Static Scoping and Name Resolution for Mobile Processes with Polymorphic Interfaces

Matt B. Pedersen & Matthew Sowders

University of Nevada, Las Vegas

What Have we Done?

- Added mobile processes to ProcessJ
  - With polymorphic interfaces
    - Multiple interfaces to the same process
    - Different set of formal parameters per interface

```
mobile void foo (int x, int y) {
  ...
  while (...) {
    ...
    suspend resume with (int z)
    ...
  }
}
```

Polymorphic Interfaces

```
mobile void foo (int x, int y) {
  ...
  while (...) {
    ...
    suspend resume with (int z)
    ...
  }
}
```

- foo has 2 interfaces
  - `(int x, int y)`
  - `(int z)`
- When `foo` is started `(int x, int y)` is used; subsequently `(int z)` is used.

Why Have we Done That?

```
MOBILE PROC reindelf (CHAN AGENT.INITIALIZE initialize?,
                      SHARED CHAN AGENT.MESSAGE report!,
                      SHARED CHAN INT santa.a!, santa.b!)
                      IMPLEMENTS AGENT
  ... local state declarations
  SEQ
    ... in station compound (initialise local state)
    WHILE TRUE
    SEQ
      ... in station compound
      SUSPEND -- move to gathering place
      ... in the gathering place
      SUSPEND -- move to santa's grotto
      ... in santa's grotto
      SUSPEND -- move to compound
:
```

From: Santa Claus – with mobile reindeer and elves, CPA Fringe presentation 2008

Why Have we Done That?

```
MOBILE PROC reindelf (CHAN AGENT.INITIALIZE initialize?,
                       SHARED CHAN AGENT.MESSAGE report!,
                       SHARED CHAN INT santa.a!, santa.b!)
                       IMPLEMENTS AGENT
  ... lo
  SEQ
    ...                                          local state)
    WHIL
    SEQ
      ... in boulder compound
      SUSPEND -- move to gathering place
      ... in the gathering place
      SUSPEND -- move to santa's grotto
      ... in santa's grotto
      SUSPEND -- move to compound
:
```

These are all the same interface:

```
(CHAN AGENT.INITIALIZE initialize?,
SHARED CHAN AGENT.MESSAGE report!,
SHARED CHAN INT santa.a!, santa.b!)
```

Why Have we Done That?

```
MOBILE PROC reindelf (CHAN AGENT.INITIALIZE initialize?,
                      SHARED CHAN AGENT.MESSAGE report!,
                      SHARED CHAN INT santa.a!, santa.b!)
                      IMPLEMENTS AGENT
  ... local state declarations
  SEQ
    ... in station
    WHILE TRUE
    SEQ
      ... in static
      SUSPEND -- mo
      ... in the ga
      SUSPEND -- mo
      ... in santa's grotto
      SUSPEND -- move to compound
:
```

The `Initialize` channel is only used in

`... local state declaration`

Subsequent re-animations of reindelf must thus provide 'dummy' values for this channel.

Why Have we Done That?

- Channel ends (or other parameters) not used in code following a resumption must still be passed
  - A dummy reading end passed could cause deadlock if ever read.
  - Made up actual parameter values must be passed to satisfy the compiler.

Advertisement: Eric and Peter's Call Channels (Fringe Talk)

Ok, so now what?

- What is the semantics of this?
  - Parameters do not retain their values between invocations.

```
mobile void foo (int x, int y) {
  B1
  while (B2) {
    B3
    suspend resume with (int z)
    B4
  }
  B5
}
```

`x` & `y` can be referenced in B_1 (first invocation); `z` in B_4 (subsequent invocations)

Invocation of Mobiles

```
mobile void foo (int x, int y) {
  B1
  while (B2) {
    B3
    suspend resume with (int z)
    B4
  }
  B5
}
```

- Example of execution of foo:

  foo(4,5); foo(4), foo(5), foo(7), ….
- Only the first time (when foo is started) is the procedure interface used.
- All other resumptions use the suspend/resume interface.

Possible Executions

```
mobile void foo (int x, int y) {
  B1
  while (B2) {
    B3
    suspend resume with (int z)
    B4
  }
  B5
}
```

foo(x,y): B_1, B_2, B_5, done!

foo(x,y): B_1, B_2, B_3, suspend/foo(z): B_4, B_2, B_5

We see that e.g. B_2 (& B_5) can be executed 'with' both `x` & `y` as well as `z`.

Possible Executions

```
mobile void foo (int x, int y) {
   B1
   while (B2) {
      B3
      suspend resume with (int z)
      B4
   }
   B5
}
```

The first time B_2 is executed `x` seems to be ‘a valid parameter’, but the second time it does not; only `z` does.

What should we do?

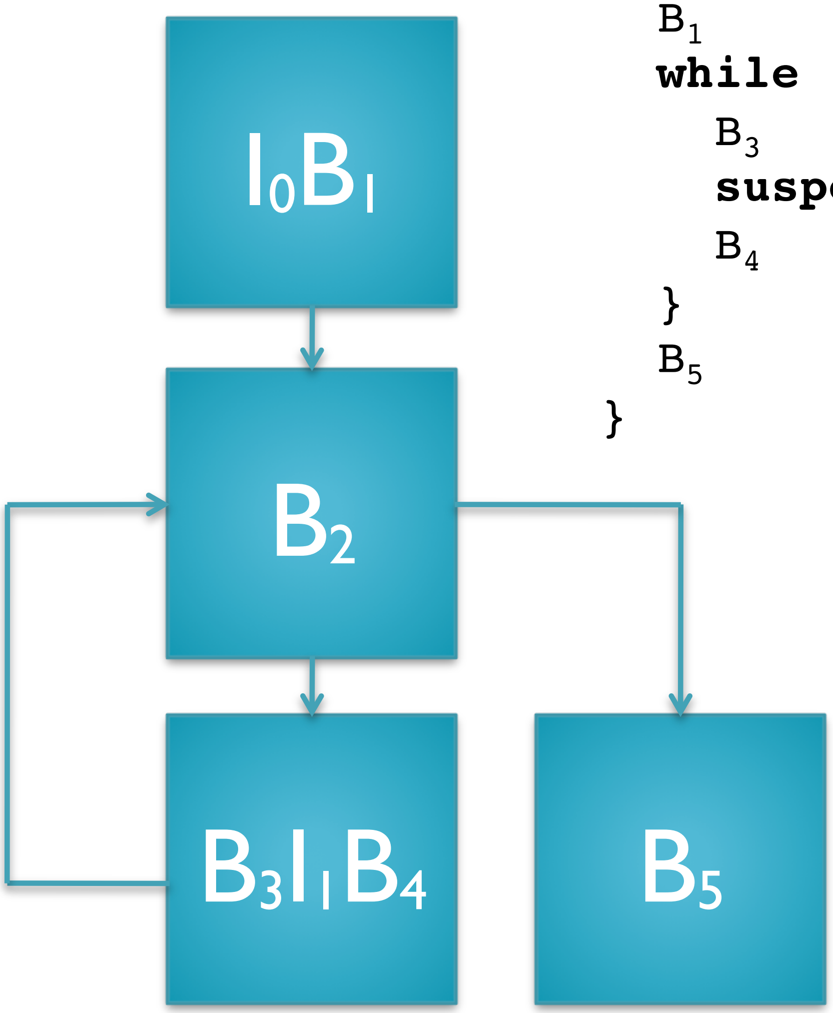
- Determine witch parameters can be referenced in all program blocks
  - Create a control flow graph (CFG) based on the source
  - Massage it a little
  - Perform an analysis using In and Out sets (to be defines shortly)

Remember:
The following concerns parameters only; no local variables are considered (yet)

Remember:
Only the parameters from the most recent invocation may be referenced

Initial Control Flow Graph

```
mobile void foo (int x, int y) {
   B₁
   while (B₂) {
      B₃
      suspend resume with (int z)
      B₄
   }
   B₅
}
```



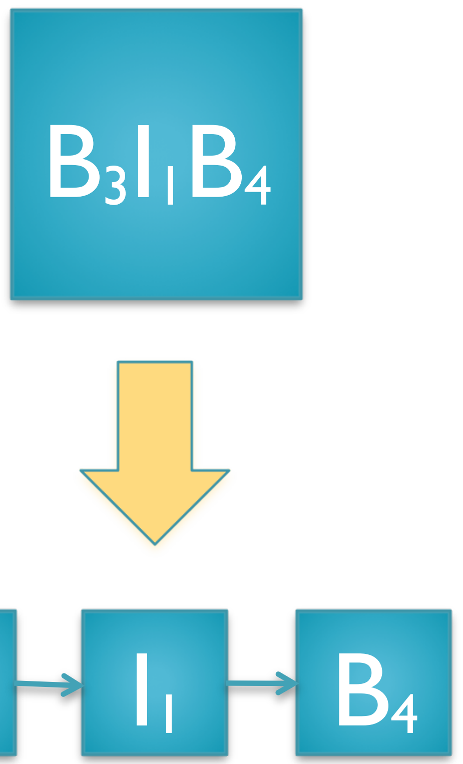
I_0 represents the original interface:

```
foo (int x, int y)
```

I_1 represents the resume interface:

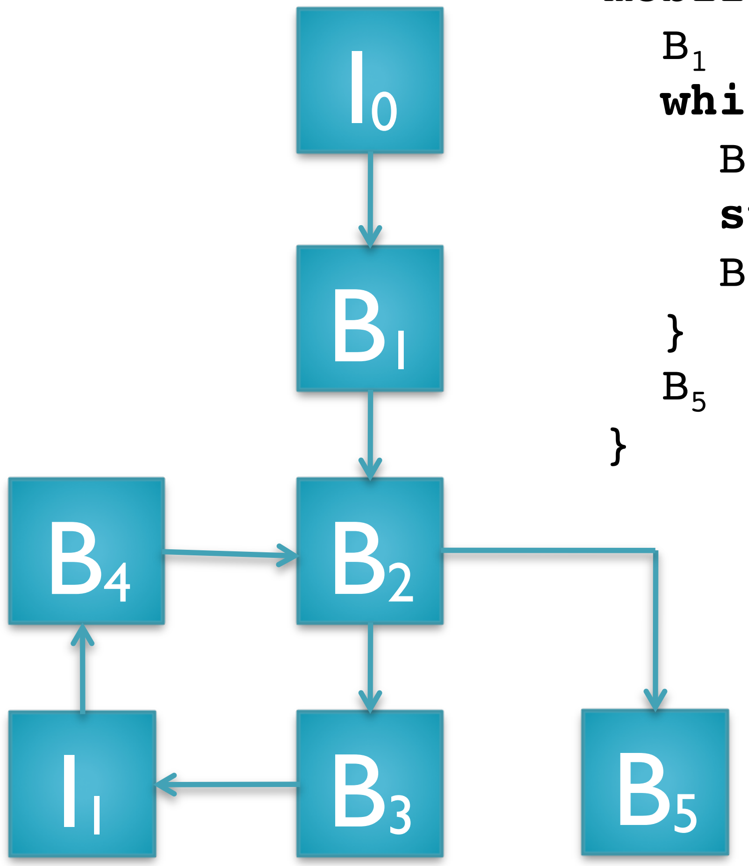
```
foo (int z)
```

CFG Transformation



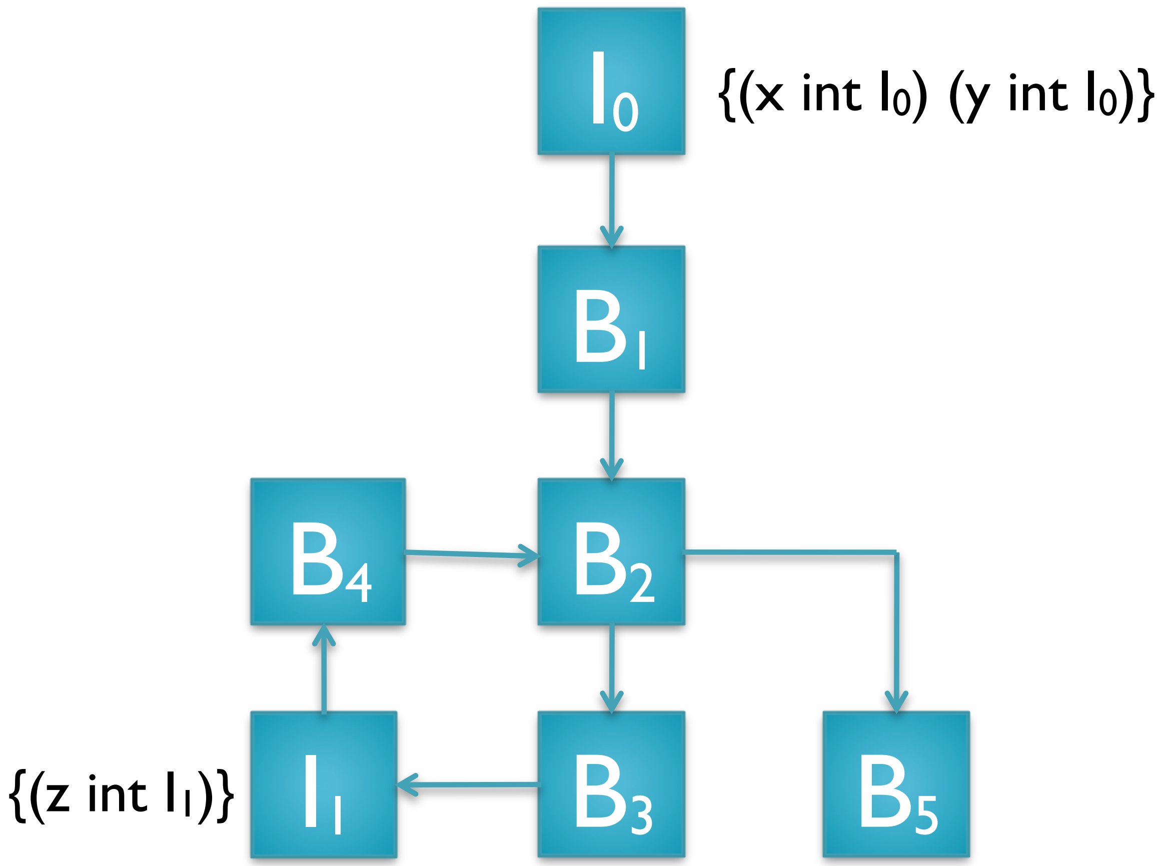
Interfaces are separated out and given their own nodes

Transformed CFG

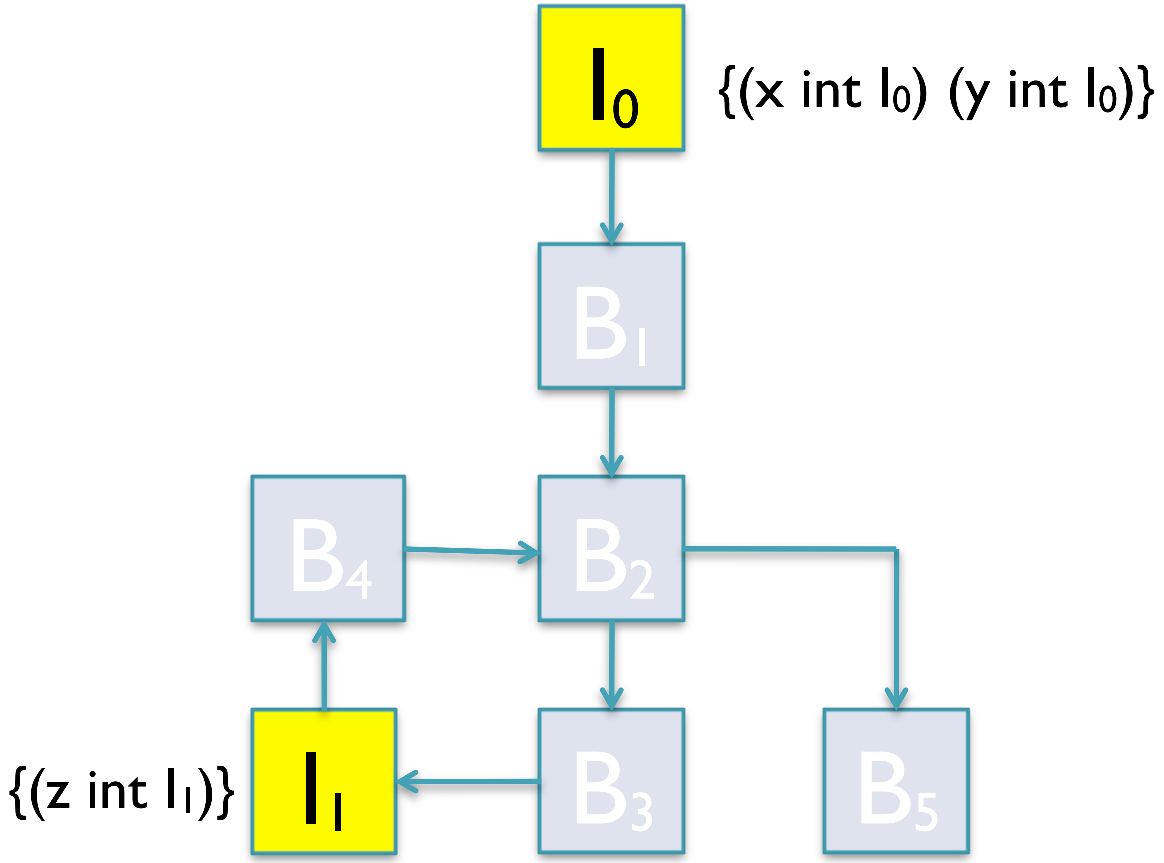


```
mobile void foo (int x, int y) {
  B₁
  while (B₂) {
    B₃
    suspend resume with (int z)
    B₄
  }
  B₅
}
```

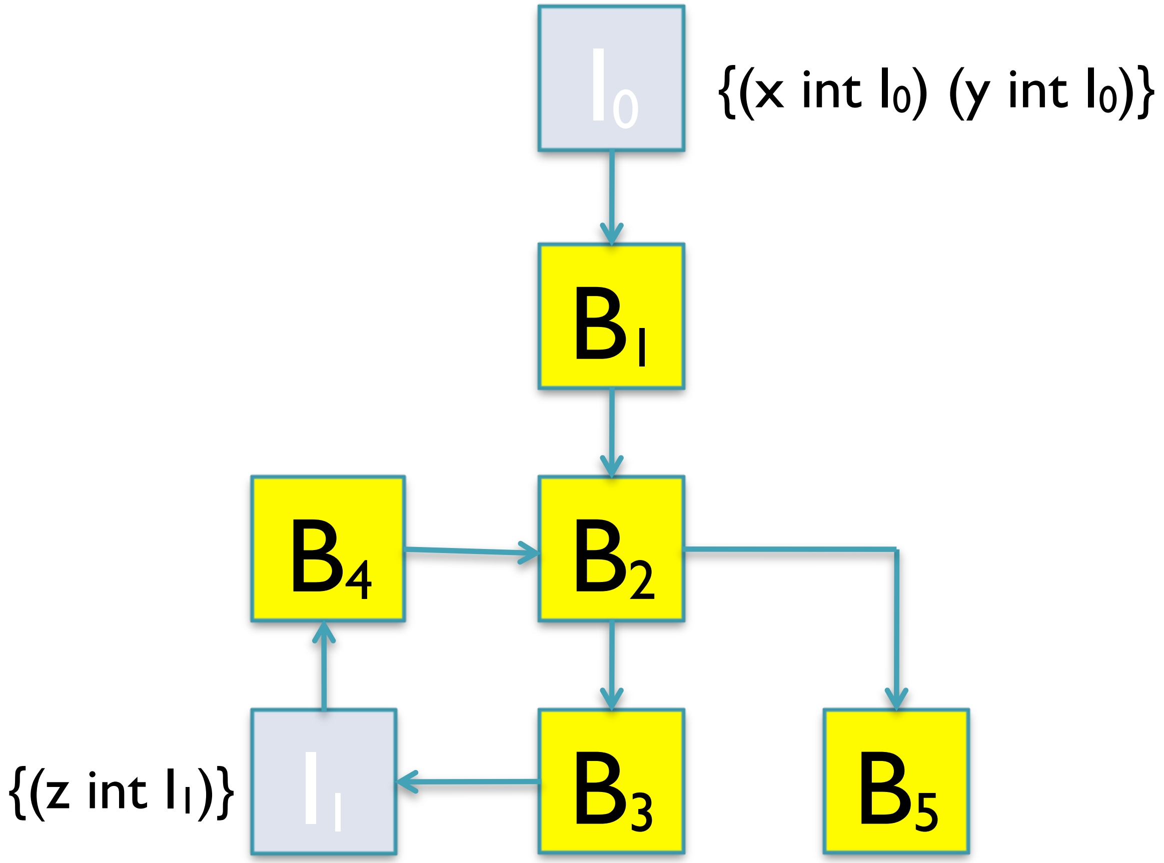
CFG with interface information



Interface Nodes



Code Nodes



I_0
$\{(x \text{ int } I_0)\ (y \text{ int } I_0)\}$
B_1
B_4
B_2
B_3
B_5
I_1
$\{(z \text{ int } I_1)\}$

So Far So Good

- Let us define In and Out sets (loosely):
  - For interface nodes:
    - In Set: Not interesting as an interface defines a new set of **parameters**
    - Out Set: The set of **parameters** defined by the interface
  - For Code nodes:
    - In Set: The set of **parameters** that can be referenced in the node (at least for the final generation of In set)
    - Out Set: a copy of the In set

Generation 0 In and Out Sets

- Interface Nodes
  - $In_0(I_i) = \{\ \}$
  - $Out_0(I_i) = \{(n_{i,1}\ t_{i,1}\ I_i) \ldots (n_{i,k_i}\ t_{i,k_i}\ I_i)\}$
    - The Outset of an interface is the set of triples (name type interface) defined by it
- Code Nodes
  - $In_0(B_j) = \{\ \}$
  - $Out_0(B_j) = \{\ \}$

