Control-Based Program Analysis

Zachary Palmer and Scott F. Smith

Swarthmore College and The Johns Hopkins University

November 23rd, 2015
Outline

- CBA Overview
- CBA by Example
- Properties of CBA
- Comparison to Related Analyses
- Implementation
- Conclusions
Outline

- CBA Overview
- CBA by Example
- Properties of CBA
- Comparison to Related Analyses
- Implementation
- Conclusions
CBA

- Incrementally builds control-flow graph (CFG)
CBA

- Incrementally builds control-flow graph (CFG)
  - Trivial for first-order programs
CBA

- Incrementally builds control-flow graph (CFG)
  - Trivial for first-order programs
  - Higher-order programs: control flow and data flow interact
CBA

- Incrementally builds control-flow graph (CFG)
  - Trivial for first-order programs
  - Higher-order programs: control flow and data flow interact
- Initial graph has no call/return edges
CBA

- Incrementally builds control-flow graph (CFG)
  - Trivial for first-order programs
  - Higher-order programs: control flow and data flow interact
- Initial graph has no call/return edges
- Add call/return edges as discovered
CBA

- Incrementally builds control-flow graph (CFG)
  - Trivial for first-order programs
  - Higher-order programs: control flow and data flow interact
- Initial graph has no call/return edges
- Add call/return edges as discovered
  - Determine which function arrives at call site
CBA

- Incrementally builds control-flow graph (CFG)
  - Trivial for first-order programs
  - Higher-order programs: control flow and data flow interact
- Initial graph has no call/return edges
- Add call/return edges as discovered
  - Determine which function arrives at call site
  - All values looked up relative to point in CFG
CBA

- Incrementally builds control-flow graph (CFG)
  - Trivial for first-order programs
  - Higher-order programs: control flow and data flow interact
- Initial graph has no call/return edges
- Add call/return edges as discovered
  - Determine which function arrives at call site
  - All values looked up relative to point in CFG
  - Relative lookup yields flow-sensitive analysis
CBA

- Incrementally builds control-flow graph (CFG)
  - Trivial for first-order programs
  - Higher-order programs: control flow and data flow interact
- Initial graph has no call/return edges
- Add call/return edges as discovered
  - Determine which function arrives at call site
  - All values looked up relative to point in CFG
  - Relative lookup yields flow-sensitive analysis
- CFG is the only data structure
CBA

- Incrementally builds control-flow graph (CFG)
  - Trivial for first-order programs
  - Higher-order programs: control flow and data flow interact
- Initial graph has no call/return edges
- Add call/return edges as discovered
  - Determine which function arrives at call site
  - All values looked up relative to point in CFG
  - Relative lookup yields flow-sensitive analysis
- CFG is the only data structure
  - No abstract environment or store
CBA

- Incrementally builds control-flow graph (CFG)
  - Trivial for first-order programs
  - Higher-order programs: control flow and data flow interact
- Initial graph has no call/return edges
- Add call/return edges as discovered
  - Determine which function arrives at call site
  - All values looked up relative to point in CFG
  - Relative lookup yields flow-sensitive analysis
- CFG is the only data structure
  - No abstract environment or store
  - So, variable lookup only needs CFG
Outline

- CBA Overview
- CBA by Example
- Properties of CBA
- Comparison to Related Analyses
- Implementation
- Conclusions
A Very Simple Example

1 `let id x = x;;`
2 `let s1 = id 1;;`
3 `let s2 = id 2;;`
A Very Simple Example

1 let id x = x;;
2 let s1 = id 1;;
3 let s2 = id 2;;

⇓

A-normalization

1 id = fun x -> ( ret = x;
2      );
3 n1 = 1;
4 s1 = id n1;
5 n2 = 2;
6 s2 = id n2;
A Very Simple Example

1 let id x = x;;
2 let s1 = id 1;;
3 let s2 = id 2;;

