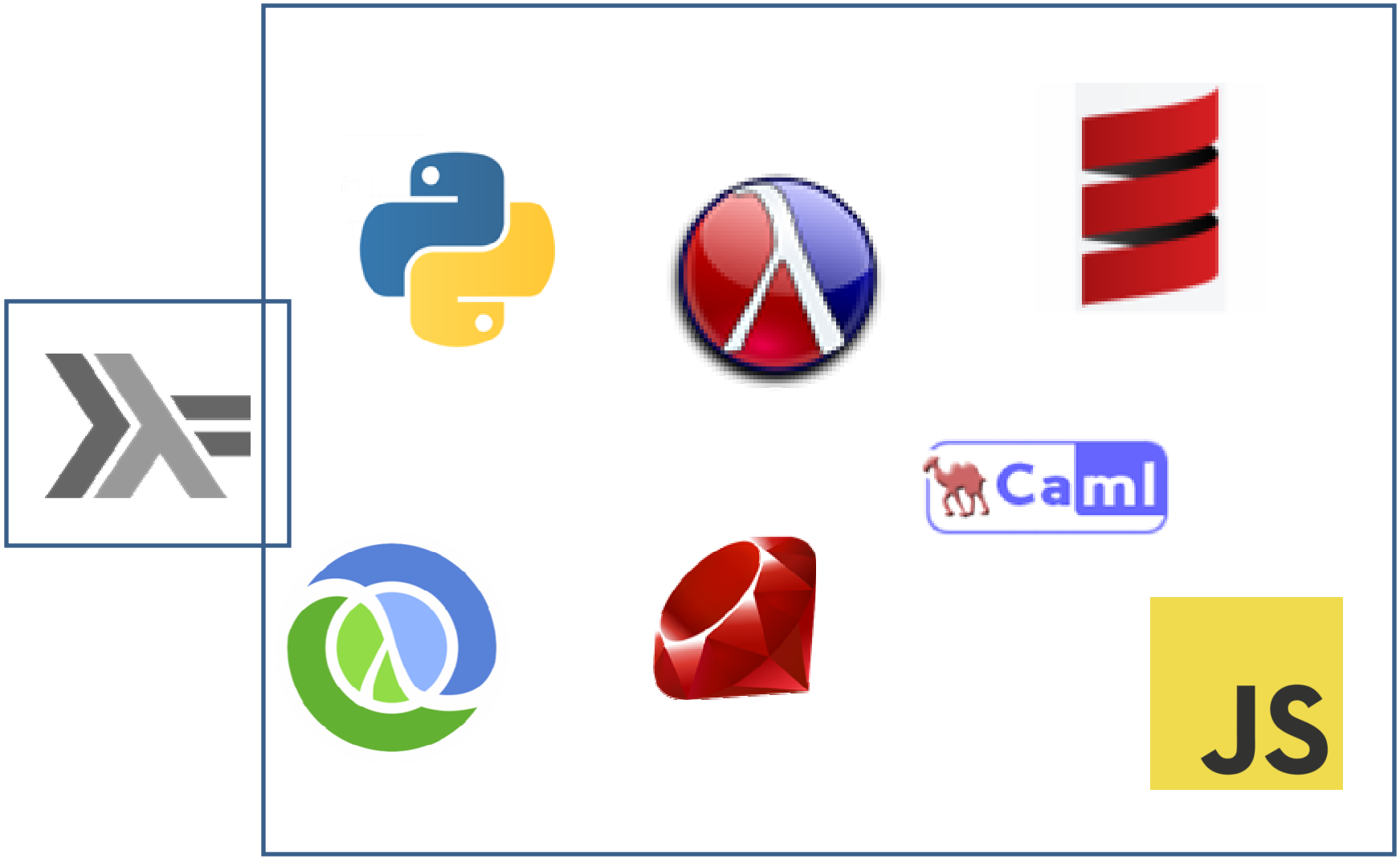
Laziness By Need

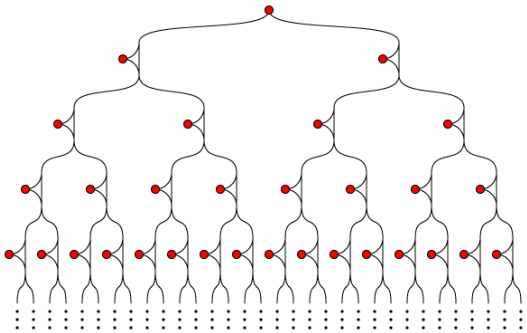
Stephen Chang
Northeastern University
10/15/2012

Everyone's Lazy!



