, [Online]. Available: `https://medium.com/criteo-engineering/introduction-to-property-based-testing-f5236229d237`, [Accessed: June 13th, 2021], Mar. 2018.

[8] Z. Hu, J. Hughes and M. Wang, "How functional programming mattered," *National Science Review*, vol. 2, no. 3, pp. 349–370, Jul. 2015, ISSN: 2095-5138. DOI: `10.1093/nsr/nwv042`. eprint: `https://academic.oup.com/nsr/article-pdf/2/3/349/31566307/nwv042.pdf`. [Online]. Available: `https://doi.org/10.1093/nsr/nwv042`.

[9] J. Duregård, P. Jansson and M. Wang, "Feat: Functional enumeration of algebraic types," in *Proceedings of the 2012 Haskell Symposium*, ser. Haskell '12, Copenhagen, Denmark: Association for Computing Machinery, 2012, pp. 61–72, ISBN: 9781450315746. DOI: `10.1145/2364506.2364515`. [Online]. Available: `https://doi.org/10.1145/2364506.2364515`.

[10] C. Cadar and K. Sen, "Symbolic execution for software testing: Three decades later," *Commun. ACM*, vol. 56, no. 2, pp. 82–90, Feb. 2013, ISSN: 0001-0782. DOI: `10.1145/2408776.2408795`. [Online]. Available: `https://doi.org/10.1145/2408776.2408795`.

[11] J. Clune, V. Ramamurthy, R. Martins and U. A. Acar, "Program equivalence for assisted grading of functional programs," *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, Nov. 2020. DOI: `10.1145/3428239`. [Online]. Available: `https://doi.org/10.1145/3428239`.

[12] K. Sen, "Concolic testing and constraint satisfaction," Jun. 2011, pp. 3–4, ISBN: 978-3-642-21580-3. DOI: `10.1007/978-3-642-21581-0_2`.

[13] A. Giantsios, N. Papaspyrou and K. Sagonas, "Concolic testing for functional languages," *Science of Computer Programming*, vol. 147, pp. 109–134, 2017, Selected and Extended papers from the International Symposium on Principles and Practice of Declarative Programming 2015, ISSN: 0167-6423. DOI: `https://doi.org/10.1016/j.scico.2017.04.008`. [Online]. Available: `https://www.sciencedirect.com/science/article/pii/S0167642317300837`.

[14] P. Chen and H. Chen, "Angora: Efficient fuzzing by principled search," in *2018 IEEE Symposium on Security and Privacy (SP)*, 2018, pp. 711–725. DOI: `10.1109/SP.2018.00046`.

A GENERATORS

```
-- Auxiliary function for QuickCheck
-- arbitrary function.
-- Uses a universally true predicate
-- (no need for type-checking)
arbUniformExpr :: Gen Expr
arbUniformExpr = uniformFilter (const
    True) spExpr (...)

-- Function for generating data
-- using the QuickCheck approach
arbNaiveExpr :: Int → Gen Expr
arbNaiveExpr 0 = fmap Val arbitrary
arbNaiveExpr n = frequency
  [
    (1, fmap Val arbitrary)
  , (2, do
          left  ← arbNaiveExpr (div n 2)
          right ← arbNaiveExpr (div n 2)
          return $ Add (left, right))
  , (2, do
          left  ← arbNaiveExpr (div n 2)
          right ← arbNaiveExpr (div n 2)
          return $ Sub (left, right))
  , (2, do
          left  ← arbNaiveExpr (div n 2)
          right ← arbNaiveExpr (div n 2)
          return $ Mul (left, right))
  , (2, do
          left  ← arbNaiveExpr (div n 2)
          right ← arbNaiveExpr (div n 2)
          return $ Div (left, right))
  ]

-- Needed for arbitratry generation.
-- Gives weights to different
-- generation methods.
instance Arbitrary Expr where
    arbitrary = frequency
      [
        (..., sized arbNaiveExpr),
        (..., arbUniformExpr)
      ]
```

Listing 11: *Generators for the arithmetic ADT*

```
-- Necessary for SmallCheck exhaustive
    generation
instance (Monad m) ⇒ Serial m Expr where
  series = cons1 Val \/ cons1 Add \/
    cons1 Sub \/ cons1 Mul \/ cons1 Div
```

Listing 12: *SmallCheck generator for the arithmetic ADT*

B LANGUAGE DEFINITIONS

This appendix includes the language definitions of two different ADTs that we used for different benchmarks in Section 4. Listing 13 contains the conditional ADT, whereas Listing 14 shows the simply-typed lambda calculus with De Bruijn indices. For the latter ADT, the `Type` in the `App` constructor denotes the type of the argument term and is useful for type-checking terms.

```
data Type = TInt | TBool
  | TClos Type Type

data Expr = EInt Int | EBool Bool
  | Id String
  | Add (Expr, Expr) | Mul (Expr, Expr)
  | Not Expr
  | Or (Expr, Expr) | And (Expr, Expr)
  | Eq (Expr, Expr) | Lt (Expr, Expr)
  | Gt (Expr, Expr)
  | Lambda ((String, Type), Expr)
  | App (Expr, Expr)
  | If (Expr, (Expr, Expr))
```

Listing 13: *ADT with conditional operations*

```
data Type = TInt
  | TFun (Type, Type)

data Expr = Var Int
  | App ((Expr, Expr), Type)
  | Lam Expr
```

Listing 14: *STLC ADT*

C OTHER RESULTS

This appendix contains some of the results that we obtained when evaluating the uniform generation method for the boolean ADT and arithmetic ADT, respectively. More specifically, figures 4 and 5 illustrate how the size of the generated terms affects the number of faults found for the respective faulty interpreters.

