Validating Type Checkers Using Property-Based Testing

Casper Bach Poulsen **Sára Juhošová** Cas van der Rest

Delft University of Technology

ABSTRACT

Manually testing definitional interpreters and their type checkers is a tedious and error-prone process which can largely benefit from automation. This study evaluates the effectiveness of property-based testing on errors in type checkers. Metrics used include the ability to catch different types of errors as well as the ability to provide a good margin of confidence in the results. We define two languages in Haskell and identify properties which should hold for their type checkers. Using a bottom-up approach to expression generation, we evaluate the effectiveness of property-based testing on test suites of type checkers with various types of errors. The method proves to be effective for both the simple and the complex language and manages to catch all defined error types with at least one property to a sufficient degree of confidence. We conclude that property-based testing is an effective tool to help with manual grading of type checkers for definitional interpreters.

1 Introduction

Writing definitional interpreters for simple languages is commonly part of the curriculum for bachelors of computer science. They are a type of interpreter written for a language defined and interpreted using a different, usually better understood, programming language [1]. Their grading can often be time-consuming and, with larger student cohorts, can quickly become infeasible. Researching methods for automatic grading of these interpreters is therefore an attractive area of research to the staff of such courses.

The question of automated testing and validation of programmes has been researched widely due to the large efforts that need to go into these things when they are done manually. Examples of attempts to automate this process include property-based testing [2], fuzzing [3], symbolic executors [4], and checking programme equivalence [5]. Most of these are designed as tools to complement the manual process and make it easier - they are not meant to replace it.

Many of these solutions have already worked with programmes written in functional styles, such as the influential paper by Claessen and Hughes introducing QuickCheck [2], the property-based testing framework for Haskell. Instead of writing specific test cases, property-based testing generates large batches of (pseudo-)random input and asserts properties about it.

Consider the Haskell method `reverse :: [a] -> [a]` which takes a list of elements and returns it in reversed order. We can determine two properties for it: (1) `reverse (reverse xs) == xs` and (2) if `length xs <= 1` then `reverse xs == xs`. The first one simply asserts that for any randomly generated list, the reverse of the reverse should be equivalent to the original list. The second one uses a precondition (namely that of the list having zero or one element) to assert that the reverse of such a list should be equivalent to the original.

Pałka et al. [6] have already explored the principles of property-based testing in relation to testing compilers. They introduced methods which come in equally handy for definitional interpreters and allow for the generation of useful terms to assert determined properties on. Our research draws largely on their work and attempts to analyse its effectiveness in this new

context. We also expand it to work on a more complex language than they work with in their paper.

An important aspect to consider when testing definitional interpreters is the existence of its type checker. As claimed by Pierce [7, p. 1], "by far the most popular and best established lightweight formal methods [for helping ensure that a system behaves correctly with respect to some specification, implicit or explicit, of its desired behavior] are type systems". Type systems have become widespread among programming languages and generally require programmes to be run through type checkers before being passed on to the interpreter.

The presence of a type checker implies that if an interpreter is being tested by running it on some expression, that input expression must be well-typed to even pass to the interpreter. Furthermore, the type checker itself must be tested to ensure the overall validity of the definitional interpreter. Exploring the possibilities of generating well-typed input expressions for definitional interpreters allows for the potential to both test type checkers and avoid the unnecessary raising of errors in the interpreters that follow.

We have decided to explore the potential of property-based testing to automatically validate type checkers for three reasons. Firstly, it is able to create large amounts of test cases in a short time. Secondly, it provides examples of input which fail for a property, meaning that we are not only aware that it found a bug but also get an intuition as to where that bug could be. Lastly, property-based testing has trusted frameworks in many programming languages, making it a robust and flexible option.

This paper answers the following research question: *How effective is property-based testing for automatically validating type checkers for definitional interpreters?* It focuses primarily on judging the extent to which property-based testing is able to find bugs in type checkers. The metrics used for determining this effectiveness include the frequency of catching a buggy implementation as well as the amount of tests needed to find a counterexample to incorrect implementations. We also reflect on which types of bugs this style of testing is able to find within the type checkers and why it might not be suitable for some.

To answer this research question, we defined two languages in Haskell along with their properties and generators. Then, we analysed how well property-based testing does on finding bugs in a test suite of type checkers. We found that well-typed expression generators catch a specific set of errors almost every test run under the correct settings, while a different set of errors needs to be attacked with ill-typed expressions. We present the following contributions:

1. A Haskell implementation of two languages of varying complexity and reference implementations of their type checkers (available in our public GitLab Repository[1]).
2. A definition and implementation of their properties which can be used for property-based testing (presented in Section 3.1).
3. An approach to generating *well-typed* expressions for both of these languages (described in Section 3.2).

The paper begins by describing the problem tackled by the research in Section 2 and presenting the contributions in Section 3. Sections 4 and 5 present, discuss, and reflect on the results. Section 6 offers some context and possibilities for further work. Finally, Section 7 summarizes and concludes the research.

