Translating Agda Postconditions to Haskell QuickCheck Properties

Author: **Aleksandra Kierska**
Supervisor: **Nathaniel Burke**
Responsible Professor: **Jesper Cockx**
TU Delft

1 Background

Agda is a dependently typed functional language and proof assistant. By the Curry–Howard correspondence, a type is a proposition and a program satisfying its type is its proof.

agda2hs [1] is a tool that translates verified Agda code into readable Haskell, so that verified components can be integrated into ordinary Haskell projects. Some Agda features — dependent types — cannot be expressed in Haskell. To handle this, such terms are marked with `@0` (erasure). Agda's type checker verifies that erased terms do not influence the computation of the program, so they can be safely dropped in the Haskell output.

QuickCheck is a Haskell library for property-based testing.

2 Motivation

Catching Unsound Postulates
Some Agda proofs rest on postulates — unproven assumptions about external Haskell. Extracting them as tests checks the assumptions against the real implementation.

POSTULATE
```
postulate
  sort : List a → List a
  sort-sorted :
    ∀ xs → Sorted (sort xs)
```
→ **AGDA PROOF**: Proof assumes the postulate holds...
→ **ERASED HASKELL**: Postulate never checked against real implementation!
→ **QUICKCHECK PROPERTY**
```
prop_sorted xs = isSorted
(sort xs)
```

Prototyping
Before investing hours in a proof, run a QuickCheck test first. If it fails, the property or the implementation is wrong.

Testing the trusted base
agda2hs assumes `Haskell.Prelude` matches the real Prelude. Postcondition tests provide independent evidence.

3 The Problem

Postconditions in Agda are encoded as indexed datatypes: types 'parameterized' by values. For example, `Sorted xs` is the proposition that `xs` is sorted:

Agda
```
data Sorted : List Nat → Set where
  Nil  : Sorted []
  One  : ∀ {x} → Sorted (x :: [])
  Cons : ∀ {x y xs} → IsTrue (x <= y)
           → Sorted (y :: xs) → Sorted (x :: y :: xs)
```

Haskell types cannot depend on values — there is no way to write Sorted [1, 2, 3] as a Haskell type. agda2hs drops these encodings in translation.

Thus postconditions in Agda, defined in the 3 forms below, all encode properties lost when translated to Haskell.

LEMMA
```
sortedLemma : ∀ xs → Sorted (sort xs)
→ relation has no Haskell form
```

SIGMA
```
sortedSigma :
  ∀ xs → ∃ (List Nat) (λ ys → Sorted ys)
→ sort :: [Nat] -> [Nat] (proof gone)
```

INDEXED DATATYPE
```
tailV : {a : Type} {@0 n : Nat}
          → Vec a (suc n) → Vec a n
→ tailV :: Vec a -> Vec a (index gone)
```

Haskell translation loses the encoded properties!

4 Research Question

*How can postconditions expressed in Agda — as **lemmas, sigma types, or indexed datatypes** — be translated to QuickCheck property tests automatically?*

5 From Lemmas to Tests

Given a lemma, how do we turn it into a runnable test? The proposition itself is just a type, but we need a function that returns true or false...

AGDA LEMMA
```
sortedLemma : ∀ xs → Sorted (sort xs)
```
→
QUICKCHECK PROPERTY
```
prop_sortedLemma xs = decSorted (sort xs)
```

Deciders
A decider checks whether a proposition holds, returning true or false along with a proof the answer is correct. After erasure, it becomes a plain Bool function, which is exactly what QuickCheck needs.

Agda
```
decSorted : {xs : List Nat} → Dec (Sorted xs)
decSorted {[]} = True ⟨ Nil ⟩
decSorted {x :: []} = True ⟨ One ⟩
decSorted {x :: y :: xs} = ...
```

Translation Pipeline
With deciders in hand, the translation is direct: (i) split the lemma into inputs and conclusion; (ii) use the lemma's inputs as the test's inputs; (iii) find the decider for the conclusion's type (iv) compile that decider to a Haskell function.

Now we can translate a lemma to a test — though each decider must still be written by hand.

6 Automating the Translation

decSorted looks a lot like Sorted — no coincidence: a decider is essentially the boolean version of its relation. **Repetitive!**

so we ask:

Can one general algorithm give us a function that checks whether a relation holds?

the idea:

In Agda, a relation holds exactly when we can construct its inhabitant — so the algorithm tries to: (i) find a constructor that could have produced these indices, (ii) check its preconditions.
Matching constructor and satisfied preconditions? ⇒ inhabitant found!