Example

```
mobile void foo (int x, int y) {        ← $l_0$
   $B_1$
   while ($B_2$) {
      $B_3$
      suspend resume with (int z)       ← $l_1$
      $B_4$
   }
   $B_5$
}
```

- For Example
  - $Out_0(l_0) = \{ (x \text{ int } l_0)\ (y \text{ int } l_0) \}$
  - $Out_0(l_1) = \{ (z \text{ int } l_1) \}$

In and Out Sets

- We generate generations of these sets until no sets change, after which we have the set of parameters that can be referenced for a node **in its In set**
    - We start out with empty sets except for Out sets of interface nodes

CFG with In and Out Sets (Gen. 0)

$$\begin{tikzcd}
 & I_0 \arrow[d] & \\
 & B_1 \arrow[d] & \\
B_4 \arrow[r] & B_2 \arrow[d] \arrow[dr] & \\
I_1 \arrow[u] & B_3 \arrow[l] & B_5
\end{tikzcd}$$

$In(I_0) = \{ \}$
$Out(I_0) = \{(x \text{ int } I_0)\ (y \text{ int } I_0)\}$

$In(B_1) = \{ \}$
$Out(B_1) = \{ \}$

$In(B_4) = \{ \}$
$Out(B_4) = \{ \}$

$In(B_2) = \{ \}$
$Out(B_2) = \{ \}$

$In(B_5) = \{ \}$
$Out(B_5) = \{ \}$

$In(I_1) = \{ \}$
$Out(I_1) = \{(z \text{ int } I_1)\}$

$In(B_3) = \{ \}$
$Out(B_3) = \{ \}$

Generation k+1 (Interface Nodes)

- $In_{k+1}(I_i) = \{\ \}$
  - Interface nodes define a new interface, In sets can be ignored.
- $Out_{k+1}(I_i) =$ $Out_k(I_i) = \{(n_{i,1}\ t_{i,1}\ I_i) \ldots (n_{i,k_i}\ t_{i,k_i}\ I_i)\}$
  - Interface nodes always define the same interface.

Generation k+1 (Code Nodes)

- $In_{k+1}(B_j) = \bigcap_{(N,B_i) \in E_{CFG}} Out_k(N)$
  - New In set is the **intersection** of all the Out sets of the code node's **predecessors** in the CFG

- $Out_{k+1}(B_j) = In_{k+1}(B_j)$
  - The Out set of a code node is the same as its In set, as it cannot define a new interface (Technically not needed but nice to have)

Generation k+1 (Code Nodes)

- $In_{k+1}(B_j) = \bigcap_{(N,B_j) \in E_{CFG}} Out_k(N)$
  - New In set is the **intersection** of all the Out sets of the code nodes **predecessors** in the CFG

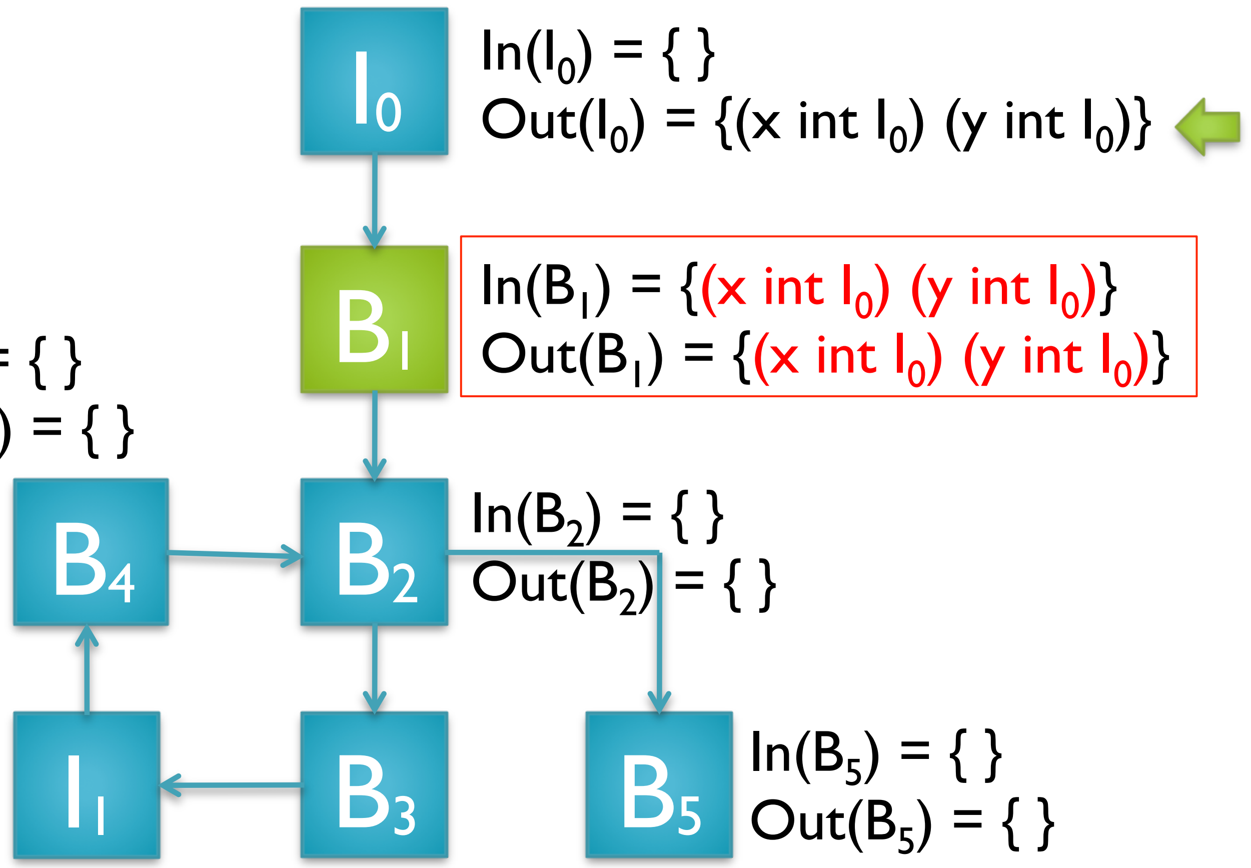
$(n_i\ t_i\ l_i) == (n_j\ t_j\ l_i)$

$<=>$

$(n_i == n_j)\ \wedge\ (t_i == t_j)$

- $Out_{k+1}($
  - The O he same as its In set, as it cannot define a new interface

Generation 1 Sets



$In(I_0)$ = { }
$Out(I_0)$ = {(x int I_0) (y int I_0)}

$In(B_1)$ = {(x int I_0) (y int I_0)}
$Out(B_1)$ = {(x int I_0) (y int I_0)}

$In(B_4)$ = { }
$Out(B_4)$ = { }

$In(B_2)$ = { }
$Out(B_2)$ = { }

$In(B_5)$ = { }
$Out(B_5)$ = { }

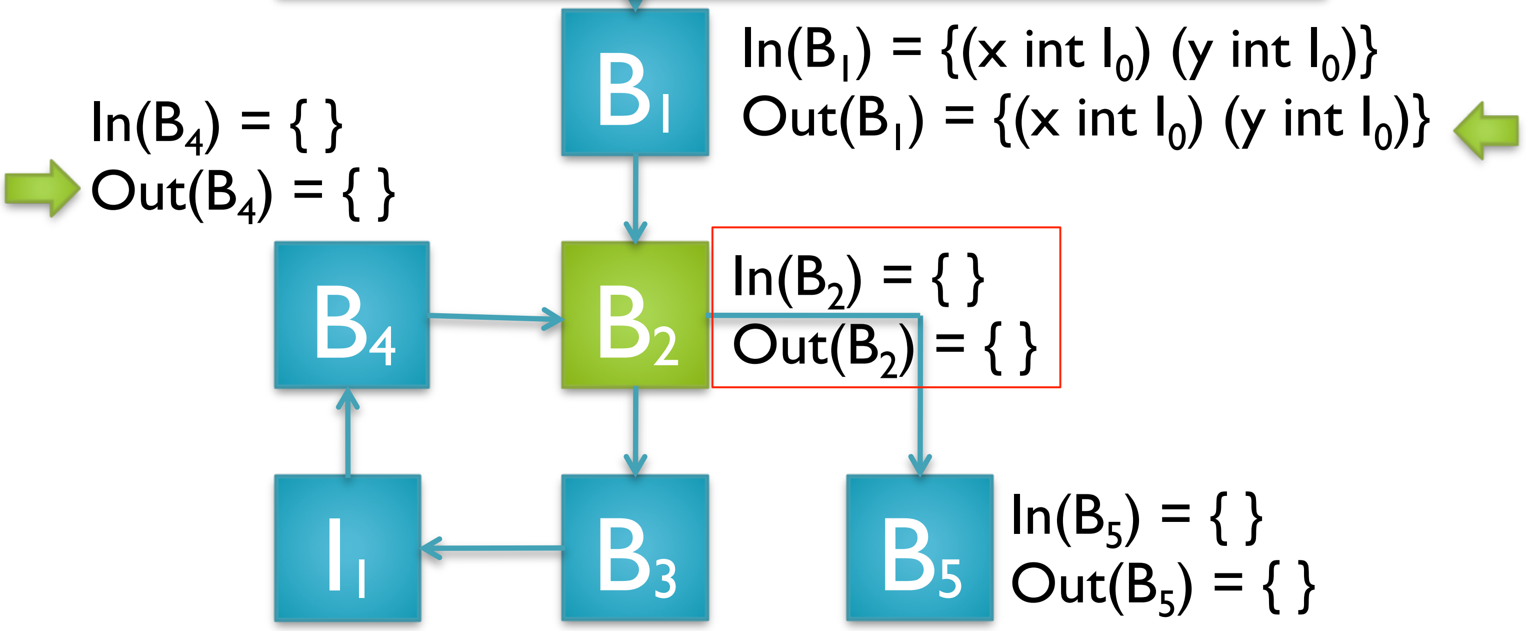
$In(I_1)$ = { }
$Out(I_1)$ = {(z int I_1)}

$In(B_3)$ = { }
$Out(B_3)$ = { }

Generation I Sets

No Changes in B_2 since
$In(B_2) = Out(B_1) \cap Out(B_4)$
$= \{ \} \cap \{(x \text{ int } I_0)\ (y \text{ int } I_0)\}$

$I_0)\}$



$In(B_1) = \{(x \text{ int } I_0)\ (y \text{ int } I_0)\}$
$Out(B_1) = \{(x \text{ int } I_0)\ (y \text{ int } I_0)\}$

