Did I get it right?

Part 5: Proofs about Programming Languages

Speaker: David Walker

COS 326

Princeton University

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

You might wonder

We've done some proofs about *individual programs*. eg:

```
let rec even n =
  match n with
    | 0 -> true
    | 1 -> false
    | n -> even (n-2)
```

```
for all naturals n,
   even (2 * n) == true
```

But can we do proofs about entire *programming languages*?

In other words, proofs about *all programs that anyone could ever write in the programming language*?

But there are so many programs ... how do we even get starte

We often think about programs as if they are functions.

But is there another way to represent these functions?

A Trick

Consider assignment #4.

We are able to represent all programs using a data type:

```
type exp =
  Var of variable
| Const of constant
| Op of exp * op * exp
...
```

A Trick

Consider assignment #4.

We are able to represent all programs using a data type:

```
type exp =
   Var of variable
| Const of constant
| Op of exp * op * exp
...
```

We know how to prove things about functions over datatypes, so we know how to prove things about programming languages

What Kinds of Things Might We Prove About PLs?

We typically prove things about functions over data types.

What kinds of functions over programs are there?

```
type exp =
  Var of variable
| Const of constant
| Op of exp * op * exp
...
```

What Kinds of Things Might We Prove About PLs?

We typically prove things about functions over data types.

What kinds of functions over programs are there?

```
type exp =
  Var of variable
| Const of constant
| Op of exp * op * exp
...
```

```
let eval (e:exp) = ...
let synthesize (s:spec) : exp = ...
let type_check (e:exp) = ...
let terminates (e:exp)  = ...
let closed (e:exp) = ...
let is_pure (e:exp) = ...
let compile (e:exp)  = ...
let optimize (e:exp) = ...
let is_correct (s:spec) (e:exp) = ...
let refactor (e:exp) = ...
```

ACM DL DIGITAL LIBRARY   Princeton University

SIGN IN SIGN UP

SEARCH

Conferences

POPL Principles of Programming Languages

The annual Symposium on Principles of Programming Languages is a forum for the discussion of all aspects of programming languages and systems, with emphasis on how principles underpin practice. Both theoretical and experimental papers are welcome, on topics ranging from formal frameworks to experience reports.

Search within POPL: SEARCH

About | Award Winners | Authors | Affiliations | Upcoming Conferences | Sponsors | Publication Archive | Web Archive

POPL subject areas

Compilers Formal language definitions Formal languages and automata theory Formal software verification Lambda calculus Language features Language types Logic Program reasoning Program semantics Program verification Semantics and reasoning Software development process management Type structures Verification

Bibliometrics: publication history

Publication years	1973-2018
Publication count	1,983
Citation Count	51,895
Available for download	1,829
Downloads (6 Weeks)	3,672
Downloads (12 Months)	42,478
Downloads (cumulative)	767,519
Average downloads per article	419.64
Average citations per article	26.17

PROOFS ABOUT PROGRAMMING LANGUAGES: AN EXAMPLE

A simple expression language

```
type id = string
type exp = Int of int | Add of exp * exp | Var of id
```

A simple expression language

```
type id = string
type exp = Int of int | Add of exp * exp | Var of id

let e1 = Add (Int 3, Var "x")
```

A simple expression language

```
type id = string
type exp = Int of int | Add of exp * exp | Var of id

type env
val lookup : env -> id -> int
```

A simple expression language

```
type id = string
type exp = Int of int | Add of exp * exp | Var of id

type env
val lookup : env -> id -> int

let rec eval (env: env) (e: exp) : int =
match e with
  Int i -> i
| Add (e1, e2) -> (eval env e1) + (eval env e2)
| Var x -> lookup env x
```

A simple optimizer

```
type id = string
type exp = Int of int | Add of exp * exp | Var of id

type env
val lookup : env -> id -> int

let rec eval (env: env) (e: exp) : int =
match e with
  Int i -> i
| Add (e1, e2) -> (eval env e1) + (eval env e2)
| Var x -> lookup env x
```

```
let rec opt (e:exp) : exp =
match e with
| Int i -> Int i
| Add (Int 0, e) -> opt e
| Add (e, Int 0) -> opt e
| Add (e1,e2) ->
   Add(opt e1, opt e2)
| Var x -> Var x
```

