Laziness By Need

Stephen Chang

Northeastern University

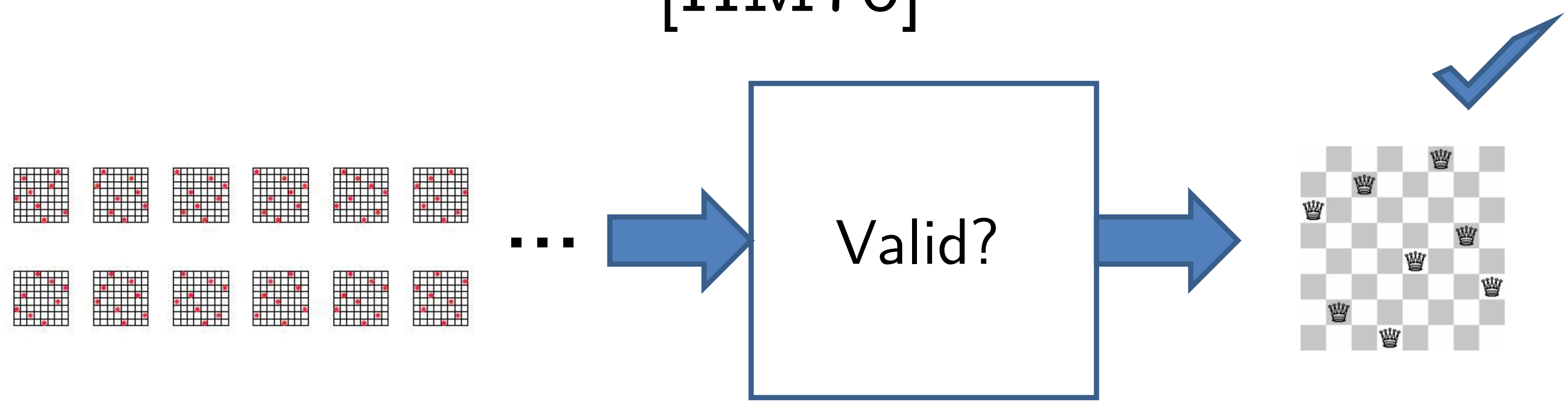
3/19/2013

ESOP 2013, Rome, Italy

"the most powerful tool for modularization
... the key to successful programming"
[Hughes90]

Laziness is great.

"pragmatically important because it enables
the producer-consumer programming style"
[HM76]



"lazy programs can exhibit astonishing poor space behavior" [HHPJW07]

"monumentally difficult to reason about time" [Harper11]

Or is it?

"in a lazy language, it's much more difficult to predict the order of evaluation" [PJ11]

I want the good
without the bad.

Solution: strict + lazy

(when needed)

via static analysis

“languages should support
both strict and lazy”
[PJ2011]

Combining lazy and strict has been done?

“The question is:
What’s the default?
How easy is it to get the other?
How do you mix them together?”

Previous Approaches

- Lenient evaluation: Id, pH [Nikhil91, NAH+95]

Adds strictness to lazy languages.

- Strictness analysis [Mycroft1981, BHA86, CPJ85]
- Cheap Eagerness [Faxen00]

How do real-world lazy programmers add strictness?

