Register Allocation by Puzzle Solving

EECS 322: Compiler Construction

Simone Campanoni
Robby Findler

4/19/2016

Materials

- Research paper:
  - Authors: Fernando Magno Quintao Pereira, Jens Palsberg
  - Title: Register Allocation by Puzzle Solving
  - Conference: PLDI 2008

- Ph.D. thesis
  - Author: Fernando Magno Quintao Pereira
  - Title: Register Allocation by Puzzle Solving
  - UCLA 2008

A compiler

Character stream (Source code)

↓

Front-end

↓

IR

↓

Middle-end

↓

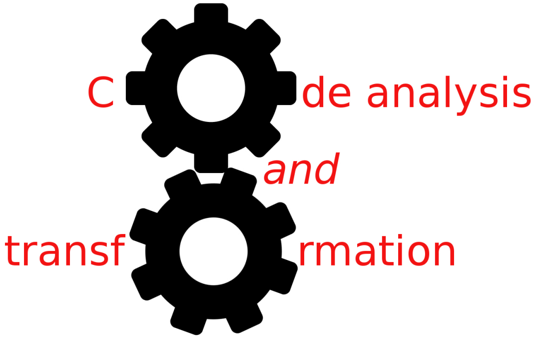
IR

↓

Back-end

↓

Machine code

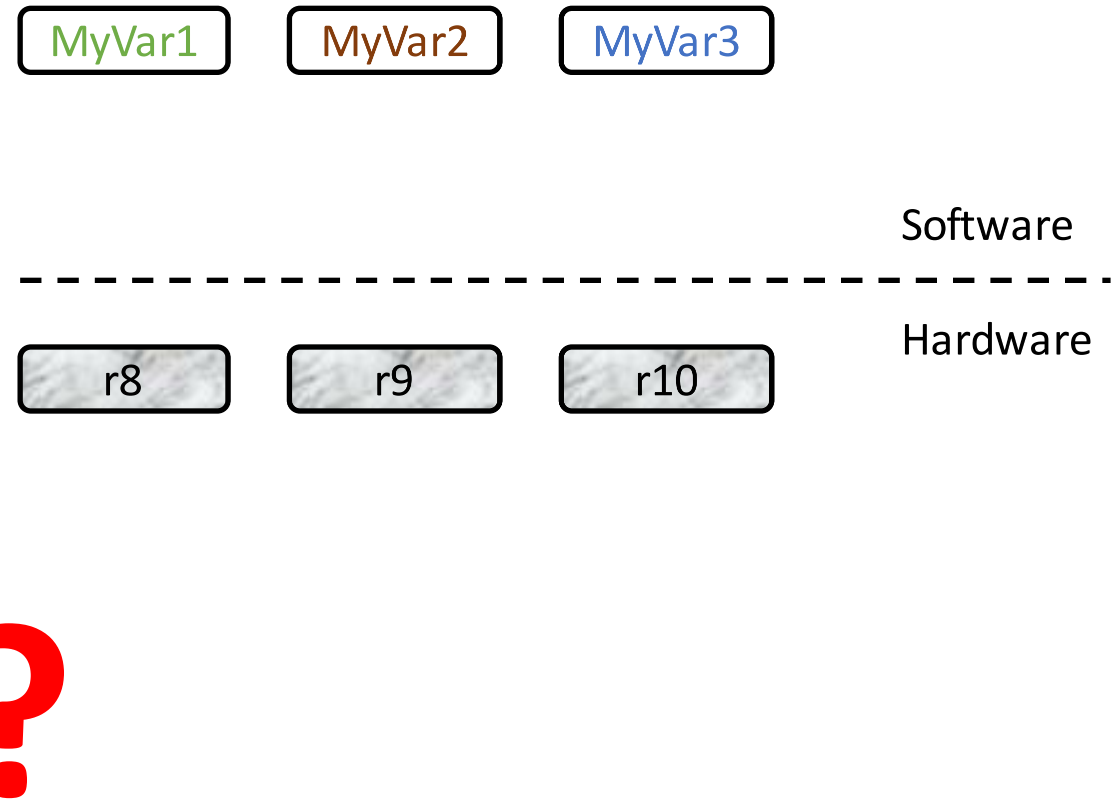


Task: From Variables to Registers

No overlapping

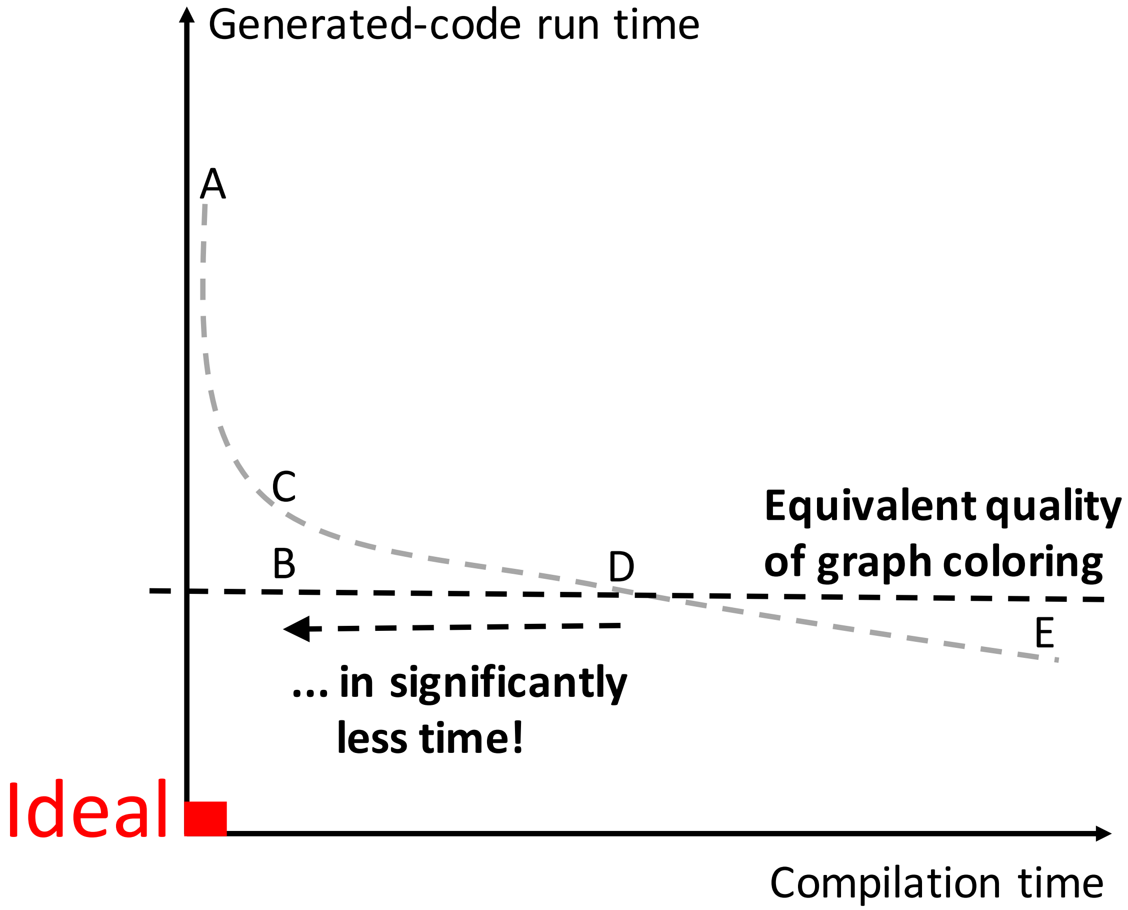
```
(:MyVeryImportantFunction

  (MyVar1 <- 2)
  (MyVar2 <- 40)
  (MyVar3 <- MyVar1)
  (MyVar3 += MyVar2)
  (print MyVar3)
)
```



Register Allocation

A. Spill all variables
B. Puzzle solving
C. Linear scan
D. Graph coloring
E. Integer linear programming



Summary

- Graph coloring abstraction: Houston we have a problem
- Puzzle abstraction
- From a program to a collection of puzzles
- Solve puzzles
- From solved puzzles to assembly code

To register allocators: what are you doing?

```
(:MyVeryImportantFunction

 (MyVar1 <- 2)
 (MyVar2 <- 40)
 (MyVar3 <- 0)
 (MyVar3 += MyVar1)
 (MyVar3 += MyVar2)
 (print MyVar3)
)
```

MyVar1 MyVar2 MyVar3

Software

Hardware

r8 r9

- MyVar1 -> stack (spilled)
- MyVar2 -> r8
- MyVar3 -> r9

Graph coloring abstraction: a problem

```
(:MyVeryImportantFunction

  (MyVar1 <- 2)
  (MyVar2 <- 40)
  (MyVar3 <- 0)
  (MyVar3 += MyVar1)
  (MyVar3 += MyVar2)
  (print MyVar3)
)
```

MyVar1 MyVar2 MyVar3

- MyVar1 : 64 bits
- MyVar2 : 32 bits
- MyVar3 : 32 bits

Software

Hardware

r8 r9

Register aliasing →
- r8 can store either one 64-bit valuel or two 32-bit values
- r9 can store 64 bit values

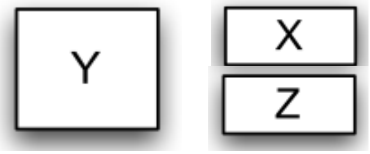
Can this be obtained
by the graph-coloring algorithm
you learned in this class?

Summary

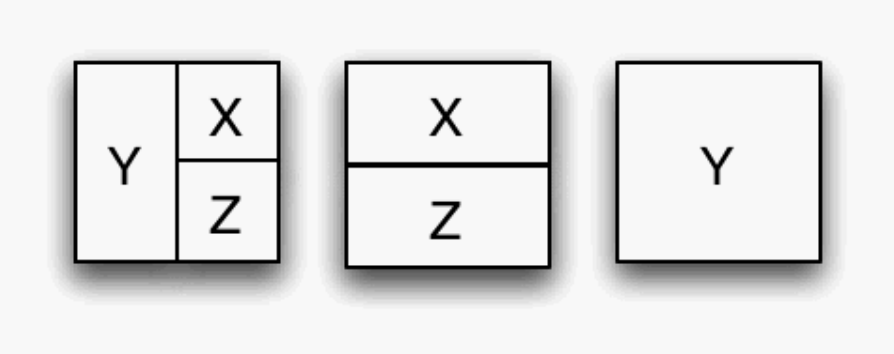
- Graph coloring abstraction: Houston we have a problem
- Puzzle abstraction
- From a program to a collection of puzzles
- Solve puzzles
- From solved puzzles to assembly code