⇓

A-normalization

1 id = fun x -> (
2 ret = x;
3 );
4 n1 = 1;
5 s1 = id n1;
6 n2 = 2;
7 s2 = id n2;

Initial graph:
A Very Simple Example
Graph closure

```
id = fun x -> ( ret = x; );
n1 = 1;
s1 = id n1;
n2 = 2;
s2 = id n2;
```
A Very Simple Example
Graph closure for call site s1

```
id = fun x -> ( ret = x; );
n1 = 1;
s1 = id n1;
n2 = 2;
s2 = id n2;
```
A Very Simple Example
Graph closure for call site s1
Look backward to find function id

```
id = fun x -> ( ret = x; );
n1 = 1;
s1 = id n1;
n2 = 2;
s2 = id n2;
```
A Very Simple Example

Graph closure for call site s1
Look backward to find function id

1 \text{id} = \text{fun} \ x \rightarrow ( \text{ret} = x; );
2 \text{n1} = 1;
3 \text{s1} = \text{id} \ \text{n1};
4 \text{n2} = 2;
5 \text{s2} = \text{id} \ \text{n2};
A Very Simple Example
Graph closure for call site s1
Look backward to find function id

```
1  id = fun x -> ( ret = x; );
2  n1 = 1;
3  s1 = id n1;
4  n2 = 2;
5  s2 = id n2;
```
A Very Simple Example

Graph closure for call site s1
Bind argument n1 to parameter x

1 id = fun x -> ( ret = x; );
2 n1 = 1;
3 s1 = id n1;
4 n2 = 2;
5 s2 = id n2;
A Very Simple Example

Graph closure for call site s1
Assign result ret to call site z1

1 \text{id} = \text{fun} \ x \rightarrow ( \text{ret} = x; );
2 \text{n1} = 1;
3 \text{s1} = \text{id} \ \text{n1};
4 \text{n2} = 2;
5 \text{s2} = \text{id} \ \text{n2};
A Very Simple Example

Graph closure for call site $s_2$

```
1  id = fun x -> ( ret = x; );
2  n1 = 1;
3  s1 = id n1;
4  n2 = 2;
5  s2 = id n2;
```
A Very Simple Example
Graph closure for call site s2
Look backward to find function id

```
1 id = fun x -> ( ret = x; );
2 n1 = 1;
3 s1 = id n1;
4 n2 = 2;
5 s2 = id n2;
```
A Very Simple Example

Graph closure for call site s2
Look backward to find function id

```
1  id = fun x -> ( ret = x; );
2  n1 = 1;
3  s1 = id n1;
4  n2 = 2;
5  s2 = id n2;
```
A Very Simple Example

Graph closure for call site s2
Bind argument n2 to parameter x

```
1 id = fun x -> ( ret = x; );
2 n1 = 1;
3 s1 = id n1;
4 n2 = 2;
5 s2 = id n2;
```
A Very Simple Example

Graph closure for call site s2
Assign result ret to call site z2

```
id = fun x -> ( ret = x; );
n1 = 1;
s1 = id n1;
n2 = 2;
s2 = id n2;
```
A Very Simple Example

Closure complete!

```
id = fun x -> ( ret = x; );
n1 = 1;
s1 = id n1;
n2 = 2;
s2 = id n2;
```
Lookup: Related Work

- Lookup is temporarily reversed and on demand
Lookup: Related Work

- Lookup is temporally reversed and on demand
- Similar to demand-driven CFL-reachability [HRS-FSE95]
  - CFL-reachability research limited to first-order programs
Lookup: Related Work

- Lookup is **temporally reversed** and **on demand**
- Similar to demand-driven CFL-reachability [HRS-FSE95]
  - CFL-reachability research limited to first-order programs
- CBA brings on-demand lookup to higher-order analyses
Lookup: Related Work

