Bridging the gap in software verification: Agda proofs to Haskell test case generators

How can preconditions in Agda be translated to a QuickCheck generator in Haskell?

Mateusz Olszewski[1]

Supervisors: Jesper Cockx[1]**, Nathaniel Burke**[1]

[1]**EEMCS, Delft University of Technology, The Netherlands**

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
June 21, 2026

Name of the student: Mateusz Olszewski
Final project course: CSE3000 Research Project
Thesis committee: Jesper Cockx, Nathaniel Burke, Annibale Panichella

Abstract

Property-based testing is a popular testing paradigm, especially common in the functional programming language Haskell through the QuickCheck library. On the other hand, Agda is a dependently-typed language used as a proof assistant and for software verification. Because of the need for software verification, Agda2hs was created as a tool for translating code between these two, combining proven functionality with practical execution.

This work explores making QuickCheck-compatible generators based on properties encoded in Agda. This should allow automating early checking of Agda proofs for correctness, as well as enabling conversion of formal specifications to executable tests for Haskell projects. A procedure is presented that permits creating advanced generators and checkers, and it is later evaluated in terms of multifaceted performance metrics.

1 Introduction

Software is undeniably a crucial foundation of modern technology. With so many aspects of our lives digitized, it is borderline impossible to avoid its ubiquitous presence, and serious weight is placed on the assumption that it will work reliably. Computer bugs that put in jeopardy our most important systems erode our trust in the system and can cause massive financial damages. Hence, the need for software verification arises.

Software verification can be divided into dynamic verification, which includes testing, and static verification - proving properties of a program before it is even run. The former is widely used in industry, ranging from focused unit tests to complex full system tests. However, as E. W. Dijkstra once said, "Program testing can be used to show the presence of bugs, but never to show their absence!" [1]. Static software verification comes to fill in this gap, with definitive proofs of properties. The most common approach is a simple but sound type system (present in Java, Rust, Go, etc.), but for popular declarative languages, it often stops at just these basic functionalities. Agda [2] is a pure functional programming language with a type system rich enough to be able to encode arbitrary properties of its programs. This contrasts with Haskell, another pure functional language, which lacks dependent types in the base implementation, but enjoys some, albeit limited, industry usage.

The research question I answer in this work is *How can preconditions in Agda be translated to efficient QuickCheck generators in Haskell?* To bridge a gap between production-ready code and logically provable theorems, Agda2hs [3] was created. It is a tool for translating a subset of Agda to Haskell, while erasing some information which cannot be encoded in Haskell. In this thesis, I focus on what to do with that erased information and how to transform it into dynamic generators for property-based tests suitable for execution in a Haskell runtime. Two sub-questions are considered: *What form can Agda preconditions take to be compatible with translation* and *How efficient are the created generators?*

Property-based testing (PBT) automatically checks if relations hold for pure operations on arbitrary input data. To check properties of more complex structures, generators of pseudo-random test cases for relevant data types are necessary. Furthermore, it might be needed that inputs satisfy some properties themselves, to check an implication between two predicates. A naïve solution would be to generate test cases of some type (a given shape), then filter them based on whether the prerequisite is satisfied; however, this might quickly become inefficient for larger sizes or more complex properties.

Hence, in this work I address the need for a better system that efficiently creates test cases for Haskell's QuickCheck, as defined by properties specified in a proof assistant - Agda. I begin by introducing relevant terms, methodology, metrics for evaluating the project, previous work, and the formal problem statement. Later, I present the main idea behind generation of data and reason about the capabilities of the procedure. That section is followed by implementation details and results. Finally, outcome discussion and future work recommendations conclude this work.

2 Preliminaries

2.1 Agda & Haskell

As briefly introduced before, Agda and Haskell are two purely functional languages. While this term is usually defined in very broad strokes, it means that functions are *first-class* objects in these programming languages, and can be passed as parameters, returned, or otherwise manipulated just like other values. Additionally, because these languages are pure, no imperative-style mutation is possible and composing functions serves as the main way of getting work done. To further support this style of computation, both languages have elaborate type systems, which encode exactly which operations are permitted between types (e.g. you cannot pass an integer where a list is expected), which helps to correct mistakes early. Agda's type system is richer than Haskell's; it extends Martin-Löf's intuitionistic type theory [4], where types also become first-class, hence allowing for dependent types. They are the key source of Agda's capabilities because they allow encoding properties as types. The standard example [5] is a list aware of its size:

```
data Nat : Set where
  Z : Nat
  S : Nat → Nat

data Vec (a : Set) : Nat → Set where
  [] : Vec a 0
  _::_ : {n : Nat} → a → Vec a n → Vec a (S n)
```

Here, `a` is a parameter, while `Nat` is an index. The former stays the same across all uses of `Vec` in this definition and is similar to Java's generics (even though already more powerful as non-type values are allowed), while the latter can differ (taking values `Z` or `(S n)` in this example). `Vec` is implemented as a linked list; `[]` is the empty list, while `::` is a binary operator that prepends an element of type `a` to pro-

duce a list with length extended by one (the S operation on naturals, short for successor).

