Demand-Driven Relative Store Fragments for Singleton Abstraction

Little Store’s Big Journey

Leandro Facchinetti\textsuperscript{1} \hspace{1cm} Zachary Palmer\textsuperscript{2} \hspace{1cm} Scott F. Smith\textsuperscript{1}

\textsuperscript{1}The Johns Hopkins University
\textsuperscript{2}Swarthmore College

September 1st, 2017
### Some Program Analyses

<table>
<thead>
<tr>
<th>First Order</th>
<th>Push Forward</th>
<th>Reverse Lookup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Classic abs. interp. data flow analysis</td>
<td>CFL-reachability reverse data flow analysis</td>
<td></td>
</tr>
<tr>
<td>kCFA</td>
<td>CFA2</td>
<td></td>
</tr>
<tr>
<td>PDCFA</td>
<td>ΓCFA</td>
<td></td>
</tr>
</tbody>
</table>
### Some Program Analyses

<table>
<thead>
<tr>
<th>First Order</th>
<th>Higher Order</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>push forward</strong></td>
<td></td>
</tr>
<tr>
<td>classic abs. interp.</td>
<td></td>
</tr>
<tr>
<td>data flow analysis</td>
<td></td>
</tr>
<tr>
<td><strong>reverse lookup</strong></td>
<td></td>
</tr>
<tr>
<td>CFL-reachability</td>
<td></td>
</tr>
<tr>
<td>reverse data flow analysis</td>
<td></td>
</tr>
<tr>
<td>kCFA</td>
<td>CFA2</td>
</tr>
<tr>
<td>PDCFA</td>
<td>ΓCFA</td>
</tr>
</tbody>
</table>
Some Program Analyses

**First Order**
for (int i=0; i<n; i++)

**Higher Order**
fold (λa e -> ...)

CFG → Data Flow

CFG → Data Flow
## Some Program Analyses

<table>
<thead>
<tr>
<th></th>
<th>push forward</th>
<th>reverse lookup</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>first order</strong></td>
<td>classic abs. interp. data flow analysis</td>
<td>CFL-reachability reverse data flow analysis</td>
</tr>
<tr>
<td><strong>higher order</strong></td>
<td>kCFA</td>
<td>CFA2</td>
</tr>
<tr>
<td></td>
<td>PDCFA</td>
<td>ΓCFA</td>
</tr>
</tbody>
</table>
# Some Program Analyses

<table>
<thead>
<tr>
<th></th>
<th>push forward</th>
<th>reverse lookup</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>first order</strong></td>
<td>classic abs. interp.</td>
<td>CFL-reachability</td>
</tr>
<tr>
<td></td>
<td>data flow analysis</td>
<td>reverse data flow analysis</td>
</tr>
<tr>
<td><strong>higher order</strong></td>
<td>kCFA</td>
<td>CFA2</td>
</tr>
<tr>
<td></td>
<td>PDCFA</td>
<td>ΓCFA</td>
</tr>
</tbody>
</table>
Some Program Analyses

<table>
<thead>
<tr>
<th>First Order</th>
<th>Push Forward</th>
<th>Reverse Lookup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Classic abs. interp.</td>
<td>data flow analysis</td>
<td>CFL-reachability</td>
</tr>
<tr>
<td>reverse data flow analysis</td>
<td></td>
<td>reverse data flow analysis</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Higher Order</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>kCFA</td>
<td>CFA2</td>
<td>DDPA</td>
</tr>
<tr>
<td>PDCFA</td>
<td>ΓCFA</td>
<td></td>
</tr>
</tbody>
</table>
# Some Program Analyses

<table>
<thead>
<tr>
<th></th>
<th>push forward</th>
<th>reverse lookup</th>
</tr>
</thead>
<tbody>
<tr>
<td>first order</td>
<td>classic abs. interp. data flow analysis</td>
<td>CFL-reachability reverse data flow analysis</td>
</tr>
</tbody>
</table>
| higher order | kCFA CFA2                              | DDPA
|          | PDCFA ΓCFA                             | DRSF

4/20
## Some Program Analyses

<table>
<thead>
<tr>
<th>first order</th>
<th>push forward</th>
<th>reverse lookup</th>
</tr>
</thead>
<tbody>
<tr>
<td>classic abs. interp.</td>
<td>data flow analysis</td>
<td>CFL-reachability</td>
</tr>
<tr>
<td>reverse data flow analysis</td>
<td>reverse data flow analysis</td>
<td></td>
</tr>
<tr>
<td>higher order</td>
<td></td>
<td></td>
</tr>
<tr>
<td>kCFA</td>
<td>CFA2</td>
<td>DDPA</td>
</tr>
<tr>
<td>PDCFA</td>
<td>ΓCFA</td>
<td>DRSF</td>
</tr>
<tr>
<td>POLYFLOW\textsubscript{CFL} (weak non-locals)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>DDPA</td>
<td>DRSF</td>
</tr>
<tr>
<td>------------------</td>
<td>--------------------------</td>
<td>--------------------------</td>
</tr>
<tr>
<td>Context-sensitive</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Flow-sensitive</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Path-sensitive</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Must-alias</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Non-local variables</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Characteristics</td>
<td>DDPA</td>
<td>DRSF</td>
</tr>
<tr>
<td>--------------------</td>
<td>-------</td>
<td>-------</td>
</tr>
<tr>
<td>Context-sensitive</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Flow-sensitive</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Path-sensitive</td>
<td>~</td>
<td></td>
</tr>
<tr>
<td>Must-alias</td>
<td>~</td>
<td></td>
</tr>
<tr>
<td>Non-local variables</td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>
## Demand-Driven Higher-Order Program Analyses

