Reasonable Agda Is Correct Haskell: Writing Verified Haskell using agda2hs

Jesper Cockx
TU Delft
Delft, Netherlands
j.g.h.cockx@tudelft.nl

Orestis Melkonian
University of Edinburgh
Edinburgh, United Kingdom
IOG
Global
orestis.melkonian@ed.ac.uk

Lucas Escot
TU Delft
Delft, Netherlands
l.f.b.escot@tudelft.nl

James Chapman
IOG
Global
james.chapman@iohk.io

Ulf Norell
Gothenburg University
Gothenburg, Sweden
ulf.norell@cse.gu.se

Abstract

Modern dependently typed languages such as Agda can be used to statically enforce the correctness of programs. However, they still lack the large ecosystem of a more popular language like Haskell. To combine the strength of both approaches, we present AGDA2HS, a tool that translates an expressive subset of Agda to readable Haskell, erasing dependent types and proofs in the process. Thanks to Agda's support for erasure annotations, this process is both safe and transparent to the user. Compared to other tools for program extraction, AGDA2HS uses a syntax that is already familiar to functional programmers, allows for both intrinsic and extrinsic approaches to verification, and produces Haskell code that is easy to read and audit by programmers with no knowledge of Agda.

We present a practical use case of AGDA2HS at IOG to verify properties of a program generator. While both AGDA2HS and its ecosystem are still young, our experiences so far show that this is a viable approach to make verified functional programming available to a broader audience.

This paper is a literate Agda script, hence all rendered (Agda) code has been typechecked.

CCS Concepts: • **Software and its engineering** → Source code generation; Functional languages; Formal software verification; Correctness.

Keywords: Dependent types, Agda, Formal verification, Program extraction

Haskell '22, September 15–16, 2022, Ljubljana, Slovenia

ACM ISBN 978-1-4503-9438-3/22/09.
https://doi.org/10.1145/3546189.3549920

ACM Reference Format:
Jesper Cockx, Orestis Melkonian, Lucas Escot, James Chapman, and Ulf Norell. 2022. Reasonable Agda Is Correct Haskell: Writing Verified Haskell using agda2hs. In *Proceedings of the 15th ACM SIGPLAN International Haskell Symposium (Haskell '22), September 15–16, 2022, Ljubljana, Slovenia.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3546189.3549920

1 Introduction

Haskell is a strongly typed, purely functional programming language. One big advantage of purity is that it makes it easy to reason about the correctness of algorithms and data structures. This is demonstrated for example in the book by Hutton [2016, Chapter 16], where equational reasoning, case analysis, and induction are used to prove properties of Haskell functions. However, since these proofs are done 'on paper', there is always a risk that the proof contains a mistake, or that the code changes to a new version but the proof is not updated.

In a dependently typed programming language such as Agda [Agda Development Team 2021], we can do better and actually write a formal proof of correctness in the language itself. This proof is checked automatically by the typechecker. Moreover, any time the program is modified, the proof is re-checked, so it is guaranteed to be up-to-date with the latest version. Thus, it provides an unusually high degree of confidence in the correctness of the program.

Unfortunately, despite several big successes in dependently typed programming such as the CompCert verified C compiler [Leroy 2009], these languages remain relatively inaccessible to the general public. Compared to a language like Haskell, they lack an extensive ecosystem of libraries, tools for testing, profiling, debugging, and integration with other languages. Since none of the current languages with theorem proving capabilities has the user base to support such an ecosystem, we must look into alternative methods to make verified functional programming accessible to a broader audience.

```
data Tree (@0 l u : Nat) : Set where
  Leaf  : (@0 pf : l ≤ u) → Tree l u
  Node : (x : Nat) → Tree l x → Tree x u → Tree l u
{-# COMPILE AGDA2HS Tree #-}

insert : {@0 l u : Nat} (x : Nat) → Tree l u
  → @0 (l ≤ x) → @0 (x ≤ u) → Tree l u
insert x (Leaf _) l≤x x≤u = Node x (Leaf l≤x) (Leaf x≤u)
insert x (Node y l r) l≤x x≤u =
  case compare x y of λ where
    (LT x≤y)  → Node y (insert x l l≤x x≤y) r
    (EQ x≡y) → Node y l r
    (GT y≤x) → Node y l (insert x r y≤x x≤u)
{-# COMPILE AGDA2HS insert #-}
```

```
data Tree = Leaf
          | Node Natural Tree Tree

insert :: Natural -> Tree -> Tree
insert x Leaf = Node x Leaf Leaf
insert x (Node y l r)
  = case compare x y of
      LT -> Node y (insert x l) r
      EQ -> Node x l r
      GT -> Node y l (insert x r)
```

Figure 1. A representation of binary search trees in Agda that enforces the internal invariants, and an intrinsically verified insertion function (left) and the corresponding Haskell definitions produced by AGDA2HS (right). Note that all traces of the verification of the bounds are erased from the Haskell code.

Another issue with current tools for code extraction is that we might start from a large existing codebase in a language such as Haskell, where it is not feasible to rewrite all code in a proof assistant and prove it correct. In this case, we might want to improve our trust in an existing program by reimplementing a critical component in a verified way and dropping the result into the existing code. We could call this a "seismic correctness retrofit", analogous to the "seismic security retrofit" [Fisher et al. 2017], which is itself inspired by the technique in structural engineering where existing buildings are made more resistant to earthquakes by improving their foundations and other critical components.

We present AGDA2HS[1], a tool that aims to combine the rich ecosystem of Haskell with the verification capabilities of Agda. In particular, AGDA2HS identifies a common subset of the two languages, and provides a faithful translation of this subset from Agda to Haskell. This translation erases all traces of dependent types and proofs embedded in the program, leaving just clean and readable Haskell code. Thus AGDA2HS allows library developers to implement libraries in Agda and verify their correctness, and then translate the result to Haskell so it can be used and understood by Haskell programmers. In particular, it can be audited by Haskell programmers without any Agda expertise of their own.

As an example, we define a type of binary trees with values between two given bounds in Fig. 1. At each node the lower and upper bounds of the subtrees are iteratively tightened, and each leaf contains evidence that the bounds are consistent. AGDA2HS makes use of *erasure annotations* `@0` [Atkey 2018; McBride 2016] to mark explicitly which parts of the code should be erased during the translation to Haskell. After this erasure is finished, we end up with a simple Haskell datatype.

Next, we implement an insertion function that requires the inserted value between the lower and upper bound and ensures the bounds are unchanged. AGDA2HS translates this function to a simple Haskell function. Thanks to the expressive Agda type of this function, we can be sure that it preserves the internal invariants of the tree.

AGDA2HS allows you to verify as many or as few properties as you desire. On one side of the spectrum, you could write what amounts to Haskell code in Agda syntax, benefitting from Agda's support for *type-driven development* with interactive features such as holes and case splitting as well as powerful metaprogramming facilities through *elaborator reflection* [Christiansen and Brady 2016]. You can then gradually increase the trustworthiness of the program by adding extra indices to datatypes or proof arguments to this code, statically enforcing pre-/post-conditions and invariants while keeping the generated Haskell code simple and clean. With these more expressive types, errors in the program are detected as it is being written.

On the other side of the spectrum, you have the full power of Agda's theorem proving capabilities at your fingertips to prove functional correctness of your algorithms, using your choice of extrinsic or intrinsic approaches to verification. In extrinsic verification the proof is completely separate from the program, while in intrinsic verification, properties are encoded directly in the type of data and functions. Extrinsic verification makes it easier to decouple the programs from their proofs, while intrinsic verification often does not require writing any proofs as such, which makes it more intuitive to functional programmers. For both intrinsic and extrinsic verification, all type indices and proofs can be erased

[1] AGDA2HS is available at https://github.com/agda/agda2hs.

from the generated code, leaving just a clean and simply-typed Haskell program.