$In(B_4) = \{ \}$
$Out(B_4) = \{ \}$

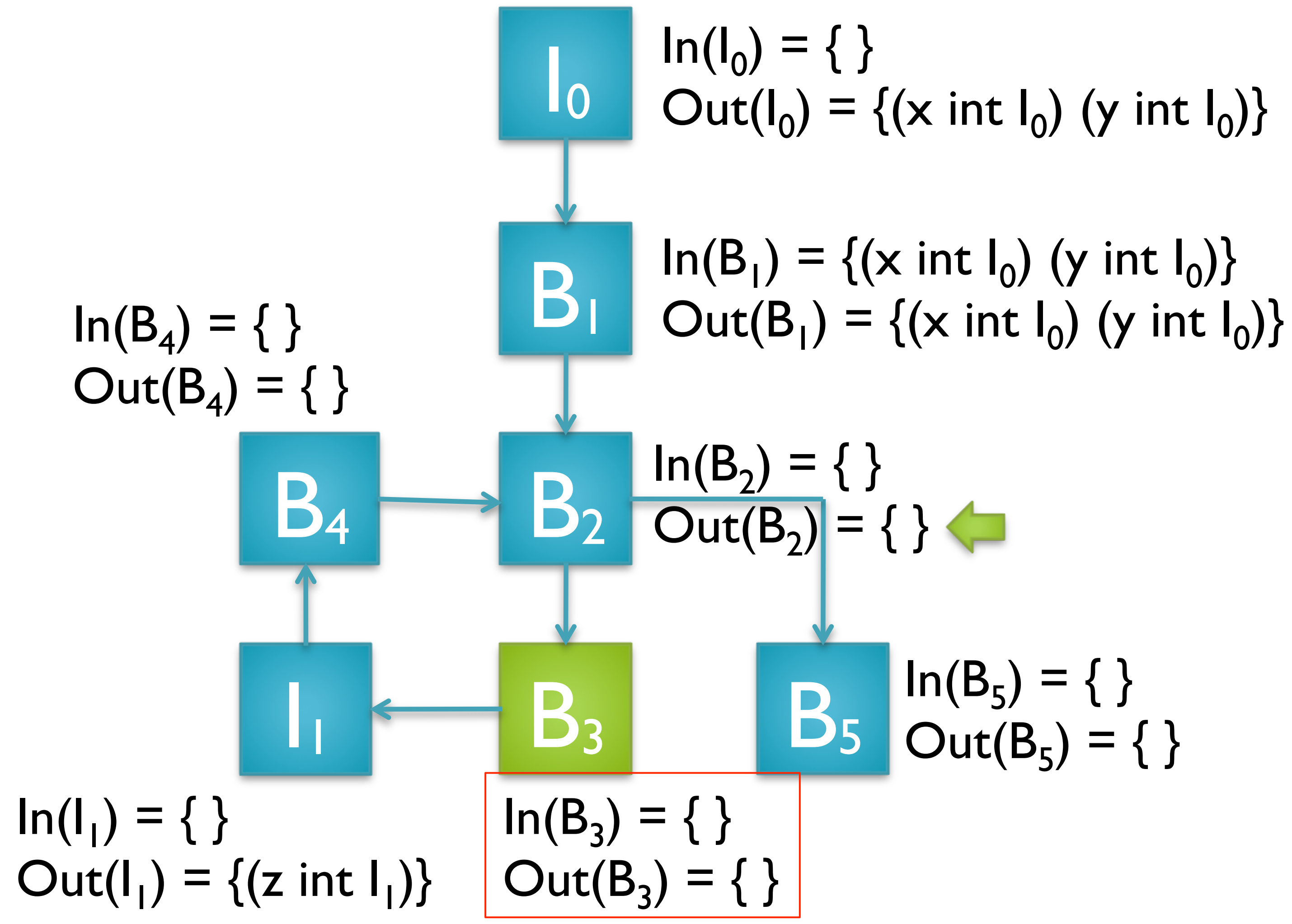
$In(B_2) = \{ \}$
$Out(B_2) = \{ \}$

$In(B_5) = \{ \}$
$Out(B_5) = \{ \}$

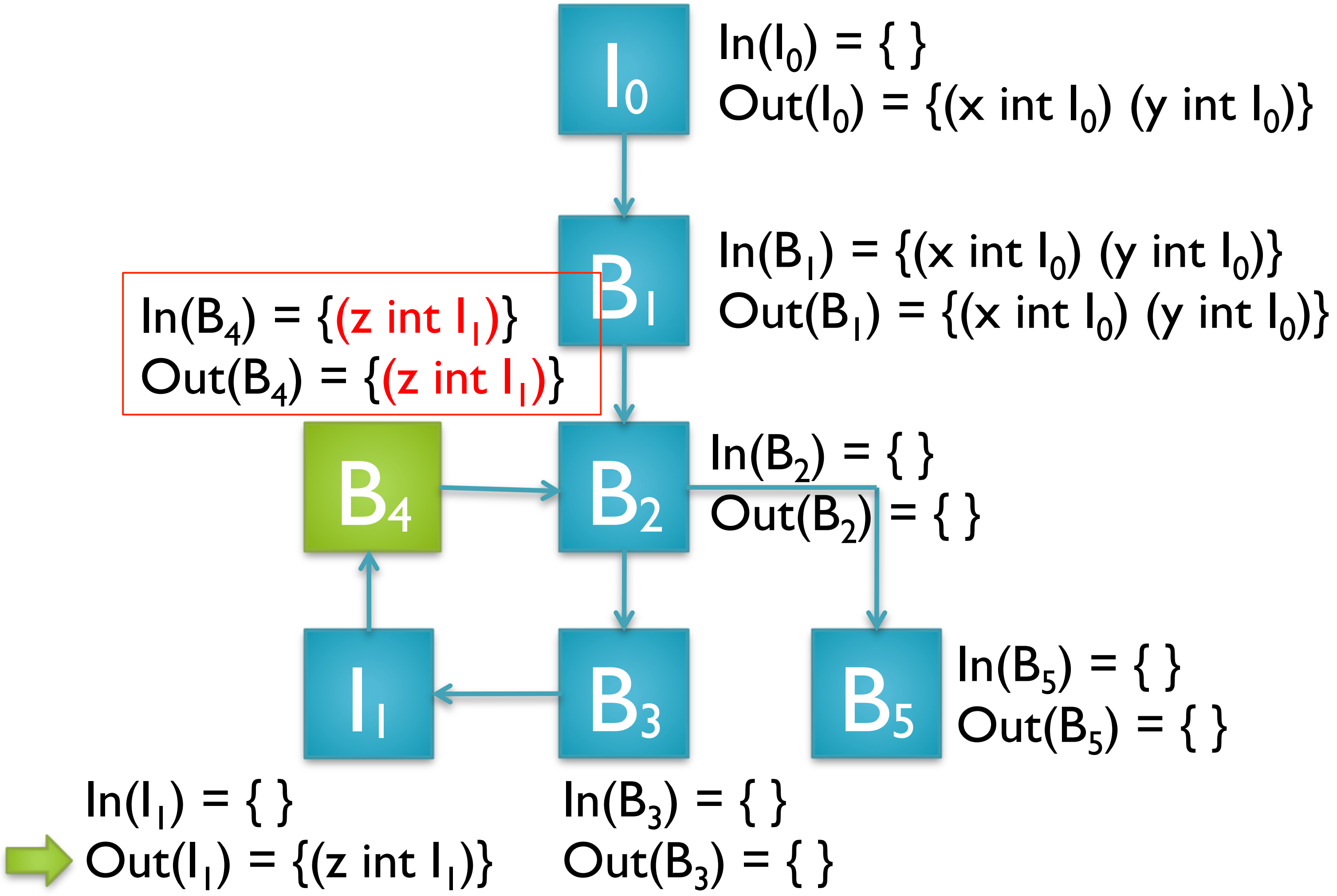
$In(I_1) = \{ \}$
$Out(I_1) = \{(z \text{ int } I_1)\}$

$In(B_3) = \{ \}$
$Out(B_3) = \{ \}$

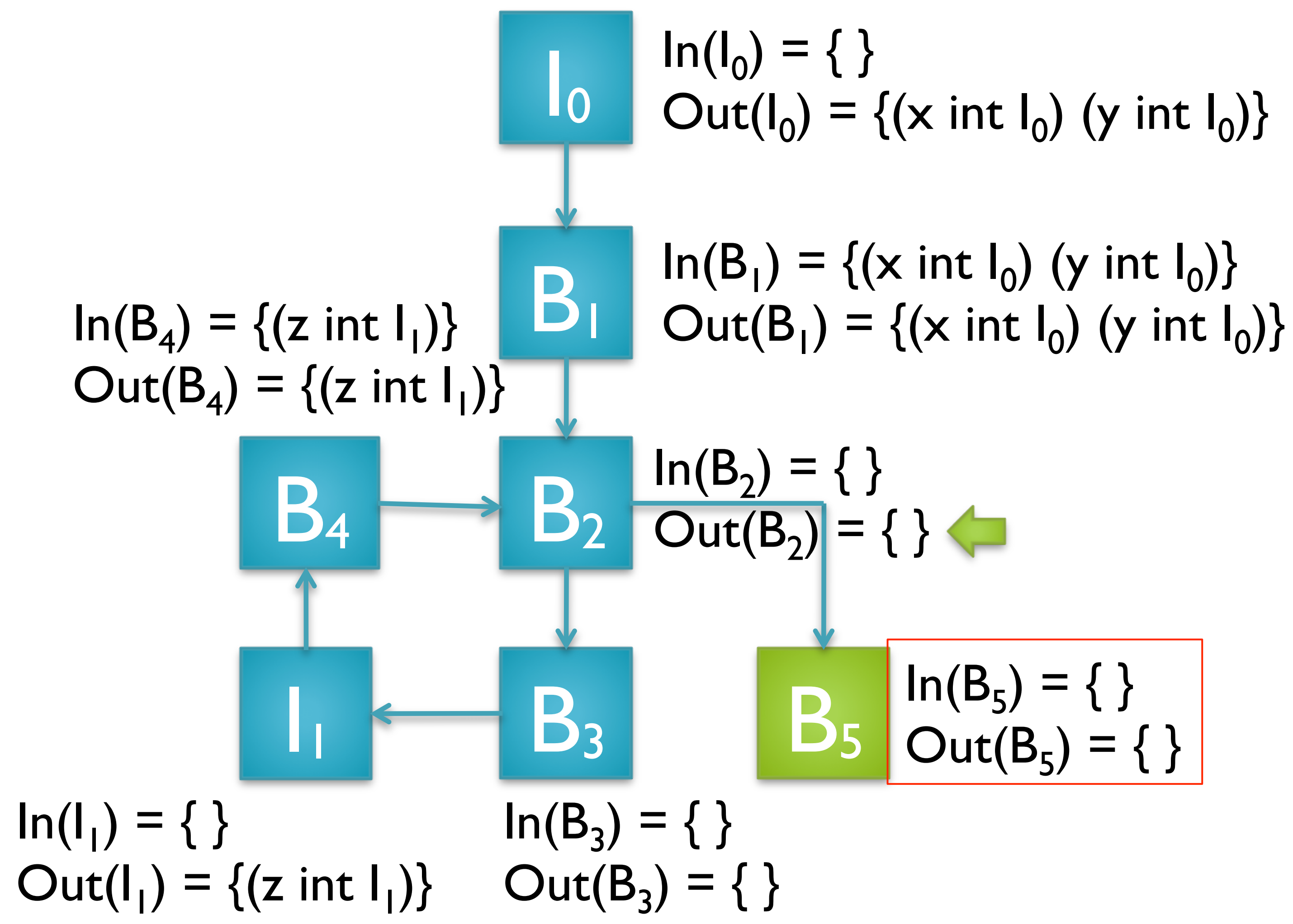
Generation 1 Sets



Generation I Sets


I_0
$In(I_0) = \{\ \}$
$Out(I_0) = \{(x\ int\ I_0)\ (y\ int\ I_0)\}$
B_1
$In(B_1) = \{(x\ int\ I_0)\ (y\ int\ I_0)\}$
$Out(B_1) = \{(x\ int\ I_0)\ (y\ int\ I_0)\}$
$In(B_4) = \{(z\ int\ I_1)\}$
$Out(B_4) = \{(z\ int\ I_1)\}$
B_4
B_2
$In(B_2) = \{\ \}$
$Out(B_2) = \{\ \}$
I_1
B_3
B_5
$In(B_5) = \{\ \}$
$Out(B_5) = \{\ \}$
$In(I_1) = \{\ \}$
$Out(I_1) = \{(z\ int\ I_1)\}$
$In(B_3) = \{\ \}$
$Out(B_3) = \{\ \}$

