Static analysis of Higher-order programs

Matthew Might
matt@might.net  http://matt.might.net/
Why static analysis matters
Why static analysis matters

• Optimization
• Parallelism
• Verification
Why: Optimization
Why: Optimization

*Hardware doubles performance every 18 months.*

Moore’s Law
Why: Optimization

Hardware doubles performance every 18 months.
Moore’s Law

Compilers double performance every 18 years.
Proebsting’s Law
Why: Optimization

Optimization is about “freedom of expressions”
Liberating features
Liberating features

• Closures
• Virtual methods
• Laziness
• Coroutines
• Comprehensions
• Garbage collection
• Precise arithmetic
• Pattern matching

• Dynamic typing
• Monads
• Continuations
• Streams
• Polymorphism
• Bounds checks
• Exceptions
• Hybrid types
Liberating features

- Closures
- Virtual methods
- Laziness
- Coroutines
- Comprehensions
- Garbage collection
- Precise arithmetic
- Pattern matching

- Dynamic typing
- Monads
- Continuations
- Streams
- Polymorphism
- Bounds checks
- Exceptions
- Hybrid types
Why: Parallelism
Why: Parallelism

“...80 cores by 2011.”
Paul Otellini, Intel
Why: Parallelism
Why: Parallelism

Source: Intel
Why: Parallelism

Source: Intel
Why: Parallelism

Source: Intel
Why: Parallelism

We are here.

Source: Intel
Why: Parallelism

Static analysis can make:

- Sequential programs parallel
- Parallel programs correct
- Parallel paradigms feasible
Why: Parallelism

Static analysis can make:

• Sequential programs parallel
• Parallel programs correct
• Parallel paradigms feasible
Why: Verification
Why: Verification

Bugs cost U.S. economy $60 billion annually.  

$80 billion in cyber-crime each year.  

FBI
Why: Security matters

- Vulnerabilities
- Fraud
- Dumb Employees
- DoS

Cost of cybercrime ($80bn)
Why: Security matters

- Vulnerabilities: 40%
- Fraud: 38%
- Dumb Employees: 17%
- DoS: 5%

Cost of cybercrime ($80bn)

Source: CSI/FBI Survey 2007
Why: Security matters

Cost of cybercrime ($80bn)

Vulnerabilities: 40%
Fraud: 38%
Dumb Employees: 17%
DoS: 5%

Type of vulnerability

Buffer Overflow: 53%
Injection: 23%
Int Overflow: 13%
Format String: 7%
Other: 3%

Source: CSI/FBI Survey 2007
Source: US CERT, Feb 2008
Why: Verification matters

<exploding-rocket-video />
A fatal exception 0E has occurred at 0137:BFFA21C9. The current application will be terminated.

* Press any key to terminate the current application.
* Press CTRL+ALT+DEL again to restart your computer. You will lose any unsaved information in all applications.

Press any key to continue _
<exploding-rocket-video />
Agenda

• Past: Control-flow analysis
• Present: Environment analysis
• Future: Logic-flow analysis
What is *higher-order*?
Higher-order languages

Higher-order = Computation as value
Higher-order languages

Higher-order = Computation as value

• Scheme
• Lisp
• ML
• Haskell
Higher-order languages

Higher-order = Computation as value

- Scheme
- Lisp
- ML
- Haskell

- Java
- C#
- C++
- Javascript

Objects and closures = Same challenge
Higher-order languages

Higher-order = Computation as value

- Scheme
- Lisp
- ML
- Haskell

- Java
- C#
- C++
- Javascript

Objects and closures = Same challenge
\( \lambda \)-calculus (Church, 1928)
\[\lambda\text{-calculus (Church, 1928)}\]

- Minimalist, universal language
\( \lambda \)-calculus (Church, 1928)

- Minimalist, universal language
- Three expression types:
  
  \[ \nu \text{ [variable]} \]
\(\lambda\)-calculus (Church, 1928)

- Minimalist, universal language
- Three expression types:
  \[v\]  [variable]
  \[e_1(e_2)\]  [function application]
\( \lambda \)-calculus (Church, 1928)

- Minimalist, universal language
- Three expression types:
  
  \[ v \quad \text{[variable]} \]
  
  \[ e_1(e_2) \quad \text{[function application]} \]
  
  \[ \lambda v. e \quad \text{[anonymous function]} \]
\( \lambda \)-calculus domains
\(\lambda\)-calculus domains

- Closure = \(\lambda\)-Term \(\times\) Environment
\lambda\text{-calculus domains}

- Closure = \lambda\text{-Term} \times \text{Environment}

- Environment = \text{Variable} \rightarrow \text{Closure}
\textit{\(\lambda\)-calculus semantics}
\( \lambda \text{-calculus semantics} \)

eval : Exp \( \times \) Environment \( \rightarrow \) Closure
\(\lambda\)-calculus semantics

\[
eval : \text{Exp} \times \text{Environment} \rightarrow \text{Closure}
\]

\[
eval(\![v]\!, env) = env(v)
\]
\(\lambda\)-calculus semantics

\[
\text{eval} : \text{Exp} \times \text{Environment} \to \text{Closure}
\]

\[
\text{eval}(\llbracket v \rrbracket, \text{env}) = \text{env}(v)
\]

\[
\text{eval}(\llbracket \lambda v. e \rrbracket, \text{env}) = (\llbracket \lambda v. e \rrbracket, \text{env})
\]
\(\lambda\)-calculus semantics

\[\text{eval} : \text{Exp} \times \text{Environment} \rightarrow \text{Closure}\]

\[\text{eval}(\llbracket v \rrbracket, env) = env(v)\]

\[\text{eval}(\llbracket \lambda v. e \rrbracket, env) = (\llbracket \lambda v. e \rrbracket, env)\]

\[\text{eval}(\llbracket e_1(e_2) \rrbracket, env) = \text{eval}(e_b, env'), \text{where}\]

\[\left(\llbracket \lambda v. e_b \rrbracket, env'\right) = \text{eval}(e_1, env)\]

\[env'' = env'[v \mapsto \text{eval}(e_2, env)]\]
Higher-order analysis:
A brief history
Early 1980s

- Mostly Lisp, Scheme
- Poor performance relative to C, Fortran
- T: Optimizing Scheme compiler (1982)
Early 1980s

- Mostly Lisp, Scheme
- Poor performance relative to C, Fortran
- T: Optimizing Scheme compiler (1982)

“...every day, I went in to WRL, failed for 8 hours, then went home.”