seq

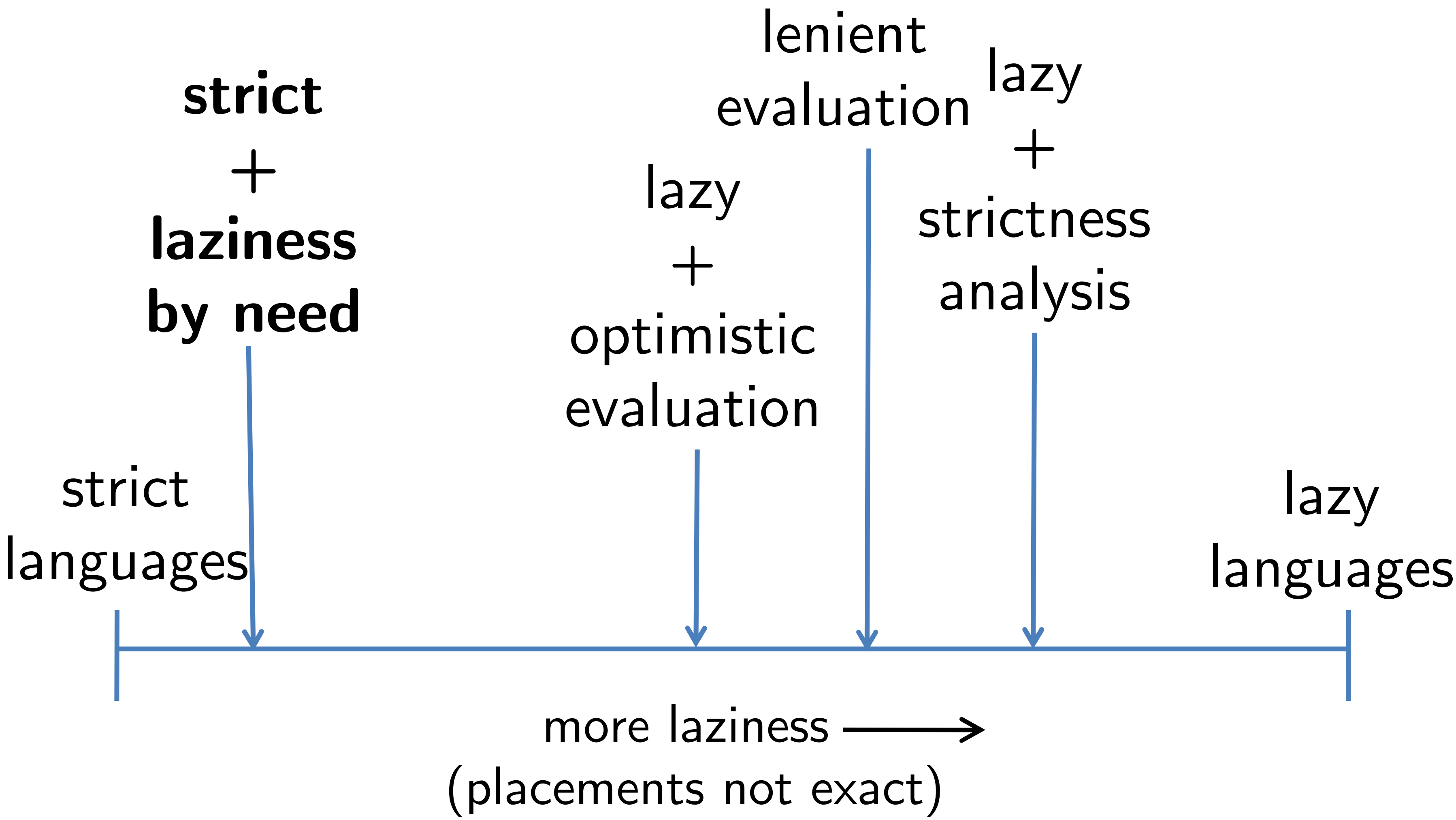
“both before and after optimization, most thunks are evaluated” [Faxen00]

“most thunks are unnecessary” [EPJ03]

What about adding laziness to strict languages?

“most Id90 programs require neither functional nor conditional non-strictness” [SG95]

“in our corpus of R programs … the average evaluation rate of promises is 90%” [MHOV12]


strict
+
laziness
by need
lenient
evaluation
lazy
+
optimistic
evaluation
lazy
+
strictness
analysis
strict
languages
lazy
languages
more laziness
(placements not exact)

