You got Agda in my Haskell!

A tale as good as peanut butter and chocolate

Ryan Orendorff

January 2023

agda2hs: Convert your Agda code to Haskell

agda2hs is a tool that allows us to transpile Agda into Haskell that looks like it was written by a person.

Much of this talk follows ideas from the code base and the associated blog post/paper.

- https://jesper.sikanda.be/files/reasonable-agda-is-correct-haskell.pdf
- https://jesper.sikanda.be/posts/agda2hs.html
- https://github.com/agda/agda2hs

Outline

We will be talking about using agda2hs in the following contexts.

- Extrinsic validation: validating existing code
- Intrinsic validation: property carrying code
- Obeying the law
- Final comments

Extrinsic Verification

Verifying properties of existing functions: map

Mapping over a list is easy–we create a new list where we apply some function `f` to every element.

```
map : {a b : Set} → (a → b) → List a → List b
map f [] = []
map f (x :: xs) = f x :: map f xs
{-# COMPILE AGDA2HS map #-} -- The peanut butter/chocolate collider
```

Through the magic of agda2hs we get the following Haskell code.

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x : xs) = f x : map f xs
```

Property based testing: maps should preserve list length

We can check some properties of our implementation. For example, we can test that running our map function does not change the size of the list.

```
import Test.QuickCheck


map-length :: [Int] -> Bool
map-length xs = length (map f xs) == length xs
  where
    f = (+ 1)
```

This is nice but we haven't shown it to work for *every input*.

Validating map-length for all inputs

Recall our function is defined as

```
map f [] = []
map f (x :: xs) = f x :: map f xs
```

We can prove our map preserves the length of the list

```
map-length : {a b : Set} → (f : a → b) → (xs : List a)
                → length (map f xs) ≡ length xs
map-length f [] = refl -- length [] ≡ length []
map-length f (x :: xs) = begin
  length (f x :: map f xs) ≡⟨⟩
  suc (length (map f xs)) ≡⟨ cong suc (map-length f xs) ⟩
  suc (length xs)          ≡⟨⟩
  length (x :: xs)         ∎
```

We have *extrinsically* proven a property of our code: we have demonstrated some property of existing code.

We can use type classes as well

We can also add type class constraints to our code. Conveniently Agda's instance arguments are mapped directly to type classes. If we write the following function

```
min : ⦃ Ord a ⦄ → a → a → a
min x y = if x < y then x else y
```

we get the this Haskell code as an output.

```
min :: Ord a ⇒ a → a → a
min x y = if x < y then x else y
```

We can prove properties on type class constraints

Say we want to prove that `min x y ≡ min y x`

```
min x y = if x < y then x else y
```

First we need a helper lemma

```
postulate
  -- Not correct reflexively; used for example only
  flip< : (x y : a) → {b : Bool} → ⦃ p : Ord a ⦄
        → ((x < y) ≡ b) → ((y < x) ≡ not b)
```

and then we can prove our `min` property for any type that implements the `Ord` type class.

```
min-commute : (x y : a) → ⦃ p : Ord a ⦄ → min x y ≡ min y x
min-commute x y with x < y in eq
... | False rewrite (flip< x y eq) = refl
... | True rewrite (flip< x y eq) = refl
```

Extrinsic validation: proofs about regular code

We have seen how to prove properties about non-dependently typed programs

- Simple properties on higher order functions (`map-length`)
- Properties on functions involving type classes (`min-commute`)

But we can do more when we encode properties in our types!

Intrinsic verification

Dependent types: types that depend on values

A non-dependent function takes an input of some type A and returns a type B, written as

$$f : A \to B$$

whereas a dependent function returns a value in a *family of types* B indexed by the *value* of x.

$$f : (x : A) \to B\ x$$

As an example, if we convert a Nat to an integer we may want to use the smallest representation of that integer.

```
optimalInt : (x : Nat) → Set
optimalInt x = if x < 2 ^ 64 - 1 then Int else Integer

natToInt : (x : Nat) → optimalInt x
natToInt x = cast (optimalInt x) x
```

The (canonical) example of dependent types: Vectors

The vector type is a list that carries its length.

```
-- The @0 means the value is erased at runtime
data Vec (a : Set) : (@0 n : Nat) → Set where
  Nil : Vec a 0
  Cons : {@0 n : Nat} → a → Vec a n → Vec a (1 + n)
```

which compiles to the following haskell

```
-- Just a list!
data Vec a = Nil
           | Cons a (Vec a)
```

Instrinsic properties enforce what we can write

Unlike extrinstic properties, intrinsic properties like the length of the vector force us to hold a property.

For example, if we try to write an incorrect map function

```
mapV : {@0 n : Nat} → (a → b) → Vec a n → Vec b n
mapV f Nil = Nil
mapV f (Cons x xs) = mapV f xs -- whoops, we dropped x!
```

then Agda will throw an error saying that `n` is not preserved.

```
n ≠ (1 + n) of type ℕ
when checking that the expression xs has type Vec a (1 + n)
```

This means we don't need to write the `map-length` proof!

Proving more complex properties: interleaving

Let's say we want to interleave two vectors:

```
interleave : {a : Set} {@0 m n : Nat} → Vec a m → Vec a n
           → Vec a (m + n)
interleave Nil ys = ys
interleave (Cons x xs) ys = Cons x (interleave ys xs)
```

But Agda will complain because the inputs to `interleave` flip in the recursive call, meaning the result is `n + m` instead of `m + n`!

An (erased) proof that + commutes

What we need to show is that addition is *commutative*, which is to say you can flip the inputs and get the same result.

