Specification and Verification of Functional Programs: Verifying Haskell’s IntMap

By Mohamed Abaker & Vikram Singh

Advised by Professor Stephanie Weirich & Yao Li

Overview

In this project, we used the Coq Proof Assistant and the hs-to-coq tool to specify and verify Haskell’s (a functional programming language) IntMap data structure. We described how we expected the various functions of the IntMap data structure to act, and then proved that it did so under all circumstances. Our work is a proof-of-concept of the Haskell tool and helps confirm the correctness of IntMap.

This project built on previous work done by the Upenn’s PLClub as part of the larger DeepSpec project – which conducts research on furthering the end-to-end correctness of software and hardware programs.

Coq

Coq is a proof assistant that has been under development since 1983.

- A system of machine-checked formal reasoning
- Used by both academia and industry (most famously to prove the 4-color map theorem).

In this project, Coq was used as the environment in which to verify the lemmas that specified what the IntMap functions were supposed to do.

hs-to-coq

Hs-to-coq is a tool that translates Haskell code into gallina code that can be verified by the coq proof assistant.

- Already been used to verify other Haskell data structures such as Set, Map & IntSet

hs-to-coq was used to generate the IntMap translation into Coq that we could then verify.

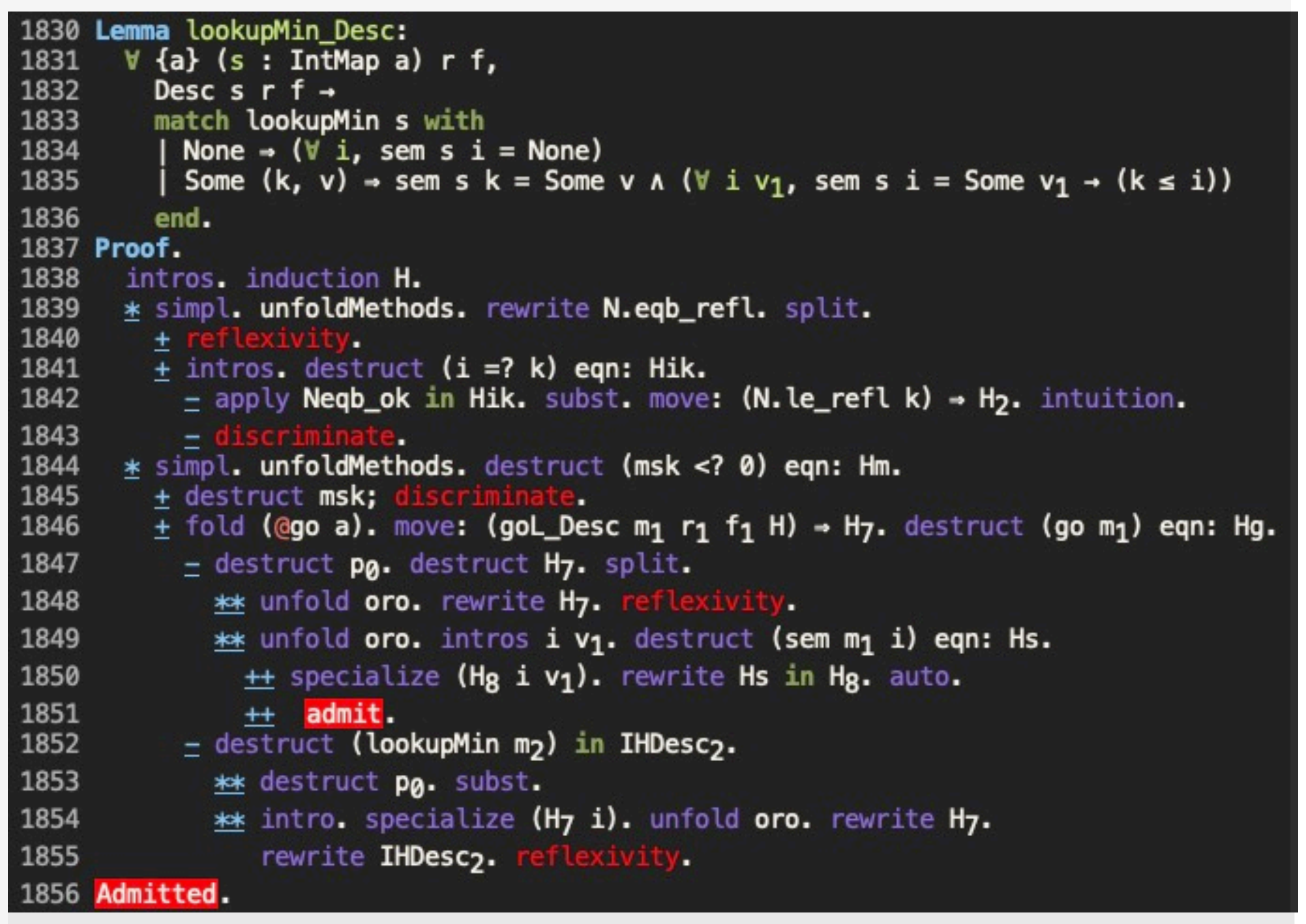
IntMap

A map is an efficient way of storing key-value pairs. An IntMap is a special map that only has integers as keys.

Internally, the IntMap is stored as a Patricia trie, which uses the binary representation of the integers as a way of creating a search tree to efficiently check if keys are in the map.