<table>
<thead>
<tr>
<th>Feature</th>
<th>DDPA</th>
<th>DRSF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Context-sensitive</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Flow-sensitive</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Path-sensitive</td>
<td>❍</td>
<td>✔</td>
</tr>
<tr>
<td>Must-alias</td>
<td>❍</td>
<td>✔</td>
</tr>
<tr>
<td>Non-local variables</td>
<td>✔</td>
<td>✔</td>
</tr>
</tbody>
</table>
## Demand-Driven Higher-Order Program Analyses

<table>
<thead>
<tr>
<th>Feature</th>
<th>DDPA</th>
<th>DRSF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Context-sensitive</td>
<td>✓ Contours</td>
<td>✓</td>
</tr>
<tr>
<td>Flow-sensitive</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Path-sensitive</td>
<td>~</td>
<td>✓</td>
</tr>
<tr>
<td>Must-alias</td>
<td>~</td>
<td>✓</td>
</tr>
<tr>
<td>Non-local variables</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
## Demand-Driven Higher-Order Program Analyses

<table>
<thead>
<tr>
<th>Feature</th>
<th>DDPA</th>
<th>DRSF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Context-sensitive</td>
<td>✔️ Contours</td>
<td>✔️</td>
</tr>
<tr>
<td>Flow-sensitive</td>
<td>✔️ Natural</td>
<td>✔️</td>
</tr>
<tr>
<td>Path-sensitive</td>
<td>~</td>
<td>✔️</td>
</tr>
<tr>
<td>Must-alias</td>
<td>~</td>
<td>✔️</td>
</tr>
<tr>
<td>Non-local variables</td>
<td>✔️</td>
<td>✔️</td>
</tr>
</tbody>
</table>
## Demand-Driven Higher-Order Program Analyses

<table>
<thead>
<tr>
<th></th>
<th>DDPA</th>
<th>DRSF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Context-sensitive</td>
<td>✓ Contours</td>
<td>✓</td>
</tr>
<tr>
<td>Flow-sensitive</td>
<td>✓ Natural</td>
<td>✓</td>
</tr>
<tr>
<td>Path-sensitive</td>
<td>~ Filters</td>
<td>✓</td>
</tr>
<tr>
<td>Must-alias</td>
<td>~</td>
<td>✓</td>
</tr>
<tr>
<td>Non-local variables</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
**Demand-Driven Higher-Order Program Analyses**

<table>
<thead>
<tr>
<th></th>
<th>DDPA</th>
<th>DRSF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Context-sensitive</td>
<td>✓ Contours</td>
<td>✓</td>
</tr>
<tr>
<td>Flow-sensitive</td>
<td>✓ Natural</td>
<td>✓</td>
</tr>
<tr>
<td>Path-sensitive</td>
<td>~ Filters</td>
<td>✓</td>
</tr>
<tr>
<td>Must-alias</td>
<td>~ A Mess</td>
<td>✓</td>
</tr>
<tr>
<td>Non-local variables</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
# Demand-Driven Higher-Order Program Analyses

<table>
<thead>
<tr>
<th>Feature</th>
<th>DDPA</th>
<th>DRSF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Context-sensitive</td>
<td>✓ Contours</td>
<td>✓</td>
</tr>
<tr>
<td>Flow-sensitive</td>
<td>✓ Natural</td>
<td>✓</td>
</tr>
<tr>
<td>Path-sensitive</td>
<td>~ Filters</td>
<td>✓</td>
</tr>
<tr>
<td>Must-alias</td>
<td>~ A Mess</td>
<td>✓</td>
</tr>
<tr>
<td>Non-local variables</td>
<td>✓ Lookup</td>
<td>✓</td>
</tr>
</tbody>
</table>
## Demand-Driven Higher-Order Program Analyses

<table>
<thead>
<tr>
<th>Feature</th>
<th>DDPA</th>
<th>DRSF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Context-sensitive</td>
<td>✓ Contours</td>
<td>✓ Little Stores</td>
</tr>
<tr>
<td>Flow-sensitive</td>
<td>✓ Natural</td>
<td>✓ Little Stores</td>
</tr>
<tr>
<td>Path-sensitive</td>
<td>~ Filters</td>
<td>✓ Little Stores</td>
</tr>
<tr>
<td>Must-alias</td>
<td>~ A Mess</td>
<td>✓ Little Stores</td>
</tr>
<tr>
<td>Non-local variables</td>
<td>✓ Lookup</td>
<td>✓ Lookup</td>
</tr>
</tbody>
</table>
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
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 $f \ 4$

```ocaml
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 \( f \ 4 \)

