Work-in-progress: "Verifying" the Glasgow Haskell Compiler Core language

Stephanie Weirich

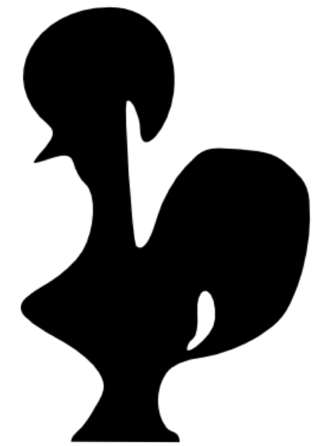
Joachim Breitner, Antal Spector-Zabusky, Yao Li,
Christine Rizkallah, John Wiegley

June 2018

Let's prove GHC correct

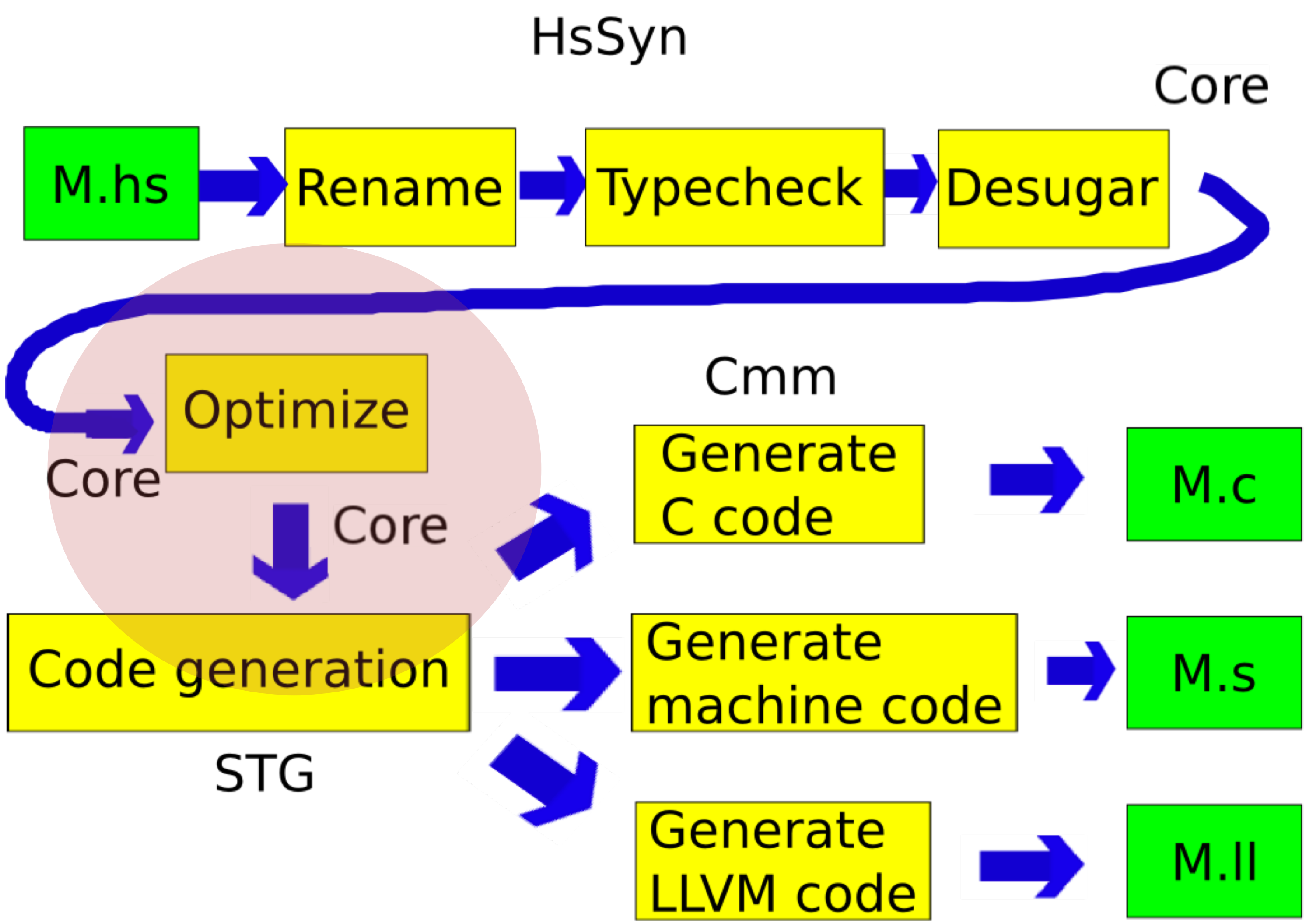
What would it take?