- Lookup is **temporally reversed** and **on demand**
- Similar to demand-driven CFL-reachability [HRS-FSE95]
  - CFL-reachability research limited to first-order programs
- CBA brings on-demand lookup to higher-order analyses
- Challenges:
Lookup: Related Work

- Lookup is temporarily reversed and on demand
- Similar to demand-driven CFL-reachability [HRS-FSE95]
  - CFL-reachability research limited to first-order programs
- CBA brings on-demand lookup to higher-order analyses
- Challenges:
  - Polyvariance
Lookup: Related Work

- Lookup is **temporally reversed** and **on demand**
- Similar to demand-driven CFL-reachability [HRS-FSE95]
  - CFL-reachability research limited to first-order programs
- CBA brings on-demand lookup to higher-order analyses
- Challenges:
  - Polyvariance
  - Non-local variables
Call Stack Alignment
Goal: polymorphism

```
id = fun x -> ( ret = x; );
n1 = 1;
s1 = id n1;
n2 = 2;
s2 = id n2;
```
Call Stack Alignment
Goal: polymorphism
Look up s2 from end of program

1 id = fun x -> ( ret = x; );
2 n1 = 1;
3 s1 = id n1;
4 n2 = 2;
5 s2 = id n2;
Call Stack Alignment

Goal: polymorphism

Look up s2 from end of program

1. \texttt{id} = \texttt{fun} x \rightarrow (\texttt{ret} = x; )
2. \texttt{n1} = 1;
3. \texttt{s1} = \texttt{id} \texttt{n1};
4. \texttt{n2} = 2;
5. \texttt{s2} = \texttt{id} \texttt{n2};
Call Stack Alignment
Goal: polymorphism
Look up s2 from end of program

```
1 id = fun x -> ( ret = x; );
2 n1 = 1;
3 s1 = id n1;
4 n2 = 2;
5 s2 = id n2;
```
Call Stack Alignment
Goal: polymorphism
Look up s2 from end of program

id = fun x -> ( ret = x; );
n1 = 1;
s1 = id n1;
n2 = 2;
s2 = id n2;
Call Stack Alignment
Goal: polymorphism
Look up s2 from end of program

1  id = fun x -> ( ret = x; );
2  n1 = 1;
3  s1 = id n1;
4  n2 = 2;
5  s2 = id n2;
Call Stack Alignment
Goal: polymorphism
Look up s2 from end of program

```
1  id = fun x -> ( ret = x; );
2  n1 = 1;
3  s1 = id n1;
4  n2 = 2;
5  s2 = id n2;
```
Call Stack Alignment
Goal: polymorphism
Look up s2 from end of program

1  id = fun x -> ( ret = x; );
2  n1 = 1;
3  s1 = id n1;
4  n2 = 2;
5  s2 = id n2;
Call Stack Alignment
Goal: polymorphism
Look up s2 from end of program

```
1 id = fun x -> ( ret = x; );
2 n1 = 1;
3 s1 = id n1;
4 n2 = 2;
5 s2 = id n2;
```
Call Stack Alignment
We need to match calls and returns.

```plaintext
1  id = fun x -> ( ret = x; );
2  n1 = 1;
3  s1 = id n1;
4  n2 = 2;
5  s2 = id n2;
```
Call Stack Alignment

We need to match calls and returns.
Annotate wiring nodes with call sites

```
1  id = fun x -> ( ret = x; );
2  n1 = 1;
3  s1 = id n1;
4  n2 = 2;
5  s2 = id n2;
```
Call Stack Alignment
We need to match calls and returns.
Maintain call stack during lookup

```
1  id = fun x -> ( ret = x; );
2  n1 = 1;
3  s1 = id n1;
4  n2 = 2;
5  s2 = id n2;
```
Call Stack Alignment
We need to match calls and returns.
Maintain call stack during lookup

```
1  id = fun x -> ( ret = x; );
2  n1 = 1;
3  s1 = id n1;
4  n2 = 2;
5  s2 = id n2;
```
Call Stack Alignment

We need to match calls and returns.