Let's see it in practice: can we construct `Sorted (1 :: 2 :: [])`? The algorithm tries each constructor:

```
nil  → Sorted []             ✗  [] ≠ (1 :: 2 :: []) → doesn't fit!
one  → Sorted (x :: [])      ✗  doesn't fit!
cons → Sorted (x :: y :: xs) ✓  fits! check preconditions:
         1 ≤ 2? ✓ Sorted (2 :: [])? ✓  ⟹  HOLDS!
```

Following this algorithm yields a checker — a decider (block 5) without the proof:

Haskell
```
checkSorted :: [Int] -> Bool
checkSorted arr = or
    [ case arr of
        []          -> True
        _           -> False
    , case arr of
        (x : [])    -> True
        _           -> False
    , case arr of
        (x : y : xs) -> x <= y && checkSorted (y : xs)
        _           -> False
    ]
```

Based on Paraskevopoulou et al. [2]

Now lemmas translate fully automatically — and so do sigma types: structurally they're lemmas (the proof is a separate component), so the same derivation handles them. Only parsing the form differs.

7 From Indexed Datatype to Test

With lemmas translation fully automated, we turn to indexed-datatype postconditions.

tailV is an example function with its postcondition encoded as an indexed datatype: the output Vec is one element shorter than the input.

Agda
```
data Vec (a : Type) : (@0 n : Nat) → Type where
  Nil  : Vec a 0
  Cons : {@0 n : Nat} → a → Vec a n → Vec a (suc n)

tailV : {a : Type} {@0 n : Nat}
          → Vec a (suc n) → Vec a n
```

Haskell
```
data Vec a = Nil
           | Cons a (Vec a)

tailV :: Vec a -> Vec a
tailV (Cons x xs) = xs
```

Applying the same test derivation as Block 5 and 6:

Haskell
```
prop_tailV n xs = checkVec n (tailV xs)
```

Input changed: block 6's checker had only indices — this one also receives the inhabitant, with its indices erased.

Thus the checker's job narrows: instead of trying every constructor to find an inhabitant, we pattern-match the one we have. The rest follows block 6: check that the indices fit and the preconditions hold.

Haskell
```
checkVec :: Vec a -> Nat -> Bool
checkVec Nil 0 = True
checkVec (Cons x xs) (Suc n) = checkVec xs n
checkVec _ _ = False
```

The checker for indexed datatype postconditions solves a sub-problem of block 6's: there, we tried every possible inhabitant and validated each; here, the inhabitant is given, so only the validation remains.

8 Related Work

Work	Target	Scope	Automatic?
Paraskevopoulou et al. '22 [2]	Rocq / QuickChick	Extrinsic Pre- and Post-conditions	Yes, automatic checkers, producers & proofs.
Losacio et al. '22 [4]	F* / QCheck	Boolean Postconditions	Yes.
Naucke '24 [3]	agda2hs	Extrinsic Preconditions	No, requires hand-written deciders.
This work	agda2hs / QuickCheck	Extrinsic & Intrinsic Postconditions	Yes, automatic checkers.

9 Conclusions

RESULTS
- **Implemented:** lemma-to-test pipeline in agda2hs (block 5)
- **Designed:** fully automated postcondition-to-test pipeline using checkers (extrinsic: adapting Paraskevopoulou et al. [2]; intrinsic: our extension); validated by worked examples
- **Motivations met** (block 2): tests for postulates derived automatically; prototyping stays in Agda, no parallel test suite to write or maintain

LIMITATIONS & FUTURE WORK

LIMITATION	FUTURE WORK
Incomplete implementation: no translation for sigma types or indexed datatypes, no automatic checker derivation	Implement translation for all three forms with automatic checker derivation
No formal correctness proof	Derive checkers as Agda functions, prove correctness via reflection, then compile to Haskell
Checkers can fail to reach a verdict on some inputs, which results in discarded tests.	Extend checker derivation with premise ordering strategies to reduce inconclusive runs

References. [1] Cockx et al. Reasonable Agda is Correct Haskell, Haskell Symp. 2022. [2] Paraskevopoulou et al. Computing Correctly with Inductive Relations, PLDI 2022. [3] Naucke. Compiling Dependent Type Preconditions to Runtime Checks with agda2hs, MSc thesis, TU Delft 2024. [4] Locascio et al. Verify, but test, ML Workshop 2022.

github.com/akierska/agda2hs-rp
github.com/akierska/agda2hs-postconditions-thesis-examples