

# REASONABLE AGDA IS CORRECT HASKELL:

## WRITING VERIFIED HASKELL USING AGDA2HS

---

Jesper Cockx, [Orestis Melkonian](#), Lucas Escot, James Chapman, Ulf Norell



## SIMILAR SUBSTANCES

Coq'aine



## SIMILAR SUBSTANCES

Coq'aine



Lean



# MOTIVATION: ISSUES WITH CURRENT PROGRAM EXTRACTORS

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



```
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)
{-# COMPILER GHC insert #-}
```



```
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 readable code, but still does not readily support typeclasses:



```
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 0 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-like Agda (no need to cover the whole source language)
2. Verify your program using Agda's dependent types

1. Writing Haskell-like 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  $\_ \leq \_$  : Nat  $\rightarrow$  Tree  $\rightarrow$  Nat  $\rightarrow$  Set

$l \leq \text{Leaf} \quad \leq u = l \leq u$

$l \leq \text{Node } x \, t^l \, t^r \leq u = (l \leq t^l \leq x) \times (x \leq t^r \leq u)$

@0 insert-correct :  $\forall \{t \, x \, l \, u\} \rightarrow l \leq t \leq u$

$\rightarrow l \leq x \rightarrow x \leq u \rightarrow l \leq \text{insert } x \, t \leq u$

insert-correct {Leaf}  $\_ \, l \leq x \, x \leq u = l \leq x, x \leq u$

insert-correct {Node  $y \, t^l \, t^r$ } { $x$ } ( $IH^l, IH^r$ )  $l \leq x \, x \leq u$

with compare  $x \, y$

... | LT  $x \leq y = \text{insert-correct } IH^l \, l \leq x \, x \leq y, IH^r$

... | EQ refl =  $IH^l, IH^r$

... | GT  $y \leq x = IH^l, \text{insert-correct } IH^r \, y \leq x \, x \leq u$

## 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)
{-# 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)
```

- Export lowercase type variables to feel like home (i.e. **variable**  $a\ b\ c\ \dots$  : **Set**):

**id** :  $a \rightarrow a$

**id**  $x = x$

# PRIMITIVES

- Export lowercase type variables to feel like home (i.e. `variable a b c ... : Set`):

`id : a → a`

`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 (i.e. `variable a b c ... : Set`):

`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`

# PRIMITIVES

- Export lowercase type variables to feel like home (i.e. `variable a b c ... : Set`):

`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.

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

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



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

```
head : (xs : List a) { @0 _ : NonEmpty xs } → a
```

```
head (x :: _) = x
```

```
head [] {p} = error i "empty list"
```

```
  where @0 i : ⊥
```

```
        i = case p of λ ()
```

```
{-# COMPILE AGDA2HS head #-}
```



```
head :: [a] -> a
```

```
head (x : _) = x
```

```
head [] = error "empty list"
```



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



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

```
head : (xs : List a) { @0 _ : NonEmpty xs } → a
```

```
head (x :: _) = x
```

```
head [] {p} = error i "empty list"
```

```
  where @0 i : ⊥
```

```
        i = case p of λ ()
```

```
{-# COMPILE AGDA2HS head #-}
```



```
head :: [a] -> a
```

```
head (x : _) = x
```

```
head [] = error "empty list"
```

**Don't forget**

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

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



<i>class definitions</i>	~	<i>record types</i>
<i>instance declarations</i>	~	<i>record values</i>
<i>constraints</i>	~	<i>instance arguments</i>



```
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 #-}
```



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

## TYPECLASSES: INSTANCE DECLARATIONS $\sim$ RECORD VALUES



instance

MonoidNat : Monoid Nat

MonoidNat =  $\lambda$  where

.mempty  $\rightarrow 0$

.mappend  $i\ j \rightarrow i + j$

.left-identity  $\rightarrow \dots$

.right-identity  $\rightarrow \dots$

.associativity  $\rightarrow \dots$

{-# COMPILE AGDA2HS MonoidNat #-}



instance Monoid Nat where

mempty = 0

mappend i j = i + j

## TYPECLASSES: CONSTRAINTS $\sim$ INSTANCE ARGUMENTS



```
sumMon : {{ Monoid a }}  $\rightarrow$  List a  $\rightarrow$  a
sumMon []      = mempty
sumMon (x :: xs) = mappend x (sumMon xs)
{-# COMPILER AGDA2HS sumMon #-}
```



```
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
```



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's Cardano blockchain

- currently the 8<sup>th</sup> largest by market cap
- smart contracts written in PLUTUS, based on System  $F_{\omega}^{\mu}$
- implemented in Haskell
- tested against Agda formalization



### IOG's Cardano blockchain

- currently the 8<sup>th</sup> largest by market cap
- smart contracts written in PLUTUS, based on System  $F_{\omega}^{\mu}$
- implemented in Haskell
- tested against Agda formalization



# IOG USE CASE: TYPE RENAMING & SUBSTITUTION