Puzzle Abstraction

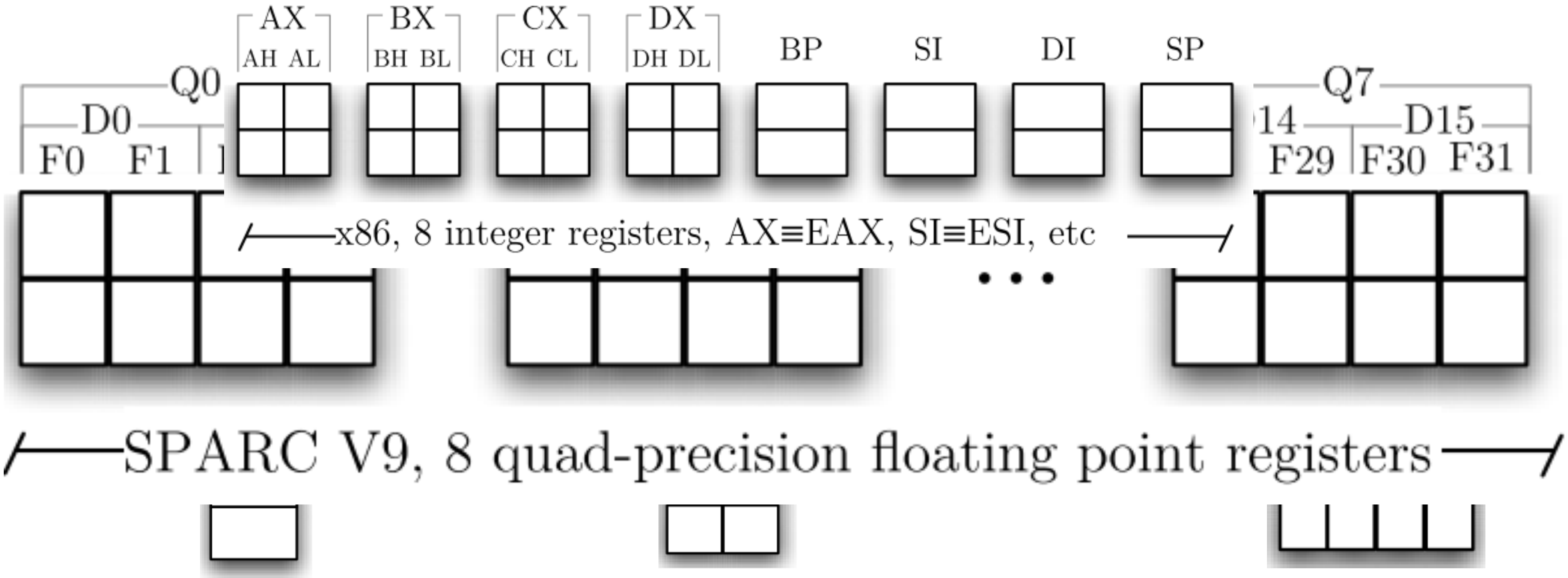
- Puzzle = board (areas = registers) + pieces (variables)



- Pieces cannot overlap
- Some pieces are already placed on the board
- Task: fit the remaining pieces on the board (register allocation)



From register file to puzzle boards

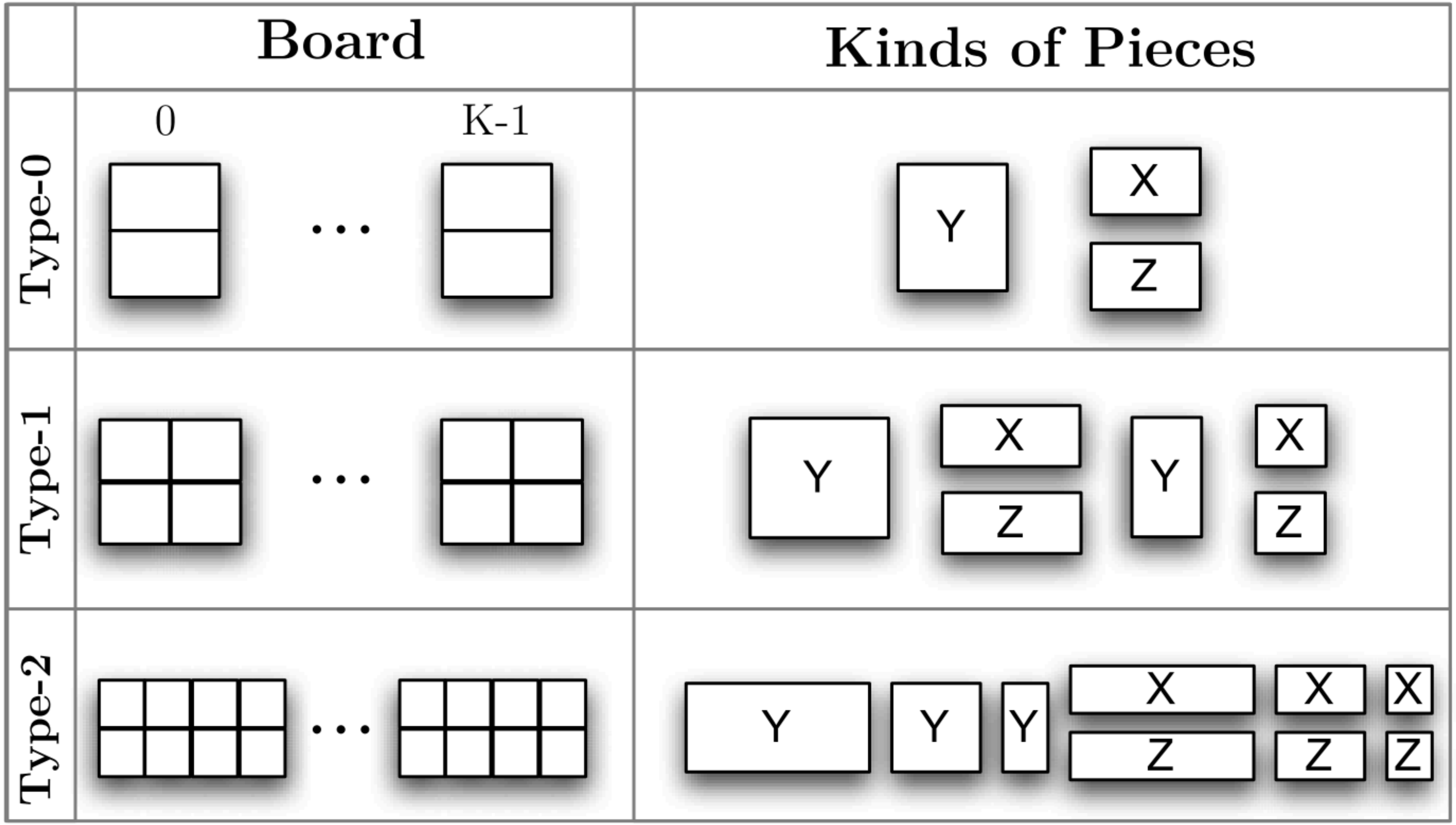


PowerPC
ARM integer registers

SPARC v8
ARM float registers

SPARC v9

Puzzle pieces accepted by boards



Summary

- Graph coloring abstraction: Houston we have a problem
- Puzzle abstraction
- From a program to a collection of puzzles
- Solve puzzles
- From solved puzzles to assembly code

From a program to puzzle pieces

1. Convert a program into an *elementary program*
    A. Transform code into SSA form
    B. Transform A into SSI form
    C. Insert in B parallel copies between every instruction pair

2. Map the elementary program into puzzle pieces

Static Single Assignment (SSA) representation

- A variable is set only by one instruction in the function body

  (myVar1 <- 5)
  (myVar2 <- 7)
  (myVar3 <- 42)

- A static assignment can be executed more than once

SSA and not SSA example

```
float myF (float par1, float par2, float par3){
    return (par1 * par2) + par3; }
```

```
float myF(float par1, float par2, float par3) {
    myVar1 = par1 * par2
    myVar1 = myVar1 + par3
    ret myVar1}
```

NOT SSA

```
float myF(float par1, float par2, float par3) {
    myVar1 = par1 * par2
    myVar2 = myVar1 + par3
    ret myVar2}
```

SSA

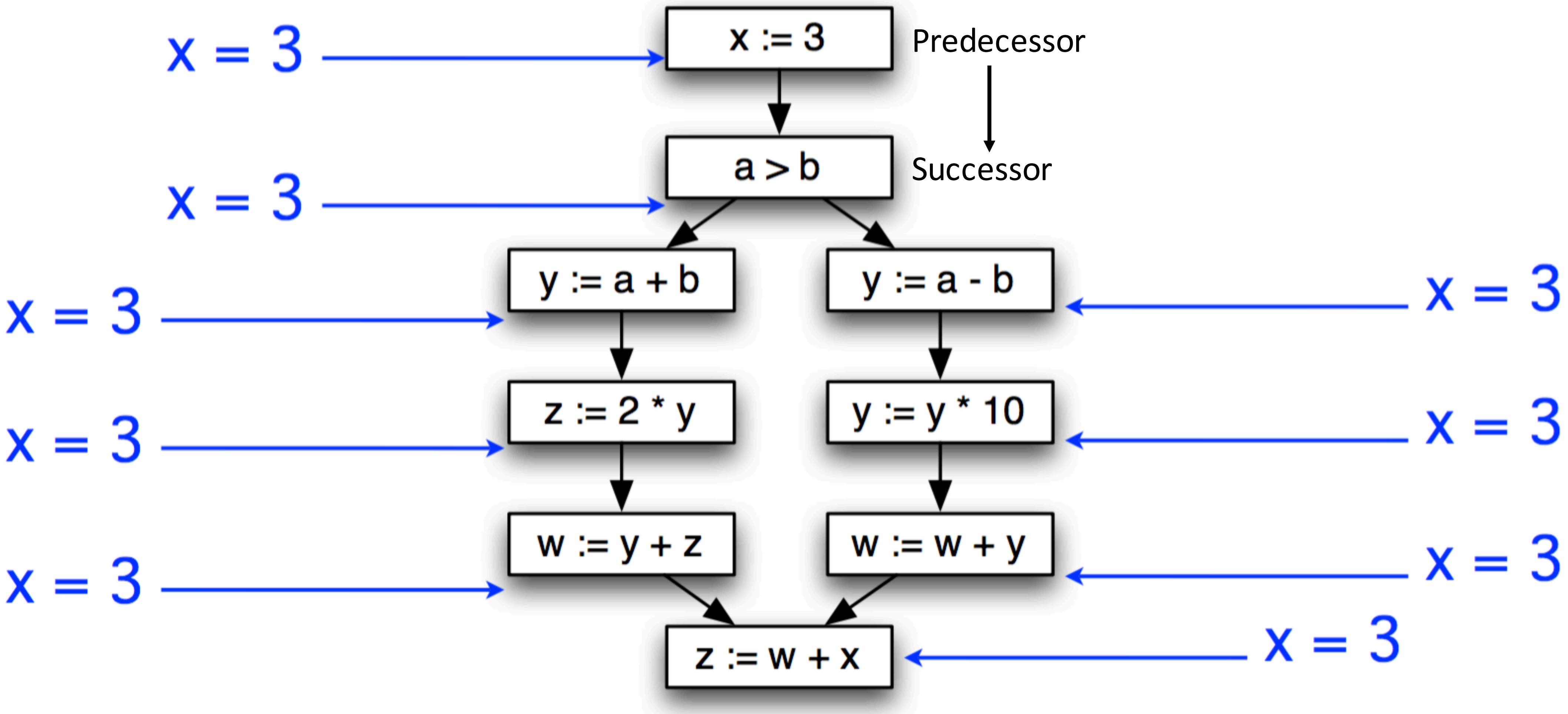
Motivation for SSA

- Code analysis needs to represent facts at every program point

```
float myF(float par1, float par2, float par3) {
    myVar1 = par1 * par2
    myVar2 = myVar1 + par3
    ret myVar2 }
```

- What if
  - There are a lot of facts and
    there are a lot of program points?
  - potentially takes a lot of space/time