Spurious results filtered by call stack

```
1 id = fun x -> ( ret = x; );
2 n1 = 1;
3 s1 = id n1;
4 n2 = 2;
5 s2 = id n2;
```
Call Stack Alignment

**We need to match calls and returns.**
Spurious results filtered by call stack

```ocaml
1  id = fun x -> ( ret = x; );
2  n1 = 1;
3  s1 = id n1;
4  n2 = 2;
5  s2 = id n2;
```
Call Stack Alignment

We need to match calls and returns.

Here, 1 is eliminated

```
1  id = fun x -> ( ret = x; );
2  n1 = 1;
3  s1 = id n1;
4  n2 = 2;
5  s2 = id n2;
```
Call Stack Alignment: Related Work

- Model control flow as a PDA
Call Stack Alignment: Related Work

- Model control flow as a PDA
- Call stack alignment induces polyvariance!
Call Stack Alignment: Related Work

- Model control flow as a PDA
- Call stack alignment induces polyvariance!
- Long history of this approach in program analysis
Call Stack Alignment: Related Work

- Model control flow as a PDA
- Call stack alignment induces polyvariance!
- Long history of this approach in program analysis
  - CFL-reachability analyses: calls and returns modeled as CFL
Call Stack Alignment: Related Work

- Model control flow as a PDA
- Call stack alignment induces polyvariance!
- Long history of this approach in program analysis
  - CFL-reachability analyses: calls and returns modeled as CFL
- CFA2 [VS-ESOP10] and PDCFA [MSV-PLDI10]: align calls and returns via PDA
Call Stack Alignment: Related Work

- Model control flow as a PDA
- Call stack alignment induces polyvariance!
- Long history of this approach in program analysis
  - CFL-reachability analyses: calls and returns modeled as CFL
- CFA2 [VS-ESOP10] and PDCFA [MSV-PLDI10]: align calls and returns via PDA
  - PDA is precisely an abstract interpreter
Handling Non-Local Variables

Non-local example: K-combinator

```ocaml
let k v j = v;;
let f = k 1;;
let g = k 2;;
let s = f 0;;
```
Handling Non-Local Variables

Non-local example: K-combinator

```
let k v j = v;;
let f = k 1;;
let g = k 2;;
let s = f 0;;
```

A-normalization

```
let k v j = v;;
let f = k 1;;
let g = k 2;;
let s = f 0;;
```

```
k = fun v -> (k0 = fun j -> (r = v;;));
a = 1;  f = k a;
b = 2;  g = k b;
z = 0;  s = f z;
```
k = fun v -> (k0 = fun j -> (r = v;));

a = 1;  f = k a;
b = 2;  g = k b;
z = 0;  s = f z;
Handling Non-Local Variables

Perform closure

...for call site $f$.

```
1 k = fun v -> (k0 = fun j -> (r = v;));
2 a = 1;  f = k a;
3 b = 2;  g = k b;
4 z = 0;  s = f z;
```
Handling Non-Local Variables

Perform closure

...for call site g.

1. \( k = \text{fun } v \rightarrow (k0 = \text{fun } j \rightarrow (r = v));); \)
2. \( a = 1;\quad f = k\; a;\)
3. \( b = 2;\quad g = k\; b;\)
4. \( z = 0;\quad s = f\; z;\)
Handling Non-Local Variables
Perform closure
...for call site s.

1. \( k = \textbf{fun} \ v \rightarrow (k0 = \textbf{fun} \ j \rightarrow (r = v;));; \)
2. \( a = 1; \ f = k \ a; \)
3. \( b = 2; \ g = k \ b; \)
4. \( z = 0; \ s = f \ z; \)
Handling Non-Local Variables
Non-locals require careful handling
Look up s from end of program.