Olin Shivers, *History of T*
The control-flow problem

g:
  x := x + 1
  if x < 3
    goto g
  else
    return x
The control-flow problem

g:
    x := x + 1
    if x < 3
    goto g
    else
    return x

\[
x := x + 1
\]
\[
x < 3
\]
The control-flow problem

g:
  x := x + 1
  if x < 3
  goto g
else
  return x
The control-flow problem

\[ g(x) = \]
\[ \text{let } x = x + 1 \text{ in } \]
\[ \text{if } x < 3 \text{ then } \]
\[ g(x) \]
\[ \text{else } \]
\[ x \]
The control-flow problem

\[ g(x) = \]
\[ \begin{align*}
let \ x &= x + 1 \ in \\
if \ x < 3 \ then \\
& \ h(x) \\
else \\
& \ x
\end{align*} \]
The control-flow problem

g(x) =
let x = x + 1 in
if x < 3 then
  h(x)
else
  x
The control-flow problem

g(x) =
   let x = x + 1 in
   if x < 3 then
      h(x)
   else
      x
The control-flow problem

apply $f \ x = f(x)$
The control-flow problem

apply \( f \ x = f(x) \)

What procedures are called here?
The control-flow problem

apply \( f \) \( x \) = \( f(x) \)

\( h \) \( x \) = \( x + 1 \)

apply \( h \) 4

\( \text{FlowsTo}[f] = \{h\} \)
The control-flow problem

apply f x = f(x)

h x = x + 1
g x = apply h 4

apply g 3

FlowsTo[f] = {h,g}
The control-flow problem

\[
\text{apply } f \ x = f(x)
\]

\[
(\text{apply apply apply apply}) \ \text{apply}
\]
The control-flow problem

animal.eat();
The control-flow problem

animal.eat();

Which classes flow here?
The control-flow problem

```java
class Dog : Animal {
    void eat() {...}
}

class Cat : Animal {
    void eat() {...}
}
```
The control-flow problem

Control-flow depends on data-flow, but data-flow depends on control-flow.
Higher-order control-flow analysis (CFA)
Observations on flow

\[ e_1(e_2) \]
Observations on flow

\[ \lambda v. e_b \xrightarrow{} e_1(e_2) \]
Observations on flow

\[ \lambda \nu. e_b \]

\[ e_1(e_2) \]

\[ val \]
Observations on flow

\[ \lambda v. e_b \]

\[ e_1(e_2) \]

\text{val}
Observations on flow

\[ \lambda v. e_b \in \text{FlowsTo}[e_1] \quad \text{and} \quad val \in \text{FlowsTo}[e_2] \]

\[ val \in \text{FlowsTo}[v] \]
Observations on flow

\[ \lambda v. e_b \]

\[ e_1(e_2) \]

\[ \text{val} \]
Observations on flow

\[ \lambda v. e_b \to e_1(e_2) \to val \]
Observations on flow

\[ \lambda v.e_b \in \text{FlowsTo}[e_1] \quad \text{and} \quad val \in \text{FlowsTo}[e_b] \]

\[ val \in \text{FlowsTo}[e_1(e_2)] \]
$\lambda v. e_b \in \text{FlowsTo}[e_1]$ and $val \in \text{FlowsTo}[e_b]$

$val \in \text{FlowsTo}[e_1(e_2)]$
0CFA (Shivers, 1988)

\[ \lambda v. e_b \in \text{FlowsTo}[\lambda v. e_b] \]

\[ \lambda v. e_b \in \text{FlowsTo}[e_1] \quad \text{and} \quad \text{val} \in \text{FlowsTo}[e_b] \]
\[ \text{val} \in \text{FlowsTo}[e_1(e_2)] \]

\[ \lambda v. e_b \in \text{FlowsTo}[e_1] \quad \text{and} \quad \text{val} \in \text{FlowsTo}[e_2] \]
\[ \text{val} \in \text{FlowsTo}[v] \]
Example: 0CFA

let id = \x.x
v1 = id(3)
v2 = id(4)
in v2
Example: 0CFA

\[
\text{FlowsTo}[\text{id}] = \{\lambda x. x\}
\]

\[
\text{let } \text{id} = \lambda x. x \\
\text{v1} = \text{id}(3) \\
\text{v2} = \text{id}(4) \\
\text{in } \text{v2}
\]
Example: 0CFA

FlowsTo[id] = {\(\lambda x. x\)}
FlowsTo[x] = {3}

let id = \(\lambda x. x\)

v1 = id(3)
v2 = id(4)
in v2
Example: 0CFA

let id = \lambda x.x
v1 = id(3)
v2 = id(4)
in v2

FlowsTo[id] = \{\lambda x.x\}
FlowsTo[x] = \{3\}
FlowsTo[id(3)] = \{3\}
Example: 0CFA

\[
\begin{align*}
\text{let } & \quad \text{id} = \lambda x. x \\
\text{v1} & = \text{id}(3) \\
\text{v2} & = \text{id}(4) \\
\text{in } & \quad \text{v2}
\end{align*}
\]

\[
\begin{align*}
\text{FlowsTo[id]} & = \{ \lambda x. x \} \\
\text{FlowsTo[x]} & = \{ 3 \} \\
\text{FlowsTo[id(3)]} & = \{ 3 \} \\
\text{FlowsTo[v1]} & = \{ 3 \}
\end{align*}
\]
Example: 0CFA

\[
\begin{align*}
\text{let } \ id &= \lambda x. x \\
v1 &= id(3) \\
v2 &= id(4) \\
in v2
\end{align*}
\]

\[
\begin{align*}
\text{FlowsTo}[id] &= \{ \lambda x. x \} \\
\text{FlowsTo}[x] &= \{ 3, 4 \} \\
\text{FlowsTo}[id(3)] &= \{ 3 \} \\
\text{FlowsTo}[v1] &= \{ 3 \}
\end{align*}
\]
Example: 0CFA

let id = \x.x
v1 = id(3)
v2 = id(4)
in v2

FlowsTo[id] = {\lambda x.x}
FlowsTo[x] = {3, 4}
FlowsTo[id(3)] = {3}
FlowsTo[v1] = {3}
FlowsTo[id(4)] = {3, 4}
Example: 0CFA

\[
\begin{align*}
\text{let } \text{id} &= \lambda x. x \\
\text{v}1 &= \text{id}(3) \\
\text{v}2 &= \text{id}(4) \\
\text{in } \text{v}2 \\
\text{FlowsTo[id]} &= \{\lambda x. x\} \\
\text{FlowsTo[x]} &= \{3, 4\} \\
\text{FlowsTo[id(3)]} &= \{3\} \\
\text{FlowsTo[v1]} &= \{3\} \\
\text{FlowsTo[id(4)]} &= \{3, 4\} \\
\text{FlowsTo[v2]} &= \{3, 4\}
\end{align*}
\]
Example: 0CFA

let id = \( \lambda x \cdot x \)

\[
\begin{align*}
\text{FlowsTo}[id] &= \{\lambda x \cdot x\} \\
\text{FlowsTo}[x] &= \{3, 4\} \\
\text{FlowsTo}[id(3)] &= \{3, 4\} \\
\text{FlowsTo}[v1] &= \{3\} \\
\text{FlowsTo}[id(4)] &= \{3, 4\} \\
\text{FlowsTo}[v2] &= \{3, 4\}
\end{align*}
\]
Example: 0CFA

let id = \x.x
v1 = id(3)
v2 = id(4)
in v2

FlowsTo[id] = \x.x
FlowsTo[x] = \{3,4\}
FlowsTo[id(3)] = \{3,4\}
FlowsTo[v1] = \{3,4\}
FlowsTo[id(4)] = \{3,4\}
FlowsTo[v2] = \{3,4\}
0CFA recap