What would it take?



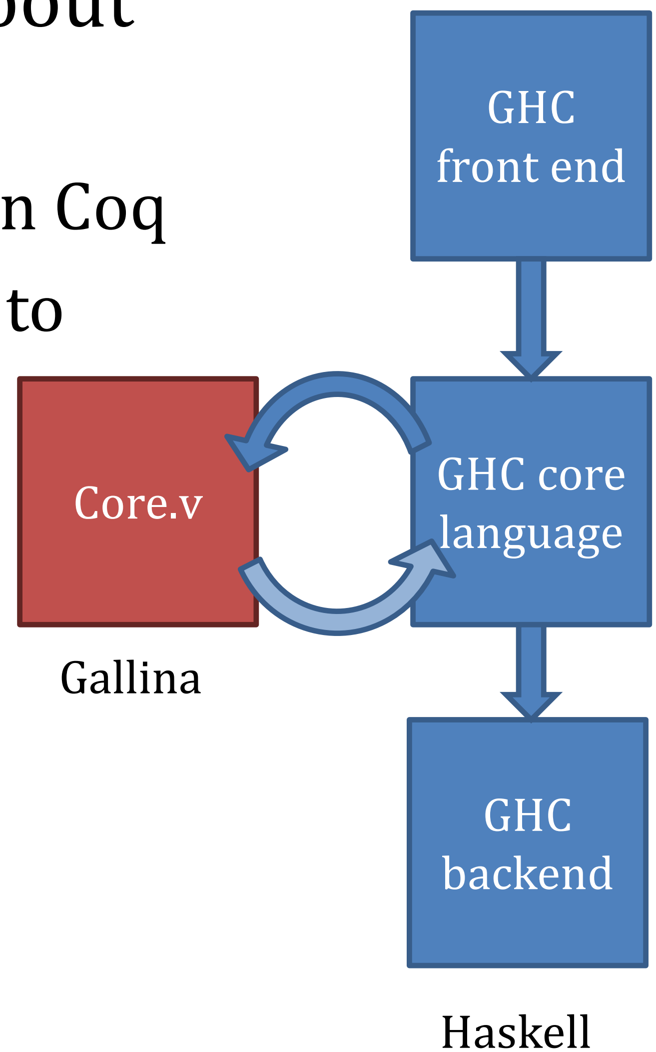
- A proof assistant
  - not doing this by hand
- A formal specification of Haskell, to define what correct means
  - That's really big and we don't have one. Maybe we can start with something smaller? GHC Core?
- A formal specification of Haskell, to prove that the *Haskell program GHC* is correct
  - That's really big and we don't have one. Maybe we can use something else?
- A lot of work
  - Maybe there is benefit to verifying only part of it, but which?

GHC Core language



Gallina is Haskell if you squint

- Want to use Coq to reason about GHC
  - Need a semantics for Haskell in Coq
  - But that is what we are trying to build!
- "Easy" approach: shallow embedding
  - Use Gallina as a stand-in for Haskell
  - Translate Haskell functions to Gallina functions, use that as semantics



Core Spec

hs-to-coq

A tool for translating Haskell code to equivalent Gallina definitions via shallow embedding [CPP' 18]

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr k z = go
          where
            go []      = z
            go (y:ys) = y `k` go ys
```