1. \( k = \text{fun} \ v \rightarrow (k0 = \text{fun} \ j \rightarrow (r = v);); \)
2. \( a = 1; \quad f = k \ a; \)
3. \( b = 2; \quad g = k \ b; \)
4. \( z = 0; \quad s = f \ z; \)
Handling Non-Local Variables
Non-locals require careful handling
Look up s from end of program.

1. \( k = \text{fun } v \rightarrow (k0 = \text{fun } j \rightarrow (r = v);); \)
2. \( a = 1; \quad f = k \ a; \)
3. \( b = 2; \quad g = k \ b; \)
4. \( z = 0; \quad s = f \ z; \)
Handling Non-Local Variables
Non-locals require careful handling
Look up s from end of program.

1. \( k = \text{fun} \ v \rightarrow (k0 = \text{fun} \ j \rightarrow (r = v);); \)
2. \( a = 1; \ f = k \ a; \)
3. \( b = 2; \ g = k \ b; \)
4. \( z = 0; \ s = f \ z; \)
Handling Non-Local Variables
Non-locals require careful handling
Look up s from end of program.

1  k = fun v -> (k0 = fun j -> (r = v;;));
2  a = 1;  f = k a;
3  b = 2;  g = k b;
4  z = 0;  s = f z;
Handling Non-Local Variables
Non-locals require careful handling

```
1  k = fun v -> (k0 = fun j -> (r = v;;));
2  a = 1;  f = k a;
3  b = 2;  g = k b;
4  z = 0;  s = f z;
```
Handling Non-Local Variables
Non-locals require careful handling

k = fun v -> (k0 = fun j -> (r = v;;));
1 a = 1;  f = k a;
2 b = 2;  g = k b;
3 z = 0;  s = f z;
Handling Non-Local Variables

Non-locals require careful handling

```
1 k = fun v -> (k0 = fun j -> (r = v,));
2 a = 1;  f = k a;
3 b = 2;  g = k b;
4 z = 0;  s = f z;
```
Handling Non-Local Variables
Non-locals require careful handling

1. \( k = \text{fun } v \rightarrow (k0 = \text{fun } j \rightarrow (r = v;)); \)
2. \( a = 1; \ f = k \ a; \)
3. \( b = 2; \ g = k \ b; \)
4. \( z = 0; \ s = f \ z; \)
Handling Non-Local Variables
Non-locals require careful handling

1. \( k = \text{fun } v \rightarrow (k0 = \text{fun } j \rightarrow (r = v);); \)
2. \( a = 1; \quad f = k \ a; \)
3. \( b = 2; \quad g = k \ b; \)
4. \( z = 0; \quad s = f \ z; \)
Handling Non-Local Variables
Non-locals require careful handling

1. \( k = \text{fun} \ v \rightarrow (\text{k0} = \text{fun} \ j \rightarrow (r = v;)); \)
2. \( a = 1; \ f = k\ a; \)
3. \( b = 2; \ g = k\ b; \)
4. \( z = 0; \ s = f\ z; \)
Handling Non-Local Variables

- Even with call stack alignment, non-locals are hard
Handling Non-Local Variables

- Even with call stack alignment, non-locals are hard
- When looking for non-local, must find definition of its closure
Handling Non-Local Variables

- Even with call stack alignment, non-locals are hard
- When looking for non-local, must find definition of its closure
  - Search for closure; then, resume looking for non-local
Handling Non-Local Variables

- Even with call stack alignment, non-locals are hard
- When looking for non-local, must find definition of its closure
  - Search for closure; then, resume looking for non-local
- Implementation: stack of lookup operations
Handling Non-Local Variables

- Even with call stack alignment, non-locals are hard.
- When looking for non-local, must find definition of its closure:
  - Search for closure; then, resume looking for non-local.
- Implementation: stack of lookup operations.
- 2-stack PDA encodes a Turing machine. 😞
Handling Non-Local Variables