Contributions.

- We present AGDA2HS, a tool for automatically translating a subset of Agda to readable Haskell code (Sect. 2).
- We demonstrate how the *erasure annotations* of McBride [2016] can be used to make erasure of proofs and type-level indices both safer and more explicit to the user (Sect. 2.1).
- We describe a practical use case of AGDA2HS at IOG, where it is used to verify properties of a program generator (Sect. 3).

We discuss the correctness of our approach in Sect. 4 and its implementation in Sect. 5. Sect. 6 presents related work, and Sect. 7 concludes.

This paper is a literate Agda script, hence all rendered (Agda) code has been typechecked.

2 AGDA2HS

The goal of AGDA2HS is to produce nice looking, readable, and verified Haskell code. One way of achieving this would be to take the Haskell code as given and translate it to Agda or another theorem prover for verification. As explained in the introduction, this makes it impossible to use intrinsic verification, and it decouples the program from its proof of correctness, which means that they risk going out of sync.

So instead we have chosen to translate in the other direction. That is, programs are written in a subset of Agda that supports a straightforward translation to readable Haskell code. Since the translation erases proofs and properties from the code, these can be written using the full Agda language without compromising the readability of the resulting Haskell code. The resulting workflow is rather satisfying; the user interactively edits an Agda file as usual, with the only difference being additional COMPILE AGDA2HS pragmas for invoking compilation of specific definitions.

2.1 Erasure of Type-Level Arguments

Some parts of a Agda type or expression may be present only for the purpose of proving some property or invariant and can thus be erased from the Haskell code. These parts often rely on dependent types in an essential way so they are difficult or impossible to represent faithfully on the Haskell side. In particular, we want to erase indices of indexed datatypes, thus translating a *correct-by-construction*, *intrinsically typed* datatype such as Tree in Fig. 1 directly to a plain Haskell type with no indices.

To determine which arguments of an expression should be erased, AGDA2HS adheres to the following rules:

- Any argument that is marked as *erased*[2] with an `@0` annotation [Atkey 2018] is erased from the Haskell code.
- Any argument of type Level (used for universe polymorphism in Agda) or Set ℓ is also erased.

By marking[3] arguments that should be erased with `@0`, we get the guarantee from Agda's type system that the runtime behaviour of the function does not depend on the value of the erased argument, so erasing the argument is sound (see Sect. 4 for more details on the correctness of our approach). Likewise, pattern matching on values of type Level or Set ℓ is impossible, so erasing them is sound as well.

2.2 Intrinsic versus Extrinsic Verification

The translation described thus far accommodates both intrinsic and extrinsic verification. We already saw an example of intrinsically verifying the insert function in Fig. 1, but one can also arrive at the exact same translation artifact–as well as prove the same guarantees–*extrinsically*, as demonstrated in Fig. 2.

In this case, we opt for a plain, non-dependent Tree with no type indices and a corresponding simply typed insert. Verification happens *a posteriori* as a separate theorem after the definitions. Notice how the form of insert remains almost identical through the translation, while there is a significant overhead to compensate for the type information we dropped from the intrinsic version.

2.3 Primitives & Prelude

If we are to accomplish seamless integration across language barriers, it is of utmost importance to make the current definitions in Haskell's Prelude accessible on the Agda side as well. To this end, we have ported most of Haskell's Prelude on the Agda side, staying as faithful as possible to the Haskell idiom. This essentially forms the *trusted computing base*, along with the code implementing the actual translation. That is, we have to be absolutely certain that the Agda implementation of these functions corresponds precisely to the original functionality in Haskell. Alas, there are cases where this process is not as straightforward as one might initially expect.

Haskell's primitive types and operations are matched directly to their Agda counterparts if they exist (e.g. Agda's Agda.Builtin.Nat is translated to Haskell's Numeric.Natural), otherwise we provide a new Agda implementation in the Haskell.Prelude module as close as possible to the original interface and extend the translation to account for this new mapping (e.g. n-ary tuples, if-then-else notation, and pattern matching with `case`). For a better user

[2]Also known as *run-time irrelevant*, see https://agda.readthedocs.io/en/v2.6.2/language/runtime-irrelevance.html.

[3]In theory, these erasure annotations could be inferred automatically, but this would require an extension of Agda with erasure inference [Tejiscak 2020], which is outside of our current scope.

```
data Tree : Set where
  Leaf  : Tree
  Node : Nat → Tree → Tree → Tree
{-# COMPILE AGDA2HS Tree #-}

insert : Nat → Tree → Tree
insert x Leaf = Node x Leaf Leaf
insert x (Node y l r) =
  case compare x y of λ where
    (LT _) → Node y (insert x l) r
    (EQ _) → Node y l r
    (GT _) → Node y l (insert x r)

{-# COMPILE AGDA2HS insert #-}

@0 _≤_≤_ : Nat → Tree → Nat → Set
l ≤ Leaf         ≤ u = l ≤ u
l ≤ Node x tˡ tʳ ≤ u = (l ≤ tˡ ≤ x) × (x ≤ tʳ ≤ u)

@0 insert-correct : ∀ {t x l u} → l ≤ t ≤ u
  → l ≤ x → x ≤ u → l ≤ insert x t ≤ u
insert-correct {Leaf} _ l≤x x≤u = l≤x , x≤u
insert-correct {Node y tˡ tʳ} {x} (IHˡ , IHʳ) l≤x x≤u
  with compare x y
... | LT x≤y  = insert-correct IHˡ l≤x x≤y , IHʳ
... | EQ refl  = IHˡ , IHʳ
... | GT y≤x = IHˡ , insert-correct IHʳ y≤x x≤u
```

```
data Tree = Leaf
          | Node Natural Tree Tree

insert :: Natural -> Tree -> Tree
insert x Leaf = Node x Leaf Leaf
insert x (Node y l r)
  = case compare x y of
      LT -> Node y (insert x l) r
      EQ -> Node y l r
      GT -> Node y l (insert x r)
```

Figure 2. An unverified insertion function and its extrinsic proof of correctness (left) and the Haskell code produced by AGDA2HS (right). The type _≤_≤_ extrinsically specifies the invariant that was enforced intrinsically by the Tree type in Fig. 1.

experience, the backend discovers when built-in types are used and automatically inserts necessary imports in the generated Haskell code.

To simulate Haskell's automatic insertion of universal quantifiers for type variables, the Haskell.Prelude also exports certain often-used lowercase letters as *generalised type variables*.[4] This allows us to write for example id : $a \to a$ instead of id : $\forall\{a\} \to a \to a$.

2.4 Partiality

A particularly striking difference between Haskell and Agda is the former's support for *partial* functions, such as `head` that throws an error on empty lists. Although there is a way to model partiality in Agda and precisely match the original behaviour in Haskell, we leave this for future work (c.f. Sect. 7) as it would introduce significant complexity to the translation. Instead, we provide a more direct work-around for the time being, by restricting all definitions to be **total**. This is achieved by attaching additional proof obligations as a function's *preconditions*, to be erased after compilation, that guarantee the function is total and thus never reaches the erroneous path. However, when actually executing the generated code, it is entirely possible for the function to be called with invalid arguments that do not satisfy these preconditions, therefore we still want an appropriate error message to be reported on this occasion.

To model this behaviour in Agda without losing logical consistency, we require that any calls to error occur only when an impossible case has been reached, in which case the auxiliary precondition would provide a witness of the empty/void type.

```
error : (@0 i : ⊥) → String → a
error ()
```

As an example, we can make `head` total by requiring the input list to be non-empty as can be seen in Fig. 3. Notice how the context of the second clause contains the absurd proposition `p : NonEmpty []`, which allows us to safely call error. Compiling down to Haskell produces the expected partial implementation.