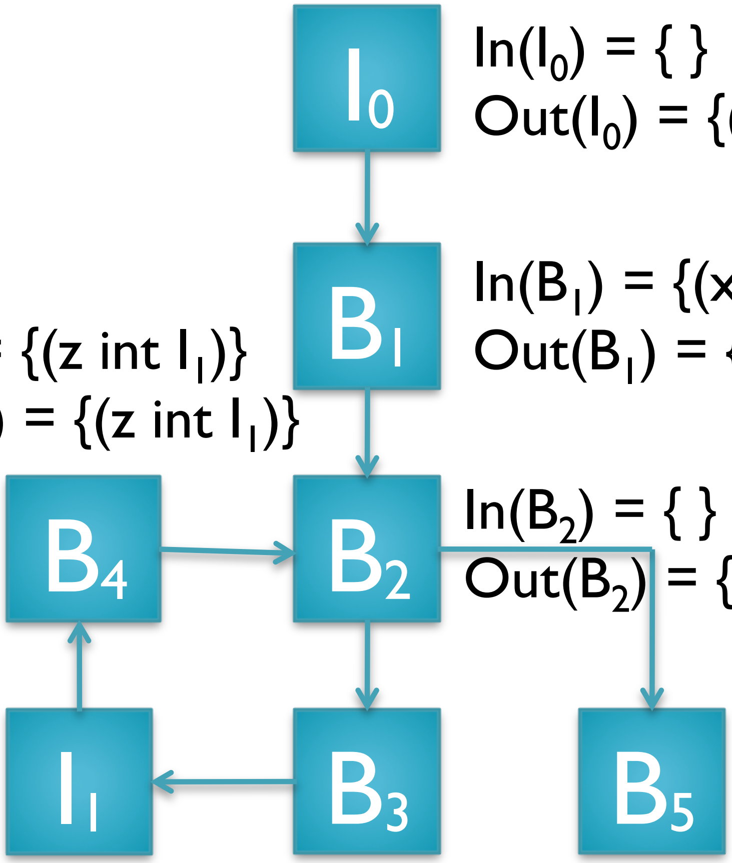
Generation I Sets



I_0
$In(I_0) = \{ \}$
$Out(I_0) = \{(x\ int\ I_0)\ (y\ int\ I_0)\}$
B_1
$In(B_1) = \{(x\ int\ I_0)\ (y\ int\ I_0)\}$
$Out(B_1) = \{(x\ int\ I_0)\ (y\ int\ I_0)\}$
$In(B_4) = \{(z\ int\ I_1)\}$
$Out(B_4) = \{(z\ int\ I_1)\}$
B_4
B_2
$In(B_2) = \{ \}$
$Out(B_2) = \{ \}$
I_1
B_3
B_5
$In(B_5) = \{ \}$
$Out(B_5) = \{ \}$
$In(I_1) = \{ \}$
$Out(I_1) = \{(z\ int\ I_1)\}$
$In(B_3) = \{ \}$
$Out(B_3) = \{ \}$

In & Out Sets after Generation 1

$$\begin{tikzcd}
 & I_0 \arrow[d] & \\
 & B_1 \arrow[d] & \\
B_4 \arrow[r] & B_2 \arrow[d] \arrow[rd] & \\
I_1 \arrow[u] & B_3 \arrow[l] & B_5
\end{tikzcd}$$

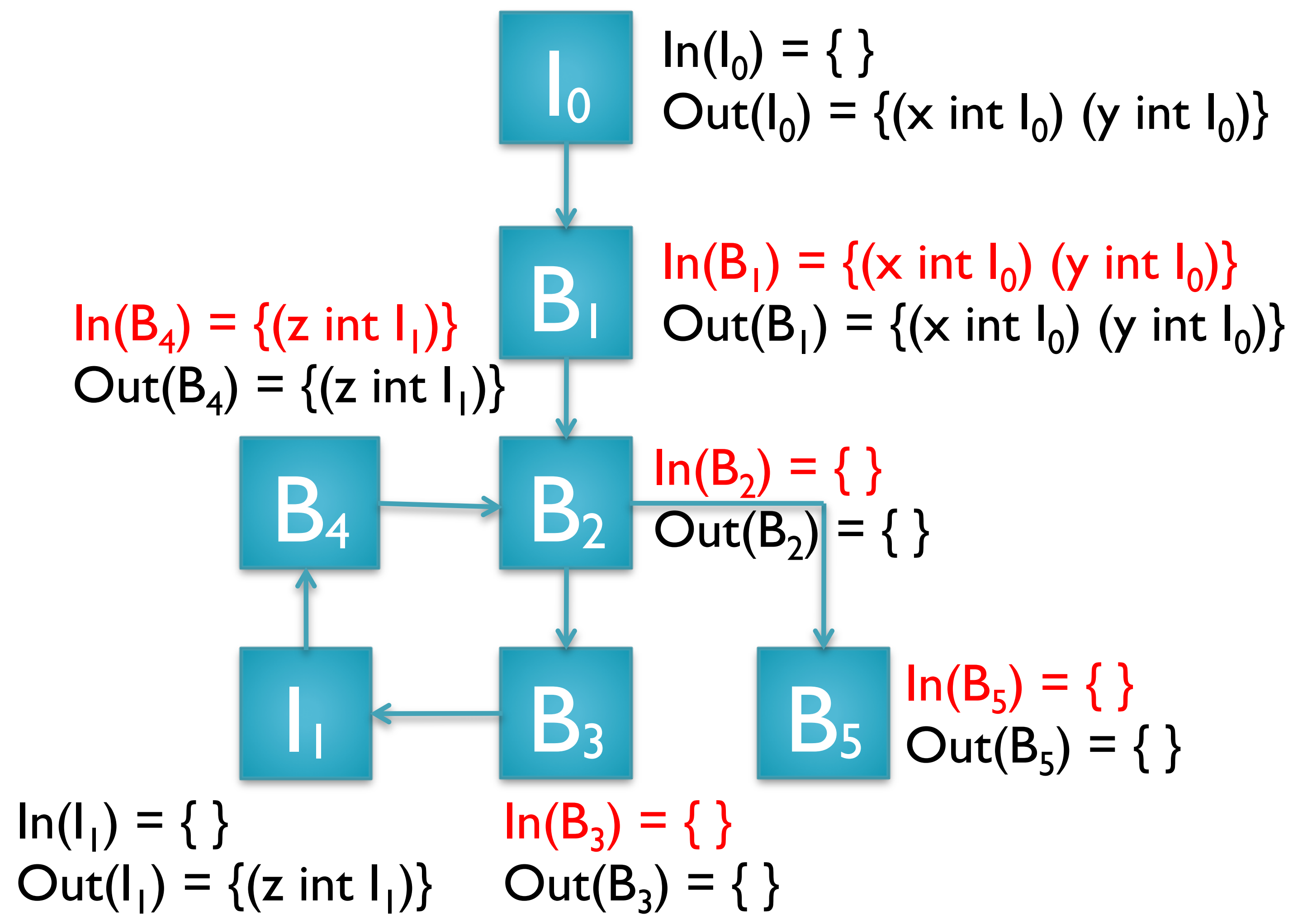


$In(I_0)$ = { }
$Out(I_0)$ = {(x int I_0) (y int I_0)}
$In(B_1)$ = {(x int I_0) (y int I_0)}
$Out(B_1)$ = {(x int I_0) (y int I_0)}
$In(B_4)$ = {(z int I_1)}
$Out(B_4)$ = {(z int I_1)}
$In(B_2)$ = { }
$Out(B_2)$ = { }
$In(B_5)$ = { }
$Out(B_5)$ = { }
$In(I_1)$ = { }
$Out(I_1)$ = {(z int I_1)}
$In(B_3)$ = { }
$Out(B_3)$ = { }

Generation 2 In and Out Sets

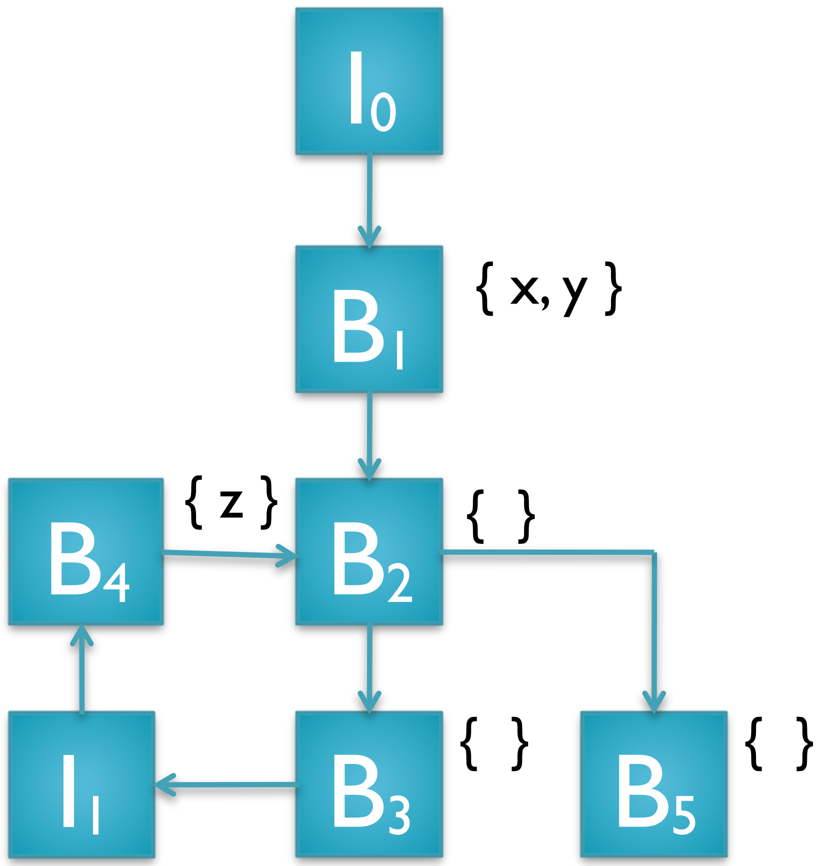
- Nothing changes when computing generation 2.