```ocaml
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

\[ f \]

\[ g \]

\[ v \]

\[ z \]

\[ z \]
DDPA by Example

Expand function call f 4

```ocaml
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
DDPA by Example
Expand function call \( f 4 \)

```ocaml
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
```

“Look Pup”
DDPA by Example
Expand function call \( f \ 4 \)

\[
\begin{align*}
\text{let } f &= \text{fun } p \rightarrow \\
& \quad \text{let } x = p \ \text{in} \\
& \quad \text{fun } y \rightarrow x + y \\
\text{in} \\
\text{let } g &= f \ 4 \ \text{in} \\
\text{let } v &= 1 \ \text{in} \\
\text{let } z &= g \ v \ \text{in} \ z
\end{align*}
\]
DDPA by Example

Expand function call \( f \ 4 \)

```ml
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 $f \ 4$

```ml
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
```

```
  f
```

Lookup

```
  f

  g

  v

  z

  z
```
DDPA by Example

Expand function call \( f \ 4 \)

```ocaml
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
DDPA by Example

Wire in function call f 4

```ocaml
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
```

---

Diagram:
DDPA by Example

Wire in function call \( f \ 4 \)

```ocaml
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
```

Diagram:
let \( f = \text{fun} \ p \rightarrow \\
let x = p \text{ in} \\
\text{fun} \ y \rightarrow x + y \)
in
let \( g = f \ 4 \ \text{in} \\
let v = 1 \ \text{in} \\
let z = g \ v \ \text{in} \ z
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
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 \( g \ v \)

```ml
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
```

Diagram:

- \( f \) is the function.
- \( x \) is the argument passed to \( f \).
- \( g \) is the function resulting from \( f \) with \( p = 4 \).
- \( v \) is the argument passed to \( g \).
- \( z \) is the result of the function call \( g \ v \).
DDPA by Example
Expand function call $g \; v$

```
let $f = \text{fun } p \rightarrow$
  let $x = p \; \text{in}$
  fun $y \rightarrow x + y$
in
let $g = f \; 4 \; \text{in}$
let $v = 1 \; \text{in}$
let $z = g \; v \; \text{in } z$
```
Expand function call \( g \ v \)

```ml
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
```

**Diagram:**
- \( f \)
- \( x \)
- \( \text{fun } y \)
- \( f \)
- \( p=4 \)
- \( g=f \)
- \( v \)
- \( z \)

**Lookup:**
- \( g \)
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 \( g \ v \)

```ml
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
```

![Diagram of function calls and variable assignments](image)
let \( f = \) fun \( p \rightarrow \)
  let \( x = p \) in
  fun \( y \rightarrow x + y \)
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: \( y \)

```ml
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
```

![Diagram showing the parameter lookup process](image-url)
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: \( x \)

```ocaml
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
```

![Diagram](https://via.placeholder.com/150)
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

```ml
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
```

Diagram:

- `f` to `x` to `fun y` to `f`
- `g` to `x + y` to `g`
- `p = 4`
- `g = fun y`
- `y = 1`
- `z = x + y`
- `v`
- `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

```plaintext
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
```

Diagram:
- `f` points to `x` which points to `fun y` which points to `f`.
- `g` points to `x + y` which points to `g`.
- `p = 4` points to `x` points to `fun y`.
- `g = fun y` points to `y = 1` which points to `z = x + y`.
- `z` points to `z`.

Lookup Stack:
- `g` is at the top.
- `x` is below `g`.
- `g` is below `f`.
- `y = 1` is below `g`.
- `z = x + y` is below `g`.
- `z` is below `z`.
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: \( x \)

```ml
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 Diagram](image)
DDPA by Example

Non-local lookup: \( x \)

```ml
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:
- **g**
- **x**

Diagram:
- **f**
- **x**
- **fun y**
- **f**
- **g**
- **x + y**
- **g**
- **g**
- **x**
- **fun y**
- **g**
- **y = 1**
- **z = x + y**
- **z**
- **z**
- **z**
- **g**
- **v**
- **z**
DDPA by Example

Non-local lookup: \( x \)

```ml
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

```
g
  x
```

7/20
DDPA by Example
Non-local lookup: \( x \)

```ml
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:
- **f**
- **x**
- **fun y**
- **f**
- **g**
- **x + y**
- **g**
- **p = 4**
- **g = fun y**
- **y = 1**
- **z = x + y**
- **z**
DDPA by Example

Non-local lookup: x

```ml
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

- `x`
DDPA by Example

Non-local lookup: \( x \)

```ocaml
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:

- \( x \)
- \( f \)
- \( g \)
- \( v \)
- \( z \)
- \( p = 4 \)
- \( g = \text{fun } y \)
- \( y = 1 \)
- \( z = x + y \)
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
DDPA by Example

Non-local lookup: x

