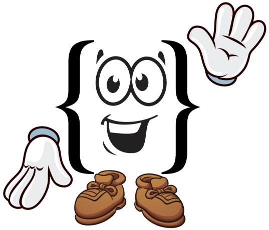
Demand-Driven Relative Store Fragments for Singleton Abstraction

Little Store's Big Journey



Leandro Facchinetti[1] Zachary Palmer[2] Scott F. Smith[1]

The Johns Hopkins University[1]

Swarthmore College[2]

September 1st, 2017

Some Program Analyses

	push forward	reverse lookup
first order	classic abs. interp. data flow analysis	CFL-reachability reverse data flow analysis
higher order	kCFA CFA2 PDCFA ΓCFA	

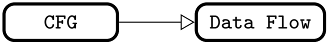
Some Program Analyses

	push forward	reverse lookup
first order	classic abs. interp. data flow analysis	CFL-reachability reverse data flow analysis
higher order	kCFA CFA2 PDCFA ΓCFA	

Some Program Analyses

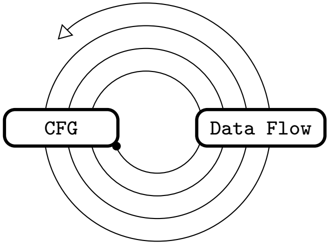
First Order

`for (int i=0;i<n;i++)`



Higher Order

`fold (λa e -> ...)`



Some Program Analyses

	push forward	reverse lookup
first order	classic abs. interp. data flow analysis	CFL-reachability reverse data flow analysis
higher order	kCFA CFA2 PDCFA ΓCFA	

Some Program Analyses

	push forward	reverse lookup
first order	classic abs. interp. data flow analysis	CFL-reachability reverse data flow analysis
higher order	kCFA CFA2 PDCFA ΓCFA	

Some Program Analyses

	push forward	reverse lookup
first order	classic abs. interp. data flow analysis	CFL-reachability reverse data flow analysis
higher order	kCFA CFA2 PDCFA ΓCFA	DDPA

Some Program Analyses

	push forward	reverse lookup
first order	classic abs. interp. data flow analysis	CFL-reachability reverse data flow analysis
higher order	kCFA CFA2 PDCFA ΓCFA	DDPA **DRSF**

Some Program Analyses

	push forward	reverse lookup
first order	classic abs. interp. data flow analysis	CFL-reachability reverse data flow analysis
higher order	kCFA CFA2 PDCFA ΓCFA	DDPA **DRSF** POLYFLOW$_{\mathrm{CFL}}$ (weak non-locals)

Demand-Driven Higher-Order Program Analyses

	DDPA	**DRSF**
Context-sensitive		
Flow-sensitive		
Path-sensitive		
Must-alias		
Non-local variables		

Demand-Driven Higher-Order Program Analyses

	DDPA	DRSF
Context-sensitive	✓	
Flow-sensitive	✓	
Path-sensitive	∼	
Must-alias	∼	
Non-local variables	✓	

Demand-Driven Higher-Order Program Analyses

	DDPA	DRSF
Context-sensitive	✓	✓
Flow-sensitive	✓	✓
Path-sensitive	∼	✓
Must-alias	∼	✓
Non-local variables	✓	✓

Demand-Driven Higher-Order Program Analyses

	DDPA	DRSF
Context-sensitive	✓ Contours	✓
Flow-sensitive	✓	✓
Path-sensitive	∼	✓
Must-alias	∼	✓
Non-local variables	✓	✓

Demand-Driven Higher-Order Program Analyses

	DDPA	**DRSF**
Context-sensitive	✓ Contours	✓
Flow-sensitive	✓ Natural	✓
Path-sensitive	∼	✓
Must-alias	∼	✓
Non-local variables	✓	✓

Demand-Driven Higher-Order Program Analyses

	DDPA		DRSF
Context-sensitive	✓	Contours	✓
Flow-sensitive	✓	Natural	✓
Path-sensitive	∼	Filters	✓
Must-alias	∼		✓
Non-local variables	✓		✓

Demand-Driven Higher-Order Program Analyses

	DDPA	DRSF
Context-sensitive	✓ Contours	✓
Flow-sensitive	✓ Natural	✓
Path-sensitive	∼ Filters	✓
Must-alias	∼ A Mess	✓
Non-local variables	✓	✓

Demand-Driven Higher-Order Program Analyses

	DDPA	**DRSF**
Context-sensitive	✓ Contours	✓
Flow-sensitive	✓ Natural	✓
Path-sensitive	∼ Filters	✓
Must-alias	∼ A Mess	✓
Non-local variables	✓ Lookup	✓

Demand-Driven Higher-Order Program Analyses

	DDPA	DRSF
Context-sensitive	✓ Contours	✓ Little Stores
Flow-sensitive	✓ Natural	✓ Little Stores
Path-sensitive	∼ Filters	✓ Little Stores
Must-alias	∼ A Mess	✓ Little Stores
Non-local variables	✓ Lookup	✓ Lookup

DDPA by Example

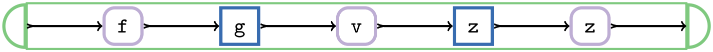
DDPA by Example

```
let f = fun p ->
    let x = p in
    fun y -> x + y
in
let g = f 4 in
let v = 1 in
let z = g v in z
```

DDPA by Example

Initial CFG

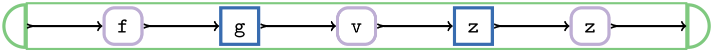
```
let f = fun p ->
    let x = p in
    fun y -> x + y
in
let g = f 4 in
let v = 1 in
let z = g v in z
```



DDPA by Example

Expand function call f 4

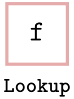
```
let f = fun p ->
    let x = p in
    fun y -> x + y
in
let g = f 4 in
let v = 1 in
let z = g v in z
```

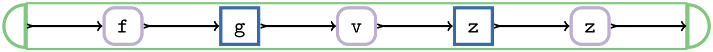


DDPA by Example

Expand function call `f 4`

```
let f = fun p ->
    let x = p in
    fun y -> x + y
in
let g = f 4 in
let v = 1 in
let z = g v in z
```

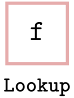


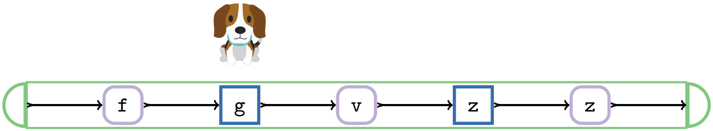


DDPA by Example

Expand function call `f 4`

```
let f = fun p ->
    let x = p in
    fun y -> x + y
in
let g = f 4 in
let v = 1 in
let z = g v in z
```





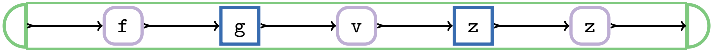
DDPA by Example

Expand function call `f 4`

```
let f = fun p ->
    let x = p in
    fun y -> x + y
in
let g = f 4 in
let v = 1 in
let z = g v in z
```

Lookup

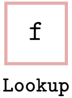
"Look Pup"

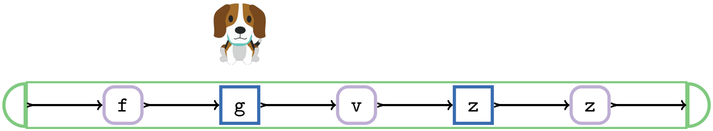


DDPA by Example

Expand function call `f 4`

```
let f = fun p ->
    let x = p in
    fun y -> x + y
in
let g = f 4 in
let v = 1 in
let z = g v in z
```

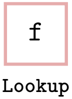


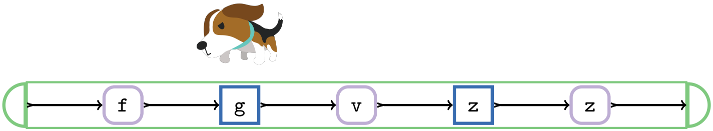


DDPA by Example

Expand function call `f 4`

```
let f = fun p ->
    let x = p in
    fun y -> x + y
in
let g = f 4 in
let v = 1 in
let z = g v in z
```

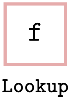


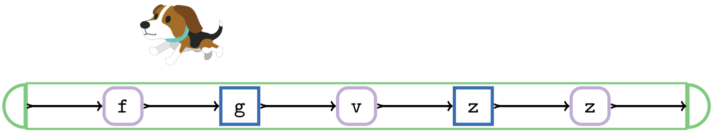


DDPA by Example

Expand function call `f 4`

```
let f = fun p ->
    let x = p in
    fun y -> x + y
in
let g = f 4 in
let v = 1 in
let z = g v in z
```

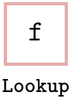


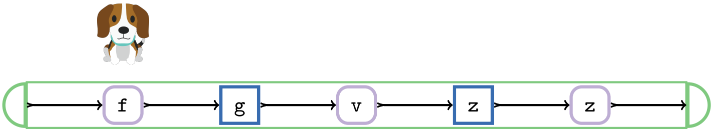


DDPA by Example

Expand function call `f 4`

```
let f = fun p ->
    let x = p in
    fun y -> x + y
in
let g = f 4 in
let v = 1 in
let z = g v in z
```

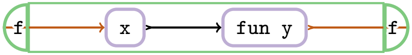


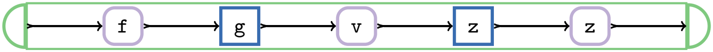


DDPA by Example

Wire in function call `f 4`

```
let f = fun p ->
    let x = p in
    fun y -> x + y
in
let g = f 4 in
let v = 1 in
let z = g v in z
```

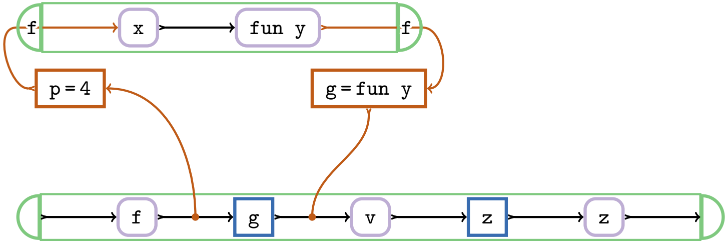




DDPA by Example

Wire in function call `f 4`

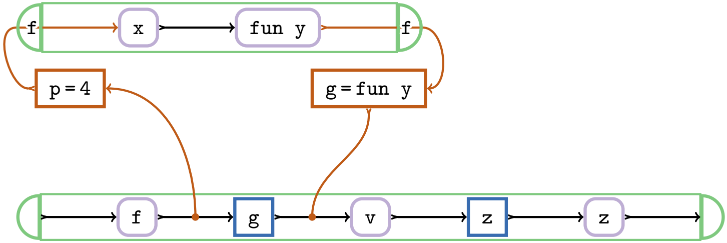
```
let f = fun p ->
    let x = p in
    fun y -> x + y
in
let g = f 4 in
let v = 1 in
let z = g v in z
```



DDPA by Example

Expand function call g v

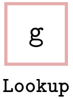
```
let f = fun p ->
    let x = p in
    fun y -> x + y
in
let g = f 4 in
let v = 1 in
let z = g v in z
```

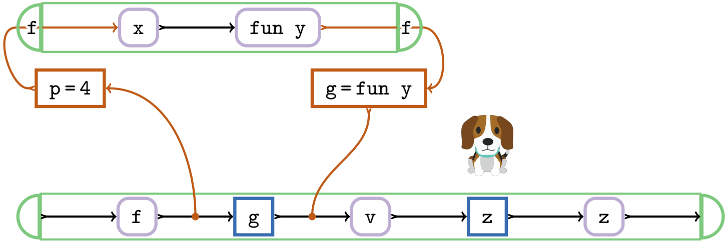


DDPA by Example

Expand function call g v

```
let f = fun p ->
    let x = p in
    fun y -> x + y
in
let g = f 4 in
let v = 1 in
let z = g v in z
```

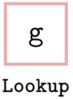


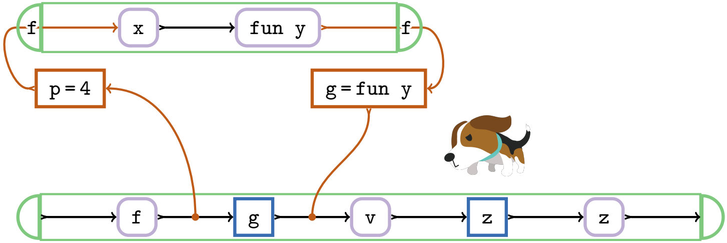


DDPA by Example

Expand function call g v

```
let f = fun p ->
    let x = p in
    fun y -> x + y
in
let g = f 4 in
let v = 1 in
let z = g v in z
```

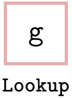


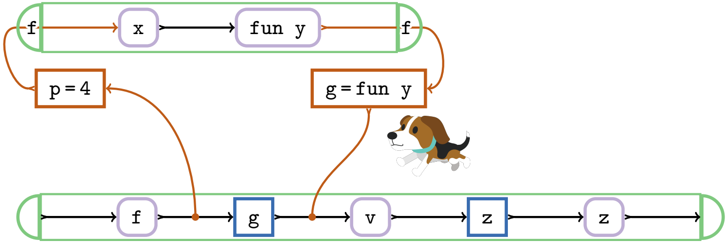


DDPA by Example

Expand function call `g v`

```
let f = fun p ->
    let x = p in
    fun y -> x + y
in
let g = f 4 in
let v = 1 in
let z = g v in z
```

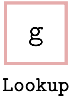


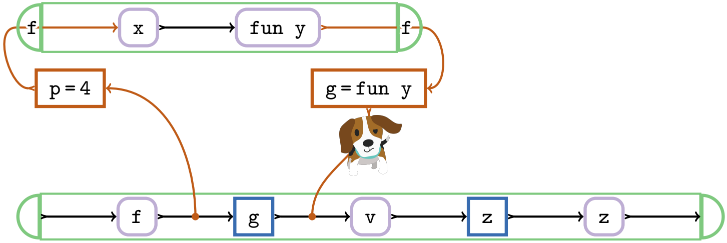


DDPA by Example

Expand function call `g v`

```
let f = fun p ->
    let x = p in
    fun y -> x + y
in
let g = f 4 in
let v = 1 in
let z = g v in z
```

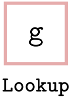


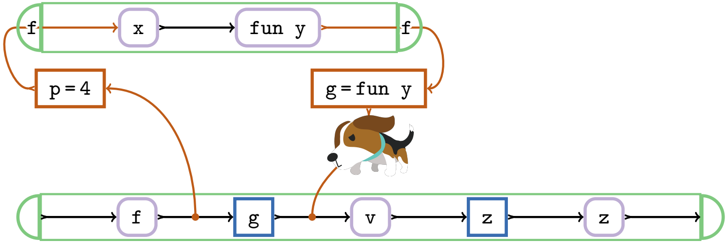


DDPA by Example

Expand function call `g v`

```
let f = fun p ->
    let x = p in
    fun y -> x + y
in
let g = f 4 in
let v = 1 in
let z = g v in z
```

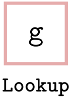


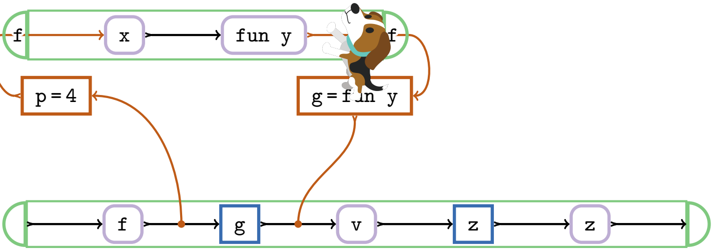


DDPA by Example

Expand function call g v

```
let f = fun p ->
    let x = p in
    fun y -> x + y
in
let g = f 4 in
let v = 1 in
let z = g v in z
```

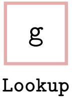


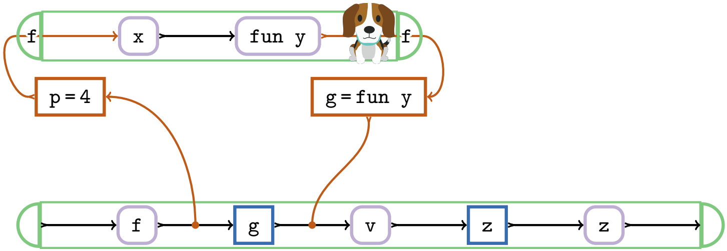


DDPA by Example

Expand function call g v

```
let f = fun p ->
    let x = p in
    fun y -> x + y
in
let g = f 4 in
let v = 1 in
let z = g v in z
```

