

Simone Campanoni
simone.campanoni@northwestern.edu

Puzzle solving

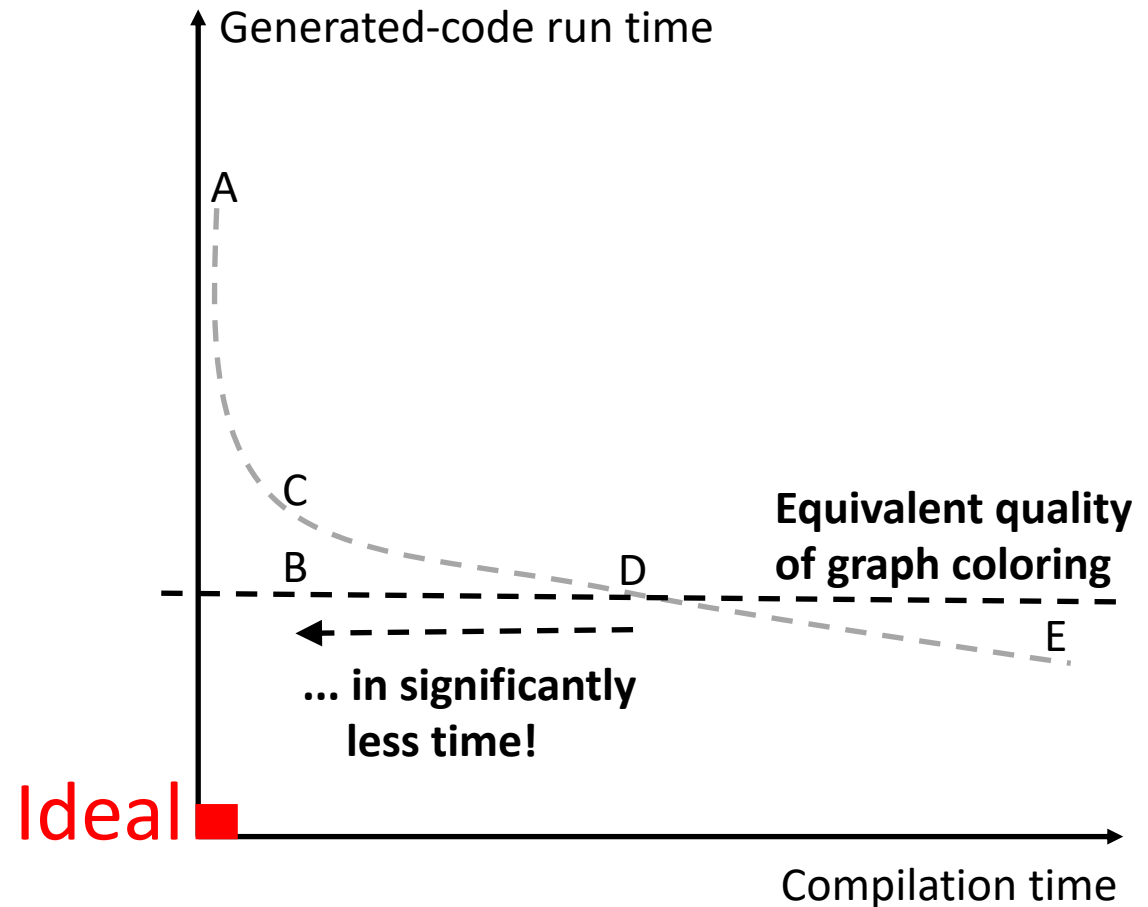


Materials

- 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

Register Allocation

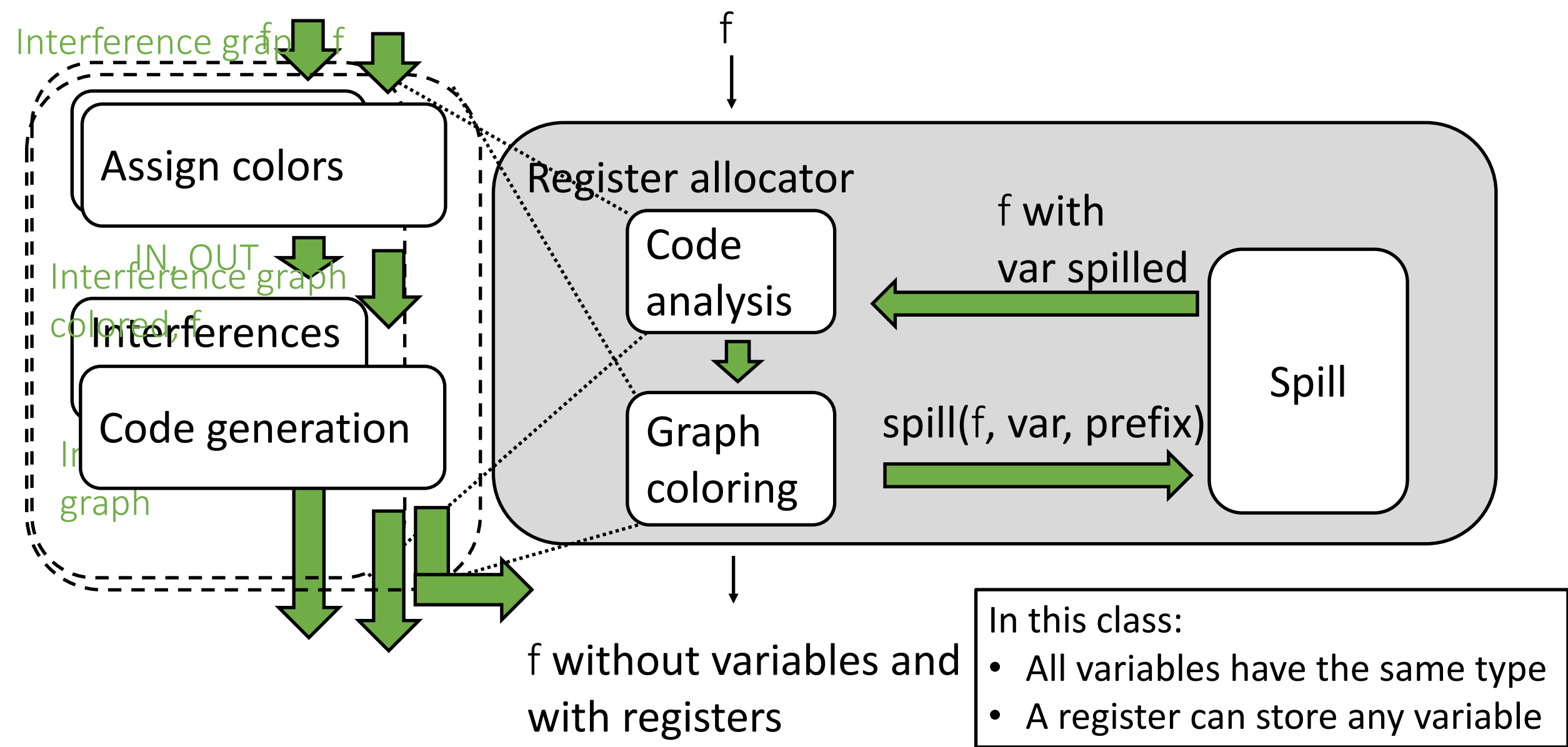
- A. Spill all variables
- B. Puzzle solving
- C. Linear scan
- D. Graph coloring
- E. Integer linear programming



Outline

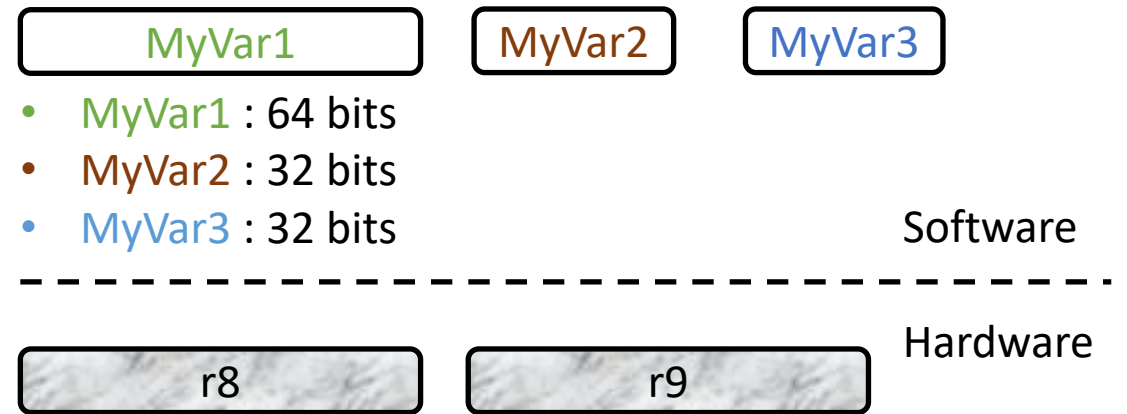
- Register allocation abstractions
- From a program to a collection of puzzles
- Solve puzzles
- From solved puzzles to assembly code