```ml
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

- p
- f
- g
- v
- z
- x
- y
- x + y
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

```ml
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**
- `p = 4` (p = 4)
- `g = fun y` (g = fun y)
- `y = 1` (y = 1)
- `z = x + y` (z = x + y)
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

$p = 4$

$g = \text{fun } y$

$y = 1$

$z = x + y$

"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 Abstract Stores

\[
\begin{align*}
\text{p} &= 4 \\
g &= \text{fun } y \\
y &= 1 \\
z &= x + y
\end{align*}
\]
$\{ p \mapsto 4 \}$

```
\begin{align*}
  f & \quad x \quad \text{fun} \quad y \quad f \\
  p &= 4 \\
  g &= \text{fun} \quad y \\
  y &= 1 \\
  z &= x + y
\end{align*}
```
DDPA and Abstract Stores

\{p \mapsto 4\} \quad \{p \mapsto 4\} \quad \{x \mapsto 4\}

\begin{align*}
&f \xrightarrow{p=4} x \xrightarrow{\text{fun } y} f \\
&g \xrightarrow{g=\text{fun } y} x + y \xrightarrow{y=1} z \xrightarrow{z=x+y} g
\end{align*}

"Big stores": complete sets of bindings

DDPA: reconstruct big stores with lookups

Lookups from a point are independent

Similar to per-point store widening
“Big stores”: complete sets of bindings
DDPA and Abstract Stores

\[
\begin{align*}
\{ & p \mapsto 4 \\
& x \mapsto 4 
\end{align*}
\]

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

\[
\begin{align*}
  \{ p \mapsto 4 \} \\
  \{ x \mapsto 4 \} 
\end{align*}
\]

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

\[
\begin{align*}
  p &= 4 \\
  g &\triangleq \text{fun } y \\
  y &= 1 \\
  z &= x + y 
\end{align*}
\]
DDPA and Abstract Stores

\[
\begin{align*}
\{ & p \mapsto 4 \\
& x \mapsto 4 
\}
\end{align*}
\]

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

Lookups from a point are independent

\[ \text{plan} + \ldots + \text{plan} = \{ \]
DDPA and Abstract Stores

\[
\begin{cases}
p \mapsto 4 \\
x \mapsto 4
\end{cases}
\]

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

Lookups from a point are independent

Similar to per-point store widening
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
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
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
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

Possible values of \( x + y \)?

