>_ Practical Verification of Functional Libraries

D.Sabharwal
@student.tudelft.nl

J.G.H.Cockx , L.F.B.Escot
@tudelft.nl

CSE-3000
29 June 2021

1

The ***Curry-Howard isomorphism*** states that logic formulas correspond to types, proofs correspond to terms, provability corresponds to inhabitation and that proof normalization corresponds to term reduction.

This isomorphism can be utilised in static verification of programs as -

- It is possible to stipulate properties of functions as types and provide proofs by defining functions of said types.
- Equational reasoning can be used to prove properties of the form lhs ≡ rhs.
- Propositions as types can be used to add pre-condition logic to functions.
- Essential invariants of data-types can be enforced through properties in their constructors.

Agda is a language with *totality* (no evaluation-stopping errors and always terminates) and *dependent types*.
Thus its type system defines all the elements needed to write propositional and first-order logic statements as types.

agda2hs defines a subset of Agda, which can be used for writing programs, for which it provides dictionary translation to Haskell. Since these programs are in Agda, proof based verification can be performed before the translation to Haskell.

2

Is *agda2hs* a viable framework for bringing formal verification to Haskell programs ?

1. Can we port the Haskell library Data.Map to the language set of agda2hs ?
2. What invariants and properties are guaranteed in this library ?
3. Can we formally state and verify these properties ?

3

Re-writing Data.Map in agda2hs

Data.Map is a library providing an efficient Map interface in Haskell.
Internally the Map is represented as a ***balanced binary-search tree***.

The library provides **~150 functions** and **~7 type-class instances** performing various operations on Maps. These were successfully re-written in Agda with minimal changes to their definitions.

Main steps :

- Remove incomplete definitions through pre-conditions
- Rewrite functions to convince Agda's termination checker
- Add type-class instances through record types

```
null :: Map k a -> Bool
null Tip      = True
null (Bin {}) = False
```

```
null : {k a : Set} → Map k a → Bool
null Tip = true
null (Bin _ _ _ _ _) = false
```

```
find :: Ord k => k -> Map k a -> a
find _ Tip = error
  "Map: given key is not present.."
find k (Bin _ kx x l r) = ...
```

```
find : {k a : Set} → ⦃ kOrd : Ord k ⦄
     → (key : k) (map : Map k a)
     → {key ∈ map} → a
find key t@(Bin sz kx x l r) {prf} = ...
```

```
instance (Ord k) => Foldable (Map k)
 where
  foldMap f Tip = mempty
  foldMap f (Bin _ _ v l r) =
    mappend (foldMap f l)
      (mappend (f v) (foldMap f r))
```

```
iFoldableMap : {k : Set} ⦃ _ : Ord k ⦄
             → Foldable (Map k)
iFoldableMap .foldMap f Tip = mempty
iFoldableMap .foldMap f (Bin _ _ v l r) =
  mappend (foldMap f l)
    (mappend (f v) (foldMap f r))
```

5

Outcome

A verified version of Data.Map written in Agda was produced.
Can be used for future verification work on the same library or dependent libraries.

Agda2hs provides a Haskell translation after checking and erasing the verification.
Required upgrades to agda2hs were also identified to support a larger subset of Haskell.

Check out the library : ***https://github.com/dxts/agda2hs-map***

Try out agda2hs yourself : ***https://github.com/agda/agda2hs***

4

Properties and Verification

- ***External Verification***

Type-class laws

The required properties of the 6 type class instances were verified.

```
foldableFunctorMap : (f : a → b) (m : Map k a) → foldMap f m ≡ (fold ∘ fmap f) m
```

Equivalent function re-writes

Prove that more optimised function definitions are equivalent to simpler ones.

```
toAscListRewrite : (sz : Nat) (kx : k) (x : a) (l r : Map k a)
  → (toAscList (Bin sz kx x l r)) ≡ (toAscList l) ++ ((kx , x) ∷ []) ++ (toAscList r)
```

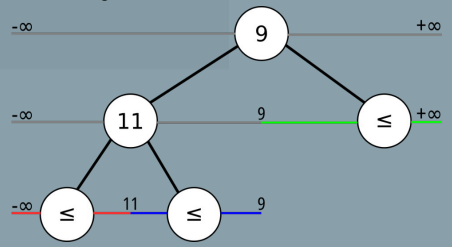
- ***Internal Verification***

Binary search tree ordering

The values in the left subtree must be smaller than the current value, and those in the right subtree must be larger.

```
data Map (k : Set) (a : Set) ⦃ _ : Ord k ⦄ {lower upper : [ k ]=} : Set where
  Tip : {{lsu : lower ≤ upper}} → Map k a {lower} {upper}
  Bin : (size : Nat) → (kx : k) → (x : a) → (l : Map k a {lower} {[ kx ]})
      → (r : Map k a {[ kx ]} {upper}) → Map k a {lower} {upper}
```

How an invalid ordering is detected.



Makes functions that modify the data-structure more involved as the return types have to be dynamically determined.

```
insert : {lower upper : [ k ]=} → (ky : k) → (y : a) → (m : Map k a {lower} {upper})
  → Map k a {min lower [ ky ]} {max upper [ ky ]}
```

TU Delft