A graph-coloring register allocator



Graph coloring abstraction: a problem

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



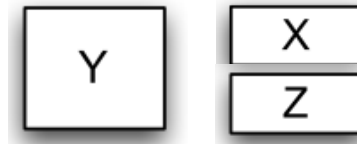
Register aliasing →

- r8 can store either one 64-bit value or two 32-bit values
- r9 can store 64-bit values

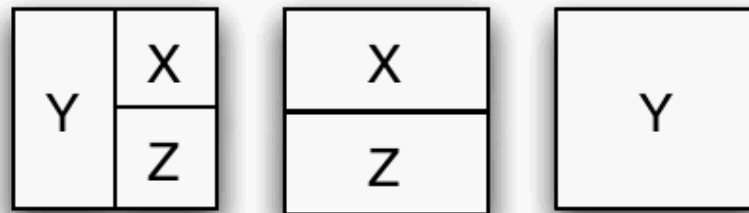
**Can this be obtained
by the graph-coloring algorithm
you learned in this class?**

Puzzle Abstraction

- Puzzle = board (1 area = 1 register) + pieces (variables)

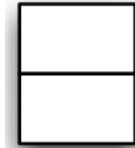


- Pieces cannot overlap
- Some pieces are already placed on the board
- **Task:** fit the remaining pieces on the board (register allocation)

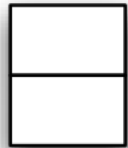


From register file to puzzle boards

- Every area of a puzzle is divided in two rows
(soon will be clear why)



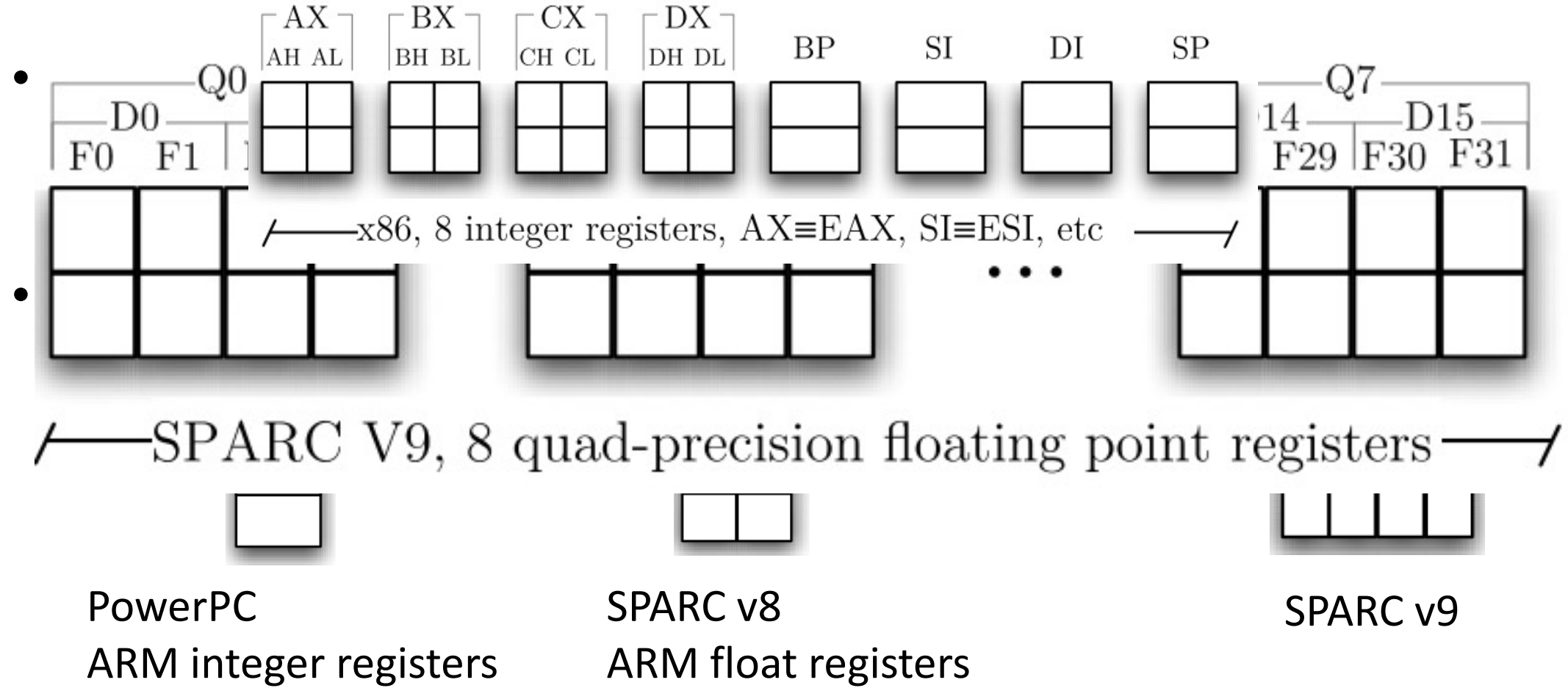
- Registers determine the shape of the puzzle board
Register aliasing determines the #columns



PowerPC



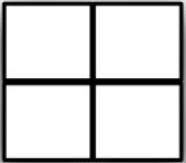
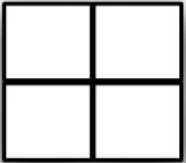
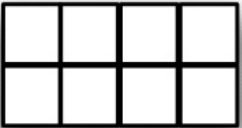
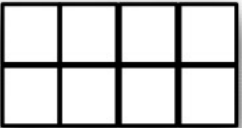
ARM integer registers