[4] https://agda.readthedocs.io/en/v2.6.2/language/generalization-of-declared-variables.html

```
head : (xs : List a) {@0 _ : NonEmpty xs} → a
head (x :: _) = x
head [] {p}   = error i "Prelude.head: empty list"
  where @0 i : ⊥
        i = case p of λ ()
```

```
head :: [a] -> a
head (x : _) = x
head [] = error "Prelude.head: empty list"
```

Figure 3. The definition of the head function in Agda, taking as argument a proof that the given list is non-empty (left) and the partial function definition generated by AGDA2HS with the proof erased (right).

The actual error definition in the AGDA2HS prelude uses a *tactic argument*[5] to automatically discover an absurdity in the current context, hence requiring no value of the empty type to be explicitly provided.

2.5 Type Classes

Haskell type classes [Wadler and Blott 1989] can easily be emulated using instance resolution[6] on the Agda side. Concretely, **class definitions** are represented as records containing the class methods as fields and the class context as type parameters. The compiler pragma for class definitions should explicitly end with the `class` keyword, to distinguish from the case of translating a plain record. **Instance declarations** are then modeled with regular definitions of the record type, instantiated to the specific type whose instance we are implementing. Finally, constraints in a qualified type or qualified instance declaration compile to instance arguments holding the translated type. Fig. 4 shows the case for the `Monoid` type class. Naturally, we can also enforce that instances satisfy the monoid laws by adding *erased* fields to the record, requiring proofs to be provided when implementating them on the Agda side, yet resulting in identical Haskell output.

Default methods. Haskell also supports *default method implementations*, as well as a way to specify the *minimal complete definition* of a class. Unfortunately, there is no equivalent functionality in Agda, so we have to devise a way to emulate the same behaviour using existing features. While doing so, we have to avoid introducing a lot of syntactic overhead on the Agda side, but at the same time keep the translation procedure as simple as possible.

The solution we propose consists of having a main record containing the method type signatures like before, and then representing each minimal complete definition as a separate record, whose fields are only a subset of the original fields and the rest are given as derived definitions. Default methods, on the other hand, have to be replicated across all minimal records and contain the exact same code. That way, all minimal records contain all class methods, divided amongst its fields and definitions. Combined with the fact that a record definition also defines a module of the same name containing all its fields and definitions,[7] we can easily retrieve a value of the original record by opening the *record-module* of the minimal record value: when defining an instance of a class C with possible minimal record C_1, the user can choose to construct a minimal record value of type $r : C_1$ instead (with strictly less fields to specify) and construct the final record value with the expression record { C_1 r }. Internally, AGDA2HS will chase down the definition of the minimal record and inline its definition to generate the expected 'flat' Haskell definition.

In Fig. 5, we see an example translation for the `Show` type class. The Show record is the original record you would expect, but there are now two additional minimal records: $Show_1$ only requires an implementation of showsPrec and derives show from it, while $Show_2$ does the inverse, and both have a (matching) default implementation for showList. Finally, the compiler directive is extended to inform the backend about these minimal records. To declare an instance for Maybe, one can now simply define a value s_1 of type $Show_1$ Maybe and transform it into a value of type Show Maybe using the Agda syntax record {$Show_1$ s_1}, which instantiates the remaining fields show and showList of Show with their implementations given in $Show_1$.

While things on the Agda side are more verbose than we would like, we have managed to offer a systematic and intuitive way to cover these type class features without compromising the quality of the generated code, as witnessed in the generated Haskell code of Fig. 5.

3 Case Study: Verifying Properties of a Program Generator

IOG uses AGDA2HS for a programming language implementation [Chapman et al. 2019]. Programs are written in Haskell and compiled to a simple core language based on System F^{μ}_{ω}. As System F^{μ}_{ω} supports computation at the type level, types are arguably programs too.

Wen Kokke contributed a novel program generator that exhaustively generates System F^{μ}_{ω} types and programs using NEAT [Duregård 2016], a property-based testing system similar to SmallCheck [Runciman et al. 2008]. At the

[5] https://agda.readthedocs.io/en/v2.6.2/language/reflection.html#tactic-arguments

[6] https://agda.readthedocs.io/en/v2.6.2/language/instance-arguments.html#instance-resolution

[7] https://agda.readthedocs.io/en/v2.6.2/language/record-types.html#record-modules

```
record Monoid (a : Set) : Set where
  field
    mempty  : a
    mappend : a → a → a
    @0 left-identity  : mappend mempty x ≡ x
    @0 right-identity : mappend x mempty ≡ x
    @0 associativity  : mappend (mappend x y) z ≡
                        mappend x (mappend y z)
open Monoid {{...}} public
{-# COMPILE AGDA2HS Monoid class #-}

instance
  MonoidNat : Monoid Nat
  MonoidNat = λ where
    .mempty      → 0
    .mappend i j → i + j
    .left-identity  → ...
    .right-identity → ...
    .associativity  → ...
{-# COMPILE AGDA2HS MonoidNat #-}

sumMon : {{ Monoid a }} → List a → a
sumMon []       = mempty
sumMon (x :: xs) = mappend x (sumMon xs)
{-# COMPILE AGDA2HS sumMon #-}
```

```
class Monoid a where
  mempty :: a
  mappend :: a -> a -> a

instance Monoid Nat where
  mempty = 0
  mappend i j = i + j

sumMon :: Monoid a => [a] -> a
sumMon [] = mempty
sumMon (x : xs) = mappend x (sumMon xs)
```

Figure 4. Using records and instance arguments to represent Haskell's Monoid type class.

time of writing, the generator is used to test the production Haskell implementation of the core language against the Agda model which is compiled using MAlonzo and makes use of the same Haskell libraries for cryptography etc. The NEAT generator for types and terms was originally written in Haskell but the type level part has subsequently been ported to Agda with minimal changes, and is compiled using AGDA2HS. Before using the NEAT generator in testing we would like to verify that it is correctly implemented. One of the key components of the generator is a type checker for terms that makes essential use of type substitution which, in turn, relies on type renaming. We could try to test the correctness properties of type renaming and substitution operations by generating types and using NEAT but instead we prove the properties in Agda. It is straightforward to prove these properties in Agda whereas to test them would be computationally expensive, as it would require not only generating individual types but also arbitrary renamings and substitutions containing an arbitrary number of types.