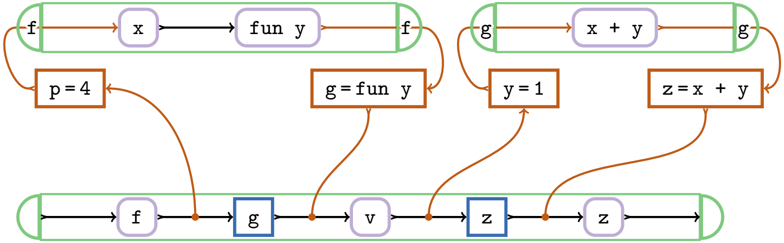




DDPA by Example

Wire in function call g v

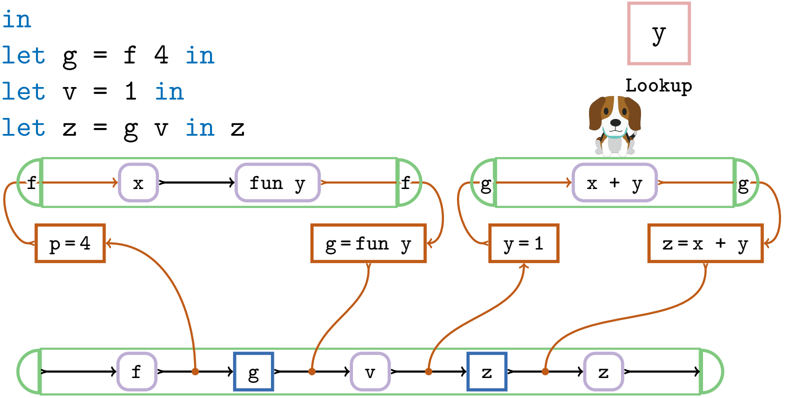
```
let f = fun p ->
    let x = p in
    fun y -> x + y
in
let g = f 4 in
let v = 1 in
let z = g v in z
```



DDPA by Example

Parameter lookup: y

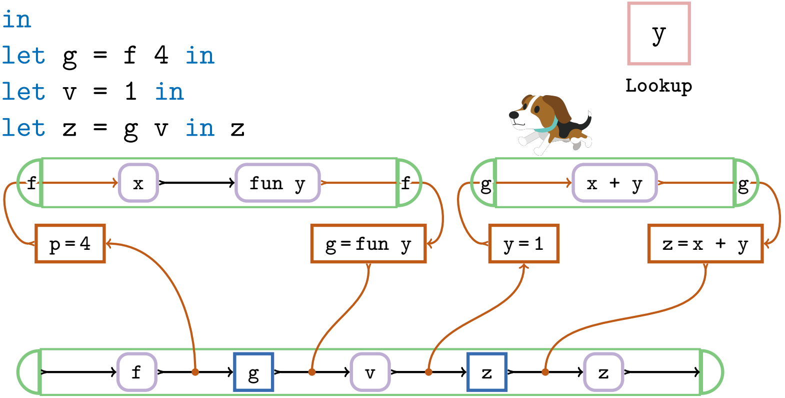
```
let f = fun p ->
    let x = p in
    fun y -> x + y
in
let g = f 4 in
let v = 1 in
let z = g v in z
```



DDPA by Example

Parameter lookup: y

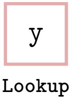
```
let f = fun p ->
    let x = p in
    fun y -> x + y
in
let g = f 4 in
let v = 1 in
let z = g v in z
```

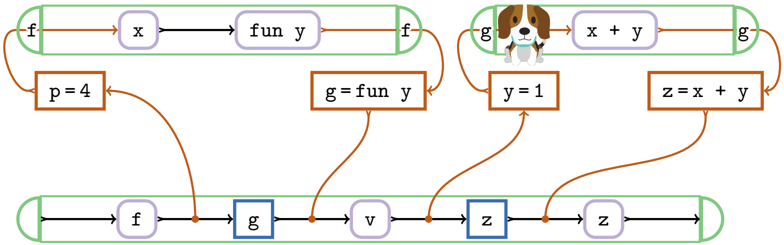


DDPA by Example

Parameter lookup: y

```
let f = fun p ->
    let x = p in
    fun y -> x + y
in
let g = f 4 in
let v = 1 in
let z = g v in z
```

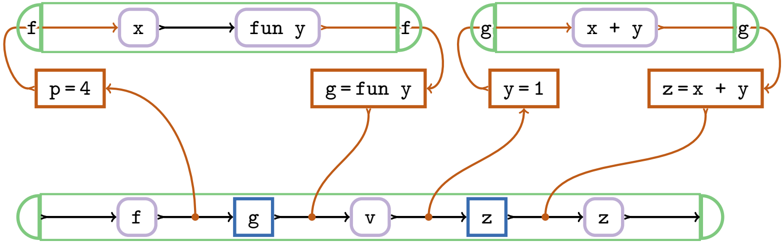




DDPA by Example

Non-local lookup: x

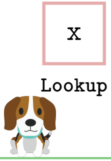
```
let f = fun p ->
    let x = p in
    fun y -> x + y
in
let g = f 4 in
let v = 1 in
let z = g v in z
```

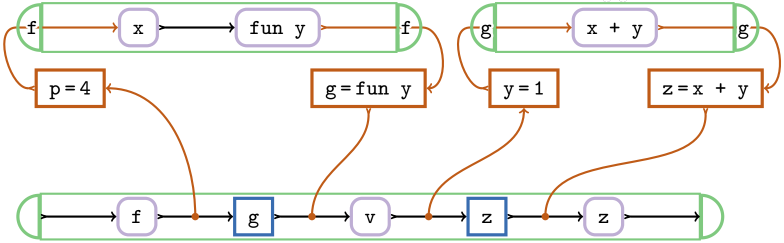


DDPA by Example

Non-local lookup: x

```
let f = fun p ->
    let x = p in
    fun y -> x + y
in
let g = f 4 in
let v = 1 in
let z = g v in z
```

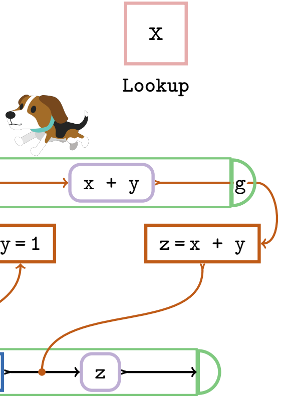


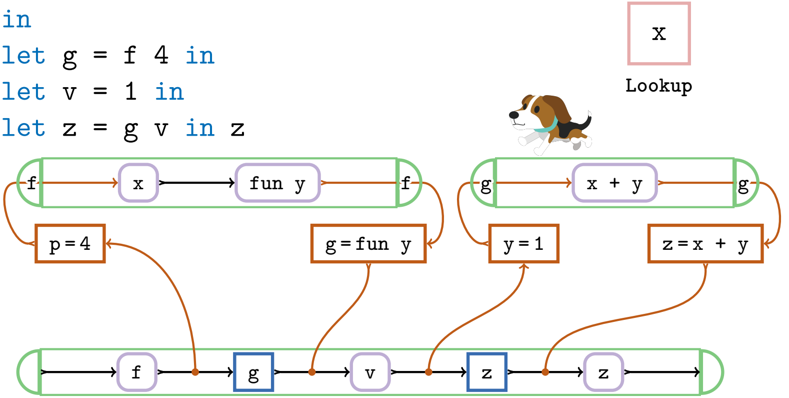


DDPA by Example

Non-local lookup: x

```
let f = fun p ->
    let x = p in
    fun y -> x + y
in
let g = f 4 in
let v = 1 in
let z = g v in z
```

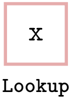


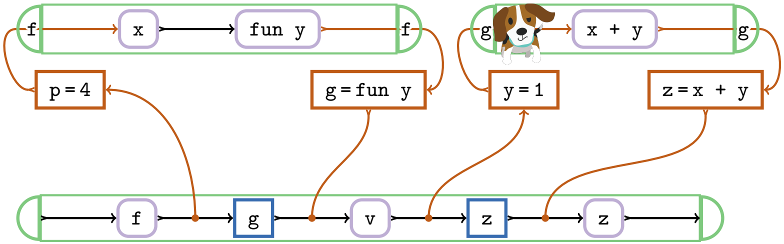


DDPA by Example

Non-local lookup: x

```
let f = fun p ->
    let x = p in
    fun y -> x + y
in
let g = f 4 in
let v = 1 in
let z = g v in z
```

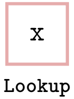


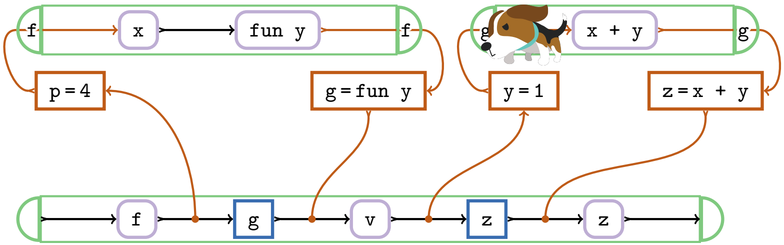


DDPA by Example

Non-local lookup: `x`

```
let f = fun p ->
    let x = p in
    fun y -> x + y
in
let g = f 4 in
let v = 1 in
let z = g v in z
```





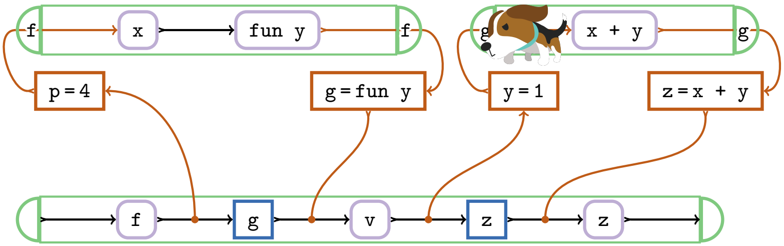
DDPA by Example

Non-local lookup: x

```
let f = fun p ->
    let x = p in
    fun y -> x + y
in
let g = f 4 in
let v = 1 in
let z = g v in z
```

g
x

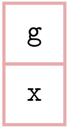
Lookup Stack



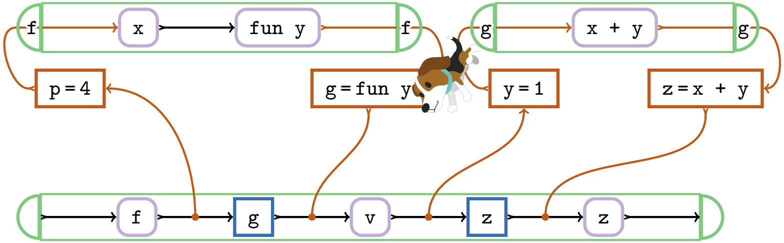
DDPA by Example

Non-local lookup: x

```
let f = fun p ->
    let x = p in
    fun y -> x + y
in
let g = f 4 in
let v = 1 in
let z = g v in z
```



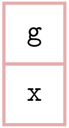
Lookup Stack



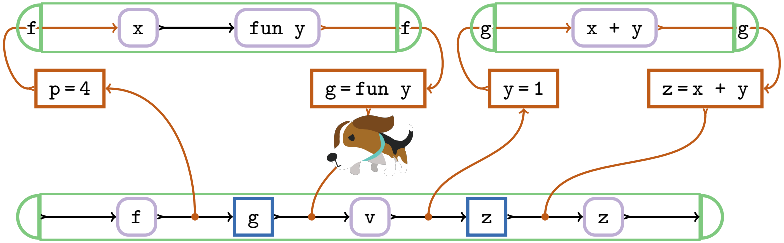
DDPA by Example

Non-local lookup: x

```
let f = fun p ->
    let x = p in
    fun y -> x + y
in
let g = f 4 in
let v = 1 in
let z = g v in z
```



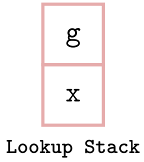
Lookup Stack

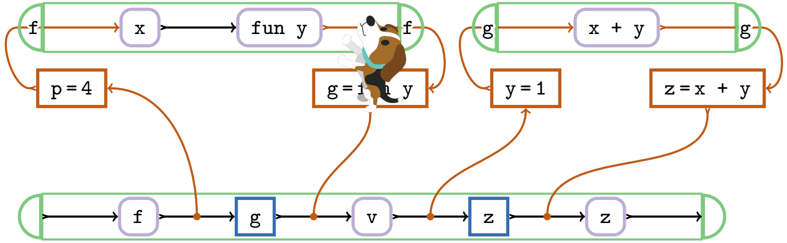


DDPA by Example

Non-local lookup: x

```
let f = fun p ->
    let x = p in
    fun y -> x + y
in
let g = f 4 in
let v = 1 in
let z = g v in z
```

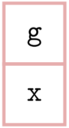




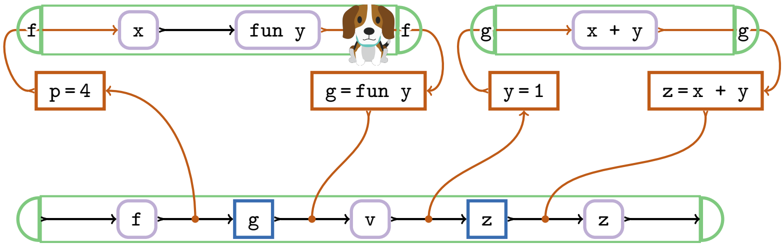
DDPA by Example

Non-local lookup: x

```
let f = fun p ->
    let x = p in
    fun y -> x + y
in
let g = f 4 in
let v = 1 in
let z = g v in z
```



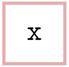
Lookup Stack



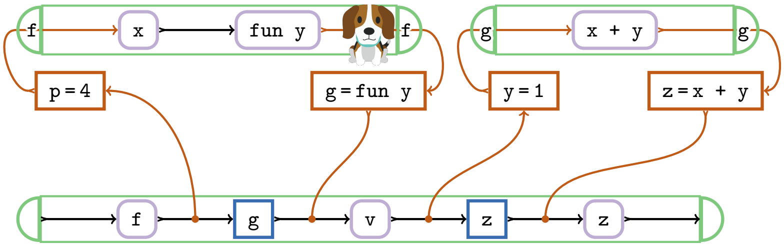
DDPA by Example

Non-local lookup: x

```
let f = fun p ->
    let x = p in
    fun y -> x + y
in
let g = f 4 in
let v = 1 in
let z = g v in z
```



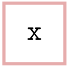
Lookup Stack



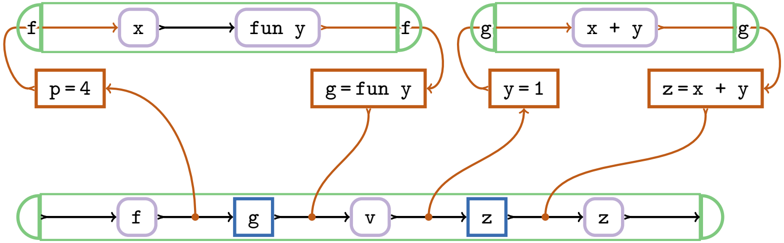
DDPA by Example

Non-local lookup: x

```
let f = fun p ->
    let x = p in
    fun y -> x + y
in
let g = f 4 in
let v = 1 in
let z = g v in z
```



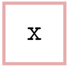
Lookup Stack



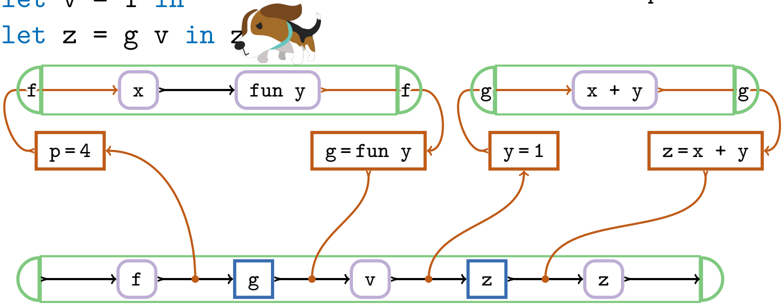
DDPA by Example

Non-local lookup: x

```
let f = fun p ->
    let x = p in
    fun y -> x + y
in
let g = f 4 in
let v = 1 in
let z = g v in z
```