2 Writing a Type Checker

The languages whose type checkers we validate are defined as algebraic data types in Haskell. An example of both a language and its possible types can be found in Listing 1. This simple language can only have expressions whose type is either an integer or a boolean.

```
1 data Type = TInt | TBool
2
3 data Expr = Id String
4   | True | False
5   | Num Int
6   | Eq Expr Expr
```

Listing 1: *Language & Type Definition*

[1] https://gitlab.ewi.tudelft.nl/cse3000-auto-test/typed-expression-generators

The outline for a type checker of such a language is shown in Listing 2. This type checker takes an expression and an environment (defined as a list of pairs of strings and types) and outputs the type of the expression. If the expression is ill-typed, the type checker should throw an error.

```
1 type TEnvironment =
2   [(String, Type)]
3 data Error = TypeError
4   | InterpError
5
6 typeOf :: Expr -> TEnvironment ->
    Either Error Type
7 typeOf e env = ...
```

Listing 2: *Type Checker Outline*

The idea behind this research is to determine the effectiveness of property-based testing on finding errors in such type checkers. These errors can be of two types: (1) incorrectly typing a well-typed expression and (2) not recognizing an ill-typed expression.

Consider the two implementations of an integer equality type check in Listing 3. An error of type (1) would arise when A incorrectly types the expressions `Eq (Num 1) (Num 4)` as `TInt` instead of a `TBool`. Typing this expression using B would yield the correct result. An error of type (2) would occur when typing expression `Eq True (Num 4)` using B, since it would not check the sub-expressions. On the other hand, A would correctly throw a `TypeError`.

```
1 -- Type Checker A
2 typeOf (Eq left right) nv =
3   do
4     l <- typeOf left nv
5     r <- typeOf right nv
6     case (l, r) of
7       (TInt, TInt) -> return TInt
8       _ -> Left TypeError
```

```
1 -- Type Checker B
2 typeOf (Eq left right) nv =
3   Right TBool
```

Listing 3: *Integer Equality*

This research focuses on detecting both of these error types. Because generating *well-typed* expressions is a more complex problem, we focus primarily on the methods behind that. However, neither a well-typed nor an ill-typed generator is sufficient on its own for catching all error types. Therefore, in the results, we use a combination of both.

2.1 A Note on Notation

In this paper, *variables* are denoted as small letters of the Latin alphabet (x, y, ...), *types* are denoted as small letters of the Greek alphabet (σ, τ, ...), and *terms* or *expressions* are denoted using capital letter of the Latin alphabet (M, N, ...). A *function type* is denoted as $\sigma \to \tau$ where σ is the type of the input parameter and τ is the output type of the function. A *list type* is denoted as $[\sigma]$, representing a list of elements of type σ. The symbols $\mathfrak{T}$ and $\mathfrak{C}$ are used to denote type checkers.

A *binding* $x : \sigma$ indicates that variable x has type σ. An *environment* is represented as a list of bindings Γ. The notation $[x : \sigma]\Gamma$ denotes that the variable x is now bound as σ in environment Γ regardless of what (if anything) it was bound as before. Choosing an *arbitrary element* x from a collection X is indicated as $x \in_R X$.

Typing judgements of the form $\Gamma \vdash M : \sigma$ are used to denote that M is well-typed as σ if all its free variables are bound within Γ. The expression $\Gamma \vdash_{\mathfrak{T}} M : \sigma$ denotes that a type checker $\mathfrak{T}$ has judged expression M to be of type σ with environment Γ.

The *syntax definition* for the language implemented in Listing 1 would be the following:

$$
\begin{aligned}
\sigma, \tau, ... &::= \texttt{Int} \mid \texttt{Bool} \\
M, N, ... &::= x \\
&\quad \mid \texttt{true} \mid \texttt{false} \\
&\quad \mid i \in \mathbb{Z} \\
&\quad \mid M == N
\end{aligned}
$$

3 Testing Lambda Calculus

We decided to explore the effectiveness of property-based testing on two languages of different complexity. We define them both in this section along with their properties. The generation of their terms is explained in the last subsection.

In this section, we only discuss the languages using their syntax definitions. To view their implementation in Haskell as well as the suite

of type checkers and generators, the interested reader can explore the source code on the public GitLab repository or look at Appendix A - C.

$$
\begin{aligned}
\sigma, \tau, \ldots &::= \texttt{Int} \mid \texttt{Bool} \\
&\mid \sigma \to \tau \\
M, N, \ldots &::= x \\
&\mid \lambda x : \sigma . M \\
&\mid M\, N
\end{aligned}
$$

Figure 1: *STLC Definition*

Simply-Typed Lambda Calculus (STLC). The simplest language of sufficient interest for this project is the simply-typed lambda calculus whose definition can be seen in Figure 1. It deals only in variables, lambda expressions, and function applications, all of which are typed as either an integer, a boolean, or a function.

$$
\begin{aligned}
\sigma, \tau, \ldots &::= \texttt{Int} \mid \texttt{Bool} \\
&\mid [\sigma] \\
&\mid \sigma \to \tau \\
M, N, \ldots &::= x \\
&\mid \texttt{true} \mid \texttt{false} \\
&\mid i \in \mathbb{Z} \\
&\mid \texttt{[]} : \sigma \\
&\mid \texttt{unop}\ M \\
&\mid M\ \texttt{binop}\ N \\
&\mid \lambda f : (\sigma \to \tau)\ x . M \\
&\mid M\, N \\
&\mid \texttt{let}\ x = M\ \texttt{in}\ N \\
&\mid \texttt{if}\ M\ \texttt{then}\ N_1\ \texttt{else}\ N_2 \\
\texttt{unop} &::= \neg \\
&\mid \texttt{head} \mid \texttt{tail} \\
&\mid \texttt{is-nil} \mid \texttt{is-list} \\
\texttt{binop} &::= \wedge \mid \vee \\
&\mid + \mid * \\
&\mid == \mid < \\
&\mid \texttt{cons}
\end{aligned}
$$