The above is an *intrinsic* definition - one that makes the properties of a data type embedded in its own signature. Conversely, *extrinsically* encoded properties exist as separate algebraic data types, with indices describing the relevant constraints:

```
data List (a : Set) : Set where
  [] : List a
  _::_ : a → List a → List a

data HasLength : Nat → List Nat → Set where
  NilIs0 : HasLength Z []
  ConsIsSuc : ∀ {n x xs} → HasLength n xs →
      HasLength (S n) (x :: xs)
```

Curry-Howard's isomorphism [6] states that there is a correspondence between dependent types with their values, and logical proofs. A type is treated hence as a proposition, an inhabitant as a proof, etc. For example, for a proposition `HasLength 1 (42 :: [])`, the relevant proof is `ConsIsSuc {x = 42} NilIs0`. A few examples are presented below:

Proposition	Type
Proof	Inhabitant / Value
Truth	Unit type ($\top$)
Falsity	Empty type ($\bot$)
Implication ($A \implies B$)	Function type (`A → B`)
Conjunction ($A \wedge B$)	Pair/Tuple `(A, B)`

2.2 Agda2hs

Agda2hs [3] is a tool for translating a subset of Agda into valid and readable Haskell. As opposed to the main, Haskell-based backend *MAlonzo* made to execute arbitrary Agda, this tool's goal is to produce clean and readable code matching Agda's specification, but in a programming language much more suitable for production (industry) use. For example, properties of a complex procedure can be proven in Agda, then translated to a Haskell library to be used in a real-world use case, with confidence in its correctness.

2.3 Red-Black Trees

Red-black trees are a popular implementation of *self-balancing* trees, that is maintaining a relatively small height compared to their node count. They were chosen as an example because of their structural invariants (below): they are simple to encode, but even basic operations to maintain them, like insertion, are notoriously difficult to implement[1] and possibly bug-prone. Hence, they are a perfect candidate for property-based testing with automatic test-case generation.

Red-black trees (Figure 1) are binary trees with an additional node annotation - *colour* - which can be either red or black. This is achieved through following three invariants:

[1]The most popular Hackage library for red-black trees uses around 150 lines of code to perform the operation

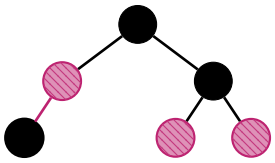

Figure 1: Example red-black tree

1. The root is black,
2. Red nodes have no red children,
3. For any node, all paths to leaves have the same number of black nodes (this value is called *black height*).

While usually leaves are defined to all be black, here they will not be coloured at all and will not be counted in regards to tree black height; this will not change program or reasoning logic.

For simplicity, the property that a red-black tree is a binary search tree (BST) will be hereinafter omitted entirely.

2.4 Property-Based Testing

Property-based testing (PBT) is a process of checking whether pure functions satisfy simple properties or algebraic relationships [7], like below:

```
prop_doubleIsEven :: Nat -> Bool
prop_doubleIsEven n = isEven (double n)
```

Here, the program checks that for any *arbitrary* natural number, doubling it will result in an even number. Writing such tests is intuitive and valuable because it can serve as an executable, algebraic specification of code's behaviour rather than an abstract collection of tests. Any pure function's desired behaviour can be understood given a rich enough PBT suite.

Property-based tests often require inputs of a type that has been defined by the user, rather than one of the multiple built-in types. Notably, even authors of the original *QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs* paper [8] that sparked the idea of PBT admit that automatically producing test data is not an easy task. Pseudo-random generation of instances of arbitrary types has been omitted from QuickCheck since its inception, as it would make the tool needlessly heavy and complicated to use. Another reason was that "it seems to be very hard to construct a generator for a type, without knowing something about the desired distribution of test cases". This observation motivates the approach I took in this work. Properties defined by the user in Agda are used to guide test-case generation so that only relevant inputs are produced. This represents exactly the missing knowledge about case distribution (in a possibilistic, rather than probabilistic sense) the authors mentioned.

2.5 Formal Problem Statement

In formal language, property-based testing of a function f involves checking through tests whether the following statement holds:

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

In this work, I will focus on generating inputs x such that $P(x)$ is true. An Agda predicate is expressed as $T \to Set$, but for the purposes of this paper let us extend it to be

$$P : I_1 \to I_2 \to \ldots \to I_n \to Set$$

where this predicate is *indexed* by its *indices* I_i. In that case, each I_i expresses some inherent property of x, for example black height or parent colour of a red-black tree, or length of a list. The goal is for any such predicate (expressed as Agda's data type via `data ... where` syntax) to make a generator of the form

$$G : I_1 \to I_2 \to \ldots \to I_{n-1} \to M_{I_n}$$

Here, M represents some monad[2], e.g. Haskell QuickCheck's `Gen` which takes a random seed and desired result size and outputs an instance of I_n. The number of indices can be zero, as it is already the case in many QuickCheck-defined generators, like `arbitrary :: Gen Int`, which yields random integral numbers.

2.6 Related Work & Research Gap

P. Dybjer et al. in *Random Generators for Dependent Types* explore making simple generators in Agda [9], then investigate how to proceed with dependent types that encode proofs. Similarly, L. Lampropoulos et al. investigate making efficient generators for inductive types in Rocq (formerly Coq) with an extensive constraint solving framework in their work *Generating Good Generators for Inductive Relations* [10]. Other work [11] further analyses the general rules for encoding the properties, as well as how pseudo-randomness needed for making the cases works in a pure language. However, there is a clear gap in the current research on property-based generators:

- Generation stays within the same language, which is a proof assistant rather than a production-ready language with some (even if small) industry usage,
- Implementation steps are in many cases to be implemented by programmers by hand,
- Performance (execution speed, reliability, output complexity) of the generation process in terms of relevance for its actual purpose - thorough PBT testing - is rarely explored.