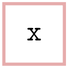
Lookup Stack



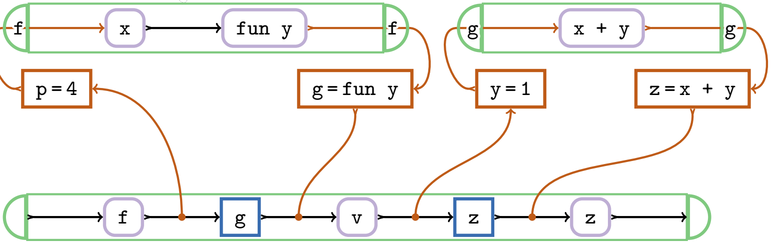
DDPA by Example

Non-local lookup: x

```
let f = fun p ->
    let x = p in
    fun y -> x + y
in
let g = f 4 in
let v = 1 in
let z = g v in z
```



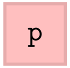
Lookup Stack



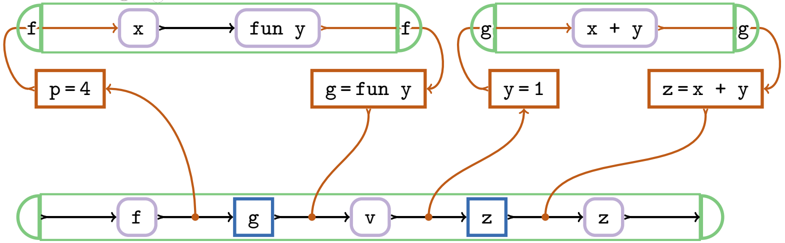
DDPA by Example

Non-local lookup: x

```
let f = fun p ->
    let x = p in
    fun y -> x + y
in
let g = f 4 in
let v = 1 in
let z =       in z
```



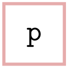
Lookup Stack



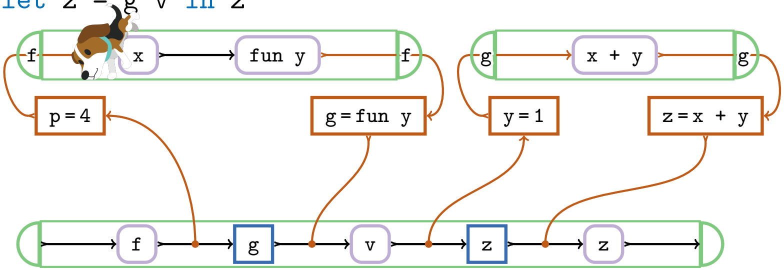
DDPA by Example

Non-local lookup: x

```
let f = fun p ->
    let x = p in
    fun y -> x + y
in
let g = f 4 in
let v = 1 in
let z = g v in z
```



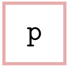
Lookup Stack

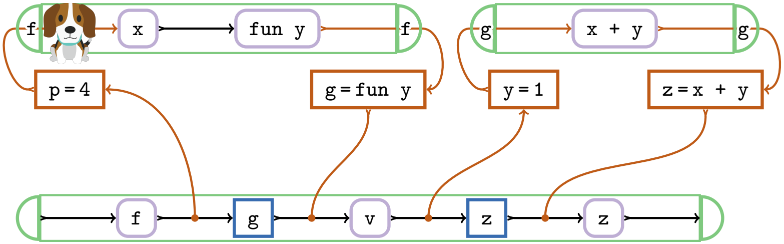


DDPA by Example

Non-local lookup: x

```
let f = fun p ->
    let x = p in
    fun y -> x + y
in
let g = f 4 in
let v = 1 in
let z = g v in z
```





DDPA

- Value lookup on demand: no explicit store!

DDPA

- Value lookup on demand: no explicit store!
- Lookup stack: intermediate lookups

DDPA

- Value lookup on demand: no explicit store!
- Lookup stack: intermediate lookups
  - Function calls
  - Record projections
  - Binary operators
  - ...

DDPA

- Value lookup on demand: no explicit store!
- Lookup stack: intermediate lookups
    - Function calls
    - Record projections
    - Binary operators
    - ...
- Polymorphism via abstract call stack

DDPA

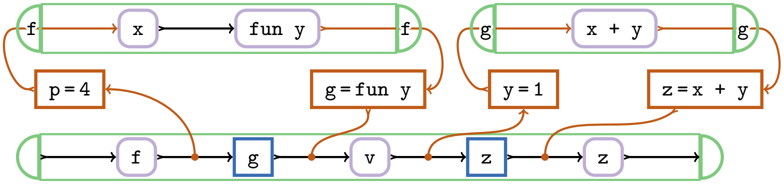
- Value lookup on demand: no explicit store!
- Lookup stack: intermediate lookups
    - Function calls
    - Record projections
    - Binary operators
    - ...
- Polymorphism via abstract call stack
- Recursion via pushdown reachability

DDPA

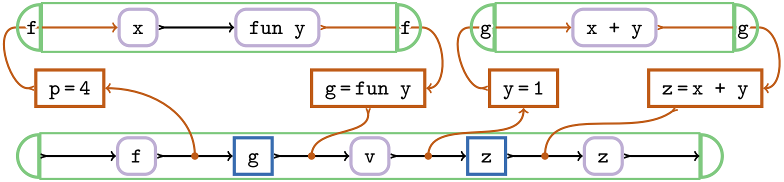
- Value lookup on demand: no explicit store!
- Lookup stack: intermediate lookups
    - Function calls
    - Record projections
    - Binary operators
    - ...
- Polymorphism via abstract call stack
- Recursion via pushdown reachability

Connection to forward analyses?

DDPA and Abstract Stores



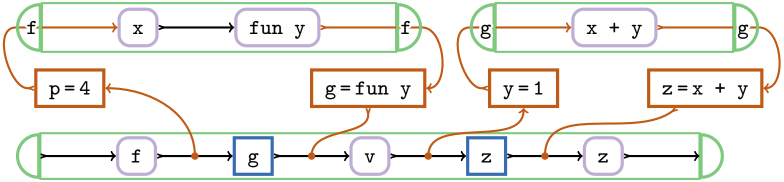
DDPA and Abstract Stores



$\{\}$

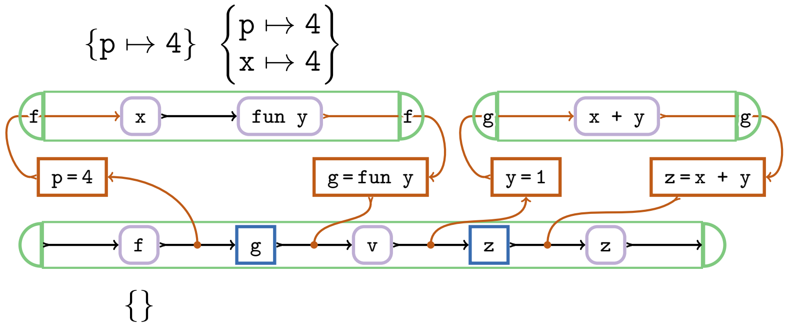
DDPA and Abstract Stores

$\{\texttt{p} \mapsto 4\}$

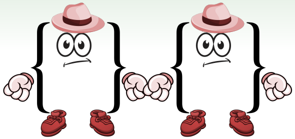


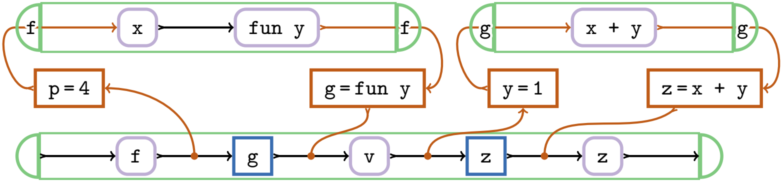
$\{\}$

DDPA and Abstract Stores



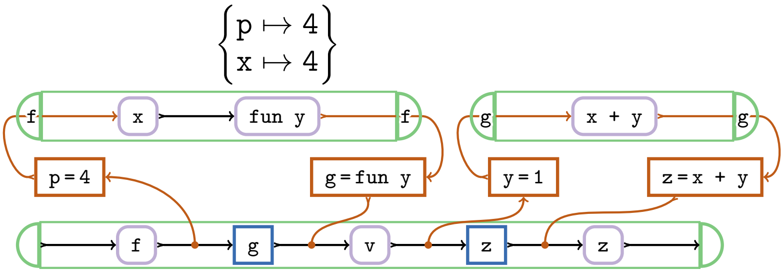
$\{p \mapsto 4\}$
$\left\{\begin{matrix} p \mapsto 4 \\ x \mapsto 4 \end{matrix}\right\}$
f
x
fun y
f
g
x + y
g
p = 4
g = fun y
y = 1
z = x + y
f
g
v
z
z
$\{\}$





- "Big stores": complete sets of bindings

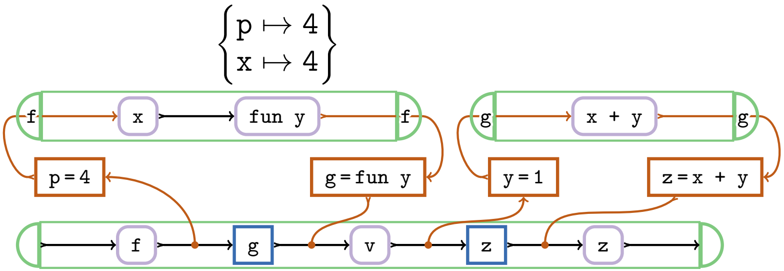
DDPA and Abstract Stores

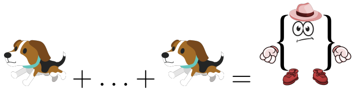


$\left\{\begin{matrix} p \mapsto 4 \\ x \mapsto 4 \end{matrix}\right\}$
f
x
fun y
f
g
x + y
g
p = 4
g = fun y
y = 1
z = x + y
f
g
v
z
z

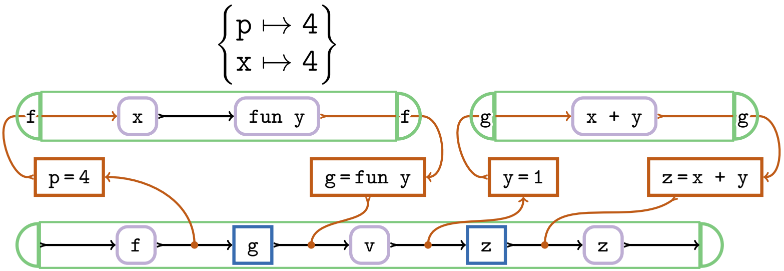
- "Big stores": complete sets of bindings
- DDPA: reconstruct big stores with lookups

DDPA and Abstract Stores

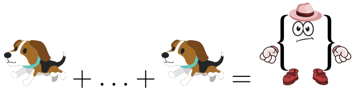


- "Big stores": complete sets of bindings
- DDPA: reconstruct big stores with lookups
- 

DDPA and Abstract Stores

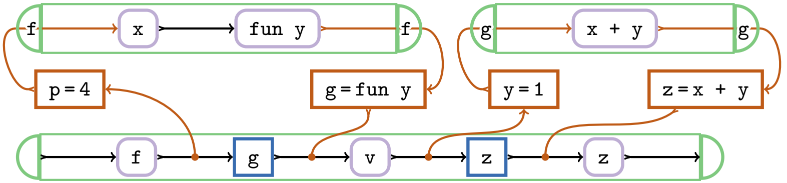


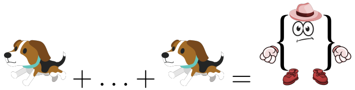
$\left\{\begin{matrix} p \mapsto 4 \\ x \mapsto 4 \end{matrix}\right\}$
f
x
fun y
f
g
x + y
g
p = 4
g = fun y
y = 1
z = x + y
f
g
v
z
z

- "Big stores": complete sets of bindings
- DDPA: reconstruct big stores with lookups
  - 
- Lookups from a point are independent

DDPA and Abstract Stores

$$\left\{\begin{matrix} p \mapsto 4 \\ x \mapsto 4 \end{matrix}\right\}$$



- “Big stores”: complete sets of bindings
- DDPA: reconstruct big stores with lookups
  - 

- Lookups from a point are independent
- Similar to per-point store widening

DDPA and Variable (Mis-)Alignment

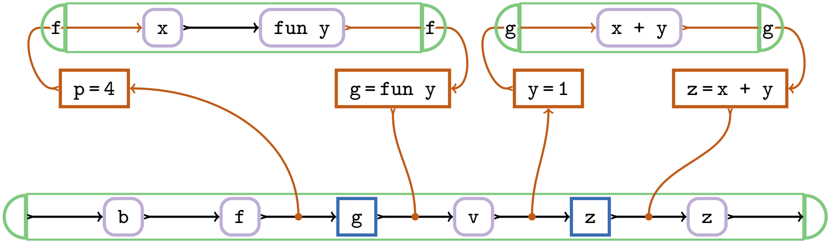
```
let f = fun p ->
    let x = p in
    fun y -> x + y
in
let g = f 4 in
let v = 1 in
let z = g v in z
```

DDPA and Variable (Mis-)Alignment

```
let b = coin_flip () in
let f = fun p ->
    let x = p in
    fun y -> x + y
in
let g = f (b?4:"s") in
let v = (b?1:"t") in
let z = g v in z
```

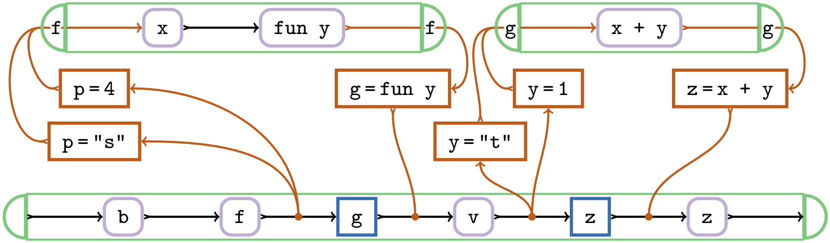
DDPA and Variable (Mis-)Alignment

```
let b = coin_flip () in
let f = fun p ->
    let x = p in
    fun y -> x + y
in
let g = f (b?4:"s") in
let v = (b?1:"t") in
let z = g v in z
```



DDPA and Variable (Mis-)Alignment

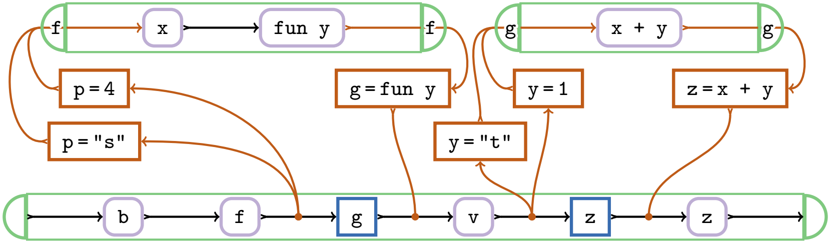
```
let b = coin_flip () in
let f = fun p ->
    let x = p in
    fun y -> x + y
in
let g = f (b?4:"s") in
let v = (b?1:"t") in
let z = g v in z
```



DDPA and Variable (Mis-)Alignment

```
let b = coin_flip () in
let f = fun p ->
      let x = p in
      fun y -> x + y
in
let g = f (b?4:"s") in
let v = (b?1:"t") in
let z = g v in z
```

Possible values of x + y?

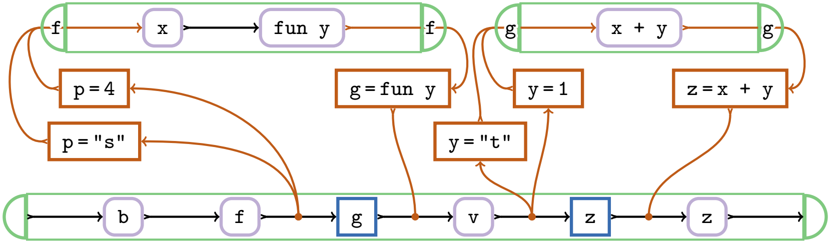


DDPA and Variable (Mis-)Alignment

```
let b = coin_flip () in
let f = fun p ->
      let x = p in
      fun y -> x + y
in
let g = f (b?4:"s") in
let v = (b?1:"t") in
let z = g v in z
```

Possible values of `x + y`?