• It worked!
• It’s not fast: $O(n^3)$
• It’s imprecise.
The Palsberg paradigm

“Higher-order analysis = first-order analysis + control-flow analysis.”

Jens Palsberg

0CFA (Shivers, 1988)

For each application $e_1 e_2$:

\[
(\lambda x. e_b) \in \text{FlowsTo}[e_1] \Rightarrow \text{FlowsTo}[e_b] \subseteq \text{FlowsTo}[(e_1 e_2)]
\]

\[
(\lambda x. e_b) \in \text{FlowsTo}[e_1] \Rightarrow \text{FlowsTo}[e_2] \subseteq \text{FlowsTo}[x]
\]
VFA (Heinglen, 1992)

For each application \( e_1 e_2 \):

\[
(\lambda x. e_b) \in \text{FlowsTo}[e_1] \Rightarrow \text{FlowsTo}[e_b] = \text{FlowsTo}[(e_1 e_2)]
\]

\[
(\lambda x. e_b) \in \text{FlowsTo}[e_1] \Rightarrow \text{FlowsTo}[e_2] = \text{FlowsTo}[x]
\]
VFA

- Fast: Almost linear
- Awful precision: `FlowsTo[3] = {3,4}`
$\kappa$-CFA (Shivers, 1991)
$k$-CFA (Shivers, 1991)

- Splits variable flow sets by calling context
$k$-CFA (Shivers, 1991)

- Splits variable flow sets by calling context

```
let id = \(\lambda x.x\)
  v1 = id(3)
  v2 = id(4)
in v2
```

\[
\text{FlowsTo}[x, \text{id}(3)] = \{3\}
\]
\[
\text{FlowsTo}[x, \text{id}(4)] = \{4\}
\]
$k$-CFA (Shivers, 1991)

- Splits variable flow sets by calling context
- EXPTIME-Complete (Van Horn & Mairson, 2008)

\[
\begin{align*}
\text{let } & \text{id} = \lambda x.x \\
v1 &= \text{id}(3) \\
v2 &= \text{id}(4) \\
in & v2
\end{align*}
\]

\[
\begin{align*}
\text{FlowsTo}[x, \text{id}(3)] &= \{3\} \\
\text{FlowsTo}[x, \text{id}(4)] &= \{4\}
\end{align*}
\]
poly-CFA (Wright, 1995)
poly-CFA (Wright, 1995)

- Same philosophy as let-bound polymorphism
poly-CFA (Wright, 1995)

- Same philosophy as let-bound polymorphism
- Calls to same let-bound function kept distinct
poly-CFA (Wright, 1995)

- Same philosophy as let-bound polymorphism
- Calls to same let-bound function kept distinct

```haskell
let id = λx.x
    v1 = id(3)
    v2 = id(4)
  in v2
```
poly-CFA (Wright, 1995)

- Same philosophy as let-bound polymorphism
- Calls to same let-bound function kept distinct

```plaintext
let id = λx.x
v1 = (λx1.x1)(3)
v2 = (λx2.x2)(4)
in v2
```

FlowsTo[v1] = {3}
FlowsTo[v2] = {4}
poly-CFA (Wright, 1995)

- Same philosophy as let-bound polymorphism
- Calls to same let-bound function kept distinct
- Speed, precision not necessarily tradeoff

\[
\text{let } \text{id} = \lambda x.x \quad \text{FlowsTo}[v1] = \{3\}
\]

\[
v1 = (\lambda x_1.x_1)(3)
\]

\[
v2 = (\lambda x_2.x_2)(4) \quad \text{FlowsTo}[v2] = \{4\}
\]

\[
in v2
\]
Other work

- Reformulations
  - As abstract interpretations
  - As type systems
- Demand-driven
- Flow sensitivity
- FlowsTo cloning (by point, context)
- Complexity bounds
Aftermath

• Optimization: Common sub-exp. elimination
• Optimization: Copy propagation
• Optimization: Context-sensitive inlining
• Optimization: Induction variable elimination
• Optimization: Invariant hoisting
• Optimization: Global register allocation
• …
• Stalin, MLton: Parity with C
Control-flow analysis is not enough.
Half the problem

\(\lambda\)-terms don’t flow anywhere.
Half the problem

$\lambda$-terms don’t flow anywhere.

Closures do.
Half the problem

$\lambda$-terms don’t flow anywhere.

Closures do.

Closure $= \lambda$-Term $\times$ Environment
Half the problem

λ-terms don’t flow anywhere.

Closures do.

Closure = λ-Term
Half the problem

Classes don’t flow anywhere.

Objects do.

Object = Class \times Record
Why environment matters
Inlining

\[ f \, x \, h = \text{if } x = 0 \text{ then } h() \text{ else } \lambda().x \]

\[ f \, 0 \, (f \, 3 \, \text{nil}) \]
f x h = if x = 0
then h()
else λ().x

f 0 (f 3 nil)

Safe to inline?
Inlining

\[ f \ x \ h = \begin{cases} \text{if } x = 0 \\ \text{then } h() \\ \text{else } \lambda().x \end{cases} \]

\[ f \ 0 \ (f \ 3 \ \text{nil}) \]

Safe to inline?
Inlining

\[ f \ x \ h = \begin{cases} \text{if } x = 0 \\ \text{then } h() \\ \text{else } \lambda().x \end{cases} \]

\[ f \ 0 \ (f \ 3 \ \text{nil}) \]

Safe to inline?
Inlining

\[ f \ x \ h = \begin{cases} \lambda().x & \text{if } x = 0 \\ h() & \text{else} \end{cases} \]

\[ f \ 0 \ (f \ 3 \ \text{nil}) \]