Final In and Out Sets



I_0
$In(I_0) = \{ \}$
$Out(I_0) = \{(x \text{ int } I_0)\ (y \text{ int } I_0)\}$
B_1
$In(B_1) = \{(x \text{ int } I_0)\ (y \text{ int } I_0)\}$
$Out(B_1) = \{(x \text{ int } I_0)\ (y \text{ int } I_0)\}$
$In(B_4) = \{(z \text{ int } I_1)\}$
$Out(B_4) = \{(z \text{ int } I_1)\}$
B_4
B_2
$In(B_2) = \{ \}$
$Out(B_2) = \{ \}$
I_1
B_3
B_5
$In(B_5) = \{ \}$
$Out(B_5) = \{ \}$
$In(I_1) = \{ \}$
$Out(I_1) = \{(z \text{ int } I_1)\}$
$In(B_3) = \{ \}$
$Out(B_3) = \{ \}$

Final Result of Analysis



Final Result of Analysis

- During all possible executions we can only guarantee that
  - `x` and `y` are always available in B_1
  - `z` is always available in B_4

Name Resolution

- We have **not** considered local variables or how to resolve name usage in the code
  - For locals, regular scoping rules apply
  - Locals can hide parameters
  - One symbol table for an interface
  - One symbol table for its body

Name Resolution

- A suspend/resume point acts like the procedure interface
  - One symbol table for the interface
  - One symbol table for the implicit 'body' following
    - End of scope determined by the closest enclosing scope of the suspend/resume statement.
- A block { } and a for-statement opens a new scope as well

Now with Locals ;-)

```
mobile void foo (int x, int y)
{
  int a;
  B1
  while (B2)
  {
    int q;
    B3
    suspend resume with (int z)
    int w,z;
    B4
  }↖ Implicit end of scopes opened by suspend
  B5
}
```

Now with Locals And Scopes

```
mobile void foo
{+T0
    (int x, int y)
    {+T1
        int a;
        B1
        while (B2)
        {+T2
            int q;
            B3
            suspend resume with
            {+T3
                (int z)
                {+T4
                    int w,z;
                    B4
                }-T4
            }-T3
        }-T2
        B5
    }-T1
}-T0
```

Implicit scopes added in red

- Parameter scope for procedure interface (T_0)
- Parameter scope for suspend/resume interface (T_3)
- Body scope for suspend/resume interface (T_4)

Name Resolution

- A symbol Table now has an ‘access list’
  - Only name-uses in code blocks listed in the access list are allowed to perform a look up in the table – if not listed, move on to the parent table.

Name	Value
int	z
Access List: {1,2,3,4,5}	

$\rightarrow$ Parent Table

Symbol Tables

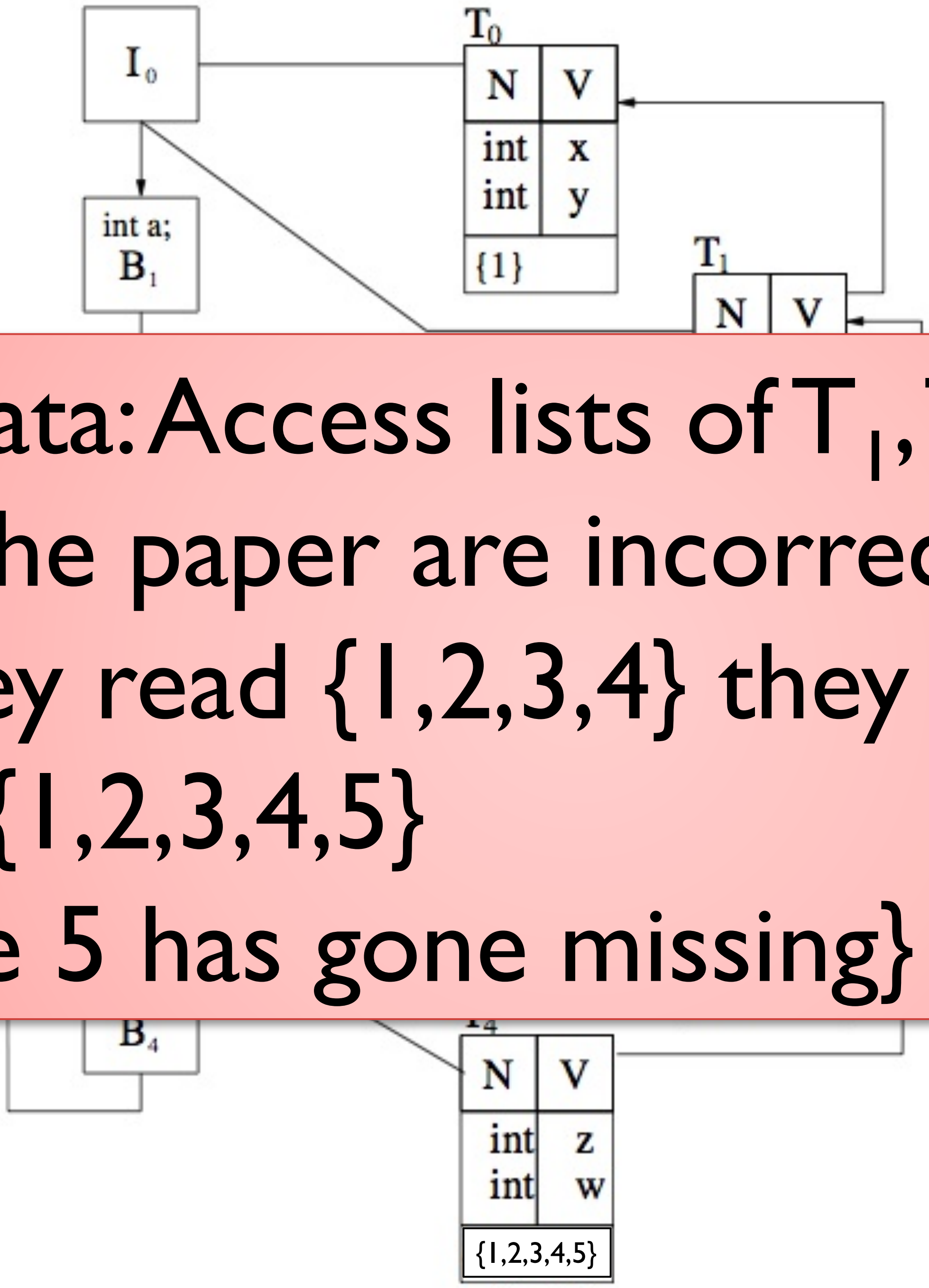
- Only symbol tables associated with the **implicit scopes** of interfaces have limited access lists; all others have full access

Table for T_0
foo (int x, int y)

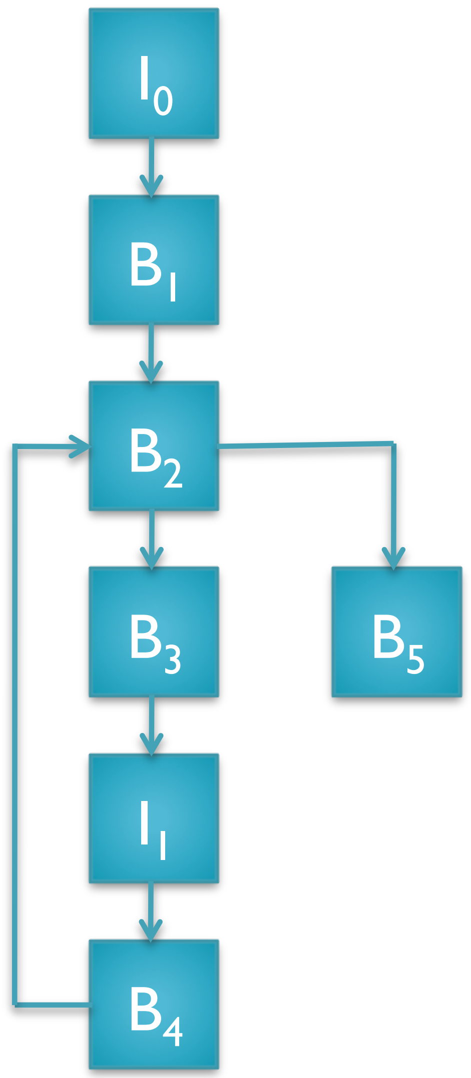
Name	Value
int	x
int	y
Access List: { 1 }	

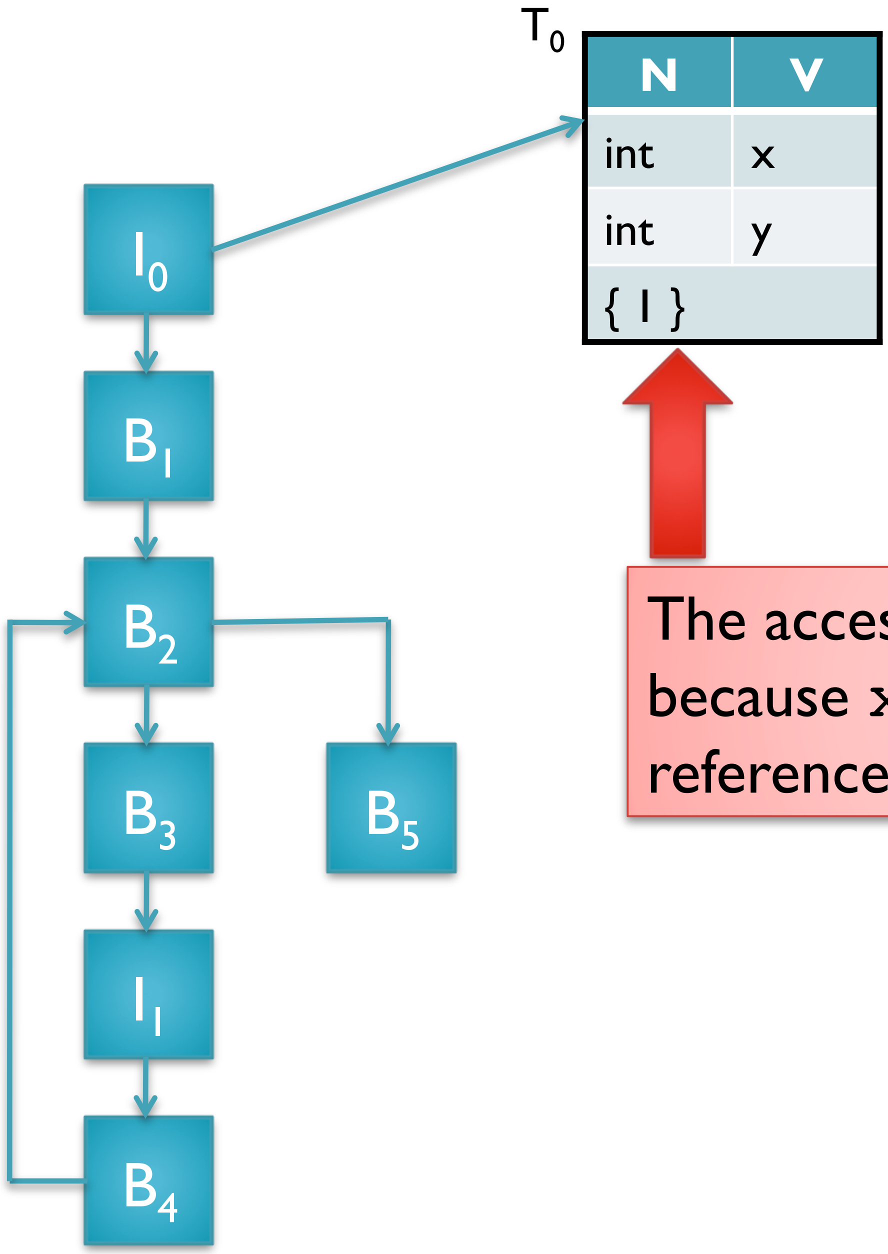
Table for T_3
suspend resume with (int z)

