Specified Agda is Testable Haskell

Translating Agda Postconditions to QuickCheck Property Tests in agda2hs

Aleksandra Kierska[1]

Supervisor(s): 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: Aleksandra Kierska
Final project course: CSE3000 Research Project
Thesis committee: Jesper Cockx, Nathaniel Burke, Annibale Panichella

Abstract

Formally verified code written in dependently typed languages can be integrated into larger, non-verified code bases through source-to-source translation. The agda2hs compiler enables such integration by translating verified Agda code to readable Haskell, erasing proofs and type indices in the process. The properties that a program is verified against are encoded as such proofs and type indices, and are therefore also erased. Yet being able to check these properties is important: in the agda2hs workflow, verified code may depend on unverified Haskell libraries whose properties are assumed without proof and never tested.

In this paper, we present an extension to agda2hs that extracts postconditions and translates them into Haskell QuickCheck property tests. Since postconditions in Agda are encoded at the type level, they are not executable and cannot be translated to Haskell directly. We address this by deriving semi-decision procedures that check whether a proposition holds for given inputs. We cover postconditions expressed as separate lemmas, sigma types, and indexed datatypes. We implement lemma-to-test translation as a working agda2hs extension; it requires the user to write the decision procedure by hand. We present the automatic derivation of such procedures as an algorithm design with worked examples.

1 Introduction

The `agda2hs` compiler [3] translates verified Agda [8] code to readable Haskell, erasing proofs and type indices in the process. Postconditions—properties that the outputs of functions must satisfy—are encoded as proofs and type indices, and are therefore also erased. The generated Haskell has no way to check these properties at runtime—yet doing so matters in three situations.

First, testing complements proving as a rapid prototyping tool: before investing hours in a formal proof, a QuickCheck test catches bugs in the property or the implementation early [9; 5].

Second, verified Agda modules can depend on external Haskell code via postulates—properties declared about the external code but never proven. If a postulate does not hold, the proof's guarantees silently fail. Extracting postconditions as tests exercises these assumptions against the real implementation.

Third, `agda2hs` itself relies on an informal correctness argument and a hand-written model of the Haskell Prelude [3]; postcondition tests provide independent evidence that the trusted base holds on the inputs covered.

In this paper, we present an extension to `agda2hs` that extracts postconditions and translates them into Haskell QuickCheck property tests. Since postconditions in Agda are expressed as types, they cannot be directly translated into the Boolean functions that QuickCheck requires. We address this by deriving checkers—semi-decision procedures that check whether a proposition holds for a given input. We cover postconditions expressed as separate lemmas, sigma types, and indexed datatypes. Concretely, our contributions are:

- We implement a translation pipeline in `agda2hs` that turns lemmas into QuickCheck property tests, given a user-supplied decider (Section 3.2).
- We adapt the checker derivation algorithm of Paraskevopoulou et al. [9] to Agda and extend it to intrinsic postconditions encoded as indexed datatypes (Sections 3.3 and 3.4).
- We give an informal correctness argument for the derived checkers, building on Paraskevopoulou et al. (Section 4.1).
- We identify which relations cause the derived checkers to fail to reach a verdict, resulting in a discarded test, and discuss remedies (Section 4.3).

Section 2 provides an overview of relevant background: Agda, `agda2hs`, and QuickCheck. Section 3 develops the translation from hand-written deciders to automatic checker derivation, covering extrinsic and intrinsic postconditions. Section 4 evaluates correctness, analyses when derived checkers fail to reach a verdict, and states the limitations. Section 5 discusses related work, and Section 7 concludes.

The prototype and the code examples from this paper are publicly available.[1,2]

2 Background

2.1 Agda

Agda [8] is a total, dependently typed functional programming language with syntax resembling Haskell.

Dependent Types. In Agda, types can depend on values. Consider `Vec` (Figure 1), a list whose type also tracks its length. In general, arguments before the colon in a datatype declaration are *parameters*, shared by every constructor; arguments after it are *indices*, which each constructor determines individually. Here the element type `a` is a parameter, while the length `n` is an index.[3] Crucially, `Vec a 0` and `Vec a 3` are different types—a vector's type *depends* on its length.

```
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)
```

Figure 1: A length-indexed vector. The parameter `a` determines the element type; the index `n` encodes the length.

[1] https://github.com/akierska/agda2hs-rp, implementation on the `postconditions/compile-property-v2` branch.

[2] https://github.com/akierska/agda2hs-postconditions-thesis-examples

[3] Curly braces mark implicit arguments, which Agda infers automatically; the `@0` annotations are explained in Section 2.3.

Propositions as Types. Thanks to totality and dependent types, Agda can also be used as a theorem prover. The Curry-Howard correspondence [4] states that proofs are to propositions what programs are to types: to prove a proposition in Agda, we express it as a type and construct a program of that type. For example, the relation `Leq` (Figure 2) encodes less-than-or-equal relation on naturals.

```
data Leq : Nat → Nat → Type where
  LeqZero : {n : Nat}   → Leq 0 n
  LeqSuc  : {m n : Nat} → Leq m n
                        → Leq (suc m) (suc n)
```

Figure 2: The less-than-or-equal relation on naturals.

The type `Leq 2 4` expresses the proposition that $2 \leq 4$. A value of this type (also called an *inhabitant*) is a proof of that claim: `LeqSuc (LeqSuc LeqZero)`, built by chaining constructors. Conversely, no chain of constructors produces `Leq 3 1`; the type has no inhabitant, so the proposition is unprovable. We say a relation *holds* exactly when it is inhabited.

