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

	push forward	demand-driven
first order	abstract interpretation data flow analysis	CFL-reachability reverse data flow analysis
higher order	kCFA CFA2 PDCFA ΓCFA	

Some Program Analyses

	push forward	demand-driven
first order	abstract interpretation data flow analysis	CFL-reachability reverse data flow analysis
higher order	kCFA CFA2 PDCFA ΓCFA	

Some Program Analyses

	push forward	demand-driven
first order	abstract interpretation data flow analysis	CFL-reachability reverse data flow analysis
higher order	kCFA, CFA2 PDCFA, ΓCFA	

Some Program Analyses

	push forward	demand-driven
first order	abstract interpretation data flow analysis	CFL-reachability reverse data flow analysis
higher order	kCFA CFA2 PDCFA ΓCFA	

Some Program Analyses

	push forward	demand-driven
first order	abstract interpretation data flow analysis	CFL-reachability reverse data flow analysis
higher order	kCFA CFA2 PDCFA ΓCFA	**DDPA**

Some Program Analyses

	push forward		demand-driven
first order	abstract interpretation data flow analysis		CFL-reachability reverse data flow analysis
higher order	kCFA PDCFA	CFA2 ΓCFA	**DDPA**

POLYFLOW$_{\text{CFL}}$

DDPA By Example

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

DDPA By Example

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

$\Downarrow$ A-normalize

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

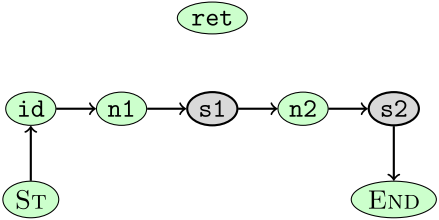
DDPA By Example

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

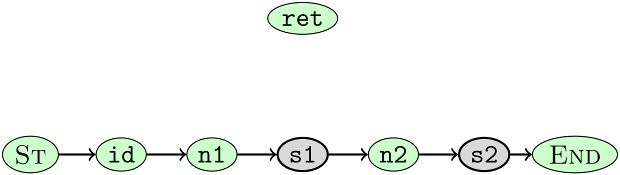
⇓ A-normalize

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

Initial graph before any calls wired:



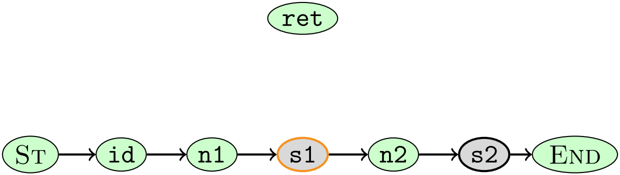
DDPA By Example




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

DDPA By Example

Analyze call site s1

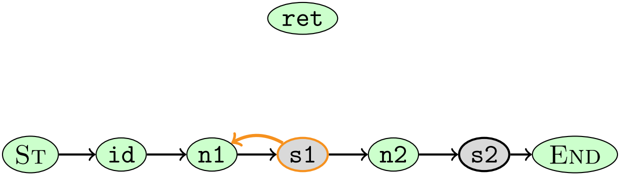



```
id = fun x -> ( ret = x; );
n1 = 1;
s1 = id n1;
n2 = 2;
s2 = id n2;
```

DDPA By Example

Analyze call site `s1`

Look backward to find function `id`

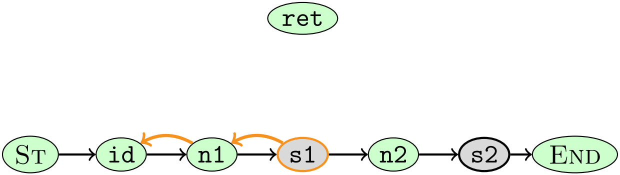



```
id = fun x -> ( ret = x; );
n1 = 1;
s1 = id n1;
n2 = 2;
s2 = id n2;
```