Hence my aim is to attempt to fill this gap with an automatic framework for translating properties to generators by integrating the procedure seamlessly and efficiently into the existing *Agda2hs* tool.

3 Generator Creation

The main purpose behind this work is to create generators of properties encoded in Agda. This section presents the capabilities, requirements, limitations, as well as the process behind this conversion.

[2] A monoid in the category of endofunctors; in simpler terms, a functional programming concept that allows for implementing concepts like state, context, or error-based control flow in a way explicitly encoded through types.

3.1 Input Form

The input to the translation procedure should have the following structure:

$$\begin{aligned}
\text{Name} \quad & n \in \text{String} \\
\text{Value} \quad & v ::= \mathbf{SC}(n) \mid \mathbf{Con}(\bar{v}) \\
\text{SimpleCons} \quad & c ::= \mathbf{SC}\big(\text{ name} : n,\ \text{parameters} : \bar{d}\big) \\
\text{SimpleData} \quad & d ::= \mathbf{SD}\big(\text{ name} : n,\ \text{constructors} : \bar{c}\big) \\
\text{PropCons} \quad & pc ::= \mathbf{PC}\big(\text{ name} : n,\ \text{variables} : \bar{d}, \\
& \qquad \text{prerequisites} : \overline{(p, \bar{v})},\ \text{idx_args} : \bar{v}\big) \\
\text{Property} \quad & p ::= \mathbf{Pr}\big(\text{ name} : n,\ \text{indices} : \bar{d}, \\
& \qquad \text{constructors} : \overline{pc}\big)
\end{aligned}$$

For example, for the case of Red-Black trees, the definitions below are *SimpleData* types or values.

```
data Color : Set where
  Red Black : Color
{-# COMPILE AGDA2HS Color #-}

data RBTree : Set where
  Leaf : RBTree
  Node : Color → RBTree → Nat → RBTree → RBTree
{-# COMPILE AGDA2HS RBTree #-}

-- Example function on simple data types
insert : Nat → RBTree → RBTree
insert n Leaf = Node Black Leaf n Leaf
insert n (Node c tl x tr) = ...
{-# COMPILE AGDA2HS insert #-}
```

Here, `Red` or `Black` are simple constructors, while `Color` or `RBTree` are simple data types. On the other hand, a property corresponds to an Agda data type like the following:

```
data IsEven : Nat → Set where
  zeroIsEven : IsEven Z
  sucSucIsEven : ∀ {n} → IsEven n
    → IsEven (S (S n))
{-# COMPILE AGDA2HS IsEven generator #-}
```

Under this system, the Agda definition of a natural number being even is represented as a *Property* from the input form above, with name *IsEven*, a single-element list of indices $\{\texttt{Nat}\}$, and two constructors. The latter constructor has a set of variables $\{n\}$, a prerequisite $(\texttt{IsEven}, n)$, and a single index argument: $idx\_args = \{S(S(n))\}$. This property is annotated with the new compiler pragma `generator`, which will produce a function like

```
type SizedGenM a = Int -> GenM a

gen_IsEven_1 :: SizedGenM Nat
gen_IsEven_1 size = ...
```

where `GenM` is the monad M_T described earlier.

3.2 Extrinsic and Intrinsic Encoding.

Extrinsic encoding of properties takes the form of a separate datatype, rather than modifying an existing one. Because the purpose is to translate a datatype (and later make its generator) to Haskell, some reasons appear that justify that approach

over intrinsic encoding, which involves modifying an existing property with erased `@0` [12; 13] values. The latter approach requires us to modify an existing data type, which might not be feasible for library types like a list or a natural number. Currently, Agda2hs is built around the idea of single compilation instruction (pragma) per declaration, and would require additional work to allow for simultaneous translation of a data type and its generators. Finally, extrinsic encoding allows for better separation of concerns as well as making multiple generators of the same type. Hence, in this work only extrinsic encoding will be explored.

3.3 Input Size and Failure

The size parameter is present in QuickCheck by default. It lets us limit the generation such that the process does not run off to infinity in terms of execution time and memory. To use it, `sized (\n -> ...)` can be used; for a single call to `generate`, this parameter's value is 30 by default, while for a test suite invoked with `quickCheck`, it ranges from 1 to 100. Notably, it does not specify an exact value of that particular metric, but rather an upper bound. The metric in question might vary depending on the implementation, but in this paper, for recursive algebraic types it corresponds to the count of the non-terminal constructed members of it. For example, the size of a binary tree is the number of its nodes, while for a (linked) list it is its length.

A generator can end up in a *failure state* if no possible test case can be generated for the given parameters, within the supplied size bound. Such a state can be encoded as e.g. `Nothing` if the result is wrapped in a `Maybe` or by calling QuickCheck's `discard` function.