```
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'
```

`ren` is a *functorial map* on `Type`.

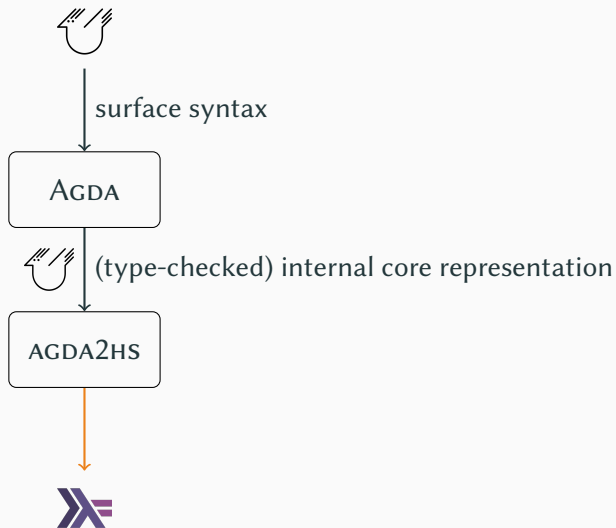
- `ren-id`:  $(ty : \text{Type } n) \rightarrow \text{ren id } ty \equiv ty$
- `ren-comp`:  $(ty : \text{Type } n) (\rho : n \rightarrow n') (\rho' : n' \rightarrow n'') \rightarrow \text{ren } (\rho' \circ \rho) ty \equiv \text{ren } \rho' (\text{ren } \rho ty)$

`ren` is a *functorial map* on `Type`.

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

`sub` is a *monadic bind* on `Type`.

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



## IMPLEMENTATION: **WHERE** CLAUSES

Surface 

$f : \text{Nat} \rightarrow \text{Nat}$

$f\ x = \text{go}$

**where**

$\text{go} : \text{Nat}$

$\text{go} = \text{TODO}$

-- may use x

Intermediate 

$\text{go} : \text{Nat} \rightarrow \text{Nat}$

$\text{go}\ x = \text{TODO}$

$f : \text{Nat} \rightarrow \text{Nat}$

$f\ x = \text{go}\ x$

Output 

$f :: \text{Natural} \rightarrow \text{Natural}$

$f\ x = \text{go}$

**where**

$\text{go} :: \text{Natural}$


$\text{go} = \text{TODO}$

Is our translation **sound**?

1. Agda that typechecks produces valid Haskell
2. Translation preserves behaviour/semantics

Is our translation **sound**?

1. Agda that typechecks produces valid Haskell
2. Translation preserves behaviour/semantics

No formal proof 



Is our translation **sound**?

1. Agda that typechecks produces valid Haskell
2. Translation preserves behaviour/semantics

No formal proof  ...yet!

c.f. Cockx's NWO grant: *A Trustworthy and Extensible Core Language for Agda*

Is our translation **sound**?

1. Agda that typechecks produces valid Haskell
2. Translation preserves behaviour/semantics

No formal proof 🤔 ...yet!

c.f. Cockx's NWO grant: *A Trustworthy and Extensible Core Language for Agda*

- Trust the ported Prelude and defined primitives
- Ensure all dependent types appear under *erased* positions
  - enforced by the AGDA2HS backend
- Translation of each language construct has equivalent behaviour
  - most cases blindingly obvious

NB: total functions + strong normalisation  $\Rightarrow$  evaluation order doesn't matter

Still many unsupported Haskell features:

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

Still many unsupported Haskell features:

- GADTs  $\sim$  covered by dependent types  $\rightarrow$  identify subset
- pattern guards, views  $\sim$  use with-matching
- 32-bit arithmetic  $\sim$  first add to Agda itself
- Monadic code  $\sim$  still need to reverse desugaring
- Infinite data  $\sim$  coinductive types
- Non-termination, general recursion  $\sim$  partiality/general monad

Still many unsupported Haskell features:

- GADTs ~ covered by dependent types → identify subset
- pattern guards, views ~ use with-matching
- 32-bit arithmetic ~ first add to Agda itself
- Monadic code ~ still need to reverse desugaring
- Infinite data ~ coinductive types
- Non-termination, general recursion ~ partiality/general monad

Extra goodies:

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

Still many unsupported Haskell features:

- GADTs ~ covered by dependent types → identify subset
- pattern guards, views ~ use with-matching
- 32-bit arithmetic ~ first add to Agda itself
- Monadic code ~ still need to reverse desugaring
- Infinite data ~ coinductive types
- Non-termination, general recursion ~ partiality/general monad

Extra goodies:

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

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

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
- code sprints
- EuroProofNet day dedicated to the topic of large formal libraries
- a hike along the Scottish seaside 🍷





*‘ When the limestone of imperative programming  
is worn away, the granite of functional programming  
will be revealed underneath. ’ —Simon Peyton Jones*





~~*‘When the limestone of imperative programming is worn away, the granite of functional programming will be revealed underneath.’ —Simon Peyton Jones*~~

*‘When the doors of perception are cleansed, every thing would appear to man as it is, dependent.’*

*—Agdus Huxley, paraphrasing William Blake*