DDPA By Example

Analyze call site `s1`

Look backward to find function `id`

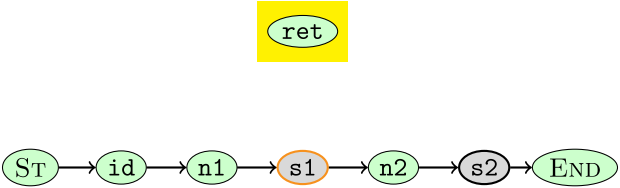



```
id = fun x -> ( ret = x; );
n1 = 1;
s1 = id n1;
n2 = 2;
s2 = id n2;
```

DDPA By Example

Analyze call site s1

Look backward to find function id

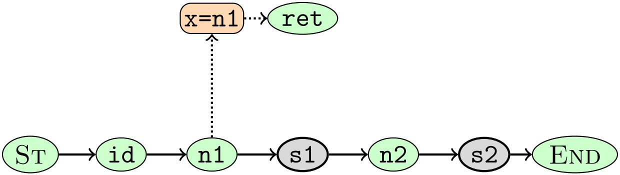



```
id = fun x -> ( ret = x; );
n1 = 1;
s1 = id n1;
n2 = 2;
s2 = id n2;
```

DDPA By Example

Analyze call site s1

Bind argument `n1` to parameter `x`

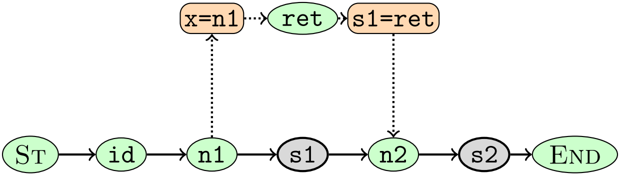



```
id = fun x -> ( ret = x; );
n1 = 1;
s1 = id n1;
n2 = 2;
s2 = id n2;
```

DDPA By Example

Analyze call site `s1`

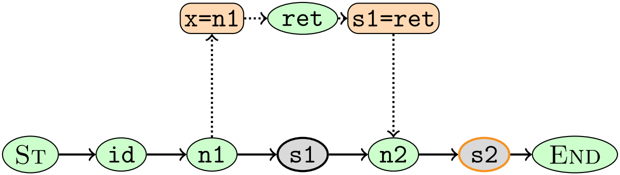
Assign result `ret` to call site `z1`




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

DDPA By Example

Analyze call site s2

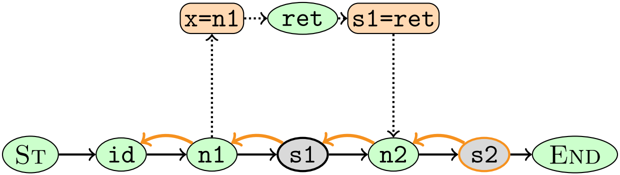



```
id = fun x -> ( ret = x; );
n1 = 1;
s1 = id n1;
n2 = 2;
s2 = id n2;
```

DDPA By Example

Analyze call site s2

Look backward to find function id

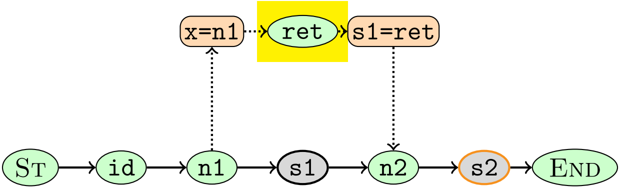



```
id = fun x -> ( ret = x; );
n1 = 1;
s1 = id n1;
n2 = 2;
s2 = id n2;
```

DDPA By Example

Analyze call site s2

Look backward to find function `id`

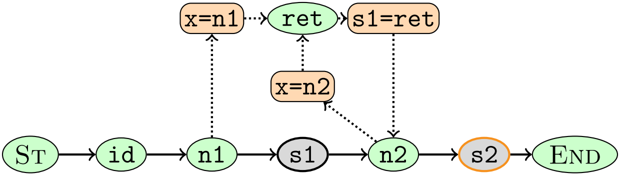



```
id = fun x -> ( ret = x; );
n1 = 1;
s1 = id n1;
n2 = 2;
s2 = id n2;
```