Example

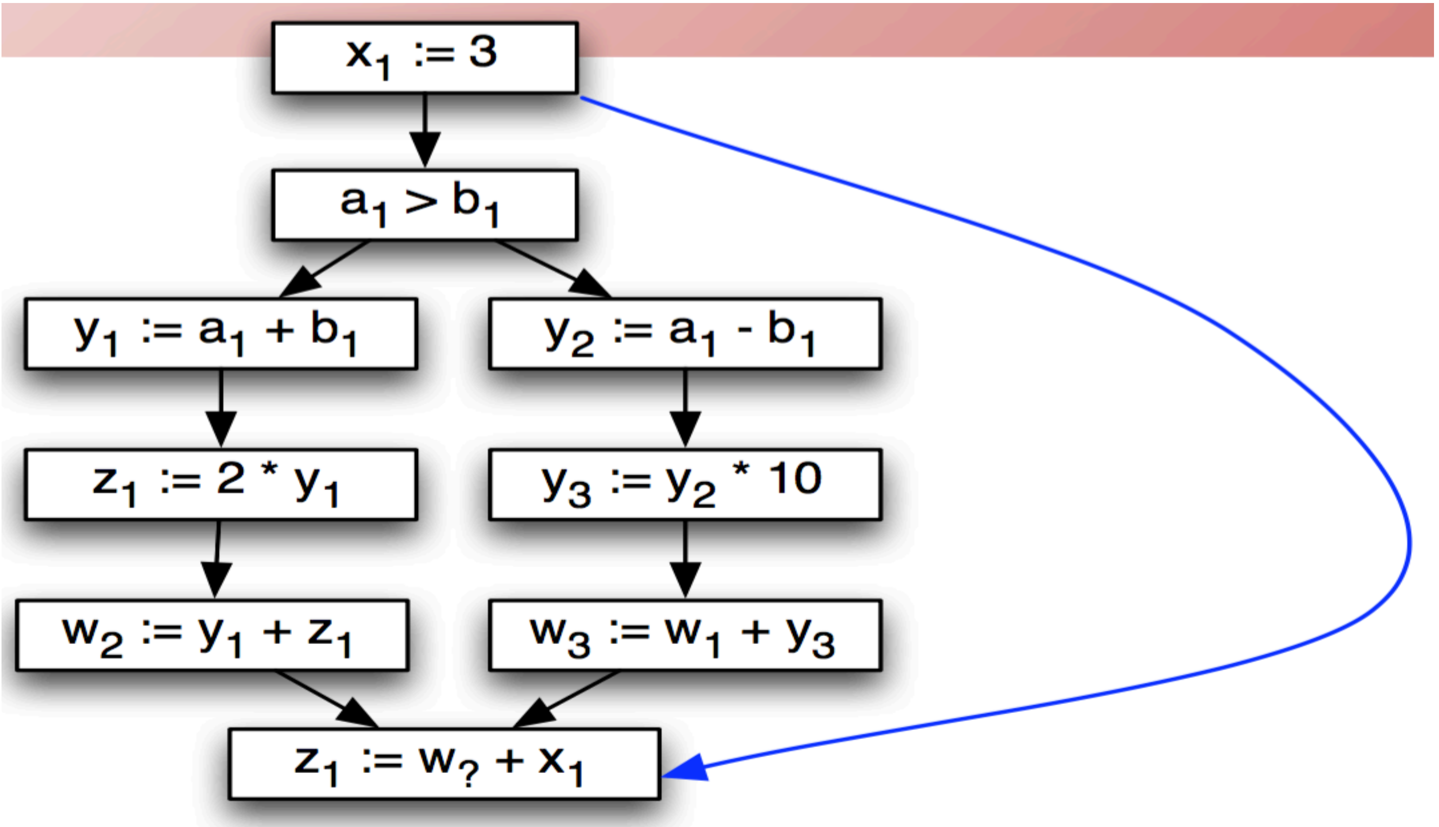


x = 3
x := 3
Predecessor
Successor
x = 3
a > b
x = 3
y := a + b
y := a - b
x = 3
x = 3
z := 2 * y
y := y * 10
x = 3
x = 3
w := y + z
w := w + y
x = 3
z := w + x
x = 3

Static Single Assignment (SSA)

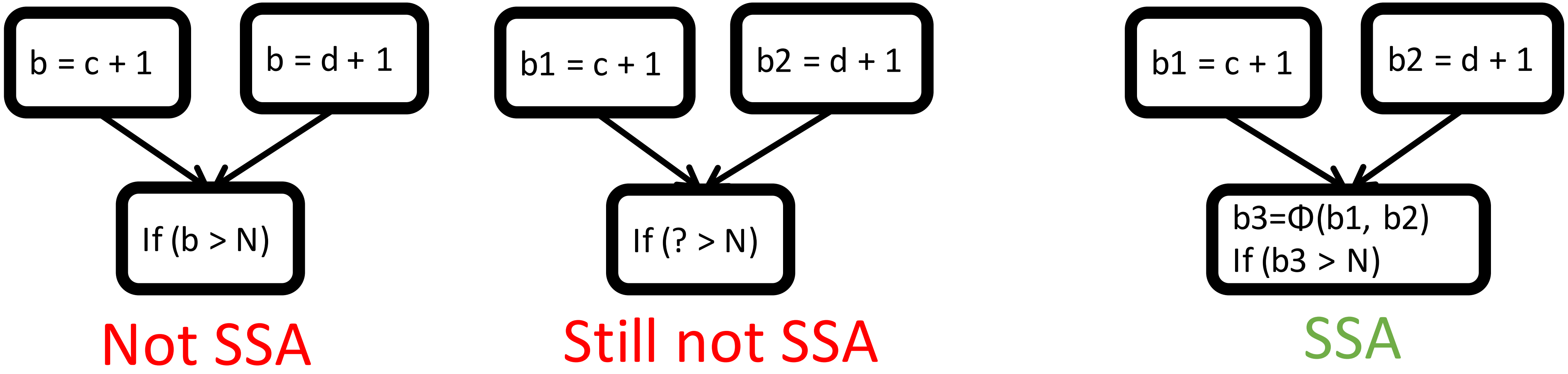
Add **SSA edges** from definitions to uses

- No intervening statements define variable
- Safe to propagate facts about x only along SSA edges



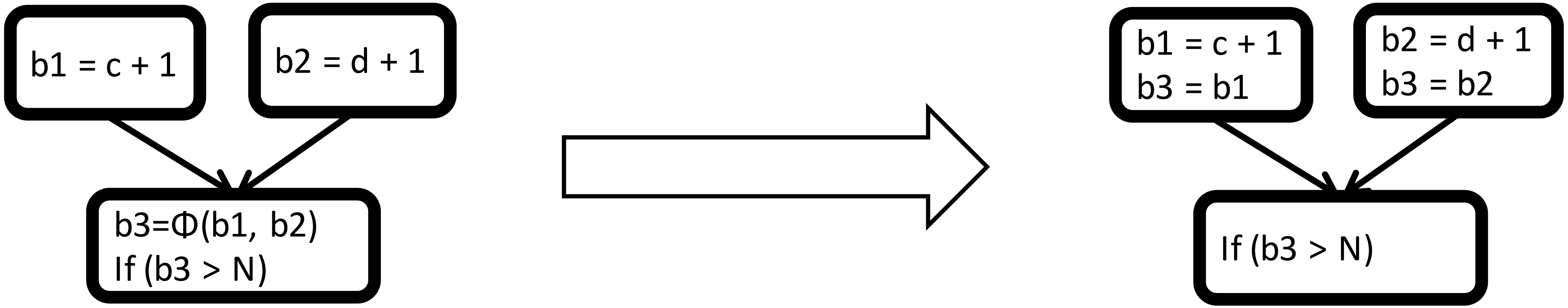
What about joins?

- Add Φ functions/nodes to model joins
  - One argument for each incoming branch
- Operationally
  - selects one of the arguments based on how control flow reach this node
- At code generation time, need to eliminate Φ nodes



Eliminating Φ

- Basic idea: Φ represents facts that value of join may come from different paths
  - So just set along each possible path



Not SSA

Eliminating Φ in practice

- Copies performed at Φ may not be useful
- Joined value may not be used later in the program

  (So why leave it in?)

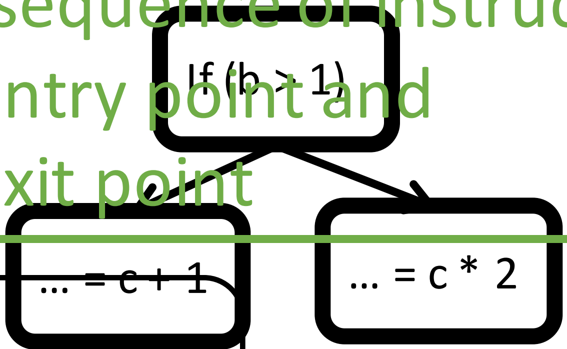
- Use dead code elimination to kill useless Φs
- Register allocation maps the variables to machine registers

Static Single Information (SSI) form

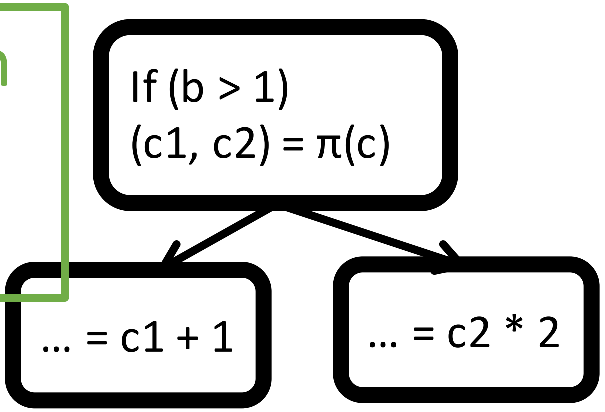
In a program in SSI form:

- Every *basic block* ends with a π-function
  that renames the variables that are live going out of the basic block

Basic block: sequence of instructions with
- **only one** entry point and
- **only one** exit point



Not SSI

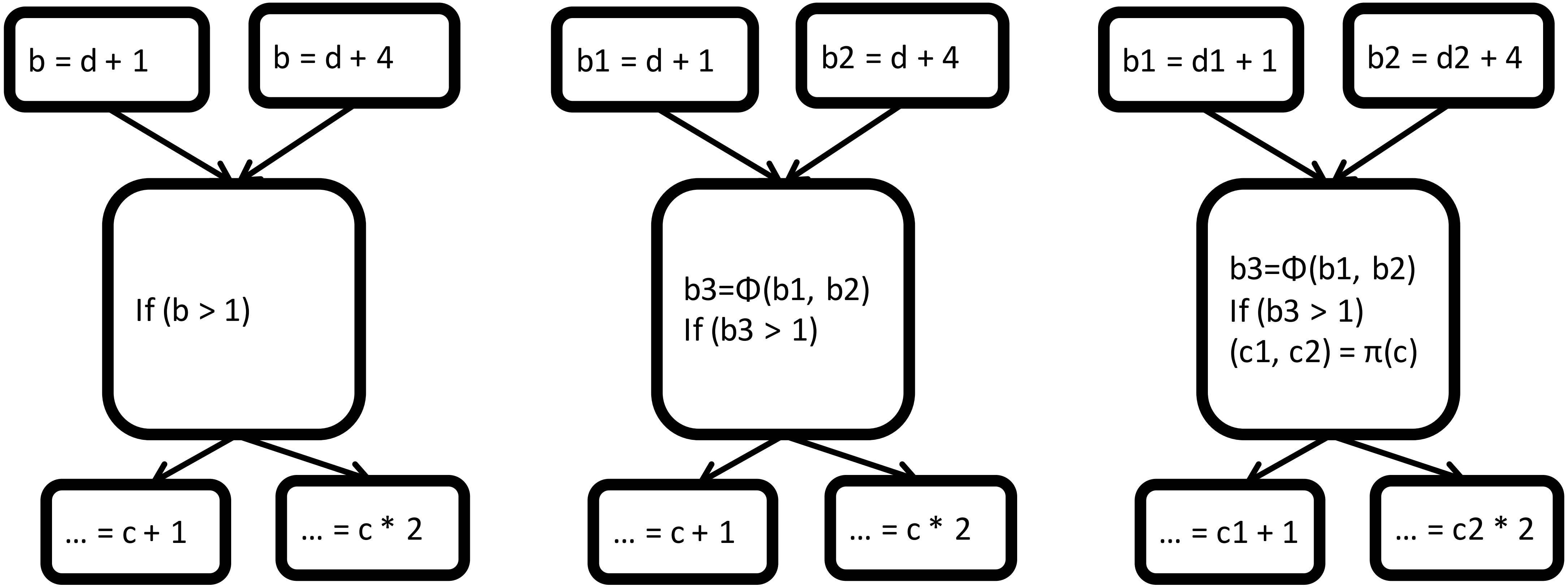


SSI

BB1
:L1
(myVar1 <- 5)
(myVar2 += myVar1)
(cjump myVar1 = myVar2 :L2 :L1)

BB2
:L2
(c <- 10)