2.2 Extrinsic and Intrinsic Verification

A property of a program can be verified in two ways. In *extrinsic* verification, the proof is completely separate from the program: one could write a `min` function, then a separate lemma stating that its result is no greater than either input. In *intrinsic* verification, the property is encoded directly in the types of data and functions: `Vec` encodes its length in its type, so `Vec a n` can only be constructed with exactly `n` elements.

2.3 agda2hs and Erasure

AGDA2HS [3] is a tool that translates a subset of Agda into readable Haskell, allowing verified code to be integrated into a larger, unverified code base. Only definitions marked with a `COMPILE AGDA2HS` pragma are translated; the rest are typechecked and then discarded.

Erasure. Verification-only terms may also appear inside translated definitions. To know what it can omit, agda2hs leverages Agda's *erasure* annotations [6; 1]. An argument marked `@0` may be used during typechecking but never in computation; Agda's type checker enforces this restriction. Recall `Vec` (Figure 1): indices do not exist in Haskell, so its index `n` is marked `@0`, and agda2hs drops it, leaving an ordinary list: `data Vec a = Nil | Cons a (Vec a)`.

2.4 QuickCheck

Property-Based Testing. QuickCheck [2] is a Haskell library for property-based testing. Instead of writing unit tests by hand, we state a *property*—how the program should behave—and QuickCheck checks that it holds on many randomly generated inputs. For example, `prop_rev xs = reverse (reverse xs) == xs` states that reversing a list twice yields the original. When a property fails, QuickCheck reports a falsifying input as a counterexample.

Generators. QuickCheck draws its random inputs from *generators*. By default, the `Arbitrary` type class provides a generator for the standard types. In this paper we instead supply our own through the `forAll` combinator, which runs a property on inputs from a given generator:

```
forAll <generator> (\input -> <property>)
```

Discards. An input that cannot be tested is *discarded*: QuickCheck draws a fresh one and retries, and a discard counts as neither a pass nor a failure. Too many discards make it give up before reaching its target number of passing tests.

3 From Postconditions to Property Tests

The translation is intricate, so we build it up in stages: Section 3.1 introduces the running example; Section 3.2 explains lemma translation using hand-written deciders for extrinsic relations; Section 3.3 improves the method by automatically deriving checkers—deciders without the proof term; Section 3.4 extends the derivation to indexed datatypes, which encode postconditions intrinsically; and Section 3.5 reuses the preceding algorithms for sigma types.

3.1 Running Example: STLC

```
data Ty : Type where
    N   : Ty
    Arr : Ty → Ty → Ty

data Term : Type where
    Con : Nat → Term
    Add : Term → Term → Term
    Var : Nat → Term
    Lam : Ty → Term → Term
    App : Term → Term → Term

{-# COMPILE AGDA2HS Ty deriving Eq #-}
{-# COMPILE AGDA2HS Term deriving Eq #-}

data Lookup : List Ty → Nat → Ty → Type where
    ...

data Typing : List Ty → Term → Ty → Type where
    TCon : ∀ {Γ n}
         → Typing Γ (Con n) N
    TAdd : ∀ {Γ e₁ e₂}
         → Typing Γ e₁ N
         → Typing Γ e₂ N
         → Typing Γ (Add e₁ e₂) N
    TVar : ∀ {Γ x t}
         → Lookup Γ x t
         → Typing Γ (Var x) t
    TLam : ∀ {Γ e t₁ t₂}
         → Typing (t₁ :: Γ) e t₂
         → Typing Γ (Lam t₁ e) (Arr t₁ t₂)
    TApp : ∀ {Γ e₁ e₂ t₁ t₂}
         → Typing Γ e₁ (Arr t₁ t₂)
         → Typing Γ e₂ t₁
         → Typing Γ (App e₁ e₂) t₂
```

Figure 3: STLC definition in Agda. Lookup definition omitted for brevity.

Throughout the solution section, we illustrate our algorithms using the simply typed lambda calculus (STLC), a standard example in programming language theory also used by Paraskevopoulou et al. [9], as it exercises most cases of our translation. We define types, terms, and a typing relation in Agda below and briefly explain each component.

Types. `Ty` represents the types in our language: `N` for natural numbers and `Arr` t_1 t_2 for functions that take an argument of type t_1 and return a result of type t_2.

Terms. `Term` represents expressions in our language. `Con`, `Add`, and `Var` represent constants, addition, and variables respectively. `Lam t e` is a lambda expression—a function with a parameter of type `t` and body `e`—and `App` e_1 e_2 applies function e_1 to argument e_2. Variables use de Bruijn indices rather than names. For example, `Lam N (Add (Var 0) (Con 1))` represents the function $\lambda x.\ x + 1$, where `Var 0` refers to the lambda's parameter.

Typing. Not every expression is well-typed—for instance, `App (Con 3) (Con 5)` attempts to apply a number as a function. The relation `Typing` Γ `e t` defines the typing rules for our language, characterising when expression `e` has type `t` in environment Γ (a list of types for the variables currently in scope). The relation `Lookup` Γ `x t`, used in `TVar`, holds when variable `x` has type `t` in Γ.

Haskell Translation. agda2hs translates[4] `Ty` and `Term` to equivalent Haskell datatypes. `Typing` and `Lookup`, on the other hand, are extrinsic relations used purely as propositions, so agda2hs does not generate Haskell code for them. In the following sections, we show how these predicates can be translated into executable checks suitable for QuickCheck testing.

3.2 Testing Lemmas via Hand-Written Deciders

How do we turn a postcondition into a QuickCheck test? We start with the simplest case: a standalone lemma. Consider the lemma in Figure 4, which states that some transformation preserves well-typedness.

```
preservesTyping : (Γ : List Ty)
                → (e : Term)
                → (t : Ty)
                → Typing Γ e t
                → Typing Γ (transform e) t
```

Figure 4: A preservation lemma: if `e` is well-typed, then `transform e` is too.