DDPA By Example

Analyze call site `s2`

Bind argument `n2` to parameter `x`

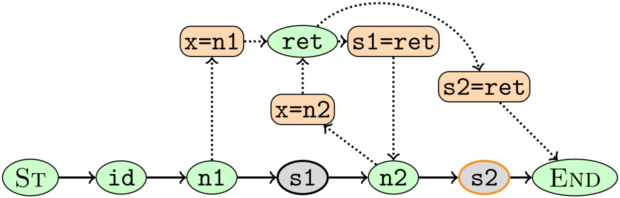



```
id = fun x -> ( ret = x; );
n1 = 1;
s1 = id n1;
n2 = 2;
s2 = id n2;
```

DDPA By Example

Analyze call site s2

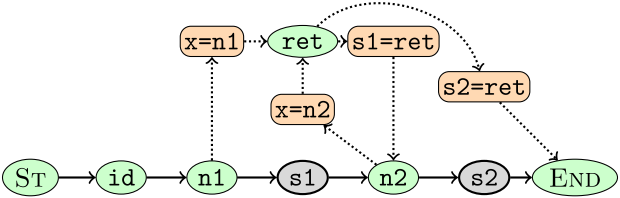
Assign result `ret` to call site `z2`




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

DDPA By Example

CFG construction complete




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

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?"

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 <u>temporally reversed</u> and <u>on demand</u>

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 <u>temporally reversed</u> and <u>on demand</u>
- 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 <u>temporally reversed</u> and <u>on demand</u>
- 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 <u>temporally reversed</u> and <u>on demand</u>
- How could this lookup be accurate? Challenges:
  - **Context-sensitivity / polymorphism?**
  - **Non-local variables?**
  - Recursion?
  - Function parameters?
  - State and heap aliases?
  - Path-sensitivity?

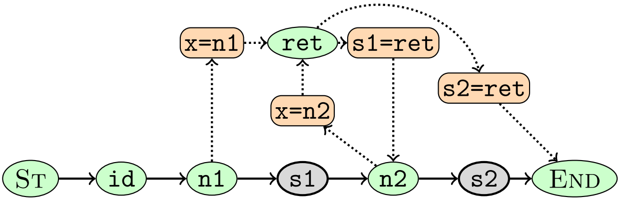
Observe

- 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 <u>temporally reversed</u> and <u>on demand</u>
- 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 <u>temporally reversed</u> and <u>on demand</u>
- 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)

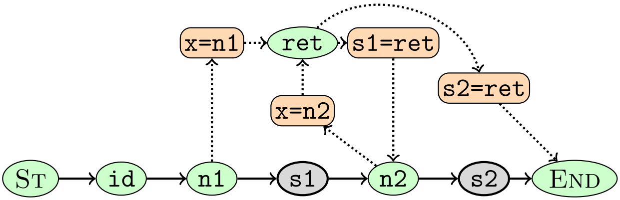
Call Stack Alignment for Context-Sensitivity