A simple optimizer

```
type id = string
type exp = Int of int | Add of exp * exp | Var of id

type env
val lookup : env -> id -> int

let rec eval (env: env) (e: exp) : int =
match e with
  Int i -> i
| Add (e1, e2) -> (eval env e1) + (eval env e2)
| Var x -> lookup env x
```

```
let rec opt (e:exp) : exp =
  match e with
| Int i -> Int i
| Add (Int 0, e) -> opt e
| Add (e, Int 0) -> opt e
| Add (e1,e2) ->
   Add(opt e1, opt e2)
| Var x -> Var x
```

Theorem:
For all e : exp, eval (opt e) == eval e

A simple optimizer

```
type id = string
type exp = Int of int | Add of exp * exp | Var of id

type env
val lookup : env -> id -> int

let rec eval (env: env) (e: exp) : int =
match e with
  Int i -> i
| Add (e1, e2) -> (eval env e1) + (eval env e2)
| Var x -> lookup env x
```

```
let rec opt (e:exp) : exp =
  match e with
| Int i -> Int i
| Add (Int 0, e) -> opt e
| Add (e, Int 0) -> opt e
| Add (e1,e2) ->
   Add(opt e1, opt e2)
| Var x -> Var x
```

Theorem:
For all e : exp, eval (opt e) == eval e

Proof: By induction on the structure of expressions e : exp.

A simple optimizer

```
type id = string
type exp = Int of int | Add of exp * exp | Var of id

type env
val lookup : env -> id -> int

let rec eval (env: env) (e: exp) : int =
match e with
  Int i -> i
| Add (e1, e2) -> (eval env e1) + (eval env e2)
| Var x -> lookup env x
```

```
let rec opt (e:exp) : exp =
  match e with
| Int i -> Int i
| Add (Int 0, e) -> opt e
| Add (e, Int 0) -> opt e
| Add (e1,e2) ->
   Add(opt e1, opt e2)
| Var x -> Var x
```

Theorem:
For all e : exp, eval (opt e) == eval e

Proof: By induction on the structure of expressions e : exp.

- Case: Int i
- Case: Add (e1, e2) →
  - Case: Add (Int 0, e2)
  - Case: Add (e1, Int 0)
  - Case: Add (e1, e2) where e1, e2 not Int 0
- Case: Var x

A simple optimizer

```
type id = string
type exp = Int of int | Add of exp * exp | Var of id

type env
val lookup : env -> id -> int

let rec eval (env: env) (e: exp) : int =
match e with
  Int i -> i
| Add (e1, e2) -> (eval env e1) + (eval env e2)
| Var x -> lookup env x
```

```
let rec opt (e:exp) : exp =
  match e with
| Int i -> Int i
| Add (Int 0, e) -> opt e
| Add (e, Int 0) -> opt e
| Add (e1,e2) ->
   Add(opt e1, opt e2)
| Var x -> Var x
```

Theorem:
For all e : exp, eval (opt e) == eval e

Proof: By induction on the structure of expressions e : exp.

- Case: Int i
- Case: Add (Int 0, e2)
- Case: Add (e1, Int 0)
- Case: Add (e1, e2)   where e1, e2 not Int 0
- Case: Var x

A simple optimizer

```
type id = string
type exp = Int of int | Add of exp * exp | Var of id

type env
val lookup : env -> id -> int

let rec eval (env: env) (e: exp) : int =
match e with
  Int i -> i
| Add (e1, e2) -> (eval env e1) + (eval env e2)
| Var x -> lookup env x
```

```
let rec opt (e:exp) : exp =
  match e with
| Int i -> Int i
| Add (Int 0, e) -> opt e
| Add (e, Int 0) -> opt e
| Add (e1,e2) ->
   Add(opt e1, opt e2)
| Var x -> Var x
```

Theorem:
For all e : exp, eval (opt e) == eval e

Proof: By induction on the structure of expressions e : exp.

