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[*]

September 26, 2018, ICFP, Saint Louis

[*]University of Pennsylvania, [†]DFINITY, [‡]University of New South Wales, [§]BAE Systems

Formal verification of real-world Haskell code is

possible!

but

Formal verification of real-world Haskell code is

possible!

but

Verification is

futile

because Haskell code has no bugs.

Verifying Haskell Code

```haskell
elem :: (Eq a) => a -> [a] -> Bool
elem _ []          = False
elem x (y:ys)      = x==y || elem x ys
```

Verifying Haskell Code

```
elem :: (Eq a) => a -> [a] -> Bool
elem _ []          = False
elem x (y:ys)      = x==y || elem x ys
```

```
Definition elem {a} `{Eq_ a} : a -> list a -> bool :=
  fix elem arg_0__ arg_1__
    := match arg_0__, arg_1__ with
       | _, nil => false
       | x, cons y ys => orb (x == y) (elem x ys)
       end.
```

Verifying Haskell Code

```haskell
elem :: (Eq a) => a -> [a] -> Bool
elem _ []          = False
elem x (y:ys)      = x==y || elem x ys
```

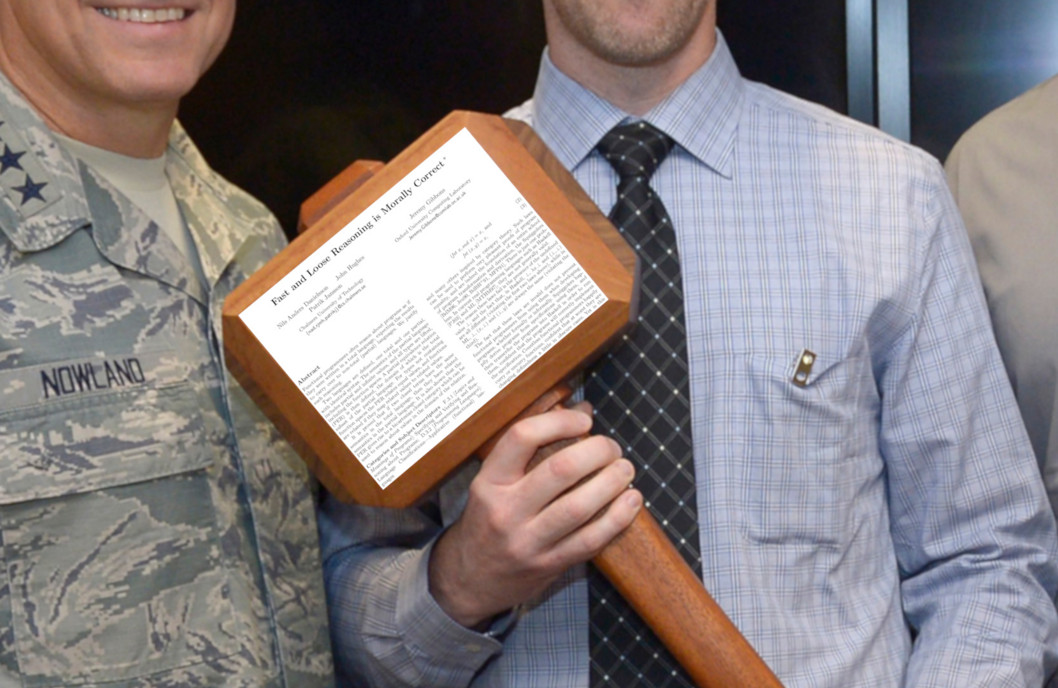
```coq
Definition elem {a} `{Eq_ a} : a -> list a -> bool :=
  fix elem arg_0__ arg_1__
    := match arg_0__, arg_1__ with
       | _, nil => false
       | x, cons y ys => orb (x == y) (elem x ys)
       end.
```

Verifying Haskell Code

```
elem :: (Eq a) => a -> [a] -> Bool
elem _ []         = False
elem x (y:ys)     = x==y || elem x ys
```

hs-to-coq [CPP'18]

```
Definition elem {a} `{Eq_ a} : a -> list a -> bool :=
  fix elem arg_0__ arg_1__
    := match arg_0__, arg_1__ with
       | _, nil => false
       | x, cons y ys => orb (x == y) (elem x ys)
       end.
```