```
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
```
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 \( x + y \)
\[
\begin{array}{c}
\text{\( x \in \{4, "s"\} \)}
\end{array}
\]

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>x + y</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>&quot;t&quot;</td>
<td>7</td>
</tr>
<tr>
<td>&quot;s&quot;</td>
<td>1</td>
<td>&quot;s&quot; + 1 = 5</td>
</tr>
<tr>
<td>&quot;s&quot;</td>
<td>&quot;t&quot;</td>
<td>&quot;s&quot; + &quot;t&quot; = 7</td>
</tr>
</tbody>
</table>
DDPA and Variable (Mis-)Alignment

1. let b = coin_flip () in
2. let f = fun p ->
   3. let x = p in
   4. fun y -> x + y
   in
5. let g = f (b?4:"s") in
6. let v = (b?1:"t") in
7. let z = g v in z

Possible values of $x + y$?
- $x \in \{4, "s"\}$
- $y \in \{1, "t"\}$

Possible values of $x + y$?
- $x + y$ = \begin{cases} 
4 + 1 & \text{if } x = 4, y = 1 \\
4 + "t" & \text{if } x = 4, y = "t" \\
"s" + 1 & \text{if } x = "s", y = 1 \\
"s" + "t" & \text{if } x = "s", y = "t" 
\end{cases}
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 $x + y$?

- $x \in \{4, "s"\}$
- $y \in \{1, "t"\}$

$x + y = \begin{cases} 
4 + 1 \\
4 + "t" \\
"s" + 1 \\
"s" + "t"
\end{cases}$
DDPA and Variable (Mis-)Alignment

1. `let b = coin_flip () in`  
2. `let f = fun p ->`  
3. `let x = p in`  
4. `fun y -> x + y`  
5. `in`  
6. `let g = f (b?4:"s") in`  
7. `let v = (b?1:"t") in`  
8. `let z = g v in z`  

Possible values of \(x + y\)?

- \(x \in \{4, "s"\}\)
- \(y \in \{1, "t"\}\)

\[
x + y = \begin{cases} 
4 + 1 \\
4 + "t" \\
"s" + 1 \\
"s" + "t"
\end{cases}
\]
DRSF
DRSF

\[ U \{ \hat{x} \} \cup \Delta = \sum \{ \hat{v} \} \]

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

\[ \delta ::= I x \mid J x \]

Little stores are incomplete. Relative (vs. DDPA's absolute)
DRSF

\[ U \{ \text{sad face} \} = \sum \{ \text{happy dog} \} \]
$U \{ \{ \text{sad} \} \} = \sum \{ \text{dog} \} \neq \{ \{ \text{happy} \} \}$

Relative (vs. DDPA's absolute)
\[ \bigcup \{ \hat{x} \} = \sum \Delta \neq \{ \} \]

\[ = \{ \hat{x} \odot \Delta \mapsto \hat{v}, \ldots \} \]
DRSF

\[ U \{ \hat{x} \} = \sum \delta \neq \{ \hat{v}, \ldots \} \]

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

\[ \bigcup \{ \hat{x} @ \Delta \mapsto \hat{v}, \ldots \} = \sum \{ \hat{x} @ \Delta \mapsto \hat{v}, \ldots \} \neq \{ \hat{x} @ \Delta \mapsto \hat{v}, \ldots \} \]

\[ \Delta = [\delta, \ldots] \quad \delta ::= \sum x | \Delta x \]

Little stores are incomplete relative (vs. DDPA's absolute)

PDCFA [JFP #24 (2014)] (stack deltas, reachability)

CFA [POPL 06] (abstract frame strings)
\[
\bigcup \{\hat{x}\} = \sum \{\hat{\nu} \oplus \Delta \mapsto \hat{\nu}, \ldots\}
\]

\[
\Delta = [\delta, \ldots] \quad \delta ::= \mathcal{D}x | \mathcal{D}x
\]

- \(\Delta\text{CFA} [\text{POPL 06}]\) (abstract frame strings)
- \(\text{PDCFA} [\text{JFP #24 (2014)}]\) (stack deltas, reachability)
ΔCFA [POPL 06] (abstract frame strings)
PDCFA [JFP #24 (2014)] (stack deltas, reachability)
Little stores are incomplete
\[ \mathbf{U} \{ \mathbf{\hat{x}} \} = \sum \{ \mathbf{\hat{x}} \} \neq \{ \mathbf{\hat{x}} \} \]

\[ \{ \mathbf{\hat{x}} \} = \{ \mathbf{\hat{x}} \circ \Delta \mapsto \mathbf{\hat{v}}, \ldots \} \]

\[ \Delta = [\delta, \ldots] \quad \delta ::= \Delta \mathbf{CFA} \quad \text{[POPL 06]} \quad \text{(abstract frame strings)} \]

\[ \Delta \mathbf{CFA} \quad \text{[JFP \#24 (2014)]} \quad \text{(stack deltas, reachability)} \]

- Little stores are **incomplete**
- **Relative** (vs. DDPA’s **absolute**)

12/20
\[ U \{ \hat{x} \} = \sum \{ \hat{\delta} \} \]

\[ \{ \hat{x} \} = \{ \hat{x} \circ \Delta \mapsto \hat{\nu}, \ldots \} \]

\[ \Delta = [\delta, \ldots ] \quad \delta ::= \bigcup x \bigcup \bigcup x \]

- DCAF [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

<table>
<thead>
<tr>
<th></th>
<th>DDPA</th>
<th>DRSF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Context-sensitive</td>
<td>✓ Contours</td>
<td>✓ Little Stores</td>
</tr>
<tr>
<td>Flow-sensitive</td>
<td>✓ Natural</td>
<td>✓ Little Stores</td>
</tr>
<tr>
<td>Path-sensitive</td>
<td>~ Filters</td>
<td>✓ Little Stores</td>
</tr>
<tr>
<td>Must-alias</td>
<td>~ A Mess</td>
<td>✓ Little Stores</td>
</tr>
<tr>
<td>Non-local variables</td>
<td>✓ Lookup</td>
<td>✓ Lookup</td>
</tr>
</tbody>
</table>
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
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
let \( b = \text{coin\_flip} () \) in
let \( f = \text{fun} \ p \rightarrow \) 
let \( x = p \) in 
fun \( y \) \( \rightarrow \) \( x + y \) 
in
let \( g = f (b?4:"s") \) in
let \( v = (b?1:"t") \) in
let \( z = g \ v \) in \( z \)
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
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
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
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
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
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
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
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

<table>
<thead>
<tr>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>(z</td>
</tr>
</tbody>
</table>

Lookup Stack
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
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}
{y@[] ↦ 1}

Lookup Stack
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
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
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] \mapsto true}
{y@[] \mapsto 1}

Lookup Stack
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
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, \\
  b@[\[] z \mapsto true \}\}

Lookup Stack

14/20
DRSF and Variable Alignment

1. let b = coin_flip () in
2. let f = fun p ->
3.   let x = p in
4.   fun y -> x + y
5. in
6. let g = f (b?4:"s") in
7. let v = (b?1:"t") in
8. let z = g v in z

```

f
  x
  \fun y
  \f

\{ y@[] \mapsto 1, \\ b@[\_z] \mapsto \text{true} \}

```

Lookup Stack

{ y@[] \mapsto 1, \\ b@[\_z] \mapsto \text{true} \}

```

```

b
f
\g
v
z
z

```
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 "t",
  b@[\&z] \mapsto false
\}
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
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
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
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
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
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
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

```
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
```
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
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
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

<table>
<thead>
<tr>
<th>b@[g]</th>
<th>true</th>
</tr>
</thead>
<tbody>
<tr>
<td>p@[]</td>
<td>4</td>
</tr>
</tbody>
</table>

\[\Downarrow g\]

\[\Downarrow z\]
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
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