3.4 Translation Framework

In general, the procedure relies heavily on recursion: when a property depends on a prerequisite (like `n` needing to be even for `(S (S n))` to be even too), the needed value is generated through a recursive call. Notably, for each property, multiple variants exist. Each index can either be considered as provided, or to be generated. Let us define the concept of *order*, defined as the decimal value of the binary encoding of whether parameters are to be produced (see table below). The Agda property of a tree being red black, which will be given fully in the next section, has the following signature:

```
data IsRedblack' : Color → Nat → RBTree → Set
  where ...
```

with indices:

- `Color` - the colour of tree's parent node
- `Nat` - the black height of the tree
- `RBTree` - the very tree referenced above

Some example variants, from all 2^3 possible combinations:

ord_{10}	ord_2	*Description*
1	001	Generator of trees given parent colour and black height
3	011	Generator of (black height, tree) pairs
2	010	Calculator of black height given a tree (and some parent colour)
0	000	Checker if a tree is Red-Black

In the last case, it might seem odd that nothing is generated. However, recall that the data type signature

$$I_1 \to \cdots \to I_n \to Set$$

can be rewritten as

$$I_1 \to \cdots \to I_n \to \top \to Set$$

without loss of generality. Then, only the unit type ($\top$ in Agda, `()` in Haskell) is set to be generated. Such a checker would have a signature

```
gen_IsRedblack'_0 :: Color -> Nat -> RBTree
                                  -> SizedGenM ()
```

where returning a *"test case"* of `()` is considered as truth, contrasting with falsity represented by yielding a failure state.

3.5 Performance Metrics

As already mentioned in the introduction, evaluating generator performance is a crucial part of this work. Because PBT runs each test multiple times (at least 100 attempts in Haskell QuickCheck) for different test cases with the goal of finding bugs, three aspects of performance can be identified:

Speed – the execution speed of the resulting generator (not the translation speed of Agda2hs or related tools). Because PBT runs so many iterations, it is important to optimize it as any speed-ups in this regard are quickly noticeable. This can be measured in test cases per second or, inversely, time to generate one test case.

Reliability – as explained in section 3.3, sometimes cases end up in a failure state, where no instance can be found within some size and under specific constraints. QuickCheck's default behaviour is to attempt rerunning the test until 100 test cases are produced, or 1000 failures are encountered – leading it to give up. Low reliability leads to long execution time with few actual tests ran, making the process impractical. This metric is measured as the percentage of successful generations out of all attempts.

Complexity – as the goal of PBT is to find bugs given arbitrary inputs, test cases must be sufficiently complex to trigger as many as possible branches of the algorithm under test. This usually corresponds to produced values being varied and relatively large compared to their upper size bound. Complexity is non-trivial to analyse quantitatively; test case size is one aspect, but ultimately the value to measure is how many bugs were detected by a QuickCheck run given some algorithm (with introduced errors) and a property expected to hold.

4 Algorithm Details

The previous section introduces in broad strokes the main idea of this work: that for any property described with the input form, generators of any combination of indices can be produced. Let us consider the property that a tree (with node colour encoding) is red-black. To translate the definition given by Paraskevopoulou et al. [11] from Rocq to Agda:

```
data IsRedblack' : Color → Nat → RBTree → Set
    where
  LeafIsRB : ∀ {c} → IsRedblack' c Z Leaf
  RedNodeIsRB : ∀ {tl tr n h}
```

```
    → IsRedblack' Red h tl
    → IsRedblack' Red h tr
    → IsRedblack' Black h (Node Red tl n tr)
  BlackNodeIsRB : ∀ {c tl tr n h} →
    IsRedblack' Black h tl → IsRedblack' Black h tr
    → IsRedblack' c (S h) (Node Black tl n tr)
{-# COMPILE AGDA2HS IsRedblack' generator #-}

data IsRedblack : RBTree → Set where
  IsRB : ∀ {h t} → IsRedblack' Red h t →
      IsRedblack t
{-# COMPILE AGDA2HS IsRedblack generator #-}
```

In this example, the internal property `IsRedblack'` is indexed by the colour of its parent node, the black height as `Nat`, and finally the tree itself. Being indexed by multiple arguments means that all of these values have relevant values at the same time: the `Nat` describes precisely the black height of the following `RBTree`.

Algorithm 1 From constructor to a recursive do expression

function UNDOCONSTRUCTOR
  $\overline{vars} \leftarrow$ variables ▷ Initialization from the
  $\overline{pres} \leftarrow$ prerequisites ▷ input form: subsection 3.1
  for $(prop, \bar{v})$ **in** $\overline{pres}$ **do**
    $\bar{b} \leftarrow \bar{v} \setminus \overline{vars}$ ▷ Were bound before
    $\bar{d} \leftarrow \bar{v} \cap \overline{vars}$ ▷ Being bound now
    produce a relevant do **binding**
    $\overline{vars} \leftarrow \overline{vars} \setminus \bar{v}$
  end for
  for var **in** $\overline{vars}$ **do** ▷ Variables remaining free
    produce `var <- arbitrary`
  end for
  $match \leftarrow \ldots$ ▷ based on idx_args of the constructor
  return $match$ and the complete do binding
end function

In the above pseudocode, *produce a relevant* do *binding* means assigning values to all variables d being bound in that expression (as a tuple) to values generated by the relevant generator. Because generators can be constructed freely in terms of which index is an argument or a result, any relevant combination is available for a recursive call. This is ultimately used in the top-level algorithm for each constructor of the property type, as seen in the following pseudocode.