NOWLAND
Fast and Loose Reasoning is Morally Correct *
Nils Anders Danielsson John Hughes
Patrik Jansson
Chalmers University of Technology
{nad,rjmh,patrikj}@cs.chalmers.se
Jeremy Gibbons
Oxford University Computing Laboratory
Jeremy.Gibbons@comlab.ox.ac.uk
Abstract
Categories and Subject Descriptors

Real-World Haskell Code

Real-World Haskell Code

The **`Data.Set`** data type:

```
data Set a = Bin !Size !a !(Set a) !(Set a)
           | Tip

type Size  = Int
```

The **`Data.IntSet`** data type:

```
data IntSet = Bin !Prefix !Mask !IntSet !IntSet
            | Tip !Prefix !BitMap
            | Nil

type Prefix = Int
type Mask   = Int
type BitMap = Word
type Key    = Int
```

Real-World Haskell Code

The

```
dat

typ
```

The

```
dat

             | Nil

type Prefix = Int
type Mask   = Int
type BitMap = Word
type Key    = Int
```

Depends on 4 packages:

array, base, deepseq, ghc-prim

Used by 4254 packages:

4Blocks, a50, abcBridge, abnf, abstract-deque, abstract-deque-tests, accelerate, accelerate-blas, accelerate-cuda, accelerate-examples, accelerate-fft, accelerate-fourier, accelerate-llvm, accelerate-llvm-native, accelerate-llvm-ptx, access-token-provider, acid-state, acid-state-dist, ActionKid, activehs, ad, adb, adblock2privoxy, adhoc-network, adict, adjunctions, ADPfusion, ADPfusionForest, ADPfusionSet, adp-multi, adp-multi-monadiccp, aern2-real, AERN-Basics, AERN-Net, AERN-RnToRm, AERN-RnToRm-Plot, aeson, aeson-bson, aeson-coerce, aeson-compat, **and many more**

Real-World Haskell Code

The

```
dat

typ
```

The

```
dat

type Pref
type Mask
type BitMa
type Key
```

Depends on 4 packages:

array, base, deepseq, ghc-prim

Used by 42

4Blocks, a5

accelerate-

llvm-native

ad, adb, adb

ADPfusionS

AERN-RnTo

The Performance of the Haskell CONTAINERS Package

Milan Straka

Department of Applied Mathematics
Charles University in Prague, Czech Republic
fox@ucw.cz

Abstract

In this paper, we perform a thorough performance analysis of the CONTAINERS package, the de facto standard Haskell containers library, comparing it to the most of existing alternatives on HackageDB. We then significantly improve its performance, making it comparable to the best implementations available. Additionally, we describe a new persistent data structure based on hashing, which offers the best performance out of available data structures containing `Strings` and `ByteStrings`.

Categories and Subject Descriptors D.2.8 [*Software Engineering*]: Metrics—Performance measures; E.1 [*Data Structures*]: Trees; Lists, stacks, and queues

General Terms Algorithms, Measurement, Performance

- ordered *sequences* of any elements,
- *trees* and *graphs*.

All data structures in this package work persistently, ie. they can be shared [Driscoll et al. 1989].

Our decision to compare and improve the CONTAINERS package was motivated not only by the wide accessibility of the package, but also by our intention to replace the GHC internal data structures with the CONTAINERS package. Therefore we wanted to confirm that the performance offered by the package is the best possible, both for small and big volumes of data stored in the structure, and possibly to improve it.

The contributions of this paper are as follows:

- We present the first comprehensive performance measurements

Finding specifications

Source 1/5: Invariants

```
data IntSet = Bin {-# UNPACK #-} !Prefix {-# UNPACK #-} !Mask !IntSet !IntSet
-- Invariant: Nil is never found as a child of Bin.
-- Invariant: The Mask is a power of 2.  It is the largest bit position at which
--            two elements of the set differ.
-- Invariant: Prefix is the common high-order bits that all elements share to
--            the left of the Mask bit.
-- Invariant: In Bin prefix mask left right, left consists of the elements that
--            don't have the mask bit set; right is all the elements that do.
            | Tip {-# UNPACK #-} !Prefix {-# UNPACK #-} !BitMap
-- Invariant: The Prefix is zero for the last 5 (on 32 bit arches) or 6 bits
--            (on 64 bit arches). The values of the set represented by a tip
--            are the prefix plus the indices of the set bits in the bit map.
            | Nil
```