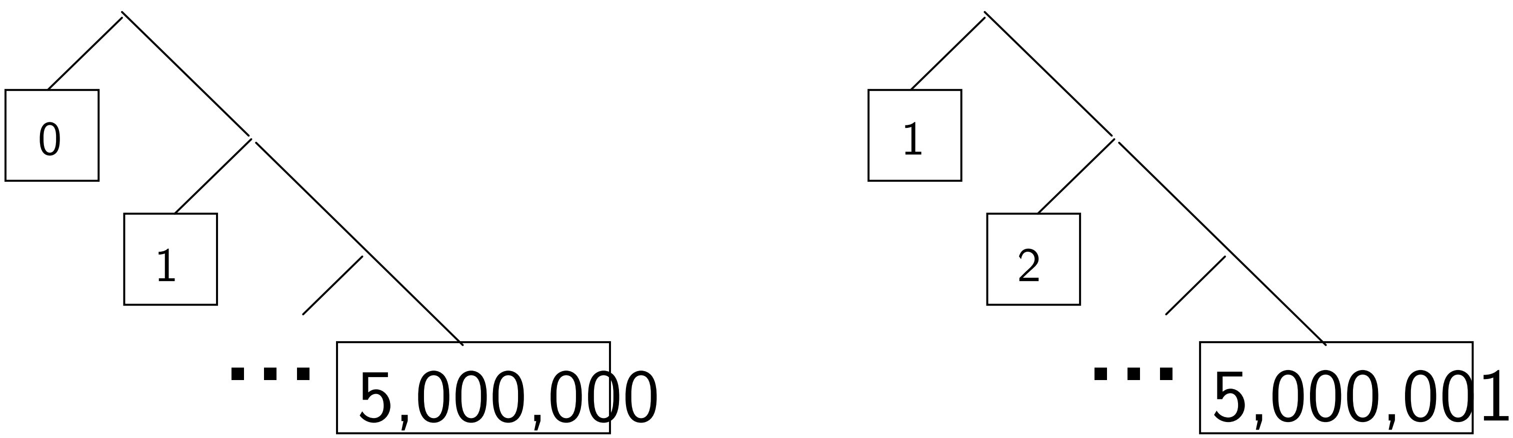
Strict languages already have laziness

So what's the problem?

- Lazy data structures are not enough.
- Lazy annotations are hard to get right.
- Laziness is a global property!

Same Fringe

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




```
samefringe tree1 tree2 =
    (flatten tree1) == (flatten tree2)
```

Same Fringe

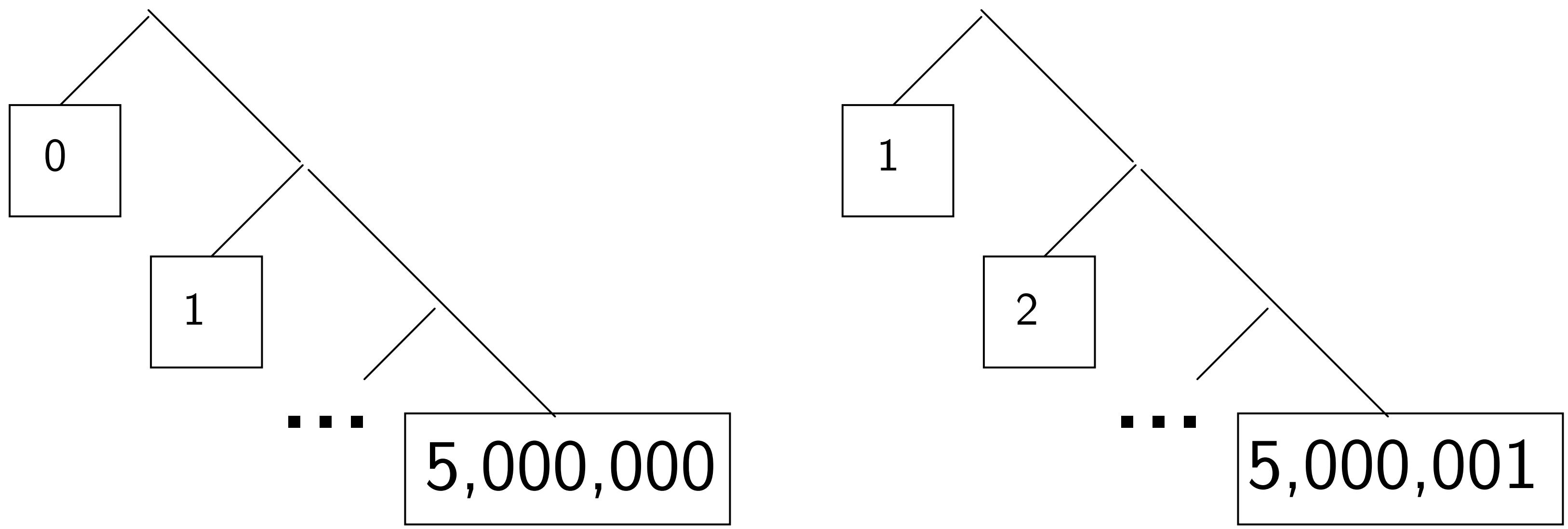
A (Tree X) is either a:
- Leaf X
- Node (Tree X) (Tree X)

```
flatten t = flat t []

flat (Leaf x) acc = x::acc
flat (Node t1 t2) acc = flat t1 (flat t2 acc)
```