\{ p@[\ D g] \mapsto 4, b@[\] \mapsto \text{true} \}
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
```
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
Merging Relative Store Fragments

\[
\begin{align*}
\{ & x[@] \mapsto 4, \\
& b[@] \mapsto \text{true} \} \oplus \{ & y[@] \mapsto 1, \\
& b[@] \mapsto \text{true} \} \\
= & \\\n\{ & x[@] \mapsto \text{"s"}, \\
& b[@] \mapsto \text{false} \} \oplus \{ & y[@] \mapsto \text{"t"}, \\
& b[@] \mapsto \text{false} \}
\end{align*}
\]
Merging Relative Store Fragments

\[ \{ x@[] \mapsto 4, \\
    b@[\downarrow z] \mapsto \text{true} \} \oplus \{ y@[] \mapsto 1, \\
    b@[\downarrow z] \mapsto \text{true} \} = \{ x@[] \mapsto 4, \\
    y@[] \mapsto 1, \\
    b@[\downarrow z] \mapsto \text{true} \} \]
Merging Relative Store Fragments

\[
\{ 
  x@[] \mapsto 4, \\
  b@[\downarrow z] \mapsto \text{true}
\} \oplus \{ 
  y@[] \mapsto 1, \\
  b@[\downarrow z] \mapsto \text{true}
\} = \{ 
  x@[] \mapsto 4, \\
  y@[] \mapsto 1, \\
  b@[\downarrow z] \mapsto \text{true}
\}
\]

\[
\{ 
  x@[] \mapsto "s", \\
  b@[\downarrow z] \mapsto \text{false}
\} \oplus \{ 
  y@[] \mapsto "t", \\
  b@[\downarrow z] \mapsto \text{false}
\} = \{ 
  x@[] \mapsto "s", \\
  y@[] \mapsto "t", \\
  b@[\downarrow z] \mapsto \text{false}
\} \oplus \{ 
  y@[] \mapsto "t", \\
  b@[\downarrow z] \mapsto \text{false}
\} = \{ 
  x@[] \mapsto "s", \\
  y@[] \mapsto "t", \\
  b@[\downarrow z] \mapsto \text{false}
\}
\]
Merging Relative Store Fragments

\[
\begin{align*}
\{ & x@[] \mapsto 4, \\
& b@[\downarrow z] \mapsto \text{true} \} 
\oplus \{ & y@[] \mapsto 1, \\
& b@[\downarrow z] \mapsto \text{true} \} = 
\{ & x@[] \mapsto 4, \\
& y@[] \mapsto 1, \\
& b@[\downarrow z] \mapsto \text{true} \}
\end{align*}
\]

\[
\begin{align*}
\{ & x@[] \mapsto "s", \\
& b@[\downarrow z] \mapsto \text{false} \} 
\oplus \{ & y@[] \mapsto "t", \\
& b@[\downarrow z] \mapsto \text{false} \} = 
\{ & x@[] \mapsto "s", \\
& y@[] \mapsto "t", \\
& b@[\downarrow z] \mapsto \text{false} \}
\end{align*}
\]
Merging Relative Store Fragments

\[
\begin{align*}
\{ x@[] \mapsto 4, \\
b@[@z] \mapsto \text{true} \} & \oplus \{ y@[] \mapsto 1, \\
b@[@z] \mapsto \text{true} \} = \{ x@[] \mapsto 4, \\
y@[] \mapsto 1, \\
b@[@z] \mapsto \text{true} \} \\
\{ x@[] \mapsto \text{"s"}, \\
b@[@z] \mapsto \text{false} \} & \oplus \{ y@[] \mapsto \text{"t"}, \\
b@[@z] \mapsto \text{false} \} = \{ x@[] \mapsto \text{"s"}, \\
y@[] \mapsto \text{"t"}, \\
b@[@z] \mapsto \text{false} \} \\
\{ x@[] \mapsto 4, \\
b@[@z] \mapsto \text{true} \} & \oplus \{ y@[] \mapsto \text{"t"}, \\
b@[@z] \mapsto \text{false} \} = \\
\{ x@[] \mapsto \text{"s"}, \\
b@[@z] \mapsto \text{false} \} & \oplus \{ y@[] \mapsto 1, \\
b@[@z] \mapsto \text{true} \} =
\end{align*}
\]
Merging Relative Store Fragments

\[
\begin{align*}
\{ x@[] \mapsto 4, \\
b@[\llbracket z \rrbracket] \mapsto \text{true} \} & \oplus \{ y@[] \mapsto 1, \\
b@[\llbracket z \rrbracket] \mapsto \text{true} \} = \{ x@[] \mapsto 4, \\
y@[] \mapsto 1, \\
b@[\llbracket z \rrbracket] \mapsto \text{true} \} \\
\{ x@[] \mapsto "s", \\
b@[\llbracket z \rrbracket] \mapsto \text{false} \} & \oplus \{ y@[] \mapsto "t", \\
b@[\llbracket z \rrbracket] \mapsto \text{false} \} = \{ x@[] \mapsto "s", \\
y@[] \mapsto "t", \\
b@[\llbracket z \rrbracket] \mapsto \text{false} \} \\
\{ x@[] \mapsto 4, \\
b@[\llbracket z \rrbracket] \mapsto \text{true} \} & \oplus \{ y@[] \mapsto "t", \\
b@[\llbracket z \rrbracket] \mapsto \text{false} \} = \times \\
\{ x@[] \mapsto "s", \\
b@[\llbracket z \rrbracket] \mapsto \text{false} \} & \oplus \{ y@[] \mapsto 1, \\
b@[\llbracket z \rrbracket] \mapsto \text{true} \} = \times
\end{align*}
\]
Merging Relative Store Fragments

\[
\{ x[@] \mapsto 4, \\
b[@[z]] \mapsto \text{true} \} \oplus \{ y[@] \mapsto 1, \\
b[@[z]] \mapsto \text{true} \} = \{ x[@] \mapsto 4, \\
y[@] \mapsto 1, \\
b[@[z]] \mapsto \text{true} \}
\]

\[
\{ x[@] \mapsto "s", \\
b[@[z]] \mapsto \text{false} \} \oplus \{ y[@] \mapsto "t", \\
b[@[z]] \mapsto \text{false} \} = \{ x[@] \mapsto "s", \\
y[@] \mapsto "t", \\
b[@[z]] \mapsto \text{false} \}
\]

\[
\{ x[@] \mapsto 4, \\
b[@[z]] \mapsto \text{true} \} \oplus \{ y[@] \mapsto "t", \\
b[@[z]] \mapsto \text{false} \} = \text{X}
\]

\[
\{ x[@] \mapsto "s", \\
b[@[z]] \mapsto \text{false} \} \oplus \{ y[@] \mapsto 1, \\
b[@[z]] \mapsto \text{true} \} = \text{X}
\]
Polymorphism via $\Delta$