Figure 2: *PCF Definition*

Programming Computable Functions (PCF). The simply-typed lambda calculus can be expanded with both more types and more expressions. For its complex alternative, we decided to use the language defined in Figure 2, where a list type is added as well as a few native operations, including `let` bindings and `if-then-else` expressions. Because it is a variation of Programming Computable Functions [8], we will refer to it as such in this paper.

3.1 Properties

For the simply-typed lambda calculus, we have identified three properties which can be defined for all implementations of its type checker. Their Haskell implementation can be found in Appendix B and their definition is as follows:

`CMP` Given two different implementations of a type checker ($\mathfrak{T}$, $\mathfrak{C}$), they should both evaluate to the same type given the same input expression M (or both throw an error). More specifically, we are interested here in comparing to a reference type checker implementation.

$$(\Gamma \vdash_{\mathfrak{T}} M : \sigma) \iff (\Gamma \vdash_{\mathfrak{C}} M : \sigma) \tag{1}$$

`GEN` If an expression M is generated using the bottom-up approach from type σ with environment Γ (as described in Section 3.2), running the type checker $\mathfrak{T}$ on that expression with environment Γ should yield the type σ.

$$(\Gamma \vdash M : \sigma) \implies (\Gamma \vdash_{\mathfrak{T}} M : \sigma) \tag{2}$$

`APP` Given an expression M of type $\sigma \to \tau$ and an expression N of type σ (both under environment Γ), running the type checker $\mathfrak{T}$ on expression MN should yield the type τ.

$$\begin{aligned}(\Gamma \vdash M : \sigma \to \tau) \wedge (\Gamma \vdash N : \sigma) \\ \implies (\Gamma \vdash M\, N : \tau)\end{aligned} \tag{3}$$

For the PCF, all of the properties defined above also hold. It is, furthermore, also possible to make expression-specific properties. For example, if M and N are both of type `Int`, then the expression $M + N$ should also be typed as `Int`.

This type of property can be defined for all the `unop` and `binop` operators as well as the `let` and the `if`. It can be used to replace writing manual tests for these expression types.

3.2 Input Generation

The main idea behind the input generation revolves around choosing a type and then choosing an applicable rule to generate an expression of that type. Here, we draw on the work of Pałka et al. [6], who present these generation rules for the Simply-Typed Lambda Calculus. We define our own for the PCF language.

The choice of type can either be arbitrary (useful in the instance of property `GEN`) or intentional (like in property `APP`). In either case, a generator for `Type` needs to be defined. In our design, a depth parameter limits the maximum size of the type to omit the possibility of infinite recursion. For the PCF, a list depth parameter is added to control the type size of elements in a list. The mathematical definition for both type generators can be found in Figures 3 and 4.

The expression generation itself works on a very similar principle. The generation function takes as parameters a depth, a type, and an environment. Then, based on the first two parameters, a generation rule is chosen from a set of applicable ones and run with all three parameters to generate input. Some of these rules recursively use other rules to generate sub-expressions. Others only consider a certain subset of possible outputs because they should never be called by impossible combinations. The functions which regulate the applicable rules for our languages are defined in Figures 5 and 6. The rules are all explained below:

`Var`(d, σ, Γ) picks a random variable x from Γ of type σ

`Lam`$(d, \sigma \to \tau, \Gamma)$ picks a random x from a pool of variable names (and a random f from a pool of function names), generates an arbitrary term with M parameters $(d-1, \tau, [x : \sigma]\Gamma)$, and returns $\lambda x : \sigma.M$ (or $\lambda f : (\sigma \to \tau)\ x.M$)

`App`(d, τ, Γ) generates an arbitrary type σ, an arbitrary term M with parameters $(d-1, \sigma \to \tau, \Gamma)$, an arbitrary term N with parameters $(d-1, \sigma, \Gamma)$, and returns $M\ N$

`Prim`(d, σ, Γ) generates an arbitrary boolean, number, or list, depending on σ

`Nat`(d, σ, Γ) generates an arbitrary `unop` or `binop`, depending on σ

`If`(d, σ, Γ) generates an arbitrary term M with parameters $(d-1, \texttt{Bool}, \Gamma)$, two arbitrary terms N_1 and N_2 with parameters $(d-1, \sigma, \Gamma)$, and returns `if` M `then` N_1 `else` N_2

`Bind`(d, σ, Γ) picks a random x from a pool of variable names, generates a random type τ of depth 1, an arbitrary expression M with parameters $(d-1, \tau, \Gamma)$, an arbitrary expression N with parameters $(d-1, \sigma, [x : \tau]\Gamma)$, and returns `let` $x = M$ `in` N

There can be a case that the generation is unsuccessful and for such reasons, the actual generator works with the `Maybe` monad. In case a rule does not successfully generate an expression, it returns a `Nothing` to the recursive call one step higher. That call then either attempts to use the `Var` rule or the `Prim` rule or simply returns a `Nothing` itself.