- Even with call stack alignment, non-locals are hard
- When looking for non-local, must find definition of its closure
  - Search for closure; then, resume looking for non-local
- Implementation: stack of lookup operations
- 2-stack PDA encodes a Turing machine. 😞
  - Our solution: finitize call stack; keep full lookup stack.
Handling Non-Local Variables

- Even with call stack alignment, non-locals are hard
- When looking for non-local, must find definition of its closure
  - Search for closure; then, resume looking for non-local
- Implementation: stack of lookup operations
- 2-stack PDA encodes a Turing machine. 😞
  - Our solution: finitize call stack; keep full lookup stack.
  - $k$CBA: maximum call stack depth $k$
Outline

- CBA Overview
- CBA by Example
- Properties of CBA
- Comparison to Related Analyses
- Implementation
- Conclusions
Properties of CBA

- Theorem: \( k \text{CBA (for fixed } k \text{) has polynomial time bound} \)
Properties of CBA

- Theorem: $k$CBA (for fixed $k$) has polynomial time bound
  - Program of size $n$

- Lemma: CBA is monotonic
  - Control flow graph: $G$
  - Lookup: $L(x, p, G)$ for var $x$ at program point $p$ in graph $G$
  - Monotonicity: $G_1 \subseteq G_2 \Rightarrow L(x, p, G_1) \subseteq L(x, p, G_2)$

Delightful mathematical property; huge win for optimization!
Properties of CBA

Theorem: $k$CBA (for fixed $k$) has polynomial time bound
- Program of size $n$
- New wiring nodes: $O(n^2)$

Lemma: CBA is monotonic
- Control flow graph: $G$
- Lookup: $L(x, p, G)$ for var $x$ at program point $p$ in graph $G$
- Monotonicity: $G_1 \subseteq G_2 \Rightarrow L(x, p, G_1) \subseteq L(x, p, G_2)$

Delightful mathematical property; huge win for optimization!
Properties of CBA

- Theorem: $k$CBA (for fixed $k$) has polynomial time bound
  - Program of size $n$
  - New wiring nodes: $O(n^2)$
  - Therefore, graph size $g$ is $O(n^2)$
Properties of CBA

- **Theorem:** \( k \text{CBA} \) (for fixed \( k \)) has polynomial time bound
  
  - Program of size \( n \)
  - New wiring nodes: \( O(n^2) \)
  - Therefore, graph size \( g \) is \( O(n^2) \)
  - Lookup: PDA of size \( O(g^{k+1}) \)
Properties of CBA

- Theorem: \( k\text{CBA} \) (for fixed \( k \)) has polynomial time bound
  - Program of size \( n \)
  - New wiring nodes: \( O(n^2) \)
  - Therefore, graph size \( g \) is \( O(n^2) \)
  - Lookup: PDA of size \( O(g^{k+1}) \) (with constant \( k \))
Properties of CBA

- **Theorem:** $k$CBA (for fixed $k$) has polynomial time bound
  - Program of size $n$
  - New wiring nodes: $O(n^2)$
  - Therefore, graph size $g$ is $O(n^2)$
  - Lookup: PDA of size $O(g^{k+1})$ (with constant $k$)

- **Lemma:** CBA is monotonic
Properties of CBA

- Theorem: $k$CBA (for fixed $k$) has polynomial time bound
  - Program of size $n$
  - New wiring nodes: $O(n^2)$
  - Therefore, graph size $g$ is $O(n^2)$
  - Lookup: PDA of size $O(g^{k+1})$ (with constant $k$)

- Lemma: CBA is monotonic
  - Control flow graph: $G$
Properties of CBA

- Theorem: \( k \text{CBA} \) (for fixed \( k \)) has polynomial time bound
  - Program of size \( n \)
  - New wiring nodes: \( O(n^2) \)
  - Therefore, graph size \( g \) is \( O(n^2) \)
  - Lookup: PDA of size \( O(g^{k+1}) \) (with constant \( k \))