Source 2/5: QuickCheck Properties

```haskell
prop_UnionAssoc :: IntSet -> IntSet -> IntSet -> Bool
prop_UnionAssoc t1 t2 t3
  = union t1 (union t2 t3) == union (union t1 t2) t3
```

Source 2/5: QuickCheck Properties

```
prop_UnionAssoc :: IntSet -> IntSet -> IntSet -> Bool
prop_UnionAssoc t1 t2 t3
  = union t1 (union t2 t3) == union (union t1 t2) t3
```

```
Theorem thm_UnionAssoc : toProp prop_UnionAssoc.
Proof.
  rewrite /prop_UnionAssoc /= => s1 WF1 s2 WF2 s3 WF3.

  move: (union_WF _ _ WF1  WF2)  => WF12.
  move: (union_WF _ _ WF2  WF3)  => WF23.
  move: (union_WF _ _ WF12 WF3)  => WF123.
  move: (union_WF _ _ WF1  WF23) => WF123'.

  apply/eqIntSetMemberP => // k.
  by rewrite !union_member // orbA.
```

Source 3/5: Rewrite Rules

```
{-# RULES "IntSet.toAscList" [~1] forall s . toAscList s = build (\c n -> foldrFB c n s) #-}
{-# RULES "IntSet.toAscListBack" [1] foldrFB (:) [] = toAscList #-}
{-# RULES "IntSet.toDescList" [~1] forall s . toDescList s = build (\c n -> foldlFB (\xs x -
{-# RULES "IntSet.toDescListBack" [1] foldlFB (\xs x -> x : xs) [] = toDescList #-}
```

Source 4/5: Type Class Laws

```
class Semigroup a => Monoid a where                # Source
```

The class of monoids (types with an associative binary operation that has an identity). Instances should satisfy the following laws:

- `x <> mempty = x`
- `mempty <> x = x`
- `x <> (y <> z) = (x <> y) <> z` (Semigroup law)
- `mconcat = foldr '(<>)' mempty`

Source 4/5: Type Class Laws

```
class Semigroup a => Monoid a where                          # Source
```

The class of monoids (types with an associative binary operation that has an identity). Instances should satisfy the following laws:

- `x <> mempty = x`
- `mempty <> x = x`
- `x <> (y <> z) = (x <> y) <> z` (Semigroup law)

```
Class MonoidLaws (t : Type) `{ Monoid t } `{SemigroupLaws t} `{ EqLaws t } :=
  { monoid_left_id   : forall x, (mappend mempty x == x) = true;
    monoid_right_id  : forall x, (mappend x mempty == x) = true;
    monoid_semigroup : forall x y, (mappend x y == (x <<>> y)) = true;
    monoid_mconcat   : forall x, (mconcat x == foldr mappend mempty x) = true
  }.
```

Source 5/5: External specifications

```
Module Type
 WSfun =
 Funsig (E0:DecidableType)
   Sig
     Definition elt : Type.
     Parameter t : Type.
     Parameter In : elt -> t -> Prop.
     Definition Equal : t -> t -> Prop.
     Definition Subset : t -> t -> Prop.
     Definition Empty : t -> Prop.
     Definition For_all : (elt -> Prop) -> t -> Prop.
     Definition Exists : (elt -> Prop) -> t -> Prop.
     Parameter empty : t.
     Parameter is_empty : t -> bool.
     Parameter mem : elt -> t -> bool.
     Parameter add : elt -> t -> t.
     Parameter singleton : elt -> t.
     Parameter remove : elt -> t -> t.
     Parameter union : t -> t -> t.
     Parameter inter : t -> t -> t
```

Source 5/5: External specifications

```
Module Type
 WSfun =
 Funsig (E0:DecidableType)
   Sig
     Definition elt : Type.
     Parameter t : Type.
     Parameter In : elt -> t -> Prop.
     Definition Equal : t -> t -> Prop.
     Definition Subset : t -> t -> Prop.
     Definition Empty : t -> Prop.
     Definition For_all : (elt -> Prop) -> t -> Prop.
     Definition Exists : (elt -> Prop) -> t -> Prop.
```

```
     Parameter singleton_1 :
       forall x y : elt, In y (singleton x) -> E0.eq x y.
     Parameter singleton_2 :
       forall x y : elt, E0.eq x y -> In y (singleton x).
     Parameter union_1 :
       forall (s s' : t) (x : elt), In x (union s s') -> In x s \/ In x s'.
     Parameter union_2 :
       forall (s s' : t) (x : elt), In x s -> In x (union s s').
     Parameter union_3 :
       forall (s s' : t) (x : elt), In x s' -> In x (union s s').
```