To test this property, we need a function that checks whether `Typing` holds for given inputs. As discussed in Section 3.1, `Typing` is an extrinsic relation with no Haskell counterpart, so we must construct such a function ourselves.

In Agda, the standard way to do so is through a *decider*: the agda2hs library defines `Dec P` as a pair of a `Bool` and a proof that the boolean correctly reflects whether `P` holds. Naucke [7] uses this mechanism to compile precondition checks in agda2hs; we adopt the same approach for postconditions. Figure 5 shows selected cases of a decider for `Typing`.[5]

```
decTyping : (Γ : List Ty) → (e : Term) → (t : Ty)
          → Dec (Typing Γ e t)

decTyping Γ (Con n)      N = True ⟨ TCon ⟩
decTyping Γ (Add e₁ e₂)  N =
  case decTyping Γ e₁ N , decTyping Γ e₂ N of λ
  ↪  where
    (True ⟨ p₁ ⟩ , True ⟨ p₂ ⟩) →
      True ⟨ TAdd p₁ p₂ ⟩
    (False ⟨ ¬p ⟩ , _) →
      False ⟨ ... ⟩
    (_ , False ⟨ ¬p ⟩) →
      False ⟨ ... ⟩
... -- remaining cases
```

Figure 5: Parts of a decider for the `Typing` relation.

The decider follows the structure of the `Typing` relation: each constructor of `Typing` becomes a case of the decider. For instance, the `TCon` case returns `True` with a proof that `Con n` has type `N`, while the `TAdd` case recursively verifies that both sub-expressions have type `N`. After agda2hs translation, the proofs are erased and only the boolean remains—giving us exactly the `decTyping` function we need. The resulting QuickCheck property is shown in Figure 6. The existence of a generator `genTyping` that produces only well-typed inputs is assumed.[6]

```
genTyping :: Gen ([Ty], Term, Ty)

prop_preservesTyping :: Property
prop_preservesTyping =
    forAll genTyping (\(ctx, e, t) ->
          decTyping ctx (transform e) t)
```

Figure 6: QuickCheck property corresponding to the lemma in Figure 4.

The translation pipeline. With a decider in hand, translating a lemma is straightforward: the compiler splits the type signature into inputs and conclusion, searches for a decider for the conclusion's predicate, and emits a QuickCheck property that applies the decider to the inputs. If no matching decider is found, the test is skipped. We implement this pipeline as an extension to agda2hs.[7]

Limitations. While this approach produces correct tests—the decider's proof guarantees that its Boolean reflects the relation—it requires the user to write a decider for every relation mentioned in a postcondition. Doing this by hand conflicts with the goal of rapid prototyping, so the following section discusses deriving these functions automatically.

[4] From now on we omit `COMPILE AGDA2HS` pragmas from figures for brevity; assume any translated Agda definition carries one.

[5] The proof terms in `TAdd`'s `False` cases are omitted for brevity.

[6] Generator derivation is out of scope for this thesis; a parallel bachelor work addresses it.

[7] https://github.com/akierska/agda2hs-rp/tree/postconditions/compile-property-v2

3.3 Automatic Checker Derivation for Extrinsic Relations

Section 3.2 tested a lemma with a hand-written decider for the relation in its conclusion. Writing one for every relation is repetitive. Paraskevopoulou et al. [9] give an algorithm that derives such functions automatically; although developed for Rocq, it carries over to Agda, which shares the same notion of inductive relation.

Checkers. We derive *checkers*: functions that return, as a plain `Bool`, whether the relation holds for the given inputs. A checker differs from the decider of Section 3.2 in what it leaves out—the proof that the Boolean reflects the relation. We omit it on purpose: deriving such proofs automatically is a separate problem of its own, which we place out of scope. Instead, we revisit correctness of the algorithm in Section 4.1.

Derivation. A checker returns true iff the relation holds for its inputs. In Agda, a relation holds exactly when it is inhabited, so deciding it means constructing an inhabitant—or showing that none exists.

For `Typing`, given Γ, `e`, and `t`, our task is to construct an inhabitant of `Typing Γ e t`. Consider a concrete case: `e` is `Add e1 e2` and `t` is `N`. Only `TAdd` matches, since its conclusion is `Typing Γ (Add e1 e2) N`. For a matching constructor, we check its preconditions—the inputs it demands; for `TAdd`, it's that `e1` and `e2` both have type `N`, which we check recursively. If they hold, we have built an inhabitant of `Typing Γ (Add e1 e2) N`.

Figure 7 shows the checker this procedure yields for `Typing`.

```
checkTyping :: [Ty] -> Term -> Ty -> Bool
checkTyping ctx e t = or
    [ case (e, t) of
        (Con _, N) -> True
        _          -> False
    , case (e, t) of
        (Add e1 e2, N) ->
            checkTyping ctx e1 N
            && checkTyping ctx e2 N
        _ -> False
    , case (e, t) of
        (Var x, _) ->
            checkLookup ctx x t
        _ -> False
    , case (e, t) of
        (Lam t1 e', Arr t1' t2) ->
            t1 == t1'
            && checkTyping (t1:ctx) e' t2
        _ -> False
    , ...
    ]
```

Figure 7: Derived checker for `Typing`, omitting TApp case.

The checker tries every constructor, combining the results with `or`—it succeeds if any branch does. Each branch handles its constructor in two steps: match the conclusion against the inputs, and check the preconditions. When a precondition involves a different relation—as `TVar`'s involves `Lookup`—it is checked by that relation's checker, derived the same way.