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

Call Stack Alignment for Context-Sensitivity

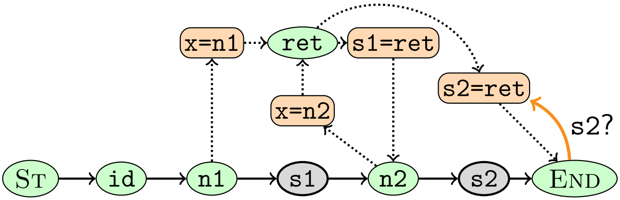
Look up s2 from end of program




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

Call Stack Alignment for Context-Sensitivity

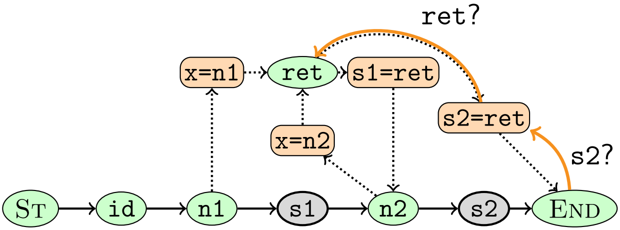
Look up s2 from end of program




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

Call Stack Alignment for Context-Sensitivity

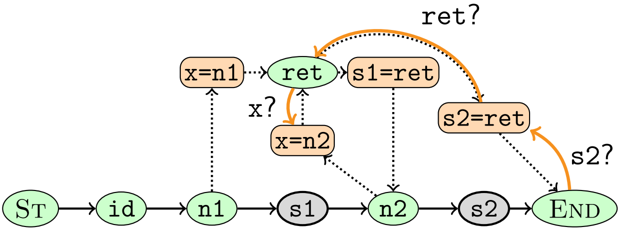
Look up s2 from end of program




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

Call Stack Alignment for Context-Sensitivity

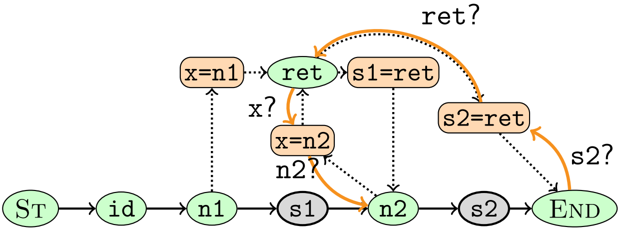
Look up s2 from end of program




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

Call Stack Alignment for Context-Sensitivity

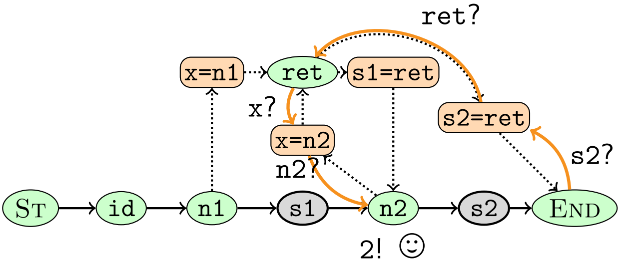
Look up s2 from end of program




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

Call Stack Alignment for Context-Sensitivity

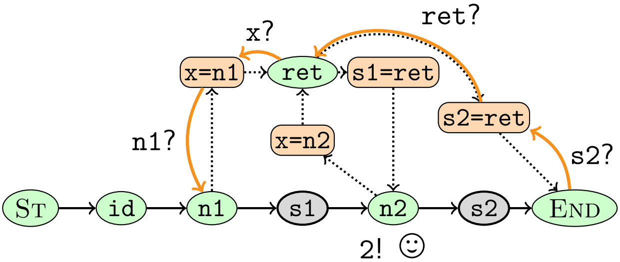
Look up s2 from end of program




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

Call Stack Alignment for Context-Sensitivity

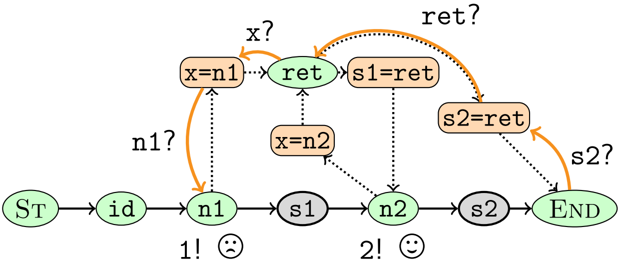
Look up s2 from end of program




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

Call Stack Alignment for Context-Sensitivity

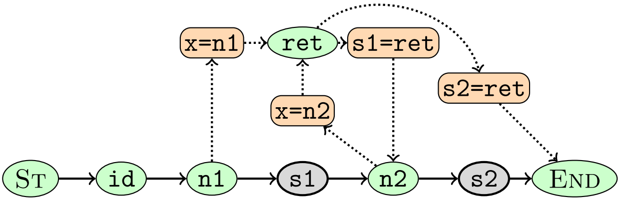
Look up s2 from end of program




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

Call Stack Alignment for Context-Sensitivity

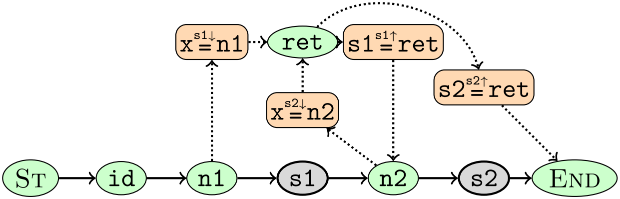
Solution: Maintain call stack during lookup, use to filter




