Reasonable Agda Is Correct Haskell:

Writing Verified Haskell using agda2hs

Jesper Cockx, Orestis Melkonian, Lucas Escot, James Chapman, Ulf Norell

Motivation: issues with current program extractors

MAlonzo covers the entirety of Agda, but produces unreadable code:

```
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
```

Motivation: issues with current program extractors

Coq extracts more reabable code, but still does not readily support typeclasses:

```coq
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.
```

```haskell
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)}
```

Goals

1. Writing Haskell within Agda (no need to cover the whole source language)
2. Verify your program using Agda's dependent types

Goals

1. Writing Haskell within Agda (no need to cover the whole source language)
2. Verify your program using Agda's dependent types

New point in the design space, enabled by:

- Agda very *similar* to Haskell + = ❤️
- Agda's *dependent type system*
- Agda's support for *erasure*

\+ allows for intrinsic verification!

Tree example (extrinsic version)

```
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 #-}
```

```
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)
```

Tree example (extrinsic proofs)

```
            ⋮

@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
```

Tree example (intrinsic version)

```
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)
```

```haskell
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)
```

Primitives

- Export lowercase type variables to feel like home:

  $\mathsf{id} : a \longrightarrow a$

  $\mathsf{id}\ x = x$

Primitives

- Export lowercase type variables to feel like home:

  $\mathrm{id} : a \longrightarrow a$

  $\mathrm{id}\ x = x$
- Match Agda built-ins to Haskell built-ins:

  e.g. Agda.Builtin.Nat $\leftrightarrow$ `Numeric.Natural`

Primitives

- Export lowercase type variables to feel like home:

  id : $a \longrightarrow a$

  id $x = x$

- Match Agda built-ins to Haskell built-ins:

  e.g. Agda.Builtin.Nat $\leftrightarrow$ `Numeric.Natural`

- If not available in Agda, define them:

  infix -2 if_then_else_

  if_then_else_ : Bool $\longrightarrow a \longrightarrow a \longrightarrow a$

  if False then x else $y = y$

  if True then x else $y = x$

Primitives

- Export lowercase type variables to feel like home:

  ```
  id : a → a
  id x = x
  ```

- Match Agda built-ins to Haskell built-ins:

  e.g. Agda.Builtin.Nat ↔ `Numeric.Natural`

- If not available in Agda, define them:

  ```
  infix -2 if_then_else_
  if_then_else_ : Bool → a → a → a
  if False then x else y = y
  if True then x else y = x
  ```

REMEMBER

We want to cover as many Haskell features as possible, not Agda features.

Prelude

Port Haskell's Prelude, staying faithful to the original functionality

What about partial functions such as `head`?

Prelude

Port Haskell's Prelude, staying faithful to the original functionality

What about partial functions such as `head`?

- ⇒ implement safe version with extra preconditions
- ⇒ only allow calls to `error` in unreachable cases:

$$
\begin{aligned}
&\mathsf{error} : (@0\ i : \bot) \longrightarrow \mathsf{String} \longrightarrow a\\
&\mathsf{error}\ ()\\[1em]
&\mathsf{head} : (xs : \mathsf{List}\ a)\ \{@0\ \_ : \mathsf{NonEmpty}\ xs\} \longrightarrow a\\
&\mathsf{head}\ (x :: \_) = x\\
&\mathsf{head}\ []\ \{p\} \quad = \mathsf{error}\ \mathsf{i}\ \text{"head: empty list"}\\
&\quad \mathsf{where}\ @0\ \mathsf{i} : \bot\\
&\qquad\qquad \mathsf{i} = \mathsf{case}\ p\ \mathsf{of}\ \lambda\ ()
\end{aligned}
$$

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

Don't forget

On the Haskell side, we can feed `head` arbitrary input!

Typeclasses

Correspondence with Agda's **instance arguments**.

- class definitions $\sim$ record types
- instance declarations $\sim$ record values
- constraints $\sim$ instance arguments

Typeclasses: class definitions ∼ record types

```agda
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 #-}
```

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

Typeclasses: instance declarations ∼ record values

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

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

Typeclasses: constraints ∼ instance arguments

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

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

Default methods & minimal complete definitions

```
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 x = showsPrec 0 x ""
  showList = defaultShowList (showsPrec 0)
record Show₂ (a : Set) : Set where
  field show : a → String
  showsPrec _ x s = show x ++ s
  showList = defaultShowList (showsPrec 0)
open Show {{...}}
{-# COMPILE AGDA2HS Show class Show₁ Show₂ #-}
```

```
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
```

Minimal Instance

```
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 #-}
```

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

IOG use case

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

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

IOG use case: Laws

ren is a *functorial map* on Type.

- ren-id: $(ty : \mathsf{Type}\ n) \longrightarrow \mathsf{ren}\ \mathsf{id}\ ty \equiv ty$
- ren-comp: $(ty : \mathsf{Type}\ n)\ (\rho : n \longrightarrow n')\ (\rho' : n' \longrightarrow n'')$
  $\longrightarrow \mathsf{ren}\ (\rho' \circ \rho)\ ty \equiv \mathsf{ren}\ \rho'\ (\mathsf{ren}\ \rho\ ty)$

IOG use case: Laws

ren is a *functorial map* on Type.