```
@0 +-comm-erased : (@0 m n : ℕ) → m + n ≡ n + m
+-comm-erased zero    n = sym (+-identityʳ n)
+-comm-erased (suc m) n = begin
  suc m + n   ≡⟨⟩
  suc (m + n) ≡⟨ cong suc (+-comm m n) ⟩
  suc (n + m) ≡⟨ sym (+-suc n m) ⟩
  n + suc m   ∎
```

Notice that we have defined the property as an erased property: we produce a proof we can store only in locations that are erased. In addition, we can not pattern match on this proof.

Safe casting by using a proof

We can cast one type to another if we can prove that their parts are equivalent.

```
transport : (@0 p : @0 a → Set) {@0 m n : a}
          → @0 m ≡ n → p m → p n
transport p refl t = t
{-# COMPILE AGDA2HS transport transparent #-}
-- The transparent makes id functions disappear!
```

This allows us to define the needed proof for interleave

```
comm-Vec : {a : Set} (@0 m n : Nat) → Vec a (m + n) → Vec a (n + m)
comm-Vec {a} m n v = transport (Vec a) (+-comm-erased m n) v
{-# COMPILE AGDA2HS comm-Vec transparent #-}
```

Using transport to implement interleave

With our new cast `Vec a (m + n) → Vec a (n + m)` from `comm-Vec`, we can finish our interleave function.

```
interleave : {a : Set} {@0 m n : Nat} → Vec a m → Vec a n
           → Vec a (m + n)
interleave Nil ys = ys
interleave {a} {n = n} (Cons {m} x xs) ys =
  Cons x (comm-Vec n m (interleave ys xs))
```

And we get simple Haskell out of it!

```
interleave :: Vec a → Vec a → Vec a
interleave Nil ys = ys
interleave (Cons x xs) ys = Cons x (interleave ys xs)
```

Instrinsic properties verify we can't be wrong in the first place

Intrinsic properties ensure we write code correctly during construction, which can lead to less proof code later.

Extrinsic verification and intrinsic verification can be used together.

What other types of Agda code an we convert to Haskell?

Type classes that obey the law

Not another Monad tutorial...

The most famous Haskell type class is the `Monad` type class, which looks like the following in Agda.

```
record Monad (m : Set → Set) : Set₁ where
  field
    _>>=_ : m a → (a → m b) → m b
    overlap ⦃ super ⦄ : Applicative m

  return : a → m a
  return = pure

-- Don't compile this, use existing Haskell definition
{-# COMPILE AGDA2HS Monad existing-class #-}
```

Monad laws

In order for a monad to act as we expect, we need it to have a few properties:

```
left-id  : (return x >>= f) ≡ f x

right-id : (k >>= return) ≡ k

assoc    : ((k >>= f) >>= g) ≡ (k >>= (λ x → f x >>= g))
```

Are all monads law abiding?

One may ask what stops a (nefarious) programmer from implementing the monad functions in ways that break the laws.

> *There are plenty of illegal monads out there.*
> *The monad police don't come and arrest you, you just lose the ability to reason about them sanely in your code.*
> *- Edward Kmett on Reddit*

... but what if we could call the monad police?

Quick, call Agda-1-1!

Figure 1: Yours for only $149!

Get it now at https://www.badgecreator.com/vb01.php!

A Lawful Monad

Our Monad Inspector is particular about the types of monads they certify. Only monads with the following properties get a certificate of authenticity.

```
record LawfulMonad (m : Set → Set) : Set₁ where
  field
    overlap ⦃ super ⦄ : Monad m

    @0 left-id : ∀ {a b} (x : a) (f : a → m b)
               → (return x >>= f) ≡ f x
    @0 right-id : ∀ {a} (k : m a)
                → (k >>= return) ≡ k
    @0 assoc : ∀ {a b c} (k : m a)
             → (f : a → m b) (g : b → m c)
             → ((k >>= f) >>= g) ≡ (k >>= (λ x → f x >>= g))
```

What happens when our Monad Inspector meets `Maybe`?

When our `Maybe` type asks for a certificate from the Monad Inspector, `Maybe` provides the following record.

```
instance
  _ : LawfulMonad Maybe
  _ = record {
    left-id = λ x f → refl;
    right-id = λ {
        Nothing → refl;
        (Just x) → refl};
    assoc = λ {
        Nothing f g → refl;
        (Just x) f g → refl} }
```

And now monadic functions can safely use `Maybe`

With our certified `Maybe` monad, we can then use it in places like `mapM_` to ensure only monads that obey the law can be used.

```
lawfulmapM_ : {a b : Set} {m t : Set → Set}
            → ⦃ LawfulMonad m ⦄ → ⦃ Foldable t ⦄
            → (a → m b) → t a → m ⊤
lawfulmapM_ f = foldr (λ x k → f x >> k) (pure tt)
```

Limitations and Conclusion

Some Agda functionality is hard to replicate

A few things cannot be done with `agda2hs` at the moment.

- GADTs.
- Default methods (partially added).
- Things that are in Haskell but not in Agda (Float32).
- Convert preconditions to runtime checks.

agda2hs is pretty dope!

agda2hs let's us do some amazing things:

- Convert Agda code to Haskell that is extrinsically verified
- Write intrinsically verified code and remove the "dependent" part for Haskell
- We can finally appease the Monad Inspector.

Questions?

Ask away!