Ready, Set, Verify!

Applying hs-to-coq to Real-World Haskell Code
(Experience Report)

Joachim Breitner\textsuperscript{*→†} Antal Spector-Zabusky\textsuperscript{*} Yao Li\textsuperscript{*}
Christine Rizkallah\textsuperscript{*→‡} John Wiegley\textsuperscript{§→†} Stephanie Weirich\textsuperscript{*}

September 26, 2018, ICFP, Saint Louis

\textsuperscript{*}University of Pennsylvania, \textsuperscript{†}DFINITY, \textsuperscript{‡}University of New South Wales, \textsuperscript{§}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

\[
\begin{align*}
\text{elem} :: (\text{Eq } a) & \Rightarrow a \rightarrow [a] \rightarrow \text{Bool} \\
\text{elem } \_ \ [\] & = \text{False} \\
\text{elem } x \ (y:ys) & = x == y \ || \ \text{elem } x \ ys
\end{align*}
\]
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

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

\[
\text{elem} :: (\text{Eq } a) \Rightarrow a \rightarrow [a] \rightarrow \text{Bool} \\
\text{elem } _\_ \text{ [] } = \text{False} \\
\text{elem } x \text{ (y:ys) } = x==y \text{ || elem } x \text{ ys}
\]

hs-to-coq [CPP'18]

\[
\text{Definition elem } \{a\} \text{ ‘(Eq } a) : a \rightarrow \text{list } a \rightarrow \text{bool } := \\
\text{fix elem } \text{ arg } 0\_\_ \text{ arg } 1\_\_ \\
\text{ := match arg } 0\_\_, \text{ arg } 1\_\_ \text{ with} \\
\text{ | _\,, nil } = \text{ false} \\
\text{ | x\,, cons } y \text{ ys } = \text{ orb } (x == y) \text{ (elem } x \text{ ys) } \\
\text{end.}
\]
Fast and Loose Reasoning is Morally Correct

John Halton

Army Research Laboratory

Abstract

The question of whether maneuvering in the stock market is morally correct has been debated for centuries. In this paper, we argue that fast and loose reasoning, which is based on intuition and experience rather than formal logic, is in fact the morally correct approach. We provide empirical evidence that supports this argument and discuss the implications for ethical decision-making in business and finance.

Keywords: Fast and Loose Reasoning, Morality, Finance
Real-World Haskell Code

The `Data.Set` data type:

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

type Size = Int
```

The `Data.IntSet` data type:

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

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

```haskell
5
```
Real-World Haskell Code

The `Data.Set` data type:

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

type Size = Int
```

The `Data.IntSet` data type:

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

type Prefix = Int
type Mask = Int
type BitMap = Word
type Key = Int
```
The `Data.Set` data type:

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

Type declarations:

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

The `Data.IntSet` data type:

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

Type declarations:

```haskell
    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-monicadccp, 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 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
```

Types

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

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, i.e. 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
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
prop_UnionAssoc :: IntSet -> IntSet -> IntSet -> Bool
prop_UnionAssoc t1 t2 t3
    = union t1 (union t2 t3) == union (union t1 t2) t3
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.
{-# 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 -> fold1FB (\xs x -> c x n xs)) #-
{-# RULES "IntSet.toDescListBack" [1] fold1FB (\xs x -> x : xs) [] = toDescList #-}
class Semigroup a => Monoid a where

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

1. \( x \cdot mempty = x \)
2. \( mempty \cdot x = x \)
3. \( x \cdot (y \cdot z) = (x \cdot y) \cdot z \) (Semigroup law)
4. \( mconcat = foldr \ (\cdot) \ mempty \)
class Semigroup a => Monoid a where

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

- \( x \leftrightarrow \text{mempty} = x \)
- \( \text{mempty} \leftrightarrow x = x \)
- \( x \leftrightarrow (y \leftrightarrow z) = (x \leftrightarrow y) \leftrightarrow z \) (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
}.
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.
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 \union 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