Case: e = Int i

eval (opt (Int i))

A simple optimizer

```
type id = string
type exp = Int of int | Add of exp * exp | Var of id

type env
val lookup : env -> id -> int

let rec eval (env: env) (e: exp) : int =
match e with
  Int i -> i
| Add (e1, e2) -> (eval env e1) + (eval env e2)
| Var x -> lookup env x
```

```
let rec opt (e:exp) : exp =
  match e with
| Int i -> Int i
| Add (Int 0, e) -> opt e
| Add (e, Int 0) -> opt e
| Add (e1,e2) ->
   Add(opt e1, opt e2)
| Var x -> Var x
```

Theorem:
For all e : exp, eval (opt e) == eval e

Proof: By induction on the structure of expressions e : exp.

Case: e = Int i

```
   eval (opt (Int i))   (RHS)
== eval (Int i)         (eval of opt)
```

A simple optimizer

```
type id = string
type exp = Int of int | Add of exp * exp | Var of id

type env
val lookup : env -> id -> int

let rec eval (env: env) (e: exp) : int =
match e with
  Int i -> i
| Add (e1, e2) -> (eval env e1) + (eval env e2)
| Var x -> lookup env x
```

```
let rec opt (e:exp) : exp =
  match e with
| Int i -> Int i
| Add (Int 0, e) -> opt e
| Add (e, Int 0) -> opt e
| Add (e1,e2) ->
   Add(opt e1, opt e2)
| Var x -> Var x
```

Theorem:
For all e : exp, eval (opt e) == eval e

Proof: By induction on the structure of expressions e : exp.

Case: e = Int i

```
   eval (opt (Int i))   (RHS)
== eval (Int i)         (eval of opt)
```

case done!
(we reached the LHS
from RHS)

A simple optimizer

```
type id = string
type exp = Int of int | Add of exp * exp | Var of id

type env
val lookup : env -> id -> int

let rec eval (env: env) (e: exp) : int =
match e with
  Int i -> i
| Add (e1, e2) -> (eval env e1) + (eval env e2)
| Var x -> lookup env x
```

```
let rec opt (e:exp) : exp =
  match e with
| Int i -> Int i
| Add (Int 0, e) -> opt e
| Add (e, Int 0) -> opt e
| Add (e1,e2) ->
   Add(opt e1, opt e2)
| Var x -> Var x
```

Theorem:

For all e : exp, eval (opt e) == eval e

Proof: By induction on the structure of expressions e : exp.

Case: e = Add(Int 0, e2)

IH: eval (opt e2) == eval e2

A simple optimizer

```
type id = string
type exp = Int of int | Add of exp * exp | Var of id

type env
val lookup : env -> id -> int

let rec eval (env: env) (e: exp) : int =
match e with
  Int i -> i
| Add (e1, e2) -> (eval env e1) + (eval env e2)
| Var x -> lookup env x
```

```
let rec opt (e:exp) : exp =
  match e with
| Int i -> Int i
| Add (Int 0, e) -> opt e
| Add (e, Int 0) -> opt e
| Add (e1,e2) ->
   Add(opt e1, opt e2)
| Var x -> Var x
```

Theorem:
For all e : exp, eval (opt e) == eval e

Proof: By induction on the structure of expressions e : exp.

Case: e = Add(Int 0, e2)   **IH:** eval (opt e2) == eval e2

eval (opt (Add(Int 0, e2))) (LHS)

A simple optimizer

```
type id = string
type exp = Int of int | Add of exp * exp | Var of id

type env
val lookup : env -> id -> int

let rec eval (env: env) (e: exp) : int =
match e with
  Int i -> i
| Add (e1, e2) -> (eval env e1) + (eval env e2)
| Var x -> lookup env x
```

```
let rec opt (e:exp) : exp =
  match e with
| Int i -> Int i
| Add (Int 0, e) -> opt e
| Add (e, Int 0) -> opt e
| Add (e1,e2) ->
   Add(opt e1, opt e2)
| Var x -> Var x
```

Theorem:
For all e : exp, eval (opt e) == eval e

Proof: By induction on the structure of expressions e : exp.