Safe to inline?
Inlining

\[ f(x,h) = \begin{cases} h() & \text{if } x = 0 \\ \lambda().x & \text{else} \end{cases} \]

\[ f(0)(f(3)\text{ nil}) \]

Safe to inline?
Inlining

$$f \ x \ h = \text{if } x = 0 \text{ then } h() \text{ else } \lambda().x$$

$$f \ 0 \ (f \ 3 \ \text{nil})$$

Safe to inline?
Inlining

\[ f \ x \ h = \text{if } x = 0 \ \text{then } h() \ \text{else } \lambda().x \]

\[ f \ 0 \ (f \ 3 \ \text{nil}) \]

Safe to inline?
Inlining

\[ f \ x \ h = \begin{cases} \text{if } x = 0 \\
\text{then } h() \\
\text{else } \lambda().x \end{cases} \]

\[ f \ 0 \ (f \ 3 \ \text{nil}) \]

\( (\lambda().x, x \to 3) \)

Safe to inline?
Inlining

\[ f \ x \ h = \begin{cases} 
    h() & \text{if } x = 0 \\
    \lambda().x & \text{else}
\end{cases} \]

Safe to inline?
Inlining

\[ f\ x\ h = \begin{cases} & x = 0 \\
\hspace{1cm} & \text{then } h() \\
\hspace{1cm} & \text{else } \lambda().x \end{cases} \]

\[ f\ 0\ (f\ 3\ \text{nil}) \]

Safe to inline?
Inlining

\[ f \, x \, h = \begin{cases} h() & \text{if } x = 0 \\ \lambda().x & \text{else} \end{cases} \]

\[ f \, 0 \, (f \, 3 \, \text{nil}) \]

Safe to inline?
Inlining

\[ f(x) = \begin{cases} 
\text{if } x = 0 \\
\text{then } (\lambda().x)() \\
\text{else } \lambda().x 
\end{cases} \]

\[ f(0) \ (f(3) \ nil) \]

\( (\lambda().x, x \rightarrow 3) \)

\[ [x \rightarrow 3] \ \lor \ [x \rightarrow 0] \]

Safe to inline? No.
And also...

- Super-beta copy propagation
- Globalization (Sestoft)
- Rematerialization
- Teleportation/environment lifting
- Register-allocated environments
- Static environment allocation
- Stack-aware inlining
- Lightweight closure conversion (Wand & Steckler)
- Generalized escape analysis
- Lightweight continuation promotion
- Coroutine fusion
- ...

...
Program run times

No optimization  + CFA  + Env. analysis
# Program run times

<table>
<thead>
<tr>
<th></th>
<th>No optimization</th>
<th>+ CFA</th>
<th>+ Env. analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td>12s</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9s</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6s</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3s</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0s</td>
<td>sboyer nboyer</td>
<td>perm earley</td>
<td>lattice fringe</td>
</tr>
<tr>
<td></td>
<td>stream doubler</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Program run times

- No optimization
- + CFA
- + Env. analysis

- sboyer
- nboyer
- perm
- earley
- lattice
- fringe
- stream
- doubler
Program run times

- No optimization
- + CFA
- + Env. analysis

<table>
<thead>
<tr>
<th>Program</th>
<th>No optimization</th>
<th>+ CFA</th>
<th>+ Env. analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td>sboyer</td>
<td>0s</td>
<td>3s</td>
<td>6s</td>
</tr>
<tr>
<td>nboyer</td>
<td>0s</td>
<td>3s</td>
<td>6s</td>
</tr>
<tr>
<td>perm</td>
<td>0s</td>
<td>3s</td>
<td>6s</td>
</tr>
<tr>
<td>earley</td>
<td>0s</td>
<td>3s</td>
<td>6s</td>
</tr>
<tr>
<td>lattice</td>
<td>9s</td>
<td>12s</td>
<td>12s</td>
</tr>
<tr>
<td>fringe</td>
<td>6s</td>
<td>9s</td>
<td>9s</td>
</tr>
<tr>
<td>stream</td>
<td>6s</td>
<td>9s</td>
<td>9s</td>
</tr>
<tr>
<td>doubler</td>
<td>6s</td>
<td>9s</td>
<td>9s</td>
</tr>
</tbody>
</table>
Program run times

- sboyer: 0s, 3s, 6s, 9s, 12s
- nboyer: 0s, 3s, 6s, 9s, 12s
- perm: 0s, 3s, 6s, 9s, 12s
- earley: 0s, 3s, 6s, 9s, 12s
- lattice: 0s, 3s, 6s, 9s, 12s
- fringe: 0s, 3s, 6s, 9s, 12s
- stream: 0s, 3s, 6s, 9s, 12s
- doubler: 0s, 3s, 6s, 9s, 12s

Categories:
- No optimization
- + CFA
- + Env. analysis
Program run times

PLDI 2006
<coroutine-fusion>
Coroutine paradigm
Coroutine paradigm
Coroutine paradigm

Input
Coroutine paradigm
Coroutine paradigm
Coroutine paradigm
Coroutine API

- **get**: Receive upstream value
- **put**: Send value downstream
Coroutine API

- **get**: Receive upstream value
- **put**: Send value downstream

```python
doubler() =
    put(2 × get())

doubler()
```
Coroutine API

- **get**: Receive upstream value
- **put**: Send value downstream

```doubler() =
    put(2 \times \text{get}());
```
Coroutine API

- **get**: Receive upstream value
- **put**: Send value downstream

```plaintext
doubler() = put(2 × get()) ;
doubler()
```

Coroutine API

- **get**: Receive upstream value
- **put**: Send value downstream

```plaintext
doubler() = put(2 × get()) ;
doubler()
```

**Diagram:**
- **get**: Receive value upstream
- **put**: Send value downstream
- **2 ×**: Multiplication operation
Coroutine API

- **get**: Receive upstream value
- **put**: Send value downstream

```
doubler() = put(2 \times \text{get}());
```

Coroutine fusion

Input → Compute → Output

Input → Compute → Output
Coroutine fusion
Coroutine fusion

\[ \lambda \]
Coroutine fusion

Input
Compute
Output

\[ \lambda \]

Input
Compute
Output
Coroutine fusion
Coroutine fusion

![Coroutine fusion diagram](image-url)
Coroutine fusion

Image of a computational diagram with arrows indicating input, computation, and output.
Coroutines to parallelism
Coroutines to parallelism
Coroutines to parallelism

CPU 1

CPU 2

CPU 3
Examples
Examples

App
TCP
IP
HW

Lex
Parse
Analyze
Opt
Emit

Input → Update → Render

54
</coroutine-fusion>
How to save the environment
The environment problem

Given two environments, are they equivalent?
The environment problem

Given two environments, are they equivalent?