Laziness Advantages

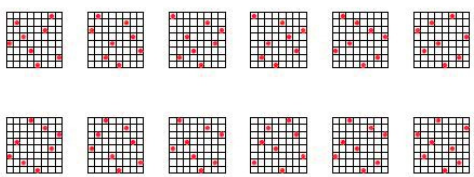
Efficient representation of large/infinite data structures



generation

validation

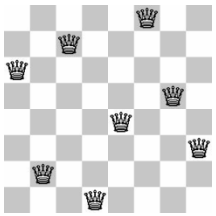
Modularity



all possible solutions

...

Rules



Same Fringe

"Two binary trees have the same fringe if they have exactly the same leaves, reading from left to right."

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

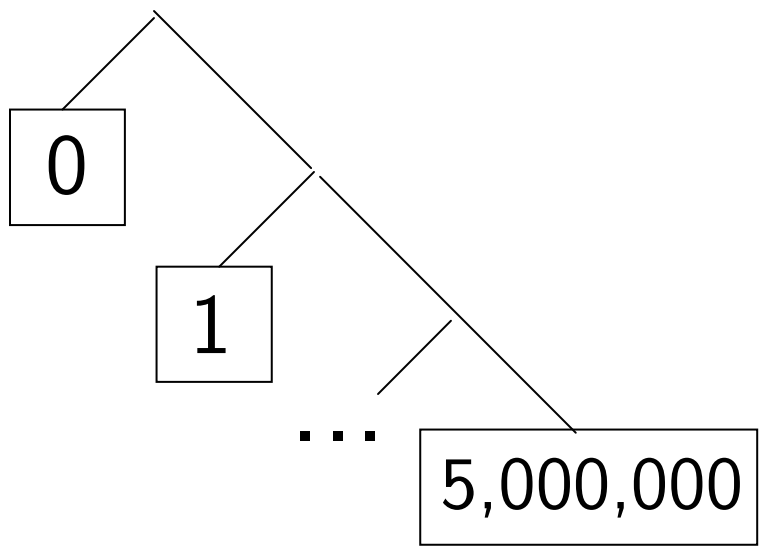
```
let same_fringe tree1 tree2 =
    (flatten tree1) == (flatten tree2)
```

Same Fringe

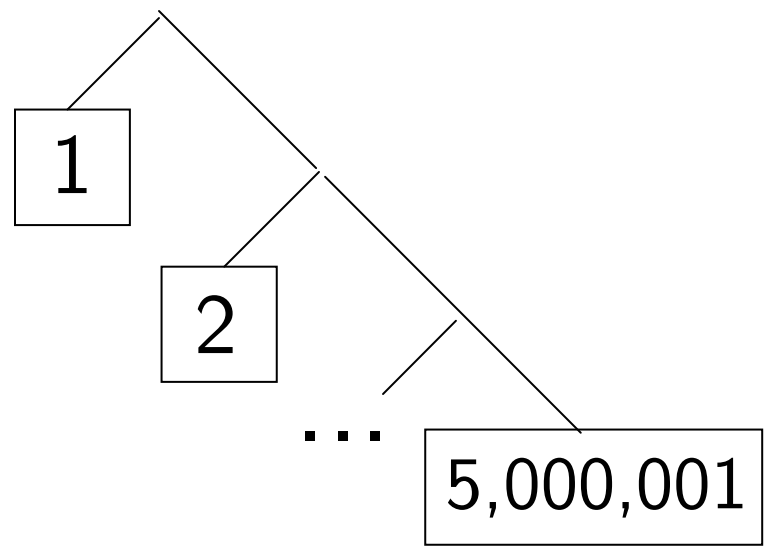
```
let flatten t = _flatten_ t []

let rec _flatten_ t flattened =
  match t with
  | Leaf x       -> x::flattened
  | Node(t1,t2) ->
    _flatten_ t1 (_flatten_ t2 flattened)
```

Same Fringe (eager)

```
let tree1 =
```




```
let tree2 =
```




```
same_fringe tree1 tree2 => false
```

0m13.363s

Same Fringe (with streams)

```
type 'a stream =
    Nil
  | LCons of 'a * 'a stream lazy_t
```