```
Definition foldr {a} {b} :
               (a -> b -> b) -> b -> list a -> b :=
  fun k z =>
    let fix go arg_0__
              := match arg_0__ with
                 | nil => z
                 | cons y ys => k y (go ys)
                 end in
    go.
```

Questions about hs-to-coq approach

1. Is there enough Haskell code out there that we can translate to make this approach worthwhile?

2. Even if we can find code to translate, is the result suitable for verification?

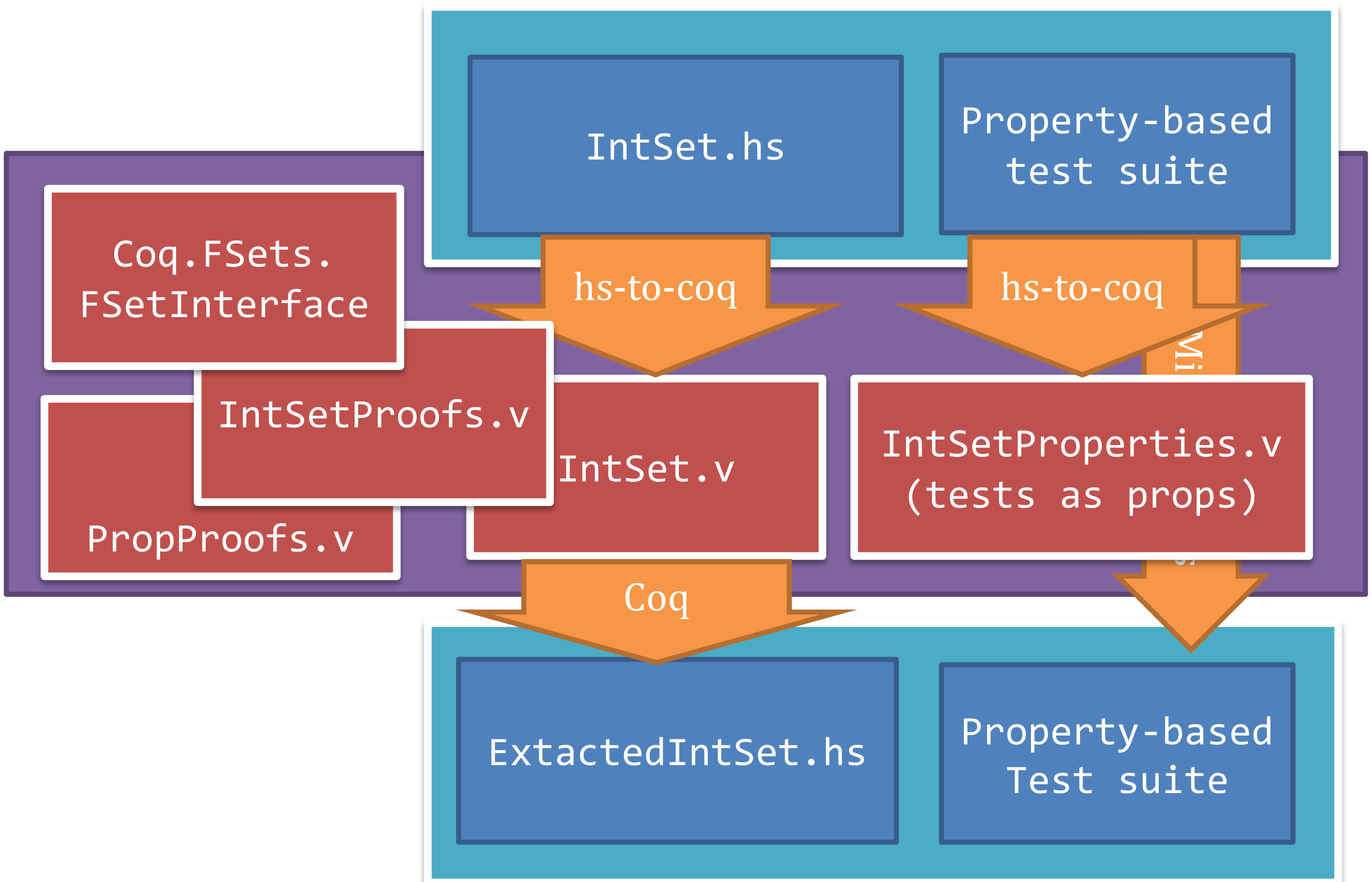
3. Even if we can do the proofs, do they mean anything about the Haskell source?

Case study: containers

- Popular Haskell libraries: Data.Set and Data.IntSet
- Used by GHC Core language implementation
- What did we prove?
  - Invariants in the source file comments (ensures the balance properties)
  - Mathematical specification (both our own and FSetInterface)
  - Quickcheck properties interpreted as theorems
  - GHC Rewrite rules

Ready, Set, Verify! Applying hs-to-coq to real-world Haskell code (Experience report)
Joachim Breitner, Antal Spector-Zabusky, Yao Li, Christine Rizkallah, John Wiegley, Stephanie Weirich
ICFP 2018

Containers case study



What did we learn?

1. We can translate these libraries*
2. We can prove what we want to prove**
3. Output is semantically equivalent (as far as we can tell by testing)
4. Haskell code is correct ☺

*Need to address partiality

**We "edit" the code during translation in support of verification

Partiality: Unsound

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