Translating Haskell

(when Coq doesn't like it)

Edits

```
elem :: (Eq a) => a -> [a] -> Bool
elem _ []          = False
elem x (y:ys)      = x==y || elem x ys
```

hs-to-coq [CPP'18]

```
Definition elem {a} `{Eq_ a} : a -> list a -> Bool :=
  fix elem arg_0__ arg_1__
    := match arg_0__, arg_1__ with
         | _, nil => False
         | x, cons y ys => (x == y) || elem x ys
         end.
```

Edits

```
elem :: (Eq a) => a -> [a] -> Bool
elem _ []         = False
elem x (y:ys)     = x==y || elem x ys
```

hs-to-coq [CPP'18]

```
Definition elem {a} `{Eq_ a} : a -> list a -> Bool :=
  fix elem arg_0__ arg_1__
    := match arg_0__, arg_1__ with
       | _, nil => False
       | x, cons y ys => (x == y) || elem x ys
       end.
```

Edits

```
elem :: (Eq a) => a -> [a] -> Bool
elem _ []          = False
elem x (y:ys)      = x==y || elem x ys
```

hs-to-coq [CPP'18]

```
Definition elem {a} `{Eq_ a} : a -> list a -> bool :=
  fix elem arg_0__ arg_1__
    := match arg_0__, arg_1__ with
        | _, nil => false
        | x, cons y ys => orb (x == y) (elem x ys)
        end.
```

Edits

```
elem :: (Eq a) => a -> [a] -> Bool
                              lse
                             =y || elem x ys
```

```
rename type  GHC.Types.Bool   = bool
rename value GHC.Types.True   = true
rename value GHC.Types.False  = false
rename value GHC.Classes.not  = negb
rename value GHC.Classes.||   = orb
rename value GHC.Classes.&&   = andb
```

hs-to-coq [CPP'18]

```
Definition elem {a} `{Eq_ a} : a -> list a -> bool :=
  fix elem arg_0__ arg_1__
    := match arg_0__, arg_1__ with
         | _, nil => false
         | x, cons y ys => orb (x == y) (elem x ys)
         end.
```

Partiality

```
-- balanceR is called when right subtree might have been inserted to or when
-- left subtree might have been deleted from.
balanceR :: a -> Set a -> Set a -> Set a
balanceR x l r = case l of
  Tip -> case r of
           Tip -> Bin 1 x Tip Tip
           (Bin _ _ Tip Tip) -> Bin 2 x Tip r
           (Bin _ rx Tip rr@(Bin _ _ _ _)) -> Bin 3 rx (Bin 1 x Tip Tip) rr
           (Bin _ rx (Bin _ rlx _ _) Tip) -> Bin 3 rlx (Bin 1 x Tip Tip) (Bin 1 rx Tip Tip)
           (Bin rs rx rl@(Bin rls rlx rll rlr) rr@(Bin rrs _ _ _))
             | rls < ratio*rrs -> Bin (1+rs) rx (Bin (1+rls) x Tip rl) rr
             | otherwise -> Bin (1+rs) rlx (Bin (1+size rll) x Tip rll) (Bin (1+rrs+size rlr) rx rlr rr)

  (Bin ls _ _ _) -> case r of
           Tip -> Bin (1+ls) x l Tip

           (Bin rs rx rl rr)
              | rs > delta*ls  -> case (rl, rr) of
                   (Bin rls rlx rll rlr, Bin rrs _ _ _)
                     | rls < ratio*rrs -> Bin (1+ls+rs) rx (Bin (1+ls+rls) x l rl) rr
                     | otherwise -> Bin (1+ls+rs) rlx (Bin (1+ls+size rll) x l rll) (Bin (1+rrs+size rlr) rx rlr rr)
                   (_, _) -> error "Failure in Data.Map.balanceR"
              | otherwise -> Bin (1+ls+rs) x l r
```