Our Work

- Drew on various sources (Haskell documentation, existing specifications for other data structures etc.) to specify the functions
- Verified functions on 3 levels
  - Resulting IntMap is described by a function over a certain range (*Desc*)
  - Resulting IntMap can be described by a function (*Sem*)
  - Resulting IntMap is well-formed (*WF*)
- Used various Coq tactics and tactic automation to prove lemmas based on the specifications
- Worked on *insert*, *delete*, *map*, *filter*, *submap, min/max* functions



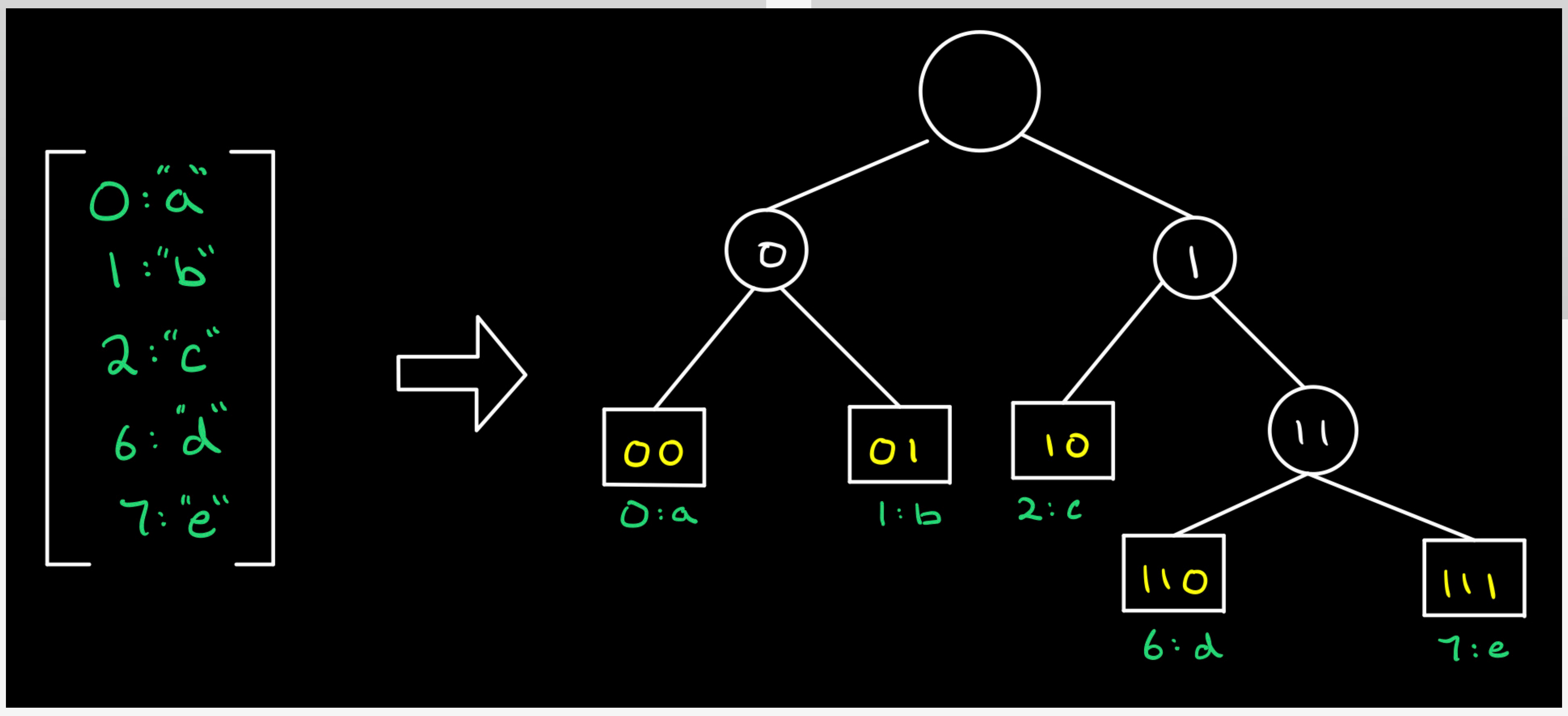
```
Lemma lookupMin_Desc:
  ∀ {a} (s : IntMap a) r f,
    Desc s r f →
    match lookupMin s with
    | None → (∀ i, sem s i = None)
    | Some (k, v) → sem s k = Some v ∧ (∀ i v₁, sem s i = Some v₁ → (k ≤ i))
    end.
Proof.
  intros. induction H.
  * simpl. unfoldMethods. rewrite N.eqb_refl. split.
    + reflexivity.
    + intros. destruct (i =? k) eqn: Hik.
      - apply Neqb_ok in Hik. subst. move: (N.le_refl k) → H₂. intuition.
      - discriminate.
  * simpl. unfoldMethods. destruct (msk <? 0) eqn: Hm.
    + destruct msk; discriminate.
    + fold (@go a). move: (goL_Desc m₁ r₁ f₁ H) → H₇. destruct (go m₁) eqn: Hg.
      - destruct p₀. destruct H₇. split.
        ** unfold oro. rewrite H₇. reflexivity.
        ** unfold oro. intros i v₁. destruct (sem m₁ i) eqn: Hs.
          ++ specialize (H₈ i v₁). rewrite Hs in H₈. auto.
          ++  admit.
      - destruct (lookupMin m₂) in IHDesc₂.
        ** destruct p₀. subst.
        ** intro. specialize (H₇ i). unfold oro. rewrite H₇.
           rewrite IHDesc₂. reflexivity.
Admitted.
```

Sample proof: Verifying lookupMin

Results & Future Work

- 1300 lines of code added
- 11 different functions verified over various function groups
- Significant coverage of the IntMap data structure
- Code pushed to PLClub’s public GitHub repository

In the future, we hope to complete the verification of IntMap and contribute to hs-to-coq in other ways.



A Patricia Trie: how an IntMap stores key/value pairs