Same Fringe (with streams)

```
type 'a stream =
      Nil | LCons of 'a * 'a stream lazy_t

let flatten t = _flatten_ t Nil

let rec _flatten_ t flattened =
  match t with
  | Leaf x       -> LCons(x, lazy flattened)
  | Node(t1,t2) ->
    _flatten_ t1 (_flatten_ t2 flattened)
```

Same Fringe (with streams)

```
type 'a stream =
      Nil | LCons of 'a * 'a stream lazy_t

let rec stream_eq s1 s2 =
  match(force s1,force s2) with
  | (Nil,Nil) -> true
  | (LCons(x1,xs1),LCons(x2,xs2)) ->
      x1=x2 && stream_eq xs1 xs2
  | _ -> false
```

Same Fringe (with streams)

```
let same_fringe tree1 tree2 =
    stream_eq (lazy(flatten tree1))
              (lazy(flatten tree2))
```

```
same_fringe tree1 tree2 => false
```

0m17.277s

(with lazy trees)
0m36.905s

Same Fringe

```
let flatten t = _flatten_ t Nil

let rec _flatten_ t flattened =
  match t with
  | Leaf x       -> LCons(x, lazy flattened)
  | Node(t1,t2) ->
    _flatten_ t1 (_flatten_ t2 flattened)
```

Same Fringe

```
let flatten t = _flatten_ t Nil

let rec _flatten_ t flattened =
  match t with
  | Leaf x       -> LCons(x, lazy flattened)
  | Node(t1,t2) ->
    _flatten_ t1 lazy(_flatten_ t2 flattened)
```

Same Fringe

```
let flatten t = _flatten_ t Nil

let rec _flatten_ t flattened =
  match t with
  | Leaf x       -> LCons(x, lazy flattened)
  | Node(t1,t2) ->
    _flatten_ t1 lazy(_flatten_ t2 flattened)
```

Same Fringe

```
let flatten t = _flatten_ t (lazy Nil)

let rec _flatten_ t flattened =
  match t with
  | Leaf x       -> LCons(x, flattened)
  | Node(t1,t2) ->
    _flatten_ t1 lazy(_flatten_ t2 flattened)
```