- Lemma: CBA is monotonic
  - Control flow graph: \( G \)
  - Lookup: \( L(x, p, G) \) for var \( x \) at program point \( p \) in graph \( G \)
Properties of CBA

- **Theorem:** $k$CBA (for fixed $k$) has polynomial time bound
  - Program of size $n$
  - New wiring nodes: $O(n^2)$
  - Therefore, graph size $g$ is $O(n^2)$
  - Lookup: PDA of size $O(g^{k+1})$ (with constant $k$)

- **Lemma:** CBA is monotonic
  - Control flow graph: $G$
  - Lookup: $L(x, p, G)$ for var $x$ at program point $p$ in graph $G$
  - Monotonicity: $G_1 \subseteq G_2 \implies L(x, p, G_1) \subseteq L(x, p, G_2)$
Properties of CBA

- **Theorem:** \( k \text{CBA} \) (for fixed \( k \)) has polynomial time bound
  - Program of size \( n \)
  - New wiring nodes: \( O(n^2) \)
  - Therefore, graph size \( g \) is \( O(n^2) \)
  - Lookup: PDA of size \( O(g^{k+1}) \) (with constant \( k \))

- **Lemma:** CBA is monotonic
  - Control flow graph: \( G \)
  - Lookup: \( L(x, p, G) \) for var \( x \) at program point \( p \) in graph \( G \)
  - Monotonicity: \( G_1 \subseteq G_2 \implies L(x, p, G_1) \subseteq L(x, p, G_2) \)
  - Delightful mathematical property; huge win for optimization!
Outline

- CBA Overview
- CBA by Example
- Properties of CBA
- Comparison to Related Analyses
- Implementation
- Conclusions
CBA and CFA

- $k$CFA: exponential time for $k > 0$, but no non-local complications
CBA and CFA

- $k$CFA: exponential time for $k > 0$, but no non-local complications
- Conjecture:
kCFA: exponential time for $k > 0$, but no non-local complications

Conjecture:
  - Suppose program with max lexical nesting depth $c$
CBA and CFA

- $k$CFA: exponential time for $k > 0$, but no non-local complications
- Conjecture:
  - Suppose program with max lexical nesting depth $c$
  - $(k + c)$CBA strictly more expressive than $k$CFA
CBA and PDCFA

- PDCFA probably closest in expressiveness
CBA and PDCFA

- PDCFA probably closest in expressiveness
- **Lookup**
  - PDCFA: Push abstract envs forward; GC limits states
  - CBA: Look back through CFG to find values; no abstract env
CBA and PDCFA

- **PDCFA** probably closest in expressiveness

  **Lookup**
  - PDCFA: Push abstract envs forward; GC limits states
  - CBA: Look back through CFG to find values; no abstract env

  **Stack Alignment**
  - PDCFA: Use PDA for call stack; limit to regexes in practice
  - CBA: Embed finitization of call stack in PDA nodes
  - Appear to have similar expressiveness
CBA and PDCFA

- PDCFA probably closest in expressiveness

  **Lookup**
  - PDCFA: Push abstract envs forward; GC limits states
  - CBA: Look back through CFG to find values; no abstract env

  **Stack Alignment**
  - PDCFA: Use PDA for call stack; limit to regexes in practice
  - CBA: Embed finitization of call stack in PDA nodes
  - Appear to have similar expressiveness

- Polyvariance
  - PDCFA: classic CFA-like graph copying
  - CBA: via call stack alignment and non-local lookup
Outline

- CBA Overview
- CBA by Example
- Properties of CBA
- Comparison to Related Analyses
- Implementation
- Conclusions
Towards a Real Implementation

- Formal definition of further language features
  - Records
  - Path-sensitivity: filters validated by PDA
  - State

Reference implementation on GitHub (slow)
Optimized implementation under development