We introduce the intrinsically well-scoped syntax of kinds and types of System F_ω (see Fig. 6). The types of System F_ω are similar to the terms of lambda calculus extended with constants for function arrow and forall. Instead of indexing by the (natural) number of variables and using Fin to manage variable scope, we index by sets of variables and use Maybe to manage variable scope. This representation, due to [Bird and Paterson 1999], has the advantage that it is readily translated to a nested datatype in Haskell rather than a GADT. The former representation, due to [Altkenkirch and Reus 1999], is more commonly used in Agda. Note the use of Maybe in the arguments of TyForall and TyLam which signifies that the variable set n has been extended by an additional element. We define operations to apply (parallel) renamings and (parallel) substitutions to types (see Fig. 7). The datatype Type is a functor where ren is the morphism map and the functor laws are given by ren-id and ren-comp:

```
ren-id   : (ty : Type n) → ren id ty ≡ ty
ren-comp : (ty : Type n) (ρ : n → n') (ρ' : n' → n'')
         → ren (ρ' ∘ ρ) ty ≡ ren ρ' (ren ρ ty)
```

The datatype Type is also a monad where TyVar is return and sub is bind. The three monad laws are given by sub-id, sub-var and sub-comp:

```
sub-id   : (t : Type n) → sub TyVar t ≡ t
sub-var  : (x : n) (σ : n → Type n')
         → sub σ (TyVar x) ≡ σ x
sub-comp : (ty : Type n)
         → (σ : n → Type n') (σ' : n' → Type n'')
         → sub (sub σ' ∘ σ) ty ≡ sub σ' (sub σ ty)
```

```
record Show (a : Set) : Set where
  field show     : a → String
        showsPrec : Nat → a → ShowS
        showList  : List a → ShowS

record Show₁ (a : Set) : Set where
  field showsPrec : Nat → a → ShowS

  show : a → String
  show x = showsPrec 0 x ""

  showList : List a → ShowS
  showList = defaultShowList (showsPrec 0)

record Show₂ (a : Set) : Set where
  field show : a → String

  showsPrec : Nat → a → ShowS
  showsPrec _ x s = show x ++ s

  showList : List a → ShowS
  showList = defaultShowList (showsPrec 0)

open Show {{...}}
{-# COMPILE AGDA2HS Show class Show₁ Show₂ #-}

instance
  ShowMaybe : {{Show a}} → Show (Maybe a)
  ShowMaybe {a = a} = record {Show₁ s₁}
    where
      s₁ : Show₁ (Maybe a)
      s₁ .Show₁.showsPrec n = λ where
        Nothing → showString "nothing"
        (Just x)  → showParen True
          ( showString "just " ∘ showsPrec 10 x )
{-# COMPILE AGDA2HS ShowMaybe #-}
```

```
class Show a where
  show :: a -> String
  showsPrec :: Nat -> a -> ShowS
  showList :: [a] -> ShowS
  {-# MINIMAL showsPrec | show #-}
  show x = showsPrec 0 x ""
  showList = defaultShowList (showsPrec 0)
  showsPrec _ x s = show x ++ s

instance (Show a)
      => Show (Maybe a) where
  showsPrec n = \case
    Nothing -> showString "nothing"
    (Just x) -> showParen True
      (showString "just " . showsPrec 10 x)
```

Figure 5. Translating the `Show` type class with a single default method (`showList`) and two possible minimal definitions (`showsPrec | show`).

```
data Kind : Set where
  Star  :  Kind
  _:=>_ :  Kind → Kind → Kind

data Type (n : Set) : Set where
  TyVar    : n → Type n
  TyFun    : Type n → Type n → Type n
  TyForall : Kind → Type (Maybe n) → Type n
  TyLam    : Type (Maybe n) → Type n
  TyApp    : Type n → Type n → Kind → Type n
```

```
data Kind
  = Star
  | Kind :=> Kind

data Type n
  = TyVar n
  | TyFun (Type n) (Type n)
  | TyForall Kind (Type (Maybe n))
  | TyLam (Type (Maybe n))
  | TyApp (Type n) (Type n) Kind
```

Figure 6. The definition of kinds and well-scoped types of System F_ω in Agda (left) and the corresponding datatypes generated by AGDA2HS (right).

```
ren : (n → n') → Type n → Type n'
sub : (n → Type n') → Type n → Type n'
```

```
ren :: (n -> n') -> Type n -> Type n'
sub :: (n -> Type n') -> Type n -> Type n'
```

Figure 7. The type signatures for parallel renaming and parallel substitution in Agda (left) and the corresponding Haskell type signatures generated by AGDA2HS (right).

We then use substitution (implemented in Agda and compiled to Haskell) to implement type reduction which is used in typechecking. Type reduction and typechecking are implemented in Haskell. Type checking is used in the generator to ensure that all generated terms are well typed.

4 Correctness of Agda2Hs

Since the goal of AGDA2HS is to produce verified Haskell code, correctness of the translation is an important consideration. On a high level, one could argue that the semantics of Agda and Haskell are sufficiently close to each other, and the translation is sufficiently simple, that the correctness of the translation is obvious. However, on a closer look one can see that AGDA2HS erases certain parts of a term, and if it erases too much then this can affect the semantics of a term. For example, in an earlier version of AGDA2HS, the translation would erase all implicit arguments instead of all arguments marked as erased with `@0`. However, with this rule the function sneaky in Fig. 8 was translated incorrectly.[8] In the new version of AGDA2HS, all implicit arguments are required to have a `@0` annotation, which means that Agda will check that the run-time result of the function is not dependent on the erased argument, causing the function sneaky to be rejected.

Because both Agda and Haskell are rather large languages and neither has a full formal specification of its semantics, it is currently not feasible to provide a formal correctness proof for AGDA2HS. However, what we can do is give a precise statement of the properties of the translation that we expect to hold, and of the assumptions that need to be true for these properties to hold.

We can break down the correctness of AGDA2HS into two main properties:

1. If AGDA2HS successfully produces a Haskell file from a given Agda file, then this file is valid Haskell code.
2. If an Agda term evaluates to some value v, then the corresponding Haskell term produced by AGDA2HS evaluates to the (translation of the) same value v.

Of these properties, the second one is the most important: when the first property fails, we at least know that something is wrong when trying to compile the produced Haskell code. However, when the second property fails (as in the example of Fig. 8) the properties we have proven about the Agda code might fail to hold for the generated Haskell code.

[8] https://github.com/agda/agda2hs/issues/88

Validity of generated Haskell code. In order for AGDA2HS to generate valid Haskell code, the Agda code needs to fall within the common subset of Haskell and Agda that is supported by AGDA2HS as described above. In particular, the code should satisfy the following requirements:

- The code needs to obey the naming rules of Haskell: type and term constructors need to start with a capital letter, while names of functions and variables need to start with a small letter.
- Names of constructors and record fields should not be overloaded (which is supported by Agda but not by Haskell).
- The Agda module needs to import the module Haskell.Prelude provided by AGDA2HS, and should not reuse any names that are used in the Haskell Prelude.
- All defined functions, types, and classes that are used in non-erased positions of a definition with a COMPILE AGDA2HS pragma should either be defined in Haskell.Prelude or have a COMPILE AGDA2HS pragma themselves. In particular, functions from the builtin Agda.Builtin.* modules should not be used directly.
- All type parameters of functions and type constructors should be of a kind that is translated to a valid Haskell kind.
- Type quantifiers and constraints should only appear at the top-level of a function type. (Higher-rank types are currently not supported, though they would not be difficult to add.)
- All type arguments and class instances used in applications of polymorphic functions should be inferrable by Haskell, since they are erased by AGDA2HS.
- Type class instances in the Agda code should adhere to Haskell's requirement of global uniqueness.

From the point of view of an Agda programmer these restrictions may seem onerous, but for a user of AGDA2HS it may help to remember that you are not writing Agda code: you are writing Haskell code which happens to be using the syntax of Agda.