```haskell
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

```haskell
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 => (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

\[
\text{elem} :: (\text{Eq } a) \Rightarrow a \rightarrow [a] \rightarrow \text{Bool}
\]
\[
\text{elem } \_ \text{ } [\_\_] = \text{False}
\]
\[
\text{elem } x \text{ } (y:ys) = x==y \lor \text{ elem } x \text{ } ys
\]

hs-to-coq [CPP’18]

\[
\text{Definition elem } \{a\} \{\{\text{Eq } a\}\} : a \rightarrow \text{list } a \rightarrow \text{bool} :=
\]
\[
\text{fix elem } \text{ } \text{arg}_0\_\_ \text{ } \text{arg}_1\_\_ \\
\text{ := match } \text{ } \text{arg}_0\_\_, \text{ } \text{arg}_1\_\_ \text{ with}
\]
\[
| \_\_, \text{nil} \Rightarrow \text{false}
\]
\[
| \text{x}, \text{cons } y \text{ } ys \Rightarrow \text{orb } (x == y) \text{ } (\text{elem } x \text{ } ys)
\]
\[
\text{end.}
\]

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

11
balanceR : a -> Set a -> Set a -> Set a

balanceR x l r = case 1 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 rlx@r(rs rlx rll rlr) rr@(rr@rrs _ _))
      | rls < ratio^rrs -> Bin (1+rs) rx (Bin (1+rls) x Tip rlx) rr
      | otherwise -> Bin (1+rs) rlx (Bin (1+size rll) x Tip rll) (Bin (1+rrs+size rlr) x rlr rr)

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

(Bin rs rx rlx rlr)
  | rs > delta^ls -> case (r1, rr) of
    (Bin rlx rll rlr, Bin rrs _ _ _)
      | rls < ratio^rrs -> Bin (1+ls+rs) rx (Bin (1+ls+rls) x l rl1) rr
      | otherwise -> Bin (1+ls+rs) rlx (Bin (1+ls+size rll) x l rll) (Bin (1+rrs+size rlr) x rlr rr)
    (_, _) -> error "Failure in Data.Map.balanceR"
  | otherwise -> Bin (1+ls+rs) x l r
Partiality

In Haskell

\[
\text{error :: String \to a}
\]
Partiality

In Haskell

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

In Coq

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

In Haskell

\[ \text{error :: String} \rightarrow \text{a} \]

In Coq

\[
\begin{align*}
\text{Class Default (a : Type) := \{ default : a \}.} \\
\text{Definition error \{a\} ‘\{Default a\} : String} \rightarrow \text{a} \\
\quad := \text{fun} \_ \Rightarrow \text{default.}
\end{align*}
\]
Partiality

In Haskell

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

In Coq

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

```
1044 fromDistinctAscList :: [a] -> Set a
1045 fromDistinctAscList [] = Tip
1046 fromDistinctAscList (x0 : xs0) = go (1::Int) (Bin 1 x0 Tip Tip) xs0
1047       where
1048          go !_ t [] = t
1049          go s l (x : xs) = case create s xs of
1050                             (r :*: ys) -> let !t' = link x l r
1051                             in go (s `shiftL` 1) t' ys
1052
1053          create !_ [] = (Tip :*: [])
1054          create s xs@(x : xs')
1055              | s == 1 = (Bin 1 x Tip Tip :*: xs')
1056              | otherwise = case create (s `shiftR` 1) xs of
1057                             res@(_ :*: []) -> res
1058                             (l :*: (y:ys)) -> case create (s `shiftR` 1) ys of
1059                             (r :*: zs) -> (link y l r :*: zs)
```
Deferred termination

To define recursive functions:

**Axiom deferredFix**: \(\forall \{a \rightarrow r\} \{\text{Default } r\},\)

\(((a \rightarrow r) \rightarrow (a \rightarrow r)) \rightarrow (a \rightarrow r)\).

To verify recursive functions:

**Definition recurses_on** \(\{a \rightarrow r\} \{P : a \rightarrow \text{Prop}\} \{R : a \rightarrow a \rightarrow \text{Prop}\} \{f : (a \rightarrow r) \rightarrow (a \rightarrow r)\} :\)

\(\forall g \, h \, x, P x \rightarrow (\forall y, P y \rightarrow R y x \rightarrow g y = h y) \rightarrow f g x = f h x.\)

**Axiom deferredFix_eq_on**: \(\forall \{a \rightarrow r\} \{\text{Default } r\} \{f : (a \rightarrow r) \rightarrow (a \rightarrow r)\} \{P : a \rightarrow \text{Prop}\} \{R : a \rightarrow a \rightarrow \text{Prop}\} \{\text{well_founded } R\} \rightarrow \)

\(\forall x, P x \rightarrow \text{deferredFix } f x = f \text{ (deferredFix } f) x.\)
Deferred termination

To define recursive functions:

\[
\text{Axiom deferredFix: forall } \{a \ r\} \ '{\text{Default } r},
((a \rightarrow r) \rightarrow (a \rightarrow r)) \rightarrow (a \rightarrow r).
\]

To verify recursive functions:

\[
\text{Definition recurses_on } \{a \ r\}
(P : a \rightarrow \text{Prop}) (R : a \rightarrow a \rightarrow \text{Prop}) (f : (a \rightarrow r) \rightarrow (a \rightarrow r))
:= \text{forall } g h x,
P x \rightarrow
(\text{forall } y, P y \rightarrow R y x \rightarrow g y = h y) \rightarrow
f g x = f h x.
\]

\[
\text{Axiom deferredFix_eq_on: forall } \{a \ r\} \ '{\text{Default } r} f P R,
\text{well_founded } R \rightarrow \text{recurses_on } P R f \rightarrow
\text{forall } x, P x \rightarrow \text{deferredFix } f x = f (\text{deferredFix } f) x.
\]
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**

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

**In Coq**

```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
A theory of dyadic intervals

Lemmas about integers and bit-wise operations

Lemmas about lists and sortedness

Automation for reasoning about Ord

Instantiating Coq’s FSetInterface

Tests and specifications

Tests

Headers, types and proof preparations

Functions and instances (verified)

(unverified)

(untranslated)