Name	Value
int	z
Access List: { 4 }	



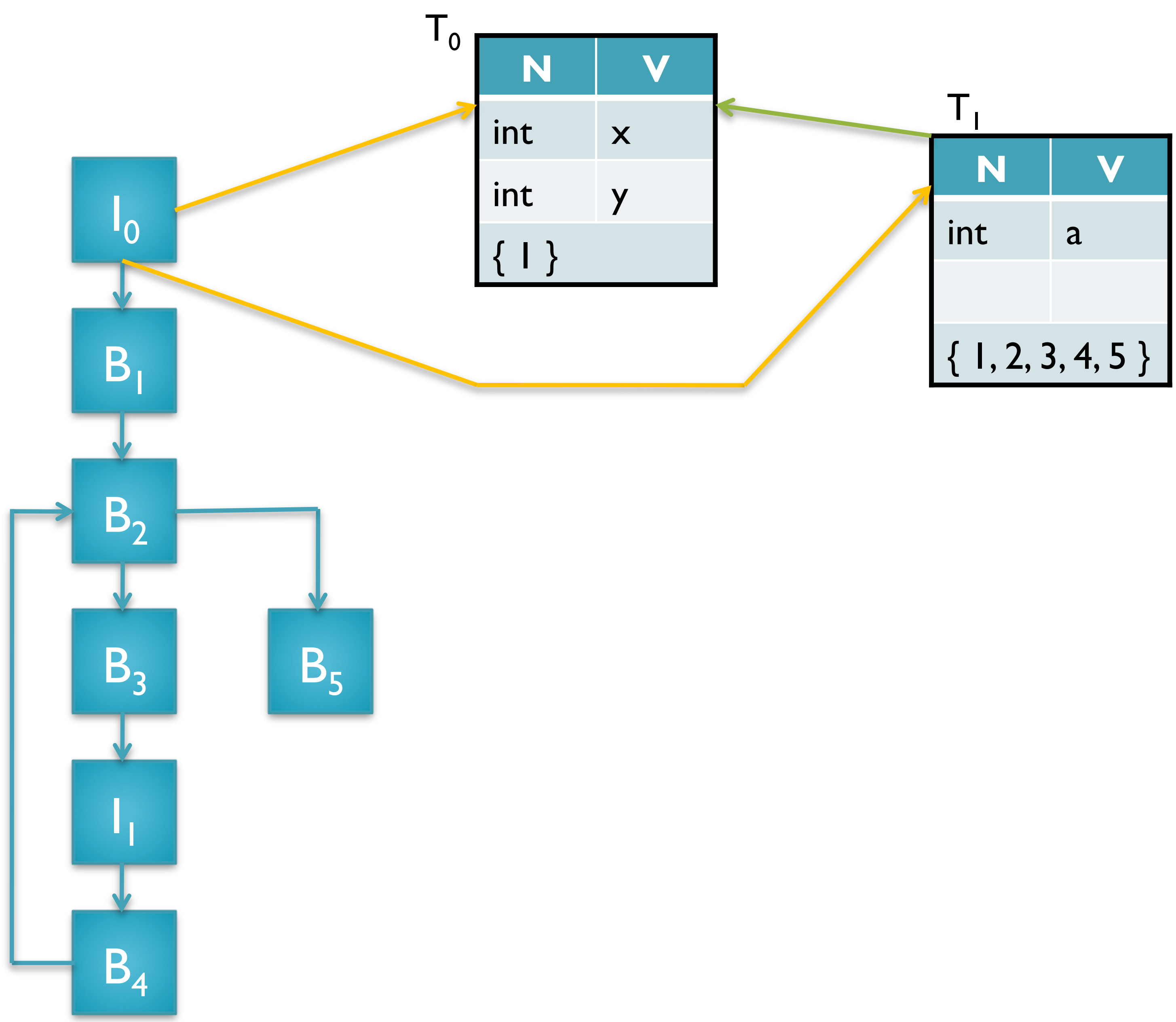
Errata: Access lists of T_1, T_2 & T_4 in the paper are incorrect.
They read {1,2,3,4} they should be {1,2,3,4,5}
(the 5 has gone missing}

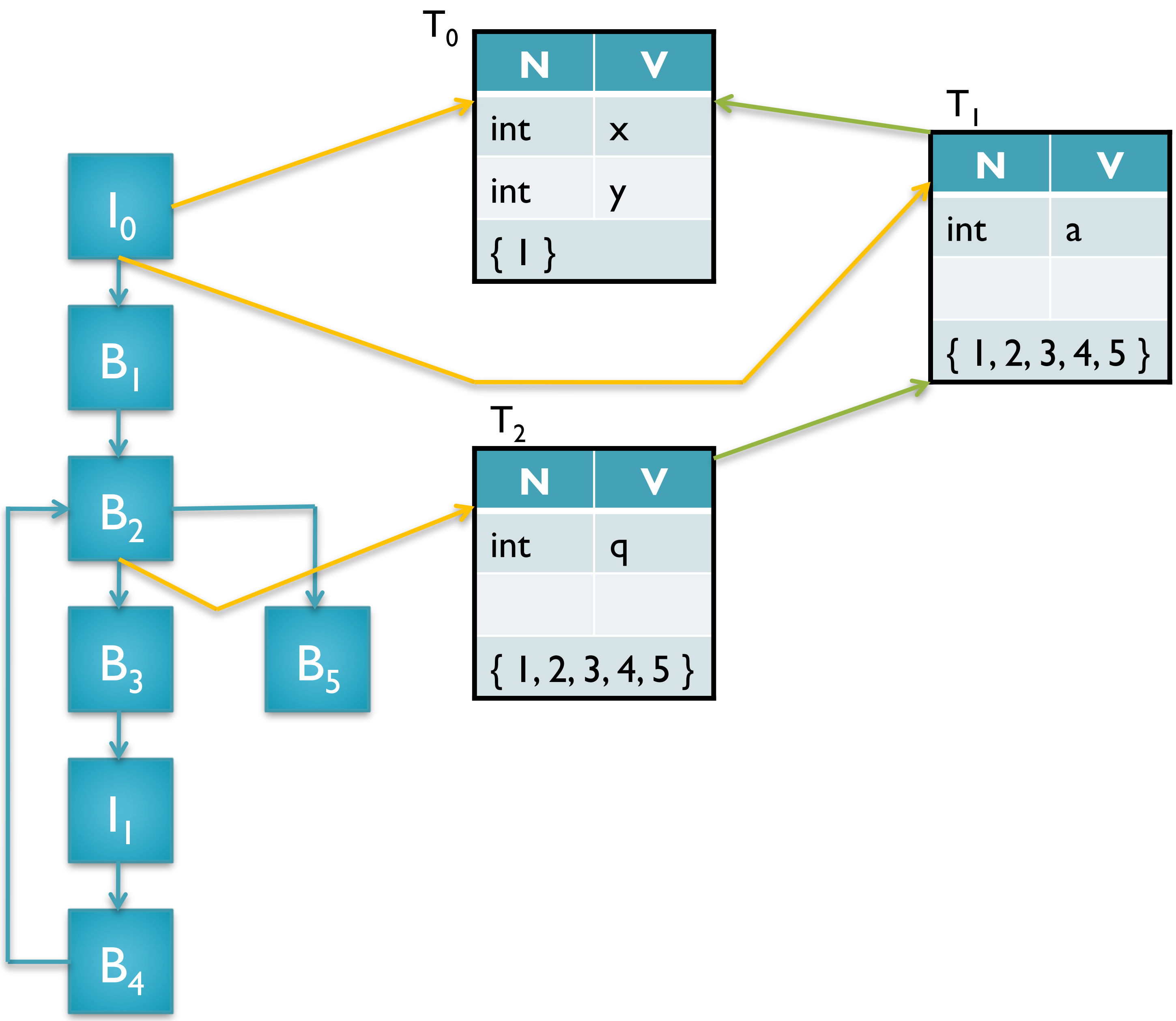

I_0
B_1
B_2
B_3
B_5
I_1
B_4



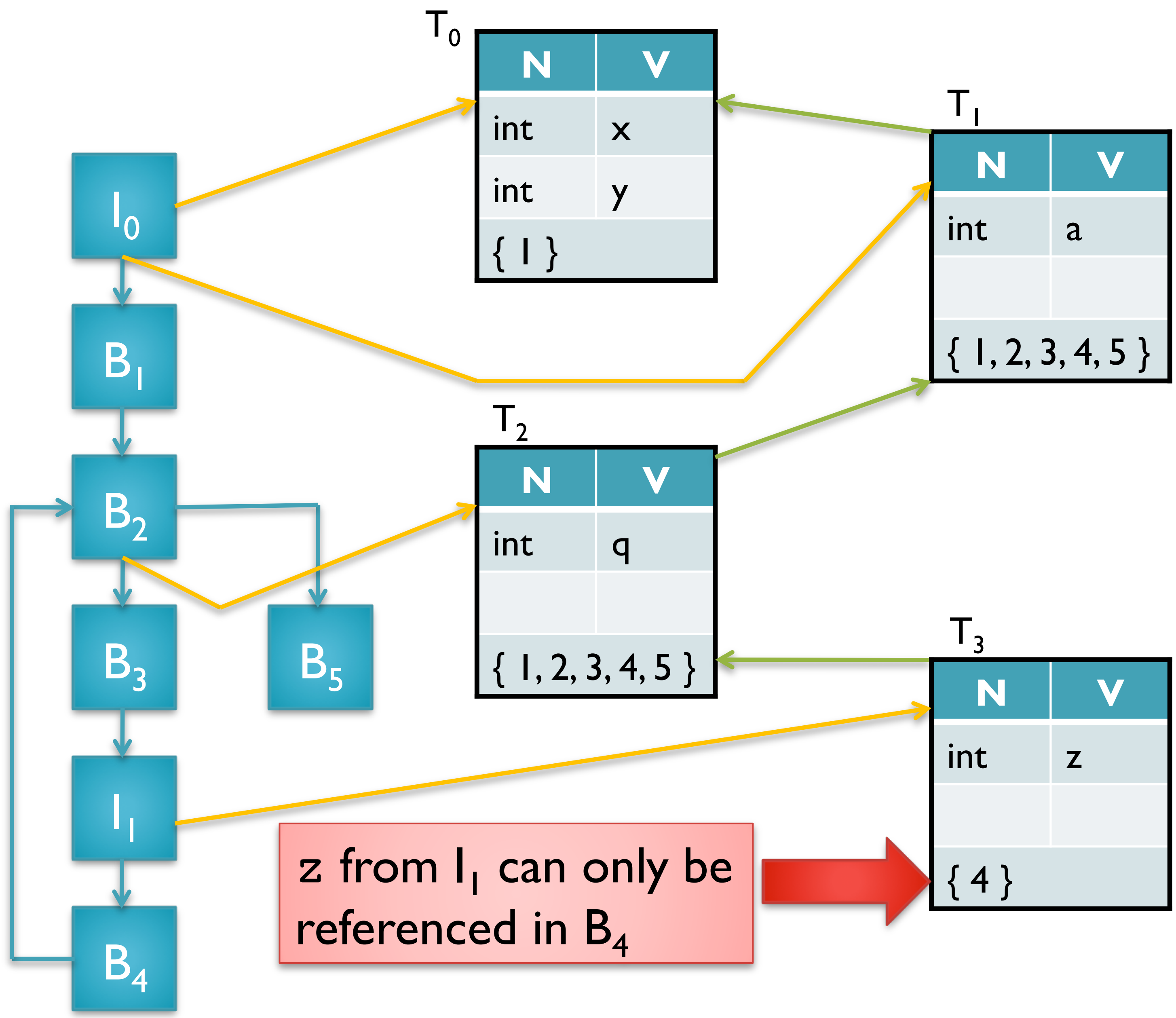
N	V
int	x
int	y
{ 1 }	

The access list contains only 1, because x and y can only be referenced in B_1.