Same Fringe (properly lazy)

```
same_fringe tree1 tree2 => false
```

0m0.002s

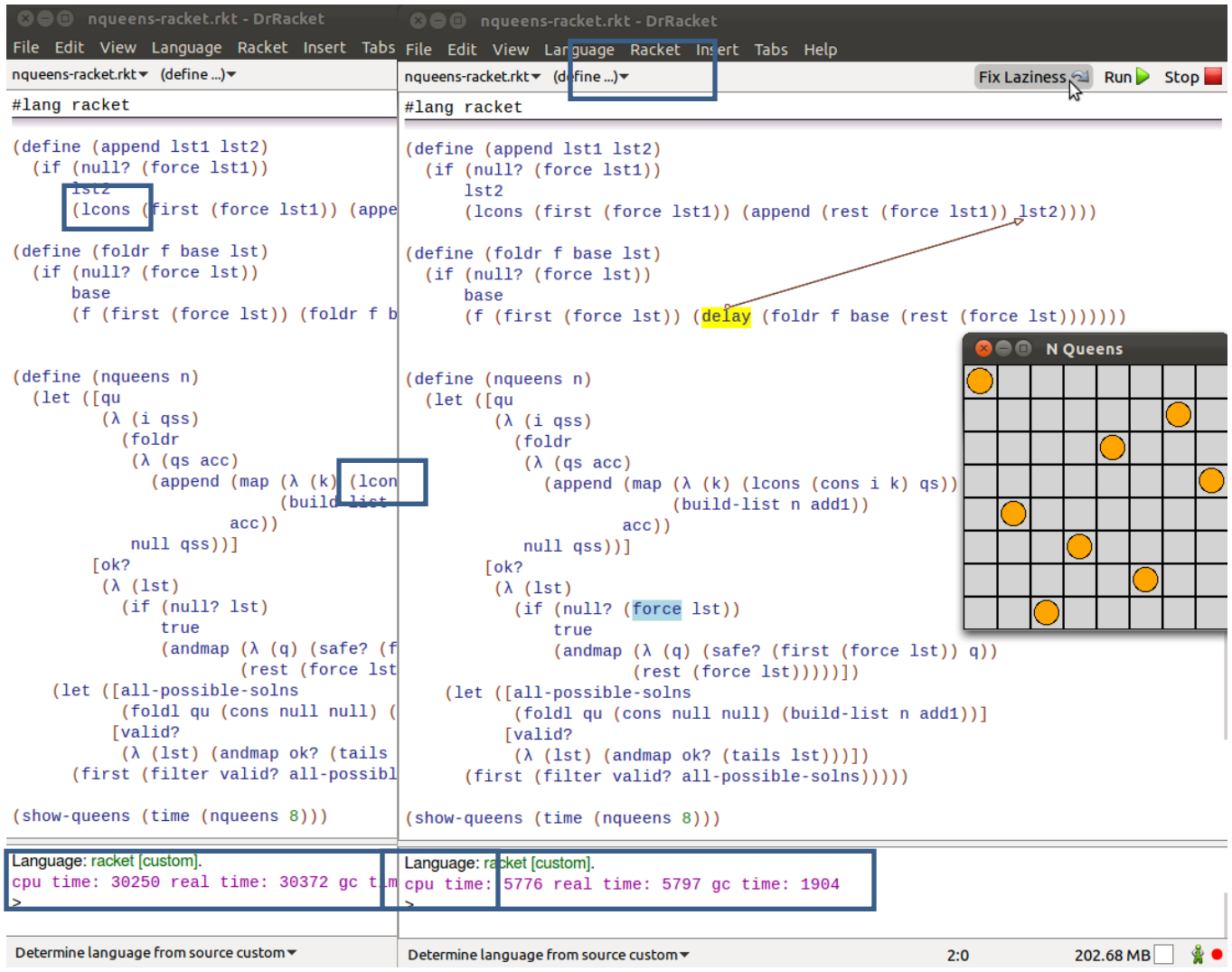
Problem Summary

- Use of lazy data structures doesn't automatically make the program behave lazily.
- Laziness is non-local: when used in one part of the program, it propagates to other parts.

Thesis Statement

Semi-automatic tool support can help programmers exploit laziness in strict contexts.

```
lcons x y
≡
cons x (delay y)
```