Case: e = Add(Int 0, e2)   **IH:** eval (opt e2) == eval e2

```
   eval (opt (Add(Int 0, e2)))  (LHS)
== eval (opt e2)                (by eval opt)
```

A simple optimizer

```
type id = string
type exp = Int of int | Add of exp * exp | Var of id

type env
val lookup : env -> id -> int

let rec eval (env: env) (e: exp) : int =
match e with
  Int i -> i
| Add (e1, e2) -> (eval env e1) + (eval env e2)
| Var x -> lookup env x
```

```
let rec opt (e:exp) : exp =
  match e with
| Int i -> Int i
| Add (Int 0, e) -> opt e
| Add (e, Int 0) -> opt e
| Add (e1,e2) ->
   Add(opt e1, opt e2)
| Var x -> Var x
```

Theorem:
For all e : exp, eval (opt e) == eval e

Proof: By induction on the structure of expressions e : exp.

Case: e = Add(Int 0, e2)   **IH:** eval (opt e2) == eval e2

```
   eval (opt (Add(Int 0, e2)))  (LHS)
== eval (opt e2)                (by eval opt)
== eval e2                      (by IH)
```

A simple optimizer

```
type id = string
type exp = Int of int | Add of exp * exp | Var of id

type env
val lookup : env -> id -> int

let rec eval (env: env) (e: exp) : int =
match e with
  Int i -> i
| Add (e1, e2) -> (eval env e1) + (eval env e2)
| Var x -> lookup env x
```

```
let rec opt (e:exp) : exp =
  match e with
| Int i -> Int i
| Add (Int 0, e) -> opt e
| Add (e, Int 0) -> opt e
| Add (e1,e2) ->
   Add(opt e1, opt e2)
| Var x -> Var x
```

Theorem:
For all e : exp, eval (opt e) == eval e

Proof: By induction on the structure of expressions e : exp.

Case: e = Add(Int 0, e2)

eval (Add(Int 0, e2)) (RHS)

```
   eval (opt (Add(Int 0, e2)))  (LHS)
== eval (opt e2)                (by eval opt)
== eval e2                      (by IH)
```

A simple optimizer

```
type id = string
type exp = Int of int | Add of exp * exp | Var of id

type env
val lookup : env -> id -> int

let rec eval (env: env) (e: exp) : int =
match e with
  Int i -> i
| Add (e1, e2) -> (eval env e1) + (eval env e2)
| Var x -> lookup env x
```

```
let rec opt (e:exp) : exp =
  match e with
| Int i -> Int i
| Add (Int 0, e) -> opt e
| Add (e, Int 0) -> opt e
| Add (e1,e2) ->
   Add(opt e1, opt e2)
| Var x -> Var x
```

Theorem:
For all e : exp, eval (opt e) == eval e

Proof: By induction on the structure of expressions e : exp.

Case: e = Add(Int 0, e2)

```
   eval (opt (Add(Int 0, e2)))  (LHS)
== eval (opt e2)                (by eval opt)
== eval e2                      (by IH)
```

```
   eval (Add(Int 0, e2))          (RHS)
== (eval(Int 0)) + (eval e2)      (eval)
```

A simple optimizer

```
type id = string
type exp = Int of int | Add of exp * exp | Var of id

type env
val lookup : env -> id -> int

let rec eval (env: env) (e: exp) : int =
match e with
  Int i -> i
| Add (e1, e2) -> (eval env e1) + (eval env e2)
| Var x -> lookup env x
```

```
let rec opt (e:exp) : exp =
   match e with
| Int i -> Int i
| Add (Int 0, e) -> opt e
| Add (e, Int 0) -> opt e
| Add (e1,e2) ->
    Add(opt e1, opt e2)
| Var x -> Var x
```

Theorem:
For all e : exp, eval (opt e) == eval e

Proof: By induction on the structure of expressions e : exp.