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

Call Stack Alignment for Context-Sensitivity

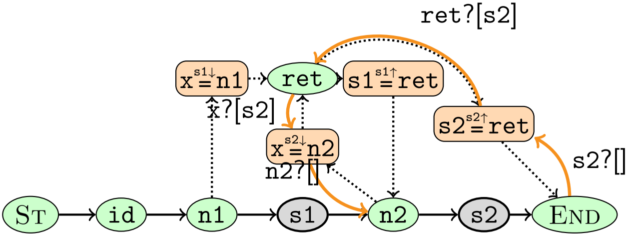
Solution: Maintain call stack during lookup, use to filter




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

Call Stack Alignment for Context-Sensitivity

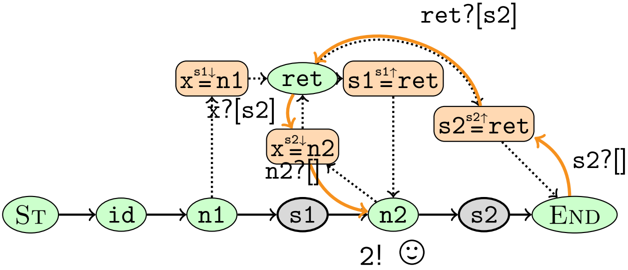
Solution: Maintain call stack during lookup, use to filter




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

Call Stack Alignment for Context-Sensitivity

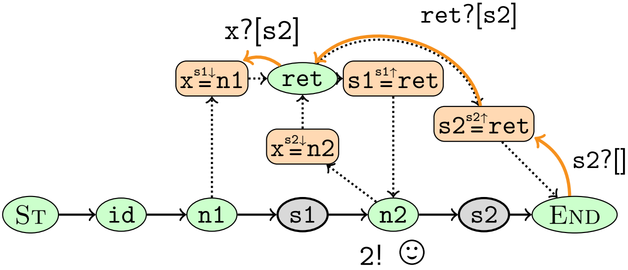
Solution: Maintain call stack during lookup, use to filter




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

Call Stack Alignment for Context-Sensitivity

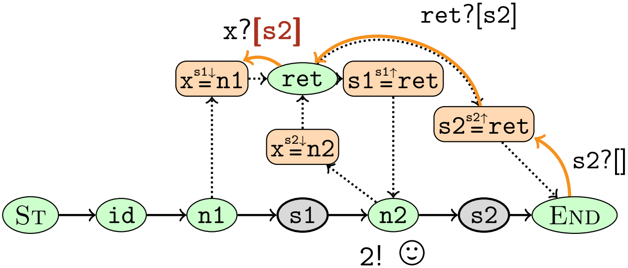
Solution: Maintain call stack during lookup, use to filter