- ren-id: $(ty : \text{Type } n) \longrightarrow \text{ren id } ty \equiv ty$
- ren-comp: $(ty : \text{Type } n)\ (\rho : n \longrightarrow n')\ (\rho' : n' \longrightarrow n'')$
  $\longrightarrow \text{ren } (\rho' \circ \rho)\ ty \equiv \text{ren } \rho'\ (\text{ren } \rho\ ty)$

sub is a *monadic bind* on Type.

- sub-id: $(t : \text{Type } n) \longrightarrow \text{sub TyVar } t \equiv t$
- sub-var: $(x : n)\ (\sigma : n \longrightarrow \text{Type } n') \longrightarrow \text{sub } \sigma\ (\text{TyVar } x) \equiv \sigma\ x$
- sub-comp: $(ty : \text{Type } n)\ (\sigma : n \longrightarrow \text{Type } n')\ (\sigma' : n' \longrightarrow \text{Type } n'')$
  $\longrightarrow \text{sub } (\text{sub } \sigma' \circ \sigma)\ ty \equiv \text{sub } \sigma'\ (\text{sub } \sigma\ ty)$

Correctness

How do we know the translation is **sound**?

1. Trust the ported Prelude and defined primitives
2. Ensure all dependent types appear under *erased* positions
3. Ensure source code also adheres to Haskell's naming conventions
    - this check is actually relegated to GHC! + = ❤️

NOTE

all functions are total $\Rightarrow$ evaluation order doesn't matter

Implementation

https://github.com/agda/agda2hs

Agda Input

↓ surface syntax

Agda

↓ (type-checked) internal core representation

AGDA2HS

↓ `:: Agda.AST -> Agda.TC Haskell.AST`

Haskell

Surface

$\mathsf{f} : \mathsf{Nat} \longrightarrow \mathsf{Nat}$
$\mathsf{f}\ x = \mathsf{go}$
  where
    $\mathsf{go} = \mathsf{TODO}$
    -- may use x

Intermediate

$\mathsf{go} : \mathsf{Nat} \longrightarrow \mathsf{Nat}$
$\mathsf{go}\ x = \mathsf{TODO}$

$\mathsf{f} : \mathsf{Nat} \longrightarrow \mathsf{Nat}$
$\mathsf{f}\ x = \mathsf{go}\ x$

Output

```
f :: Natural -> Natural
f x = go
  where go = TODO
```

Future work

Still many unsupported Haskell features:

- GADTs
- pattern guards, views
- 32-bit arithmetic
- Infinite data
- Non-termination, general recursion

Future work

Still many unsupported Haskell features:

- GADTs $\sim$ identify subset of dependent types
- pattern guards, views $\sim$ use with-matching
- 32-bit arithmetic $\sim$ first add to Agda itself
- Infinite data $\sim$ coinductive types
- Non-termination, general recursion $\sim$ partiality/general monad

Future work

Still many unsupported Haskell features:

- GADTs $\sim$ identify subset of dependent types
- pattern guards, views $\sim$ use with-matching
- 32-bit arithmetic $\sim$ first add to Agda itself
- Infinite data $\sim$ coinductive types
- Non-termination, general recursion $\sim$ partiality/general monad

Extra goodies:

- Generate **runtime checks** for decidable properties
- **QuickCheck** postulated properties
- HS2AGDA: inverse translation $\Rightarrow$ streamline porting of **existing** libraries

Future work

Still many unsupported Haskell features:

- GADTs $\sim$ identify subset of dependent types
- pattern guards, views $\sim$ use with-matching
- 32-bit arithmetic $\sim$ first add to Agda itself
- Infinite data $\sim$ coinductive types
- Non-termination, general recursion $\sim$ partiality/general monad

Extra goodies:

- Generate **runtime checks** for decidable properties
- **QuickCheck** postulated properties
- HS2AGDA: inverse translation $\Rightarrow$ streamline porting of **existing** libraries

More **applications** + **comparisons** with LiquidHaskell, `hs-to-coq`, etc..

Shameless plug https://wiki.portal.chalmers.se/agda/Main/AIMXXXI

AGDA2HS was developed during the last two **Agda Implementors' Meetings**

- biannual event where Agda users of all levels hack on Agda, its ecosystem, etc..

AIM XXXI in Edinburgh November 10-16, will include:

- talks
- coding sprints
- EuroProofNet day dedicated to the topic of large formal libraries
- a hike along the Scottish seaside 🥃

Shameless plug

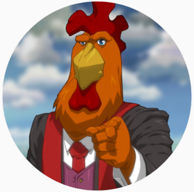
https://wiki.portal.chalmers.se/agda/Main/AIMXXXI

AGDA2HS was developed during the last two **Agda Implementors' Meetings**

- biannual event where Agda users of all levels hack on Agda, its ecosystem, etc..

AIM XXXI in Edinburgh November 10-16, will include:

- talks
- coding sprints
- EuroProofNet day dedicated to the topic of large formal libraries
- a hike along the Scottish seaside 🥃



Questions?