Handling Existentials. `TApp` is the one constructor whose premises need a value the conclusion does not provide. Recall the rule:

```
TApp : ∀ {Γ e₁ e₂ t₁ t₂}
     → Typing Γ e₁ (Arr t₁ t₂)
     → Typing Γ e₂ t₁
     → Typing Γ (App e₁ e₂) t₂
```

If we pattern match on the conclusion, we have access to `e1`, `e2`, and `t2`:

```
case (e, t) of
       (App e1 e2, t2) -> ...
```

The premises also need `t1`, which does not appear in the conclusion. This means the checker cannot check the preconditions unless it finds `t1`.

To find `t1` we use a *producer*: where a checker decides whether a relation holds for given inputs, a producer enumerates the inputs that make it hold. Here `enumTyping ctx e2` lists the types `t1` satisfying `Typing ctx e2 t1`, and the branch checks the remaining premise for each:

```
enumTyping :: [Ty] -> Term -> [Ty]

...
, case (e, t) of
    (App e1 e2, t2) ->
        any (\t1 -> checkTyping ctx e1 (Arr t1 t2))
            (enumTyping ctx e2)
    _ -> False
```

A producer works over the relation's structure, just as the checker does, so it is more targeted than enumerating all values of `t1`. Deriving producers is a problem of its own; we take them as given and refer to a colleague's parallel thesis for the construction (see also Paraskevopoulou et al. [9]).

Handling existentials has a cost: a producer may list infinitely many candidates, so the check need not terminate. For `Typing` this is harmless—each term has at most one type in a given context—but the derivation must handle the general case.

Fuel. Following Paraskevopoulou et al., we bound the search of our checkers and producers with *fuel*: a limit on their recursion depth. A checker that may run out of fuel can no longer always answer, so its output type changes from `Bool` to `Maybe Bool`: `Nothing` means the fuel ran out before definitive answer could be reached.

Each checker takes two fuel budgets. The first it spends: every recursive call on the relation itself receives one unit less. The second remembers the starting amount and is passed, in full, to the checkers of other relations appearing in premises—like `Lookup` for checkTyping. A test invokes the checker with both budgets equal. Producers follow the same.

Adding fuel expands the checker code (Figure 8). When no fuel is left, only the constructors without recursive premises are tried; if none succeeds, the checker returns `Nothing`, since the recursive cases it had no fuel for might still have succeeded.

```
checkTyping :: Fuel -> Fuel -> [Ty]
            -> Term -> Ty -> Maybe Bool

checkTyping 0 topFuel ctx e t = backtracking
  [ case (e, t) of
      (Con _, N) -> Just True
      _          -> Just False
  , Nothing ]
checkTyping fuel topFuel ctx e t = backtracking
  [ ... -- Con case
  , case (e, t) of
      (Add e1 e2, N) ->
        checkTyping (fuel-1) topFuel ctx e1 N
        `andM`
        checkTyping (fuel-1) topFuel ctx e2 N
      _ -> Just False
  , case (e, t) of
      (Var x, _) ->
        checkLookup topFuel topFuel ctx x t
      _ -> Just False
  , ... -- remaining cases
  ]
```

Figure 8: The fueled checker for `Typing`. `backtracking` returns `Just True` if any branch succeeds, `Just False` if all fail, and `Nothing` otherwise. `andM` short-circuits on the first `Just False` or `Nothing`.

Why use fuel? Paraskevopoulou et al. introduce fuel because Rocq's termination checker rejects unbounded recursion; Haskell has no such restriction, but fuel remains useful. Without it, a non-terminating check blocks the entire test run, preventing other inputs from being tested. With fuel, the checker returns `Nothing`, QuickCheck discards that input, and testing continues. A fuel budget gives the programmer control over the test suite—a higher budget produces fewer discards at the cost of longer runs. The producers we assume from Paraskevopoulou et al. also require fuel for their correctness guarantees (Section 4.1), so checkers use the same interface.

Omitted cases. We omit the treatment of nonlinear variables and functions as indices; for these and the full algorithm we refer to Paraskevopoulou et al. [9].

3.4 Testing Indexed Datatype Postconditions

Section 3.3 derived checkers for extrinsic relations such as `Typing`—datatypes that serve purely as propositions and have no direct Haskell counterpart. We now turn to intrinsic postconditions: datatypes like `Vec` (recall Figure 1 and its Haskell translation Figure 9), which encode invariants but also carry data that agda2hs translates to Haskell.

Both extrinsic and intrinsic postconditions take the same form in Agda: an indexed datatype. The shared form suggests the same checker derivation should apply—and it does, with two adjustments.

The figures in this section omit fuel, keeping the structural change in focus rather than buried under fuel machinery. Fuel remains part of the design; adding it back follows Section 3.3.

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

Figure 9: Haskell translation of `Vec`

Supplying the erased indices. The indices are erased from Haskell, so the checker has no expected values to check against. Consider `concat`, which concatenates two vectors:

```
concatV : {a : Type} {@0 n m : Nat}
        → Vec a n → Vec a m → Vec a (n + m)

concatV :: Vec a -> Vec a -> Vec a
```

If we directly mirror this signature as a test, the expected index is missing:

```
prop_concatV xs ys = checkVec ??? (concatV xs ys)
```

We resolve this by generating the index explicitly. Instead of generating only the input vectors, the test first generates `n` and `m`, then constructs the input vectors of respective lengths, finally the value of `n + m` is passed to the checker as an *expected index*:

```
prop_concatV n m =
  forAllJust (genVec n) $ \xs ->
  forAllJust (genVec m) $ \ys ->
  checkVec (n + m) (concatV xs ys)
```

Figure 10: Generated QuickCheck test for `concatV`. `forAllJust` runs the property on generated values, discarding any `Nothing`. `genVec` is a generator for Vec, we assume exists.