Algorithm 2 Top-level algorithm for translation

function PROPERTYTOGENERATOR
  $C \leftarrow \emptyset$ ▷ Resulting function's clauses
  for pc **in** $\overline{cons}$ **do**
    $C \leftarrow C \cup \{\text{UNDOCONSTRUCTOR}(pc)\}$
  end for
  $(C_0, C_S) \leftarrow$ **split** C **on** being recursive
  $\ldots$ ▷ Add short circuit calls to C_0 equivalents for C_S
  return $\text{UNIFY}(C_0) \cup \text{UNIFY}(C_S)$
end function

This procedure returns clauses of the generator function, equipped with pattern matches on indices and the `size` parameter, and of course the relevant expression. Sometimes, multiple resulting expressions are combined. The provided size is only an upper bound and not a hard-coded value, so calls can short circuit with probability $1/(\text{size}+1)$ to a clause with the same parameters, but with size set to zero; this will produce values uniformly distributed in terms of size, matching many default generator implementations in QuickCheck. The purpose of the *unification* procedure is to combine clauses that can match the same arguments together, then pick arbitrarily one of the results. These features are possible using functions analogous to `frequency` and `oneof` from QuickCheck, respectively.

4.1 Worked Example

Let us consider the idea of generating valid Red-Black trees. Below is an Agda encoding of the property that a tree is Red-Black:

```
data IsRedblack : RBTree → Set where
  IsRB : ∀ {h t} → IsRedblack' Red h t →
      IsRedblack t
{-# COMPILE AGDA2HS IsRedblack generator #-}
```

The only constructor refers to another property - `IsRedblack'` - but binds the first index, specifying that the tree must be compatible with being a subtree to a red node. This ensures invariant (1) - the root must be black. This shell-like property is useful as it allows use as a simple predicate taking one argument, such that parameters like black height need not be supplied explicitly. It has only a single constructor, `IsRB`. The procedure UNDOCONSTRUCTOR for it begins with these values:

$$\overline{vars} \equiv \{h, t\} \quad ; \quad \overline{pres} \equiv \{\texttt{IsRedBlack'}_{\text{Red},*,*}(h, t)\}$$

Here, the sole prerequisite has been written with the last two of indices as $*$ symbols to emphasize that these indices were not bound before. As this *binds* all variables, none remain to be considered *arbitrary*, and the resulting Haskell function is:

```
gen_IsRedblack_1 :: SizedGenM RBTree
gen_IsRedblack_1 size = do
  (h, t) <- gen_IsRedblack'_3 Red size
  return t
```

As seen above, the notion of arguments and results is inverted when transforming a property into a generator. Parameters h and t, which were used as inputs to a prerequisite, now are produced by the generator of that property.

```
data IsRedblack' : Color → Nat → RBTree → Set
    where
  LeafIsRB : ∀ {c} → IsRedblack' c Z Leaf
  RedNodeIsRB : ∀ {tl tr n h}
    → IsRedblack' Red h tl
    → IsRedblack' Red h tr
    → IsRedblack' Black h (Node Red tl n tr)
  BlackNodeIsRB : ∀ {c tl tr n h} → (pre :
      IsRedblack' Black h tl)
    → IsRedblack' Black h tr
    → IsRedblack' c (S h) (Node Black tl n tr)
{-# COMPILE AGDA2HS IsRedblack' generator #-}
```

The top level algorithm is given these three constructors:

$$cons \equiv \{\texttt{LeafIsRB}, \texttt{RedNodeIsRB}, \texttt{BlackNodeIsRB}\}$$

Undoing the non-recursive leaf case produces a value for any supplied colour. As there are no prerequisites, this is turned into a trivial return statement:

```
gen_IsRedblack'_3 c 0 = return (Z, Leaf)
```

For one of the two recursive constructors – `BlackNodeIsRB` which describes how black nodes can be formed – the algorithm recognizes these general inputs:

$$\overline{vars} \equiv \{tl, tr, n, h\}$$
$$\overline{pres} \equiv \{\texttt{IsRedBlack'}_{\texttt{Black},*,*}(h, tl), \texttt{IsRedBlack'}_{\texttt{Black},h,*}(tr)\}$$

It is worth noting in the Haskell result below that recursive calls are made to generators of different orders; the discrepancy comes from whether the variable h was considered free before a prerequisite using it was iterated over. As explained at the beginning of section 4, for clause undoing a recursive constructor, it should sometimes short circuit to a non-recursive call instead. Hence, the `freq` function is used, which picks from a list of generators with weighted probabilities.

```
gen_IsRedblack'_3 c size = freq [
  (1, gen_IsRedblack'_3 c 0),
  (size, do
    (h, tl) <- gen_IsRedblack'_3 Black (size `div`
        2)
    tr <- gen_IsRedblack'_1 Black h (size `div` 2)
    n <- lift arbitrary
    return ((S h), (Node Black tl n tr)))]
```

What follows finally is the *unification* of remaining constructors, grouped by the size argument they match on, e.g.:

```
gen_IsRedblack'_3 Black size = unR
gen_IsRedblack'_3 c     size = (unB c)
```

is unified to produce:

```
gen_IsRedblack'_3 c@Black size = unR | (unB c)
gen_IsRedblack'_3 c       size = (unB c)
```

This unification algorithm is similar to the one presented by Lampropoulos et al. [10]

4.2 Extending to Dependent Types?

As mentioned before, dependent types like U_T shown below are not permitted as indices for properties to be generated.

```
data P : (t : T) → (@0 u : U t) → Set where ...
```

The first reason is that for types with dependently typed indices like the one presented above, making a generator producing t given U_t is a difficult or perhaps even impossible task. This makes generators yielding values on which types of given values depend difficult. Furthermore, as dependently typed values are not allowed in Haskell anyway, they would have to be erased (marked in code using `@0`), making hence the encoding of `P` partially intrinsic. This work already does not consider intrinsically defined properties, even with only simple indices. Hence, dependently typed erased arguments might be a consideration later, but only if some future work is investigated beforehand.