Currently, most of these properties are not yet checked by AGDA2HS. However, as noted before this just means that the Haskell compiler will reject the generated code. In practice, this can sometimes be annoying as the user gets an error message from Haskell rather than from AGDA2HS, however there is no risk of unsoundness.

```
data Vec (a : Set) : {n : Nat} → Set where
  Nil  : Vec a {zero}
  Cons : {n : Nat} → a → Vec a {n} → Vec a {suc n}

sneaky : {n : Nat} → Vec Bool {n} → Nat
sneaky     Nil          = 0
sneaky {1} (Cons True xs) = 1
sneaky     (Cons x xs)   = 2
```

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

sneaky :: Vec Bool -> Int
sneaky Nil = 0
sneaky (Cons True xs) = 1
sneaky (Cons x xs) = 2
```

Figure 8. A function that would be translated incorrectly if AGDA2HS would erase all implicit arguments: in Agda, sneaky (Cons True (Cons False Nil)) evaluates to 2, but the translated Haskell term evaluates to 1 instead.

Preservation of semantics. Most syntactic constructs of Agda are just mapped one-to-one to the corresponding Haskell construct. Hence, the most important assumption that needs to hold in order to guarantee the preservation of the semantics of programs compiled by AGDA2HS is that the Agda and Haskell semantics of these constructs agree. Since both Agda and Haskell are declarative and purely functional languages, it is sufficient to check that each evaluation step of an Agda term corresponds to zero or more evaluation steps on the Haskell side. We distinguish the following cases:

- Lambda-abstractions and function applications in Agda are translated to the corresponding constructs in Haskell, so a β-reduction on the Agda side corresponds to a β-reduction of the translated Haskell term.
- Record values and record projections in Agda are translated to their equivalent constructs in Haskell, so their evaluation is likewise preserved.
- Function clauses in Agda are translated to function clauses in Haskell, so any evaluation step of a function in Agda corresponds to an evaluation of the translated function in the Haskell code. Both languages use a first-match approach to pattern matching, and Agda forbids pattern matching on arguments that are marked as erased (as well as arguments of type Set or Level), hence the same clause will always be chosen on both sides of the translation.
- Record values that are marked as an instance on the Agda side are translated to type class instances in the Haskell code. Any call to a field of this record that uses this instance is translated to a call to the corresponding method in the Haskell code. Thanks to Haskell's guarantee of global uniqueness of type class instances, we know that the same instance will be inferred by Haskell (or else the translation will fail).

Note also that Agda guarantees that all terms are strongly normalizing, so any differences in the order of evaluation do *not* influence the correctness of AGDA2HS.

Other assumptions that need to hold to ensure preservation of the semantics of translated programs are:

- **Prelude**: The implementation of functions in Haskell.Prelude and Haskell.Prim.* should be consistent with their Haskell implementation.
- **Erasure**: All implicit arguments that are erased by AGDA2HS are either marked with `@0`, or have a return type of the form Level or Set *i*.
- **Type classes**: All instance arguments (that are not marked with `@0`) should be canonical, i.e. they should be constructed from declared instances that are compiled to Haskell instances.

The first property is currently only checked manually, but only needs to be checked once for each function in the AGDA2HS prelude. The other two properties are checked by AGDA2HS, so any violation will lead to an error during the translation to Haskell.

5 The Implementation of AGDA2HS

In order to implement our translation we use Agda's support for custom backends, which allows easy development of custom code generators similar to the built-in ones for generating Haskell,[9] Javascript, HTML and LaTeX. In contrast to the builtin backends that can be activated with a flag, new custom backends in Agda are standalone Haskell executables that import Agda's internals as a library. This makes them faster and easier to develop as it shortens the edit-compile-run loop considerably. It also means that backend releases do not need to be tied to Agda releases.

To implement the backend one needs to provide a compilation function from Agda definitions to Haskell ones. Crucially, the backend operates on the phase after typechecking, reading the generated interface `.agdai` files for the (successfully checked) definitions and spitting out their Haskell translations. (Practically, our translation function is executed under Agda's typechecking monad, allowing us to conveniently re-use Agda internals for our own purposes.)

[9] We compare the existing Haskell code generator with our approach and explain why it is unsuitable for our purposes in Sect. 6.1.

The user can then invoke the translation on specific definitions via the COMPILE pragma or include verbatim Haskell code in a FOREIGN block.[10]

As the source and target languages share a lot of similarities, a major part of the translation process is straightforward, merely translating Agda constructs to the corresponding Haskell constructs (e.g. for lambda abstractions). However, there are some cases that are subtle due to differences between the source and target language that need more careful attention.

First and foremost, the backend only handles a certain subset of Agda, excluding the parts that have no Haskell equivalent (e.g. defining a type by pattern matching, indexed datatypes). In these cases, the backend rejects the unsupported input with an appropriate error message. Note that the user can still use dependent types to prove theorems about their non-dependently typed programs, but these are only typechecked without any counterpart on the Haskell side.

In some other cases, the design of the two languages diverge and a similar construct is handled in a totally different way, as in the case of 'where' clauses: where clauses in Agda introduce anonymous modules with parameters drawn from the enclosing scope, so we have to remedy this during translation by tracking down this module and compiling its contained definitions applied to the current scope, essentially reversing Agda's *lambda-lifting* phase.

6 Related Work

We distinguish two main forms of related work: other tools for extracting executable code from a proof assistant, and tools for verifying code written in Haskell directly.

6.1 Program Extraction

AGDA2HS can be seen as a form of *program extraction*, as used in theorem provers such as Coq[11] [Letouzey 2002, 2008], Isabelle/HOL [Bulwahn et al. 2008; Haftmann and Nipkow 2010], Minlog [Berger et al. 2011], ACL2 [Greve et al. 2008], and Russell [Sozeau 2007]. Compared to these approaches, we expect our approach to be much more familiar and easy to learn to a (Haskell) programmer: instead of first having to write a mathematical proof and then extracting the executable program from that, the programmer can instead write an Agda program with precise and expressive types without being forced into proving anything. One important difference with other program extractors is that we do not consider the full source language and instead restrict ourselves to a subset of Agda that corresponds to valid Haskell, so we do not need to insert any calls to `unsafeCoerce` or similar functions. This restriction also means we can generate code that is easy to read and understand independently, so a Haskell programmer does not need to learn Agda in order to use a library written using AGDA2HS. Moreover, a programmer using AGDA2HS can collaborate easily with other programmers/maintainers who are using Haskell directly as they are able to review the generated code they would normally do. This is not the case for extraction mechanisms that generate reliable, performant, but unreadable code even if the mechanism is fully verified. Furthermore if the AGDA2HS programmer makes no further contributions to the project the other developers can take over maintenance of Haskell code without having to use AGDA2HS. This is a particular advantage in open source projects where activity from a particular contributor may be transient. Another difference with extraction is that we use explicit erasure annotations, which allows us to erase all traces of proof arguments. For example, an Agda function $\mathsf{div} : (x\ y : \mathsf{Nat}) \to @0\ y \not\equiv 0 \to \mathsf{Nat}$ is translated to a Haskell function `div : Natural -> Natural -> Natural`. The price we pay for this is that nothing prevents a programmer from calling `div 1 0` on the Haskell side. Finally, it is important to note that the goal of AGDA2HS is not necessarily to compete directly with existing program extraction tools. Rather, our aim is to build a lightweight tool that is easy to use and produces readable Haskell code in a predictable manner.

MAlonzo backend. Agda already has a Haskell backend called MAlonzo, so one might wonder why we do not just use that instead. MAlonzo is designed to compile any Agda program to Haskell. Hence it can handle arbitrarily complex dependently typed programs and produce relatively performant Haskell code. The downside is that it does so by circumventing the Haskell type system. The nature of this process also makes the generated code almost unreadable and generally hard to work with: see Fig. 9 for an example of the code generated by MAlonzo. While many of the coercions inserted by MAlonzo are not necessary to make the code be accepted by GHC, they cannot be avoided in general because Agda supports full dependent types, while GHC (currently) does not. The coercions inserted by MAlonzo make it difficult to make the jump from having an Agda prototype of a Haskell program (or a component in a larger Haskell project) to having a production Haskell program. Arguably, when we use MAlonzo our trustworthy verified Agda code is compiled to Haskell code that is less trustworthy than handwritten Haskell code as it cannot be meaningfully checked by either a human reader or the Haskell typechecker. The AGDA2HS project is aimed at solving this problem, producing trustworthy Haskell output from verified Agda input.

Coq extraction. The built-in extraction command in Coq seems to produce considerably more readable Haskell output than MAlonzo, however we *do* get unwanted implicit

[10] https://agda.readthedocs.io/en/v2.6.2/language/foreign-function-interface.html#compiler-pragmas
[11] https://coq.inria.fr/refman/addendum/extraction.html