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

Call Stack Alignment for Context-Sensitivity

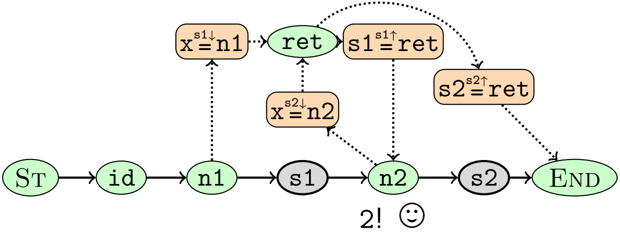
Solution: Maintain call stack during lookup, use to filter




```
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




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

Implementing stack alignment

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
- $\text{POLYFLOW}_{\text{CFL}}$
  - 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
- $\text{POLYFLOW}_{\text{CFL}}$
  - 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
- $\text{POLYFLOW}_{\text{CFL}}$
  - 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$_{\text{CFL}}$
  - 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

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

Handling Non-Local Variables

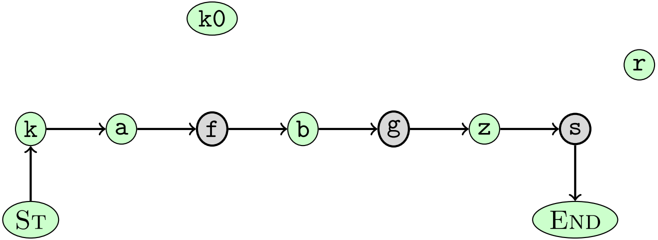
Non-local example: K-combinator

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

$\Downarrow$ A-normalization

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

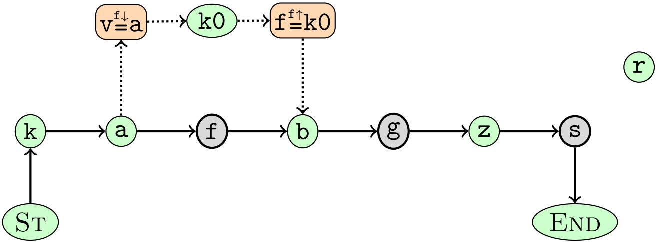
Non-Local Variable Lookup




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

Non-Local Variable Lookup

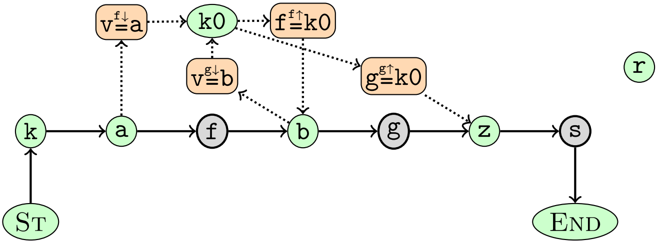
Analyze call site `f`.




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

Non-Local Variable Lookup

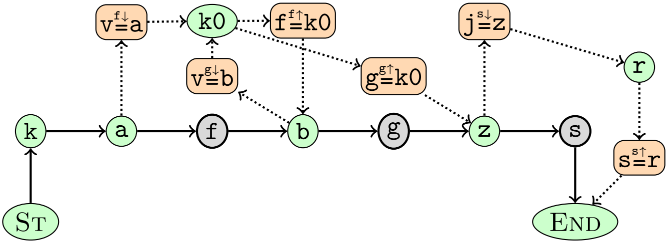
Analyze call site g.




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

Non-Local Variable Lookup

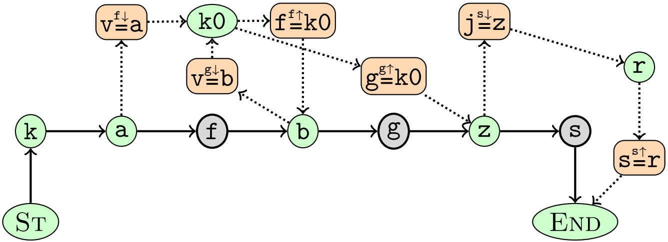
Analyze call site `s`.




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

Non-Local Variable Lookup

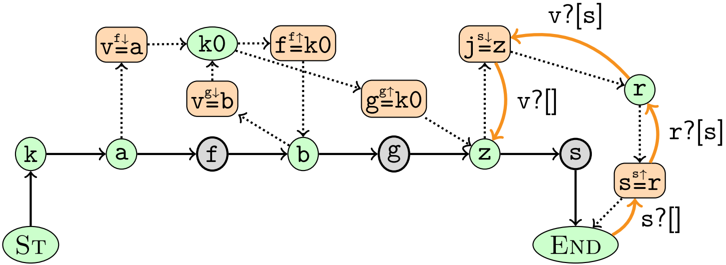
Look up s from end of program.




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

Non-Local Variable Lookup

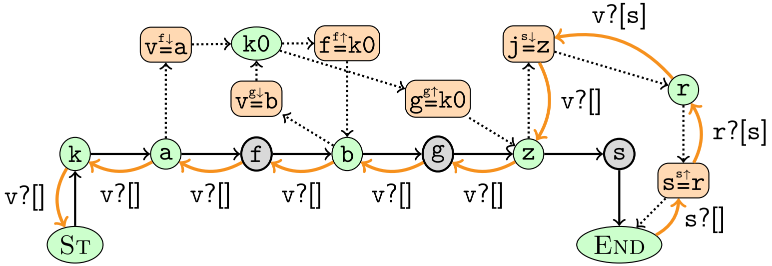
Look up s from end of program.




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

Non-Local Variable Lookup

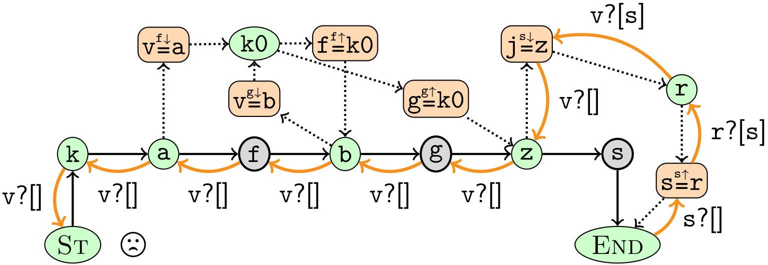
Look up s from end of program.




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

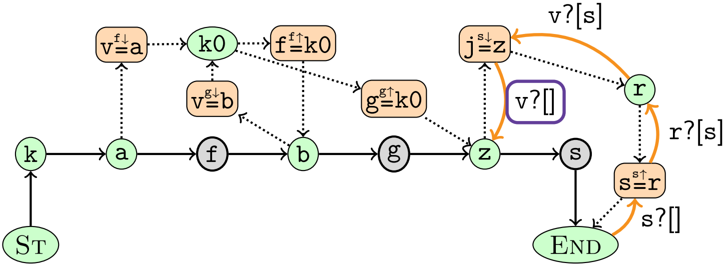
Non-Local Variable Lookup

Look up s from end of program.



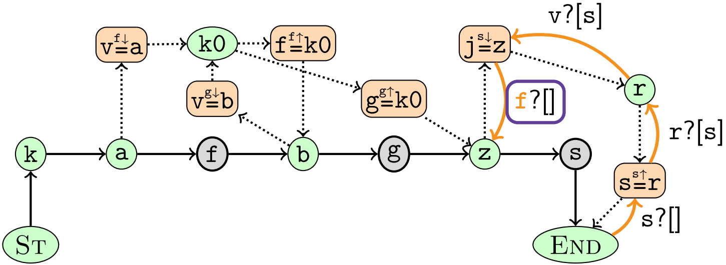