SSA and SSI code



Not SSA and not SSI

SSA but not SSI

SSA and SSI

Parallel copies

- Rename variables in parallel

```
V = X + Y
Z = A + B
```

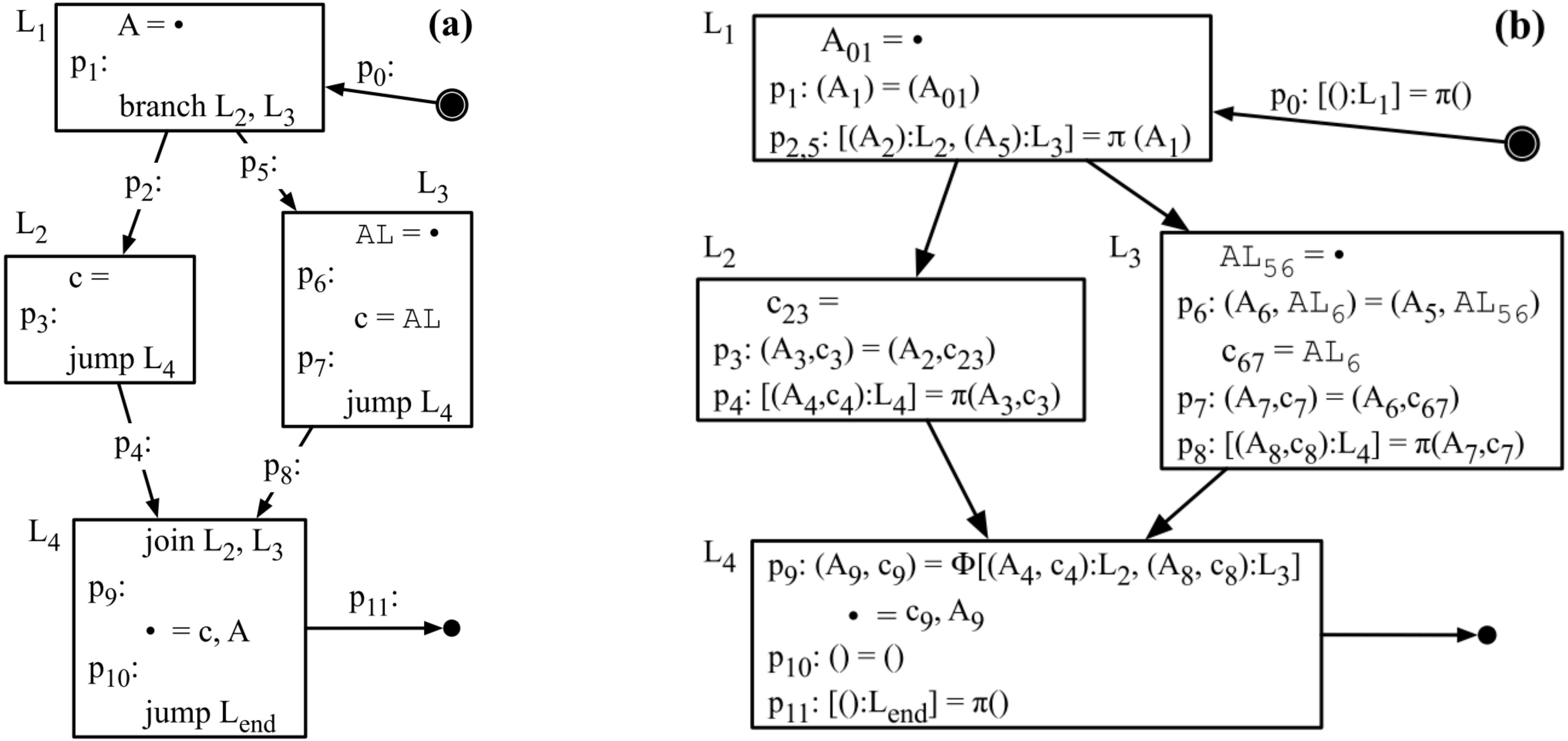
⇒

```
(V1, X1, Y1, Z1, A1, B1) = (V, X, Y, Z, A, B)
V1 = X1 + Y1
(V2, X2, Y2, Z2, A2, B2) = (V1, X1, Y1, Z1, A1, B1)
Z2 = A2 + B2
```

From a program to puzzle pieces

1. Convert a program into an *elementary program*
   A. Transform code into SSA form
   B. Transform A into SSI form
   C. Insert in B parallel copies between every instruction pair

Elementary form: an example

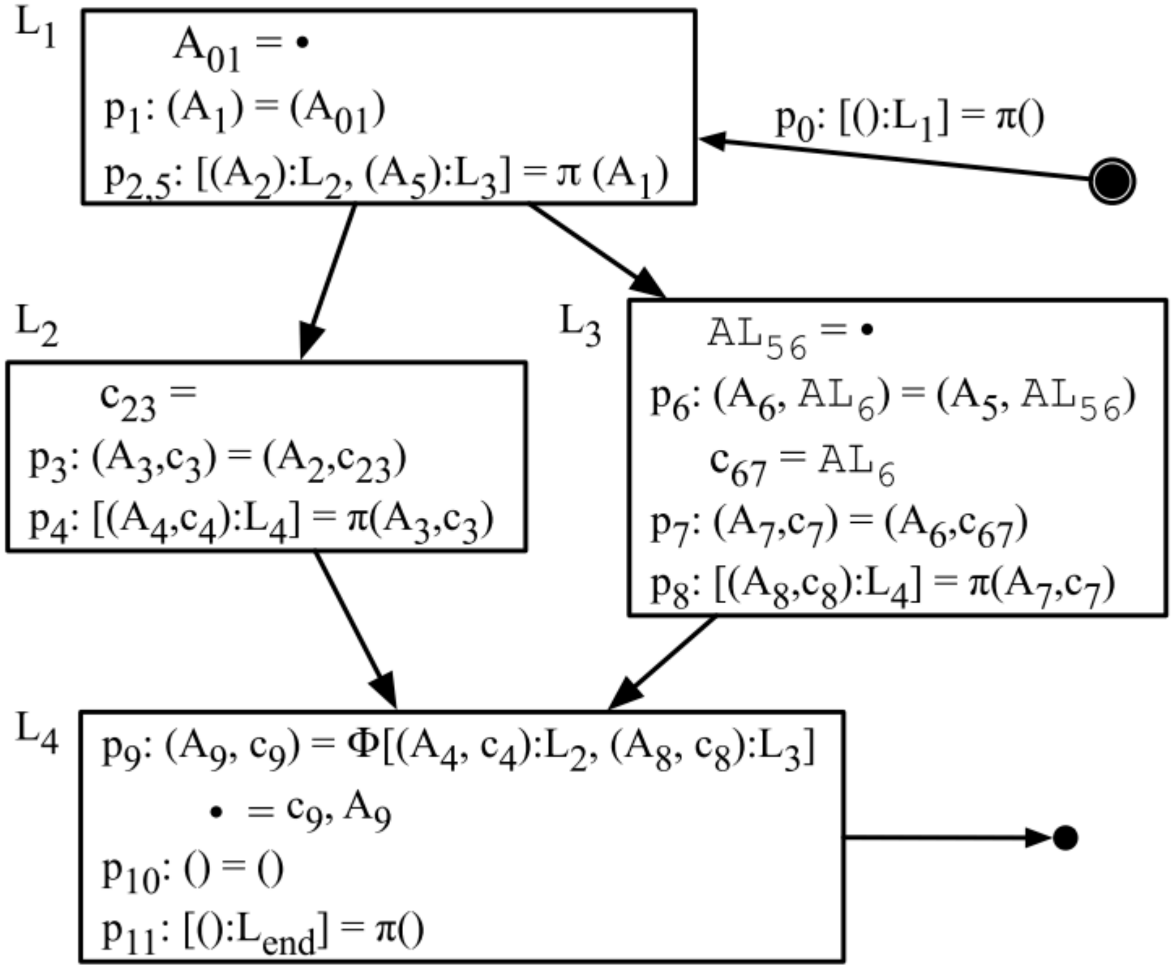


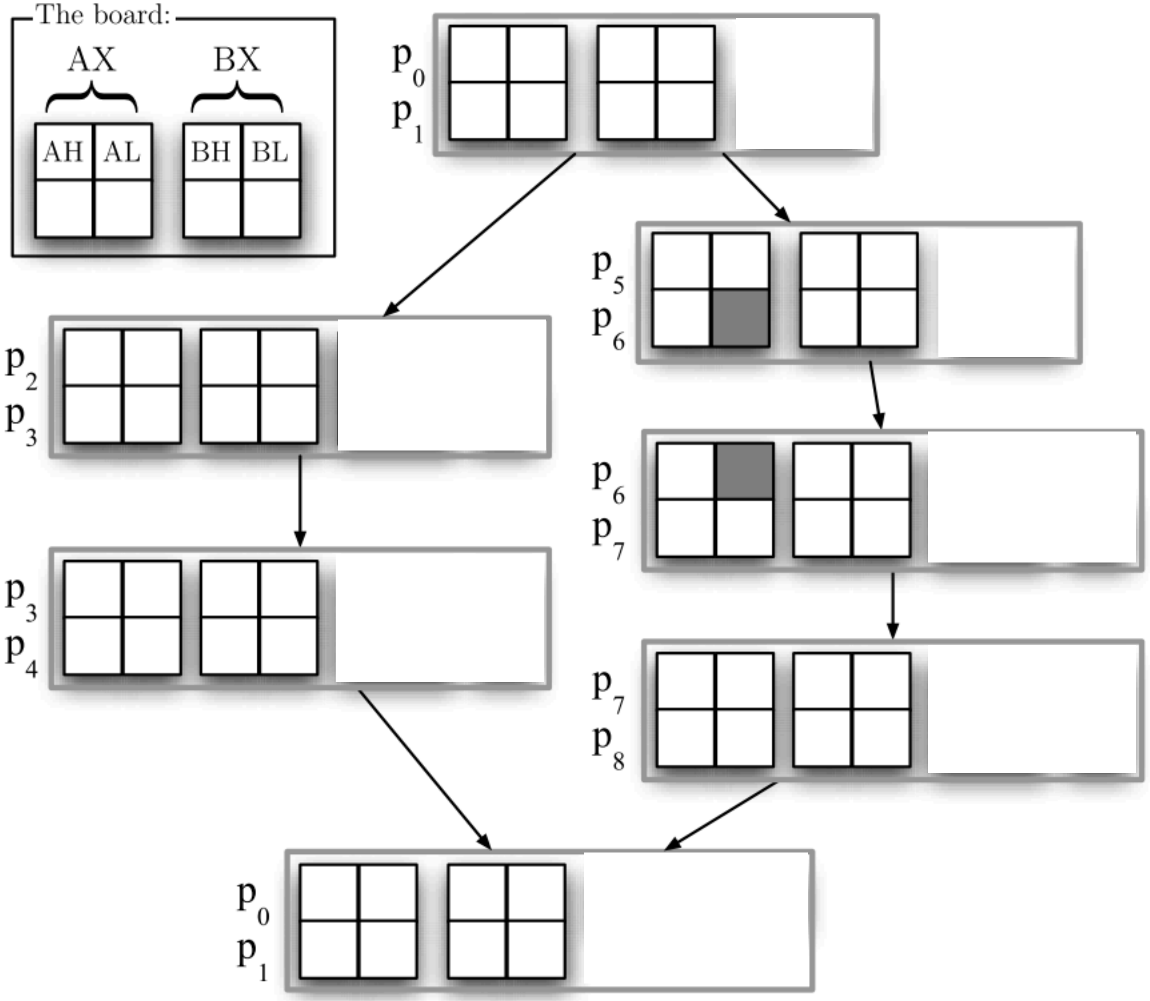
From a program to puzzle pieces

1. Convert a program into an *elementary program*
   A. Transform code into its SSA form
   B. Transform code into its SSI form
   C. Insert parallel copies between every instruction pair