Partiality

In Haskell

```
error :: String -> a
```

Partiality

In Haskell

```
error :: String -> a
```

In Coq

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

Partiality

In Haskell

```
error :: String -> a
```

In Coq

```
Class Default (a : Type) := { default : a }.
Definition error {a} `{Default a} : String -> a
  := fun _ => default.
```

Partiality

In Haskell

```
error :: String -> a
```

In Coq

```
Class Default (a : Type) := { default : a }.
Definition error {a} `{Default a} : String -> a.
Proof. exact (fun _ => default). Qed.
```

Deferred termination

```
fromDistinctAscList :: [a] -> Set a
fromDistinctAscList [] = Tip
fromDistinctAscList (x0 : xs0) = go (1::Int) (Bin 1 x0 Tip Tip) xs0
  where
    go !_ t [] = t
    go s l (x : xs) = case create s xs of
                        (r :*: ys) -> let !t' = link x l r
                                      in go (s `shiftL` 1) t' ys

    create !_ [] = (Tip :*: [])
    create s xs@(x : xs')
      | s == 1 = (Bin 1 x Tip Tip :*: xs')
      | otherwise = case create (s `shiftR` 1) xs of
                      res@(_ :*: []) -> res
                      (l :*: (y:ys)) -> case create (s `shiftR` 1) ys of
                        (r :*: zs) -> (link y l r :*: zs)
```

Deferred termination

To define recursive functions:

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

Deferred termination

To define recursive functions:

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

To verify recursive functions:

```
Definition recurses_on {a r}
  (P : a -> Prop) (R : a -> a -> Prop) (f : (a -> r) -> (a -> r))
  := forall g h x,
     P x ->
     (forall y, P y -> R y x -> g y = h y) ->
     f g x = f h x.

Axiom deferredFix_eq_on: forall {a r} ‘{Default r} f P R,
  well_founded R -> recurses_on P R f ->
  forall x, P x -> deferredFix f x = f (deferredFix f) x.
```

Really unsafe pointer equality

In Haskell

```
Prelude> :set -XMagicHash
Prelude> :info GHC.Exts.reallyUnsafePtrEquality#
GHC.Prim.reallyUnsafePtrEquality# :: a -> a -> GHC.Prim.Int#
        -- Defined in ‘GHC.Prim’
```

Really unsafe pointer equality

In Haskell

```
Prelude> :set -XMagicHash
Prelude> :info GHC.Exts.reallyUnsafePtrEquality#
GHC.Prim.reallyUnsafePtrEquality# :: a -> a -> GHC.Prim.Int#
        -- Defined in ‘GHC.Prim’
```

In Coq

```
Definition ptrEq : forall {a}, a -> a -> bool
  := fun _ _ => false.
```

Really unsafe pointer equality

In Haskell

```
Prelude> :set -XMagicHash
Prelude> :info GHC.Exts.reallyUnsafePtrEquality#
GHC.Prim.reallyUnsafePtrEquality# :: a -> a -> GHC.Prim.Int#
        -- Defined in ‘GHC.Prim’
```

In Coq

```
Definition ptrEq : forall {a}, a -> a -> bool.
```

Really unsafe pointer equality

In Haskell

```
Prelude> :set -XMagicHash
Prelude> :info GHC.Exts.reallyUnsafePtrEquality#
GHC.Prim.reallyUnsafePtrEquality# :: a -> a -> GHC.Prim.Int#
        -- Defined in ‘GHC.Prim’
```

In Coq

```
Definition ptrEq : forall {a}, a -> a -> bool.

Lemma ptrEq_eq :
  forall {a} (x : a) (y : a),
  ptrEq x y = true ->
  x = y.
```

15 744 lines of Coq proofs later…

- 0 bugs found
- The proofs are maintainable
- Insights on the theory of weight-balanced binary search trees
- `Data.Set.union` is now a bit faster

Formal verification of real-world Haskell code is

possible!

Ongoing work

- GHC
  Antal Spector-Zabusky et. al., University of Pennsylvania
  https://github.com/antalsz/hs-to-coq/tree/master/examples/ghc/
- RISC V semantics
  Samuel Gruetter, MIT
  https://github.com/mit-plv/riscv-coq
- RISC V semantics (again!)
  Rishiyur S. Nikhil, Bluespec Inc.
  https://github.com/rsnikhil/RISCV-ISA-Spec/tree/master/verification
- Documentation!
  https://hs-to-coq.readthedocs.io/

Thank you!

