DDPA
A Higher-Order Demand-Driven Program Analysis

Zachary Palmer$^1$  Scott F. Smith$^2$

Swarthmore College$^1$
The Johns Hopkins University$^2$

July 20th, 2016
### Some Program Analyses

<table>
<thead>
<tr>
<th></th>
<th>push forward</th>
<th>demand-driven</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>first order</strong></td>
<td>abstract interpretation</td>
<td>CFL-reachability</td>
</tr>
<tr>
<td></td>
<td>data flow analysis</td>
<td>reverse data flow analysis</td>
</tr>
<tr>
<td><strong>higher order</strong></td>
<td>kCFA</td>
<td>CFA2</td>
</tr>
<tr>
<td></td>
<td>PDCFA</td>
<td>ΓCFA</td>
</tr>
</tbody>
</table>

**Notes:**
- CFL: Context-Free Languages
- PDCFA: Partially Dynamic Control Flow Analysis
- Γ: Higher order analysis symbol
Some Program Analyses

<table>
<thead>
<tr>
<th>push forward</th>
<th>demand-driven</th>
</tr>
</thead>
<tbody>
<tr>
<td>abstract interpretation</td>
<td>CFL-reachability</td>
</tr>
<tr>
<td>data flow analysis</td>
<td>reverse data flow analysis</td>
</tr>
<tr>
<td>kCFA</td>
<td>CFA2</td>
</tr>
<tr>
<td>PDCFA</td>
<td>ΓCFA</td>
</tr>
</tbody>
</table>

first order

higher order
Some Program Analyses

- push forward
- demand-driven

<table>
<thead>
<tr>
<th>First order</th>
<th>Higher order</th>
</tr>
</thead>
<tbody>
<tr>
<td>abstract interpretation</td>
<td>CFL-reachability</td>
</tr>
<tr>
<td>data flow analysis</td>
<td>reverse data flow analysis</td>
</tr>
<tr>
<td>kCFA</td>
<td>CFA2</td>
</tr>
<tr>
<td>PDCFA</td>
<td>ΓCFA</td>
</tr>
</tbody>
</table>
# Some Program Analyses

<table>
<thead>
<tr>
<th>First Order</th>
<th>Push Forward</th>
<th>Demand-Driven</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract Interpretation</td>
<td>Data Flow Analysis</td>
<td>CFL-Reachability</td>
</tr>
<tr>
<td>Data Flow Analysis</td>
<td></td>
<td>Reverse Data Flow Analysis</td>
</tr>
<tr>
<td>kCFA</td>
<td>CFA2</td>
<td></td>
</tr>
<tr>
<td>PDCFA</td>
<td>ΓCFA</td>
<td></td>
</tr>
</tbody>
</table>
### Some Program Analyses

<table>
<thead>
<tr>
<th></th>
<th>push forward</th>
<th>demand-driven</th>
</tr>
</thead>
<tbody>
<tr>
<td>first order</td>
<td>abstract interpretation</td>
<td>CFL-reachability</td>
</tr>
<tr>
<td></td>
<td>data flow analysis</td>
<td>reverse data flow analysis</td>
</tr>
<tr>
<td>higher order</td>
<td>kCFA</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CFA2</td>
<td>DDPA</td>
</tr>
<tr>
<td></td>
<td>PDCFA</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ΓCFA</td>
<td></td>
</tr>
</tbody>
</table>
## Some Program Analyses

<table>
<thead>
<tr>
<th>First Order</th>
<th>Push Forward</th>
<th>Demand-Driven</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract interpretation</td>
<td>Data flow analysis</td>
<td>CFL-reachability</td>
</tr>
<tr>
<td>Reverse data flow analysis</td>
<td></td>
<td></td>
</tr>
<tr>
<td>kCFA</td>
<td>CFA2</td>
<td>DDPA</td>
</tr>
<tr>
<td>PDCFA</td>
<td>ΓCFA</td>
<td>POLYFLOW$_{CFL}$</td>
</tr>
</tbody>
</table>
let id x = x;;
let s1 = id 1;;
let s2 = id 2;;
DDPA By Example

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

\[\Downarrow\] A-normalize

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

A-normalize

id = fun x -> (ret = x);

n1 = 1;
s1 = id n1;

n2 = 2;
s2 = id n2;