There are also, of course, other backtracking options, such as trying out any of the other possible rules. We have found, however, that this one is easy to implement and at the same time sufficient for successfully generating terms in the languages we have defined above as long as arbitrary types are not generated with a too high depth. Providing a basic environment which already contains variables of the basic types to the STLC allows the `Var` rule to often be sufficient. In PCF, we have the option to generate an integer or a boolean in addition to choosing a variable from the environment. Since we use Haskell library methods for those, we are guaranteed a successful result for the simple types.

In case the entire expression is unsuccessfully generated, a new type is generated and the process is repeated again until success. The Haskell implementation of this loop can be found in Appendix C, Listing 9. This can, in theory, lead to infinite recursion. However, the chance of that is so small, it has never caused us a problem throughout all the tests we ran. A timeout or attempt-limit condition can be added to help mitigate this issue when using the generator. Lowering the type depth also lowers the risks, since the environment is more likely to be able to provide a variable.

$$\text{arb}(d_t) = \begin{cases} x \in_R \{\texttt{Int}, \texttt{Bool}\} & \text{if } d_t \leq 0 \\ x \in_R \{\texttt{Int}, \texttt{Bool}, \text{arb}(d_t - 1) \rightarrow \text{arb}(d_t - 1)\} & \text{else} \end{cases}$$

Figure 3: *Type Generation (STLC)*

$$\text{arb}(d_t, d_{list}) = \begin{cases} x \in_R \{\texttt{Int}, \texttt{Bool}\} & \text{if } d_t \leq 0 \\ x \in_R \{\texttt{Int}, \texttt{Bool}, \text{arb}(d_t - 1) \rightarrow \text{arb}(d_t - 1), \texttt{[}\text{arb}(d_{list})\texttt{]}\} & \text{else} \end{cases}$$

Figure 4: *Type Generation (PCF)*

4 Results

In this section, we present and discuss the results of the research. For both of the languages, a test suite of type checkers was devised and each type checker was tested with 100 tests run 100 times on each property. We refer to each 100 tests as a "test run" within an "iteration" of 100 runs. To capture all the relevant information, we decided to present three values in the results:

`ecf`: *(error-catching frequency)* the percentage of test runs which caught the error

`avg`: from all test runs which caught the error, the average of the number of tests needed

`max`: from all test runs which caught the error, the maximum of the number of tests needed

4.1 STLC Results

The following test suite was used to test the effectiveness of the generator and properties for the Simply-Typed Lambda Calculus:

- a no case match for the variable
- b no case match for the lambda
- c no case match for the application
- d incorrect binding in the lambda (binding is appended to the list and thus old binding is used in case of overlap)
- e no binding for the lambda parameter
- f parameter and body type are swapped in function type
- g application returns the parameter type instead of the body type
- h no check whether the parameter and argument type match in an application
- i no type check at all for the type of an argument in an application

We present the results for the analysis of the STLC when run with expression depth 10, type depth 2, and size 3 for the variable pool[2]. We also provided a basic environment as input into the generator to facilitate generation of more varied expressions. The contents of the basic environment can be found in Appendix D.

	a	b	c	d	e	f	g	h	i
CMP	100	100	100	50	96	100	100	0	0
GEN	100	100	100	60	99	100	100	0	0
APP	100	100	100	88	100	100	100	0	0
MIS	100	100	100	0	0	95	12	100	100

Table 1: *Results for the error-catching frequency of the properties on STLC (in %)*

Table 1 gives an overview of the `ecf` for all the properties on all the type checkers defined in the test suite. Tables 2 - 4 provide more information about these properties within each test run.

	a	b	c	d	e	f	g	h	i
avg	1	3	4	46	19	3	5	-	-
max	1	17	15	97	71	21	26	-	-

Table 2: *Results for* `CMP` *(STLC)*

	a	b	c	d	e	f	g	h	i
avg	1	2	3	45	16	2	5	-	-
max	1	9	13	100	82	8	20	-	-

Table 3: *Results for* `GEN` *(STLC)*

[2] The raw data for these results can be found in this repository: `https://gitlab.ewi.tudelft.nl/cse3000-auto-test/sara-results`.

$$\text{rule}(d,\sigma) = \begin{cases} \texttt{Var} & \text{if } d \leq 0 \\ x \in_R \{\texttt{App}, \texttt{Lam}, \texttt{Var}\} & \text{if } d > 0 \text{ and isFun}(\sigma) \\ x \in_R \{\texttt{App}, \texttt{Var}\} & \text{else} \end{cases}$$

Figure 5: *Rule Choice Function (STLC)*

$$\text{rule}(d,\sigma) = \begin{cases} \texttt{Var} & \text{if } d \leq 0 \text{ and isFun}(\sigma) \\ x \in_R \{\texttt{Prim}, \texttt{Var}\} & \text{if } d \leq 0 \text{ and not isFun}(\sigma) \\ x \in_R \{\texttt{App}, \texttt{Bind}, \texttt{If}, \texttt{Nat}, \texttt{Var}\} & \text{if } d > 0 \text{ and isFun}(\sigma) \\ x \in_R \{\texttt{App}, \texttt{Bind}, \texttt{If}, \texttt{Lam}, \texttt{Var}\} & \text{else} \end{cases}$$

Figure 6: *Rule Choice Function (PCF)*

	a	b	c	d	e	f	g	h	i
avg	1	1	1	38	7	1	1	-	-
max	1	3	1	93	54	5	3	-	-

Table 4: *Results for* `APP` *(STLC)*

The first type of errors introduced was a missing case match for an expression (a-c). These are the ones on which all the properties had a perfect `ecf`, with `GEN` and `CMP` being only slightly slower than the almost immediate `APP`.