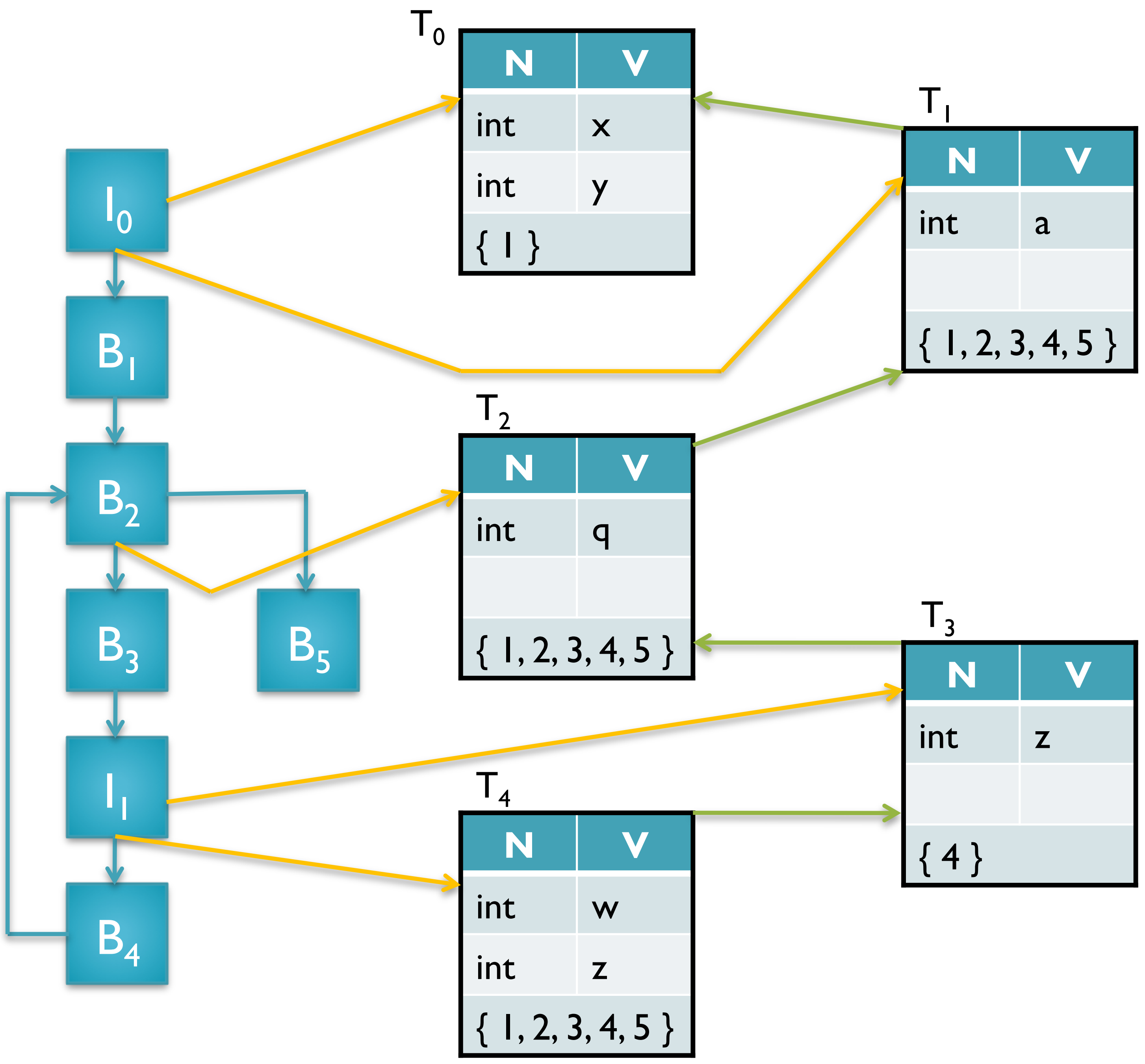

T_0
N
V
int
x
int
y
{ 1 }
T_1
N
V
int
a
{ 1, 2, 3, 4, 5 }
I_0
B_1
B_2
B_3
B_5
I_1
B_4



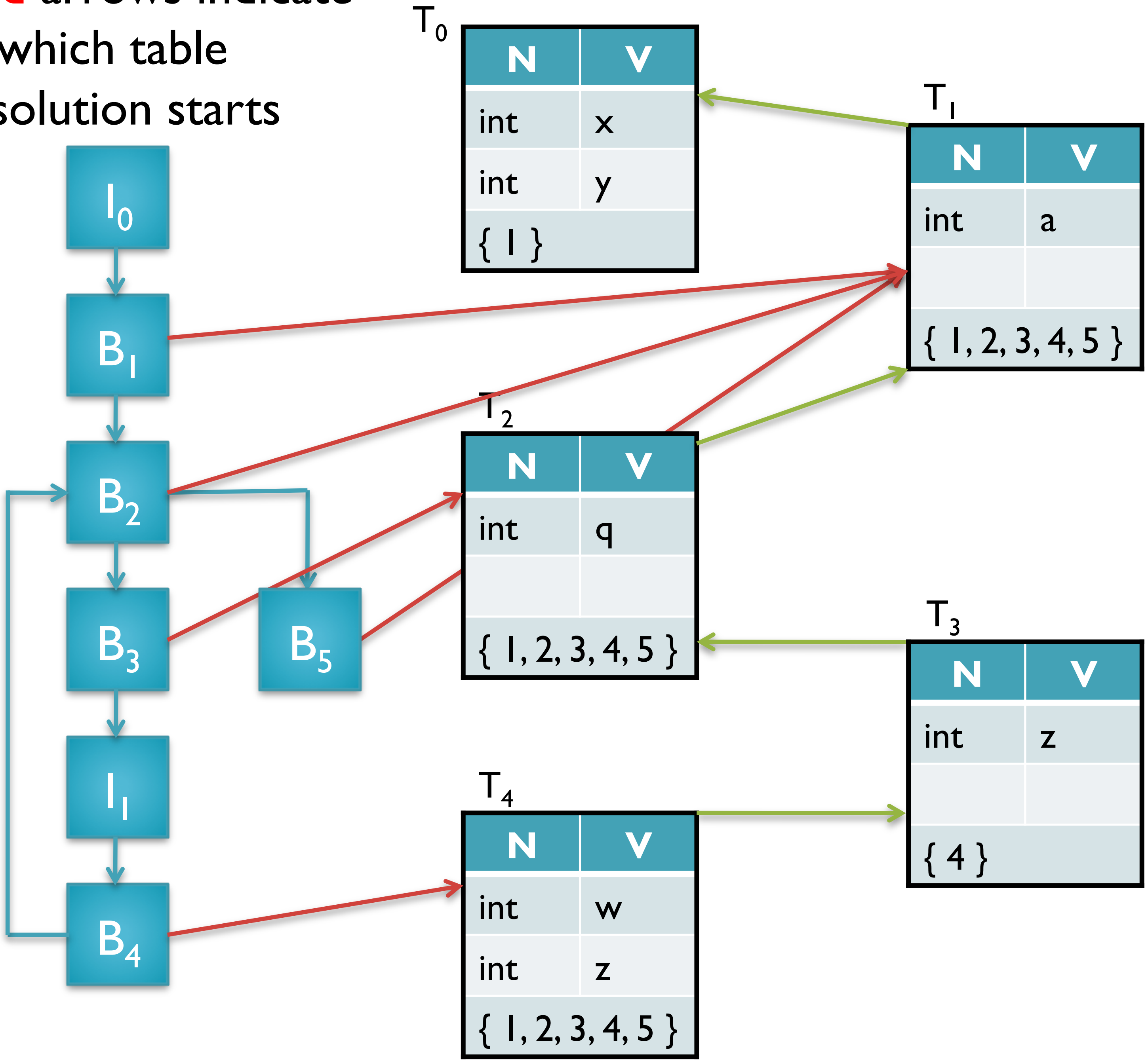
T_0
N
V
int
x
int
y
{ 1 }
T_1
N
V
int
a
{ 1, 2, 3, 4, 5 }
T_2
N
V
int
q
{ 1, 2, 3, 4, 5 }
I_0
B_1
B_2
B_3
B_5
I_1
B_4



z from I_1 can only be referenced in B_4


T0
N
V
int
x
int
y
{ 1 }
T1
N
V
int
a
{ 1, 2, 3, 4, 5 }
I0
B1
B2
B3
B5
I1
B4
T2
N
V
int
q
{ 1, 2, 3, 4, 5 }
T3
N
V
int
z
{ 4 }
T4
N
V
int
w
int
z
{ 1, 2, 3, 4, 5 }

Red arrows indicate in which table resolution starts



Final Results

- We get the following table of blocks and which parameters and locals they can reference.

Block	Locals	Parameter
B_1	$a \in T_1$	$x \in T_0, y \in T_0$
B_2	$a \in T_1$	
B_3	$q \in T_2,\ a \in T_1$	
B_4	$w \in T_4, z \in T_4, q \in T_2, a \in T_1$	$z \in T_3$
B_5	$a \in T_1$	

Errata: Table 4 Parameter for B4 should read $z \in T_3$ and **not** $z \in T_4$

Conclusion

- We have defined and implemented mobile procedures with polymorphic interfaces in ProcessJ
- Provided a new scope resolution mechanism for polymorphic mobiles that performs correct name resolution

Conclusion

- Provided an implementation in Java/JCSP (ProcessJ translated to Java with JCSP)
    - Paper on the implementation (similar to our 2009 paper at CPA but without byte code rewriting) is being presented at PDPTA 2011 in July.

processj.cs.unlv.edu (currently turned off cause of hackers but we will be back up soon)

Questions

or

Lunch?