2. Map the elementary program into puzzle pieces

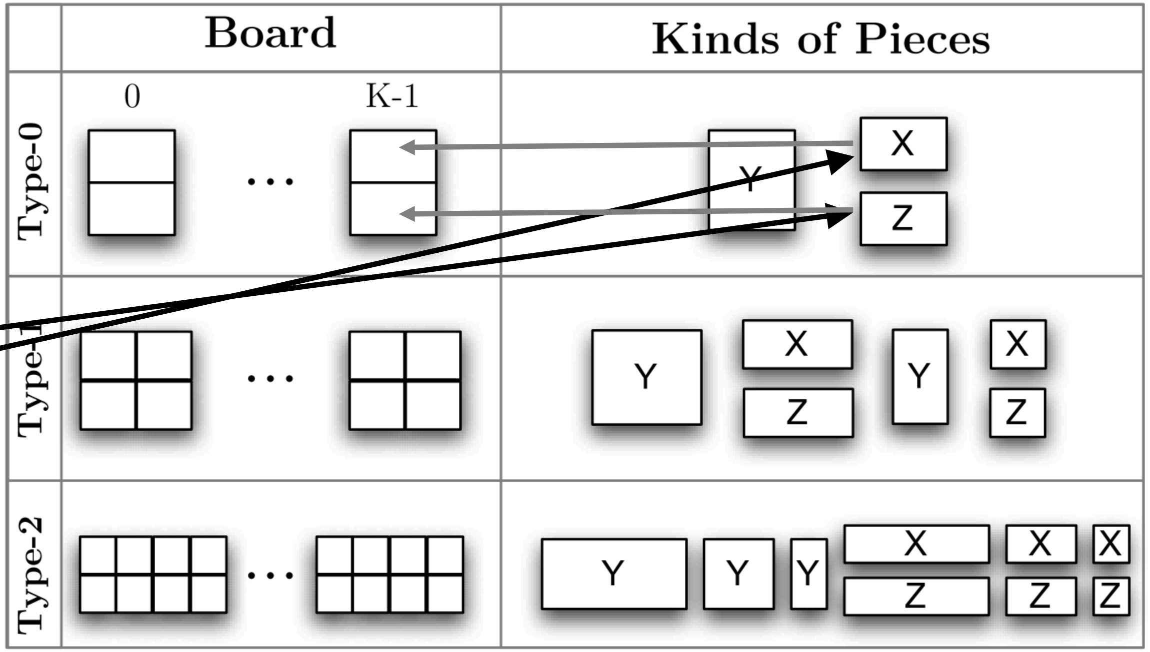
Add puzzle boards





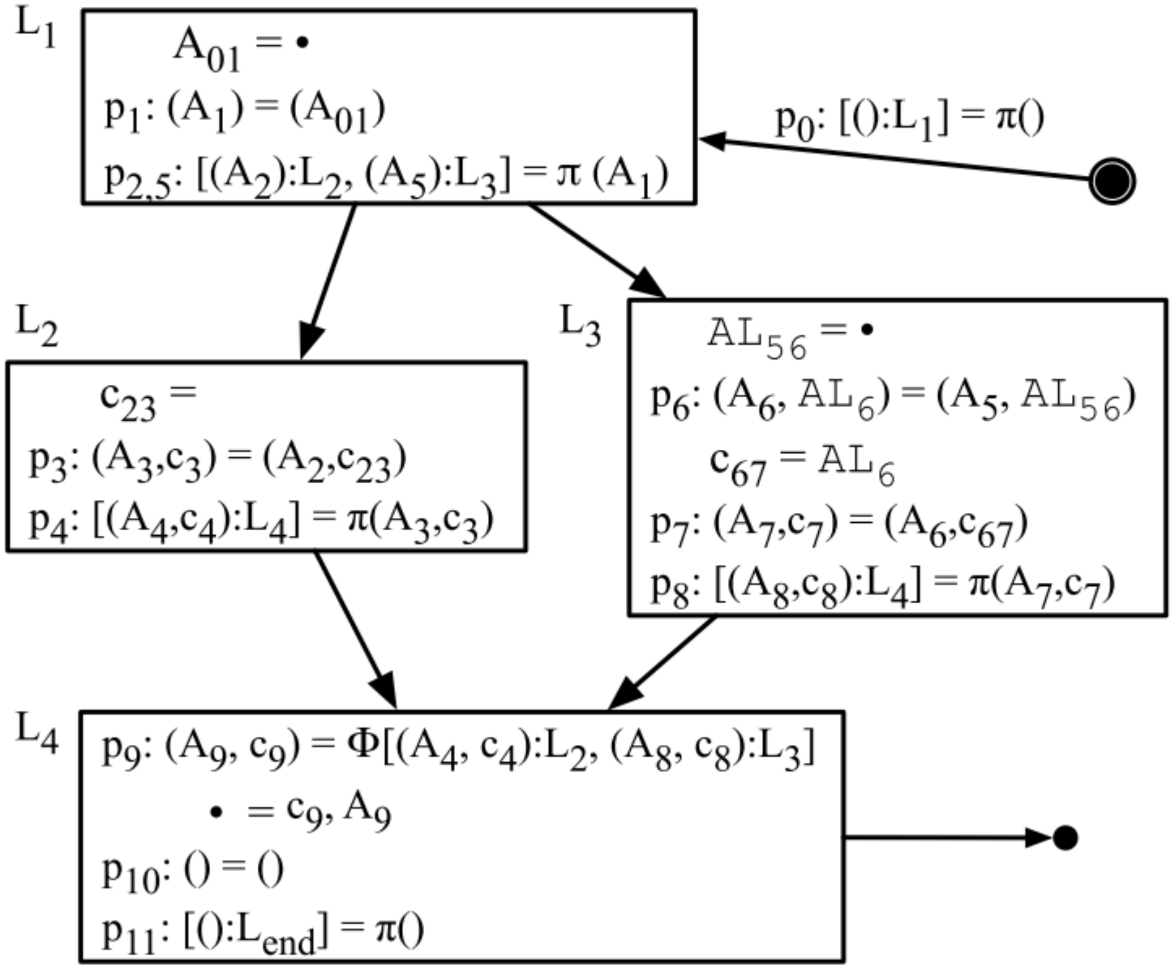
Generating puzzle pieces

- For each instruction i
  - Create one puzzle piece for each live-in and live-out variable
  - If the live range ends at i, then the puzzle piece is X
  - If the live range begins at i, then Z-piece
  - Otherwise Y-piece

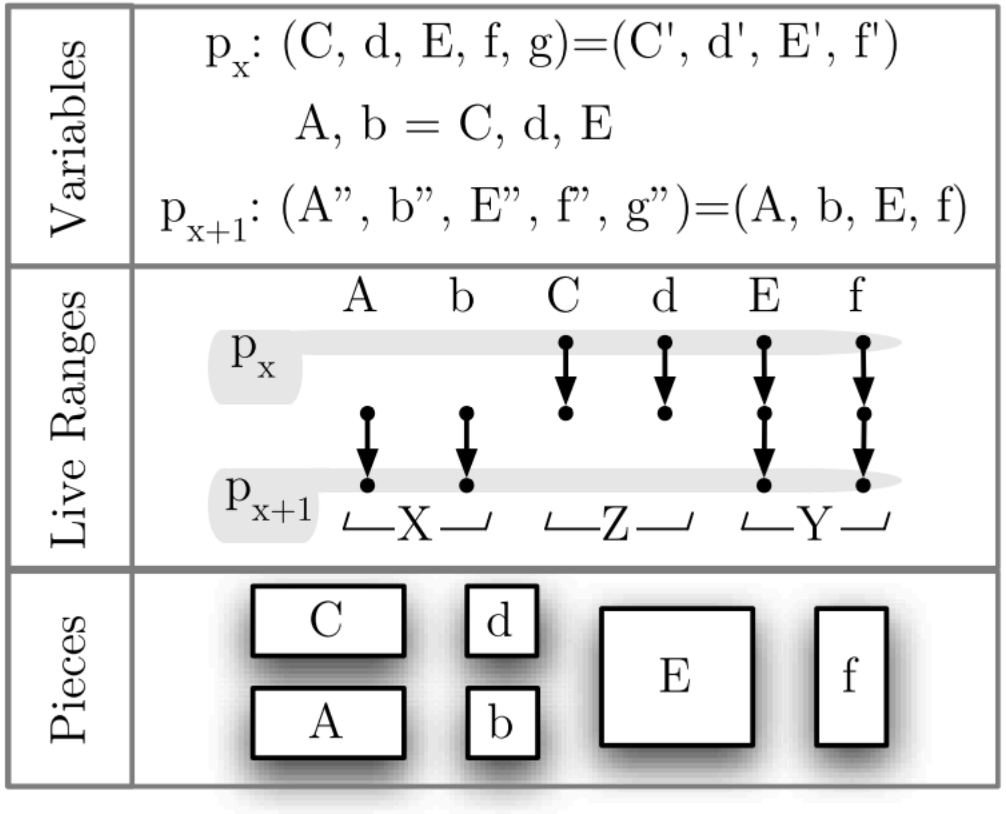


V1 (used later) = V2 (last use) + 3
r10 = r10 + 3

Example

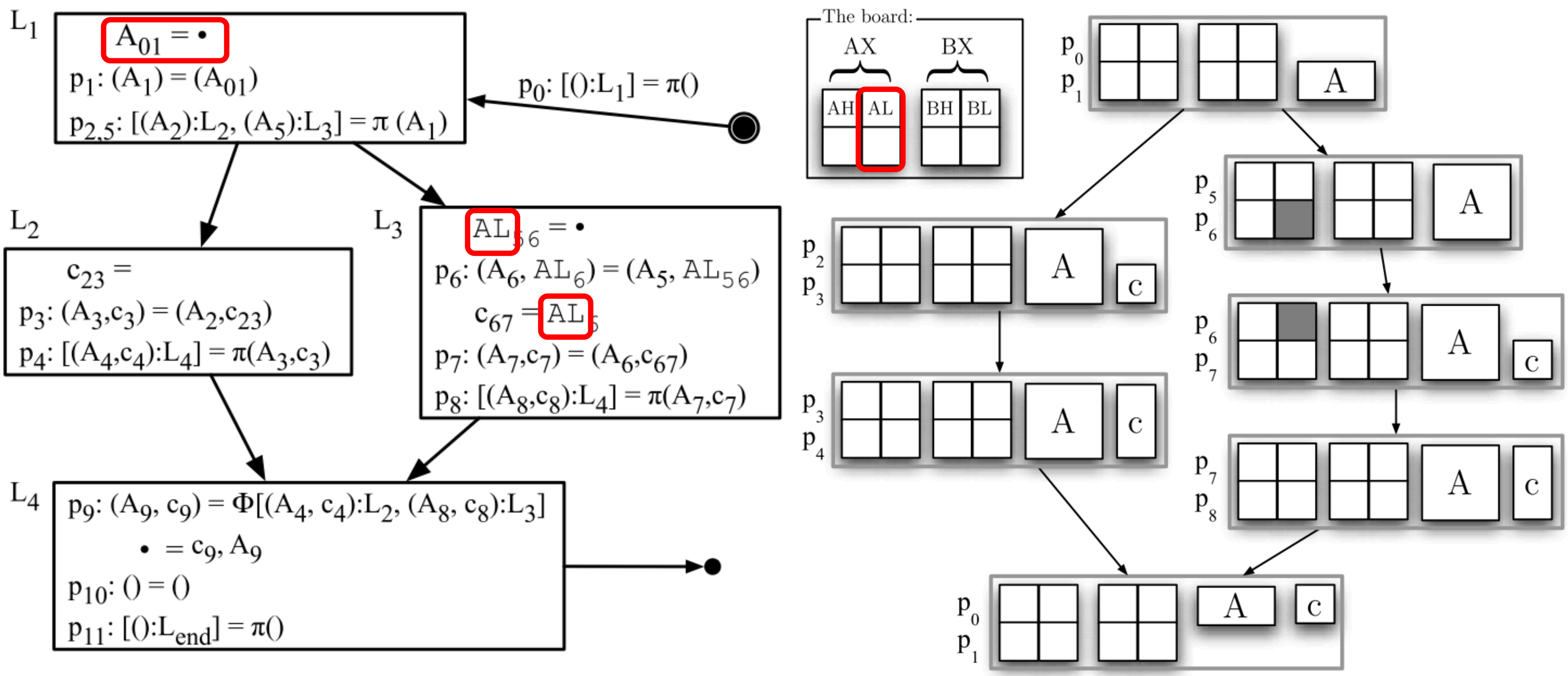


L_1
$A_{01} = \bullet$
p_1: $(A_1) = (A_{01})$
$p_{2,5}$: $[(A_2){:}L_2, (A_5){:}L_3] = \pi(A_1)$
p_0: $[(){:}L_1] = \pi()$
L_2
$c_{23} =$
p_3: $(A_3,c_3) = (A_2,c_{23})$
p_4: $[(A_4,c_4){:}L_4] = \pi(A_3,c_3)$
L_3
$\mathtt{AL}_{56} = \bullet$
p_6: $(A_6, \mathtt{AL}_6) = (A_5, \mathtt{AL}_{56})$
$c_{67} = \mathtt{AL}_6$
p_7: $(A_7,c_7) = (A_6,c_{67})$
p_8: $[(A_8,c_8){:}L_4] = \pi(A_7,c_7)$
L_4
p_9: $(A_9, c_9) = \Phi[(A_4, c_4){:}L_2, (A_8, c_8){:}L_3]$
$\bullet = c_9, A_9$
p_{10}: $() = ()$
p_{11}: $[(){:}L_{end}] = \pi()$



