Did I Get it Right?

Part 4: Induction for Datatypes

Speaker: David Walker

COS 326

Princeton University

http://~cos326/notes/reasoning-data.php

Equational Reasoning: Some Key Ideas

What is the fundamental *definition of expression equality* (e1 == e2)?

- two expressions are equal if:
  - they evaluate to equal values, or
  - they both raise the same exception
  - they both fail to terminate
- note: we won’t ask you to do proofs about expressions that don't terminate, use I/O or mutable data structures

What are some consequences of this definition?

- expression equality is reflexive, symmetric and transitive
- if e1 --> e2 then e1 == e2
- if e1 == e2 then e[e1/x] == e[e2/x]. (substitution of equals for equals)

How do we prove things about recursive functions?

- we use proofs by induction
- to reason about recursive calls on *smaller* data, we assume the property we are trying to prove (ie, we use the *induction hypothesis*)

More General Template for Inductive Datatypes

```
type t =  C1 of t1 | C2 of t2 | ... | Cn of tn
```

types t1, t2 ... tn, may contain 1 or more occurrences of t within them.

Examples:

```
type mylist =
  MyNil
| MyCons of int * mylist
```

```
type ‘a tree =
  Leaf
| Node of ‘a * ‘a tree * ‘a tree
```

recursive occurrences

More General Template for Inductive Datatypes

type t = C1 of t1 | C2 of t2 | ... | Cn of tn

Theorem: For all x : t, property(x).

Proof: By induction on structure of values x with type t.

More General Template for Inductive Datatypes

type t = C1 of t1 | C2 of t2 | ... | Cn of tn

Theorem: For all x : t, property(x).

Proof: By induction on structure of values x with type t.

Case: x == C1 v:

... use IH on components of v that have type t ...

Case: x == C2 v:

... use IH on components of v that have type t ...

Case: x == Cn v:

... use IH on components of v that have type t ...

A PROOF ABOUT TREES

Another example

```
type ‘a tree = Leaf | Node of ‘a * ‘a tree * ‘a tree

let rec tm f t =
  match t with
    | Leaf -> Leaf
    | Node (x, l, r) -> Node (f x, tm f l, tm f r)

let (<>) f g =
  fun x -> f (g x)
```

Another example

```
type ‘a tree = Leaf | Node of ‘a * ‘a tree * ‘a tree

let rec tm f t =
  match t with
    | Leaf -> Leaf
    | Node (x, l, r) -> Node (f x, tm f l, tm f r)

let (<>) f g =
  fun x -> f (g x)
```

Theorem:

For all (total) functions f : b -> c,
For all (total) functions g : a -> b,
For all trees t : a tree,
tm f (tm g t) == tm (f <> g) t

“Forall intro”

Theorem:
For all (total) functions f : b -> c,
For all (total) functions g : a -> b,
For all trees t : a tree,
tm f (tm g t) == tm (f <> g) t

```
let rec tm f t =
  match t with
    | Leaf -> Leaf
    | Node (x, l, r) -> Node (f x, tm f l, tm f r)

let (<>) f g =
  fun x -> f (g x)
```

To begin, let’s *pick an arbitrary total function f and total function g*.
We’ll prove the theorem without assuming any particular properties of f or g (other than the fact that the types match up). So, for the f and g we picked, we’ll prove:

Theorem:
For all trees t : a tree,
tm f (tm g t) == tm (f <> g) t

Another example

Theorem:

For all trees t : a tree,

tm f (tm g t) == tm (f <> g) t

```
let rec tm f t =
  match t with
    | Leaf -> Leaf
    | Node (x, l, r) -> Node (f x, tm f l, tm f r)

let (<>) f g =
  fun x -> f (g x)
```

Another example

Theorem:
For all trees t : a tree,
tm f (tm g t) == tm (f <> g) t

```
let rec tm f t =
  match t with
    | Leaf -> Leaf
    | Node (x, l, r) -> Node (f x, tm f l, tm f r)

let (<>) f g =
  fun x -> f (g x)
```

Case: t = Leaf

No inductive hypothesis to use.
(Leaf doesn't contain any smaller components with type tree.)