In general, every variable in a function's output type is bound by its inputs—Agda does not type-check otherwise—so the expected indices are always available from input generation.[8]

Validation as a subproblem of search. The second adjustment concerns the checker's role. In Section 3.3, the checker determines whether `Typing` Γ `e t` has an inhabitant—it searches for one by trying constructors. Here, we already have an inhabitant: the function's return value. The checker's role changes: given the inhabitant and the expected indices, verify that its construction does not break the encoded invariants.

This is a subproblem of what the extrinsic checker solves: the extrinsic checker searches over constructors and validates each candidate; the intrinsic checker pattern-matches on the constructor of given inhabitant directly and performs only the validation.

Applying the derivation to STLC. To demonstrate, we encode our running example intrinsically. `Expr` (Figure 11) merges `Term` and `Typing` from Figure 3 into a single indexed datatype.

[8] This assumes the index types exist in Haskell. If an index is an extrinsic relation such as `Lookup`, it has no Haskell counterpart and cannot be generated directly. We place such cases out of scope.

```
data Expr (@0 Γ : List Ty) : @0 Ty → Type where
    ECon : Nat → Expr Γ N
    EAdd : Expr Γ N → Expr Γ N → Expr Γ N
    EVar : ∀ {@0 t}
         → (x : Nat)
         → @0 Lookup Γ x t
         → Expr Γ t
    ELam : ∀ {@0 t₂}
         → (t₁ : Ty)
         → Expr (t₁ :: Γ) t₂
         → Expr Γ (Arr t₁ t₂)
    EApp : ∀ {@0 t₁ t₂}
         → Expr Γ (Arr t₁ t₂)
         → Expr Γ t₁
         → Expr Γ t₂
```

Figure 11: Agda definition of Expr.

agda2hs translates `Expr` to a Haskell datatype with the same structure as `Term`. Figure 12 shows the derived checker. Each branch mirrors the corresponding branch of `checkTyping` (Figure 7) from Section 3.3—with one difference: the expression is pattern-matched directly rather than searched for. The `EApp` case uses a producer to enumerate the existential type `t1`, exactly as `TApp` does in Section 3.3.

```
enumExprTy :: [Ty] -> Expr -> [Ty]

checkExpr :: [Ty] -> Ty -> Expr -> Bool
checkExpr _ ty (ECon _) =
    ty == N
checkExpr ctx ty (EAdd l r) =
    ty == N
    && checkExpr ctx N l
    && checkExpr ctx N r
checkExpr ctx ty (EVar x) =
    checkLookup ctx x ty
checkExpr ctx ty (ELam t1 body) =
   case ty of
    (Arr t1' t2') ->
        t1' == t1
        && checkExpr (t1 : ctx) t2' body
    _ -> False
checkExpr ctx ty (EApp e1 e2) =
    any (\t1 -> checkExpr ctx (Arr t1 ty) e1
             && checkExpr ctx t1 e2)
        (enumExprTy ctx e2)
```

Figure 12: Derived checker for `Expr`.

3.5 Testing Sigma Type Postconditions

The transform lemma from Section 3.2 (Figure 4) can be re-expressed as a sigma type that bundles typing preservation into the return type:

```
transform : (@0 Γ : List Ty) → (e : Term)
    → (@0 t : Ty) → @0 Typing Γ e t
    → ∃ Term (λ e' → Typing Γ e' t)
```

When translated by agda2hs, `transform` becomes a regular function returning a `Term`, and the postcondition translates to a test identical to Figure 6. We use the checker derivation of Section 3.3.

4 Evaluation

We evaluate the derived checkers for correctness (Section 4.1), examine how well the approach fulfils the motivations from the introduction (Section 4.2), analyse when checkers fail to reach a verdict and discard tests (Section 4.3), and state the limitations (Section 4.4).

4.1 Correctness

A fueled checker is a semi-decision procedure—its correctness is characterised by three properties [9]: soundness, completeness, and monotonicity. We argue informally that the checkers of Sections 3.3 and 3.4 satisfy them (Section 4.4 explains why a formal proof is out of reach).

Soundness. The checker returning `Just True` for some fuel implies the relation P holds.

$$\forall\, s.\ \texttt{check}\ s\ e_1\ \ldots\ e_m = \texttt{Just True} \implies P\ e_1\ \ldots\ e_m$$

Completeness. If the relation holds, some fuel yields `Just True`.

$$P\ e_1\ \ldots\ e_m \implies \exists\, s.\ \texttt{check}\ s\ e_1\ \ldots\ e_m = \texttt{Just True}$$

Monotonicity. More fuel never changes a definitive answer.

$$\forall\, s_1 \le s_2,\ b.\ \texttt{check}\ s_1\ e_1\ \ldots\ e_m = \texttt{Just}\ b \implies \texttt{check}\ s_2\ e_1\ \ldots\ e_m = \texttt{Just}\ b$$

Properties of negation. Soundness and completeness as stated concern `Just True`—a natural question is whether they extend to `Just False`. Monotonicity and completeness together imply soundness of negation. Completeness for negation does not hold in general—inductive relations can encode infinite computations, leaving the checker unable to definitively reject an input [9].

Extrinsic correctness argument. Paraskevopoulou et al. establish the three properties at two levels. For each derived checker, an Ltac2 [10] script constructs a mechanised proof in Rocq alongside the checker. Separately, they sketch why the same scheme succeeds for every checker their algorithm produces—a metatheorem about the algorithm itself, argued informally. Our extrinsic derivation (Section 3.3) follows the same algorithm, so the same informal argument applies to our emitted Haskell, under two assumptions.