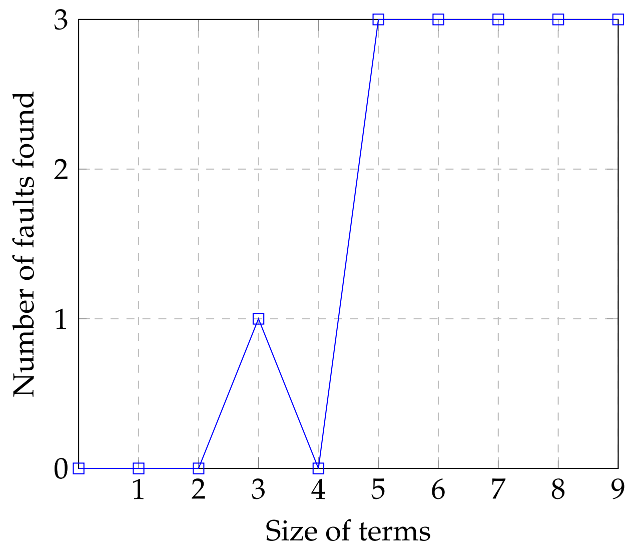

Figure 4: *Number of faults found as size of terms increases using the uniform generation method for the boolean ADT. Three faults were introduced in total.*

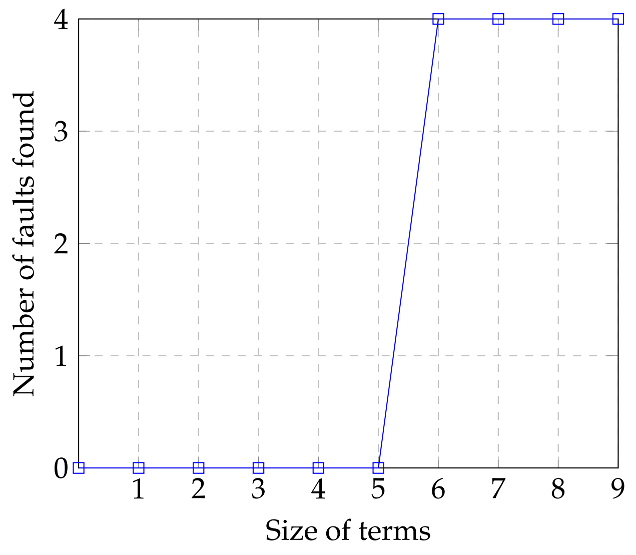

Figure 5: *Number of faults found as size of terms increases using the uniform generation method for the arithmetic ADT. Four faults were introduced in total.*

D ANALYSIS OF THE MIXED GENERATOR

For the mixed generator, we used the same test suite as mentioned in Section 4 for the conditional ADT. The results of this analysis, however, did not improve, in the sense that the mixed generator still was not capable of detecting the (e-i) faults. This directly stems from the fact that the uniform generation size choice of 5 does not allow for larger terms, such as the ones that contain errors (e.g. `If`, `Eq`) and because the QuickCheck generator happened to generate only well-typed terms of small sizes. Tables 6, 7, 8 use the same size for the uniform generator, but utilize different generator weights.

	a	b	c	d	e	f	g	h	i
`#Runs`	10	10	10	10	-	-	-	-	-
`#Tests`	14	14	5	10	-	-	-	-	-

Table 6: *Results for the mixed generator with equal weights and size 5 for* `UG`

	a	b	c	d	e	f	g	h	i
`#Runs`	10	10	10	10	-	-	-	-	-
`#Tests`	16	11	5	6	-	-	-	-	-

Table 7: *Results for the mixed generator with weight 4 and term size 5 for* `UG` *and weight 1 for* `QC`

	a	b	c	d	e	f	g	h	i
`#Runs`	10	10	10	10	-	-	-	-	-
`#Tests`	14	11	7	7	-	-	-	-	-

Table 8: *Results for the mixed generator with weight 1 and term size 5 for* `UG` *and weight 4 for* `QC`

However, Table 9 illustrates the results of 10 test runs using a mixed generator and uniform generator of terms of size 6. While this benchmark took a considerable amount of time, it managed to find the (h-i) faults, thus once again proving that bigger term sizes in the uniform generator amount to better results at the expense of time.

	a	b	c	d	e	f	g	h	i
`#Runs`	10	10	10	10	-	-	-	10	9
`#Tests`	19	22	3	5	-	-	-	18	20

Table 9: *Results for the mixed generator with equal weights and size 6 for* `UG`