```
Axiom error : forall {a} , String -> a.

Definition head {a}
    (xs : list a) : a :=
  match xs with
  | (x::_) => x
  | _       => error "hea   mpty
  end.
```

Partiality: Annoying

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

```
Inductive Partial (a:Type) :=
  | return : a -> Partial a
  | error  : String -> Partial a
  | …

Definition head {a} (xs : list a) : Partial a :=
  match xs with
  | (x::_) => return x
  | _      => error "head: empty list"
  end.
```

Partiality: Pragmatic approach

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

```
Definition error : forall {a} `{Default a},
                      String -> a := default.
Definition head {a} `{Default a}
    (xs : list a) : a :=
  match xs with
  | (x::_) => x
  | _      => error "head: empty list"
  end.
```

☞ "default" is an opaque definition so proofs must work for any value of the appropriate type.

Partiality: Pragmatic approach

- Can also use this approach for difficult termination arguments (with classical logic/axiom of choice)

```
Definition deferredFix:
   forall {a r} `{Default r},
       ((a -> r) -> (a -> r)) -> (a -> r).

Definition deferredFix_eq_on:
   forall {a b} `{Default b}
      (f : (a -> b) -> (a -> b))
      (P : a -> Prop) (R : a -> a -> Prop),
   well_founded R -> recurses_on P R f ->
   forall x, P x ->
   deferredFix f x = f (deferredFix f) x.
```

A Formalization Gap is a *good* thing

- Machine integers are fixed width. Do we want to reason about overflow?
- No!
  - In Data.Set, Ints track size of tree for balance
  - GHC uses Data.IntSet to generate unique names
  - Both cases will run out of memory before overflow
- Control translation with hs-to-coq rewrites
  - type GHC.Num.Int = Coq.ZArith.BinNum.Z
  - Formalization gap is explicit & recorded

A Formalization Gap is a *good* thing

- Machine integers store positive and negative numbers. Do we want that?
- No!
  - In Data.Set, Ints track size of tree for balance
  - GHC uses Data.IntSet to generate unique names
  - Both cases never need to store negative numbers
- Control translation with hs-to-coq rewrites
  - type GHC.Num.Int = Coq.NArith.BinNat.N
  - (But, need *partial* implementation of subtraction)
  - Formalization gap is explicit & recorded

What about GHC?

Core Spec

Questions about GHC

1. Is there enough code *in GHC* that we can translate to make this approach worthwhile?

2. Even if we can find code to translate, is the result suitable for verification?

3. Even if we can do the proofs, do they mean anything about the GHC implementation? (Note: Core plug-in option available)

GHC: Current status

- Base libraries (9k loc)
  - 45 separate modules
  - Some written by-hand: GHC.Prim, GHC.Num, GHC.Tuple
  - Most translated: GHC.Base, Data.List, Data.Foldable, Control.Monad, etc.
- Containers (6k loc)
  - Translated & (mostly) verified: 4 modules
  - (Data.Set, Data.Map, Data.IntSet, Data.IntMap)
- GHC, version 8.4.1 (19k loc)
  - 55 modules so far (327 modules total in GHC, but we won't need them all)
  - hs-to-coq edits (2k LOC)
- *First verification goal*: Exitify compiler pass

Core AST

```haskell
data Expr b
  = Var  Id
  | Lit  Literal
  | App  (Expr b) (Arg b)
  | Lam  b (Expr b)
  | Let  (Bind b) (Expr b)
  | Case (Expr b) b Type
         [Alt b]
  | Cast (Expr b) Coercion
  | Tick (Tickish Id) (Expr b)
  | Type Type
  | Coercion Coercion
   deriving Data
data Bind b =
   NonRec b (Expr b)
 | Rec [(b, (Expr b))]
  deriving Data
```