The second type of errors is that of misbound variables (d-e). While e, which simply has no binding for the lambda parameter, was always caught by all three properties, d had a significantly lower success rate in all but the `APP` property. This success is closely correlated with the size of the variable pool.

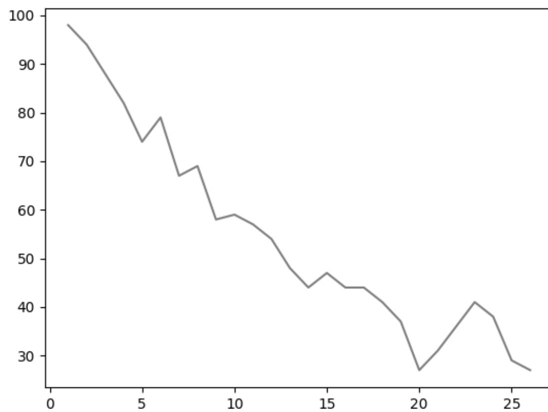

Figure 7: *variable pool size vs. #test runs which caught the error*

To display this, we ran 100 iterations of 100 test runs for variable pool sizes from 1 until 26. Figure 7 displays the number of test runs per iteration which successfully caught the error. A downwards trend is easily spotted as the size of the variable pool increases. Figure 8 displays the average number of tests which had to be run to catch an error. Conversely, an upwards trend can be spotted as the size of the variable pool increases. Based on this information, we suggest using variable pools of size smaller than 4 to guarantee as much success as possible.

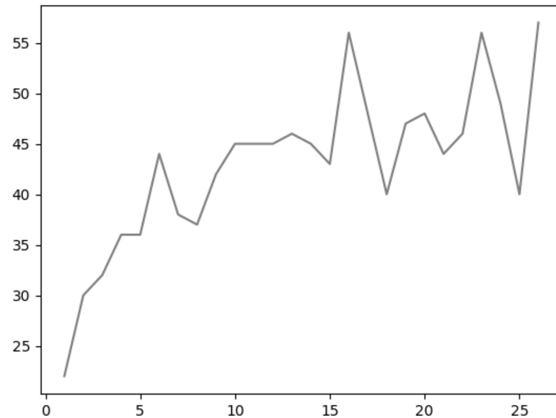

Figure 8: *variable pool size vs. average #tests needed to catch the error (excluding runs where the error was not caught)*

The third type of errors is simple mistakes in returning the wrong type (f-g). These errors seem to be trivially caught by all three properties, with `APP` once again proving to be the most reliable.

Lastly, errors which stem from ill-typed expressions were not caught at all by any of the properties. This, of course, has an obvious explanation: we used generators for *well-typed* ex-

pressions. To mitigate this issue, we wrote a simple bogus generator which takes no notice of what type it generates for sub-expressions. We ran this on the test suite using the simple `MIS` property of "if this bogus generator generates an expression that fails on the correct type checker, it should also fail on the incorrect one" which is equivalent to the `CMP` property but uses a different generator.

	a	b	c	d	e	f	g	h	i
avg	1	1	1	-	-	32	47	1	1
max	1	1	1	-	-	96	86	1	1

Table 5: *Results for* `MIS` *(STLC)*

As can be seen in Figures 1 and 5, this approach worked perfectly for the last two buggy type checkers as well as for the first three missing case matches. For the remaining four type checkers, the errors are subtle enough to often pass undetected since they often first throw an error on the ill-typed expression before even having a chance to return their incorrect typing. This shows that both the ill-typed and the well-typed generators are necessary to be able to catch the various different types of errors.

4.2 PCF Results

The results for the PCF were very similar to those of the STLC. We analysed the results when run with expression depth 10, type depth 2, list depth 1, and variable pool size 3. The following test suite was used:

- a no case match for the integer
- b no case match for the `tail` unary operator
- c no case match for the `let` case
- d the == case match returns an `Int` type
- e lambda doesn't bind the function name in the environment
- f lambda doesn't bind the function name or the parameter in the environment
- g `unop` and `binop` don't check the types of their sub-expressions
- h `if` doesn't check type equality of the `then` and `else` case
- i no check whether the condition of `if` has a `Bool` type

Table 6 displays the overview for the `ecf` of all the properties on the whole test suite. We decided to only include the `avg` and `max` values in Appendix E, since their range is very similar to that of the matching error types from the STLC test suite.

	a	b	c	d	e	f	g	h	i
CMP	100	100	100	100	100	100	0	0	0
GEN	100	100	100	100	100	100	0	0	0
APP	100	100	100	100	100	100	0	0	0
MIS	100	100	100	0	0	0	100	4	6

Table 6: *Results for the error-catching frequency of the properties on PCF (in %)*

Just as in the STLC, the properties which used the well-typed generator effectively caught errors in type checkers which missed a case match, returned an incorrect type, or had incorrect bindings. Similarly, the `MIS` property which uses the ill-typed generator caught errors in type checkers which missed a case match or didn't check types of sub-expressions.