First, the producers the derived checkers call are correct: every value emitted satisfies the predicate, and every value satisfying the predicate is emitted at some fuel [9]. Producer derivation is outside the scope of this thesis; we assume our algorithm uses producers described and proved correct by Paraskevopoulou et al.

Second, the checker is sound only if == decides propositional equality: `a == b` returns `True` exactly when $a \equiv b$. The algorithm uses == in a few algorithm cases—for example, `t1 == t1'` in the `TLam` branch (Figure 7), which arises from rewriting the nonlinear `t1` in `TLam`'s conclusion. Implementation-wise, one could guarantee this by requiring an `IsLawfulEq` instance[9] for each type compared with ==.

[9] Defined at https://github.com/agda/agda2hs/blob/acb521e2b66cbccf3694c264929ae6298fb25637/lib/base/Haskell/Law/Eq/Def.agda

Intrinsic correctness argument. As noted in Section 3.4, validating an intrinsic relation is a subproblem of the extrinsic case. The intrinsic algorithm is a special case of the extrinsic one, in which the search always matches exactly one branch. The extrinsic argument therefore carries over, under the same two assumptions.

4.2 Motivation Fulfillment

The introduction listed three situations in which postcondition tests are useful. We examine each in turn.

Quick prototyping. Software development is iterative; verified development could be too, but proofs take time. An attempt to prove a false specification or implementation costs hours. Moreover, later revisions can invalidate previously completed proofs. Inserting tests before proofs reduces both costs: the user specifies, implements, tests, iterates, and only then attempts a proof. We evaluate our solution by how well it supports this prototyping cycle, taking hand-written tests as the natural baseline.

In principle, hand-written tests achieve the same goal: a failing test surfaces a bug before the user spends proof effort. In practice, hand-writing doubles the work: every postcondition needs a Haskell test, and every relation needs a Haskell checker. It also breaks DRY: every iteration on the specification requires a matching change in the Haskell tests, slowing prototyping.

With our test derivation, the user prototypes entirely in Agda, adding no work to the workflow—no Haskell to write or maintain by hand. The contribution is thus one of usability rather than capability: hand-written tests achieve the same goal at the cost of maintaining a parallel test suite.

Catching Unsound Postulates. Previously, properties assumed via postulates were never tested against the real implementation. Our derivation allows testing them without additional work; as with prototyping, the contribution is one of usability rather than capability.

Testing the trusted base. This benefit is hard to measure: postcondition tests may catch a bug in agda2hs someday, but until then they only provide additional evidence that translations behave as expected—limited to the properties the postconditions actually test.

4.3 Discarded Tests

Section 3.3 introduced fuel: a checker that exhausts its budget returns `Nothing` and the test run gets discarded. A discard is wasted work—QuickCheck replaces each one until it reaches its quota of successes—so a checker that often causes discards makes its tests slow to run. In this section we examine why discards occur, which relations cause discards, and what can be done about them.

The bottleneck: enumeration. When a checker enumerates, it re-checks the remaining premises for every candidate a producer supplies. Some relations have producers that emit large, or even infinite, streams of candidates. Since rechecking is repeated for every candidate, the fuel can run out long before a valid one is reached.

To illustrate, consider `Between`, built from the less-than-or-equal relation `Leq` (recall Figure 2). `Between` is an example of a relation whose checker can produce an infinite stream of candidates: when `Between` does not hold, the checker always exhausts its fuel.

```
data Between (k : Nat) : Nat → Type where
  mk : {m j : Nat} → Leq k m → Leq m j
                               → Between k j
```

Figure 13: `Between k j` holds when some `m` satisfies $k \leq m \leq j$, where `Leq` (Figure 2) is the less-than-or-equal relation.

Since `m` does not appear in the conclusion of `mk`, the checker has to find it: the producer for `Leq k m` emits candidates `m = k`, `k + 1`, `k + 2`, ...—and for each one the checker tests whether `Leq m j` also holds.

Consider `checkBetween 1 5`. The checker tries `m = 1` and checks `Leq 1 5`—it holds, so it returns `Just True`. Now consider `checkBetween 5 1`. The producer for `Leq 5 m` emits `m = 5, 6, 7, ...`; none satisfy `Leq m 1`, so the checker keeps going until it exhausts its fuel and returns `Nothing`.

The positive case resolves immediately; the negative case exhausts any budget. If a function under test has `Between` as its postcondition and produces a buggy output that violates it, the checker does not report a failure—it discards the test.

When do discards happen? Discards stem from enumeration. The checker must enumerate whenever a constructor's premise mentions a variable absent from its conclusion—there is no other way to find it. Enumeration is necessary for discards, but not sufficient: some enumerations are cheap. Recall `t1` in `TApp`: enumeration is necessary to find `t1`, but each expression has exactly one type in a given context, so the search always yields a single candidate—the checker ends up nearly as cheap as with no enumeration at all.

Remedies. Increasing the fuel budget helps when the number of candidates is finite, but for a producer like `Leq k m`, whose stream has no end, no budget is ever enough.

A second option is reordering premises. If the checker produces `m` from `Leq m j` instead of `Leq k m`, the stream is finite: `m = j, j-1, ..., 0`. A user could reorder the premises manually, but expecting such knowledge is unreasonable for general use. Extending the derivation to automatically reorder premises for fewer discards is a direction for future work.

For intrinsic postconditions there is a third option: keeping the enumerated value at runtime. Currently `App` compiles to

```
data Expr = ... | EApp Expr Expr | ...
```

because `t1` is erased. Removing the `@0` annotation produces

```
data Expr = ... | EApp Ty Expr Expr | ...
```

and the checker reads `t1` directly from the constructor—no enumeration.[10]

[10] In this specific case the enumeration is already cheap; the example illustrates the general principle.