Backup slides

Verified functions

Set: `delete`, `deleteMax`, `deleteMin`, `difference`, `disjoint`, `drop`, `elems`, `empty`, `filter`, `foldl`, `foldl'`, `foldr`, `foldr'`, `fromAscList`, `fromDescList`, `fromDistinctAscList`, `fromDistinctDescList`, `fromList`, `insert`, `intersection`, `isSubsetOf`, `lookupMax`, `lookupMin`, `map`, `mapMonotonic`, `maxView`, `member`, `minView`, `notMember`, `null`, `partition`, `singleton`, `size`, `split`, `splitAt`, `splitMember`, `take`, `toAscList`, `toDescList`, `toList`, `union`, `unions`

Instances: `Eq`, `Eq1`, `Monoid`, `Ord`, `Ord1`, `Semigroup`

Internal functions: `balanceL`, `balanceR`, `combineEq`, `glue`, `insertMax`, `insertMin`, `insertR`, `link`, `maxViewSure`, `merge`, `minViewSure`, `valid`

IntSet: `delete`, `difference`, `disjoint`, `elems`, `empty`, `filter`, `foldl`, `foldr`, `fromList`, `insert`, `intersection`, `isProperSubsetOf`, `isSubsetOf`, `map`, `member`, `notMember`, `null`, `partition`, `singleton`, `size`, `split`, `splitMember`, `toAscList`, `toDescList`, `toList`, `union`, `unions`

Instances: `Eq`, `Monoid`, `Ord`, `Semigroup`

Internal functions: `branchMask`, `equal`, `highestBitMask`, `mask`, `nequal`, `nomatch`, `revNat`, `shorter`, `valid`, `zero`

Untranslated functions

Set: `deleteAt`, `deleteFindMax`, `deleteFindMin`, `elemAt`, `findIndex`, `findMax`, `findMin`

Instances: `Data`, `IsList`, `NFData`, `Read`, `Show`, `Show1`

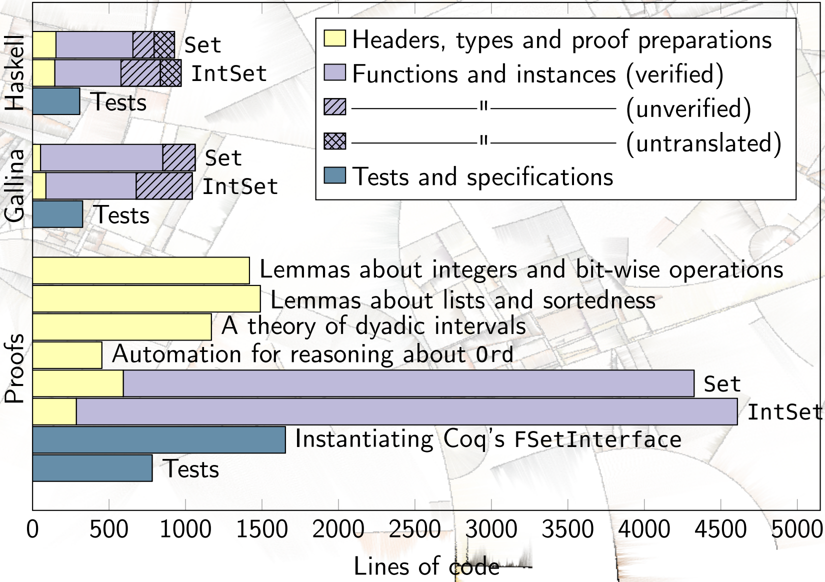
Internal functions: `showTree`

IntSet: `deleteFindMax`, `deleteFindMin`, `findMax`, `findMin`, `fromAscList`, `fromDistinctAscList`

Instances: `Data`, `IsList`, `NFData`, `Read`, `Show`

Internal functions: `showTree`

Stats



Haskell
Set
IntSet
Tests
Gallina
Set
IntSet
Tests
Proofs
Lemmas about integers and bit-wise operations
Lemmas about lists and sortedness
A theory of dyadic intervals
Automation for reasoning about Ord
Set
IntSet
Instantiating Coq's FSetInterface
Tests
Headers, types and proof preparations
Functions and instances (verified)
———"——— (unverified)
———"——— (untranslated)
Tests and specifications
0
500
1000
1500
2000
2500
3000
3500
4000
4500
5000
Lines of code