$x \in \{4, \texttt{"s"}\}$



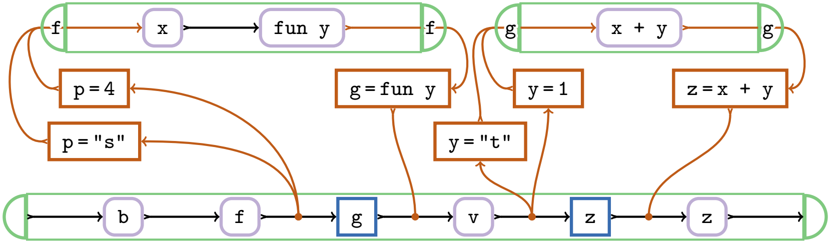
DDPA and Variable (Mis-)Alignment

```
let b = coin_flip () in
let f = fun p ->
     let x = p in
     fun y -> x + y
in
let g = f (b?4:"s") in
let v = (b?1:"t") in
let z = g v in z
```

Possible values of `x + y`?

$x \in \{4, "s"\}$

$y \in \{1, "t"\}$



DDPA and Variable (Mis-)Alignment

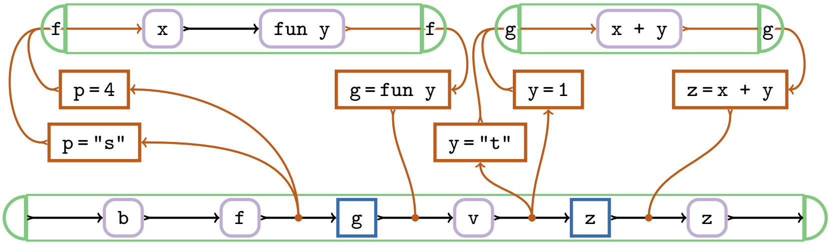
```
let b = coin_flip () in
let f = fun p ->
      let x = p in
      fun y -> x + y
in
let g = f (b?4:"s") in
let v = (b?1:"t") in
let z = g v in z
```

Possible values of `x + y`?

$\texttt{x} \in \{4, \texttt{"s"}\}$

$\texttt{y} \in \{1, \texttt{"t"}\}$

$$\texttt{x + y} = \left\{ \begin{array}{ccc} 4 & + & 1 \\ 4 & + & \texttt{"t"} \\ \texttt{"s"} & + & 1 \\ \texttt{"s"} & + & \texttt{"t"} \end{array} \right\}$$



DDPA and Variable (Mis-)Alignment

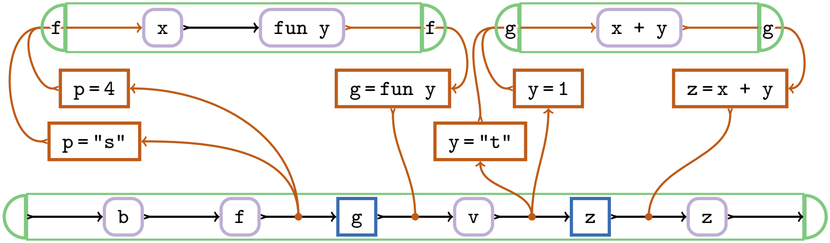
```
let b = coin_flip () in
let f = fun p ->
      let x = p in
      fun y -> x + y
in
let g = f (b?4:"s") in
let v = (b?1:"t") in
let z = g v in z
```

Possible values of x + y?

$x \in \{4, \text{"s"}\}$

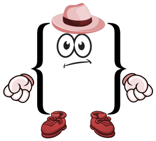
$y \in \{1, \text{"t"}\}$

$$x + y = \left\{ \begin{array}{ccc} 4 & + & 1 \\ 4 & + & \text{"t"} \\ \text{"s"} & + & 1 \\ \text{"s"} & + & \text{"t"} \end{array} \right\}$$

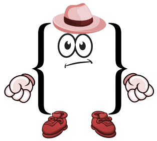


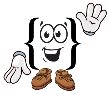
DRSF

DRSF

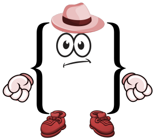
$\bigcup$  $= \sum$

DRSF

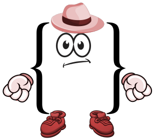
$\bigcup$  $=\sum$



DRSF

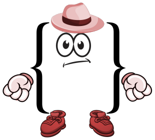
$\bigcup$  $= \sum$ $\neq$

DRSF

$\bigcup$  $= \sum$ $\neq$

$= \{\hat{x}@\Delta \mapsto \hat{v}, \ldots\}$

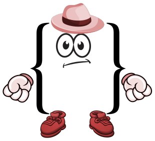
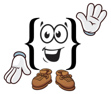
DRSF

$\bigcup$  $= \sum$ $\neq$

$= \{\hat{x} @ \Delta \mapsto \hat{v}, \ldots\}$

$$\Delta = [\delta, \ldots]$$

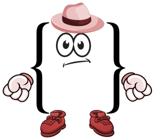
DRSF

$\bigcup$  $= \sum$ $\neq$ 

$= \{\hat{x} @ \Delta \mapsto \hat{v}, \ldots\}$

$\Delta = [\delta, \ldots] \quad \delta ::= ◖\mathtt{x} \mid ◗\mathtt{x}$

DRSF

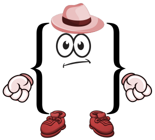
$\bigcup$  $= \sum$ $\neq$

$= \{\hat{x}@\Delta \mapsto \hat{v}, \ldots\}$

$$\Delta = [\delta, \ldots] \quad \delta ::= \llparenthesis \mathtt{x} \mid \rrparenthesis \mathtt{x}$$

- ΔCFA [POPL 06] (abstract frame strings)
- PDCFA [JFP #24 (2014)] (stack deltas, reachability)

DRSF

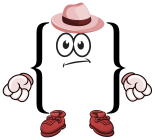
$\bigcup$  $= \sum$ $\neq$

$= \{\hat{x}@\Delta \mapsto \hat{v}, \ldots\}$

$$\Delta = [\delta, \ldots] \quad \delta ::= ◖\mathtt{x} \mid ◗\mathtt{x}$$

- ΔCFA [POPL 06] (abstract frame strings)
- PDCFA [JFP #24 (2014)] (stack deltas, reachability)
- Little stores are <u>incomplete</u>

DRSF

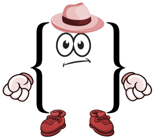
$\bigcup$  $=\sum$ $\neq$

$=\{\hat{x}@\Delta \mapsto \hat{v},\ldots\}$

$$\Delta = [\delta,\ldots] \quad \delta ::= \llparenthesis \mathtt{x} \mid \rrparenthesis \mathtt{x}$$

- ΔCFA [POPL 06] (abstract frame strings)
- PDCFA [JFP #24 (2014)] (stack deltas, reachability)
- Little stores are incomplete
- Relative (vs. DDPA's absolute)

DRSF

$\bigcup$  $= \sum$ $\neq$

$= \{\hat{x} @ \Delta \mapsto \hat{v}, \ldots\}$

$$\Delta = [\delta, \ldots] \quad \delta ::= \llparenthesis \mathtt{x} \mid \rrparenthesis \mathtt{x}$$

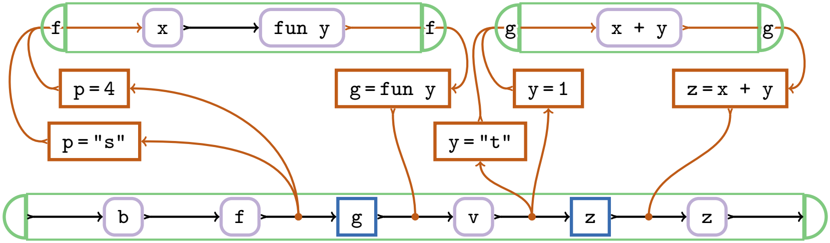
- ΔCFA [POPL 06] (abstract frame strings)
- PDCFA [JFP #24 (2014)] (stack deltas, reachability)
- Little stores are incomplete
- Relative (vs. DDPA's absolute)

Demand-Driven Higher-Order Program Analyses

	DDPA	**DRSF**
Context-sensitive	✓ Contours	✓ Little Stores
Flow-sensitive	✓ Natural	✓ Little Stores
Path-sensitive	∼ Filters	✓ Little Stores
Must-alias	∼ A Mess	✓ Little Stores
Non-local variables	✓ Lookup	✓ Lookup

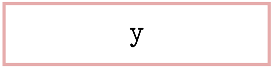
DRSF and Variable Alignment

```
let b = coin_flip () in
let f = fun p ->
    let x = p in
    fun y -> x + y
in
let g = f (b?4:"s") in
let v = (b?1:"t") in
let z = g v in z
```

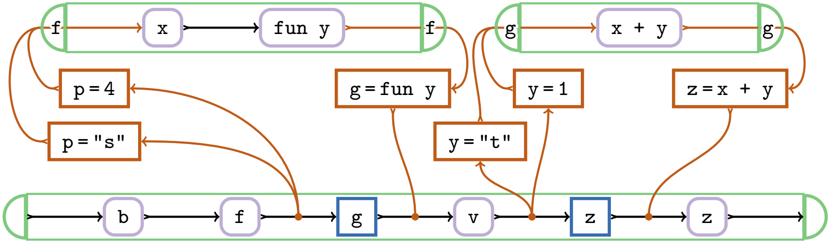


DRSF and Variable Alignment

```
let b = coin_flip () in
let f = fun p ->
      let x = p in
      fun y -> x + y
in
let g = f (b?4:"s") in
let v = (b?1:"t") in
let z = g v in z
```

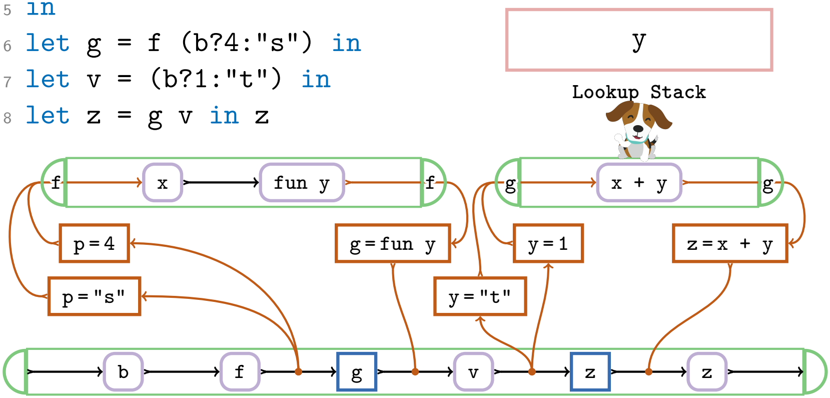


Lookup Stack



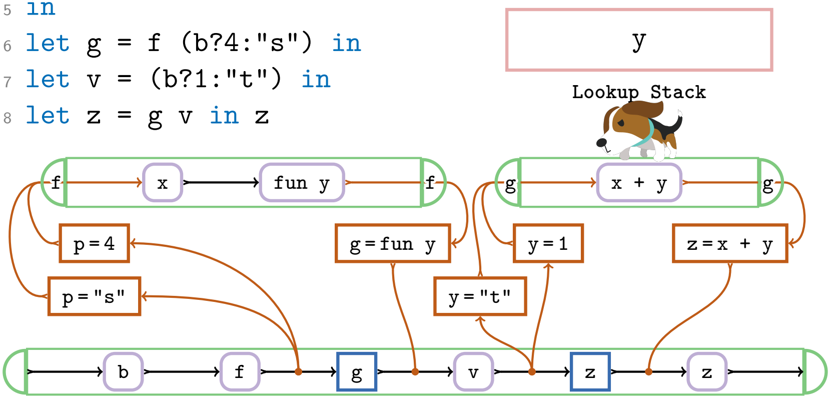
DRSF and Variable Alignment

```
let b = coin_flip () in
let f = fun p ->
      let x = p in
      fun y -> x + y
in
let g = f (b?4:"s") in
let v = (b?1:"t") in
let z = g v in z
```



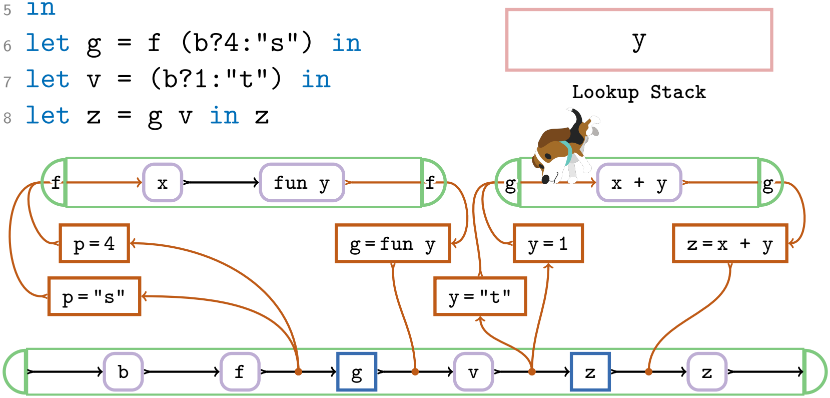
DRSF and Variable Alignment

```
let b = coin_flip () in
let f = fun p ->
      let x = p in
      fun y -> x + y
in
let g = f (b?4:"s") in
let v = (b?1:"t") in
let z = g v in z
```



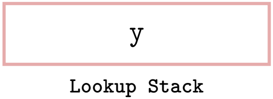
DRSF and Variable Alignment

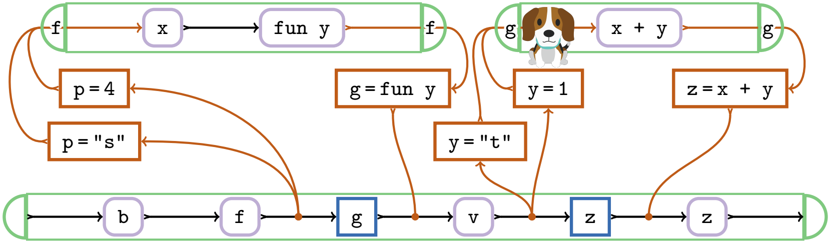
```
let b = coin_flip () in
let f = fun p ->
      let x = p in
      fun y -> x + y
in
let g = f (b?4:"s") in
let v = (b?1:"t") in
let z = g v in z
```



DRSF and Variable Alignment

```
let b = coin_flip () in
let f = fun p ->
    let x = p in
    fun y -> x + y
in
let g = f (b?4:"s") in
let v = (b?1:"t") in
let z = g v in z
```



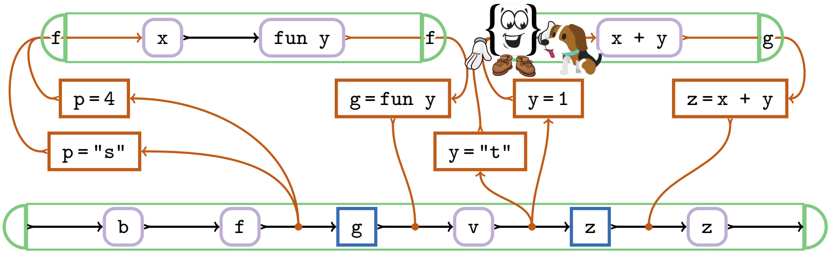


DRSF and Variable Alignment

```
let b = coin_flip () in
let f = fun p ->
      let x = p in
      fun y -> x + y
in
let g = f (b?4:"s") in
let v = (b?1:"t") in
let z = g v in z
```