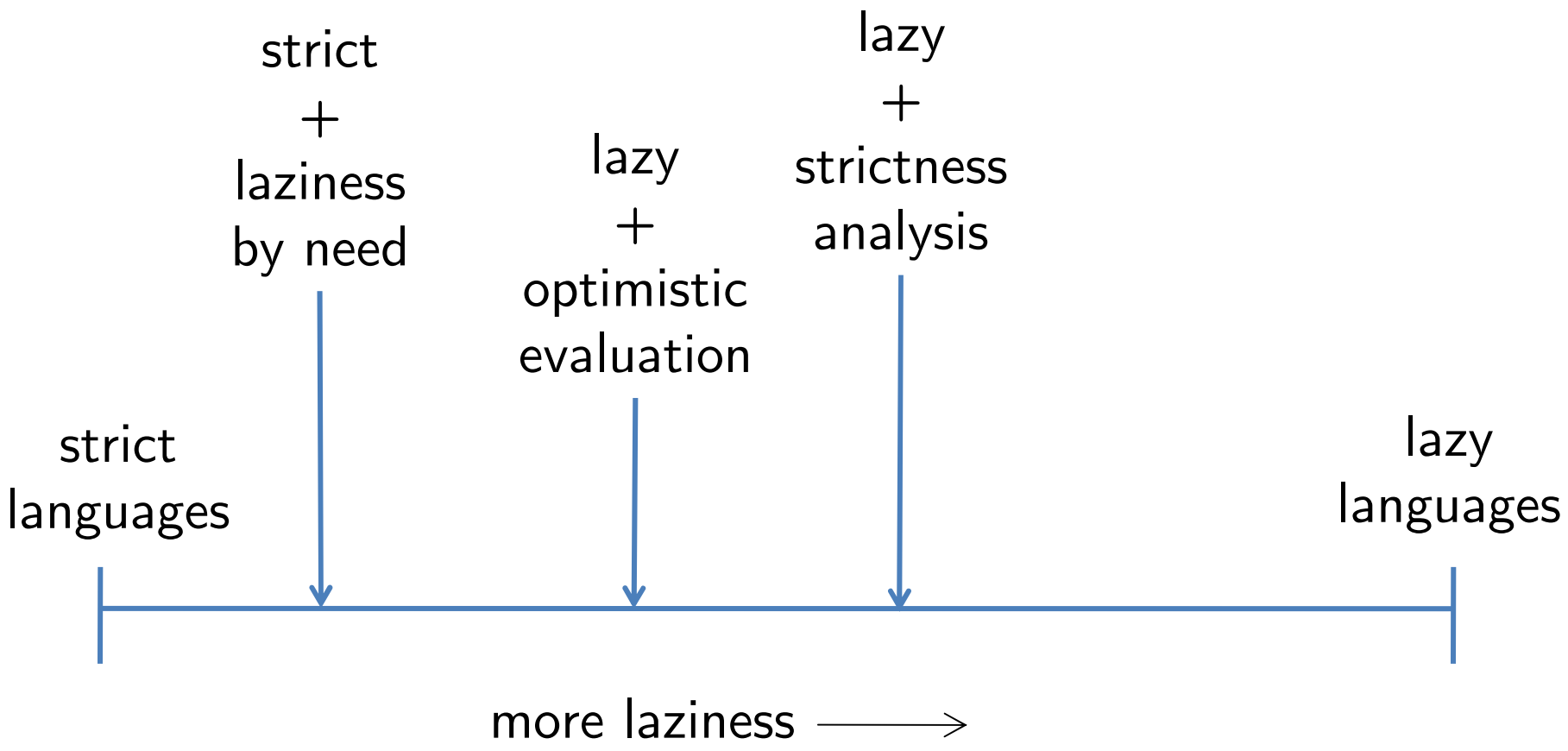
Related Work

- Strictness Analysis [Mycroft1981]
- Cheap Eagerness [Faxen1995]
- Eager Haskell [Maessen2002]
- Optimistic Evaluation [Ennals&Jones2003]

“Most of these thunks are not needed”

[Ennals&Jones2003]

[Maessen2002] [MHOV2012] [Schauser&Goldstein1995]



control flow analysis

+

laziness flow analysis

control flow analysis

$$\widehat{\rho} \in \ell \cup x \to \mathcal{P}(\widehat{v})$$

$$+$$

laziness flow analysis

$$\widehat{\mathcal{D}} \in \mathcal{P}(\ell)$$

$$\widehat{\mathcal{S}} \in \mathcal{P}(\ell)$$

$$\widehat{\mathcal{F}} \in \mathcal{P}(\ell)$$

$\widehat{\mathcal{D}} =$ arguments that reach a lazy construct

$\widehat{\mathcal{S}} =$ arguments that reach a strict context

$\widehat{\mathcal{F}} =$ expressions to force

Transformation

- Delay all $\ell : \ell \in \widehat{\mathcal{D}}, \ell \notin \widehat{\mathcal{S}}$

- Force all $\ell : \ell \in \widehat{\mathcal{F}}$

```
let rec _flatten_ t flattened =
  match t with
  | Leaf x       -> LCons(x, lazy flattened)
  | Node(t1,t2) ->
    _flatten_ t1 (_flatten_ t2 flattened)
```

Abstract value

$$(\texttt{arg}\ \ell)$$

tracks flow of functions arguments.

Analysis specified with constraint rules:

$$(\widehat{\rho}, \widehat{\mathcal{D}}, \widehat{\mathcal{S}}, \widehat{\mathcal{F}}) \models e \text{ iff } c_1, \ldots, c_n$$

Read: Sets $\widehat{\rho}, \widehat{\mathcal{D}}, \widehat{\mathcal{S}}, \widehat{\mathcal{F}}$ approximate expression e if and only if constraints $c_1, \ldots, c_n$ hold.

$$\widehat{\rho} \models (e_f^{\ell_f}\ e_1^{\ell_1} \ldots)^{\ell} \text{ iff} \qquad [app]$$

$$\boxed{\widehat{\rho} \models e_f^{\ell_f} \ \wedge\ \widehat{\rho} \models e_1^{\ell_1} \ \wedge\ \ldots\ \wedge}$$

$$\boxed{(\forall \lambda(x_1 \ldots).\ell_0 \in \widehat{\rho}(\ell_f) :}$$

$$\boxed{\widehat{\rho}(\ell_1) \subseteq \widehat{\rho}(x_1) \ \wedge\ \ldots} \wedge \boxed{\widehat{\rho}(\ell_0) \subseteq \widehat{\rho}(\ell))}$$

$$\widehat{\rho} \models (e_f^{\ell_f}\ e_1^{\ell_1} \ldots)^{\ell} \ \text{iff} \qquad [app]$$