Case: e = Add(Int 0, e2)

```
   eval (opt (Add(Int 0, e2)))  (LHS)
== eval (opt e2)                (by eval opt)
== eval e2                      (by IH)
```

```
   eval (Add(Int 0, e2))           (RHS)
== (eval(Int 0)) + (eval e2)    (eval)
== 0 + eval e2                  (eval)
```

A simple optimizer

```
type id = string
type exp = Int of int | Add of exp * exp | Var of id

type env
val lookup : env -> id -> int

let rec eval (env: env) (e: exp) : int =
match e with
  Int i -> i
| Add (e1, e2) -> (eval env e1) + (eval env e2)
| Var x -> lookup env x
```

```
let rec opt (e:exp) : exp =
  match e with
| Int i -> Int i
| Add (Int 0, e) -> opt e
| Add (e, Int 0) -> opt e
| Add (e1,e2) ->
   Add(opt e1, opt e2)
| Var x -> Var x
```

Theorem:
For all e : exp, eval (opt e) == eval e

Proof: By induction on the structure of expressions e : exp.

Case: e = Add(Int 0, e2)

```
   eval (opt (Add(Int 0, e2)))  (LHS)
== eval (opt e2)                (by eval opt)
== eval e2                      (by IH)
```

```
   eval (Add(Int 0, e2))         (RHS)
== (eval(Int 0)) + (eval e2)    (eval)
== 0 + eval e2                  (eval)
== eval e2                      (math)
```

A simple optimizer

```
type id = string
type exp = Int of int | Add of exp * exp | Var of id

type env
val lookup : env -> id -> int

let rec eval (env: env) (e: exp) : int =
match e with
  Int i -> i
| Add (e1, e2) -> (eval env e1) + (eval env e2)
| Var x -> lookup env x
```

```
let rec opt (e:exp) : exp =
  match e with
| Int i -> Int i
| Add (Int 0, e) -> opt e
| Add (e, Int 0) -> opt e
| Add (e1,e2) ->
   Add(opt e1, opt e2)
| Var x -> Var x
```

Theorem:
For all e : exp, eval (opt e) == eval e

Proof: By induction on the structure of expressions e : exp.

Case: e = Add(Int 0, e2)

```
   eval (opt (Add(Int 0, e2)))  (LHS)
== eval (opt e2)                (by eval opt)
== eval e2                      (by IH)
```

```
   eval (Add(Int 0, e2))           (RHS)
== (eval(Int 0)) + (eval e2)       (eval)
== 0 + eval e2                     (eval)
== eval e2                         (math)
```

A simple optimizer

```
type id = string
type exp = Int of int | Add of exp * exp | Var of id

type env
val lookup : env -> id -> int

let rec eval (env: env) (e: exp) : int =
match e with
  Int i -> i
| Add (e1, e2) -> (eval env e1
| Var x -> lookup env x
```

```
let rec opt (e:exp) : exp =
  match e with
| Int i -> Int i
| Add (Int 0, e) -> opt e
| Add (e, Int 0) -> opt e
| Add (e1,e2) ->
   Add(opt e1, opt e2)
| Var x -> Var x
```

Theorem:
xp, eval (opt e) == eval e

case done!
(we showed the
LHS == RHS)

Proof: By induction on the structure o essions e : exp.

Case: e = Add(Int 0, e2)

```
   eval (opt (Add(Int 0, e2)))  (LHS)
== eval (opt e2)                (by eval opt)
== eval e2                      (by IH)
```

```
   eval (Add(Int 0, e2))          (RHS)
== (eval(Int 0)) + (eval e2)      (eval)
== 0 + eval e2                    ( al)
== eval e2                        (math)
```

A simple optimizer

```
type id = string
type exp = Int of int | Add of exp * exp | Var of id

type env
val lookup : env -> id -> int

let rec eval (env: env) (e: exp) : int =
match e with
  Int i -> i
| Add (e1, e2) -> (eval env e1) + (eval env e2)
| Var x -> lookup env x
```

```
let rec opt (e:exp) : exp =
  match e with
| Int i -> Int i
| Add (Int 0, e) -> opt e
| Add (e, Int 0) -> opt e
| Add (e1,e2) ->
   Add(opt e1, opt e2)
| Var x -> Var x
```

Theorem:
For all e : exp, eval (opt e) == eval e

Proof: By induction on the structure of expressions e : exp.