$\{y@[] \mapsto 1\}$

Lookup Stack

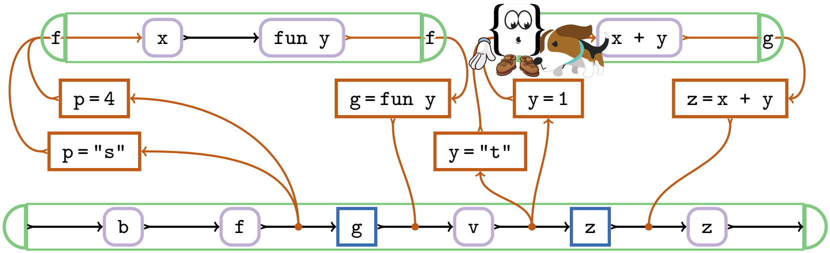


DRSF and Variable Alignment

```
let b = coin_flip () in
let f = fun p ->
      let x = p in
      fun y -> x + y
in
let g = f (b?4:"s") in
let v = (b?1:"t") in
let z = g v in z
```

b
$\{y@[] \mapsto 1\}$

Lookup Stack

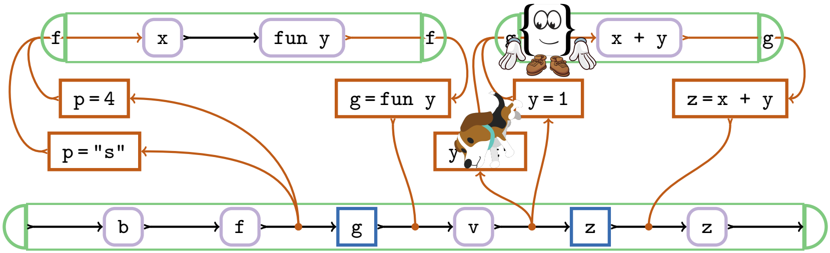


DRSF and Variable Alignment

```
let b = coin_flip () in
let f = fun p ->
    let x = p in
    fun y -> x + y
in
let g = f (b?4:"s") in
let v = (b?1:"t") in
let z = g v in z
```

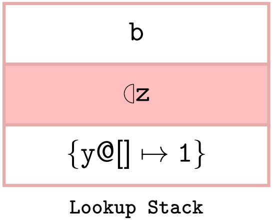
b
$\{y@[] \mapsto 1\}$

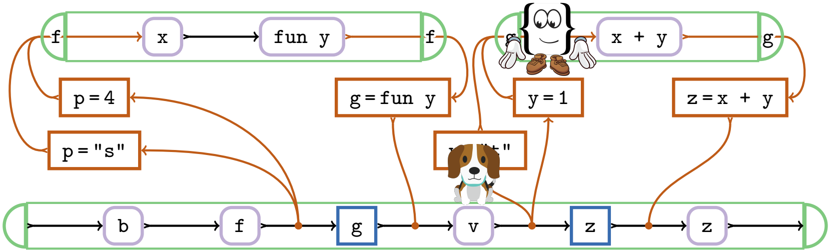
Lookup Stack



DRSF and Variable Alignment

```
let b = coin_flip () in
let f = fun p ->
    let x = p in
    fun y -> x + y
in
let g = f (b?4:"s") in
let v = (b?1:"t") in
let z = g v in z
```



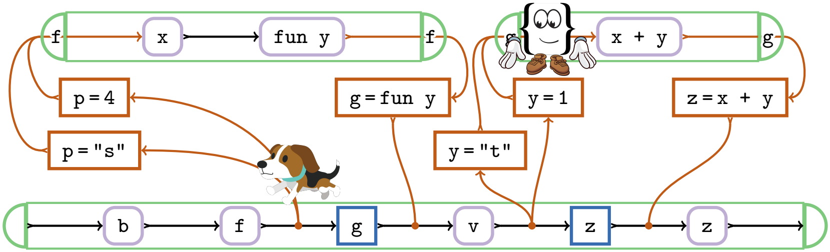


DRSF and Variable Alignment

```
let b = coin_flip () in
let f = fun p ->
      let x = p in
      fun y -> x + y
in
let g = f (b?4:"s") in
let v = (b?1:"t") in
let z = g v in z
```

b
◖z
{y@[] ↦ 1}

Lookup Stack

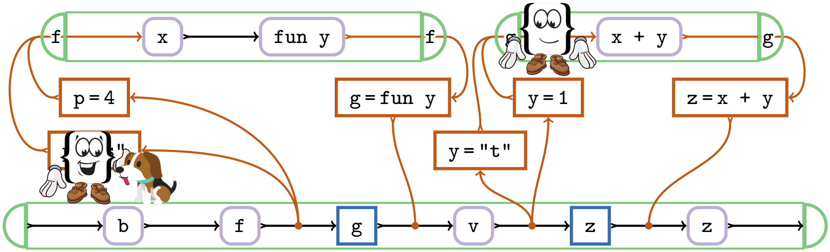


DRSF and Variable Alignment

```
let b = coin_flip () in
let f = fun p ->
      let x = p in
      fun y -> x + y
in
let g = f (b?4:"s") in
let v = (b?1:"t") in
let z = g v in z
```

$\{$b@[] $\mapsto$ true$\}$
◖z
$\{$y@[] $\mapsto$ 1$\}$

Lookup Stack

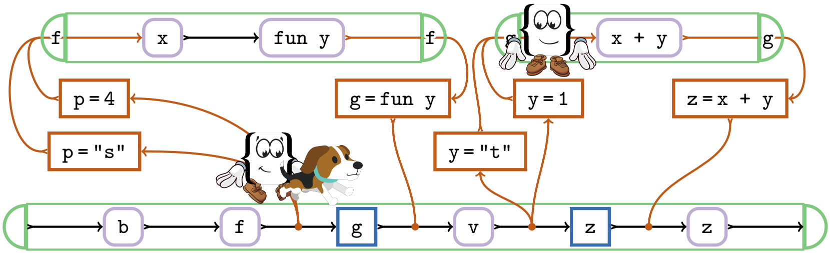


DRSF and Variable Alignment

```
let b = coin_flip () in
let f = fun p ->
    let x = p in
    fun y -> x + y
in
let g = f (b?4:"s") in
let v = (b?1:"t") in
let z = g v in z
```

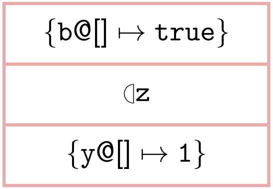
{b@[] ↦ true}
◖z
{y@[] ↦ 1}

Lookup Stack

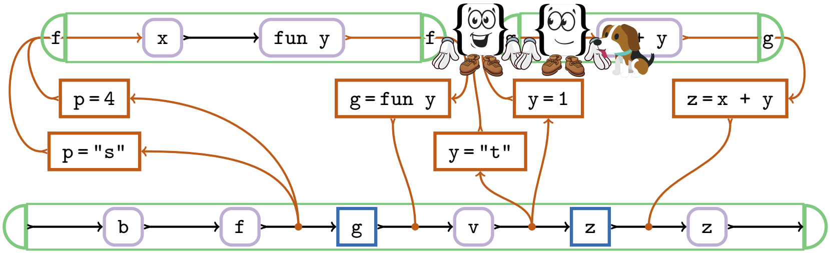


DRSF and Variable Alignment

```
let b = coin_flip () in
let f = fun p ->
      let x = p in
      fun y -> x + y
in
let g = f (b?4:"s") in
let v = (b?1:"t") in
let z = g v in z
```



Lookup Stack



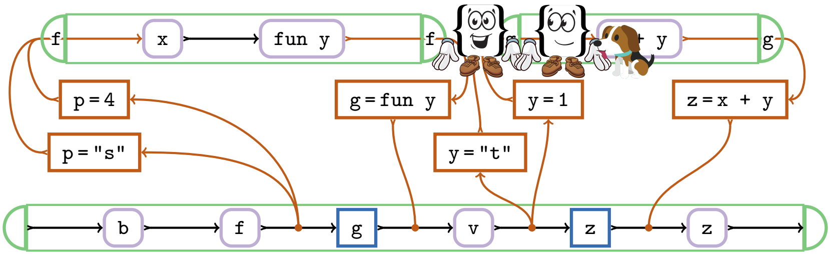
DRSF and Variable Alignment

```
let b = coin_flip () in
let f = fun p ->
    let x = p in
    fun y -> x + y
in
let g = f (b?4:"s") in
let v = (b?1:"t") in
let z = g v in z
```

$\{b@[\triangleleft z] \mapsto \texttt{true}\}$

$\{y@[] \mapsto 1\}$

Lookup Stack

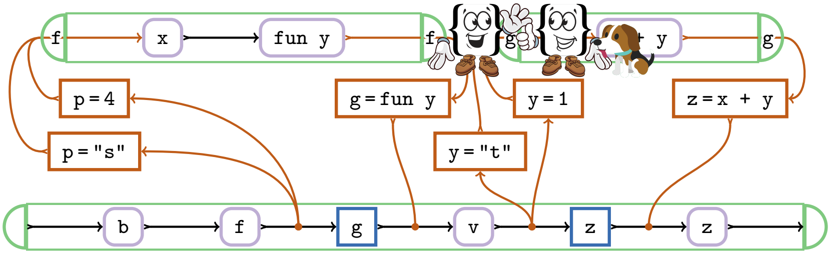


DRSF and Variable Alignment

```
let b = coin_flip () in
let f = fun p ->
      let x = p in
      fun y -> x + y
in
let g = f (b?4:"s") in
let v = (b?1:"t") in
let z = g v in z
```

$\{b@[\langle z] \mapsto \texttt{true}\}$
$\{y@[] \mapsto 1\}$

Lookup Stack

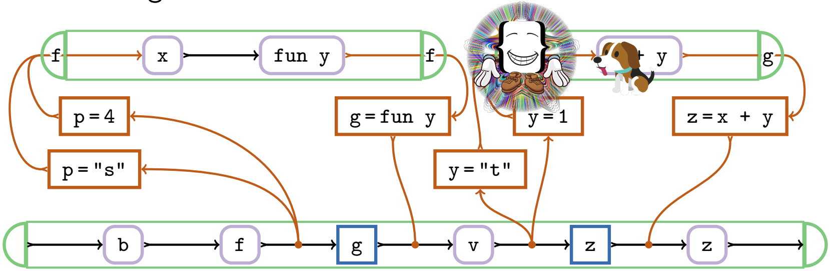


DRSF and Variable Alignment

```
let b = coin_flip () in
let f = fun p ->
      let x = p in
      fun y -> x + y
in
let g = f (b?4:"s") in
let v = (b?1:"t") in
let z = g v in z
```

$$\left\{ \begin{array}{c} \texttt{y@[]} \mapsto \texttt{1}, \\ \texttt{b@[◖z]} \mapsto \texttt{true} \end{array} \right\}$$

Lookup Stack

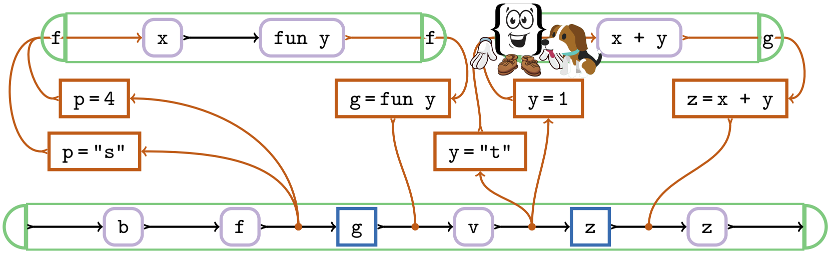


DRSF and Variable Alignment

```
let b = coin_flip () in
let f = fun p ->
      let x = p in
      fun y -> x + y
in
let g = f (b?4:"s") in
let v = (b?1:"t") in
let z = g v in z
```

$$\left\{\begin{array}{c} \texttt{y}@[] \mapsto \texttt{1}, \\ \texttt{b}@[\langle\texttt{z}] \mapsto \texttt{true} \end{array}\right\}$$

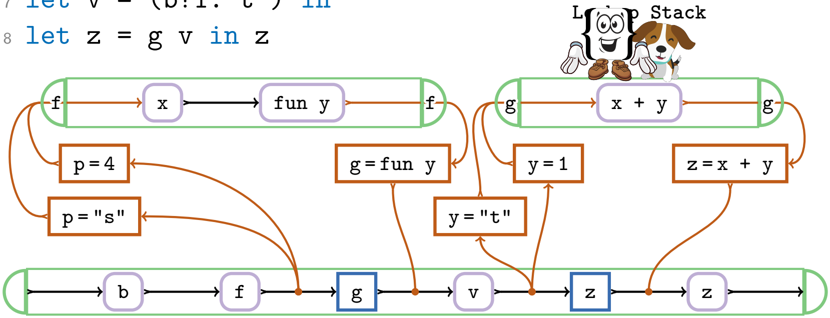
Lookup Stack



DRSF and Variable Alignment

```
let b = coin_flip () in
let f = fun p ->
    let x = p in
    fun y -> x + y
in
let g = f (b?4:"s") in
let v = (b?1:"t") in
let z = g v in z
```

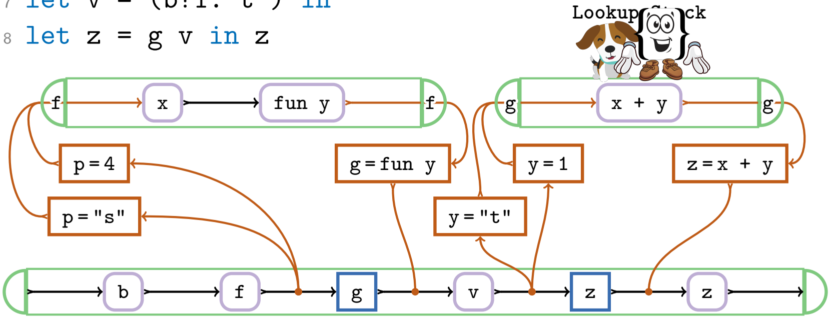
$$\left\{ \begin{array}{c} \mathtt{y}@[] \mapsto 1, \\ \mathtt{b}@[\triangleleft \mathtt{z}] \mapsto \mathtt{true} \end{array} \right\}$$



DRSF and Variable Alignment

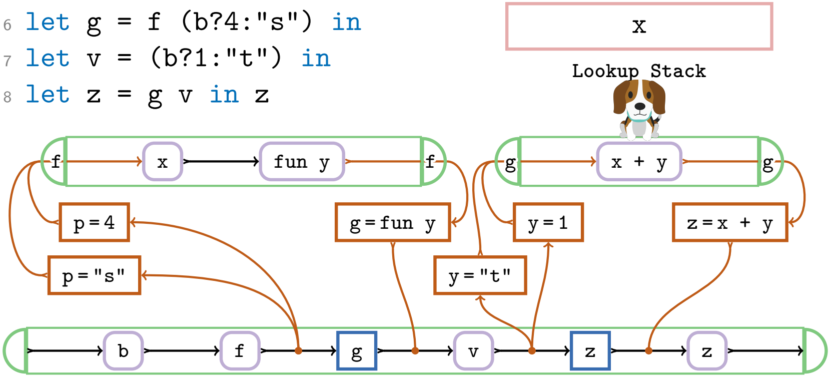
```
let b = coin_flip () in
let f = fun p ->
    let x = p in
    fun y -> x + y
in
let g = f (b?4:"s") in
let v = (b?1:"t") in
let z = g v in z
```

$$\left\{\begin{array}{c} y@[] \mapsto \texttt{"t"}, \\ b@[\lhd z] \mapsto \texttt{false} \end{array}\right\}$$



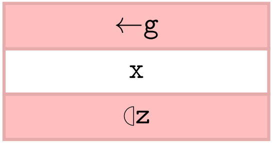
DRSF and Variable Alignment

```
let b = coin_flip () in
let f = fun p ->
      let x = p in
      fun y -> x + y
in
let g = f (b?4:"s") in
let v = (b?1:"t") in
let z = g v in z
```

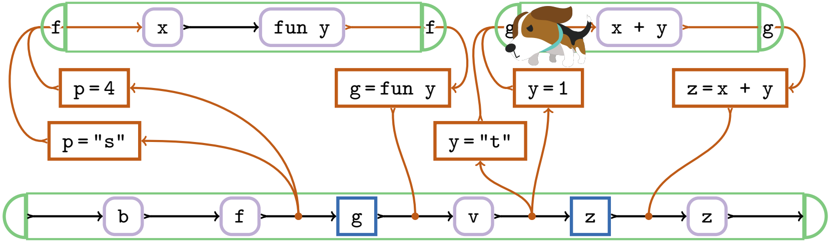


DRSF and Variable Alignment

```
let b = coin_flip () in
let f = fun p ->
    let x = p in
    fun y -> x + y
in
let g = f (b?4:"s") in
let v = (b?1:"t") in
let z = g v in z
```



