TU Delft — Bridging the Gap in Software Verification: Agda Proofs to Haskell Test Case Generators

Student: Mateusz Olszewski
Supervisors: Jesper Cockx, Nathaniel Burke

1. Introduction

Agda2hs is a tool for translating a subset of Agda into readable Haskell code. Property-Based Testing (PBT) evaluates a function f against an algebraic specification of the form:

$$\forall x \in T : P(x) \to Q(f(x))$$

where P is a prerequisite invariant and Q is the consequent property expected to hold. While Haskell can execute this dynamically over arbitrary inputs via **QuickCheck**, Agda allows formal mathematical proof of the statement. Translating Agda proofs into executable PBT suites bridges the gap in software verification, empowering proof-writers with early correctness checks and Haskell developers with automated test generation. This work explores procedural creation of values of x, given Agda-defined type and some relevant property, such that $P(x)$ holds true.

The naïve solution would be to generate arbitrary $x \in T$, then filter based on predicate P. However, this quickly becomes unusable for non-trivial properties with few valid x values. A better solution would use the underlying property to guide the generation process.

2. Research Gap

Previous research in dependent-type generation, such as *Random Generators for Dependent Types* (Dybjer et al. [1]) and *Generating Good Generators for Inductive Relations* (Lampropoulos et al. [2]), introduce constraint-solving methodologies but leave research gaps

- **Language Boundaries:** Generation stays within the same language which is a proof assistant (e.g. Rocq) rather than one with any presence in industry.
- **Manual Overhead:** Implementation sometimes demands error-prone manual coding from developers.
- **Performance Profiles:** Operational metrics, i.e. runtime execution speed, reliability, and structural distribution, are not examined.

3. Research Question

Research Question:

How can preconditions defined in Agda be translated to an efficient, native QuickCheck generator in Haskell?

Sub-questions:

1. What form should the properties take to be compatible with the given procedure
2. How efficient are the created generators?

4. Algorithmic Architecture

For each property, its data constructors are *undone* by iterating over prerequisites and binding free variables (defined with ∀) one after the other. Before a generator is made, indices are selected as either generated or supplied. The latter appear in output function pattern matches, while latter are given in the final return call of each `do` block. This is specified with the number suffix: 1 - last index is generated; 3 - last two indices are generated, like `(Nat, ListNat)` values `(x, l)` in the second to last line on the right.

```
data IsSorted' : Nat → ListNat → Set where <=>
  empty : ∀ {x} → IsSorted' x [] -------------
  many : ∀ {x y ys} -------------------------
    → x ≤ y
    → IsSorted' y ys
    → IsSorted' x (x ∷ ys)
```

```
gen_IsSorted'_1 :: Nat -> SizedGenM ListNat
gen_IsSorted'_1 x 0 = return []
gen_IsSorted'_1 x size = do
  y <- gen_leq_1 x defaultSize
  ys <- gen_IsSorted'_1 y (pred size)
  return (x, x : ys)
```

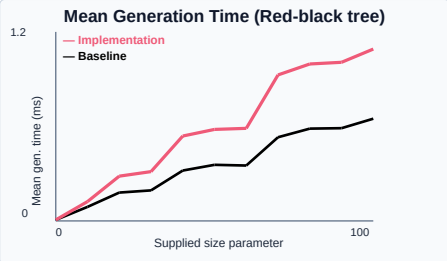
```
data IsSorted : ListNat → Set where <=========>
  sortedIsSorted : ∀ {x l} -------------------
    → IsSorted' x l
    → IsSorted l
```

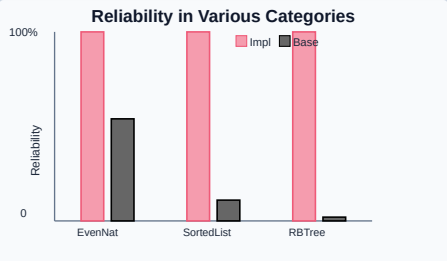
```
gen_IsSorted_1 :: SizedGenM ListNat
gen_IsSorted_1 size = do
  (x, l) <- gen_IsSorted'_3 size
  return l
```

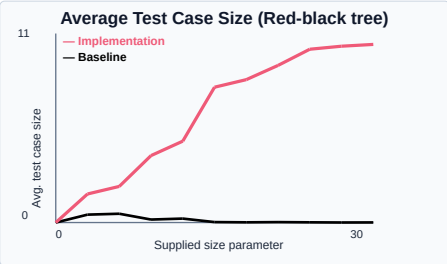
In `gen_IsSorted'_1`, in constructor `many`, the first argument is matched as `x`. In case there are more arguments, function clauses are unified according to the procedure in [2].

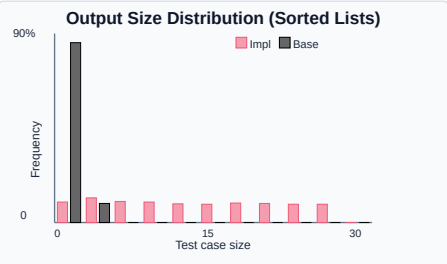
5. Results - Practical

Generators are evaluated for speed (time to generate a single test case), reliability (rate of returning success rather than failure states) and complexity (output size).









6. Results - Theoretical

The input form required for translation is:

```
data P : I1 -> I2 -> ... -> In -> Set where
```

with further requirements:

- Indices must belong to simple, concrete algebraic types (e.g., `Nat`, `Bool`). Dependent types or erasure (`@0`) are unsupported.
- Constraints must be defined using the `data` keyword; Records or functions are not permitted.
- Datatype parameters are not allowed.

In such a case, any subset of $\overline{I_i}$ can be generated; remaining indices are parameters. Example: for the property of being a red-black tree, generating `RBTree` given `Color` and `Nat`.

7. Conclusion

It is possible to efficiently generate test cases given an Agda property. This property must come in a specific input form, where generation and parametrization is expressed through simply typed indices.

While slightly slower due to overhead of the generalized procedure, the implemented generators are much better for reliably generating complex test cases. The difference is especially visible for non-trivial properties, where the naïve baseline solution does not provide meaningfully complex results.

Future work includes:

- investigating the possibility of using erased dependent types for additional expressivity of properties
- Allowing for parametrized datatypes.
- Extending the possible input form to records and functions.

Overall, I hope that this research will form another small step towards better verified software.

References
[1] Dybjer, P. et al. (2005) - *Random generators for dependent types*. Theoretical Aspects of Computin
[2] Lampropoulos L. et al. (2017) - *Generating good generators for inductive relations*. Proc. ACM Program. Lang

CSE3000 Research Project — Bachelor of Computer Science and Engineering
Faculty of EEMCS, Delft University of Technology (TU Delft) — June 24, 2026
Code Repository: `github.com/mateusz-z-olszewski/RP-Proofs-to-Generators`