$$\widehat{\rho} \models e_f^{\ell_f} \ \wedge \ \widehat{\rho} \models e_1^{\ell_1} \ \wedge \ \ldots \ \wedge$$

$$(\forall \lambda(x_1 \ldots).\ell_0 \in \widehat{\rho}(\ell_f) :$$

$$\widehat{\rho}(\ell_1) \subseteq \widehat{\rho}(x_1) \ \wedge \ \ldots \ \wedge \ \widehat{\rho}(\ell_0) \subseteq \widehat{\rho}(\ell)$$

$$\boxed{\wedge \quad (\mathtt{arg}\ \ell_1) \in \widehat{\rho}(x_1) \ \wedge \ \ldots}$$

$$(\widehat{\rho}, \widehat{\mathcal{D}}) \models (\texttt{lcons}\ e_1^{\ell_1}\ e_2^{\ell_2})^{\ell} \ \text{ iff} \qquad [\mathit{lcons}]$$

$$\boxed{(\widehat{\rho}, \widehat{\mathcal{D}}) \models e_1^{\ell_1} \wedge (\widehat{\rho}, \widehat{\mathcal{D}}) \models e_2^{\ell_2}} \wedge \boxed{(\texttt{lcons}\ \ell_1\ \ell_2) \in \widehat{\rho}(\ell)}$$

$$\wedge\ \boxed{(\forall x \in fv(e_2)} : \boxed{(\forall (\texttt{arg}\ \ell_3) \in \widehat{\rho}(x)} : \boxed{\ell_3 \in \widehat{\mathcal{D}}}))$$

strict contexts:

contexts where a thunk should not appear

examples:

- arguments to primitives
- `if` test expression
- function position in an application

$$(\widehat{\rho}, \widehat{\mathcal{D}}, \widehat{\mathcal{S}}, \widehat{\mathcal{F}}) \models S[e^{\ell}] \;\; \text{iff} \;\; \ldots \;\; \wedge \qquad [\textit{strict}]$$

$$\boxed{(\forall(\texttt{arg}\ \ell_1) \in \widehat{\rho}(\ell) : \ell_1 \in \widehat{\mathcal{S}})} \quad \wedge$$

$$\boxed{(\exists \texttt{delay} \in \widehat{\rho}(\ell) \Rightarrow \ell \in \widehat{\mathcal{F}})}$$

Future Work

1) Additional language features.
   i. Types
   ii. Mutable State

2) Improve interactivity of the tool.