Variables
p_x: (C, d, E, f, g)=(C', d', E', f')
A, b = C, d, E
p_{x+1}: (A”, b”, E”, f”, g”)=(A, b, E, f)
Live Ranges
A b C d E f
p_x
p_{x+1}
X
Z
Y
Pieces
C
d
E
f
A
b

Example



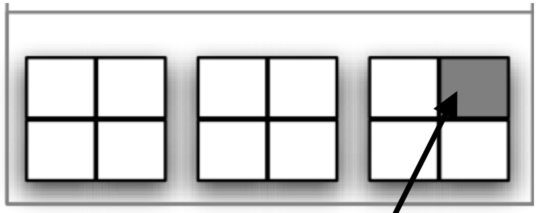
L_1
$A_{01} = \bullet$
$p_1: (A_1) = (A_{01})$
$p_{2,5}: [(A_2):L_2, (A_5):L_3] = \pi\,(A_1)$
$p_0: [():L_1] = \pi()$
L_2
$c_{23} =$
$p_3: (A_3,c_3) = (A_2,c_{23})$
$p_4: [(A_4,c_4):L_4] = \pi(A_3,c_3)$
L_3
$AL_{56} = \bullet$
$p_6: (A_6, AL_6) = (A_5, AL_{56})$
$c_{67} = AL_6$
$p_7: (A_7,c_7) = (A_6,c_{67})$
$p_8: [(A_8,c_8):L_4] = \pi(A_7,c_7)$
L_4
$p_9: (A_9, c_9) = \Phi[(A_4, c_4):L_2, (A_8, c_8):L_3]$
$\bullet = c_9, A_9$
$p_{10}: () = ()$
$p_{11}: [():L_{end}] = \pi()$
The board:
AX
BX
AH
AL
BH
BL
p_0
p_1
A
p_5
p_6
A
p_2
p_3
A
c
p_6
p_7
A
c
p_3
p_4
A
c
p_7
p_8
A
c
p_0
p_1
A
c

Summary

- Graph coloring abstraction: Houston we have a problem
- Puzzle abstraction
- From a program to a collection of puzzles
- Solve puzzles
- From solved puzzles to assembly code

Solving type 1 puzzles

- Approach proposed: complete one area at a time
- For each area:
    - Pad a puzzle with size-1 X- and Z-pieces until the area of puzzle pieces == board



Board with 1 pre-assigned piece

Padding

- Solve the puzzle

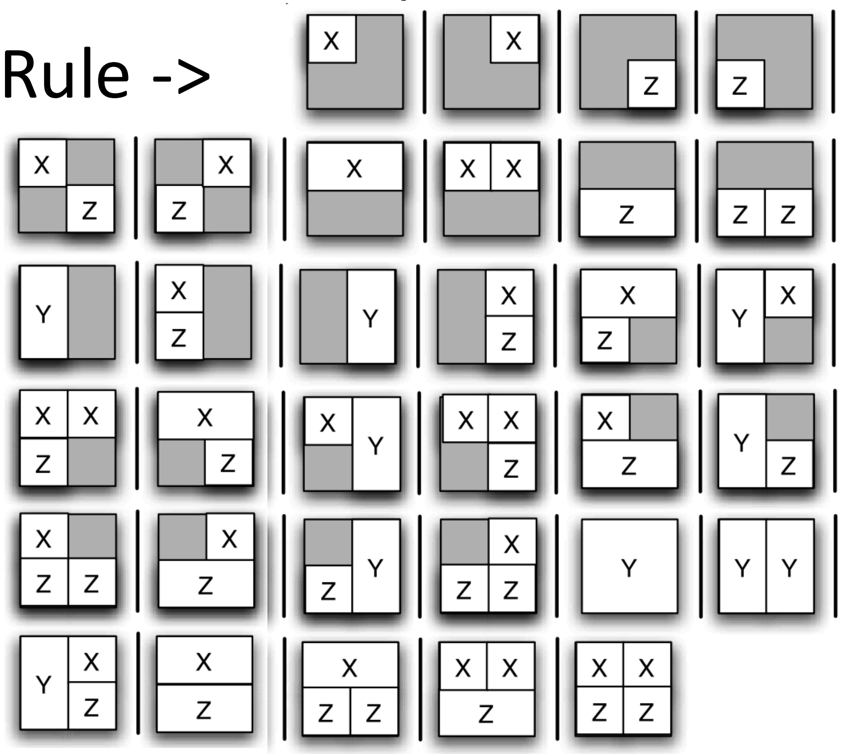
Solving type 1 puzzles: a visual language

Puzzle solver -> Statement+

Statement -> Rule | Condition

Condition -> (Rule : Statement)

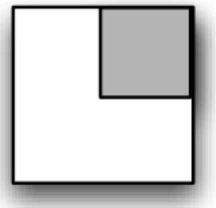
Rule ->



- Rule = how to complete an area
- Rule composed by
  pattern:
  what needs to be already filled
  (match/not-match an area)

  strategy:
  what type of pieces to add and where
- A rule *r* succeeds in an area *a* iff
  i. *r* matches *a*
  ii. pieces of the strategy of *r* are available

Area *a*



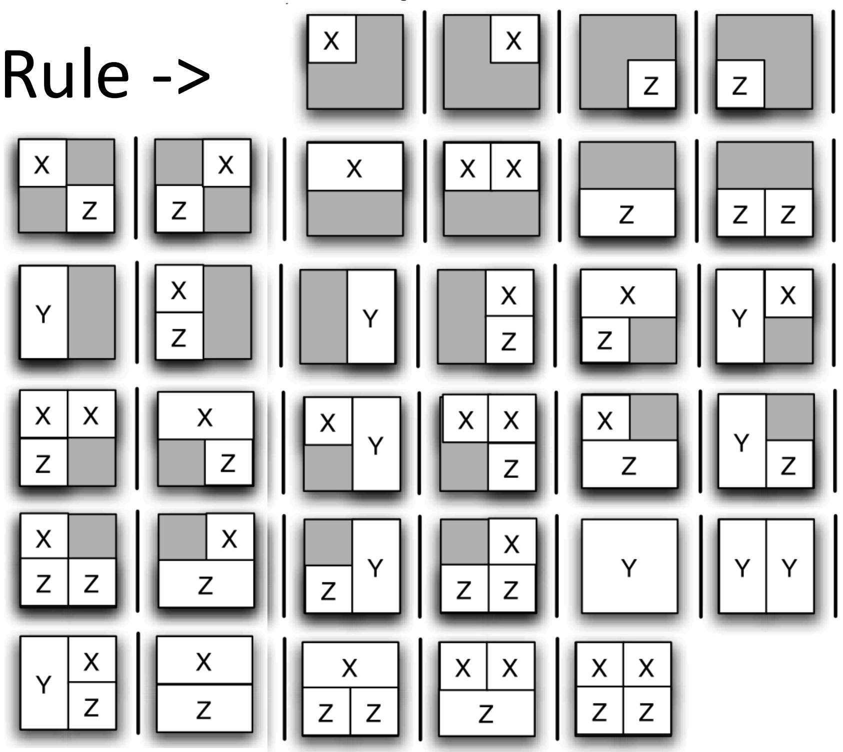
Solving type 1 puzzles: a visual language

Puzzle solver -> Statement+

Statement -> Rule | Condition

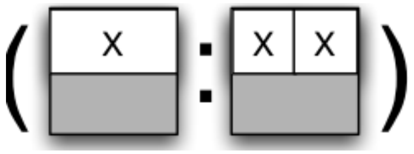
Condition -> (Rule : Statement)

Rule ->



Puzzle solver success

- A program succeeds iff all statements succeeds
- A rule *r* succeeds in an area *a* iff
    i. *r* matches *a*
    ii. pieces of the strategy of *r* are available
- A condition (*r* : *s)* succeeds iff
    - r succeeds or
    - s succeeds



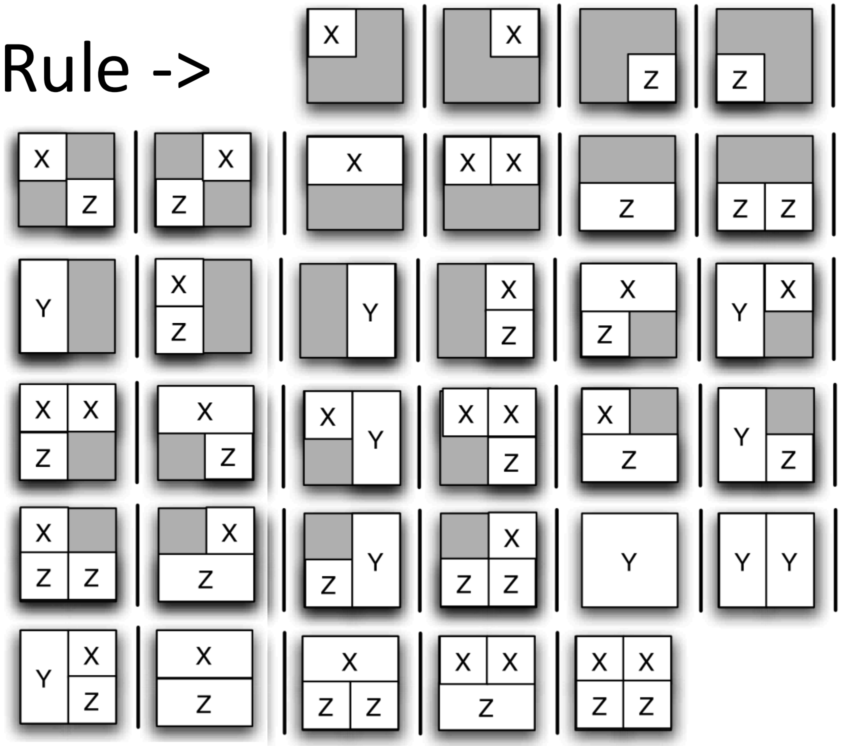
Solving type 1 puzzles: a visual language

Puzzle solver -> Statement+

Statement -> Rule | Condition

Condition -> (Rule : Statement)