Lookup Stack

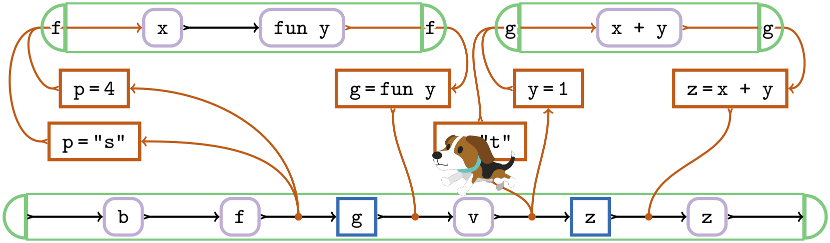


DRSF and Variable Alignment

```
let b = coin_flip () in
let f = fun p ->
    let x = p in
    fun y -> x + y
in
let g = f (b?4:"s") in
let v = (b?1:"t") in
let z = g v in z
```

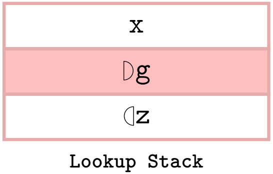
←g
x
◖z

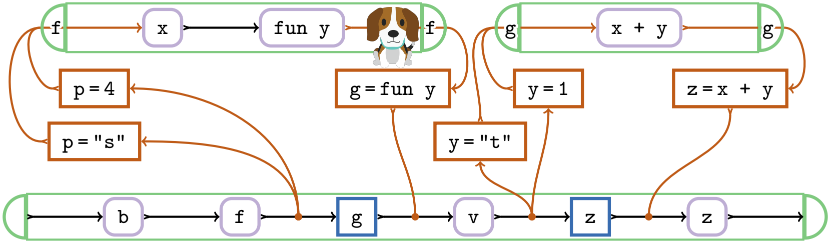
Lookup Stack



DRSF and Variable Alignment

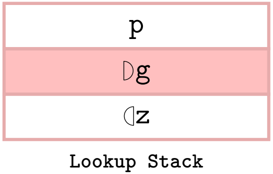
```
let b = coin_flip () in
let f = fun p ->
      let x = p in
      fun y -> x + y
in
let g = f (b?4:"s") in
let v = (b?1:"t") in
let z = g v in z
```

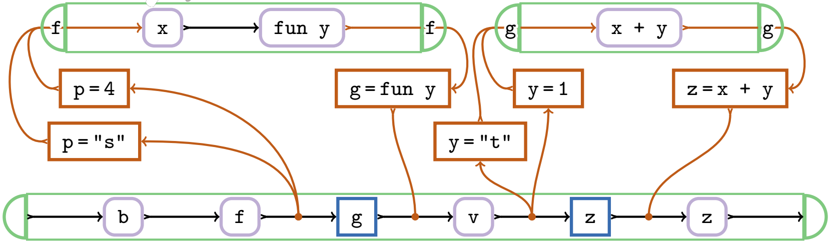




DRSF and Variable Alignment

```
let b = coin_flip () in
let f = fun p ->
      let x = p in
      fun y -> x + y
in
let g = f (b?4:"s") in
let v = (b?1:"t") in
let z = g v in z
```



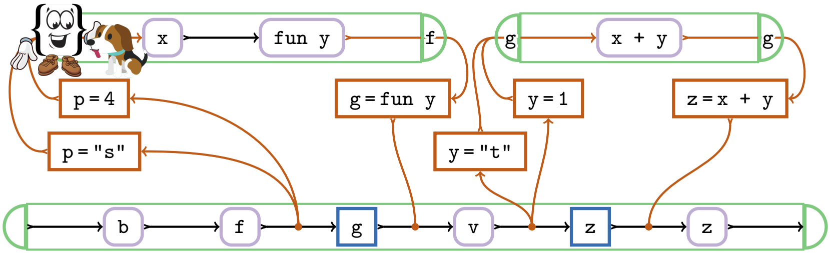


DRSF and Variable Alignment

```
let b = coin_flip () in
let f = fun p ->
    let x = p in
    fun y -> x + y
in
let g = f (b?4:"s") in
let v = (b?1:"t") in
let z = g v in z
```

$\{p@[] \mapsto 4\}$
◗g
◖z

Lookup Stack

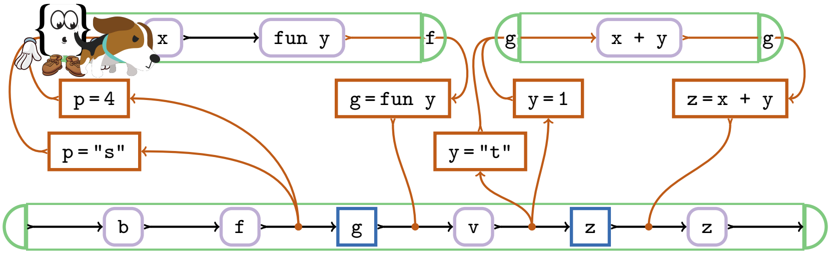


DRSF and Variable Alignment

```
let b = coin_flip () in
let f = fun p ->
      let x = p in
      fun y -> x + y
in
let g = f (b?4:"s") in
let v = (b?1:"t") in
let z = g v in z
```

b
{p@[] $\mapsto$ 4}
◗g
◖z

Lookup Stack

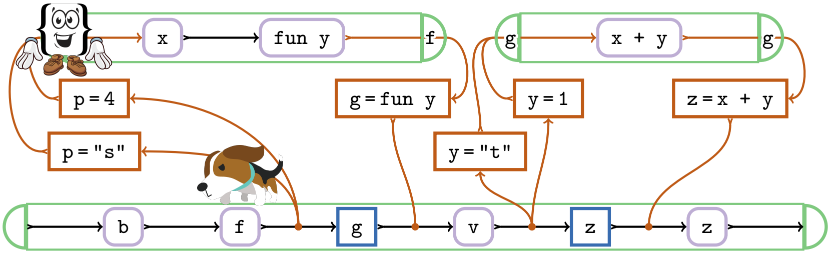


DRSF and Variable Alignment

```
let b = coin_flip () in
let f = fun p ->
      let x = p in
      fun y -> x + y
in
let g = f (b?4:"s") in
let v = (b?1:"t") in
let z = g v in z
```

b
◖g
{p@[] $\mapsto$ 4}
◗g
◖z

Lookup Stack

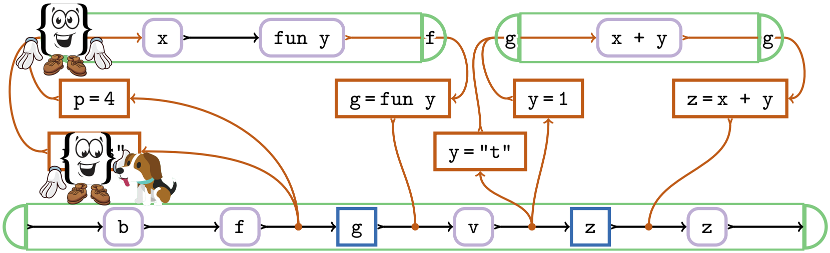


DRSF and Variable Alignment

```
let b = coin_flip () in
let f = fun p ->
          let x = p in
          fun y -> x + y
in
let g = f (b?4:"s") in
let v = (b?1:"t") in
let z = g v in z
```

{b@[] ↦ true}
◖g
{p@[] ↦ 4}
◗g
◖z

Lookup Stack

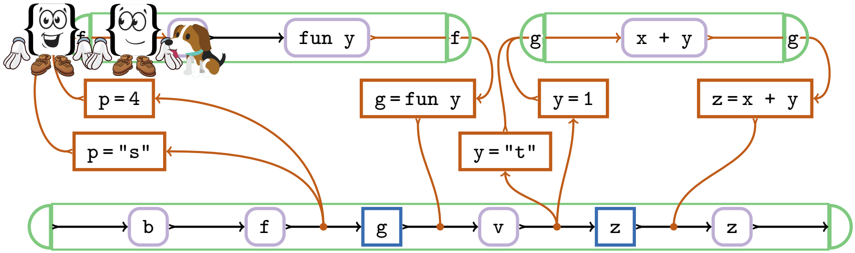


DRSF and Variable Alignment

```
let b = coin_flip () in
let f = fun p ->
      let x = p in
      fun y -> x + y
in
let g = f (b?4:"s") in
let v = (b?1:"t") in
let z = g v in z
```

{b@[◖g] ↦ true}
{p@[] ↦ 4}
◗g
◖z

Lookup Stack

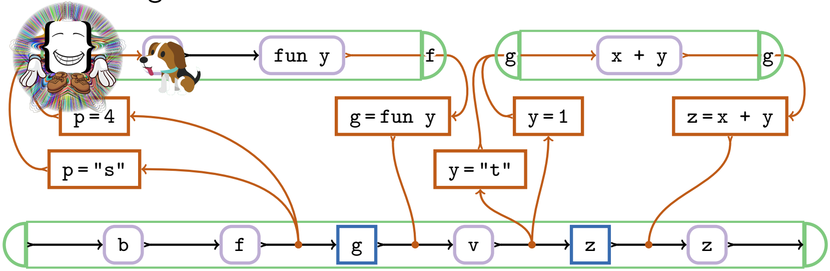


DRSF and Variable Alignment

```
let b = coin_flip () in
let f = fun p ->
      let x = p in
      fun y -> x + y
in
let g = f (b?4:"s") in
let v = (b?1:"t") in
let z = g v in z
```

$\left\{ \begin{array}{c} \texttt{p@[]} \mapsto 4, \\ \texttt{b@[◖g]} \mapsto \texttt{true} \end{array} \right\}$
◗g
◖z

Lookup Stack



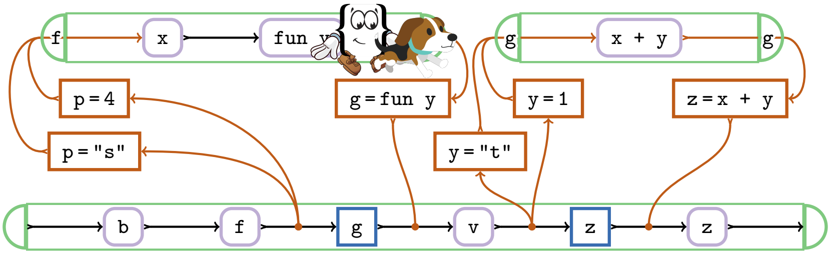
DRSF and Variable Alignment

```
let b = coin_flip () in
let f = fun p ->
    let x = p in
    fun y -> x + y
in
let g = f (b?4:"s") in
let v = (b?1:"t") in
let z = g v in z
```

$$\left\{\begin{array}{l} \texttt{p}@[◗\texttt{g}] \mapsto 4, \\ \texttt{b}@[] \mapsto \texttt{true} \end{array}\right\}$$

◖z

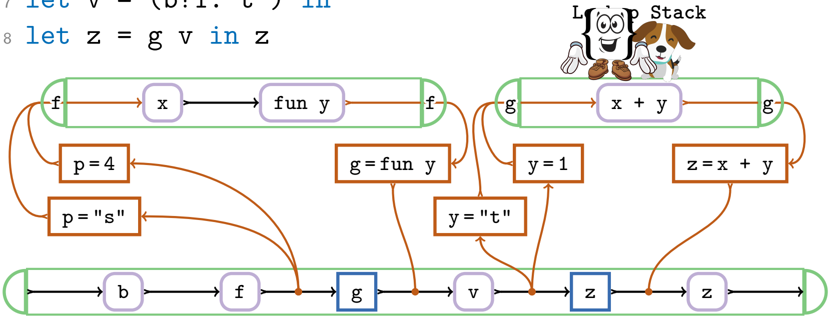
Lookup Stack



DRSF and Variable Alignment

```
let b = coin_flip () in
let f = fun p ->
      let x = p in
      fun y -> x + y
in
let g = f (b?4:"s") in
let v = (b?1:"t") in
let z = g v in z
```

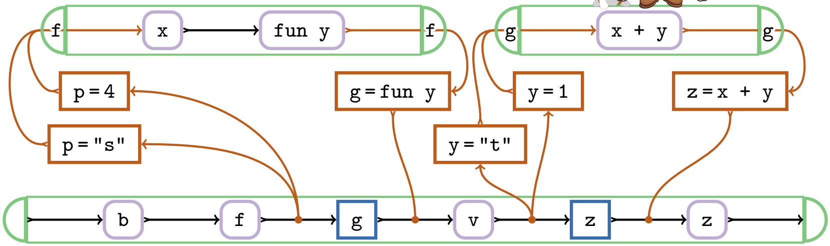
$$\left\{ \begin{array}{l} p@[\rhd g \lhd z] \mapsto 4, \\ b@[\lhd z] \mapsto \mathtt{true} \end{array} \right\}$$



DRSF and Variable Alignment

```
let b = coin_flip () in
let f = fun p ->
      let x = p in
      fun y -> x + y
in
let g = f (b?4:"s") in
let v = (b?1:"t") in
let z = g v in z
```

$$\left\{ \begin{array}{l} p@[\triangleright g \triangleleft z] \mapsto \texttt{"s"}, \\ b@[\triangleleft z] \mapsto \texttt{false} \end{array} \right\}$$



Merging Relative Store Fragments

$$\left\{\begin{matrix} \texttt{x@[]} \mapsto 4, \\ \texttt{b@[}\LEFTcircle\texttt{z]} \mapsto \texttt{true} \end{matrix}\right\} \oplus \left\{\begin{matrix} \texttt{y@[]} \mapsto 1, \\ \texttt{b@[}\LEFTcircle\texttt{z]} \mapsto \texttt{true} \end{matrix}\right\} =$$

Merging Relative Store Fragments

$$\left\{\begin{matrix} \mathtt{x@[]} \mapsto \mathtt{4}, \\ \mathtt{b@[\lhd z]} \mapsto \mathtt{true} \end{matrix}\right\} \oplus \left\{\begin{matrix} \mathtt{y@[]} \mapsto \mathtt{1}, \\ \mathtt{b@[\lhd z]} \mapsto \mathtt{true} \end{matrix}\right\} = \left\{\begin{matrix} \mathtt{x@[]} \mapsto \mathtt{4}, \\ \mathtt{y@[]} \mapsto \mathtt{1}, \\ \mathtt{b@[\lhd z]} \mapsto \mathtt{true} \end{matrix}\right\}$$

Merging Relative Store Fragments

$$\left\{\begin{array}{c}\texttt{x@[]} \mapsto \texttt{4}, \\ \texttt{b@[}\text{◖}\texttt{z]} \mapsto \texttt{true}\end{array}\right\} \oplus \left\{\begin{array}{c}\texttt{y@[]} \mapsto \texttt{1}, \\ \texttt{b@[}\text{◖}\texttt{z]} \mapsto \texttt{true}\end{array}\right\} = \left\{\begin{array}{c}\texttt{x@[]} \mapsto \texttt{4}, \\ \texttt{y@[]} \mapsto \texttt{1}, \\ \texttt{b@[}\text{◖}\texttt{z]} \mapsto \texttt{true}\end{array}\right\}$$

$$\left\{\begin{array}{c}\texttt{x@[]} \mapsto \texttt{"s"}, \\ \texttt{b@[}\text{◖}\texttt{z]} \mapsto \texttt{false}\end{array}\right\} \oplus \left\{\begin{array}{c}\texttt{y@[]} \mapsto \texttt{"t"}, \\ \texttt{b@[}\text{◖}\texttt{z]} \mapsto \texttt{false}\end{array}\right\} =$$

Merging Relative Store Fragments

$$\left\{\begin{matrix}\texttt{x@}[] \mapsto 4, \\ \texttt{b@}[\llparenthesis\texttt{z}] \mapsto \texttt{true}\end{matrix}\right\} \oplus \left\{\begin{matrix}\texttt{y@}[] \mapsto 1, \\ \texttt{b@}[\llparenthesis\texttt{z}] \mapsto \texttt{true}\end{matrix}\right\} = \left\{\begin{matrix}\texttt{x@}[] \mapsto 4, \\ \texttt{y@}[] \mapsto 1, \\ \texttt{b@}[\llparenthesis\texttt{z}] \mapsto \texttt{true}\end{matrix}\right\}$$

$$\left\{\begin{matrix}\texttt{x@}[] \mapsto \texttt{"s"}, \\ \texttt{b@}[\llparenthesis\texttt{z}] \mapsto \texttt{false}\end{matrix}\right\} \oplus \left\{\begin{matrix}\texttt{y@}[] \mapsto \texttt{"t"}, \\ \texttt{b@}[\llparenthesis\texttt{z}] \mapsto \texttt{false}\end{matrix}\right\} = \left\{\begin{matrix}\texttt{x@}[] \mapsto \texttt{"s"}, \\ \texttt{y@}[] \mapsto \texttt{"t"}, \\ \texttt{b@}[\llparenthesis\texttt{z}] \mapsto \texttt{false}\end{matrix}\right\}$$