Initial graph before any calls wired:
DDPA By Example

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

Analyze call site s1

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

Analyze 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};
DDPA By Example

Analyze call site `s1`

Look backward to find function `id`

```
id = fun x -> ( ret = x; );
n1 = 1;
s1 = id n1;
n2 = 2;
s2 = id n2;
```
DDPA By Example
Analyze call site s1
Look backward to find function id

\[\begin{align*}
S_T & \rightarrow \text{id} \rightarrow n_1 \rightarrow s_1 \rightarrow n_2 \rightarrow s_2 \rightarrow \text{END}
\end{align*}\]
DDPA By Example

Analyze call site s1

Bind argument n1 to parameter x

```
id = fun x -> ( ret = x; );
n1 = 1;
s1 = id n1;
n2 = 2;
s2 = id n2;
```
DDPA By Example

Analyze call site `s1`
Assign result `ret` to call site `z1`

1  `id = fun x -> ( ret = x; );`
2  `n1 = 1;`
3  `s1 = id n1;`
4  `n2 = 2;`
5  `s2 = id n2;`
id = fun x -> ( ret = x; );
n1 = 1;
s1 = id n1;
n2 = 2;
s2 = id n2;
DDPA By Example
Analyze 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;
DDPA By Example

Analyze call site s2
Look backward to find function id

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} ;
DDPA By Example

Analyze call site s2
Bind argument n2 to parameter x

1 \text{id} = \text{fun} \ x \ -> \ ( \text{ret} = \ x; \ );
2 \ n1 = 1;
3 \ s1 = \text{id} \ n1;
4 \ n2 = 2;
5 \ s2 = \text{id} \ n2;
DDPA By Example

Analyze call site `s2`
Assign result `ret` to call site `z2`

```
1 id = fun x -> ( ret = x; );
2 n1 = 1;
3 s1 = id n1;
4 n2 = 2;
5 s2 = id n2;
```
DDPA By Example
CFG construction complete

```ml
let id = fun x -> ( ret = x; );
let n1 = 1;
let s1 = id n1;
let n2 = 2;
let s2 = id n2;
```
Observe

- Incrementally built control-flow graph (CFG)
Observe

- Incrementally built control-flow graph (CFG)
- Function bodies were wired to call sites as discovered

Challenges:
- Context-sensitivity / polymorphism?
- Non-local variables?
- Recursion?
- Function parameters?
- State and heap aliases?
- Path-sensitivity?

(Flow-sensitivity comes for free)
Observe

- Incrementally built control-flow graph (CFG)
- Function bodies were wired to call sites as discovered
- Analysis focused on variable lookup
  - “What values might variable x have at program point p?”

Challenges:
- Context-sensitivity / polymorphism?
- Non-local variables?
- Recursion?
- Function parameters?
- State and heap aliases?
- Path-sensitivity?
  (Flow-sensitivity comes for free)
Observe

- Incrementally built control-flow graph (CFG)
- Function bodies were wired to call sites as discovered
- Analysis focused on variable lookup
  - “What values might variable x have at program point p?”
- Lookup is temporally reversed and on demand
Observe

- Incrementally built control-flow graph (CFG)
- Function bodies were wired to call sites as discovered
- Analysis focused on variable lookup
  - “What values might variable $x$ have at program point $p$?”
- Lookup is temporally reversed and on demand
- How could this lookup be accurate? Challenges:
Observe

- Incrementally built control-flow graph (CFG)
- Function bodies were wired to call sites as discovered
- Analysis focused on variable lookup
  - “What values might variable x have at program point p?”
- Lookup is temporally reversed and on demand
- How could this lookup be accurate? Challenges:
  - Context-sensitivity / polymorphism?
  - Non-local variables?
  - Recursion?
  - Function parameters?
  - State and heap aliases?
  - Path-sensitivity?
Observe

- Incrementally built control-flow graph (CFG)
- Function bodies were wired to call sites as discovered
- Analysis focused on variable lookup
  - “What values might variable x have at program point p?”
- Lookup is temporally reversed and on demand
- How could this lookup be accurate? Challenges:
  - Context-sensitivity / polymorphism?
  - Non-local variables?
  - Recursion?
  - Function parameters?
  - State and heap aliases?
  - Path-sensitivity?
Observe