However, we can spot a lower effectiveness on type checkers h and i where errors only occur in one specific case match. This is simply because unlike the STLC language which has 3 possible expression types, this one has 21. That does mean that, in theory, less than 5% of the generated content is the expression type needed to catch the error. Possible solution to making the `ecf` higher include tweaking the distribution of the generated expressions as well as including more tests within one test run.

4.3 Generation Speed

We also analysed the generation speed of the expressions with each of the four generators (`STLC` (well-typed), `STLC-ILL`, `PCF` (well-typed), `PCF-ILL`). For expression depths ranging from 1-20, we measured the time it took to generate 100 expressions with each generator. This process was repeated 100 times.

Initially, an upwards trend was expected in time taken with the increase of the expression depths. However, this proved to not be the case. Upon further evaluation, we determined the cause. While higher expression depths allow for deeper expressions, the reaching of those maxima is still very unlikely. Therefore, Table 7 displays the average, median, and maximum

values analysed over the results from all the expression depths combined.

	STLC	STLC-ILL	PCF	PCF-ILL
avg	0.0068	0.0065	0.0032	0.0029
med	0.0022	0.0020	0.0010	0.0010
max	0.0690	0.0677	0.0419	0.0814

Table 7: *Generation speed (in seconds) of 100 expressions analysed over 100 runs for each expression depth ranging from 1-20*

From the data gathered, it is shown that even in the maximal case, the generation of 100 expressions takes below a tenth of a second. This means that even with the test start-up and the property checking, this approach is fast enough to be useful as a feedback tool as well as as a grading tool.

5 Reflection

In this section, we reflect on the threats to the validity of our research. Furthermore, we also discuss the ethical aspects of our research and reflect on its reproducibility. With this, we aim to inform the reader of important things to consider when using it for future work.

5.1 Threats to Validity

Firstly, we consider whether the results are representative. We ran all the test runs with 100 iterations, trying to see how often a bug would be caught by the properties. Granted, it is possible to examine these statistics on more iterations. However, we often obtained results close to the extremas (either 0 or 100). This leads us to believe that the results are representative.

Secondly, we consider the scalability of this solution. Obviously, the more complex the defined languages get, the more complex the generators become. The problem is that this is not a general solution and so it will always take time and effort to setup up a generator for each language with even a little variance. Luckily, this process becomes quite automatic once the basic rules (`App`, `Lam`, and `Var`) have been implemented and is easily scalable to more granular languages. One simply needs to be aware of the output type each new expressions can have and define rules based on this knowledge. For example, generating a `if` expression of type σ simply requires generating two expressions of type σ and wrapping them in an `if` layer.

Furthermore, as we saw in Section 4.3, once these generators are implemented, the size and complexity of the language does not seem to affect the speed of the generation. Since the two languages we defined vary greatly in complexity, we believe that the solution is in general scalable to larger languages.

5.2 Ethical Aspects

The first ethical aspect to consider for this research is that of objectivity. While the test suite of type checkers which were explored has been written with the goal to explore as many types of bugs as possible, it was still only written by the authors of this paper and not directly gathered from students. We did, however, use the knowledge gathered by teaching assistants from the CPL[3] course to write this and thus believe to have captured a sufficient variety of possible errors.

The second ethical aspect to consider is the idea of fairness. While automated validation eases the grading process for the course staff, it takes away the benefits that come from personal attention to the students when used on its own. It can also result in unequal grading for similar submissions due to the random factor of property-based testing. For this reason, we not only included whether a bug was caught by a test run but also the number of the test it was caught on. We strongly recommend that course staff use this information to their advantage and evaluate the risks of not complementing the automated validation with manual grading.

5.3 Reproducibility

When it comes to reproducibility, the biggest problem of this research is the aspect of randomness in the written generators. Even while the code for the languages and their generators and properties is publicly available on an open source repository, simply running the code will result in slightly different results. We decided

[3]CSE2120: Concepts of Programming Languages

to not solve this by seeding the random input, since we believe this would create a bias in our results. Instead, this issue is mitigated by including the log of generated inputs for all the evaluations done in Section 4 in a public repository. This means that the output is easily verifiable by simply taking these inputs and running them on the defined type checkers.

6 Context & Further Work

Property-based testing allows for running large amounts of tests with little effort once the setup has been completed. This setup, however, takes quite a bit of effort as it does not port directly from one language to another (unlike with other forms of automated validation such as symbolic execution). This means that for every language and type checker, a new generator must be written for both well-typed expressions and ill-typed expressions. Furthermore, for every programming language in which the type checked language is defined, this needs to be done anew. Interesting further research could explore the possibilities of automating this process within one programming language.

While this study was specific to type checkers, using this research and evaluating its effectiveness on the definitional interpreters themselves could prove useful. The work by Claessen et al. [9], for example, examines generation of uniformly distributed data. Comparing and combining this research with the one done by them gives opportunities for exercising a definitional interpreter with uniform distribution while also ensuring that the expression passes through the type checker (which, in that case, would not be under test).