```ocaml
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 $\Delta$

1. let $f = \text{fun } x \to x$ in
2. let $a = 4$ in
3. let $b = f\ a$ in
4. let $c = "s"$ in
5. let $d = f\ c$ in
6. 0
Polymorphism via $\Delta$

```
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 $\Delta$

```ocaml
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 $\Delta$

1. let $f = \text{fun} \ x \rightarrow \ x$ in
2. let $a = 4$ in
3. let $b = f \ a$ in
4. let $c = "s"$ in
5. let $d = f \ c$ in
6. 0
Polymorphism via $\Delta$

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

Lookup Stack
Polymorphism via $\Delta$

```plaintext
1 let f = fun x -> x in
2 let a = 4 in
3 let b = f a in
4 let c = "s" in
5 let d = f c in
6 0
```

Lookup Stack

```
0
```

```
f a b c d
```

```
f f x
```

```
x=a \& b
```

```
b=x \& b
```

```
x=c \& d
```

```
d=x \& d
```
Polymorphism via $\triangle$

1. `let f = fun x -> x in`
2. `let a = 4 in`
3. `let b = f a in`
4. `let c = "s" in`
5. `let d = f c in`
6. `0`
Polymorphism via $\Delta$

1. let $f = \text{fun} \ x \rightarrow x \ \text{in}$
2. let $a = 4 \ \text{in}$
3. let $b = f \ a \ \text{in}$
4. let $c = "s" \ \text{in}$
5. let $d = f \ c \ \text{in}$
6. 0

```
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 $\Delta$

```ocaml
1 let f = fun x -> x in
2 let a = 4 in
3 let b = f a in
4 let c = "s" in
5 let d = f c in
6 0
```

![Lookup Stack](image)
Polymorphism via \( \Delta \)

```plaintext
1 let f = fun x -> x in
2 let a = 4 in
3 let b = f a in
4 let c = "s" in
5 let d = f c in
6 0
```

```
<table>
<thead>
<tr>
<th>c@[]</th>
<th>&quot;s&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>⊢d</td>
<td></td>
</tr>
<tr>
<td>⊢d</td>
<td></td>
</tr>
</tbody>
</table>
```

Lookup Stack
Polymorphism via $\Delta$

1. `let f = fun x -> x in`
2. `let a = 4 in`
3. `let b = f a in`
4. `let c = "s" in`
5. `let d = f c in`
6. `0`
Polymorphism via $\Delta$

```
1 let f = fun x -> x in
2 let a = 4 in
3 let b = f a in
4 let c = "s" in
5 let d = f c in
6 0
```
Polymorphism via \( \Delta \)

1. `let f = fun x -> x in`
2. `let a = 4 in`
3. `let b = f a in`
4. `let c = "s" in`
5. `let d = f c in`
6. 0

```
0 f a b c d
```

```
I b
b=x \( \Downarrow b \)
J b
```
Polymorphism via $\Delta$

1. `let f = fun x -> x in`
2. `let a = 4 in`
3. `let b = f a in`
4. `let c = "s" in`
5. `let d = f c in`

![Lookup Stack](image)