Case: e = Add(e2, Int 0)

IH: eval (opt e2) == eval e2

A simple optimizer

```
type id = string
type exp = Int of int | Add of exp * exp | Var of id

type env
val lookup : env -> id -> int

let rec eval (env: env) (e: exp) : int =
match e with
  Int i -> i
| Add (e1, e2) -> (eval env e1) + (eval env e2)
| Var x -> lookup env x
```

```
let rec opt (e:exp) : exp =
  match e with
| Int i -> Int i
| Add (Int 0, e) -> opt e
| Add (e, Int 0) -> opt e
| Add (e1,e2) ->
   Add(opt e1, opt e2)
| Var x -> Var x
```

Theorem:
For all e : exp, eval (opt e) == eval e

Proof: By induction on the structure of expressions e : exp.

Case: e = Add(e2, Int 0) **IH:** eval (opt e2) == eval e2

Very similar to the last case – go through it yourself for practice.

A simple optimizer

```
type id = string
type exp = Int of int | Add of exp * exp | Var of id

type env
val lookup : env -> id -> int

let rec eval (env: env) (e: exp) : int =
match e with
  Int i -> i
| Add (e1, e2) -> (eval env e1) + (eval env e2)
| Var x -> lookup env x
```

```
let rec opt (e:exp) : exp =
  match e with
| Int i -> Int i
| Add (Int 0, e) -> opt e
| Add (e, Int 0) -> opt e
| Add (e1,e2) ->
   Add(opt e1, opt e2)
| Var x -> Var x
```

Theorem:
For all e : exp, eval (opt e) == eval e

Proof: By induction on the structure of expressions e : exp.

Case: e = Add(e1, e2)

IH1: eval (opt e1) == eval e1
IH2: eval (opt e2) == eval e2

A simple optimizer

```
type id = string
type exp = Int of int | Add of exp * exp | Var of id

type env
val lookup : env -> id -> int

let rec eval (env: env) (e: exp) : int =
match e with
  Int i -> i
| Add (e1, e2) -> (eval env e1) + (eval env e2)
| Var x -> lookup env x
```

```
let rec opt (e:exp) : exp =
  match e with
| Int i -> Int i
| Add (Int 0, e) -> opt e
| Add (e, Int 0) -> opt e
| Add (e1,e2) ->
   Add(opt e1, opt e2)
| Var x -> Var x
```

Theorem:
For all e : exp, eval (opt e) == eval e

Proof: By induction on the structure of expressions e : exp.

Case: e = Add(e1, e2)

eval (opt (Add(e1, e2)))   (LHS)

A simple optimizer

```
type id = string
type exp = Int of int | Add of exp * exp | Var of id

type env
val lookup : env -> id -> int

let rec eval (env: env) (e: exp) : int =
match e with
  Int i -> i
| Add (e1, e2) -> (eval env e1) + (eval env e2)
| Var x -> lookup env x
```

```
let rec opt (e:exp) : exp =
  match e with
| Int i -> Int i
| Add (Int 0, e) -> opt e
| Add (e, Int 0) -> opt e
| Add (e1,e2) ->
   Add(opt e1, opt e2)
| Var x -> Var x
```

Theorem:
For all e : exp, eval (opt e) == eval e

Proof: By induction on the structure of expressions e : exp.

Case: e = Add(e1, e2)

```
   eval (opt (Add(e1, e2)))       (LHS)
== eval (Add (opt e1, opt e2))   (by eval opt)
```

A simple optimizer

```
type id = string
type exp = Int of int | Add of exp * exp | Var of id

type env
val lookup : env -> id -> int

let rec eval (env: env) (e: exp) : int =
match e with
  Int i -> i
| Add (e1, e2) -> (eval env e1) + (eval env e2)
| Var x -> lookup env x
```

```
let rec opt (e:exp) : exp =
  match e with
| Int i -> Int i
| Add (Int 0, e) -> opt e
| Add (e, Int 0) -> opt e
| Add (e1,e2) ->
   Add(opt e1, opt e2)
| Var x -> Var x
```

Theorem:
For all e : exp, eval (opt e) == eval e

Proof: By induction on the structure of expressions e : exp.