```
d_insert_1494 :: Integer -> Integer -> Integer
              -> T_Tree_1340 -> T__'8804'__1324 -> T__'8804'__1324 -> T_Tree_1340
d_insert_1494 ~v0 ~v1 v2 v3 ~v4 ~v5 = du_insert_1494 v2 v3
du_insert_1494 :: Integer -> T_Tree_1340 -> T_Tree_1340
du_insert_1494 v0 v1 = case coe v1 of
  C_Leaf_1348 -> coe C_Node_1352 (coe v0) (coe C_Leaf_1348) (coe C_Leaf_1348)
  C_Node_1352 v2 v3 v4 -> coe MAlonzo.Code.Haskell.Prim.du_case_of__54
    (coe d_compare_1474 (coe v0) (coe v2))
    (coe du_'46'extendedlambda0_1514 (coe v0) (coe v2) (coe v3) (coe v4))
  _ -> MAlonzo.RTE.mazUnreachableError
```

Figure 9. A fragment of the Haskell code generated by MAlonzo from the Agda code in Fig. 1. MAlonzo circumvents Haskell's type system with unsafe casts with `coe`.

arguments and automatically generated variable names in the extracted Haskell code of Fig. 10. (This can be remedied by manually instructing the erasure of these arguments: `Extraction Implicit insert [ l u ].`)

In contrast to AGDA2HS, Coq does not seem to readily support type classes, as seen in Fig. 11 where classes get translated to plain datatypes and instances are given in *dictionary-passing style*.

We can easily conceive a port of our work to Coq that adds proper Prelude support and handles type classes, but it still remains more natural to choose Agda whose syntax and idioms are closer in spirit to Haskell's.

Reflection-based extraction. Sinkarovs and Cockx [2021] show how Agda's reflection API can be used for the purpose of developing custom code extractors from within Agda. This work is mostly orthogonal to ours: they explain how to implement an extractor, while we focus here on the use of a specific extraction tool AGDA2HS. We expect it would have been possible to implement AGDA2HS using reflection instead, but we decided to use Agda's backend mechanism instead because it is more mature and allows easy integration with existing libraries for generating Haskell syntax.

6.2 Verification of Haskell Code

Since Haskell code is amenable to equational reasoning and formal verification, there have been many attempts to verify code written in Haskell directly. One obvious point of comparison for AGDA2HS is `hs-to-coq` [Breitner et al. 2018; Spector-Zabusky et al. 2018]. While `hs-to-coq` starts from Haskell code that is translated to Coq code that must then be verified by hand, we start from Agda code that is (intrinsically or extrinsically) verified and translate that to Haskell, erasing proofs and type dependencies in the process. Compared to `hs-to-coq`, with AGDA2HS it is possible to:

- change or refine the type signature of functions to ensure totality,
- verify properties using a mix of intrinsic and extrinsic verification,
- develop and maintain programs using the interactive features of Agda,
- build libraries that are usable from both Haskell and Agda.

Of course, there is a price to be paid for these advantages, the most significant part is that AGDA2HS does not support verification of existing Haskell code directly. While this is a real disadvantage for some applications, writing both the program and its proof of correctness in the same language allows for a much tighter integration between the two. Also, thanks to Agda's FFI features we can still make use of existing Haskell functions as long as we are happy to postulate their properties.

Abel et al. [2005] also used Agda to verify properties of Haskell programs. However, their approach is closer to `hs-to-coq` than to ours: they translate GHC's Core language into a monadic embedding in Agda. As a consequence, most of the advantages and disadvantages from the comparison with `hs-to-coq` apply here as well. In the 16 years since publication of this paper, there have been many changes to Agda so applying this approach would require a whole new implementation. Thanks to significant advances in Agda's usability as a programming language, it is now entirely feasible to develop both programs and their proofs side-by-side in Agda directly.

Christiansen et al. [2019] use Coq to prove properties of *partial* Haskell programs by wrapping computations in a (free) monad. This could similarly be supported in AGDA2HS by working with the Maybe monad to model possible failure and/or the General monad to model general recursion [McBride 2015].

Liquid Haskell [Vazou 2016; Vazou et al. 2018, 2014] extends Haskell with refinement types in order to specify and automatically check correctness properties in Haskell code directly. When it can be applied, this approach clearly has a much lower overhead than using either `hs-to-coq` or AGDA2HS. However, not all properties can easily be expressed as refinement types. Moreover, using an automatic prover means that it is much harder to guide the proving process in case it fails. Vazou et al. [2017] make a detailed

```
Inductive Tree {l u : nat} : Set :=
    Leaf : l <= u -> Tree
  | Node : forall x,
      @Tree l x -> @Tree x u -> Tree.

Fixpoint insert {l u} x (t : @Tree l u)
 (p : l <= x) (q : x <= u) : Tree :=
  match t with
  | Leaf _ => Node x (Leaf p) (Leaf q)
  | Node y l r => match cmp x y with
    | LT q => Node y (insert x l p q) r
    | EQ q => Node y l r
    | GT p => Node y l (insert x r p q)
    end
  end.
```

```
data Tree = Leaf
          | Node Nat Tree Tree

insert :: Nat -> Nat -> Nat
       -> Tree -> Tree
insert l u x t = case t of {
  Leaf -> Node x Leaf Leaf;
  Node y l0 r -> case cmp x y of {
    LT -> Node y (insert l y x l0) r;
    EQ -> Node y l0 r;
    GT -> Node y l0 (insert y u x r)}}
```

Figure 10. The insertion example from Fig. 1; implicit arguments `l` and `u` survive the translation.

```
Class Monoid (a : Set) :=
  { mempty  : a
  ; mappend : a -> a -> a }.

Instance MonoidNat : Monoid nat :=
  { mempty := 0
  ; mappend i j := i + j }.

Fixpoint sumMon {a} `{Monoid a}
  (xs : list a) : a :=
  match xs with
  | [] => mempty
  | x :: xs => mappend x (sumMon xs)
  end.
```

```
data Monoid a = Build_Monoid a (a -> a -> a)

mempty :: (Monoid a1) -> a1
mempty = ...
mappend :: (Monoid a1) -> a1 -> a1 -> a1
mappend = ...
monoidNat :: Monoid Nat
monoidNat = Build_Monoid O add