```
k = fun v -> (k0 = fun j -> (r = v;););
a = 1;  f = k a;
b = 2;  g = k b;
z = 0;  s = f z;
```

Non-Local Variable Lookup



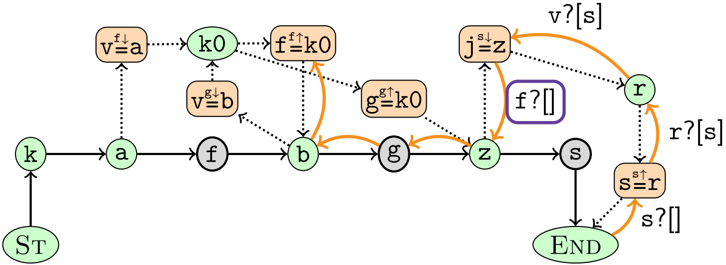

```
k = fun v -> (k0 = fun j -> (r = v;););
a = 1;  f = k a;
b = 2;  g = k b;
z = 0;  s = f z;
```

Non-Local Variable Lookup



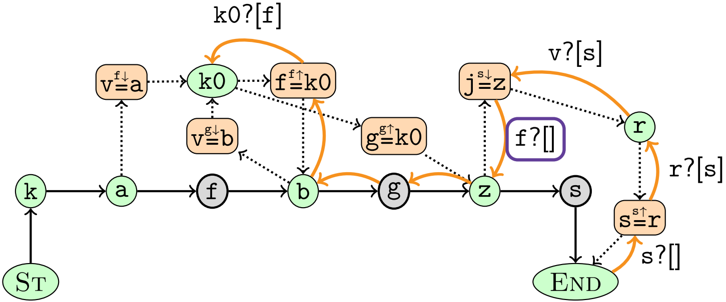