• What does equivalent mean?
The environment problem

Given two environments, are they equivalent?

• What does *equivalent* mean?

• *Which* two environments?
Equivalence

$env_1$ is equivalent to $env_2$
over variables $v_1, \ldots, v_n$
if $env_1(v_i) = env_2(v_i)$
Equivalence

\[ o_1 \text{ is equivalent to } o_2 \]

over fields \( f_1, \ldots, f_n \)

if \[ o_1.f_i = o_2.f_i \]
Equivalence

$o_1$ is equivalent to $o_2$
over fields $f_1, \ldots, f_n$
if $o_1.f_i = o_2.f_i$

The environment problem is
a higher-order analog
of the must-alias problem.
Which environments?

\[ f(x) = \lambda z. x+z \]

\[
\text{loop } n = \\
\text{print } f(n)(n) ; \\
\text{loop } (n+2)
\]

\[ \text{loop } 0 \]
Which environments?

\[ f(x) = \lambda z. x + z \]

\text{loop } n =

\text{print } f(n)(n) ;
\text{loop } (n+2)

\text{loop } 0
Which environments?

\[ f(x) = \lambda z. x + z \]

loop \( n = \)

print \( f(n)(n) \)

loop \( (n+2) \)

loop 0

\[ x \rightarrow 0 \]
\[ x \rightarrow 2 \]
\[ x \rightarrow 4 \]
\[ x \rightarrow 6 \]
\[ x \rightarrow 8 \]

...
Building environment analysis
Building environment analysis

1. Build concrete machine for λ-calculus
Building environment analysis

1. Build concrete machine for $\lambda$-calculus

2. Abstract into analysis (Cousot$^2$, 1977)
Building environment analysis

1. Build concrete machine for λ-calculus
2. Abstract into analysis (Cousot\textsuperscript{2}, 1977)
3. Use counting to derive equivalence
Concrete machine
Concrete machine

- Convert program $e$ into machine state $s_0$
Concrete machine

- Convert program $e$ into machine state $s_0$
- Transition from state $s_n$ to state $s_{n+1}$

$e$

$s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow \ldots$
Abstract interpretation

\[
\begin{align*}
  &s_0 &s_1 &s_2 &s_3 &s_4 &\ldots \\
  e &\downarrow &\rightarrow &\rightarrow &\rightarrow &\rightarrow &\ldots \\
\end{align*}
\]
Abstract interpretation

\[
\begin{align*}
e \\ s_0 & \rightarrow s_1 & s_2 & \rightarrow s_3 & \rightarrow s_4 & \rightarrow \ldots
\end{align*}
\]
Abstract interpretation

\[ e \], \[ \hat{s}_0 \rightarrow s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow \ldots \]
Abstract interpretation

\[ e \]

\[ s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow \ldots \]

\[ \hat{s}_0 \rightarrow \hat{s}_1 \rightarrow \hat{s}_2 \rightarrow \hat{s}_3 \rightarrow \hat{s}_{3,1} \]
Abstract interpretation
Abstract interpretation

\( e \)

\( s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow \ldots \)

\( \hat{s}_0 \rightarrow \hat{s}_1 \rightarrow \hat{s}_2 \rightarrow \hat{s}_3 \rightarrow \hat{s}_4 \)

\( \hat{s}_{3,1} \)
Abstract interpretation

Theorem: The abstract simulates the concrete.
CPS $\lambda$-calculus syntax

- Two expression term types ($e$)
  - $v$
  - $\lambda v_1 \ldots v_n . \text{call}$

- One call term type ($\text{call}$)
  - $(e_0 \ e_1 \ldots e_n)$
Concrete semantics (1)
Concrete semantics (1)

- Machine state: \((\text{call}, \text{env}, \text{mem})\)
Concrete semantics (1)

- Machine state: \((call, env, mem)\)
  - \(env : \text{Var} \rightarrow \text{Addr}\) [address of variable]
Concrete semantics (1)

- Machine state: \((call, env, mem)\)
  - \(env: \text{Var} \rightarrow \text{Addr}\) [address of variable]
  - \(mem: \text{Addr} \rightarrow \text{Value}\) [value of address]
Concrete semantics (1)

• Machine state: \((\textit{call}, \textit{env}, \textit{mem})\)
  - \(\textit{env} : \text{Var} \rightarrow \text{Addr}\) [address of variable]
  - \(\textit{mem} : \text{Addr} \rightarrow \text{Value}\) [value of address]

• \(\text{eval}(v, \textit{env}, \textit{mem}) = \text{mem}(\textit{env}(v))\)
Concrete semantics (1)

- Machine state: \((call, env, mem)\)
  - \(env: \text{Var} \rightarrow \text{Addr}\) [address of variable]
  - \(mem: \text{Addr} \rightarrow \text{Value}\) [value of address]

- \(\text{eval}(v, env, mem) = mem(env(v))\)
- \(\text{eval}(\lambda, env, mem) = (\lambda, env)\)
Concrete semantics (2)

\[
(\llbracket (e_0 \ e_1 \ldots \ e_n) \rrbracket, \ env, \ mem) \Rightarrow (call, \ env'', \ mem')
\]
Concrete semantics (2)

\[(\llbracket (e_0 \ e_1 \ldots \ e_n) \rrbracket, \ env, \ mem) \Rightarrow (\text{call}, \ env'', \ mem')\]

\[val_i = \text{eval}(e_i, \ env, \ mem)\]
Concrete semantics (2)

\[
\frac{(\llbracket (e_0 \ e_1 \ldots \ e_n) \rrbracket, \ env, \ mem) \Rightarrow (call, \ env', \ mem')}{val_i = \text{eval}(e_i, \ env, \ mem)}
\]

\[
val_0 = (\llbracket \lambda v_1 \ldots \ v_n . call \rrbracket, \ env')
\]
Concrete semantics (2)

\[ ((e_0 \ e_1 \ \ldots \ e_n), \ env, \ mem) \Rightarrow (\text{call, env}', \text{mem}') \]

\[
\begin{align*}
val_i &= \text{eval}(e_i, \ env, \ mem) \\
val_0 &= ([\lambda v_1 \ \ldots \ v_n. \text{call}], \text{env}') \\
a_i &= \text{alloc}(s, \ v_i)
\end{align*}
\]
Concrete semantics (2)

\[
(\langle e_0 \; e_1 \; \ldots \; e_n \rangle, \; env, \; mem) \Rightarrow (\text{call}, \; env'', \; mem')
\]

\[
val_i = \text{eval}(e_i, \; env, \; mem)
\]