Rule ->

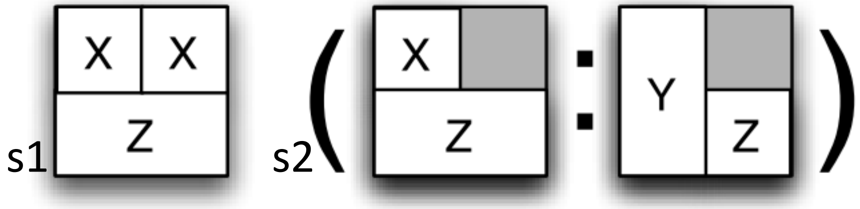


Puzzle solver execution

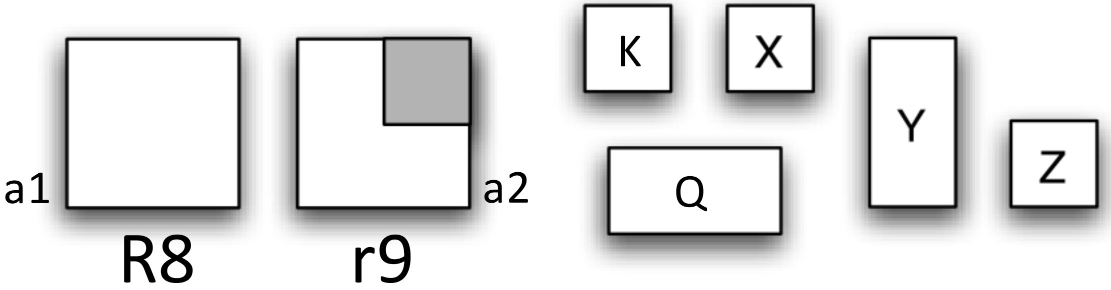
- For each statement *s1, …, sn*
  - For each area *a* such that the pattern of *si* matches *a*
    - Apply *si* to *a*
    - If *si* fails, terminate and report failure

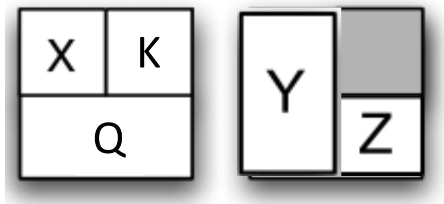
Program execution: an example

- A puzzle solver



- Puzzle





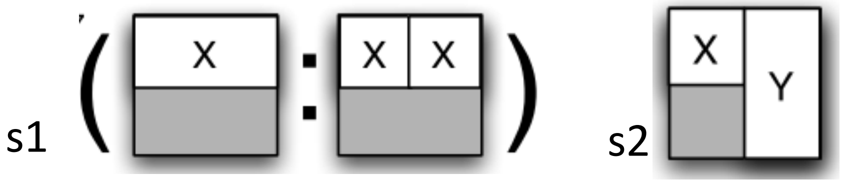
Puzzle solved!

1. s1 matches a1 only
2. Apply s1 to a1 succeeds and returns this puzzle

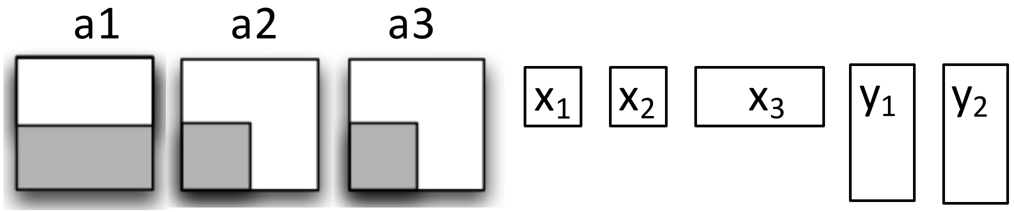
3. s2 matches a2 only
4. Apply s2 to a2
   A. Apply first rule of s2: fails
   B. Apply second rule of s2: success

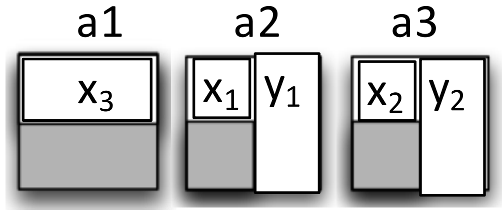
Program execution: another example

- A puzzle solver



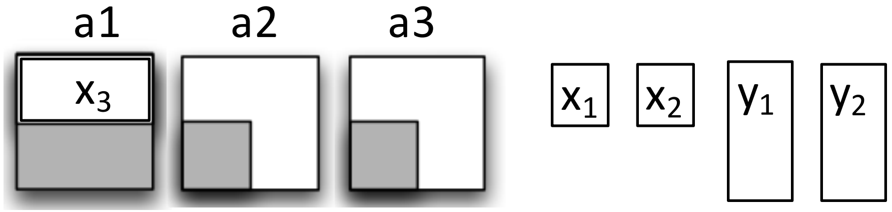
- Puzzle



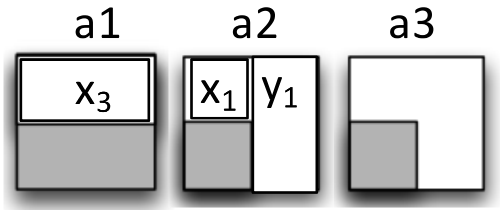


Puzzle solved!

1. s1 matches a1 only
2. Apply s1 to a1
   A. Apply first rule of s1: success



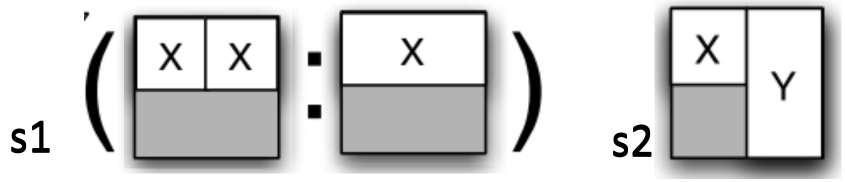
3. s2 matches a2 and a3
4. Apply s2 to a2



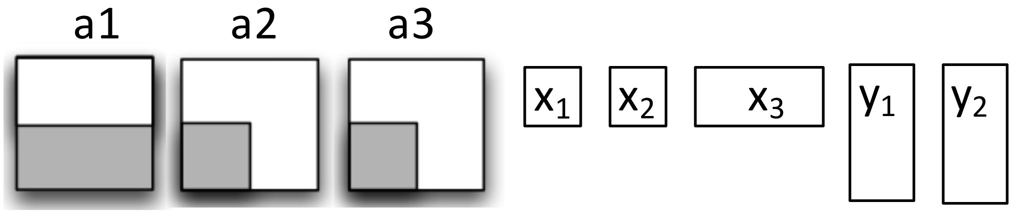
5. Apply s2 to a3

Program execution: yet another example

- A puzzle solver

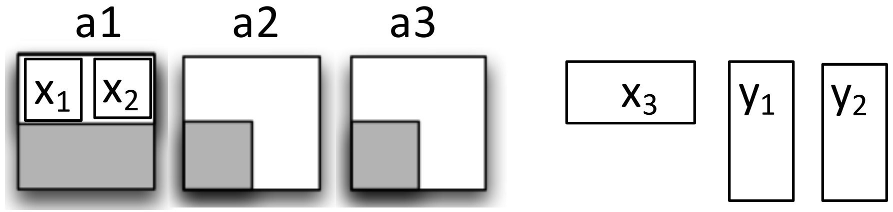


- Puzzle



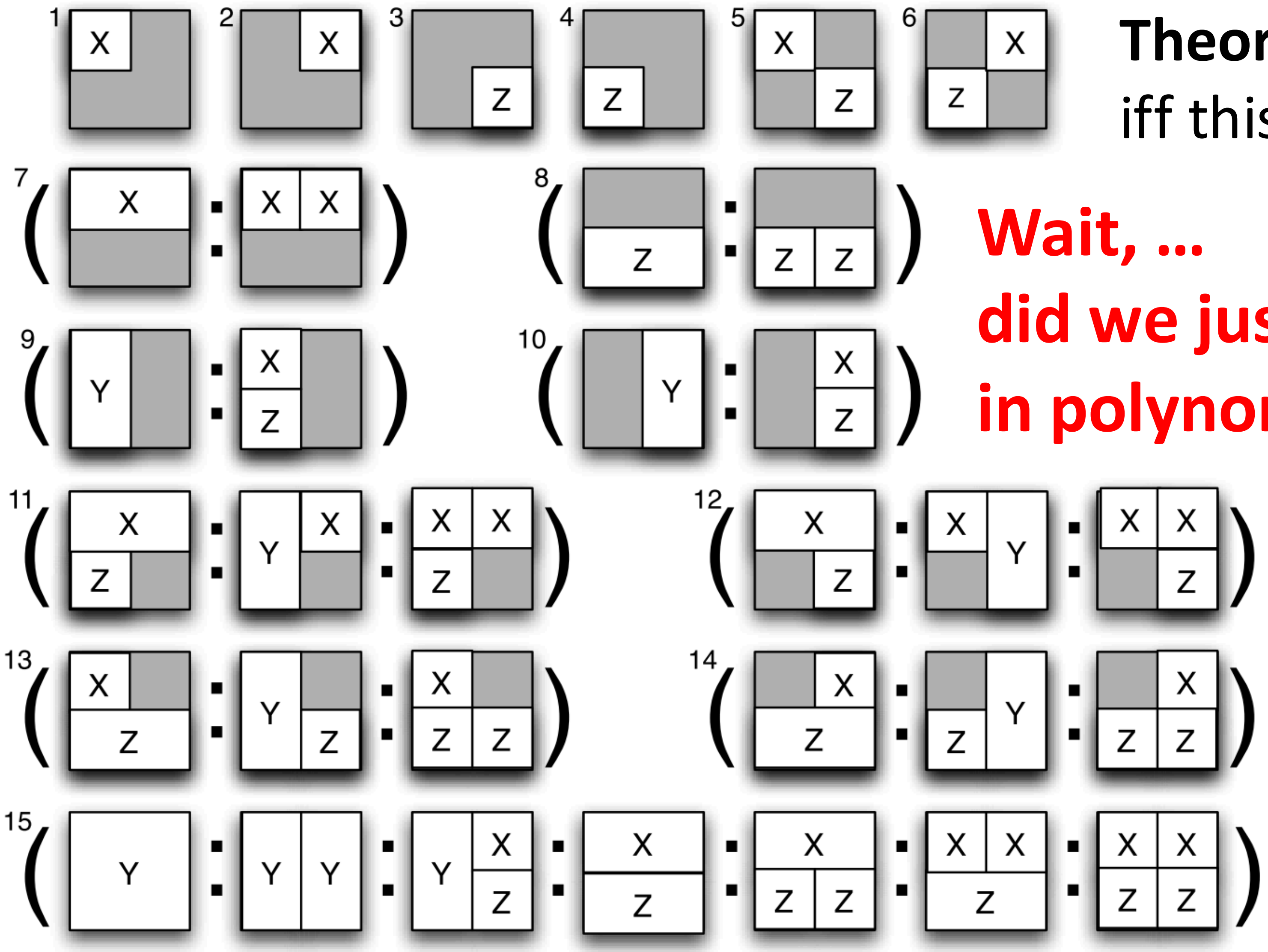
Finding the right puzzle solver is the key!

1. s1 matches a1 only
2. Apply s1 to a1
   A. Apply first rule of s1: success



3. s2 matches a2 and a3
4. Apply s2 to a2: fail
   No 1-size x pieces,
   we used them all in s1

Solution to solve type 1 puzzles



Theorem: a type-1 area is solvable iff this program succeeds

Wait, …
did we just solve a NP problem in polynomial time?