Proof:

    tm f (tm g Leaf)

Another example

Theorem:

For all trees t : a tree,

tm f (tm g t) == tm (f <> g) t

```
let rec tm f t =
  match t with
    | Leaf -> Leaf
    | Node (x, l, r) -> Node (f x, tm f l, tm f r)

let (<>) f g =
  fun x -> f (g x)
```

Case: t = Leaf

No inductive hypothesis to use.
(Leaf doesn't contain any smaller components with type tree.)

Proof:

```
   tm f (tm g Leaf)
== tm f Leaf              (eval tm g Leaf)
== Leaf                   (eval tm f Leaf)
== tm (f <> g) Leaf       (reverse eval)
```

Another example

Theorem:
For all trees t : a tree,
tm f (tm g t) == tm (f <> g) t

Case: t = Node(v, l, r)

IH1: tm f (tm g l) == tm (f <> g) l
IH2: tm f (tm g r) == tm (f <> g) r

```
let rec tm f t =
  match t with
    | Leaf -> Leaf
    | Node (x, l, r) -> Node (f x, tm f l, tm f r)

let (<>) f g =
  fun x -> f (g x)
```

Another example

Theorem:
For all trees t : a tree,
tm f (tm g t) == tm (f <> g) t

Case: t = Node(v, l, r)

IH1: tm f (tm g l) == tm (f <> g) l
IH2: tm f (tm g r) == tm (f <> g) r

Proof:
    tm f (tm g (Node (v, l, r)))

== tm (f <> g) (Node (v, l, r))

```
let rec tm f t =
  match t with
    | Leaf -> Leaf
    | Node (x, l, r) -> Node (f x, tm f l, tm f r)

let (<>) f g =
  fun x -> f (g x)
```

Another example

Theorem:
For all trees t : a tree,
tm f (tm g t) == tm (f <> g) t

Case: t = Node(v, l, r)

IH1: tm f (tm g l) == tm (f <> g) l
IH2: tm f (tm g r) == tm (f <> g) r

Proof:

```
    tm f (tm g (Node (v, l, r)))
== tm f (Node (g v, tm g l, tm g r))          (eval inner tm)



== tm (f <> g) (Node (v, l, r))
```

```
let rec tm f t =
  match t with
   | Leaf -> Leaf
   | Node (x, l, r) -> Node (f x, tm f l, tm f r)

let (<>) f g =
  fun x -> f (g x)
```

Another example

Theorem:
For all trees t : a tree,
tm f (tm g t) == tm (f <> g) t

Case: t = Node(v, l, r)

IH1: tm f (tm g l) == tm (f <> g) l
IH2: tm f (tm g r) == tm (f <> g) r

```
let rec tm f t =
  match t with
    | Leaf -> Leaf
    | Node (x, l, r) -> Node (f x, tm f l, tm f r)

let (<>) f g =
  fun x -> f (g x)
```

Proof:

tm f (tm g (Node (v, l, r)))	
== tm f (Node (g v, tm g l, tm g r))	(eval inner tm)
Node ((f <> g) v, tm (f <> g) l, tm (f <> g) r)	
== tm (f <> g) (Node (v, l, r))	(eval reverse)

Another example

Theorem:
For all trees t : a tree,
tm f (tm g t) == tm (f <> g) t

Case: t = Node(v, l, r)

IH1: tm f (tm g l) == tm (f <> g) l
IH2: tm f (tm g r) == tm (f <> g) r

```
let rec tm f t =
  match t with
    | Leaf -> Leaf
    | Node (x, l, r) -> Node (f x, tm f l, tm f r)

let (<>) f g =
  fun x -> f (g x)
```

Proof:
    tm f (tm g (Node (v, l, r)))
== tm f (Node (g v, tm g l, tm g r))   (eval inner tm)
== Node (f (g v), tm f (tm g l), tm f (tm g r))   (eval – since g, tm are total)

    Node ((f <> g) v, tm (f <> g) l, tm (f <> g) r)
== tm (f <> g) (Node (v, l, r))   (eval reverse)

Another example

Theorem:
For all trees t : a tree,
tm f (tm g t) == tm (f <> g) t

Case: t = Node(v, l, r)