5 Results

As the research question - *How can preconditions in Agda be translated to efficient QuickCheck generators in Haskell?* - touches on both the theoretical (*How?*) and practical (*efficiently*) aspects of the process, it is important to consider them both, then assemble the findings together

5.1 Theoretical

This paper postulates that **automatic translation of Agda properties to efficient Haskell generators is possible** for the broad collection of properties that can be expressed in the input form described earlier. This permits creating advanced generators for common use cases, i.e. Red-black trees, even natural numbers, BSTs, sorted lists, etc. Notably, dependent types are not permitted among property's own indices, as explained in section 4.2. Furthermore, *calculators* of properties like black height for a given red black tree, as well as checkers of a property are allowed. Finally, checkers with Haskell signature `... -> SizedGenM ()` are possible, which will generate a success state of `()` if indeed a property is satisfied, or a failure state otherwise. This allows for translating an entire property, in the form of an implication, together: the antecedent (left side input) is translated to a generator, while the consequent (right side conclusion) is translated as a checker. Entire test suites could be hence defined in Agda and ran as Haskell executable code.

An example of a type not yet possible to be generated using this procedure is a generalized variant of `HasLength` from subsection 2.1, where the type of list's contents is not limited to `Nat`. Implementing a way to define the type parameter as one supporting generation of arbitrary values is left for future work on this topic.

5.2 Practical

For a random data generator to be usable in practice, it needs to produce test cases quickly, reliably, and of sufficient complexity to actually find subtle bugs in implementation. The *baseline* for any data type is the naïve solution of first producing instances of the desired simple algebraic data type [9], then filtering them based on a predicate made from the property [11]. While this guarantees validity, the generator will be far from reliable, as for complex properties satisfied by relatively few values, most cases will be rejected. Furthermore, the produced values will most likely be too trivial to serve their actual purpose. Below, generators produced from Agda-encoded properties, subsequently referred to as *implementation*, will be compared with the baseline in terms of speed of generation, reliability, and test case size.

Generation time for red-black trees is shown on Figure 2. For both solutions, generation time increases approximately linearly with the supplied size parameter.[3] In general, implementation incurs approximately 80% overhead in time execution. This is caused by additional complexity from wrapping the implemented computation in a `MaybeT` monad for

[3]The "staircase" pattern appears around powers of two as both solutions use `size `div` 2` for recursive calls, which results in these places in a sharp increase in result size, and hence generation time.

generality, and the lack of problem-specific optimizations. However, baseline results in this case are not filtered for being valid red-black trees. Considering that for larger sizes, the probability of randomly making a valid red-black tree is close to zero, the quickly generated results using the naïve approach are almost guaranteed to be unusable as PBT test cases.

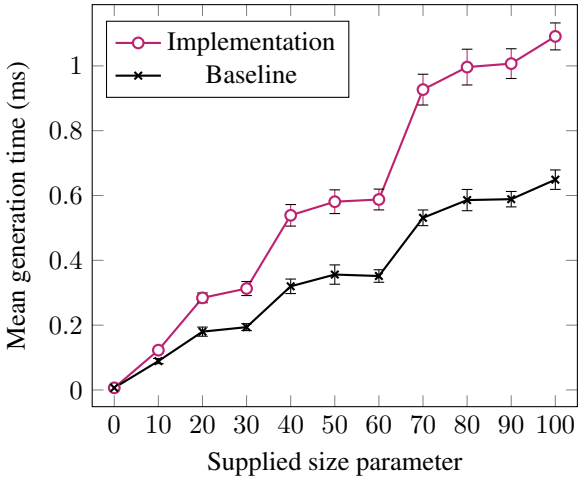

Figure 2: Generation time per test case; Error bars indicate $\pm 2\sigma$.

A pronounced difference can be seen in Figure 3, which shows that implementation's test cases are on average half the requested size, matching what would be expected from a uniform distribution. On the other hand, baseline produces appropriate results only for smallest sizes. For attempts to produce a test case with size larger than 12, the probability of succeeding is negligible, resulting in an average of zero.

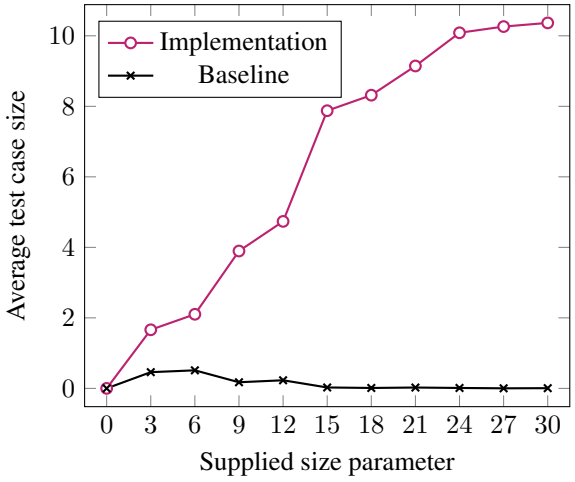

Figure 3: Generator complexity - average test size (Red-black tree generation; sample size for each calculated average: 2000; failures counted as 0)

Complexity is also presented in Figure 4. There, reliability in three different categories is shown: red-black trees, sorted lists, and even naturals. In each, reliability of the implementation is 100%, as in each problem a valid value can be generated for any size, and the generated Haskell code uses custom backtracking selectors `freq` and `choose` that will attempt to select a different option if an intermediate result is a failure state. For the baseline, reliability is much more limited. In the problem of generating even naturals, the result is close to 50% - matching expectations as half of all natural numbers are even. For more complex properties, the baseline results are even worse. While it might appear that in the case of generating sorted lists using that approach, 10% should be valid, this is not the full picture: most of these cases have few elements. Figure 5 shows that for the baseline, barely one in ten attempts makes a sorted list with at least 3 elements. Such trivial test cases might not be enough to reveal bugs in complex functions. On the other hand, the implementation produces values uniformly distributed in terms of size.

