CS 456 (Fall 2018)
Scribe Notes: 2

Albert Yu, Adam Johnston

- Lists
  - Inductive data type that has an infinite collection of elements
  - Not through enumeration but inductive type definition allowing us to build up from smaller elements
- Bags
  - These are simply Lists with set operations
  - If Bag is just a List, why do we need to define a Bag?
    - If we hide the implementation through abstraction, we can change the implementation for Bag, and it should not affect clients of Bag.
    - The interface of Bag is defined through the operations, the implementation should not matter, and can be hidden away.
  -
- Maps
  - Instead of using lists of keys and values, we use functions such that given the function F and an input key, F(key) will be the value associated with key.
  - Total Maps

    ```
    Definition total_map (A:Type) := string → A.
    ```

    Given a type A, we can give our total map any string and receive a value of type A.

    ```
    Definition t_empty {A:Type} (v : A) : total_map A :=
     (fun _ ⇒ v).
    ```

    Defines a total map function that given any key value returns a default value v

    ---

    ```
    Definition t_update {A:Type} (m : total_map A)
                        (x : string) (v : A) :=
     fun x' ⇒ if beq_string x x' then v else m x'.
    ```

    Given a map and a key value pair (x and v), essentially wraps the map in another map whose function takes in input x’ and checks if x’ is equal to the key x,. If so it returns v, otherwise calls the input map with argument x’. This means that if none of the t_update maps contain x’, the function will eventually reach the t_empty default map, which returns v regardless of what x’ is.

- Partial Maps
  - It is like a total map, except the values are all option types. If the key is found in the map, it returns a value of (Some v) otherwise it goes down to the empty map and returns a None value. In other words, the v above is replaced with (Some v), and the default value is now None.

    ```
    Definition partial_map (A:Type) := total_map (option A).
    ```

    ```
    Definition empty {A:Type} : partial_map A :=
      t_empty None.
    ```

    ```
    Definition update {A:Type} (m : partial_map A)
                      (x : string) (v : A) :=
      m & { x --> (Some v) }.
    ```

- Curry-Howard Correspondence
  - This shows how algorithmic proofs are directly related to computer programs. This can be shown in how propositions and types both use the Inductive keyword to build cases.
  - Programs transform (e.g. call functions on) inductive data types such as nat
  - Proofs transform propositions into other propositions (e.g. A -> B)
    - Propositions are essentially types in the same way nat is a type
- Implication
  - There are two values with implication (->) in their types (e.g. nat -> Prop)
    - Inductive Constructors

      Example: determining if a number is even:

      ```
      Inductive ev : nat -> Prop :=
        | ev_0 : ev 0
        | ev_SS : forall n, ev n -> ev (S (S n)).
      ```

    - Functions

      ```
      Theorem ev_plus4 : ∀ n, ev n → ev (4 + n).
      Proof.
        intros n H. simpl.
        apply ev_SS.
        apply ev_SS.
        apply H.
      Qed.
      ```

      The function takes in ev n and returns ev (4 + n)

- Polymorphic Data Types

- Types (sets) can be viewed at the same level as values. Therefore, we can supply them to constructors.
- Coq is powerful enough that we can often omit stuff, such as omitting that X:Type. If we use Argument Const {X}. The curly brackets say that there should be that argument, but it doesn't necessarily have to be specified, and Coq will try to figure out what it is.
- We can also do it in the argument of a function, like repeat''' {X: Type} which says that the user doesn't necessarily have to specify what X is.
- polymorphic functions work on the structure of the object, not the nature of the object. A function that adds 1 to each value of the list cannot be polymorphic. When we work on only the structure and not the actual values, we can use polymorphism.
- In Java, polymorphism works in a very different way. Since java is object oriented and has classes, polymorphism is closely related to classes. Polymorphism is only possible when two classes inherit the same parent class. In Java, it is the class hierarchy that determines whether polymorphism is possible or not, instead of the structure of the variable as is done in coq.
- In Python, polymorphism works in a way that's similar to coq, since variables aren't explicitly given types, polymorphism works very naturally. You can write a function that takes an input without specifying the type of the input, and the function could work over multiple different data types. This is very similar to polymorphism in coq, where the Type variable is implicit. The structure of the variables determines whether or not polymorphism is possible.

- Proof Topics

- Proof Tactics
  - Apply
    - "Prove a goal using a hypothesis, lemma , or constructor
    - Assume the hypothesis space and goal looks like:
      -----Hypothesis-------
      Hab: A -> B
      Ha: A
      ---------Goal------------
      B

      You can either [apply Hab] to the goal to end up with
      -----Hypothesis-------
      Hab: A -> B
      Ha: A

---------Goal----------
A

This uses backwards reasoning to show that, if we know B, and A implies B, then if we prove A, then B is true by Hab.
Alternatively, we can [apply Hab in Ha], resulting in:
-----Hypothesis-------
Hab: A -> B
Ha: B
---------Goal----------
B

This uses forward reasoning to show that, since we know A is true (via Ha), and A implies B (Hab), we know B must be true.
With either approach, all that is left is to apply Ha to the goal.