This eliminates enumeration but makes the compiled Haskell less idiomatic, and storing the type allows construction of ill-typed terms on the Haskell side.

4.4 Limitations

This work has a central limitation: we cannot claim our checker derivation correct with confidence. Confidence would come from a running implementation, to exercise the algorithm by deriving checkers for many relations and testing each on generated inputs, or from a correctness proof. We have neither. Instead, the derivation rests on the hand-derived examples in this paper and the informal argument of Section 4.1.

No implementation. The prototype implements only the lemma pipeline of Section 3.2, which turns a lemma into a test through a user-written decider. We do not implement automatic checker derivation—neither for extrinsic relations (Section 3.3) nor for the intrinsic.

No formal proof. A formal proof of the algorithm as a whole is out of reach: it would require formalising the dependent types it operates on, a separate undertaking that even Paraskevopoulou et al. [9] do not attempt. Instead, they prove each derived checker individually, generating each proof by metaprogramming. Such per-checker proofs do not generalise to the whole algorithm, but they would confirm that the checkers a user actually runs are correct.

Path forward. Future work could address both limitations. Implementing the derivation in agda2hs, emitting the checkers as Haskell, would let us test it in practice even without a formal proof. That implementation could go one step further: derive each checker as an Agda function first, then compile it to Haskell with agda2hs. The advantage is provability—a checker in Agda is a term we can prove correct. We could then investigate whether generating proofs automatically with a reflection metaprogram is feasible in this case, much as Paraskevopoulou et al. do with Ltac2.

5 Related Work

Deriving checkers for inductive relations in Rocq. Paraskevopoulou et al. [9] present an algorithm that automatically derives checkers, producers, and correctness proofs for inductive relations. Their work extends QuickChick, a property-based testing plugin for Rocq. As part of our solution, we adapt their checker derivation algorithm for extrinsic relations (Section 3.3), and extend it to the intrinsic case (Section 3.4). The checkers we describe are emitted directly as Haskell, so they cannot be proved correct as their Rocq counterparts are—a trade-off we discuss in Section 4.4.

Translating F* specifications to QCheck tests. Locascio et al. [5] present a tool for deriving QCheck tests from F* specifications. F* is a proof-oriented language whose programs compile to OCaml; QCheck is OCaml's QuickCheck counterpart. The main difference between our work and theirs is scope: their tool targets only F* programs with Boolean pre- and postconditions, whereas we also handle propositions encoded as types. Since their postconditions are Boolean functions, they translate to OCaml directly; accordingly, their work focuses on the translation step: how to use metaprogramming to read pre- and postconditions out of refinements and `requires`/`ensures` clauses and assemble them into a test. Our core problem is instead how to turn a proposition encoded as a type into an executable check. Like us, they only generate the tests themselves and leave generator construction out of scope.

Compiling preconditions to runtime checks in agda2hs. Naucke [7] presents an extension to agda2hs that compiles Agda preconditions into runtime checks in the generated Haskell. Their work is the closest to ours: it shares our tool and addresses the same challenge—translating a property expressed in Agda that has no direct counterpart in Haskell. Naucke compiles each function with a precondition into one that checks the precondition, runs the original function if it holds, and raises an error otherwise, whereas we produce a test from the postcondition. Naucke relies on hand-written deciders, as our lemma pipeline (Section 3.2) also does, but we additionally present an algorithm that derives them automatically (Section 3.3). Within its scope, their approach handles only extrinsic preconditions, whereas ours covers both extrinsic and intrinsic postconditions (Section 3.4).

6 Responsible Research

6.1 Reproducibility

Implementation. The implementation of the translation pipeline described in Section 3.2 is publicly available, under the MIT License, on the `postconditions/compile-property-v2` branch of my agda2hs fork.[11] Readers can set up the repository by following the "Manual installation" section of `docs/source/introduction.md`. The `agda2hs.cabal` file specifies all dependencies and version ranges, except QuickCheck (version 2.18.0.0), which we used to run the generated tests.

Examples. The code examples from this paper are collected in a separate MIT-licensed repository.[12] Readers can verify the Agda examples by type-checking them and run the resulting QuickCheck properties.

Limitations. We derive the checkers presented in this paper by hand, following the algorithm and its extension described in Sections 3.3 and 3.4. Section 3.3 omits some algorithmic details that we explicitly defer to the QuickChick paper. Reproducing the examples therefore requires consulting that paper.

Another limitation is the scope of the implementation. The full translation pipeline is not implemented (as elaborated in Section 4.4). In particular, readers can only run the lemma decider translation (Section 3.2). For indexed datatypes and sigma types, readers can only run the hand-derived examples in the second repository.

[11] https://github.com/akierska/agda2hs-rp

[12] https://github.com/akierska/agda2hs-postconditions-thesis-examples

6.2 Integrity of Results.

The correctness argument in Section 4.1 is informal, but builds on the mechanised proofs in prior work. The hand-derived checkers in Section 3 demonstrate the algorithm on examples, but examples alone are not sufficient evidence of correctness. We state both limits explicitly in the paper, so readers can judge how much confidence the results warrant.

6.3 Ethical considerations.

This research is theoretical and produces a compiler extension. No human data is collected or processed, and the work has no foreseeable malicious applications.

6.4 Use of generative AI.

Generative AI was used during both writing and development of this work. Example prompts can be found in the Appendix A.

Writing. The research content, structure, and conclusions of this paper are my own, and I take full responsibility for the submitted text. AI assisted by generating alternative, often conciser phrasings, prototyping the structure of sections, and reviewing using the system prompt reproduced in Appendix A.1. Each AI output was reviewed and edited further; AI review supplemented rather than replaced my own review.