3) Effect of improved analysis precision on laziness flow calculations.

```
type 'a stream = Nil | LCons of 'a * 'a stream lazy_t

let rec foldr
  (f:'a->'b->'b) (base:'b)
  (lst:'a stream lazy_t):'b =
    match force lst with
    | Nil -> base
    | LCons(x,xs) -> f x (foldr f base xs)

let f (x:'a) (acc:'a stream):'a stream =
  if ... then acc else LCons(x,lazy acc)

let (x:'a stream) = foldr f Nil _some_stream_
```

```
type 'a stream = Nil | LCons of 'a * 'a stream lazy_t

let rec foldr
  (f:'a->'b->'b) (base:'b)
  (lst:'a stream lazy_t):'b =
    match force lst with
    | Nil -> base
    | LCons(x,xs) -> f x lazy(foldr f base xs)

let f (x:'a) (acc:'a stream):'a stream =
  if ... then acc else LCons(x,lazy acc)

let (x:'a stream) = foldr f Nil _some_stream_
```

```
type 'a stream = Nil | LCons of 'a * 'a stream lazy_t

let rec foldr
  (f:'a->'b->'b) (base:'b)
  (lst:'a stream lazy_t):'b =
    match force lst with
    | Nil -> base
    | LCons(x,xs) -> f x (foldr f base xs)

let f (x:'a) (acc:'a stream):'a stream =
  if ... then acc else LCons(x,lazy acc)

let (x:'a stream) = foldr f Nil _some_stream_
```

```
type 'a stream = Nil | LCons of 'a * 'a stream lazy_t

let rec foldr
  (f:'a->'b->'b) (base:'b)
  (lst:'a stream lazy_t):'b =
    match force lst with
    | Nil -> base
    | LCons(x,xs) -> f x lazy(foldr f base xs)

let f (x:'a) (acc:'a stream):'a stream =
  if ... then acc else LCons(x,lazy acc)

let (x:'a stream) = foldr f Nil _some_stream_
```

```
type 'a stream = Nil | LCons of 'a * 'a stream lazy_t

let rec foldr
  (f:'a->'b lazy_t->'b) (base:'b)
  (lst:'a stream lazy_t):'b =
    match force lst with
    | Nil -> base
    | LCons(x,xs) -> f x lazy(foldr f base xs)

let f (x:'a) (acc:'a stream lazy_t):'a stream =
  if ... then acc else LCons(x,lazy acc)

let (x:'a stream) = foldr f Nil _some_stream_
```

```
type 'a stream = Nil | LCons of 'a * 'a stream lazy_t

let rec foldr
  (f:'a->'b lazy_t->'b lazy_t) (base:'b)
  (lst:'a stream lazy_t):'b =
    match force lst with
    | Nil -> base
    | LCons(x,xs) -> f x lazy(foldr f base xs)

let f (x:'a) (acc:'a stream lazy_t):'a stream lazy_t =
  if ... then acc else lazy(LCons(x,lazy acc))

let (x:'a stream) = foldr f Nil _some_stream_
```

```
type 'a stream = Nil | LCons of 'a * 'a stream lazy_t

let rec foldr
  (f:'a->'b lazy_t->'b lazy_t) (base:'b)
  (lst:'a stream lazy_t):'b =
    match force lst with
    | Nil -> base
    | LCons(x,xs) -> f x lazy(foldr f base xs)

let f (x:'a) (acc:'a stream lazy_t):'a stream lazy_t =
  if ... then acc else lazy(LCons(x,acc))

let (x:'a stream) = foldr f Nil _some_stream_
```

```
type 'a stream = Nil | LCons of 'a * 'a stream lazy_t

let rec foldr
  (f:'a->'b lazy_t->'b lazy_t) (base:'b)
  (lst:'a stream lazy_t):'b lazy_t =
    match force lst with
    | Nil -> lazy base
    | LCons(x,xs) -> f x lazy(foldr f base xs)

let f (x:'a) (acc:'a stream lazy_t):'a stream lazy_t =
  if ... then acc else lazy(LCons(x,acc))

let (x:'a stream lazy_t) = foldr f Nil _some_stream_
```

Future Work

1) Additional language features.
   i. Types
   ii. Mutable State

2) Improve interactivity of the tool.

3) Effect of improved analysis precision on laziness flow calculations.

```
(define (trace* stx expander)
  (let ([events (expand/events stx expander)])
      ...
       (delay (parse-derivation
               (events->token-generator events)))))
```

!!!

Future Work

1) Additional language features.
   i. Types
   ii. Mutable State

2) Improve interactivity of the tool.

3) Effect of improved analysis precision on laziness flow calculations.

nqueens-racket.rkt - DrRacket

File Edit View Language Racket Insert Tabs Help

nqueens-racket.rkt (define ...) Fix Laziness Run Stop

```
#lang racket

(define (append lst1 lst2)
  (if (null? (force lst1))
      lst2
      (lcons (first (force lst1)) (append (rest (force lst1)) lst2))))

(define (foldr f base lst)
  (if (null? (force lst))
      base
      (f (first (force lst)) (delay (foldr f base (rest (force lst)))))))


(define (nqueens n)
  (let ([qu
         (λ (i qss)
           (foldr
            (λ (qs acc)
              (append (map (λ (k) (lcons (cons i k) qs))
                           (build-list n add1))
                      acc))
            null qss))]
        [ok?
         (λ (lst)
           (if (null? (force lst))
               true
               (andmap (λ (q) (safe? (first (force lst)) q))
                       (rest (force lst)))))])
    (let ([all-possible-solns
           (foldl qu (cons null null) (build-list n add1))]
          [valid?
           (λ (lst) (andmap ok? (tails lst)))])
      (first (filter valid? all-possible-solns)))))

(show-queens (time (nqueens 8)))
```

?

?

N Queens

Language: racket [custom].

```
cpu time: 5776 real time: 5797 gc time: 1904
>
```

Determine language from source custom 2:0 202.68 MB

Future Work

1) Additional language features.
   i. Types
   ii. Mutable State

2) Improve interactivity of the tool.

3) Effect of improved analysis precision on laziness flow calculations.

Thesis Statement

Semi-automatic tool support can help programmers exploit laziness in strict contexts.