- Incrementally built control-flow graph (CFG)
- Function bodies were wired to call sites as discovered
- Analysis focused on variable lookup
  - “What values might variable $x$ have at program point $p$?”
- Lookup is temporally reversed and on demand
- How could this lookup be accurate? Challenges:
  - Context-sensitivity / polymorphism?
  - Non-local variables?
  - Recursion? (brief mention)
  - Function parameters?
  - State and heap aliases?
  - Path-sensitivity?
Observe

- Incrementally built control-flow graph (CFG)
- Function bodies were wired to call sites as discovered
- Analysis focused on variable lookup
  - “What values might variable x have at program point p?”
- Lookup is temporally reversed and on demand
- How could this lookup be accurate? Challenges:
  - Context-sensitivity / polymorphism?
  - Non-local variables?
  - Recursion? (brief mention)
  - Function parameters?
  - State and heap aliases?
  - Path-sensitivity?
  - (Flow-sensitivity comes for free)
Call Stack Alignment for Context-Sensitivity

1 \text{id} = \text{fun} \ x \ \rightarrow \ ( \ \text{ret} = x; \ );
2 \ n1 = 1;
3 \ s1 = \text{id} \ n1;
4 \ n2 = 2;
5 \ s2 = \text{id} \ n2;
Call Stack Alignment for Context-Sensitivity
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 for Context-Sensitivity

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 for Context-Sensitivity

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 for Context-Sensitivity
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 for Context-Sensitivity

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 for Context-Sensitivity

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 for Context-Sensitivity

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 for Context-Sensitivity
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 for Context-Sensitivity
Solution: Maintain call stack during lookup, use to filter

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 for Context-Sensitivity

Solution: Maintain call stack during lookup, use to filter

```
1  id = fun x -> ( ret = x; );
2  n1 = 1;
3  s1 = id n1;
4  n2 = 2;
5  s2 = id n2;
```
Call Stack Alignment for Context-Sensitivity
Solution: Maintain call stack during lookup, use to filter

```ocaml
1 id = fun x -> ( ret = x; );
2 n1 = 1;
3 s1 = id n1;
4 n2 = 2;
5 s2 = id n2;
```
Call Stack Alignment for Context-Sensitivity
Solution: Maintain call stack during lookup, use to filter

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

Solution: Maintain call stack during lookup, use to filter

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

Solution: Maintain call stack during lookup, use to filter

id = fun x -> ( ret = x; );

n1 = 1;
s1 = id n1;
n2 = 2;
s2 = id n2;
Call Stack Alignment for Context-Sensitivity

Solution: Maintain call stack during lookup, use to filter

```plaintext
1  id = fun x -> ( ret = x; );
2  n1 = 1;
3  s1 = id n1;
4  n2 = 2;
5  s2 = id n2;
```
Implementing stack alignment

CFA2/PDCFA: Push-Down System (PDS) for aligning calls and returns
Does not align non-locals; uses context copying for that

POLYFLOW
CFL
Uses CFL/PDA for aligning calls and returns
Not flow-sensitive; lesser precision

DDPA:
Also uses PDS: lookup decision
$\equiv$ automata reachability
PDS stack is not call stack
We need the PDS stack for something else...
Implementing stack alignment

- CFA2/PDCFA:
  - Push-Down System (PDS) for aligning calls and returns
Implementing stack alignment

- **CFA2/PDCFA:**
  - Push-Down System (PDS) for aligning calls and returns
  - Does not align non-locals; uses context copying for that
Implementing stack alignment

- **CFA2/PDCFA:**
  - Push-Down System (PDS) for aligning calls and returns
  - Does not align non-locals; uses context copying for that

- **POLYFLOW\textsubscript{CFL}**
  - Uses CFL/PDA for aligning calls and returns
Implementing stack alignment

- **CFA2/PDCFA:**
  - Push-Down System (PDS) for aligning calls and returns
  - Does not align non-locals; uses context copying for that

- **POLYFLOW\textsubscript{CFL}**
  - Uses CFL/PDA for aligning calls and returns
  - Not flow-sensitive; lesser precision
Implementing stack alignment