Merging Relative Store Fragments

$$\left\{\begin{matrix}\texttt{x@}[] \mapsto 4, \\ \texttt{b@}[\llparenthesis\texttt{z}] \mapsto \texttt{true}\end{matrix}\right\} \oplus \left\{\begin{matrix}\texttt{y@}[] \mapsto 1, \\ \texttt{b@}[\llparenthesis\texttt{z}] \mapsto \texttt{true}\end{matrix}\right\} = \left\{\begin{matrix}\texttt{x@}[] \mapsto 4, \\ \texttt{y@}[] \mapsto 1, \\ \texttt{b@}[\llparenthesis\texttt{z}] \mapsto \texttt{true}\end{matrix}\right\}$$

$$\left\{\begin{matrix}\texttt{x@}[] \mapsto \texttt{"s"}, \\ \texttt{b@}[\llparenthesis\texttt{z}] \mapsto \texttt{false}\end{matrix}\right\} \oplus \left\{\begin{matrix}\texttt{y@}[] \mapsto \texttt{"t"}, \\ \texttt{b@}[\llparenthesis\texttt{z}] \mapsto \texttt{false}\end{matrix}\right\} = \left\{\begin{matrix}\texttt{x@}[] \mapsto \texttt{"s"}, \\ \texttt{y@}[] \mapsto \texttt{"t"}, \\ \texttt{b@}[\llparenthesis\texttt{z}] \mapsto \texttt{false}\end{matrix}\right\}$$

$$\left\{\begin{matrix}\texttt{x@}[] \mapsto 4, \\ \texttt{b@}[\llparenthesis\texttt{z}] \mapsto \texttt{true}\end{matrix}\right\} \oplus \left\{\begin{matrix}\texttt{y@}[] \mapsto \texttt{"t"}, \\ \texttt{b@}[\llparenthesis\texttt{z}] \mapsto \texttt{false}\end{matrix}\right\} =$$

$$\left\{\begin{matrix}\texttt{x@}[] \mapsto \texttt{"s"}, \\ \texttt{b@}[\llparenthesis\texttt{z}] \mapsto \texttt{false}\end{matrix}\right\} \oplus \left\{\begin{matrix}\texttt{y@}[] \mapsto 1, \\ \texttt{b@}[\llparenthesis\texttt{z}] \mapsto \texttt{true}\end{matrix}\right\} =$$

Merging Relative Store Fragments

$$\left\{\begin{array}{c}\texttt{x}@[] \mapsto 4,\\ \texttt{b}@[\llparenthesis\texttt{z}] \mapsto \texttt{true}\end{array}\right\} \oplus \left\{\begin{array}{c}\texttt{y}@[] \mapsto 1,\\ \texttt{b}@[\llparenthesis\texttt{z}] \mapsto \texttt{true}\end{array}\right\} = \left\{\begin{array}{c}\texttt{x}@[] \mapsto 4,\\ \texttt{y}@[] \mapsto 1,\\ \texttt{b}@[\llparenthesis\texttt{z}] \mapsto \texttt{true}\end{array}\right\}$$

$$\left\{\begin{array}{c}\texttt{x}@[] \mapsto \texttt{"s"},\\ \texttt{b}@[\llparenthesis\texttt{z}] \mapsto \texttt{false}\end{array}\right\} \oplus \left\{\begin{array}{c}\texttt{y}@[] \mapsto \texttt{"t"},\\ \texttt{b}@[\llparenthesis\texttt{z}] \mapsto \texttt{false}\end{array}\right\} = \left\{\begin{array}{c}\texttt{x}@[] \mapsto \texttt{"s"},\\ \texttt{y}@[] \mapsto \texttt{"t"},\\ \texttt{b}@[\llparenthesis\texttt{z}] \mapsto \texttt{false}\end{array}\right\}$$

$$\left\{\begin{array}{c}\texttt{x}@[] \mapsto 4,\\ \texttt{b}@[\llparenthesis\texttt{z}] \mapsto \texttt{true}\end{array}\right\} \oplus \left\{\begin{array}{c}\texttt{y}@[] \mapsto \texttt{"t"},\\ \texttt{b}@[\llparenthesis\texttt{z}] \mapsto \texttt{false}\end{array}\right\} = \text{✗}$$

$$\left\{\begin{array}{c}\texttt{x}@[] \mapsto \texttt{"s"},\\ \texttt{b}@[\llparenthesis\texttt{z}] \mapsto \texttt{false}\end{array}\right\} \oplus \left\{\begin{array}{c}\texttt{y}@[] \mapsto 1,\\ \texttt{b}@[\llparenthesis\texttt{z}] \mapsto \texttt{true}\end{array}\right\} = \text{✗}$$

Merging Relative Store Fragments

$$\left\{\begin{array}{c}\texttt{x@[]} \mapsto 4,\\ \texttt{b@[}\llparenthesis\texttt{z]} \mapsto \texttt{true}\end{array}\right\} \oplus \left\{\begin{array}{c}\texttt{y@[]} \mapsto 1,\\ \texttt{b@[}\llparenthesis\texttt{z]} \mapsto \texttt{true}\end{array}\right\} = \left\{\begin{array}{c}\texttt{x@[]} \mapsto 4,\\ \texttt{y@[]} \mapsto 1,\\ \texttt{b@[}\llparenthesis\texttt{z]} \mapsto \texttt{true}\end{array}\right\}$$

$$\left\{\begin{array}{c}\texttt{x@[]} \mapsto \texttt{"s"},\\ \texttt{b@[}\llparenthesis\texttt{z]} \mapsto \texttt{false}\end{array}\right\} \oplus \left\{\begin{array}{c}\texttt{y@[]} \mapsto \texttt{"t"},\\ \texttt{b@[}\llparenthesis\texttt{z]} \mapsto \texttt{false}\end{array}\right\} = \left\{\begin{array}{c}\texttt{x@[]} \mapsto \texttt{"s"},\\ \texttt{y@[]} \mapsto \texttt{"t"},\\ \texttt{b@[}\llparenthesis\texttt{z]} \mapsto \texttt{false}\end{array}\right\}$$

$$\left\{\begin{array}{c}\texttt{x@[]} \mapsto 4,\\ \texttt{b@[}\llparenthesis\texttt{z]} \mapsto \texttt{true}\end{array}\right\} \oplus \left\{\begin{array}{c}\texttt{y@[]} \mapsto \texttt{"t"},\\ \texttt{b@[}\llparenthesis\texttt{z]} \mapsto \texttt{false}\end{array}\right\} = \quad \text{✗}$$

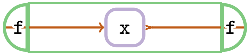
$$\left\{\begin{array}{c}\texttt{x@[]} \mapsto \texttt{"s"},\\ \texttt{b@[}\llparenthesis\texttt{z]} \mapsto \texttt{false}\end{array}\right\} \oplus \left\{\begin{array}{c}\texttt{y@[]} \mapsto 1,\\ \texttt{b@[}\llparenthesis\texttt{z]} \mapsto \texttt{true}\end{array}\right\} = \quad \text{✗}$$

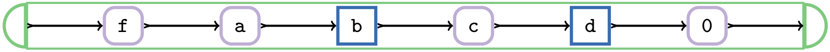
Polymorphism via Δ

```
let f = fun x -> x in
let a = 4 in
let b = f a in
let c = "s" in
let d = f c in
0
```

Polymorphism via Δ

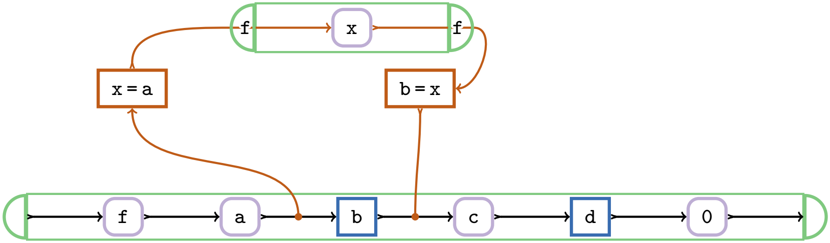
```
let f = fun x -> x in
let a = 4 in
let b = f a in
let c = "s" in
let d = f c in
0
```





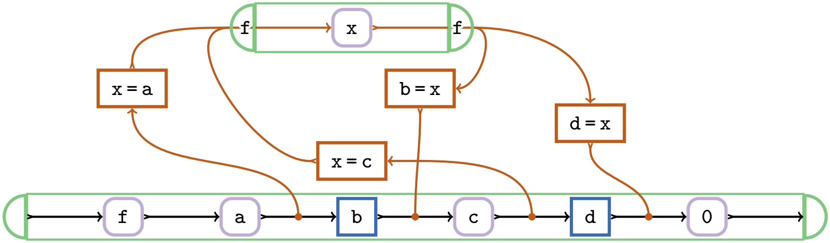
Polymorphism via Δ

```
let f = fun x -> x in
let a = 4 in
let b = f a in
let c = "s" in
let d = f c in
0
```



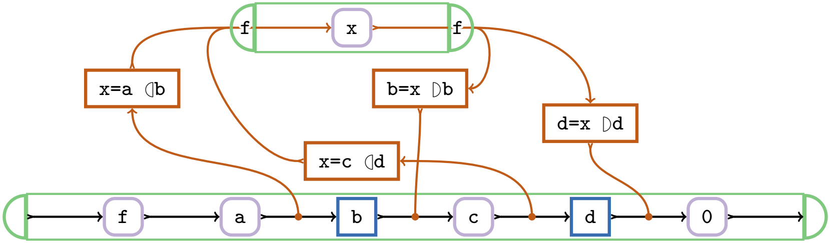
Polymorphism via Δ

```
let f = fun x -> x in
let a = 4 in
let b = f a in
let c = "s" in
let d = f c in
0
```



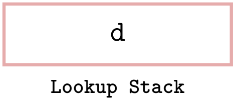
Polymorphism via Δ

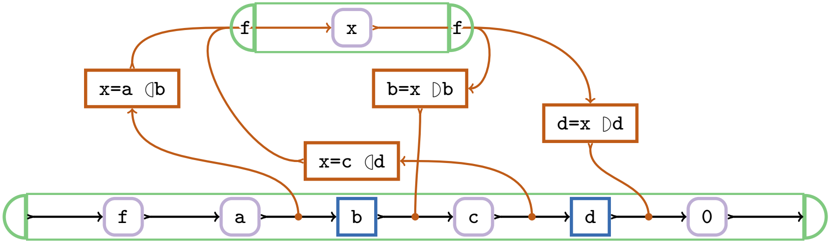
```
let f = fun x -> x in
let a = 4 in
let b = f a in
let c = "s" in
let d = f c in
0
```



Polymorphism via Δ

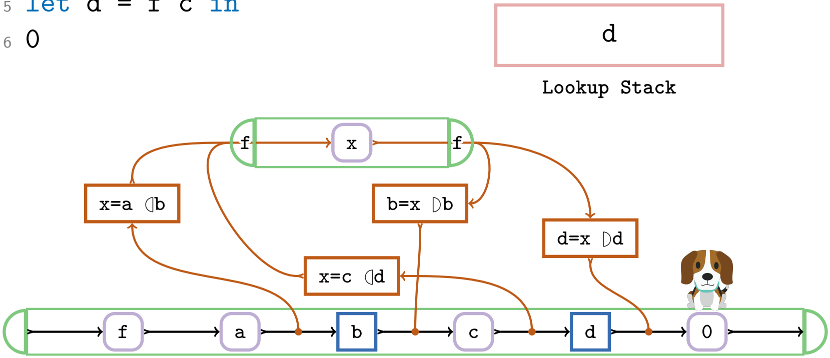
```
let f = fun x -> x in
let a = 4 in
let b = f a in
let c = "s" in
let d = f c in
0
```





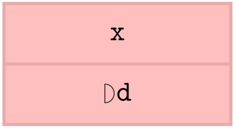
Polymorphism via Δ

```
let f = fun x -> x in
let a = 4 in
let b = f a in
let c = "s" in
let d = f c in
0
```

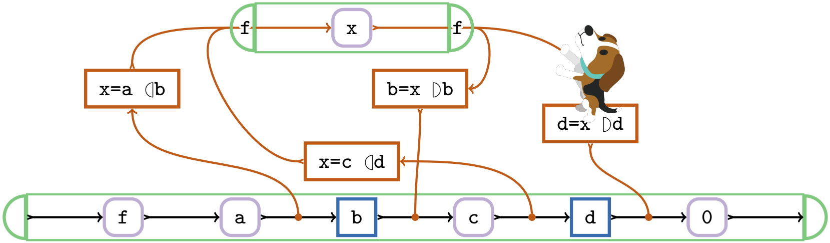


Polymorphism via Δ

```
let f = fun x -> x in
let a = 4 in
let b = f a in
let c = "s" in
let d = f c in
0
```

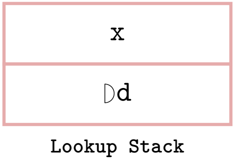


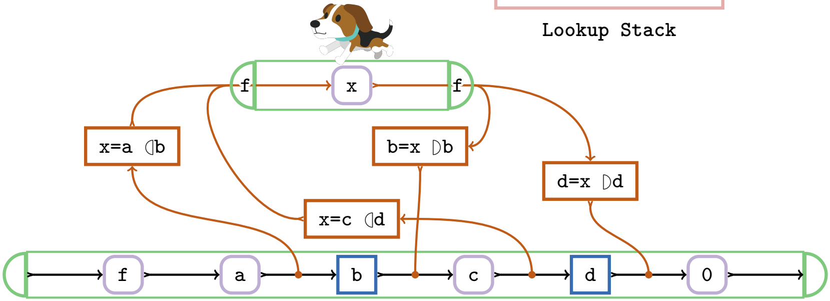
Lookup Stack



Polymorphism via Δ

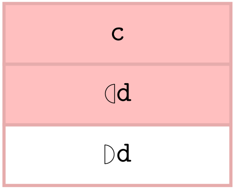
```
let f = fun x -> x in
let a = 4 in
let b = f a in
let c = "s" in
let d = f c in
0
```



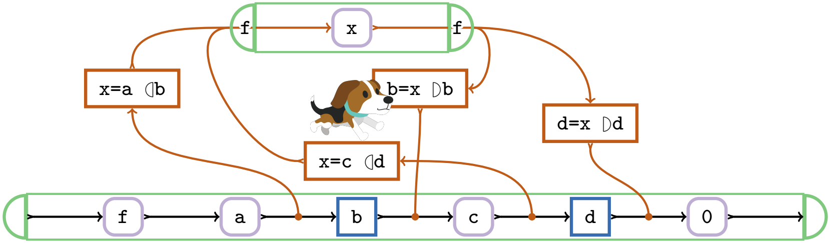


Polymorphism via Δ

```
let f = fun x -> x in
let a = 4 in
let b = f a in
let c = "s" in
let d = f c in
0
```

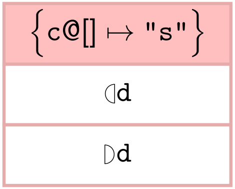


Lookup Stack

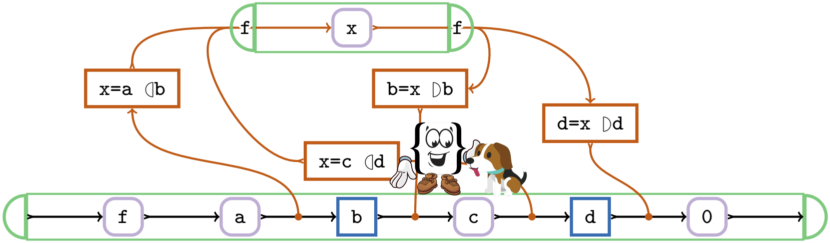


Polymorphism via Δ

```
let f = fun x -> x in
let a = 4 in
let b = f a in
let c = "s" in
let d = f c in
0
```