Aside from searching for a way to generate well-distributed data, *targeted* property-based testing is another approach to generating better expressions and obtaining better results. Introduced by Löscher and Sagonas [10], targeted property-based testing is "an enhanced form of [property-based testing] that makes its input generator component of a [property-based testing] tool guided by a search strategy instead of being random". Research into this field could help improve discovering errors which only appear in one of many expressions types (such as were discussed in Section 4.2).

Furthermore, conducting research on generating ill-typed, yet not completely bogus expressions would help improve the ability to catch errors in type checkers which are too lenient. While defining such a generator was quite trivial for the Simply-Typed Lambda Calculus, it proved to be a bit more challenging for the complex one. The distribution is not clearly decided and more refined methods should be explored for obtaining an optimal one.

Finally, we suggest research into expression shrinking. While property-based testing is able to detect bugs using a property, the counterexample it finds is often too big and "noisy". QuickCheck includes an option for users to define a way to "shrink" their expression to something simpler yet still breaking to the property [11]. Researching ways to effectively shrink input expressions into definitional interpreters would allow for both better understanding of the bugs as well as for the option of providing useful and easily obtainable feedback to the students.

7 Conclusion

Our goals were to evaluate the effectiveness of property-based testing on type checkers for definitional interpreters. We have found that using a combination of two generators (one for well-typed and one for ill-typed expressions) and merely two properties enables the test suite to catch errors of all defined types for languages of various degrees of complexity.

There are a few factors within these generators which can affect distribution and effectiveness. For example, keeping the variable pool size to a minimum offers more security in catching binding errors and generating less deep types has a higher success rate of expression generation. These factors can be modified and adapted to suite the entity using this approach.

In conclusion, this method of automated validation for type checkers of definitional interpreters is fast, effective, and easily scalable once implemented for a first simple language.

REFERENCES

[1] J. C. Reynolds, “Definitional interpreters for higher-order programming languages,” in *Proceedings of the ACM Annual Conference - Volume 2*, ser. ACM ’72, Boston, Massachusetts, USA: Association for Computing Machinery, 1972, pp. 717–740, ISBN: 9781450374927. DOI: 10.1145/800194.805852. [Online]. Available: https://doi.org/10.1145/800194.805852.

[2] 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.

[3] B. P. Miller, L. Fredriksen and B. So, “An empirical study of the reliability of unix utilities,” *Commun. ACM*, vol. 33, no. 12, pp. 32–44, Dec. 1990, ISSN: 0001-0782. DOI: 10.1145/96267.96279. [Online]. Available: https://doi.org/10.1145/96267.96279.

[4] A. D. Mensing, H. van Antwerpen, C. Bach Poulsen and E. Visser, “From definitional interpreter to symbolic executor,” in *Proceedings of the 4th ACM SIGPLAN International Workshop on Meta-Programming Techniques and Reflection*, ser. META 2019, Athens, Greece: Association for Computing Machinery, 2019, pp. 11–20, ISBN: 978145
0369855. DOI: 10.1145/3358502.3361269. [Online]. Available: https://doi.org/10.1145/3358502.3361269.

[5] J. Clune, V. Ramamurthy, R. Martins and U. A. Acar, “Program equivalence for assisted grading of functional programs,” vol. 4, no. OOPSLA, Nov. 2020. DOI: 10.1145/3428239. [Online]. Available: https://doi.org/10.1145/3428239.

[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] B. C. Pierce, *Types and Programming Languages*. Massachusetts Institute of Technology: The MIT Press, 2002.

[8] I. Dasseville and M. Denecker, “Transpiling programming computable functions to answer set programs,” in. Jan. 2019, pp. 3–17, ISBN: 978-3-030-16201-6. DOI: 10.1007/978-3-030-16202-3_1.

[9] K. Claessen, J. Duregård and M. H. Pałka, “Generating constrained random data with uniform distribution,” *Journal of Functional Programming*, vol. 25, e8, 2015. DOI: 10.1017/S0956796815000143.

[10] A. Löscher and K. Sagonas, “Targeted property-based testing,” Jul. 2017, pp. 46–56. DOI: 10.1145/3092703.3092711.

[11] P. Vasconcelos. (May 2021). “Property Testing using QuickCheck,” [Online]. Available: https://www.dcc.fc.up.pt/~pbv/aulas/tapf/handouts/quickcheck.html.