Case: e = Add(e1, e2)

```
   eval (opt (Add(e1, e2)))       (LHS)
== eval (Add (opt e1, opt e2))   (by eval opt)
== eval (opt e1) + eval (opt e2) (by eval eval)
```

A simple optimizer

```
type id = string
type exp = Int of int | Add of exp * exp | Var of id

type env
val lookup : env -> id -> int

let rec eval (env: env) (e: exp) : int =
match e with
  Int i -> i
| Add (e1, e2) -> (eval env e1) + (eval env e2)
| Var x -> lookup env x
```

```
let rec opt (e:exp) : exp =
  match e with
| Int i -> Int i
| Add (Int 0, e) -> opt e
| Add (e, Int 0) -> opt e
| Add (e1,e2) ->
   Add(opt e1, opt e2)
| Var x -> Var x
```

Theorem:
For all e : exp, eval (opt e) == eval e

Proof: By induction on the structure of expressions e : exp.

Case: e = Add(e1, e2)

eval (Add(e1, e2)) (RHS)

```
   eval (opt (Add(e1, e2)))      (LHS)
== eval (Add (opt e1, opt e2))   (by eval opt)
== eval (opt e1) + eval (opt e2) (by eval eval)
```

A simple optimizer

```
type id = string
type exp = Int of int | Add of exp * exp | Var of id

type env
val lookup : env -> id -> int

let rec eval (env: env) (e: exp) : int =
match e with
  Int i -> i
| Add (e1, e2) -> (eval env e1) + (eval env e2)
| Var x -> lookup env x
```

```
let rec opt (e:exp) : exp =
  match e with
| Int i -> Int i
| Add (Int 0, e) -> opt e
| Add (e, Int 0) -> opt e
| Add (e1,e2) ->
   Add(opt e1, opt e2)
| Var x -> Var x
```

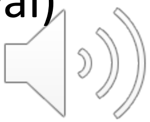
Theorem:
For all e : exp, eval (opt e) == eval e

Proof: By induction on the structure of expressions e : exp.

Case: e = Add(e1, e2)

```
   eval (opt (Add(e1, e2)))      (LHS)
== eval (Add (opt e1, opt e2))   (by eval opt)
== eval (opt e1) + eval (opt e2) (by eval eval)
```

```
   eval (Add(e1, e2))          (RHS)
== (eval e1) + (eval e2)  (eval)
```



A simple optimizer

```
type id = string
type exp = Int of int | Add of exp * exp | Var of id

type env
val lookup : env -> id -> int

let rec eval (env: env) (e: exp) : int =
match e with
  Int i -> i
| Add (e1, e2) -> (eval env e1) + (eval env e2)
| Var x -> lookup env x
```

```
let rec opt (e:exp) : exp =
  match e with
| Int i -> Int i
| Add (Int 0, e) -> opt e
| Add (e, Int 0) -> opt e
| Add (e1,e2) ->
   Add(opt e1, opt e2)
| Var x -> Var x
```

Theorem:

For all e : exp, eval (opt e) == eval e

Proof: By induction on the structure of expressions e : exp.

Case: e = Add(e1, e2)

```
   eval (opt (Add(e1, e2)))       (LHS)
== eval (Add (opt e1, opt e2))   (by eval opt)
== eval (opt e1) + eval (opt e2) (by eval eval)
```

```
   eval (Add(e1, e2))           (RHS)
== (eval e1) + (eval e2)  (eval)
== eval (opt e1) + eval (opt  )
              (by IH1 and IH2)
```

A simple optimizer

```
type id = string
type exp = Int of int | Add of exp * exp | Var of id

type env
val lookup : env -> id -> int

let rec eval (env: env) (e: exp) : int =
match e with
  Int i -> i
| Add (e1, e2) -> (eval env e1) + (e
| Var x -> lookup env x
```

```
let rec opt (e:exp) : exp =
  match e with
| Int i -> Int i
| Add (Int 0, e) -> opt e
| Add (e, Int 0) -> opt e
| Add (e1,e2) ->
   Add(opt e1, opt e2)
| Var x -> Var x
```

(opt e) == eval e

case done!
(we showed the
LHS == RHS)

Proof: By induction on the structure of expre[...]ns e : exp.