```
k = fun v -> (k0 = fun j -> (r = v;););
a = 1;  f = k a;
b = 2;  g = k b;
z = 0;  s = f z;
```

Non-Local Variable Lookup



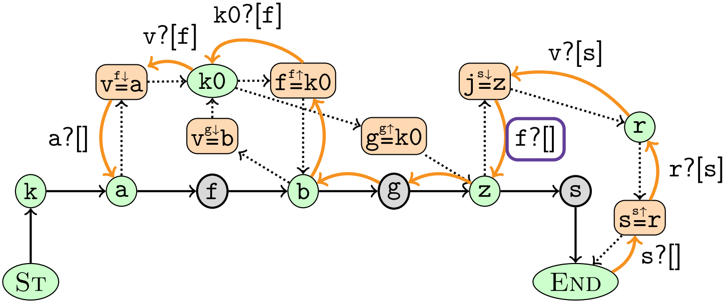

```
k = fun v -> (k0 = fun j -> (r = v;););
a = 1;  f = k a;
b = 2;  g = k b;
z = 0;  s = f z;
```

Non-Local Variable Lookup



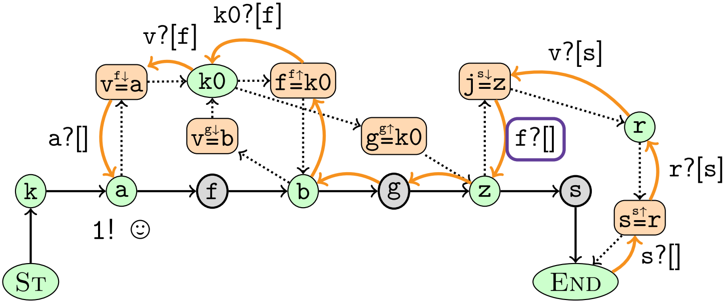

```
k = fun v -> (k0 = fun j -> (r = v;););
a = 1;  f = k a;
b = 2;  g = k b;
z = 0;  s = f z;
```

Non-Local Variable Lookup




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

Non-Local Variable Lookup




```
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

- 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.
  - kDDPA: 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: kDDPA (for fixed k) has polynomial time bound

Properties

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

Properties

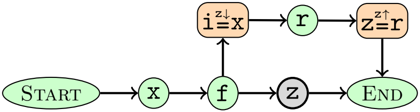
- Recursion: handled by PDS lookup; cycles are fine in a PDS
- Theorem: kDDPA (for fixed k) has polynomial time bound
- Lemma: both CFG and PDS are monotonically increasing over analysis
  - Allows for analysis to be purely additive – efficient sharing

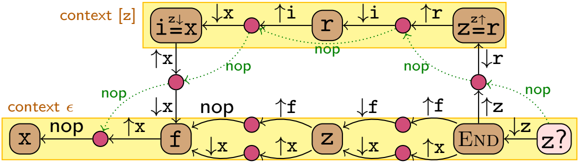
Properties

- Recursion: handled by PDS lookup; cycles are fine in a PDS
- Theorem: kDDPA (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!

Source / CFG / PDS - the whole analysis

```
x = {};
f = fun i -> ( r = i );
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 kDDPA not directly comparable to kCFA, 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 kDDPA not directly comparable to kCFA, etc.
  - k of kDDPA 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 kDDPA not directly comparable to kCFA, etc.
  - k of kDDPA also needs to be bigger for handling non-locals
  - Which makes sense: for fixed k, kCFA is EXPTIME but kDDPA 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 kDDPA not directly comparable to kCFA, etc.
  - k of kDDPA also needs to be bigger for handling non-locals
  - Which makes sense: for fixed k, kCFA is EXPTIME but kDDPA 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 reasonabily 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`