Same Fringe (eager)

`let tree1 =` `let tree2 =`



samefringe `tree1 tree2` $\boxed{\texttt{=> false}}$

0m13.363s

Same Fringe (with streams)

A (Stream X) is either a:

- Nil
- Lcons X **$**(Stream X)

Same Fringe (with streams)

```
flatten t = flat t Nil

flat (Leaf x) acc = Lcons x $acc
flat (Node t1 t2) acc = flat t1 (flat t2 acc)
```

Same Fringe (with streams)

```
streameq $Nil $Nil = true
streameq $(Lcons x1 xs1) $(Lcons x2 xs2)=
  x1==x2 && streameq xs1 xs2
streameq _ _ = false
```

Same Fringe (with streams)

```
samefringe tree1 tree2 =
   streameq $(flatten tree1) $(flatten tree2)
```

```
samefringe tree1 tree2 => false
```

0m17.277s

(with lazy trees)

0m36.905s

Same Fringe (naïvely lazy)

```
flatten t = flat t Nil

flat (Leaf x) acc = Lcons x $acc
flat (Node t1 t2) acc = flat t1 (flat t2 acc)
```

Same Fringe (properly lazy)

```
flatten t = flat t Nil

flat (Leaf x) acc = Lcons x $acc
flat (Node t1 t2) acc = flat t1 $(flat t2 acc)
```

Same Fringe (properly lazy)