- **CFA2/PDCFA:**
  - Push-Down System (PDS) for aligning calls and returns
  - Does not align non-locals; uses context copying for that

- **POLYFLOW\textsubscript{CFL}**
  - Uses CFL/PDA for aligning calls and returns
  - Not flow-sensitive; lesser precision

- **DDPA:**
  - Also uses PDS: lookup decision $\equiv$ automata reachability
Implementing stack alignment

- **CFA2/PDCFA:**
  - Push-Down System (PDS) for aligning calls and returns
  - Does not align non-locals; uses context copying for that

- **POLYFLOW\textsubscript{CFL}**
  - Uses CFL/PDA for aligning calls and returns
  - Not flow-sensitive; lesser precision

- **DDPA:**
  - Also uses PDS: lookup decision $\equiv$ automata reachability
  - PDS stack is not call stack
  - We need the PDS stack for something else...
Handling Non-Local Variables

Non-local example: K-combinator

1 \texttt{let} \ k \ v \ j = v;;
2 \texttt{let} \ f = k \ 1;;
3 \texttt{let} \ g = k \ 2;;
4 \texttt{let} \ s = f \ 0;;
Handling Non-Local Variables

Non-local example: K-combinator

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

A-normalization

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

A-normalization

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

Analyze 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;
```
Non-Local Variable Lookup

Analyze call site g.

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. \( z = 0; \ s = f \ z; \)
Non-Local Variable Lookup

Analyze call site $s$.

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. $z = 0; \ s = f\ z;$
Non-Local Variable Lookup

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. \( z = 0; \ s = f \ z; \)
Non-Local Variable Lookup

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. \( z = 0; \ s = f \ z; \)
Non-Local Variable Lookup
Look up $s$ from end of program.

$$k = \text{fun } v \to (k0 = \text{fun } j \to (r = v;));$$

1. $a = 1$; $f = k\ a$
2. $b = 2$; $g = k\ b$
3. $z = 0$; $s = f\ z$
Non-Local Variable Lookup
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;
```
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;`
Non-Local Variable Lookup

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;
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. \( z = 0; \ s = f \ z; \)
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;
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. \( z = 0; \quad s = f\ z; \)
1  \texttt{k = fun v \rightarrow (k0 = fun j \rightarrow (r = v;));;
2  a = 1; \texttt{f = k a;}
3  b = 2; \texttt{g = k b;}
4  z = 0; \texttt{s = f z;}

Non-Local Variable Lookup
Handling Non-Local Variables

- Search for defining closure; then, resume looking for variable
Handling Non-Local Variables

- Search for defining closure; then, resume looking for variable
- General case: continuation stack needed for non-local lookups
Handling Non-Local Variables

- Search for defining closure; then, resume looking for variable
- General case: continuation stack needed for non-local lookups
- Can't have a 2-stack PDS!
Handling Non-Local Variables

- Search for defining closure; then, resume looking for variable
- General case: continuation stack needed for non-local lookups
- Can’t have a 2-stack PDS!
  - Solution: finitize call stack in PDS nodes; keep full lookup stack.
Handling Non-Local Variables

- Search for defining closure; then, resume looking for variable
- General case: continuation stack needed for non-local lookups
- Can't have a 2-stack PDS!
  - Solution: finitize call stack in PDS nodes; keep full lookup stack.
  - kDDPA: maximum call stack depth $k$
Handling Non-Local Variables

- Search for defining closure; then, resume looking for variable
- General case: continuation stack needed for non-local lookups
- Can’t have a 2-stack PDS!
  - Solution: finitize call stack in PDS nodes; keep full lookup stack.
  - $k$DDPA: maximum call stack depth $k$
  - Lookup still translates to PDS reachability decision problem
Properties

- Recursion: handled by PDS lookup; cycles are fine in a PDS
Properties

- Recursion: handled by PDS lookup; cycles are fine in a PDS
- Theorem: $k$DDPA (for fixed $k$) has polynomial time bound
Properties

- Recursion: handled by PDS lookup; cycles are fine in a PDS
- Theorem: $k$DDPA (for fixed $k$) has polynomial time bound
- Lemma: both CFG and PDS are monotonically increasing over analysis
Properties