Development. AI was not used to write code for the prototype, but it was used to familiarise myself with the agda2hs codebase.

7 Conclusions and Future Work

The agda2hs compiler translates verified Agda to readable Haskell, erasing the proofs and type indices that encode postconditions in the process. The generated Haskell therefore has no way to check these properties at runtime. This paper asks how to translate Agda postconditions—expressed as lemmas, indexed datatypes, or sigma types—into QuickCheck property tests automatically.

The biggest challenge turns out not to be constructing the test itself, but turning the postcondition into something QuickCheck can run. When a postcondition is already Boolean, the translation is direct, but when it is encoded as a type there is nothing executable to translate. We address this by deriving checkers—Boolean functions that return whether the postcondition holds on given inputs. For extrinsic postconditions, expressed in lemmas and sigma types, we reuse the checker-derivation algorithm of Paraskevopoulou et al. [9]. For intrinsic postconditions, expressed in indexed datatypes, we adapt the algorithm.

The work has two main limitations. We present the automatic checker derivation as an algorithm design, not implemented in practice; the only running artefact is a lemma translation pipeline, which requires the user to write checking functions by hand. We also do not formally prove the derivation correct, so our confidence rests on an informal argument and the worked examples.

Future research could go in several directions. The most direct is implementing the full automatic checker derivation in agda2hs, which would let us exercise it on relations beyond our worked examples. A step further would be to derive each checker as an Agda term first and compile it to Haskell with agda2hs—that way a reflection metaprogram could generate a correctness proof alongside it. A third direction concerns when derived checkers fail to reach a verdict on certain inputs, discarding the test rather than deciding it. The order in which a checker resolves a relation's premises can determine whether it reaches a decision, so choosing that order well would discard fewer tests.

The results show that translating Agda postconditions to Haskell tests is feasible and, in principle, fully automatable; what remains is an implementation that makes this concrete.

A AI Prompts

This appendix presents sample prompts used during writing and development, in support of Section 6.

A.1 Review Prompt

We used the following prompt to review draft sections of this paper:

```
When asked to review parts of the thesis, evaluate
according to the following:

1. Jesper's writing issues
(jesper.cx/posts/writing-issues.html)

2. PL paper conventions: compare content, flow, and
style against papers in my Zotero. Flag deviations
from typical PL paper style.

3. Audience: a reader who has (nearly) completed a
Bachelor's degree in CS and is capable of finding
and reading scientific literature. The paper must
be understandable without referring to other work.
The paper need not explain a known method or
problem in detail, unless the contribution is a
modification of that method.

4. Correctness and precision: check every technical
claim. Flag anything imprecise, overstated, or
subtly wrong.

5. Redundancy and flow: flag repeated explanations,
sentences that say the same thing twice, paragraphs
that could be merged or cut, and transitions that
feel abrupt or missing.

6. Structure and navigation:
   - Does the overall section ordering make logical
   sense?
   - Do sections transition smoothly into each
   other?
   - Does each section deliver on what the previous
   section promised?
   - Are section and subsection titles descriptive
   and consistent?
   - Are figures and listings referenced correctly
   (no broken refs)?
   - Do figure captions explain the figure on their
   own?

Output: paragraph by paragraph, identify what needs
fixing, ordered by importance.
```

A.2 Conciseness Prompt

We used the following prompt to identify redundancies during revision:

```
Identify all redundancies in this paper.

1. Code-wise:
   - Replicated examples
   - Newly introduced examples that could instead
   use pre-existing figures
   - Non-compact presentation of the code (but
   still fixable with being readable)
   - Things to just omit in the code (irrelevant
   for the prose)

   Overall look for stuff that will reduce the
   space for code
   examples, while keeping the narrative.

2. Prose redundancies:
   - Re-explanation of a concept
   - Overly long explanation of items of low
   relevance
```

B Development Prompt

```
Which compile file would be sufficient to
understand enough to be able to write compileProp?
[...] AI suggested compilePostulate and a scan of
compileFun
[...] Cool, then generate an example input and I'm
gonna trace how it flows through the code
flag me if my trace is wrong.
```

References

[1] Robert Atkey. Syntax and semantics of quantitative type theory. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*, pages 56–65, 2018.

[2] Koen Claessen and John Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, pages 268–279. Association for Computing Machinery, 2000. doi:10.1145/351240.351266.

[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*, pages 108–122. Association for Computing Machinery, 2022. doi:10.1145/3546189.3549920.

[4] William A. Howard. The formulae-as-types notion of construction. In *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490. 1980.

[5] Antonio Locascio, Germán Andrés Delbianco, and Marco Stronati. Extracting property-based tests from F* specifications. In *ML Family Workshop*, 2022.

[6] Conor McBride. I got plenty o'nuttin'. In *A List of Successes That Can Change the World: Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*, pages 207–233. Springer, 2016.

[7] Jakob Naucke. Compiling Dependent Type Preconditions to Runtime Checks With Agda2Hs. Master's thesis, Delft University of Technology, 2024.

[8] Ulf Norell. *Towards a practical programming language based on dependent type theory*, volume 32. Chalmers University of Technology, 2007.

[9] Zoe Paraskevopoulou, Aaron Eline, and Leonidas Lampropoulos. Computing correctly with inductive relations. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 966–980. Association for Computing Machinery, 2022. doi:10.1145/3519939.3523707.

[10] Pierre-Marie Pédrot. Ltac2: tactical warfare. In *The Fifth International Workshop on Coq for Programming Languages, CoqPL*, volume 2019, 2019.