From register file to puzzle boards



Puzzle pieces accepted by boards

Our class ->

	Board	
Type-0	0 	K-1 
Type-1		
Type-2		

Outline

- Register allocation abstractions
- 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
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}
```

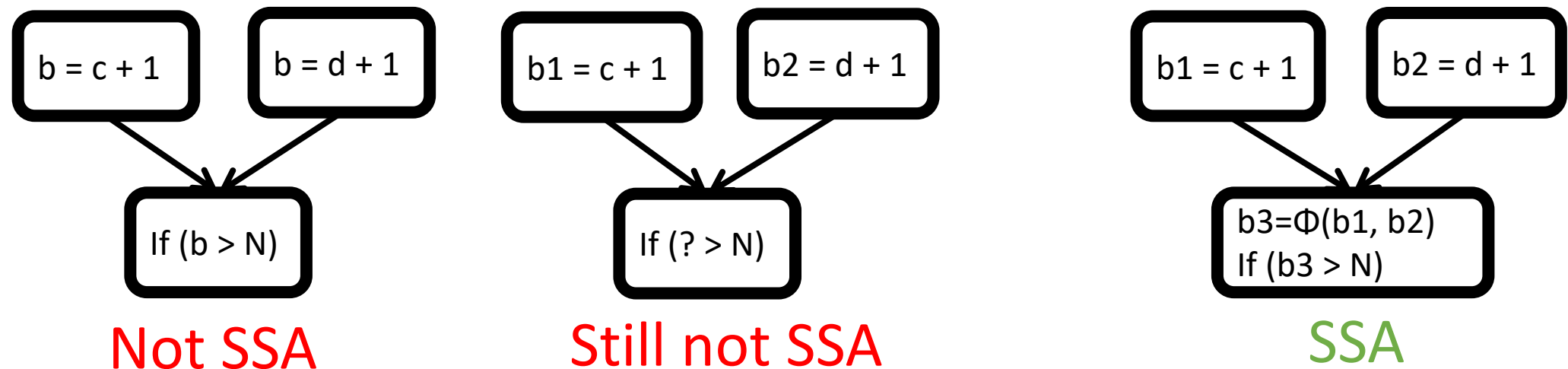
NOT SSA

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

SSA

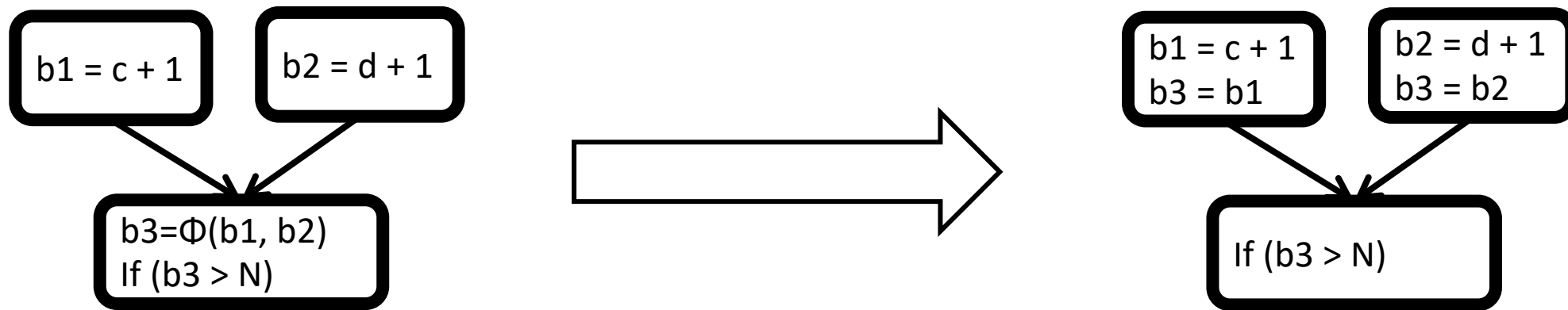
What about joins?

- 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 Φ s
- Register allocation maps the variables to machine registers

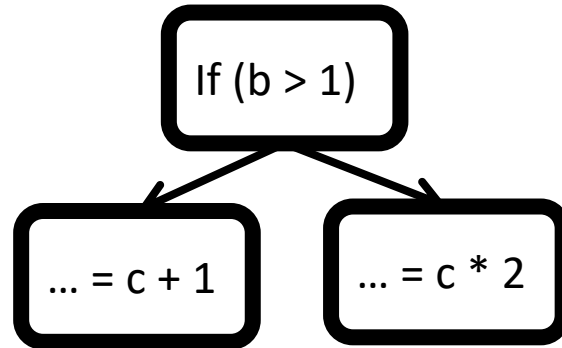
From a program to puzzle pieces

1. Convert a program into an *elementary program*
 - A. Transform code into SSA form
 - B. Transform A into SSI form
2. Map the elementary program into puzzle pieces

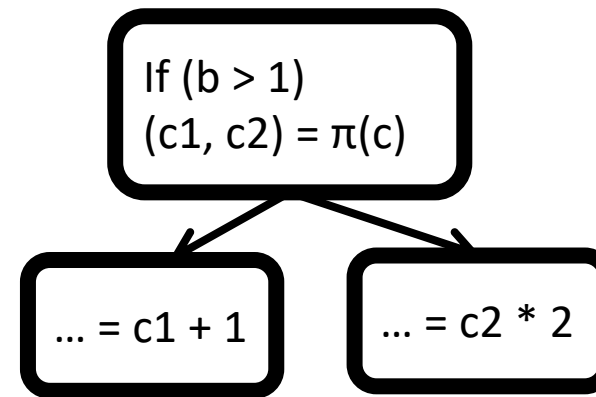
Static Single Information (SSI) form

In a program in SSI form:

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

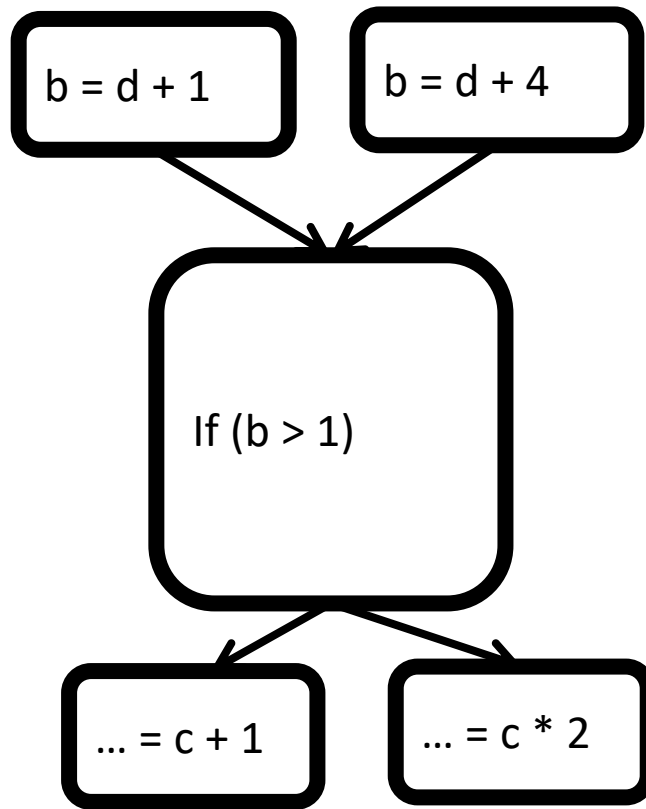


Not SSI

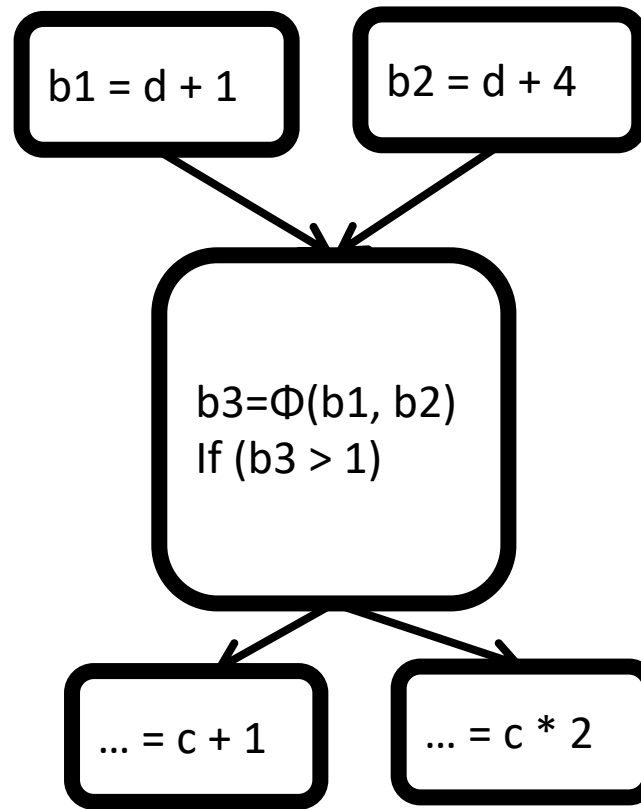


SSI

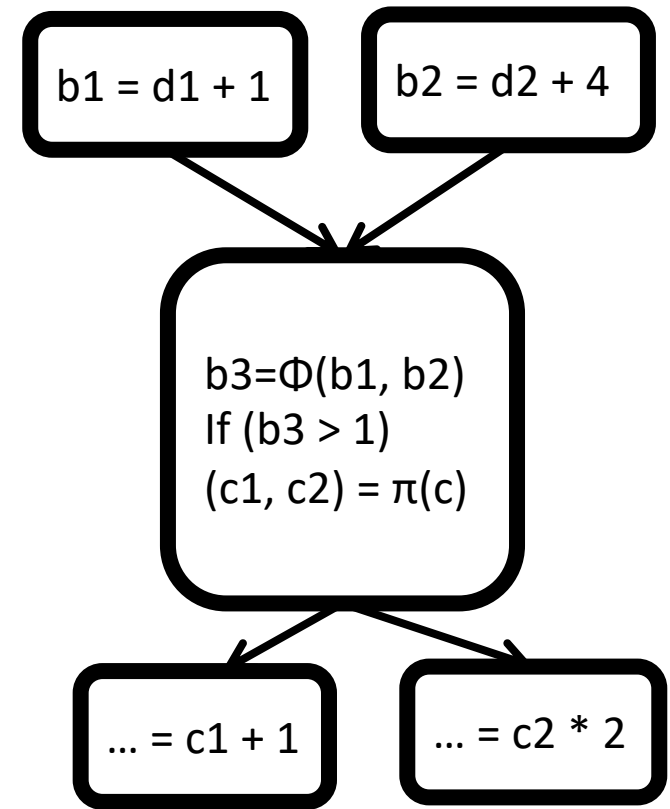
SSA and SSI code



Not SSA and not SSI



SSA but not SSI



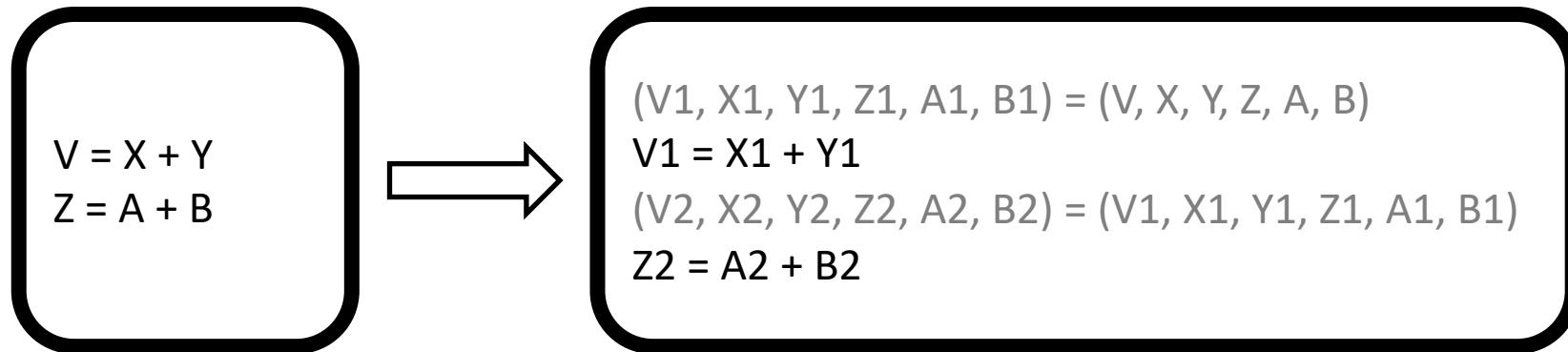
SSA and SSI

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

Parallel copies

- Rename variables in parallel

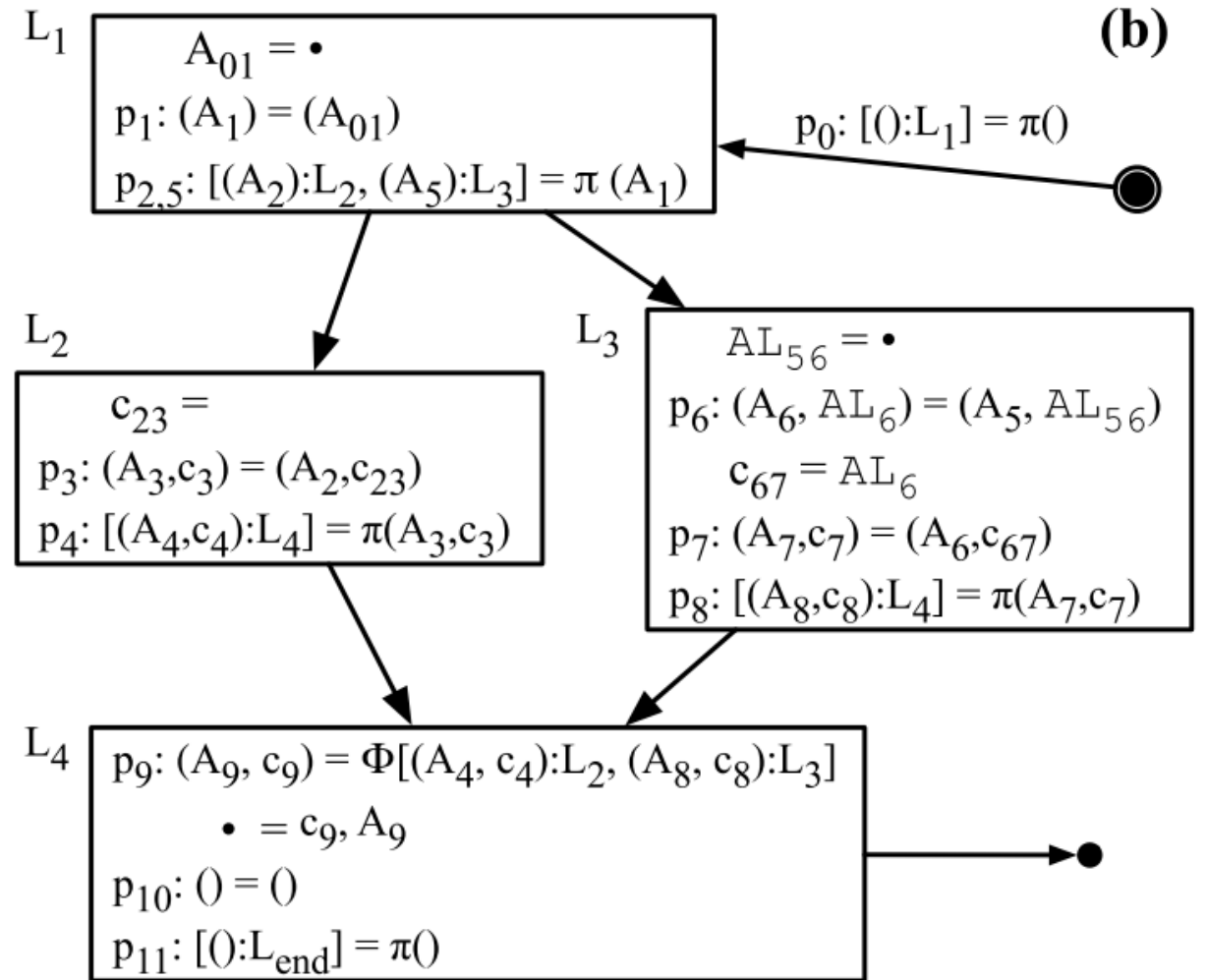
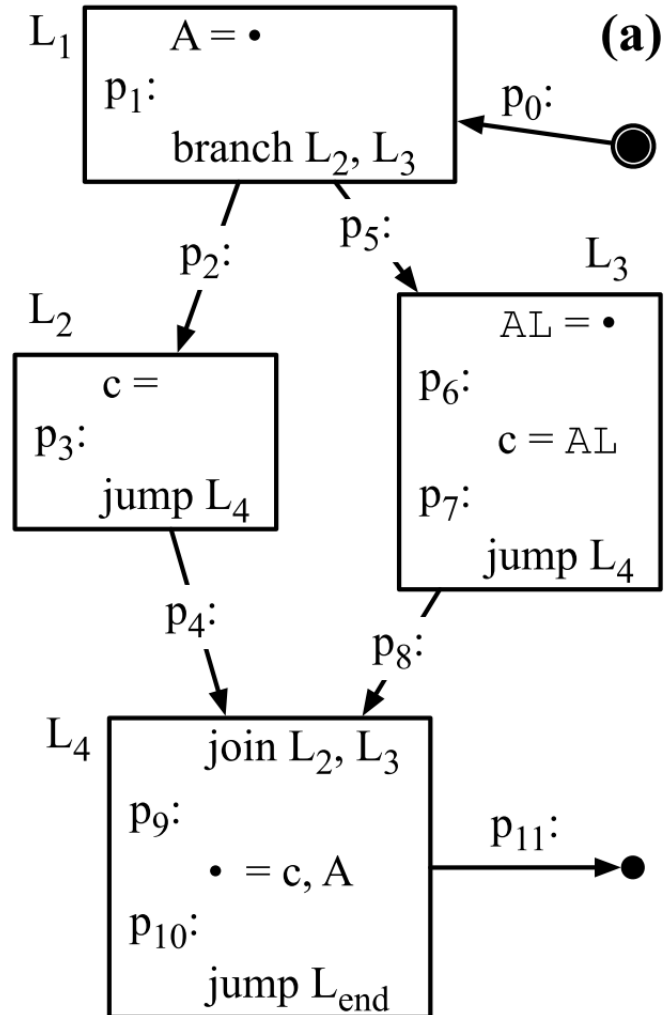


From a program to puzzle pieces

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

We have obtained an elementary program!

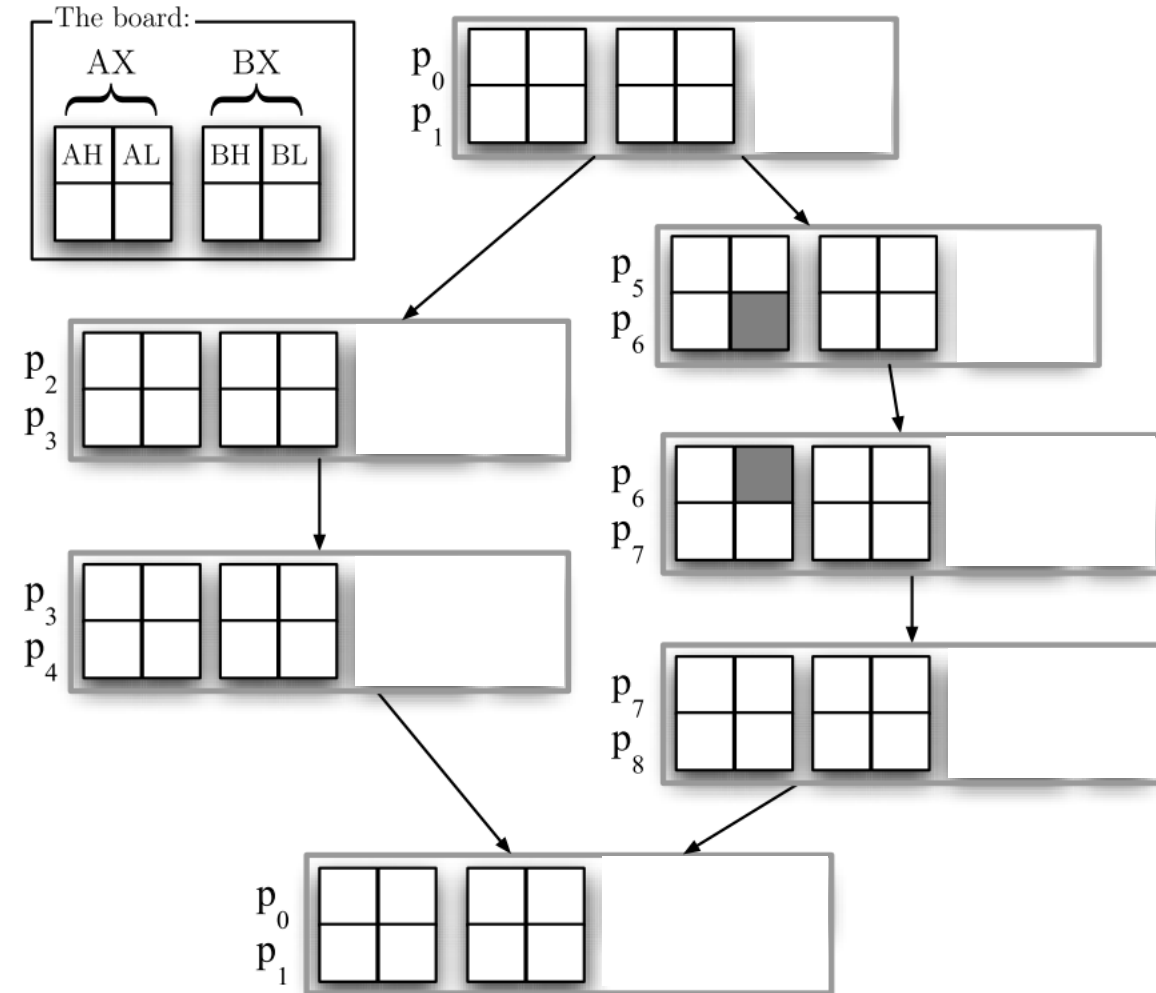
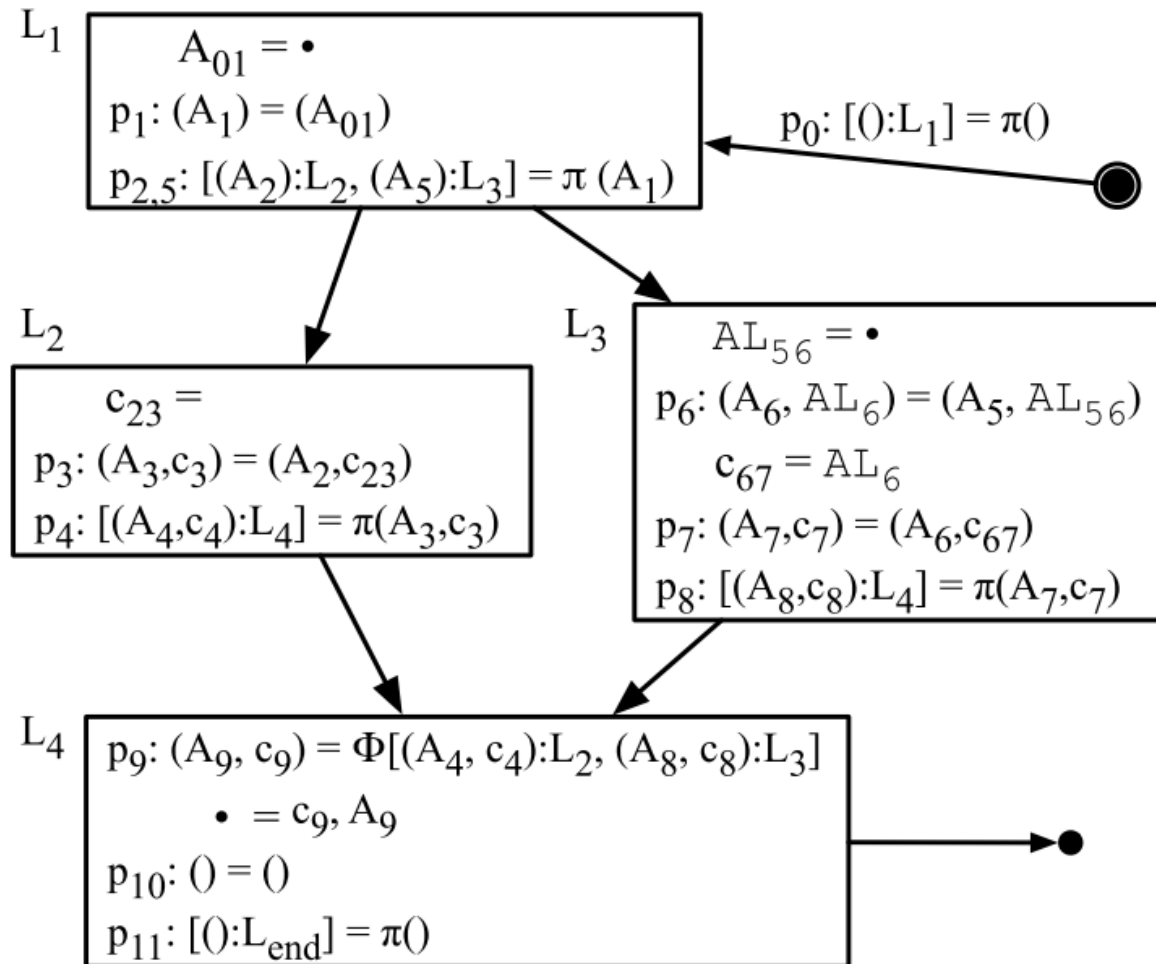
Elementary form: an example



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



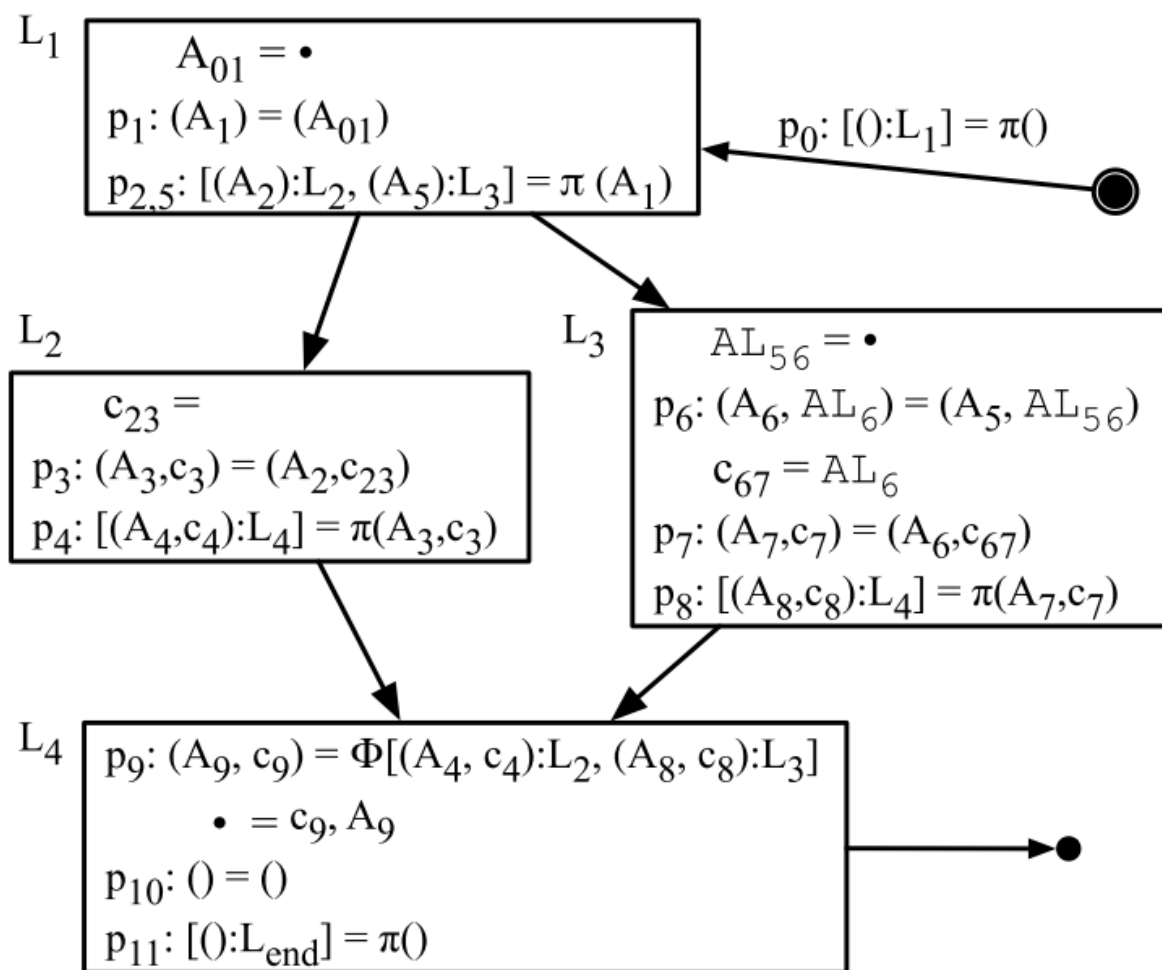
Generating puzzle pieces

- 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

$V1 \text{ (used later)} = V2 \text{ (last use)} + 3$