- Assert
  - Used to define a lemma within a proof. This has the benefit of having access to any information we have in our hypothesis prior to the assert.
- Assumption
  - Used to solve a goal given that it is already in the hypothesis
- Constructor
  - Given an inductive definition
    Inductive mydef: nat -> Prop :=
    | base: mydef 0
    | buildingUp: forall n, mydef n -> mydef (S(n))
    If your goal matches the right side of the inductive definition, such as mydef (S(S(n)), you can use constructor to transform the goal to the left side of the definition (mydef (S(n)) until it reaches the base.
- Contradiction
  - Looks for False in the hypothesis, or two hypothesis that contradict each other. If it finds a False or contradictory hypothesis, then the goal is resolved.
- Destruction
  - Destruct something into its various possible forms, typically generating one case for each constructor. In a sense, it is induction without the inductive hypothesis.
  - If we have a conjunction as one of our hypothesis, we can use Destruct to break it into 2 hypothesis
    -----Hypothesis-------

H: A /\ B

Destruct H as {HA HB}.

-----Hypothesis-------
HA: A
HB: B

- If we have a disjunction as one of our hypothesis, we can use Destruct to break it into 2 hypothesis. This would generate two sub goals, each identical to our goal. One would have the left part of the disjunction as one of the hypothesis, and the other sub goal would have the right part of the disjunction as one of the hypothesis.
- Destructing, like induction, breaks things into cases. However each case does not provide any information to the others. For example, destructing over a boolean b would yield two cases in the goal where b is replaced with true in one, and false in the other.
    - Destructing using eqn (destruct b eqn: eq) gives you a hypothesis with the value of b at each case. For example destructing a boolean function boolFun(x) would yield H: boolFun(x)= true, and in the second case H: boolFun(x) = false.
    - It is used to preserve information that is usually lost when destruct is called.

- Discriminate
    - Similar to part of inversion. If there is an equality where the two sides are structurally different (based on properties of inductive types, such as having different constructors), then discriminate will discharge the goal.
- Exact
    - If our goal is exactly the same as one of our hypothesis, we can use exact to discharge the goal.
- Generalize Dependent
    - Generalize Dependent is basically the inversion of intros.Generalized Dependent takes a variable out of the hypothesis and back into the context.
- Induction
    - Given an inductive constructor, with a base case and inductive cases, will split the goal cases corresponding to each case of the constructor. Once the base case is proven, an inductive hypothesis is available for use.

        - For example induction on a natural number n will result in two cases, n being 0 and n being one plus another number (S(n')).
        - Induction can also be performed on a hypothesis with multiple induction definitions in it (e.g. H: n <= m), which allows the relationships between them to be maintained in each case.
    - Intuition
        - Intuition will try to find false hypothesis and discharge the goal.
    - Intros
        - Introduces known variables and relations into the hypothesis
    - Inversion
        - Inversion looks at properties inherent to inductive types. when in our hypothesis we have a statement, and we know that it is the consequence of a property of the inductive type, the inversion tactic will add such property into the hypothesis.
        - For example, constructors of inductive types are injective. If constructor(x) = constructor(y), then x = y.
          -----Hypothesis-------
          H: Const(x) = Const(y)
          We can call inversion H, and we'll get.
          -----Hypothesis-------
          H: Const(x) = Const(y)
          H1: n = m
          Additionally, inversion also does some other things.
          For example, it will also substitute variables in the goal with the new hypothesis.
        - If the hypothesis we call inversion on contradicts the properties of induction, inversion will also discharge the goal.
          For example: every constructor is disjoint.
          -----Hypothesis-------
          H: Const1(x) = Const2(y)
          We can call inversion H, and the goal will be discharged.
    - Left/Right
        - If we have a disjunction as our goal, we can use left/right to select which one of the parts we want to pick as our goal.
    - Omega
        - Given a set of equalities and inequalities in the goal and hypotheses, omega attempts to solve the goal by determining the relationships between the inequalities. Omega uses Presburger arithmetic.
    - Reflexivity

        - Reflexivity performs some basic simplification and checks to see if both sides of the goal equate to the same value, and if so, solves the goal. If it cannot solve the goal, the tactic fails.
    - Repeat
        - Repeat is used to repeatedly use another tactic.
    - Rewrite
        - If we have a equality in our hypothesis, we can rewrite one side of it into the other side, both inside other hypothesis and in our goals
    - Simpl
        - Simpl does some basic simplifications.
    - Split
        - If we have a conjunction as our goal, split will split the goal into two sub goals.
    - Subst
        - If we have two variables that are equal to each other, we can call subst with the variable name to substitute all occurance of one variable with the other, and remove the other variable.
        - Simply calling Subst will cause coq to automatically find all such variables that are equal to each other, and substitute all occurrences of all pairs of equal variables.
    - Symmetry
        - Symmetry switches the two sides of a equality.
    - Unfold
        - Given a definition add2(n) = n + 2, unfold takes a call to add2 and replaces it with the right hand side of the definition (n+2).
        - Usually used to unfold a function. It will often unfold a function and replace the function call with the actual code of the function