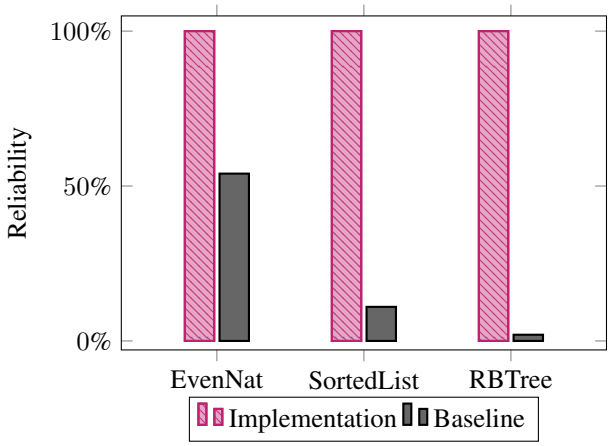

Figure 4: Reliability in various categories (generator size set to 30).

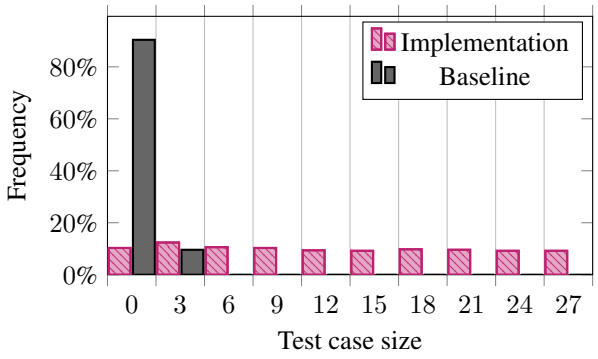

Figure 5: Generator output complexity: Sorted List. Failures not counted, retrying until success found.

5.3 Limitations

There exist some limitations concerning the implemented generators. One such issue comes from the distribution of samples provided in the data. Below, two encodings of the sorted list property are presented, along with their corresponding translated generators.

```
data IsSorted : List Nat → Set where
  empty : IsSorted []
  one : ∀ {x} → IsSorted (x :: [])
  many : ∀ {x y ys} → IsSorted (y :: ys) → x ≤ y
      → IsSorted (x :: y :: ys)
```

Which produces this code:

```
gen_IsSorted_1 0 = choose [
  return [],
  do
    x <- arbitrary
    return (x : [])]
gen_IsSorted_1 size = do
  l <- gen_IsSorted_1 (pred size)
  case l of
    (y : ys) -> do
      -- produces x given some y such that x <= y
      x <- gen_le_2 y size
      return (x : y : ys)
    _ -> mzero
```

This generator will fail to produce a result around half the time because if in the base case the result `[]` is chosen, the prerequisite list will not be in the required shape `(y :: ys)`. Additionally, another problem arises: sorted list's upper bound is its last element, which, if generated first, significantly limits values in it. This is an example of a list generated when the tail is bound before the head: `[0,0,0,...,0,0,0,0,1,1,6,8,14,21,23,29]`. Here, the initial `0`s take up almost the entire list.

On the other hand, by adding a piece of additional information into the type (then proceeding with an auxiliary data type to specify ∀ arguments) we achieve this:

```
data IsSortedGood' : (lowest : Nat) → List Nat →
    Set where
  empty : ∀ {x} → IsSortedGood' x []
  many : ∀ {x y ys} → x ≤ y
    → IsSortedGood' y ys
    → IsSortedGood' x (x :: ys)

data IsSortedGood : List Nat → Set where
  ItIs : ∀ {x l} → IsSortedGood' x l
    → IsSortedGood l
```

Which translates into this code:

```
genIsSortedGood' :: Nat -> SizedGen [Nat]
genIsSortedGood' x 0 = oneof [
  return [],
  return [x]]
genIsSortedGood' x size = do
  y <- gen_le_1 x
  ys <- genIsSortedGood' y (pred size)
  return (x : ys)

genIsSortedGood size = ...
```

This never returns `Nothing`, and produces large and varied lists one could expect from a generator:

```
[13,39,62,86,96,112,135,149,158,164,171,189,191,
209,224,224,237,243,255,265,273,274,279,284,291,
296,296,299,299,301,302]
```

While there always exist some concerns, like list's uniform step distribution, they can be deemed irrelevant as they would arise in hand-crafted generators too.

6 Responsible Research

This paper is concerned with theoretical considerations of capabilities of computer science. It proposes an algorithm for generation of abstract data. No human data was ever collected or processed for purposes of this work.