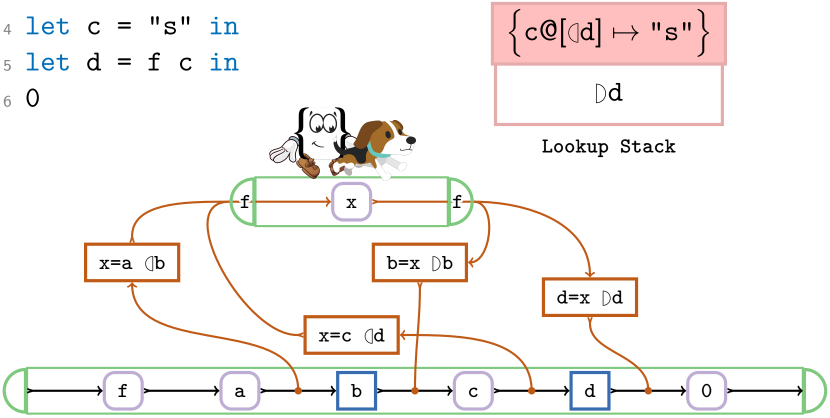


Lookup Stack



Polymorphism via Δ

```
let f = fun x -> x in
let a = 4 in
let b = f a in
let c = "s" in
let d = f c in
0
```

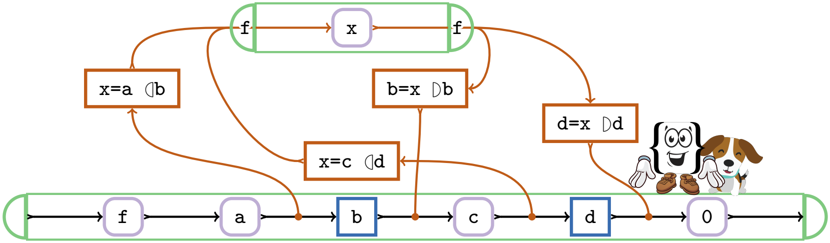


Polymorphism via Δ

```
let f = fun x -> x in
let a = 4 in
let b = f a in
let c = "s" in
let d = f c in
0
```

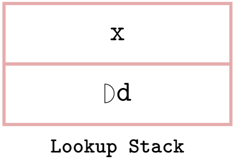
$\left\{ \texttt{c@[]} \mapsto \texttt{"s"} \right\}$

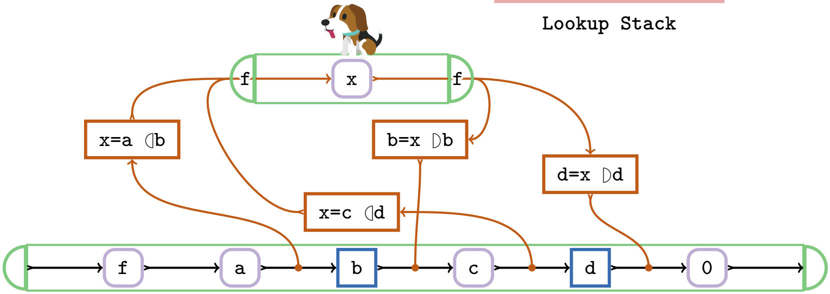
Lookup Stack



Polymorphism via Δ

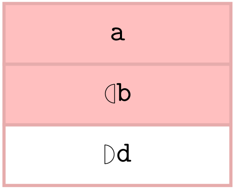
```
let f = fun x -> x in
let a = 4 in
let b = f a in
let c = "s" in
let d = f c in
0
```





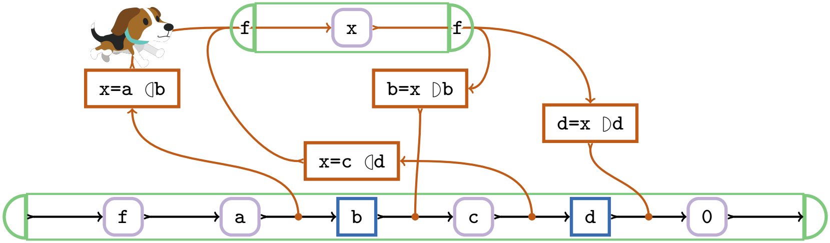
Polymorphism via Δ

```
let f = fun x -> x in
let a = 4 in
let b = f a in
let c = "s" in
let d = f c in
0
```



a
◖b
◗d

Lookup Stack

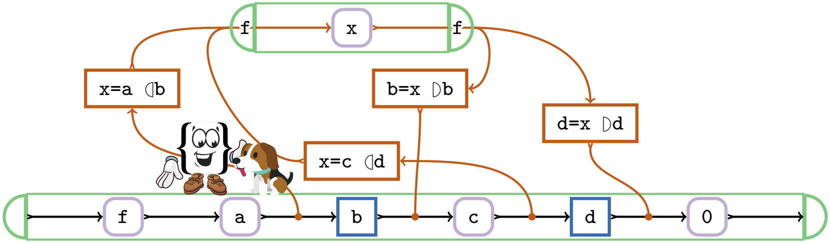


Polymorphism via Δ

```
let f = fun x -> x in
let a = 4 in
let b = f a in
let c = "s" in
let d = f c in
0
```

$\left\{ \texttt{c@[]} \mapsto 4 \right\}$
◖b
◗d

Lookup Stack

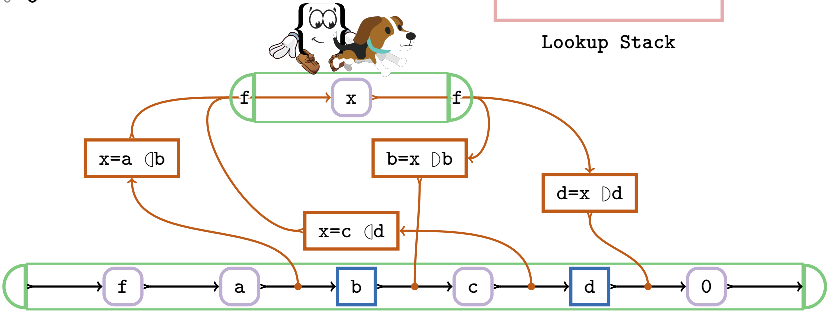


Polymorphism via Δ

```
let f = fun x -> x in
let a = 4 in
let b = f a in
let c = "s" in
let d = f c in
0
```

$\left\{ \mathtt{c}@[◖\mathtt{b}] \mapsto 4 \right\}$
◗d

Lookup Stack

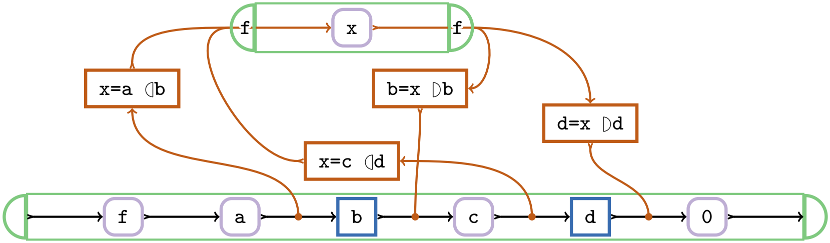


Polymorphism via Δ

```
let f = fun x -> x in
let a = 4 in
let b = f a in
let c = "s" in
let d = f c in
0
```

✗

Lookup Stack



Singleton Abstractions via Full Traces

- Traces Δ represent relative stack adjustments

Singleton Abstractions via Full Traces

- Traces Δ represent relative stack adjustments
- Decidability? Finitization

Singleton Abstractions via Full Traces

- Traces Δ represent relative stack adjustments
- Decidability? Finitization
  - kDRSF: DRSF with max Δ length k

Singleton Abstractions via Full Traces

- Traces Δ represent relative stack adjustments
- Decidability? Finitization
  - kDRSF: DRSF with max Δ length k
  - $\underline{\text{Not}}$ the same meaning as k in kCFA

Singleton Abstractions via Full Traces

- Traces Δ represent relative stack adjustments
- Decidability? Finitization
  - kDRSF: DRSF with max Δ length k
  - Not the same meaning as k in kCFA
  - Longer Δ truncated to suffix, marked partial

Singleton Abstractions via Full Traces

- Traces Δ represent relative stack adjustments
- Decidability? Finitization
  - *k*DRSF: DRSF with max Δ length *k*
  - Not the same meaning as *k* in *k*CFA
  - Longer Δ truncated to suffix, marked partial
    - $[\rhd \mathtt{a} \lhd \mathtt{b}] + \lhd \mathtt{c} \Rightarrow (\lhd \mathtt{b} \lhd \mathtt{c}]$

Singleton Abstractions via Full Traces

- Traces Δ represent relative stack adjustments
- Decidability? Finitization
    - kDRSF: DRSF with max Δ length k
    - Not the same meaning as k in kCFA
    - Longer Δ truncated to suffix, marked partial
        - $[\mathbin{\rhd}\mathtt{a}\mathbin{\lhd}\mathtt{b}] + \mathbin{\lhd}\mathtt{c} \Rightarrow (\mathbin{\lhd}\mathtt{b}\mathbin{\lhd}\mathtt{c}]$
    - Other models are possible

Singleton Abstractions via Full Traces

- Traces Δ represent relative stack adjustments
- Decidability? Finitization
  - *k*DRSF: DRSF with max Δ length *k*
  - Not the same meaning as *k* in *k*CFA
  - Longer Δ truncated to suffix, marked partial
    - [◗**a**◖**b**] + ◖**c** ⇒ (◖**b**◖**c**]
  - Other models are possible
- Full traces imply unique allocation/evaluation

Singleton Abstractions via Full Traces

- Traces Δ represent relative stack adjustments
- Decidability? Finitization
  - kDRSF: DRSF with max Δ length k
  - Not the same meaning as k in kCFA
  - Longer Δ truncated to suffix, marked partial
    - [◗**a**◖**b**] + ◖**c** ⇒ (◖**b**◖**c**]
  - Other models are possible
- Full traces imply unique allocation/evaluation
  - Used to establish shallow singleton abstractions for e.g. must-alias

Singleton Abstractions via Full Traces

- Traces Δ represent relative stack adjustments
- Decidability? Finitization
  - *k*DRSF: DRSF with max Δ length *k*
  - <u>Not</u> the same meaning as *k* in *k*CFA
  - Longer Δ truncated to suffix, marked <u>partial</u>
    - [◗**a**◖**b**] + ◖**c** ⇒ (◖**b**◖**c**]
  - Other models are possible
- Full traces imply <u>unique</u> allocation/evaluation
  - Used to establish shallow singleton abstractions for e.g. must-alias
  - Partial traces gracefully degrade

Relative Store Fragments

- Partial sets of bindings

Relative Store Fragments

- Partial sets of bindings occurring simultaneously

Relative Store Fragments

- Partial sets of bindings occurring simultaneously
- Merge discards dissonant store fragments

Relative Store Fragments

- Partial sets of bindings occurring simultaneously
- Merge discards dissonant store fragments
- Precision similar to non-store-widening analyses

Relative Store Fragments

- Partial sets of bindings occurring simultaneously
- Merge discards dissonant store fragments
- Precision similar to non-store-widening analyses
  - Worst-case complexity, too ($O(2^n)$ vs DDPA's $O(n^k)$)

Relative Store Fragments

- Partial sets of bindings occurring simultaneously
- Merge discards dissonant store fragments
- Precision similar to non-store-widening analyses
  - Worst-case complexity, too ($O(2^n)$ vs DDPA's $O(n^k)$)
- Tunable!
  - Merges described in algebraic lookup function
  - Set complex policies for precision loss

Relative Store Fragments

- Partial sets of bindings occurring simultaneously
- Merge discards dissonant store fragments
- Precision similar to non-store-widening analyses
  - Worst-case complexity, too ($O(2^n)$ vs DDPA's $O(n^k)$)
- <u>Tunable!</u>
  - Merges described in algebraic lookup function
  - Set complex policies for precision loss
  - Know <u>needs</u> before deciding what to lose

Relative Store Fragments

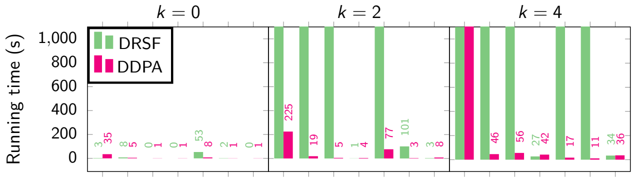
- Partial sets of bindings occurring simultaneously
- Merge discards dissonant store fragments
- Precision similar to non-store-widening analyses
  - Worst-case complexity, too ($O(2^n)$ vs DDPA's $O(n^k)$)
- <u>Tunable!</u>
  - Merges described in algebraic lookup function
  - Set complex policies for precision loss
  - Know <u>needs</u> before deciding what to lose
- Versatile
  - Context-sensitivity
  - Flow-sensitivity
  - Path-sensitivity
  - Must-alias analysis
  - Non-local variable alignment

What's Next?

- Performance!

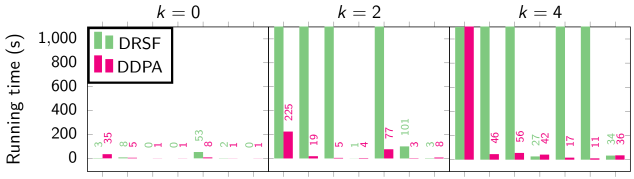
What's Next?

- Performance!



What's Next?

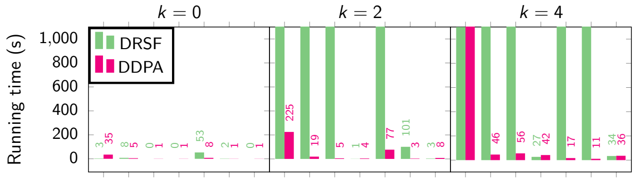
- Performance!



  - Worst-case recursion is slow

What's Next?

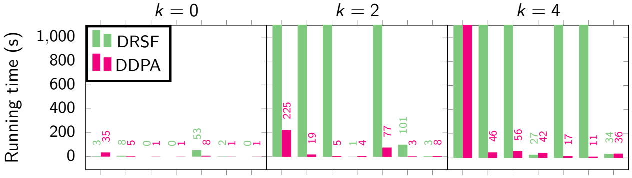
- Performance!



  - Worst-case recursion is slow (in DDPA too)

What's Next?

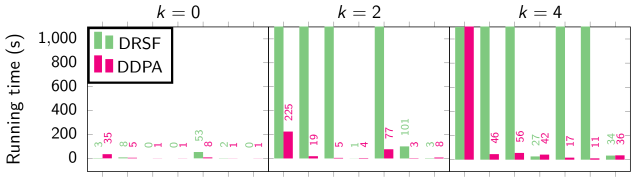
- Performance!



  - Worst-case recursion is slow (in DDPA too)
  - Currently retaining too much on merge

What's Next?

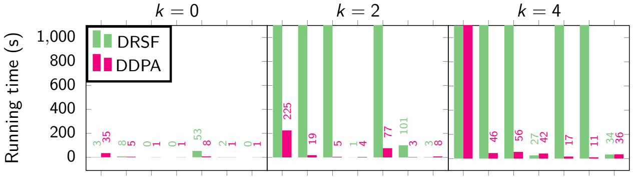
- Performance!



  - Worst-case recursion is slow (in DDPA too)
  - Currently retaining too much on merge
- Extending little store: partial set of bindings

What's Next?

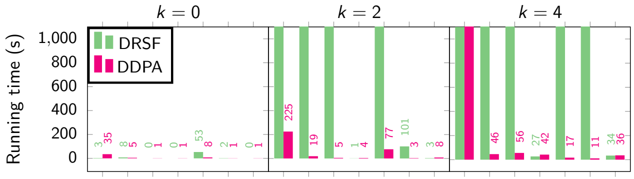
- Performance!



  - Worst-case recursion is slow (in DDPA too)
  - Currently retaining too much on merge
- Extending little store: partial set of bindings/constraints?

What's Next?

- Performance!



  - Worst-case recursion is slow (in DDPA too)
  - Currently retaining too much on merge
- Extending little store: partial set of bindings/constraints?/facts?

Questions?