sumMon :: (Monoid a1) -> (List a1) -> a1
sumMon h xs = case xs of {
  ([]) -> mempty h;
  (:) x xs0 -> mappend h x (sumMon h xs0)}
```

Figure 11. Default class extraction in Coq that produces no Haskell type classes and exposes internal names.

comparison between Liquid Haskell and Coq. We expect AGDA2HS to share many of the characteristics of Coq in this comparison.

GHC Haskell is also steadily getting closer to becoming a full-fledged dependently typed language itself [Eisenberg 2016; Weirich et al. 2017]. Once this transformation is complete, we expect it will replace AGDA2HS for some use cases, in particular for encoding pre-/post-conditions and invariants at the type level. However, even with dependent types Haskell will remain a partial language and hence not suited for actual theorem proving, while Agda comes with a termination checker as well as a large standard library that can be used for writing proofs. Another benefit of AGDA2HS is that it produces *simple* Haskell code that can be used by any Haskell programmer, while effectively using code written in dependent Haskell will still require a learning curve.

7 Conclusion and Future Work

In this paper, we have presented AGDA2HS and demonstrated its usefulness for producing verified Haskell code. We were particularly delighted by the streamlined development process provided by Agda's backend infrastructure; we implemented the translation without significant development effort, and the codebase is minimal enough to allow for future extensions.

Supporting more Haskell features. While the current subset of Haskell and Agda used by AGDA2HS is expressive enough to write many programs, our experience has shown that there is a need for more features that facilitate working with intrinsically typed data structures. In the future, we plan to continue extending AGDA2HS with additional features to make it possible to verify a wider range of programs. Our aim is to represent more features of Haskell in Agda, not to compile arbitrary Agda code to Haskell. This is what allows us to produce clean and usable Haskell code in the end. Among the features of Haskell that are not yet supported by AGDA2HS, we can identify three rough categories:

- First, we have Haskell features that already have an Agda equivalent that is equally or more expressive. For example, this is the case for generalized algebraic

data types (GADTs) [Jones et al. 2006] and higher-rank types. Since these Agda features are more general than their Haskell counterparts, the main difficulty here is to identify the fragment of the Agda syntax that can be supported by AGDA2HS.
- Second, there are Haskell features that have no direct Agda equivalent but that can be emulated with some creativity. One example of this is the support for default methods we already implemented (see Sect. 2.5). Another example would be to use Agda's with abstraction [McBride and McKinna 2004] to support pattern guards and view patterns.
- Finally, some features have no built-in equivalent in Agda and are not easily emulated either. One example of this is 32-bit arithmetic, which is supported by Haskell (`Float`, `Int32`, `Word32`), but has no built-in equivalent in Agda. Other examples include strictness annotations, unboxed datatypes, and specialize pragmas. In order to support these features in AGDA2HS, they would first need to be added to Agda itself.

Compiling static properties to run-time checks. With our current approach, function preconditions and datatype invariants that are enforced statically on the Agda side are erased in the translation to Haskell. This is problematic when AGDA2HS is used to develop a part of a larger Haskell project: 'plain' Haskell code can call a function without checking the preconditions, or construct an element of a datatype that violates the invariants. To provide an extra layer of safety, we could add run-time checks to the generated Haskell code. This is possible for any property that is decidable, which is usually the case (e.g. the decidability of the tree invariants from Sect. 1 is a consequence of the decidable ordering of the natural numbers).

Another possible extension would be to integrate static verification and property-based testing by allowing the user of AGDA2HS to postulate certain properties on the Agda side and translate those properties to QuickCheck properties [Claessen and Hughes 2000]. However, it remains unclear how to account for random generation and shrinking of test data.

Totality and infinite data. A limitation of the current approach of AGDA2HS is that all properties are proven in Agda by assuming functions are total, while Haskell functions can result in an error or fail to terminate. While there is a good argument to be made for why this is usually not a big problem [Danielsson et al. 2006], it would be interesting to investigate the possibility of compiling monadic Agda code (using some kind of partiality monad) to pure Haskell code. This would give a more accurate representation of Haskell code at the price of additional overhead on the Agda side.

We do not provide any support for infinite data structures yet either, but we are currently experimenting with representing them using *coinductive types*.[12]

HS2AGDA: translating in the inverse direction. Sometimes we have existing Haskell code that we would like to re-use, rather than porting it in AGDA2HS from scratch. To this end, it would be convenient to provide the developer with a *back-translation* from Haskell to AGDA2HS-compliant Agda code, which can then serve as a starting point for verification. Such a tool would streamline the verification process and make AGDA2HS easier to adopt within an existing project.

Applications of AGDA2HS. On the side of applications, so far we have focused on the verification of libraries that offer purely functional data structures and algorithms. However, typical Haskell programs involve a significant amount of monads to manage side effects. Further experiments are needed to investigate how to apply AGDA2HS to monadic code. We would also like to evaluate the usability and effectiveness of our approach by comparing the verification effort with competing approaches such as Liquid Haskell and `hs-to-coq`.

Acknowledgments

We would like to thank Philip Wadler and the anonymous reviewers for their helpful feedback and insightful comments.

This work was supported by Input Output (iohk.io) through their funding of the Edinburgh Blockchain Technology Lab.

Jesper Cockx holds an NWO Veni grant on 'A trustworthy and extensible core language for Agda' (VI.Veni.202.216).

References

Andreas Abel, Marcin Benke, Ana Bove, John Hughes, and Ulf Norell. 2005. Verifying Haskell programs using constructive type theory. In *Proceedings of the ACM SIGPLAN Workshop on Haskell, Haskell 2005, Tallinn, Estonia, September 30, 2005*, Daan Leijen (Ed.). ACM, 62–73. https://doi.org/10.1145/1088348.1088355

Agda Development Team. 2021. *Agda 2.6.2 documentation.* https://agda.readthedocs.io/en/v2.6.2/

Thorsten Altkenkirch and Bernhard Reus. 1999. Monadic presentations of lambda terms using generalized inductive types. In *Computer Science Logic (CSL), 1999.*

Robert Atkey. 2018. Syntax and Semantics of Quantitative Type Theory. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*, Anuj Dawar and Erich Grädel (Eds.). ACM, 56–65. https://doi.org/10.1145/3209108.3209189

Ulrich Berger, Kenji Miyamoto, Helmut Schwichtenberg, and Monika Seisenberger. 2011. Minlog - A Tool for Program Extraction Supporting Algebras and Coalgebras. In *Algebra and Coalgebra in Computer Science - 4th International Conference, CALCO 2011, Winchester, UK, August 30 - September 2, 2011. Proceedings (Lecture Notes in Computer Science,*

[12] https://agda.readthedocs.io/en/v2.6.2/language/coinduction.html

Vol. 6859). Springer, 393–399. https://doi.org/10.1007/978-3-642-22944-2_29

Richard Bird and Ross Paterson. 1999. de Bruijn notation as a nested datatype. *The Journal of Functional Programming* 9 (January 1999). Issue 1.

Joachim Breitner, Antal Spector-Zabusky, Yao Li 0004, Christine Rizkallah, John Wiegley, and Stephanie Weirich. 2018. Ready, set, verify! Applying hs-to-coq to real-world Haskell code (experience report). *Proceedings of the ACM on Programming Languages* 2, ICFP (2018). https://doi.org/10.1145/3236784