While essentially theoretical, this work consists of implementing the described algorithms in Haskell, for purposes of testing the performance of this approach. Keeping in mind best practices in terms of reproducibility and verifiability, the code is available at `https://github.com/mateusz-z-olszewski/RP-Proofs-to-Generators`, where it can be inspected and downloaded to run at users' own discretion. All programs and libraries used to achieve these results are free and open source.

Artificial Intelligence (AI) – including Large Language Models – was used in the project for the purposes of spellchecking, LaTeX assistance, and retrieving information about Agda or Haskell functions that were underdocumented or hard to find using other tools. AI was not used for writing the text of this work, summarising academic papers, planning implementation, or writing Haskell code.

7 Conclusion and Future Work

This paper shows how preconditions in Agda can be translated to efficient QuickCheck generators in Haskell. The details on which properties can be encoded and what form they should have is presented in section 3. Section 4 elaborates on the internal process of translation and concerns when using the process. The theoretical and practical results are presented in section 5; they investigate how *efficient* the translated generators truly are, splitting this vague term into concrete metrics.

The theoretical findings exceeded the original scope of this work. Because of how general the main conversion algorithm is, full freedom is granted in terms of whether a property index is to be supplied or generated. Hence, not only simple generators are possible, but also complex multi-output ones and checkers are available to users. For example, for property $x < y$, the user can specify if they need a generator of y bigger than x, x smaller than y, or pairs (x, y) ; $x < y$. Finally, a checker is possible, that returns a success state if x is in fact smaller than y (up to some maximum depth of computation, to not run too long on extremely large structures).

The practical performance was investigated in terms of three aspects: speed, reliability, and test case complexity. Due to the slight overhead of the general solution, the automatic implementation was slower, with $\sim 2\times$ increase in time per test case. However, due to the careful undoing of property constructors to produce valid data, this solution was clearly better than the baseline in terms of reliability (returning a valid test case, not a failure state). The difference reached over $1000\times$ for more complex properties and bigger desired

output sizes. Finally, test cases were found to be drastically larger than the baseline for less trivial properties.

Based on the findings above, it will be possible to use the tool in the future when full integration with Agda2hs is completed. This should allow writers of proofs in Agda to verify early that an implication between properties is actually true without spending futile effort on proving falsity. On the other hand, Haskell users will be able to use mostly proven functions translated using broader Agda2hs and make sure any small modifications to the results are still correct. This will happen with little overhead, as an entire collection of Agda proofs can be easily converted into a Haskell QuickCheck test suite because both generators and checkers can be made using this method.

Further work on this problem includes investigating how to translate intrinsically defined properties, or allow for parameters in data types. Dependently typed erased indices could be a valuable addition, but additional theoretical research is needed for specifying the implementation. Translating Agda's records or functions is also a possible direction for exploration. Additionally, this work involves only possibilistic analysis of result distribution (except for result size) rather than probabilistic. Finally, adding support for function application in prerequisites or property index values is surely a useful additional functionality that will make using the tool more practical.

To conclude, it is possible to translate Agda properties, supplied in a reasonable input form, to efficient QuickCheck generators in Haskell. This should enable users of both Agda and Haskell to work efficiently and harness the opportunities given by the proving power of one and industry viability of the other. Hopefully, this work will be another small step in bridging the gap in software verification, for the sake of better made, tested, and proven software.

References

[1] Edsger W. Dijkstra. Chapter I: Notes on structured programming, page 1–82. Academic Press Ltd., GBR, 1972.

[2] Ulf Norell. Towards a practical programming language based on dependent type theory. PhD thesis, Chalmers University of Technology, 2007.

[3] Jesper Cockx, Orestis Melkonian, Lucas Escot, James Chapman, and Ulf Norell. Reasonable agda is correct haskell: writing verified haskell using agda2hs. In Proceedings of the 15th ACM SIGPLAN International Haskell Symposium, Haskell 2022, page 108–122, New York, NY, USA, 2022. Association for Computing Machinery.

[4] Per Martin-Löf. Intuitionistic Type Theory. Bibliopolis, 1980.

[5] The Agda Development Team. The Agda Manual, 2026. Release 2.8.0.

[6] Morten Heine B. Sørensen and Paweł Urzyczyn. Lectures on the Curry-Howard Isomorphism, chapter 2. Elsevier, 2006.

[7] John Hughes. Quickcheck testing for fun and profit. volume 4354, pages 1–32, 01 2007.

[8] Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. ACM SIGPLAN Notices, 35(9):268–279, September 2000.

[9] Peter Dybjer, Qiao Haiyan, and Makoto Takeyama. Random generators for dependent types. In Zhiming Liu and Keijiro Araki, editors, Theoretical Aspects of Computing - ICTAC 2004, pages 341–355, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[10] Leonidas Lampropoulos, Zoe Paraskevopoulou, and Benjamin C. Pierce. Generating good generators for inductive relations. Proc. ACM Program. Lang., 2(POPL), December 2017.

[11] Zoe Paraskevopoulou, Cătălin HriŢcu, Maxime Dénès, Leonidas Lampropoulos, and Benjamin C. Pierce. Foundational property-based testing. In Christian Urban and Xingyuan Zhang, editors, Interactive Theorem Proving, pages 325–343, Cham, 2015. Springer International Publishing.

[12] Robert Atkey. The syntax and semantics of quantitative type theory. In LICS '18: 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, July 9–12, 2018, Oxford, United Kingdom, pages 56–65. Association for Computing Machinery, 2018.

[13] Conor McBride. I got plenty o' nuttin'. In A List of Successes That Can Change the World, 2016.