Register allocation:
complete all areas

Simplified problem solved:
complete one area
at a time

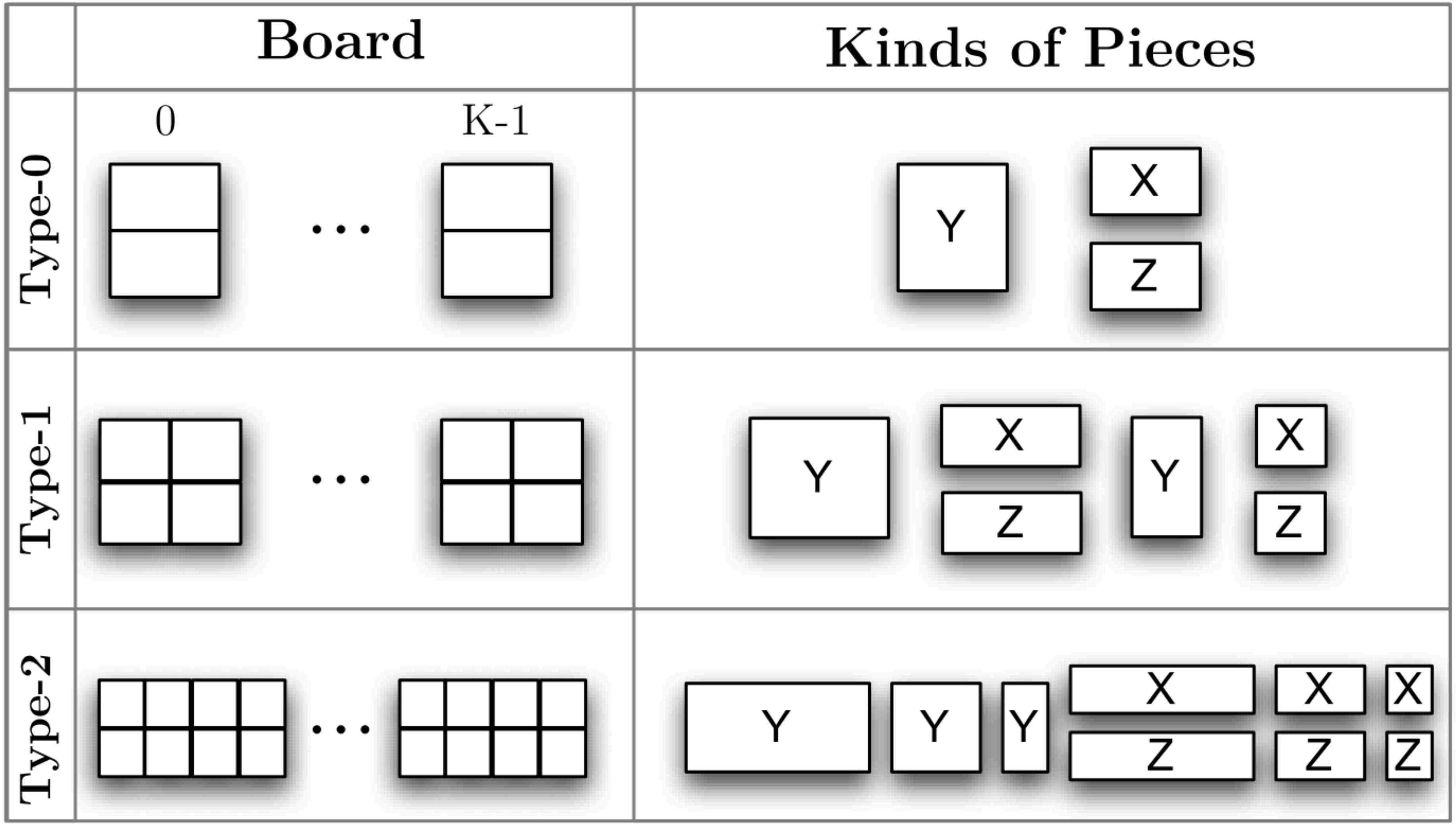
Solution to solve type 1 puzzles: complexity

Corollary 3.
Spill-free register allocation with pre-coloring for an elementary program P and K registers is solvable in O(|P| x K) time

For one instruction in P:

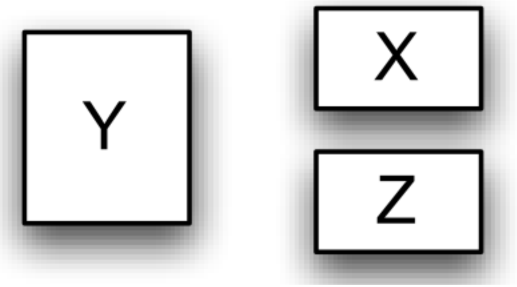
- Application of a rule to an area: O(1)
- A puzzle solver O(1) rules on each area of a board
- Execution of a puzzle solver on a board with K areas takes O(K) time

Solving type 0 puzzles



Solving type 0 puzzles: algorithm

- Place all Y-pieces on the board



- Place all X- and Z-pieces on the board

Spilling

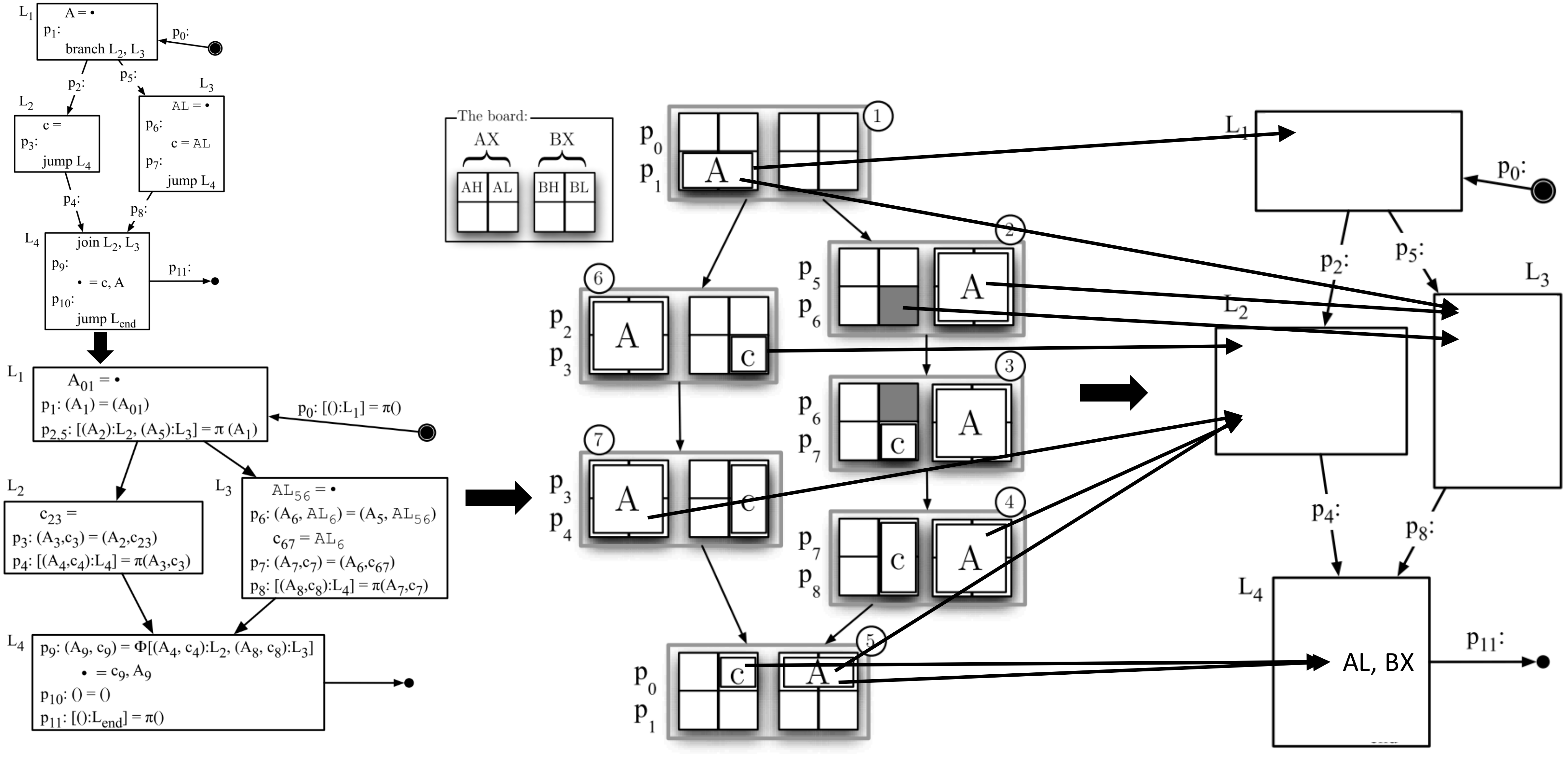
- If the algorithm to solve a puzzles fails
  i.e., the need for registers exceeds the number of available registers
  => spill

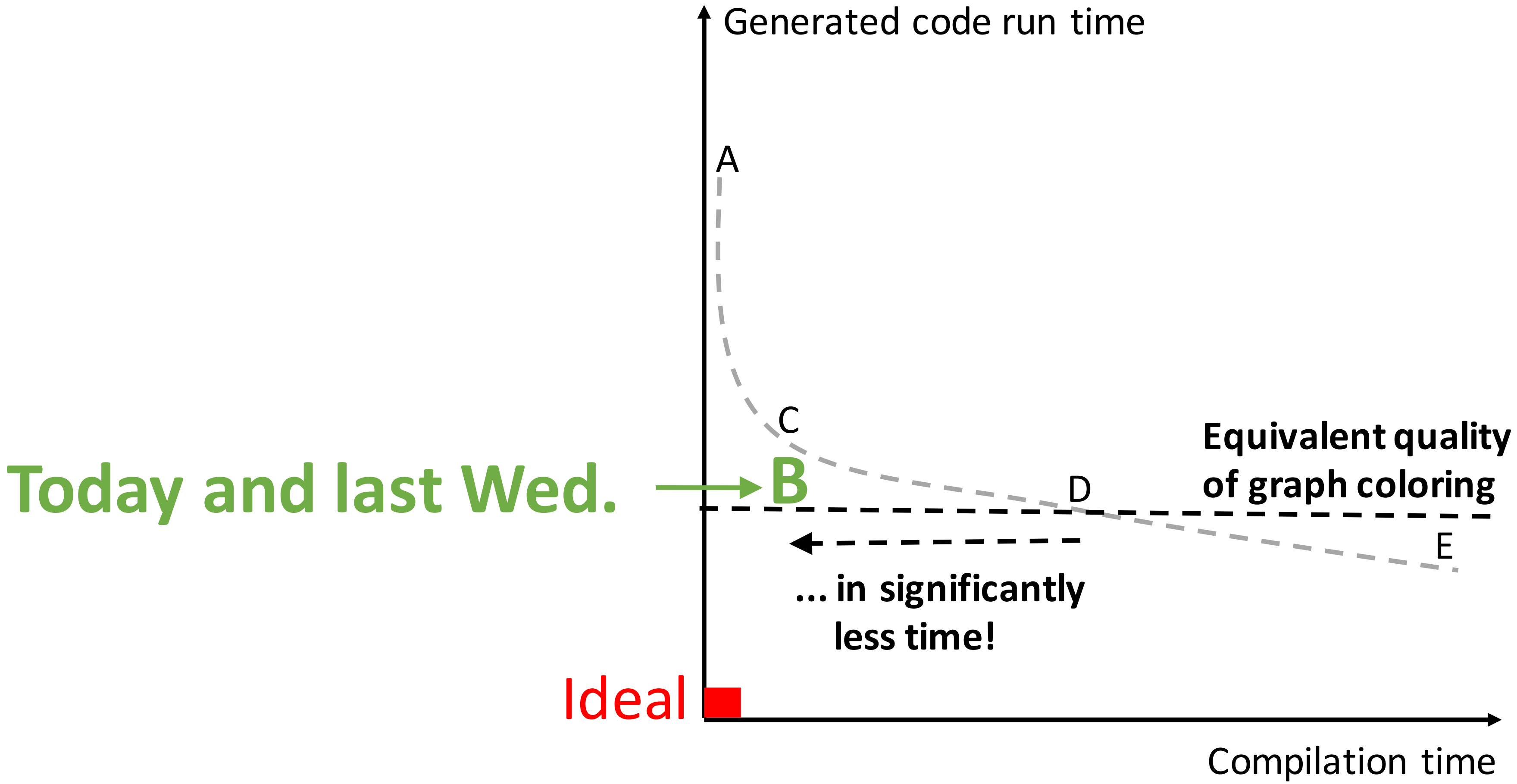
- **Observation**: translating a program into its elementary form creates families of variables, one per original variable
- **To spill**:
  - Choose a variable *v* to spill from the original program
  - Spill all variables in the elementary form
    that belong to the same family of *v*

Summary

- Graph coloring abstraction: Houston we have a problem
- Puzzle abstraction
- From a program to a collection of puzzles
- Solve puzzles
- From solved puzzles to assembly code

From solved puzzles to assembly code




Generated code run time
A
C
Today and last Wed.
B
D
Equivalent quality
of graph coloring
E
... in significantly
less time!
Ideal
Compilation time

Thank you!