Uses monotonicity lemma: same lazy PDA for all lookups
Towards a Real Implementation

- Formal definition of further language features
  - Records
  - Path-sensitivity: filters validated by PDA
  - State
- Reference implementation on GitHub (slow)
Towards a Real Implementation

- Formal definition of further language features
  - Records
  - Path-sensitivity: filters validated by PDA
  - State
- Reference implementation on GitHub (slow)
- Optimized implementation under development
Towards a Real Implementation

- Formal definition of further language features
  - Records
  - Path-sensitivity: filters validated by PDA
  - State
- Reference implementation on GitHub (slow)
- Optimized implementation under development
  - Uses monotonicity lemma: same lazy PDA for all lookups
Outline

- CBA Overview
- CBA by Example
- Properties of CBA
- Comparison to Related Analyses
- Implementation
- Conclusions
Conclusions

- CBA is interesting and worth studying!
Conclusions

- CBA is interesting and worth studying!
- Not claiming strictly better, but very different
Conclusions

- CBA is interesting and worth studying!
- Not claiming strictly better, but very different
- May be suitable to particular applications
Conclusions

- CBA is interesting and worth studying!
- Not claiming strictly better, but very different
- May be suitable to particular applications
  - No abstract environment: could make concurrency easier
Conclusions

- CBA is interesting and worth studying!
- Not claiming strictly better, but very different
- May be suitable to particular applications
  - No abstract environment: could make concurrency easier
  - Path-sensitivity model: possible theorem-proving applications
Questions?

Example of $k$CBA Imprecision

Consider code:

```plaintext
1 let f x = x;;
2 let g y = f y;;
3 let a = g 1;;
4 let b = g 2;;
```
Example of $k$CBA Imprecision

Consider code:

```
let f x = x;;
let g y = f y;;
let a = g 1;;
let b = g 2;;
```

1CBA: $a \subseteq \{1, 2\}$
Example of $k$CBA Imprecision

Consider code:

1. `let f x = x;;`
2. `let g y = f y;;`
3. `let a = g 1;;`
4. `let b = g 2;;`

- 1CBA: $a \subseteq \{1, 2\}$
  - From within $f$, we can't remember where $g$ was called
Example of $k$CBA Imprecision

- Consider code:

```ml
1 let f x = x;;
2 let g y = f y;;
3 let a = g 1;;
4 let b = g 2;;
```

- $1$CBA: $a \subseteq \{1, 2\}$
  - From within $f$, we can't remember where $g$ was called

- $1$CFA: same problem
Example of $k$CBA Imprecision

- Consider code:

```plaintext
1 let f x = x;;
2 let g y = f y;;
3 let a = g 1;;
4 let b = g 2;;
```

- **1CBA**: $a \subseteq \{1, 2\}$
  - From within $f$, we can't remember where $g$ was called

- **1CFA**: same problem

- **2CBA**: $a \subseteq \{1\}$
**Example of $k$CBA Imprecision**

- Consider code:

  ```ocaml
  let f x = x;;
  let g y = f y;;
  let a = g 1;;
  let b = g 2;;
  ```

  - **1CBA:** $a \subseteq \{1, 2\}$
    - From within $f$, we can't remember where $g$ was called
  - **1CFA:** same problem
  - **2CBA:** $a \subseteq \{1\}$
  - Alternative CBA call stack finitizations exist (e.g. regex)
Example of $k$CBA Imprecision

- Consider code:

```ml
1 let f x = x;;
2 let g y = f y;;
3 let a = g 1;;
4 let b = g 2;;
```

- **1CBA:** $a \subseteq \{1, 2\}$
  - From within $f$, we can’t remember where $g$ was called

- **1CFA:** same problem

- **2CBA:** $a \subseteq \{1\}$

- Alternative CBA call stack finitizations exist (e.g. regex)
  - Such as used in pushdown-assisted CFA