Case: e = Add(e1, e2)

```
    eval (opt (Add(e1, e2)))       (LHS)
== eval (Add (opt e1, opt e2))    (by eval opt)
== eval (opt e1) + eval (opt e2) (by eval eval)
```

```
   eval (Add(e1, e2))          (RHS)
== (eval e1) + (eval e2)  (eval)
== eval (opt e1) + eval (opt e2)
              (by IH1 and IH2)
```

A simple optimizer

```
type id = string
type exp = Int of int | Add of exp * exp | Var of id

type env
val lookup : env -> id -> int

let rec eval (env: env) (e: exp) : int =
match e with
  Int i -> i
| Add (e1, e2) -> (eval env e1) + (eval env e2)
| Var x -> lookup env x
```

```
let rec opt (e:exp) : exp =
  match e with
| Int i -> Int i
| Add (Int 0, e) -> opt e
| Add (e, Int 0) -> opt e
| Add (e1,e2) ->
   Add(opt e1, opt e2)
| Var x -> Var x
```

Theorem:
For all e : exp, eval (opt e) == eval e

Proof: By induction on the structure of expressions e : exp.

Case: e = Var x

No IH to use because there are no sub-structures with type exp!

A simple optimizer

```
type id = string
type exp = Int of int | Add of exp * exp | Var of id

type env
val lookup : env -> id -> int

let rec eval (env: env) (e: exp) : int =
match e with
  Int i -> i
| Add (e1, e2) -> (eval env e1) + (eval env e2)
| Var x -> lookup env x
```

```
let rec opt (e:exp) : exp =
  match e with
| Int i -> Int i
| Add (Int 0, e) -> opt e
| Add (e, Int 0) -> opt e
| Add (e1,e2) ->
   Add(opt e1, opt e2)
| Var x -> Var x
```

Theorem:
For all e : exp, eval (opt e) == eval e

Proof: By induction on the structure of expressions e : exp.

Case: e = Var x

```
   eval (opt (Var x))      (LHS)
== eval (Var x)            (by eval opt)
```

A simple optimizer

```
type id = string
type exp = Int of int | Add of exp * exp | Var of id

type env
val lookup : env -> id -> int

let rec eval (env: env) (e: exp) : int =
match e with
  Int i -> i
| Add (e1, e2) -> (eval env e1) + (eval env e2)
| Var x -> lookup env x
```

```
let rec opt (e:exp) : exp =
  match e with
| Int i -> Int i
| Add (Int 0, e) -> opt e
| Add (e, Int 0) -> opt e
| Add (e1,e2) ->
   Add(opt e1, opt e2)
| Var x -> Var x
```

Theorem:
For all e : exp, eval (opt e) == eval e

Proof: By induction on the structure of e

Case: e = Var x

```
    eval (opt (Var x))      (LHS)
== eval (Var x)             (by eval opt)
```

case done!
(we showed the
LHS == RHS)

A simple optimizer

```
type id = string
type exp = Int of int | Add of exp * exp | Var of id

type env
val lookup : e          -> int

let rec eval (env: e
match e
  Int i ->
| Add (e1, e2
| Var x -> lookup
```

```
let rec opt (e:exp) : exp =
  match e with
| Int i -> Int i
| Add (Int 0, e) -> opt e
| Add (e, Int 0) -> opt e
|       e1,e2) ->
        (opt e1, opt e2)
|       Var x
```

al (opt e) == eval e

PROOF DONE!!!

e!

the

LHS   )

Proof: P

Case: e = Var x

    eval (opt (Var x))   (LHS
== eval (Var x)   (by e   opt)

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 ...