```
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 $\Delta$

1. let $f = \text{fun } x \rightarrow x$ in
2. let $a = 4$ in
3. let $b = f\ a$ in
4. let $c = "s"$ in
5. let $d = f\ c$ in
6. $0$

```
{c[@[]] \mapsto 4}
{d}

Lookup Stack
```
Polymorphism via $\Delta$

1. `let f = fun x -> x in`
2. `let a = 4 in`
3. `let b = f a in`
4. `let c = "s" in`
5. `let d = f c in`
6. `0`
Polymorphism via \( \Delta \)

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

Lookup Stack
Singleton Abstractions via Full Traces

- Traces $\Delta$ represent relative stack adjustments
Singleton Abstractions via Full Traces

- Traces $\Delta$ represent relative stack adjustments
- Decidability? Finitization
Singleton Abstractions via Full Traces

- Traces $\Delta$ represent relative stack adjustments
- Decidability? Finitization
  - $k$DRSF: DRSF with max $\Delta$ length $k$
Singleton Abstractions via Full Traces

- Traces $\Delta$ represent relative stack adjustments
- Decidability? Finitization
  - $k$DRSF: DRSF with max $\Delta$ length $k$
  - Not the same meaning as $k$ in $k$CFA
Singleton Abstractions via Full Traces

- Traces $\Delta$ represent relative stack adjustments
- Decidability? Finitization
  - $k$DRSF: DRSF with max $\Delta$ length $k$
  - Not the same meaning as $k$ in $k$CFA
  - Longer $\Delta$ truncated to suffix, marked partial
Singleton Abstractions via Full Traces

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

Other models are possible

Full traces imply unique allocation/evaluation

Used to establish shallow singleton abstractions for e.g. must-alias

Partial traces gracefully degrade
Singleton Abstractions via Full Traces

- Traces $\Delta$ represent relative stack adjustments
- Decidability? Finitization
  - $k$DRSF: DRSF with max $\Delta$ length $k$
  - Not the same meaning as $k$ in $k$CFA
  - Longer $\Delta$ truncated to suffix, marked partial
    - $[\Delta a \Delta b] + \Delta c \Rightarrow (\Delta b \Delta c)$
  - Other models are possible
Singleton Abstractions via Full Traces

- Traces $\Delta$ represent relative stack adjustments
- Decidability? Finitization
  - $k$DRSF: DRSF with max $\Delta$ length $k$
  - Not the same meaning as $k$ in $k$CFA
  - Longer $\Delta$ truncated to suffix, marked partial
    - $[\Delta a \Delta b] + \Diamond c \Rightarrow (\Diamond b \Diamond c)$
  - Other models are possible
- Full traces imply unique allocation/evaluation
Singleton Abstractions via Full Traces

- Traces $\Delta$ represent relative stack adjustments
- Decidability? Finitization
  - $k$DRSF: DRSF with max $\Delta$ length $k$
  - Not the same meaning as $k$ in $k$CFA
  - Longer $\Delta$ truncated to suffix, marked \textit{partial}
    - $[\Delta a \Delta b] + \Delta c \Rightarrow (\Delta b \Delta c)$
  - Other models are possible
- Full traces imply \textit{unique} allocation/evaluation
  - Used to establish shallow singleton abstractions for e.g. must-alias
Singleton Abstractions via Full Traces

- Traces $\Delta$ represent relative stack adjustments
- Decidability? Finitization
  - $k$DRSF: DRSF with max $\Delta$ length $k$
  - Not the same meaning as $k$ in $k$CFA
  - Longer $\Delta$ truncated to suffix, marked partial
    - $\lbrack \Delta a \circ b \rbrack + \circ c \Rightarrow (\circ b \circ c)$
  - Other models are possible
- Full traces imply unique 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

Tunable!
Merges described in algebraic lookup function
Set complex policies for precision loss
Know needs before deciding what to lose

Versatile
Context-sensitivity
Flow-sensitivity
Path-sensitivity
Must-alias analysis
Non-local variable alignment
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

Know needs before deciding what to lose

Versatile

- Context-sensitivity
- Flow-sensitivity
- Path-sensitivity
- Must-alias analysis
- Non-local variable alignment
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

Know needs before deciding what to lose

Versatile

- Context-sensitivity
- Flow-sensitivity
- Path-sensitivity
- Must-alias analysis
- Non-local variable alignment
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 \((O(2^n) \text{ vs } \text{DDPA’s } O(n^k))\)
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 $O(n^k)$)
- Tunable!
  - Merges described in algebraic lookup function
  - Set complex policies for precision loss
  - Know needs 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 $O(n^k)$)
- Tunable!
  - Merges described in algebraic lookup function
  - Set complex policies for precision loss
  - Know needs 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!**

---

Running time (s)

- k = 0
  - DRSF: 225
  - DDPA: 19

- k = 2
  - DRSF: 27
  - DDPA: 46

- k = 4
  - DRSF: 36
  - DDPA: 36

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

19/20
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**!

![Graph showing running time for different values of k.]

- 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!**

![Bar chart showing running time comparison between DRSF and DDPA for different values of k.]

- 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?