\[
val_0 = (\langle \lambda v_1 \; \ldots \; v_n . \text{call} \rangle, \; env')
\]

\[
a_i = \text{alloc}(s, \; v_i)
\]

\[
env'' = env'[v_i \rightarrow a_i]
\]
Concrete semantics (2)

$$\llbracket\langle e_0 \ e_1 \ \ldots \ e_n \rangle\rrbracket, \ env, \ mem) \Rightarrow (call, \ env'', mem')$$

$$val_i = eval(e_i, env, mem)$$
$$val_0 = (\llbracket\lambda v_1 \ \ldots \ v_n.\ call\rrbracket, env')$$
$$a_i = alloc(s, v_i)$$
$$env'' = env'[v_i \rightarrow a_i]$$
$$mem' = mem[a_i \rightarrow val_i]$$
Abstract semantics

\[(\llbracket (e_0 \ e_1 \ldots \ e_n) \rrbracket, env, mem) \Rightarrow (\text{call}, env', mem')\]

\[val_i = \text{eval}(e_i, env, mem)\]

\[val_0 = (\llbracket \lambda v_1 \ldots v_n.\text{call} \rrbracket, env')\]

\[a_i = \text{alloc}(s, v_i)\]

\[env'' = env'[v_i \rightarrow a_i]\]

\[mem' = mem[a_i \rightarrow val_i]\]
Abstract semantics

\[
(\llbracket (e_0 \ e_1 \ldots \ e_n) \rrbracket, \overline{env}, \overline{mem}) \Rightarrow (\text{call}, \overline{env'}, \overline{mem'})
\]

\[
\overline{val}_i = \overline{\text{eval}}(e_i, \overline{env}, \overline{mem})
\]

\[
\overline{val}_0 = (\llbracket \lambda v_1 \ldots \ v_n. \text{call} \rrbracket, \overline{env'})
\]

\[
\overline{a}_i = \overline{\text{alloc}}(\overline{s}, \overline{v}_i)
\]

\[
\overline{env''} = \overline{env'}[v_i \rightarrow \overline{a}_i]
\]

\[
\overline{mem'} = \overline{mem}[\overline{a}_i \rightarrow \overline{val}_i]
\]
Abstract semantics

\[
\begin{align*}
&\left[\left(e_0 \; e_1 \; \ldots \; e_n\right), \; \overline{env}, \; \overline{mem}\right] \Rightarrow \left(call, \; \overline{env'}, \overline{mem'}\right) \\
&\overline{val}_i = \overline{eval}(e_i, \; \overline{env}, \; \overline{mem}) \\
&\overline{val}_0 = \left(\left[\lambda v_1 \; \ldots \; v_n. call\right], \overline{env'}\right) \\
&\overline{a}_i = \overline{alloc}(\overline{s}, \; v_i) \\
&\overline{env''} = \overline{env'}[v_i \rightarrow \overline{a}_i] \\
&\overline{mem'} = \overline{mem}[^{\overline{a}_i \rightarrow \overline{val}_i}] \\
&\overline{Addr} = \text{finite}
\end{align*}
\]
Abstract semantics

\[
\begin{align*}
\llbracket (e_0 \ e_1 \ldots \ e_n) \rrbracket, \ env, \ mem & \Rightarrow (call, \ env', mem') \\
\overline{val}_i &= \overline{eval}(e_i, env, mem) \\
\overline{val}_0 &= \llbracket \lambda v_1 \ldots \ v_n. call \rrbracket, \overline{env}' \\
\overline{a}_i &= \overline{alloc}(\overline{s}, v_i) \\
\overline{env}'' &= \overline{env}'[v_i \rightarrow \overline{a}_i] \\
\overline{mem}' &= \overline{mem} \uplus [\overline{a}_i \rightarrow \overline{val}_i] \\
\overline{Addr} &= \text{finite}
\end{align*}
\]
Abstract semantics

\(\llbracket (e_0 \; e_1 \; \ldots \; e_n) \rrbracket, \; \text{env}, \; \text{mem} \Rightarrow (\text{call}, \; \text{env}''', \; \text{mem}')\)

\(\overline{val}_i = \overline{eval}(e_i, \; \text{env}, \; \text{mem})\)

\(\overline{val}_0 = (\llbracket \lambda v_1 \; \ldots \; v_n. \text{call} \rrbracket, \text{env}')\)

\(\overline{a}_i = \overline{alloc}(\overline{s}, \; v_i)\)

\(\overline{env}'' = \overline{env}'[v_i \rightarrow \overline{a}_i]\)

\(\overline{mem}' = \overline{mem} \cup [\overline{a}_i \rightarrow \overline{val}_i]\)

\(\overline{Addr} = \text{finite}\)

\(\overline{Value} = \text{Closure}\)
Abstract semantics

\[(\llbracket e_0 \; e_1 \; \ldots \; e_n \rrbracket, \overline{env}, \overline{mem}) \Rightarrow (\text{call}, \overline{env'}, \overline{mem'})\]

\[
\overline{val}_i = \overline{eval}(e_i, \overline{env}, \overline{mem})
\]

\[
\overline{val}_0 = (\llbracket \lambda v_1 \; \ldots \; v_n . \text{call} \rrbracket, \overline{env'})
\]

\[
\overline{a}_i = \overline{alloc}(\overline{s}, v_i)
\]

\[
\overline{env''} = \overline{env'}[v_i \rightarrow \overline{a}_i]
\]

\[
\overline{mem'} = \overline{mem} \cup [\overline{a}_i \rightarrow \overline{val}_i]
\]

\[
\overline{Addr} = \text{finite}
\]

\[
\overline{Value} = \mathcal{P}(\overline{\text{Closure}})
\]
Universal CFA

\[(\llbracket(e_0 \; e_1 \; \ldots \; e_n)\rrbracket, \overline{\text{env}}, \overline{\text{mem}}) \Rightarrow (\text{call}, \overline{\text{env}}', \overline{\text{mem}}')\]

\[\overline{\text{val}}_i = \overline{\text{eval}}(e_i, \overline{\text{env}}, \overline{\text{mem}})\]

\[\overline{\text{val}}_0 \ni (\llbracket\lambda v_1 \; \ldots \; v_n. \text{call}\rrbracket, \overline{\text{env}}')\]

\[\overline{\text{a}}_i = \overline{\text{alloc}}(\overline{s}, \overline{v}_i)\]

\[\overline{\text{env}}' = \overline{\text{env}}'[\overline{v}_i \rightarrow \overline{\text{a}}_i]\]

\[\overline{\text{mem}}' = \overline{\text{mem}} \cup [\overline{\text{a}}_i \rightarrow \overline{\text{val}}_i]\]

\[\overline{\text{Addr}} = \text{finite}\]

\[\overline{\text{Value}} = \mathcal{P}(\overline{\text{Closure}})\]
Which environments
Which environments