 $r10 = r10 + 3$

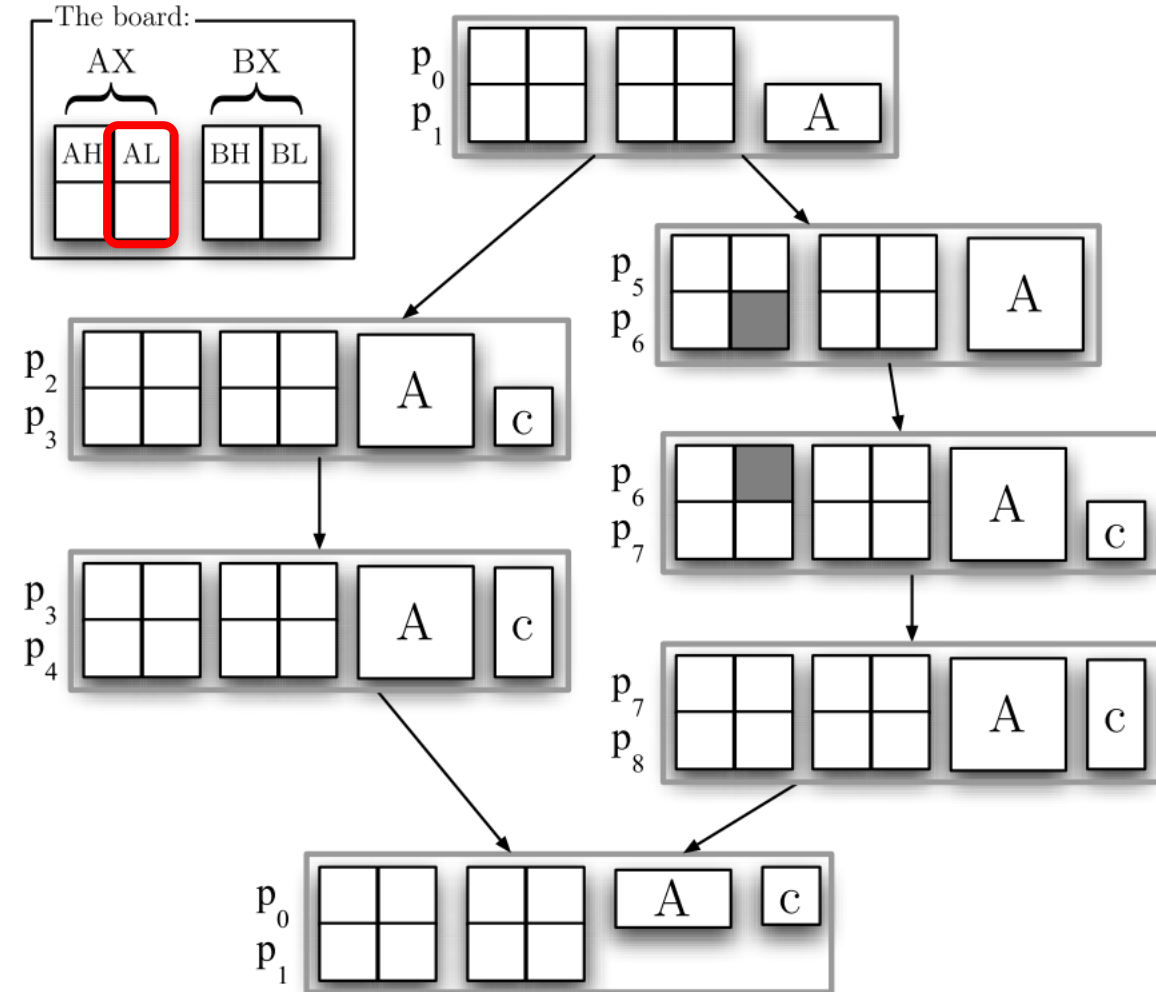
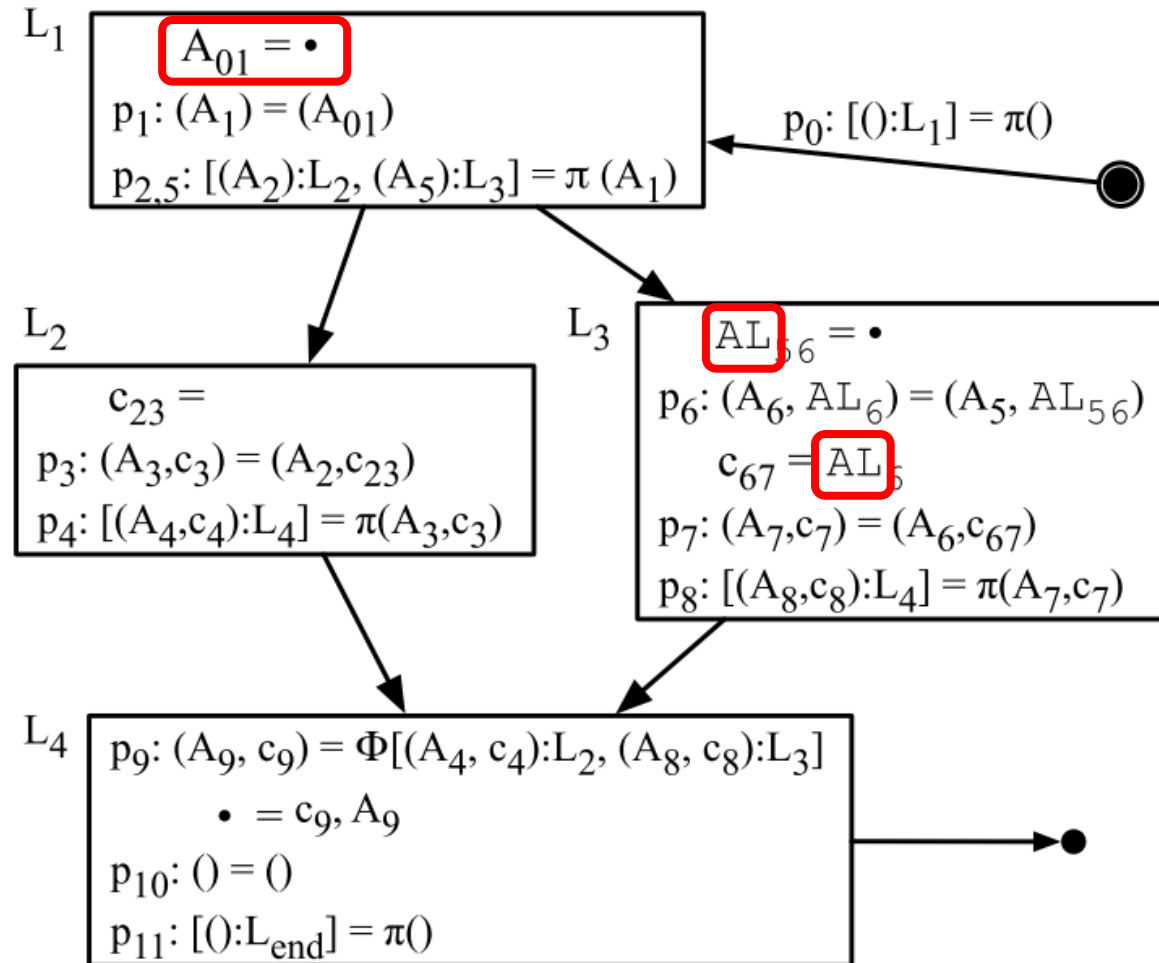
	Board	Kinds of Pieces
Type-0	<div>0</div> <div>...</div> <div>K-1</div>	<div>Y</div> <div>X</div> <div>Z</div>
Type-1	<div>2x2 grid</div> <div>...</div> <div>2x2 grid</div>	<div>Y</div> <div>X</div> <div>Z</div> <div>Y</div> <div>X</div> <div>Z</div>
Type-2	<div>3x3 grid</div> <div>...</div> <div>3x3 grid</div>	<div>Y</div> <div>Y</div> <div>Y</div> <div>X</div> <div>Z</div> <div>X</div> <div>Z</div> <div>X</div> <div>Z</div>

Example



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	
Pieces	

Example

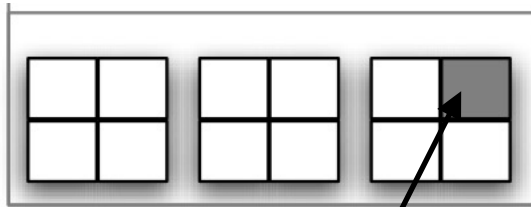


Outline

- Register allocation abstractions
- 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

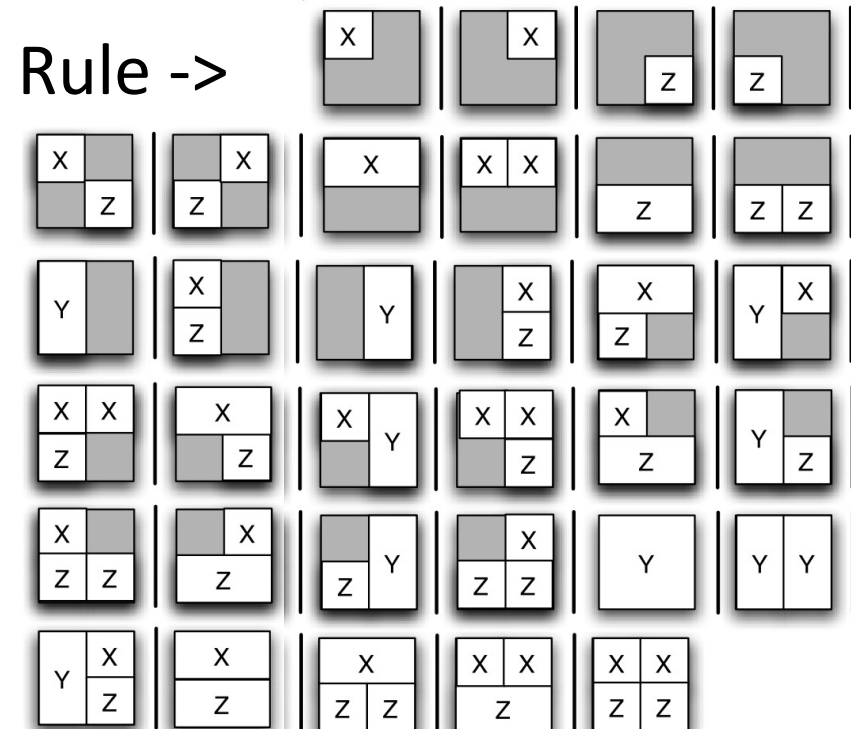
- Solve the puzzle

Solving type 1 puzzles: a visual language

Puzzle solver -> Statement+

Statement -> Rule | Condition

Condition -> (Rule : Statement)