samefringe `tree1 tree2` $\boxed{\texttt{=> false}}$

`0m0.002s`

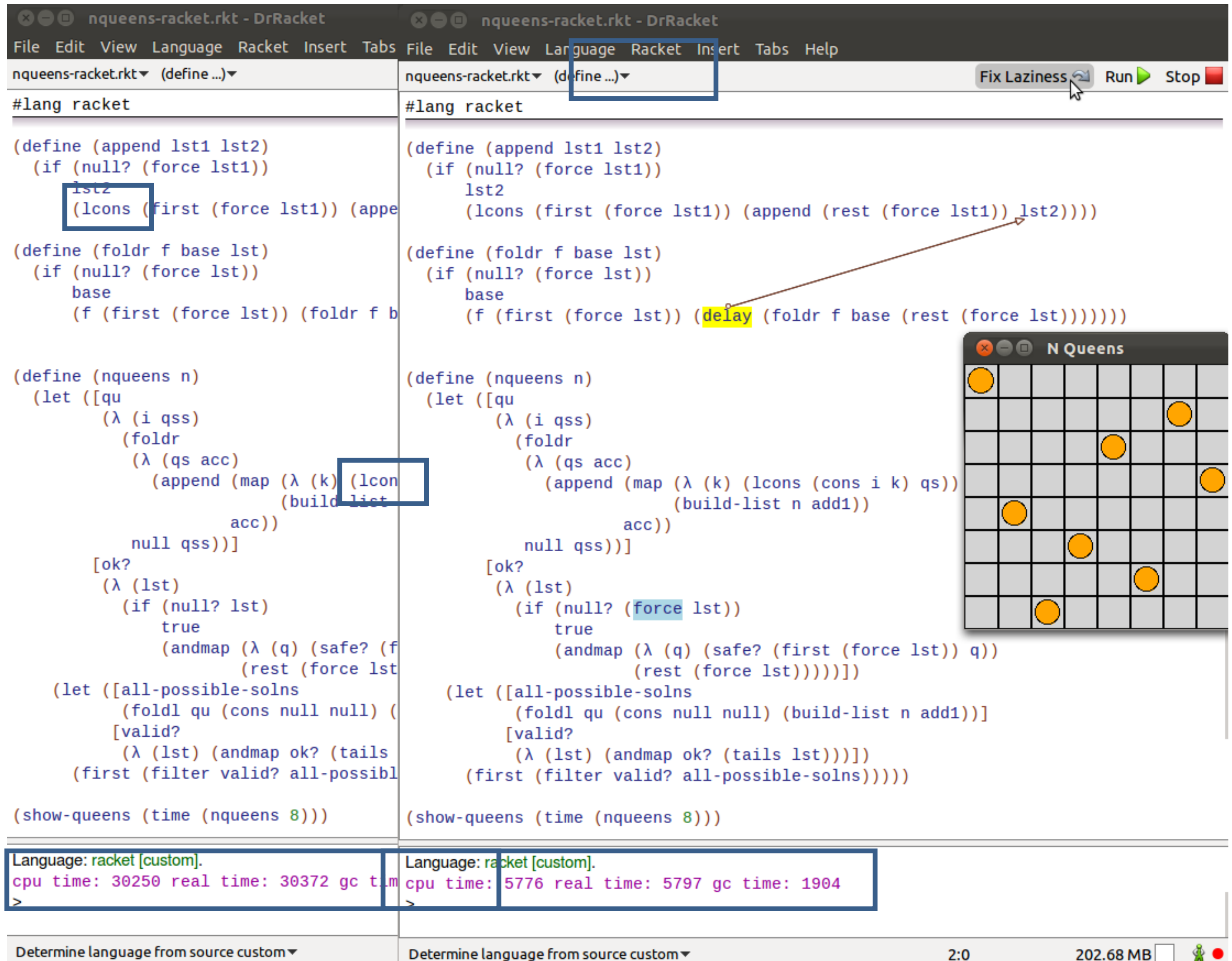
Takeaway

- Using lazy data structures is not enough.
- Additional annotations are needed but can be tricky.
- If only there was a tool that could help with the process . . .

lcons x y

≡

cons x $y



30s

5s

Same Fringe (naïvely lazy)

```
flatten t = flat t Nil

flat (Leaf x) acc = Lcons x $acc
flat (Node t1 t2) acc = flat t1 (flat t2 acc)
```

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 $\quad \ell : \ell \in \widehat{\mathcal{D}}, \ell \notin \widehat{\mathcal{S}}$

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

Abstract value

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

tracks flow of functions arguments.

Analysis specified with 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.

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

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

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

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

$$
\begin{aligned}
&\widehat{\rho} \models (e_f^{\ell_f}\ e_1^{\ell_1} \ldots)^{\ell} \ \text{ iff} && [app]\\
&\qquad \widehat{\rho} \models e_f^{\ell_f} \ \wedge \ \widehat{\rho} \models e_1^{\ell_1} \ \wedge \ \ldots \ \wedge\\
&\qquad (\forall \lambda(x_1 \ldots).\ell_0 \in \widehat{\rho}(\ell_f)) :\\
&\qquad\qquad \widehat{\rho}(\ell_1) \subseteq \widehat{\rho}(x_1) \ \wedge \ \ldots \ \wedge \ \widehat{\rho}(\ell_0) \subseteq \widehat{\rho}(\ell)\\
&\quad \boxed{\wedge \quad (\mathtt{arg}\ \ell_1) \in \widehat{\rho}(x_1) \ \wedge \ \ldots}
\end{aligned}
$$

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

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

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

strict contexts

contexts where a thunk should not appear

examples:

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

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

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

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

We used our tool ...

... and found some bugs.

```
define enqueue(elem dq) =  ...
  let strictprt = ⟨extract strict part of dq⟩
      newstrictprt = ⟨combine elem and strictprt⟩
      lazyprt = force ⟨extract lazy part of dq⟩
      lazyprt1 = ⟨extracted from lazyprt⟩
      lazyprt2 = ⟨extracted from lazyprt⟩
  in Deque newstrictprt (delay ⟨combine lazyprt1 and lazyprt2⟩)
```

Conclusions

- Get the benefits of laziness by starting strict and adding laziness by need.

- A flow-analysis-based tool can help in adding laziness to strict programs.

Thanks.