$\hat{s}_0 \rightarrow \hat{s}_1 \rightarrow \hat{s}_2 \rightarrow \hat{s}_3 \rightarrow \hat{s}_4$

$\hat{s}_3.1 \rightarrow \hat{s}_2 \rightarrow \hat{s}_3 \rightarrow \hat{s}_4 \rightarrow \hat{s}_3.1$
Which environments
Which environments
Which environments

(call, \texttt{env}, \texttt{mem})
Which environments

(call, env, mem)
Which environments

(call, \overline{env}, \overline{mem})
Which environments

$$(\lambda_1, [x \rightarrow \overline{a}_1])$$

$$(\lambda_2, [x \rightarrow \overline{a}_2])$$

(call, env, mem)
Which environments

\((\lambda_1,[x \rightarrow \bar{a}_1])\)  \((\lambda_2,[x \rightarrow \bar{a}_2])\)

If \(\bar{a}_1 = \bar{a}_2\), are the concrete environments they represent equivalent?
Which environments

\[ (\lambda_1, [x \rightarrow \bar{a}_1]) \quad (\lambda_2, [x \rightarrow \bar{a}_2]) \]

\{a_1, a_2\}

If \(\bar{a}_1 = \bar{a}_2\), are the concrete environments they represent equivalent?

Maybe.
Which environments

\((\lambda_1,[x \rightarrow \bar{a}_1])\) \quad (\lambda_2,[x \rightarrow \bar{a}_2])

\{ a \}

If \(\bar{a}_1 = \bar{a}_2\), are the concrete environments they represent equivalent?

Yes!
Which environments

\((\lambda_1,[x \rightarrow \bar{a}_1])\) \hspace{1cm} (\lambda_2,[x \rightarrow \bar{a}_2])

If \(\bar{a}_1 = \bar{a}_2\), are the concrete environments they represent equivalent?

It depends.
Counting principle

If \( \{x\} = \{y\} \), then \( x = y \).
Counting

- Add measure to each state: $\mu$
- $\mu : \text{Addr} \to \{0, 1, \infty\}$ counts counterparts
Counting

\[ \mu(\bar{a}) = 0 \]
Counting

\[ \mu(\bar{a}) = 1 \]
Counting

\[ \mu(\bar{a}) = \infty \]
Abstract semantics

\((\llbracket (e_0 \ e_1 \ ... \ e_n) \rrbracket, \overline{env}, \overline{mem}) \Rightarrow (call, \overline{env}'', \overline{mem}')\)

\[\overline{val}_i = \overline{eval}(e_i, \overline{env}, \overline{mem})\]
\[\overline{a}_i = \overline{alloc}(\bar{s}, v_i)\]
\[\overline{val}_0 \ni (\llbracket \lambda v_1 \ ... \ v_n.call \rrbracket, \overline{env}')\]
\[\overline{env}'' = \overline{env}'[v_i \rightarrow \overline{a}_i]\]
\[\overline{mem}' = \overline{mem} \sqcup [\overline{a}_i \rightarrow \overline{val}_i]\]
Abstract semantics

\[ (\llbracket (e_0 \ e_1 \ \ldots \ \ e_n) \rrbracket, \ \overline{env}, \ \overline{mem}, \mu) \Rightarrow (\text{call}, \ \overline{env}'', \overline{mem}', \mu') \]

\[ \overline{val}_i = \overline{eval}(e_i, \ \overline{env}, \ \overline{mem}) \]

\[ \overline{a}_i = \overline{alloc}(\overline{s}, \ v_i) \]

\[ \overline{val}_0 \ \in \ (\llbracket \lambda v_1 \ \ldots \ v_n. \ \text{call} \rrbracket, \overline{env}') \]

\[ \overline{env}'' = \overline{env}'[v_i \rightarrow \overline{a}_i] \]

\[ \overline{mem}' = \overline{mem} \ \sqcup \ [\overline{a}_i \rightarrow \overline{val}_i] \]
Abstract semantics

\[
\boxed{\left[\left( e_0 \ e_1 \ldots \ e_n \right) \right], \ \overline{env}, \ \overline{mem}, \mu} \Rightarrow (\text{call}, \ \overline{env}'', \overline{mem}', \mu')
\]

\[
\overline{val_i} = \overline{eval}(e_i, \overline{env}, \overline{mem})
\]

\[
\overline{a_i} = \overline{alloc}(\overline{s}, v_i)
\]

\[
\overline{val_0} \ni \left( \boxed{\left[ \lambda v_1 \ldots v_n . \text{call} \right], \overline{env}' \right)
\]

\[
\overline{en\nu''} = \overline{en\nu}[v_i \rightarrow \overline{a_i}]
\]

\[
\overline{mem'} = \overline{mem} \sqcup [\overline{a_i} \rightarrow \overline{val_i}]
\]

\[
\mu' = \mu \oplus [\overline{a_i} \rightarrow 1]
\]
Example: Inlining

\[ \lambda v \ k. (k \ a) \]

\[ (f \ x \ c) \]
Example: Inlining

\[ \lambda v \ k. (k \ a) \]

\[ (f \ x \ c) \]
Example: Inlining

\[ \lambda v \ k. (k \ a) \]

\[ ([f \times c], \overline{env}, \overline{mem}, \mu) \]
Example: Inlining

$$\overline{\text{mem}}(\overline{\text{env}}[f]) = \{(\overline{\lambda v \ k. (k \ a)}, \overline{\text{env}})\}$$

$$(\overline{[(f \times c)]}, \overline{\text{env}}, \overline{\text{mem}}, \mu)$$
Example: Inlining

\[ \overline{\text{mem}}(\overline{\text{env}}[f]) = \{ (\overline{\lambda v \ k. (k \ a)}, \overline{\text{env}}) \} \]

(\[ (f \times c) \], \overline{\text{env}}, \overline{\text{mem}}, \mu)

1. \[ \overline{a} = \overline{\text{env}}[a] = \overline{\text{env}}[a] \]
Example: Inlining

\[
\text{mem}(\text{env}[f]) = \{(\lambda v. k.(k \ a), \text{env})\}
\]

\[
((f \times c), \text{env}, \text{mem}, \mu)
\]

1. \(\bar{a} = \text{env}[a] = \text{env}[a]\)