APPENDICES

A Language Definitions (Code)

Here, we present the definitions for the languages in Haskell. Listing 4 displays the definition for the Simply-Typed Lambda Calculus and Listing 5 displays the definition of the PCF language which we used in this study.

```
1 data Type = TInt
2   | TBool
3   | TFun Type Type
4
5 data Expr = Id String
6   | Lambda (String, Type) Expr
7   | App Expr Expr
```

Listing 4: *STLC Code*

```
1 data Type = TInt | TBool
2   | TList Type
3   | TFun Type Type
4
5 data Expr = TrueE | FalseE
6   | And Expr Expr | Or Expr Expr
7   | Not Expr
8   | NumE Int
9   | Add Expr Expr | Mul Expr Expr
10  | Eq Expr Expr | Lt Expr Expr
11  | Nil Type | Cons Expr Expr
12  | Head Expr | Tail Expr
13  | IsNil Expr | IsList Expr
14  | Id String
15  -- name (param, pTy) rTy body
16  | Lam String (String, Type)
17      Type Expr
18  | App Expr Expr
19  | Let (String, Expr) Expr
20  | If Expr Expr Expr
```

Listing 5: *PCF Code*

B Properties (Code)

This appendix contains the code implementation of the properties used in the property-based testing. They are shown in Listing 6.

```
1 propCompare :: Expr -> TEnvironment ->
      Property
2 propCompare e nv =
3   typeOf e nv == typeOfBuggy e nv
```

```
1 propGenerate :: Type -> TEnvironment
      -> Property
2 propGenerate t nv = forAll (
      typedArbitrary t) $ \e ->
3   case typeOfBuggy e nv of
4     Right res -> res == t
5     _ -> False
```

```
1 propApply :: Type -> Type ->
      TEnvironment -> Property
2 propApply param body nv =
3   forAll (arbitraryLamAndApp param
      body) $ \(lam, arg) ->
4     case typeOfBuggy lam nv of
5       Right (TFun _ b) ->
6         case typeOfBuggy (App lam arg)
     nv of
7           Right t -> b == t
8           _ -> False
9       _ -> False
```

```
1 propMistyped :: Property
2 propMistyped = forAll mistyped $ \e ->
3   (case typeOf e basicEnv of
4     Left _ -> True
5     _ -> False) ==> case typeOfBuggy e
     basicEnv of
6       Left _ -> True
7       _ -> False
```

Listing 6: *Properties*

C Generators (Code)

This appendix displays the code for the general methods of the generators. Listing 7 displays the arbitrary type generator for the STLC language. Similarly, Listing 8 displays the arbitrary type generator for the PCF language. Listing 9 contains the implementation of the generation loop for any arbitrary expression.

```
1 arbitrary :: Int -> Gen Type
2 arbitrary i
3   | d <= 0 = pickOf [TInt, TBool]
4   | otherwise = do
5     x <- chooseInt (1, 3)
6     case x of
7       1 -> do
8         p <- arbitrary (i-1)
9         TFun p <$> arbitrary (i-1)
10      _ -> pickOf [TInt, TBool]
```

Listing 7: *STLC Type Generator*

```
1 arbitrary :: Int -> Int -> Gen Type
2 arbitrary d ld
3   | d <= 0 = pickOf [TInt, TBool]
4   | otherwise = do
5     x <- chooseInt (1, 4)
6     case x of
```

```
7      1 -> do
8        TList <$> arbitrary ld
9      2 -> do
10       p <- arbitrary (i-1)
11       TFun p <$> arbitrary (i-1)
12     _ -> pickOf [TInt, TBool]
```

Listing 8: *PCF Type Generator*

```
1 arbitrary :: Gen Expr
2 arbitrary = do
3   t <- arbitrary
4   e <- arbitraryExpr (depth, t)
5   maybe arbitrary return e
6
7 arbitraryExpr :: (Int, Type) -> Gen (
      Maybe Expr)
8 arbitraryExpr (i, t) = ...
```

Listing 9: *Expression Generator*

D Basic Environment (LC)

The basic environment in Listing 10 was used to facilitate more varied generation for the Simply-Typed Lambda Calculus:

```
1 basicEnv :: TEnvironment
2 basicEnv = [
3   ("tru", TBool),
4   ("fls", TBool),
5   ("not", TFun TBool TBool),
6   ("and", TFun TBool (TFun TBool TBool
      )),
7   ("or", TFun TBool (TFun TBool TBool)
      ),
8   ("1", TInt),
9   ("2", TInt),
10  ("5", TInt),
11  ("10", TInt),
12  ("20", TInt),
13  ("50", TInt),
14  ("100", TInt),
15  ("200", TInt),
16  ("500", TInt),
17  ("neg", TFun TInt TInt),
18  ("*", TFun TInt (TFun TInt TInt)),
19  ("+", TFun TInt (TFun TInt TInt)),
20  ("isZero", TFun TInt TBool),
21  ("isNeg", TFun TInt TBool),
22  ("toInt", TFun TBool TInt)]
```

Listing 10: *Basic Environment*

E Detailed Results for PCF

Tables 8 - 11 contain the more detailed results for the swiftness of the error detection with each of the properties for the test suite.

	a	b	c	d	e	f	g	h	i
avg	2	3	2	5	14	3	-	-	-
max	5	10	9	27	56	12	-	-	-

Table 8: *Results for* `CMP` *(PCF)*

	a	b	c	d	e	f	g	h	i
avg	2	3	2	6	13	3	-	-	-
max	6	11	6	55	69	13	-	-	-

Table 9: *Results for* `GEN` *(PCF)*

	a	b	c	d	e	f	g	h	i
avg	1	2	1	3	5	2	-	-	-
max	3	6	4	13	12	5	-	-	-

Table 10: *Results for* `APP` *(PCF)*

	a	b	c	d	e	f	g	h	i
avg	2	6	2	-	-	-	5	60	76
max	5	52	10	-	-	-	28	92	98

Table 11: *Results for* `MIS` *(PCF)*