Lukas Bulwahn, Alexander Krauss, Florian Haftmann, Levent Erkök, and John Matthews. 2008. Imperative Functional Programming with Isabelle/HOL. In *Theorem Proving in Higher Order Logics, 21st International Conference, TPHOLs 2008, Montreal, Canada, August 18-21, 2008. Proceedings (Lecture Notes in Computer Science, Vol. 5170)*, Otmane Aït Mohamed, César Muñoz, and Sofiène Tahar (Eds.). Springer, 134–149. https://doi.org/10.1007/978-3-540-71067-7_14

James Chapman, Roman Kireev, Chad Nester, and Philip Wadler. 2019. System F in Agda, for Fun and Profit. In *Mathematics of Program Construction - 13th International Conference, MPC 2019, Porto, Portugal, October 7-9, 2019, Proceedings (Lecture Notes in Computer Science, Vol. 11825)*, Graham Hutton (Ed.). Springer, 255–297. https://doi.org/10.1007/978-3-030-33636-3_10

David Christiansen and Edwin Brady. 2016. Elaborator reflection: extending Idris in Idris. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*, Jacques Garrigue, Gabriele Keller, and Eijiro Sumii (Eds.). ACM, 284–297. https://doi.org/10.1145/2951913.2951932

Jan Christiansen, Sandra Dylus, and Niels Bunkenburg. 2019. Verifying Effectful Haskell Programs in Coq. In *Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell* (Berlin, Germany) *(Haskell 2019)*. Association for Computing Machinery, New York, NY, USA, 125–138. https://doi.org/10.1145/3331545.3342592

Koen Claessen and John Hughes. 2000. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18-21, 2000*, Martin Odersky and Philip Wadler (Eds.). ACM, 268–279. https://doi.org/10.1145/351240.351266

Nils Anders Danielsson, John Hughes, Patrik Jansson, and Jeremy Gibbons. 2006. Fast and loose reasoning is morally correct. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006*, J. Gregory Morrisett and Simon L. Peyton Jones (Eds.). ACM, 206–217. https://doi.org/10.1145/1111037.1111056

Jonas Duregård. 2016. *Automated Black-Box Property Based Testing*. Ph. D. Dissertation. Chalmers University and Göteborg University.

Richard A. Eisenberg. 2016. Dependent Types in Haskell: Theory and Practice. *CoRR* abs/1610.07978 (2016). arXiv:1610.07978 http://arxiv.org/abs/1610.07978

Kathleen Fisher, John Launchbury, and Raymond Richards. 2017. The HACMS program: using formal methods to eliminate exploitable bugs. *Philosophical Transactions of the Royal Society* 375 (September 2017). Issue 2104.

David A. Greve, Matt Kaufmann, Panagiotis Manolios, J. Strother Moore, Sandip Ray, José-Luis Ruiz-Reina, Rob Sumners, Daron Vroon, and Matthew Wilding. 2008. Efficient execution in an automated reasoning environment. *Journal of Functional Programming* 18, 1 (2008), 15–46. https://doi.org/10.1017/S0956796807006338

Florian Haftmann and Tobias Nipkow. 2010. Code Generation via Higher-Order Rewrite Systems. In *Functional and Logic Programming, 10th International Symposium, FLOPS 2010, Sendai, Japan, April 19-21, 2010. Proceedings (Lecture Notes in Computer Science, Vol. 6009)*, Matthias Blume, Naoki Kobayashi, and Germán Vidal (Eds.). Springer, 103–117. https://doi.org/10.1007/978-3-642-12251-4_9

Graham Hutton. 2016. *Programming in Haskell* (2nd edition ed.). Cambridge University Press.

Simon L. Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. 2006. Simple unification-based type inference for GADTs. In *Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming, ICFP 2006, Portland, Oregon, USA, September 16-21, 2006*, John H. Reppy and Julia L. Lawall (Eds.). ACM, 50–61. https://doi.org/10.1145/1159803.1159811

Xavier Leroy. 2009. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (2009), 107–115. https://doi.org/10.1145/1538788.1538814

Pierre Letouzey. 2002. A New Extraction for Coq. In *Types for Proofs and Programs, Second International Workshop, TYPES 2002, Berg en Dal, The Netherlands, April 24-28, 2002, Selected Papers (Lecture Notes in Computer Science, Vol. 2646)*, Herman Geuvers and Freek Wiedijk (Eds.). Springer, 200–219. http://link.springer.de/link/service/series/0558/bibs/2646/26460200.htm

Pierre Letouzey. 2008. Extraction in Coq: An Overview. In *Logic and Theory of Algorithms, 4th Conference on Computability in Europe, CiE 2008, Athens, Greece, June 15-20, 2008, Proceedings (Lecture Notes in Computer Science, Vol. 5028)*, Arnold Beckmann, Costas Dimitracopoulos, and Benedikt Löwe (Eds.). Springer, 359–369. https://doi.org/10.1007/978-3-540-69407-6_39

Conor McBride. 2015. Turing-Completeness Totally Free. In *Mathematics of Program Construction - 12th International Conference, MPC 2015, Königswinter, Germany, June 29 - July 1, 2015. Proceedings (Lecture Notes in Computer Science, Vol. 9129)*, Ralf Hinze and Janis Voigtländer (Eds.). Springer, 257–275. https://doi.org/10.1007/978-3-319-19797-5_13

Conor McBride. 2016. 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 (Lecture Notes in Computer Science, Vol. 9600)*, Sam Lindley, Conor McBride, Philip W. Trinder, and Donald Sannella (Eds.). Springer, 207–233. https://doi.org/10.1007/978-3-319-30936-1_12

Conor McBride and James McKinna. 2004. The view from the left. *J. Funct. Program.* 14, 1 (2004), 69–111. https://doi.org/10.1017/S0956796803004829

Colin Runciman, Matthew Naylor, and Fredrik Lindblad. 2008. Smallcheck and lazy smallcheck: automatic exhaustive testing for small values. In *Proceedings of the 1st ACM SIGPLAN Symposium on Haskell, Haskell 2008, Victoria, BC, Canada, 25 September 2008*, Andy Gill (Ed.). ACM, 37–48. https://doi.org/10.1145/1411286.1411292

Artjoms Sinkarovs and Jesper Cockx. 2021. Extracting the power of dependent types. In *GPCE '21: Concepts and Experiences, Chicago, IL, USA, October 17 - 18, 2021*, Eli Tilevich and Coen De Roover (Eds.). ACM, 83–95. https://doi.org/10.1145/3486609.3487201

Matthieu Sozeau. 2007. Program-ing finger trees in Coq. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming, ICFP 2007, Freiburg, Germany, October 1-3, 2007*, Ralf Hinze and Norman Ramsey (Eds.). ACM, 13–24. https://doi.org/10.1145/1291220.1291156

Antal Spector-Zabusky, Joachim Breitner, Christine Rizkallah, and Stephanie Weirich. 2018. Total Haskell is reasonable Coq. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2018, Los Angeles, CA, USA, January 8-9, 2018*, June Andronick and Amy P. Felty (Eds.). ACM, 14–27. https://doi.org/10.1145/3167092

Matus Tejiscak. 2020. A dependently typed calculus with pattern matching and erasure inference. *Proceedings of the ACM on Programming Languages* 4, ICFP (2020). https://doi.org/10.1145/3408973

Niki Vazou. 2016. *Liquid Haskell: Haskell as a Theorem Prover*. Ph. D. Dissertation. University of California, San Diego, USA. https://escholarship.org/uc/item/8dm057ws

Niki Vazou, Joachim Breitner, Rose Kunkel, David Van Horn, and Graham Hutton. 2018. Theorem proving for all: equational reasoning in Liquid Haskell (functional pearl). In *Proceedings of the 11th ACM SIGPLAN*

International Symposium on Haskell, Haskell@ICFP 2018, St. Louis, MO, USA, September 27-17, 2018, Nicolas Wu (Ed.). ACM, 132–144. https://doi.org/10.1145/3242744.3242756

Niki Vazou, Leonidas Lampropoulos, and Jeff Polakow. 2017. A tale of two provers: verifying monoidal string matching in Liquid Haskell and Coq. In *Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell, Oxford, United Kingdom, September 7-8, 2017*, Iavor S. Diatchki (Ed.). ACM, 63–74. https://doi.org/10.1145/3122955.3122963

Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon L. Peyton Jones. 2014. Refinement types for Haskell. In *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014*, Johan Jeuring and Manuel M. T. Chakravarty (Eds.). ACM, 269–282. https://doi.org/10.1145/2628136.2628161

Philip Wadler and Stephen Blott. 1989. How to Make ad-hoc Polymorphism Less ad-hoc. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, January 11-13, 1989*. ACM Press, 60–76. https://doi.org/10.1145/75277.75283

Stephanie Weirich, Antoine Voizard, Pedro Henrique Avezedo de Amorim, and Richard A. Eisenberg. 2017. A specification for dependent types in Haskell. *Proceedings of the ACM on Programming Languages* 1, ICFP (2017). https://doi.org/10.1145/3110275