- Rule = how to complete an area

- Rule composed by

pattern:

what needs to be already filled
(match/not-match an area)

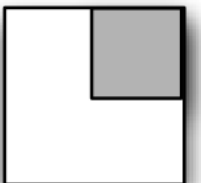
strategy:

what type of pieces to add and where

- A rule r succeeds in an area a iff

- r matches a and
- pieces of the strategy of r are available

Area a

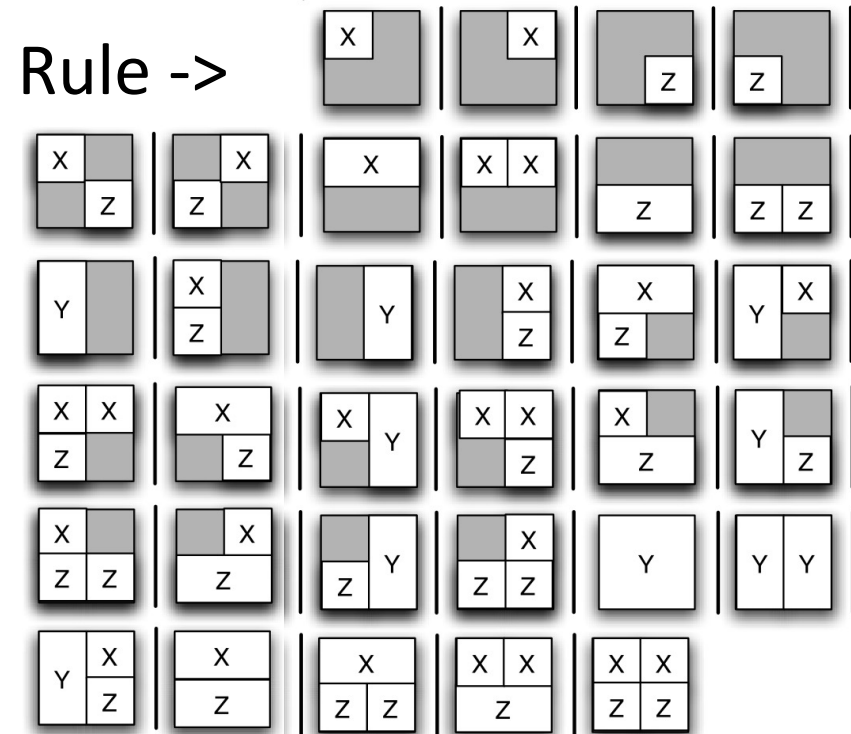


Solving type 1 puzzles: a visual language

Puzzle solver \rightarrow Statement+

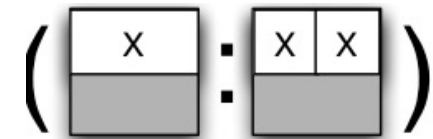
Statement \rightarrow Rule | Condition

Condition \rightarrow (Rule : Statement)



Puzzle solver success

- A program succeeds iff all statements succeeds
- A rule r succeeds in an area a iff
 - r matches a
 - pieces of the strategy of r are available
- A condition $(r : s)$ succeeds iff
 - r succeeds or
 - s succeeds
 - All rules of a condition must have the same pattern

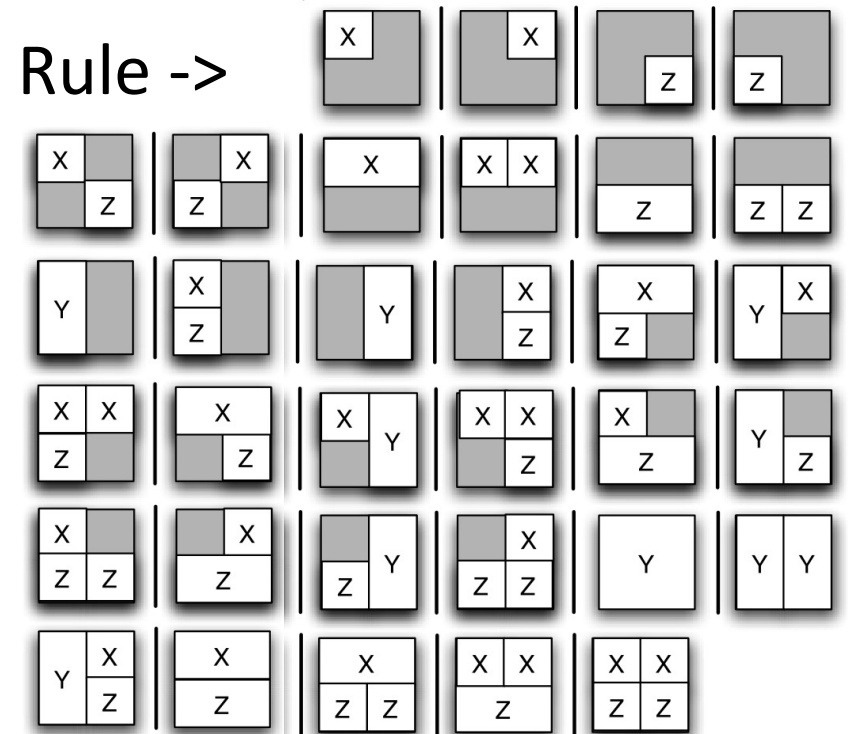


Solving type 1 puzzles: a visual language