2. \(\mu(\bar{a}) = 1\)
Summary: Counting
Summary: Counting

• Simple
Summary: Counting

- Simple
- Correct
Summary: Counting

- Simple
- Correct
- Worthless
Results: Counting
Results: Counting

% Addresses with $\mu \leq 1$

100

75

50

25

0

early    fringe    stream    lattice    nboyer    perm    doubler    sboyer
Results: Counting

\% Addresses with \( \mu \leq 1 \)
Results: Counting

% Addresses with $\mu \leq 1$

- Early
- Fringe
- Stream
- Lattice
- Nboyer
- Perm
- Doubler
- Sboyer
Abstract garbage collection

Before an abstract transition, discard unreachable structure from the heap.
Example: Zombies & GC

\textit{mem} \quad \textit{mem}
Example: Zombies & GC

\[
\begin{align*}
\text{mem} & \quad \text{mem} \\
\phantom{a} & \phantom{a} \\
\phantom{a} & \phantom{a} \\
\phantom{a} & \phantom{a} \\
\phantom{a} & \phantom{a} \\
\end{align*}
\]

\[
\begin{align*}
 a_1 \\
 a_2 \\
 a_3 \\
\end{align*}
\]
Example: Zombies & GC

\[ \text{mem} \]

- \( a_1 \)
- \( a_2 \)
- \( a_3 \)

\[ \text{mem} \]

- \( \bar{a}_{1,2} \)
- \( \bar{a}_3 \)
Example: Zombies & GC

mem

\[ \text{mem} \]

\[ a_1 \]

\[ a_2 \]

\[ a_3 \]

\[ \bar{a}_{1,2} \]

\[ \bar{a}_3 \]
Example: Zombies & GC

\[
\begin{align*}
\text{mem} & \quad a_1 \\
& \quad a_2
\end{align*}
\]

\[
\begin{align*}
\bar{a}_{1,2} & \\
\text{mem} & \quad a_3 \\
& \quad \bar{a}_3
\end{align*}
\]
Example: Zombies & GC

\[
\begin{align*}
mem & \\
\rightarrow & a_1 & \rightarrow a_1,2 \\
\rightarrow & a_2 & \rightarrow \bar{a}_3 \\
\rightarrow & a_3 & \rightarrow \bar{a}_1,2 \\
\end{align*}
\]
Example: Zombies & GC

\[ \text{mem} \]

\[ a_1 \]

\[ a_2 \]

\[ a_3 \]

\[ \tilde{a}_{1,2} \]

\[ \tilde{a}_3 \]
Example: Zombies & GC

\[ \text{mem} \quad a_1 \quad \tilde{a}_{1,2} \quad \text{mem} \]

\[ \text{mem} \quad a_2 \quad a_3 \quad \tilde{a}_3 \quad \text{mem} \]
Example: Zombies & GC
Example: Zombies & GC
Example: Zombies & GC

\[ mem \]

\[ a_1 \]

\[ a_2 \]

\[ a_3 \]

\[ \tilde{a}_{1,2} \]

\[ \tilde{a}_3 \]
Example: Zombies & GC

\[ \text{mem} \]

\[ a_1 \]

\[ \text{mem} \]

\[ a_2 \] \quad \text{\ldots} \quad \tilde{a}_{1,2}

\[ a_3 \] \quad \text{\ldots} \quad \tilde{a}_3

\[ \text{mem} \]
Effect of Abstract GC
Effect of Abstract GC
Effect of Abstract GC
Vicious cycle

Merging (⊔)

Forking (∈)
Virtuous cycle

Un-merging

No forking
Results: Analysis time

- Normal
- With abstract garbage collection
Results: Analysis time

- Normal
- With abstract garbage collection

<table>
<thead>
<tr>
<th>Task</th>
<th>Normal Time</th>
<th>With GC Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>earley</td>
<td>260s</td>
<td></td>
</tr>
<tr>
<td>fringe</td>
<td>65s</td>
<td></td>
</tr>
<tr>
<td>stream</td>
<td>0s</td>
<td></td>
</tr>
<tr>
<td>lattice</td>
<td>65s</td>
<td></td>
</tr>
<tr>
<td>nboyer</td>
<td>130s</td>
<td></td>
</tr>
<tr>
<td>perm</td>
<td>195s</td>
<td></td>
</tr>
<tr>
<td>doubler</td>
<td>0s</td>
<td></td>
</tr>
<tr>
<td>sboyer</td>
<td>260s</td>
<td></td>
</tr>
</tbody>
</table>
Results: Analysis time

![Bar chart showing analysis time for different categories.](chart)

Legend:
- **Normal**
- **With abstract garbage collection**
Environment analysis is not enough.
Example: Overflow

\[ a[i] \]
Example: Overflow

Can we prove that $i$ is in bounds?
Example: Overflow

```plaintext
let loop i =
  if i < length a
  then f(a[i]) ;
    loop (i+1)
  else ()
in loop 0
```

Can we prove that \(i\) is in bounds?
Example: Overflow

```javascript
indices.each(\(\lambda i. f(a[i])\))
```

Can we prove that \(i\) is in bounds?
Example: Overflow

\[ \lambda(a,i). f(a[i]) \]

Can we prove that \( i \) is in bounds?
Example: Overflow

Can we prove that $i$ is in bounds?

Harder than “what flows here?”

$\lambda(a, i). f(a[i])$
Logic-flow analysis (POPL 2007)
Logic-flow analysis (POPL 2007)

• Abstract states directly ($\hat{s}$)
Logic-flow analysis (POPL 2007)

- Abstract states directly ($\hat{s}$)
- Abstract states to sets of propositions ($\Pi$)
Logic-flow analysis (POPL 2007)

- Abstract states directly ($\hat{s}$)
- Abstract states to sets of propositions ($\Pi$)
- Conduct joint abstract interpretation
Logic-flow analysis (POPL 2007)

- Abstract states directly ($\hat{s}$)
- Abstract states to sets of propositions ($\Pi$)
- Conduct joint abstract interpretation
Logic-flow analysis (POPL 2007)

- Abstract states directly ($\hat{s}$)
- Abstract states to sets of propositions ($\Pi$)
- Conduct joint abstract interpretation
The Palsberg paradigm

“Higher-order analysis =
   first-order analysis + control-flow analysis.”

Jens Palsberg
The post-Palsberg paradigm

“Higher-order analysis =
  first-order analysis × control-flow analysis.”
My research in the wild

- Optimization: Scheme 48 (Knauel)
- Optimization: Waterloo (Zwarich)
- Coroutines: MLton (Chambers, Harvey)
- Security: LLVM, C/C++ (Diagis)
Thank you
Ongoing work

• Automatic parallelization
• Polyvariance completeness theorem
• Anodizing semantics
• Abstractions of Peano pointer arithmetic
• Soundness modulo congruence