```coq
Inductive Expr b : Type
  := Mk_Var : Id -> Expr b
   | Lit : Literal -> Expr b
   | App :
       Expr b -> Arg b -> Expr b
   | Lam : b -> Expr b -> Expr b
   | Let :
      Bind b -> Expr b -> Expr b
   | Case : Expr b -> b -> unit
       -> list (Alt b) -> Expr b
   | Cast :
       Expr b -> unit -> Expr b
   | Tick : Tickish Id
       -> Expr b -> Expr b
   | Type_ : unit -> Expr b
   | Coercion : unit -> Expr b
with Bind b : Type
  := NonRec : b -> Expr b
                 -> Bind b
   |  Rec : list (b * (Expr b))
                -> Bind b
```

Core Optimization : Exitify

```
-- | Given a recursive group of a joinrec, identifies
--   “exit paths” and binds them as
--   join-points outside the joinrec.

exitify :: InScopeSet -> [(Var,CoreExpr)] ->
           (CoreExpr -> CoreExpr)
exitify in_scope pairs =
    \body -> mkExitLets exits (mkLetRec pairs' body)
  where
     pairs' = …   // updated recursive group
     exits  = …   // exit paths

-- 215 LOC, incl comments
```

- Requires moving code from one binding scope to another
- First proof: show that well-scoped terms stay well-scoped

Bug found!

- Exitify does not always produced well-scoped code
  - Missed by GHC test suite
  - (Perhaps not exploitable at source level)
- Fixed in GHC HEAD
  - Proofs updated to new version
- What is the general workflow?
  - Always work on HEAD? Maintain separate branch?
  - Axiomatize failing lemma?
  - Fix code via hs-to-coq edits?

Conclusion & More questions

Let's take advantage of the semantic similarity of Haskell and Gallina for developing verified compilers

- "Formalization gap" is pragmatic
- How far can we push this approach?
- Can we make it easier to verify just a part of a large system?
- Can we get good performance of extracted code? (And plug back into GHC?)
- Can we say anything about linking with nonverified code?

Back up slides…

What would we get from verifying GHC?

- A better GHC…
  - Maybe we would find some bugs
  - Maybe we would make the implementation simpler/clearer/easier to reason about
- A semantics for Core
  - First steps towards a definition of Haskell
- A better understanding about the interaction between pure higher-order functional programming and verification

Why not use CoInductive?

- Another formalization gap
  - Haskell datatypes are co-inductive by default
- But inductive reasoning is useful for compilers and languages
  - Termination of functions depends on decreasing size of data structure
- This is an example of an invariant about the core language
  - We assume it never needs to work with infinite terms, and prove that it never generates infinite terms
  - Never going to create an AST term with an "infinite" number of lambda expressions

What's in GHC Core?

- Additional general purpose libraries
  - Bag, State, Maybes, Pair, FiniteMap, OrdList, MonadUtils, BooleanFormula, …
- Compiler-specific utilities
  - SrcLoc, Module, DynFlags, Constants,
  - Unique, UniqSupply, UniqSet, UniqFM, …
- Core AST representation
  - IdInfo, Var, VarSet, VarEnv, Name, Id, Demand
  - Class, TyCon, DataCon, **CoreSyn**
- Core operations and optimization
  - CoreFVs, CoreSubst, CallArity, CoreArity,
  - **Exitify**

Exitify example

Example:

```
let t = foo bar
joinrec
    go 0     x y = t (x*x)
    go (n-1) x y = jump go (n-1) (x+y)
in …
```

We'd like to inline `t`, but that does not happen: Because t is a thunk and is used in a recursive function, doing so might lose sharing in general. In this case, however, `t` is on the _exit path_ of `go`, so called at most once.

Example result:

```
let t = foo bar
join exit x = t (x*x)
joinrec
    go 0     x y = jump exit x
    go (n-1) x y = jump go (n-1) (x+y)
in …
```

Now `t` is no longer in a recursive function, and good things happen!

What about Haskell?
We *already* have a compiler. Can we verify that in the process of constructing a language spec?

Core Spec