Puzzle solver \rightarrow Statement+

Statement \rightarrow Rule | Condition

Condition \rightarrow (Rule : Statement)



Puzzle solver execution

○ For each statement $s1, \dots, sn$

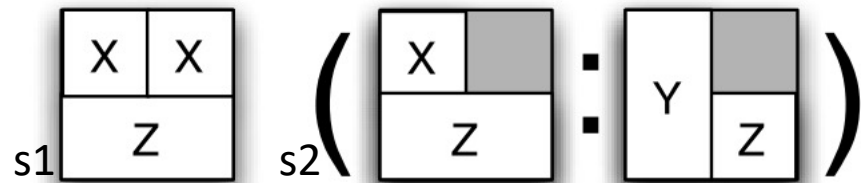
❖ For each area a such that the pattern of si matches a

□ Apply si to a

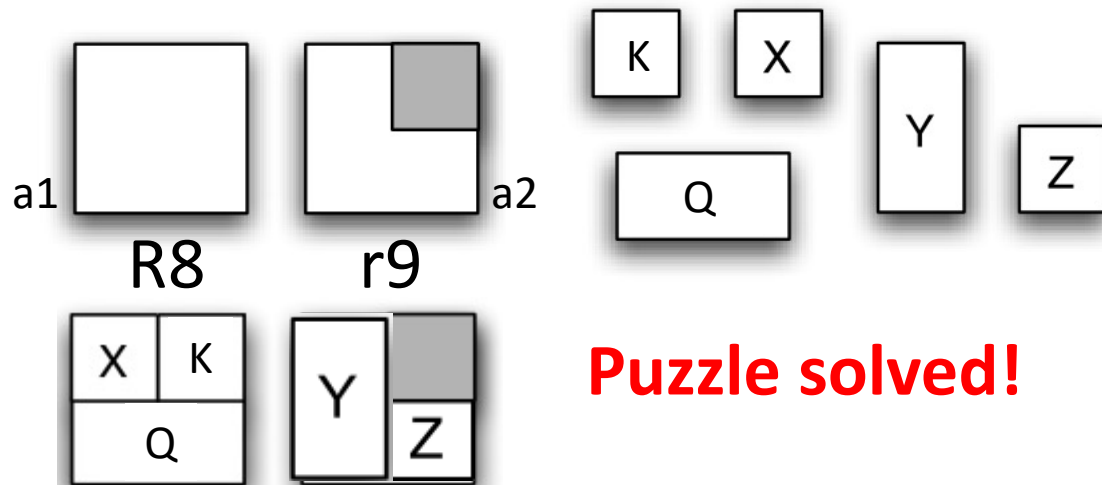
□ If si fails,
terminate
and report failure

Program execution: an example

- A puzzle solver



- Puzzle



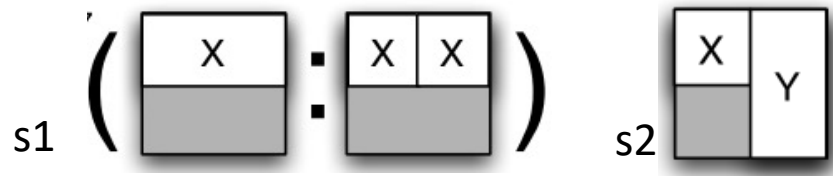
1. s_1 matches a_1 only
2. Apply s_1 to a_1 succeeds and returns this puzzle



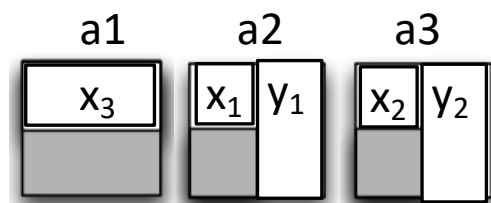
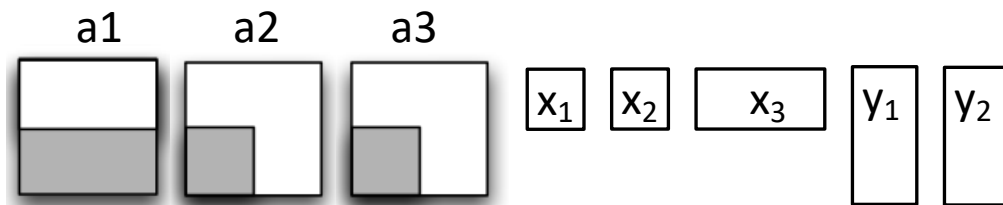
3. s_2 matches a_2 only
4. Apply s_2 to a_2
 - A. Apply first rule of s_2 : fails
 - B. Apply second rule of s_2 : success

Program execution: another example

- A puzzle solver



- Puzzle

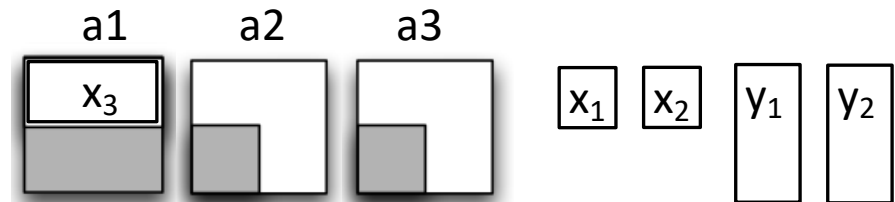


Puzzle solved!

1. s1 matches a1 only

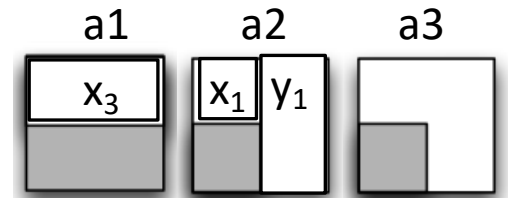
