CS3234 – Logic for Proofs and Programs
AY24/25, Y2S2
Notes

Sim Ray En Ryan (A0273780X, e1123183@u.nus.edu)

December 3, 2025

Contents

When there is an =, both sides can be equivalent using:

- Reflexivity
- Symmetry
- Transitivity

5.1.1 Reflexivity

$\forall A, A = A$ `reflexivity`

5.1.2 Symmetry

$A = BB = A$ `H_j : 1 = j symmetry in H_j H_j : j = 1`

5.1.3 Transitivity

5.2 The clear tactic

5.3 Using specifications to state other specifications

For example, the recursive specification of multiplication relies on the recursive specification of addition.

5.4 More on soundness and completeness

Soundness: add i j = k -> k = i + j

Completeness: i + j = k -> k = add i j

5.4.1 Eureka Lemma

```
Lemma about_a_recursive_addition :
  forall add : nat -> nat -> nat,
    recursive_specification_of_addition add ->
    forall i j : nat,
      add i j = i + j.
```

This lemma can be proved by induction, and makes two propositions about soundness and completeness that follow.

5.4.2 The injection tactic

Creates a chain of implications about a pair, depth first, from left to right.
`injection H_ij as H_i H_j.`

5.4.3 The discriminate tactic

Used when an assumption is an equality involving distinct data constructors `0 = 1` Since 0 is O and 1 is S (O), they use different constructors and can thus be discriminated. Now that the hypothesis is false, the implication is vacuously true.

5.4.4 The contradiction tactic

Used when an assumption can imply false. Once that is done, the discriminate tactic can be used to prove. Note that this is not related to proof by contradiction.

5.5 Proving Correctness of Unit Tests

Unit tests are sound whenever they always succeed for an implementation that satisfies the specification of the function to test. Unit test functions are sound if it succeeds given a function that satifies the given specification.

5.5.1 The compute tactic

Use it to compute mathematical equations, such as `(2 =? 2) && (2 =? 2) && (4 =? 4)` to `true`. It systematically unfolds all functions that are called, then simplifies it.

5.6 Backward and forward proofs

Usually, proofs are proved backward in tCPA (from final goal to initial hypothesis).

- Backwards
  - `apply H_C_implies_D. ->` C
  - `apply H_B_implies_C. ->` B
  - `apply H_C_implies_D. ->` A
  - `apply H_A ->` QED
- Forwards
  - `assert (H_B := H_A_implies_B H_A).`
  - `assert (H_C := H_B_implies_C H_B).`
  - `exact (H_C_implies_D H_C).`

5.7 The fold and unfold tactics

Use `fold_unfold_tactic`, but only in the proofs of fold-unfold lemmas.

6 Structuring Proofs

7 Lists in tCPA

7.1 Equality Predicate for lists

Is sound and complete given the completeness and soundness of the equality predicate of V.

7.2 Length

Can be implemented recursively using S (length vs'), or with an accumulator length_acc vs' (S a).

7.3 Copy

Can be implemented recursively using v :: copy vs', or simply by returning vs. Copy is idempotent, meaning that applying it twice, or any amount of times, is the same as applying it once. In the case of copy, it is also the same as applying it zero times since it is its own inverse. Copying a list preserves its length, and in tCPA, is exactly the same. This is not true for languages that can have references to two different objects.

7.4 Append

Implemented using v1 :: append v1s' v2s. Nil is left and right neutral for append, is associative, but not commutative. It commutes with copy. Concatenating two lists preserves the length of the sum of the two lists. append vs nil is the same as copy, and append nil vs is the same as simple copy.

7.5 Reverse

Implemented recursively using append (reverse vs) (v :: nil). It is involuntary, meaning that applying it tiwce produces the original value. Reversing a list preserves its length, and commutes with append. A accumulator version can be implemented using reverse_ vs' (v :: a).

7.6 Map

Implemented recursively using f v :: map f vs'. Can implement the copy function using an identity function for f. It preserves length, and is commutative with list append, list reverse.

7.7 List Fold Functions

LFR is also known as reduce, and LFL is also known as accumulate. LFR traverses a list recursively, while LFL traverses a list tail recursively with an accumulator.

```
Fixpoint list_fold_right (V W : Type) (nil_case : W) (cons_case : V -> W -> W) (vs :
    list V) : W :=
  match vs with
    nil =>
    nil_case
  | v :: vs' =>
    cons_case v (list_fold_right V W nil_case cons_case vs')
  end.

Fixpoint list_fold_left (V W : Type) (nil_case : W) (cons_case : V -> W -> W) (vs :
    list V) : W :=
  match vs with
    nil =>
    nil_case
  | v :: vs' =>
    list_fold_left V W (cons_case v nil_case) cons_case vs'
  end.
```

Both functions take in two types V and W, a nil case of type W, and a cons case which is a function that takes in a V and W and outputs a W, and a list containing elements of type V. The idea of both functions is to take an element of the list (from the right or the left respectively), transform the nil case into something else of type W based on the element and cons case.

7.7.1 Sketches of LFR and LFL

LFR LFR n c [v1, v2, v3, nil] = c v1 (c v2 (c v3 n))

Length with LFR LFR 0 (v n: S n) [v1, v2, v3, nil] = 3

LFL LFL n c [v1, v2, v3, nil] = c v3 (c v2 (c v1 n))

Length with LFL LFR 0 (v n: S n) [v1, v2, v3, nil] = 3

7.7.2 Function implemented using LFR and LFL

LFR list_length

LFL list_length_acc

7.7.3 Properties proven about LFR and LFL

If the arguments are the same (such as with list length), then the function is left permutatitve.

8 Existential Proofs

9 Resetting the accumulator

10 Isometries in the equilateral triangle

11 Streams and Lazy Lists

12 Observational Equivalence and coinduction

13 Extracting programs from proofs and Delimited Continuations