IH1: tm f (tm g l) == tm (f <> g) l
IH2: tm f (tm g r) == tm (f <> g) r

```
let rec tm f t =
  match t with
    | Leaf -> Leaf
    | Node (x, l, r) -> Node (f x, tm f l, tm f r)

let (<>) f g =
  fun x -> f (g x)
```

Proof:

tm f (tm g (Node (v, l, r)))	
== tm f (Node (g v, tm g l, tm g r))	(eval inner tm)
== Node (f (g v), tm f (tm g l), tm f (tm g r))	(eval – since g, tm are total)
Node ((f <> g) v, tm (f <> g) l, tm f (tm g r))	
== Node ((f <> g) v, tm (f <> g) l, tm (f <> g) r)	(IH2)
== tm (f <> g) (Node (v, l, r))	(eval reverse)

Another example

Theorem:
For all trees t : a tree,
tm f (tm g t) == tm (f <> g) t

Case: t = Node(v, l, r)

IH1: tm f (tm g l) == tm (f <> g) l
IH2: tm f (tm g r) == tm (f <> g) r

```
let rec tm f t =
  match t with
    | Leaf -> Leaf
    | Node (x, l, r) -> Node (f x, tm f l, tm f r)

let (<>) f g =
  fun x -> f (g x)
```

Proof:

tm f (tm g (Node (v, l, r)))	
== tm f (Node (g v, tm g l, tm g r))	(eval inner tm)
== Node (f (g v), tm f (tm g l), tm f (tm g r))	(eval – since g, tm are total)
== Node ((f <> g) v, tm f (tm g l), tm f (tm g r))	
== Node ((f <> g) v, tm (f <> g) l, tm f (tm g r))	(IH1)
== Node ((f <> g) v, tm (f <> g) l, tm (f <> g) r)	(IH2)
== tm (f <> g) (Node (v, l, r))	(eval reverse)

Another example

Theorem:
For all trees t : a tree,
tm f (tm g t) == tm (f <> g) t

Case: t = Node(v, l, r)

IH1: tm f (tm g l) == tm (f <> g) l
IH2: tm f (tm g r) == tm (f <> g) r

```
let rec tm f t =
  match t with
    | Leaf -> Leaf
    | Node (x, l, r) -> Node (f x, tm f l, tm f r)

let (<>) f g =
  fun x -> f (g x)
```

Proof:

tm f (tm g (Node (v, l, r)))	
== tm f (Node (g v, tm g l, tm g r))	(eval inner tm)
== Node (f (g v), tm f (tm g l), tm f (tm g r))	(eval – since g, tm are total)
== Node ((f <> g) v, tm f (tm g l), tm f (tm g r))	(eval reverse)
== Node ((f <> g) v, tm (f <> g) l, tm f (tm g r))	(IH1)
== Node ((f <> g) v, tm (f <> g) l, tm (f <> g) r)	(IH2)
== tm (f <> g) (Node (v, l, r))	(eval reverse)

Summary: Proof Template for Trees

```
type 'a tree = Leaf | Node of 'a * 'a tree * 'a tree
```

Theorem: For all x : ‘a tree, property(x).

Proof: By induction on the structure of trees x.

Case: x == Leaf:

... no use of inductive hypothesis (this is the smallest tree) ...

Case: x == Node (v, left, right):

IH1: property(left)
IH2: property(right)

... use IH1 and IH 2 in your proof ...

Summary of Template for Inductive Datatypes

type t = C1 of t1 | C2 of t2 | ... | Cn of tn

Theorem: For all x : t, property(x).

Proof: By induction on structure of values x with type t.

use patterns that divide up the cases

Take inspiration from the structure of the program

Case: x == C1 v:

... use IH on components of v that have type t ...

Case: x == C2 v:

... use IH on components of v that have type t ...

Case: x == Cn v:

... use IH on components of v that have type t ...

Exercise

```
type ‘a tree =  Leaf of ‘a| Node of ‘a tree * ‘a tree


let rec flip (t: ‘a tree) =
 match t with
 | Leaf _ -> t
 | Node (a,b) -> Node (flip b, flip a)
```

Theorem: for all t: ‘a tree, flip(flip t) = t.

Theorem: for all t: ‘a tree, flip(flip (flip t)) = flip t.