- Recursion: handled by PDS lookup; cycles are fine in a PDS
- Theorem: $k$DDPA (for fixed $k$) has polynomial time bound
- Lemma: both CFG and PDS are monotonically increasing over analysis
  - Allows for analysis to be purely additive – efficient sharing
Properties

- Recursion: handled by PDS lookup; cycles are fine in a PDS
- Theorem: $k$DDPA (for fixed $k$) has polynomial time bound
- Lemma: both CFG and PDS are monotonically increasing over analysis
  - Allows for analysis to be purely additive – efficient sharing
  - Observe we have reduced program analysis to incremental (PDS) model checking - fast!
1 \( x = \{\} \); 
2 \( f = \text{fun } i \rightarrow ( r = i ) \); 
3 \( z = f \, x \);

- Build both CFG and PDS incrementally; above is final result
Forward and reverse analyses compared

- Forward analyses are more natural to derive from operational semantics
Forward and reverse analyses compared

- Forward analyses are more natural to derive from operational semantics
- DDPA is demand-driven and model-checking: potentially more efficient than forward analyses
Forward and reverse analyses compared

- Forward analyses are more natural to derive from operational semantics
- DDPA is demand-driven and model-checking: potentially more efficient than forward analyses
- \( k \) in \( k\text{DDPA} \) not directly comparable to \( k\text{CFA} \), etc.
Forward and reverse analyses compared

- Forward analyses are more natural to derive from operational semantics
- DDPA is demand-driven and model-checking: potentially more efficient than forward analyses
- $k$ in $k$DDPA not directly comparable to $k$CFA, etc.
  - $k$ of $k$DDPA also needs to be bigger for handling non-locals
Forward and reverse analyses compared

- Forward analyses are more natural to derive from operational semantics.
- DDPA is demand-driven and model-checking: potentially more efficient than forward analyses.
- $k$ in $k$DDPA not directly comparable to $k$CFA, etc.
  - $k$ of $k$DDPA also needs to be bigger for handling non-locals.
  - Which makes sense: for fixed $k$, $k$CFA is EXPTIME but $k$DDPA is polynomial.
Forward and reverse analyses compared

- Forward analyses are more natural to derive from operational semantics
- DDPA is demand-driven and model-checking: potentially more efficient than forward analyses
- $k$ in $k$DDPA not directly comparable to $k$CFA, etc.
  - $k$ of $k$DDPA also needs to be bigger for handling non-locals
  - Which makes sense: for fixed $k$, $k$CFA is EXPTIME but $k$DDPA is polynomial
- Practically speaking, expressiveness appears similar
Implementation

- Paper artifact: inefficient proof-of-concept
Implementation

- Paper artifact: inefficient proof-of-concept
- Now: efficient implementation with additional features
Implementation

- Paper artifact: inefficient proof-of-concept
- Now: efficient implementation with additional features
  - Records
Implementation

- Paper artifact: inefficient proof-of-concept
- Now: efficient implementation with additional features
  - Records
  - Path-sensitivity – paths validated by PDS
Implementation

- Paper artifact: inefficient proof-of-concept
- Now: efficient implementation with additional features
  - Records
  - Path-sensitivity – paths validated by PDS
  - Heap-sensitive state including may/must alias information
Implementation

- Paper artifact: inefficient proof-of-concept
- Now: efficient implementation with additional features
  - Records
  - Path-sensitivity – paths validated by PDS
  - Heap-sensitive state including may/must alias information
- Lazily constructs PDS according to regular definition
Implementation

- Paper artifact: inefficient proof-of-concept
- Now: efficient implementation with additional features
  - Records
  - Path-sensitivity – paths validated by PDS
  - Heap-sensitive state including may/must alias information
- Lazily constructs PDS according to regular definition
- Looks to be reasonably efficient
Future Work

- Variable alignment for precision
Future Work

- Variable alignment for precision
- Better call stack model for performance
Future Work

- Variable alignment for precision
- Better call stack model for performance
- Application to existing languages
Conclusions

- DDPA: first flow-sensitive, demand-driven, higher-order program analysis
- Program analysis based on incremental model checking
  - Promising for efficiency
- Appears comparable in expressiveness with state-of-the-art forward analyses
- Code: https://github.com/JHU-PL-Lab/odefa