2. Apply s1 to a1

A. Apply first rule of s1: success



3. s2 matches a2 and a3

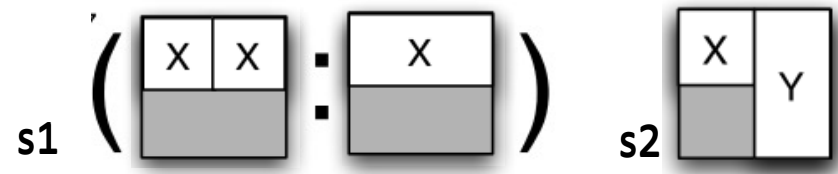
4. Apply s2 to a2



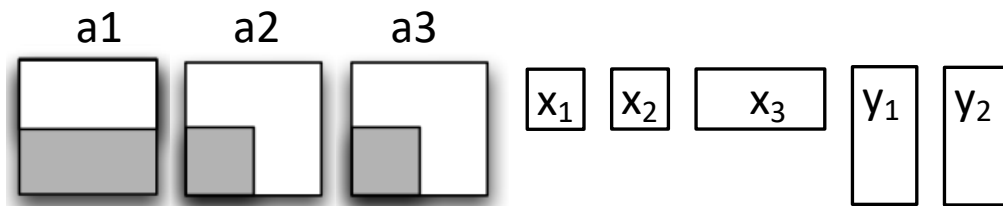
5. Apply s2 to a3

Program execution: yet another example

- A puzzle solver

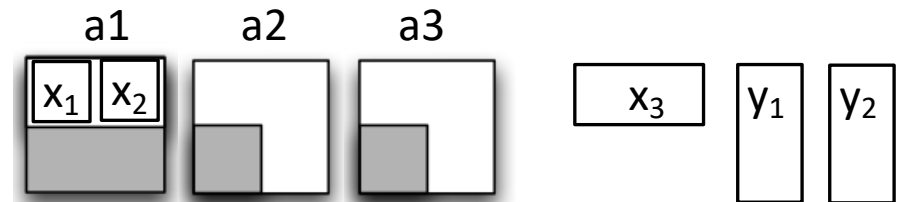


- Puzzle



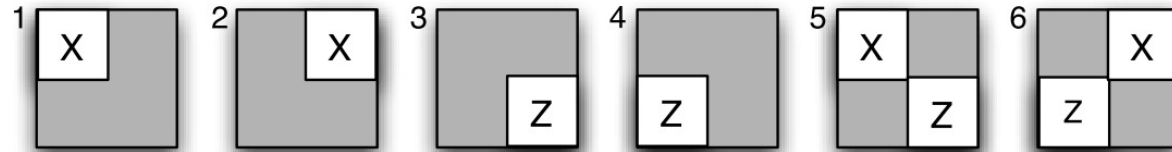
Finding the right puzzle solver
is the key!

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

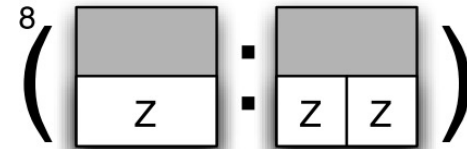


3. s2 matches a2 and a3
4. Apply s2 to a2: **fail**
No 1-size x pieces,
we used them all in s1

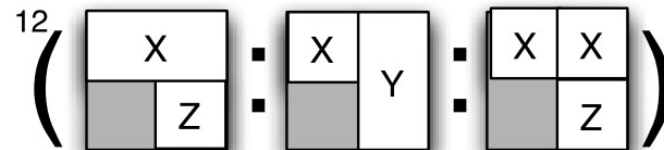
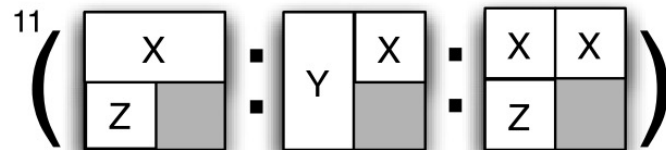
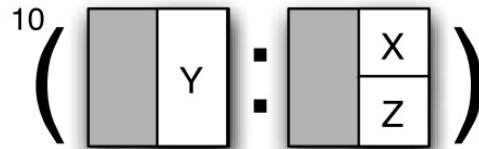
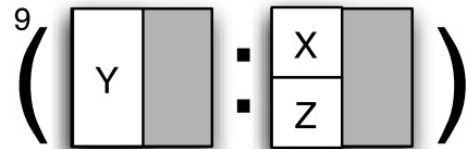
Solution to solve type 1 puzzles



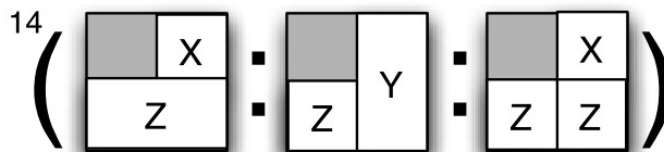
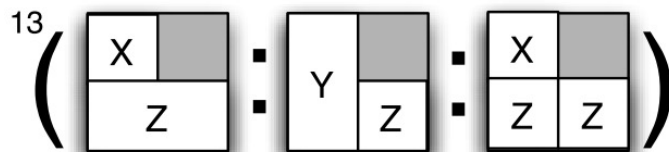
Theorem: a type-1 area is solvable
iff this program succeeds



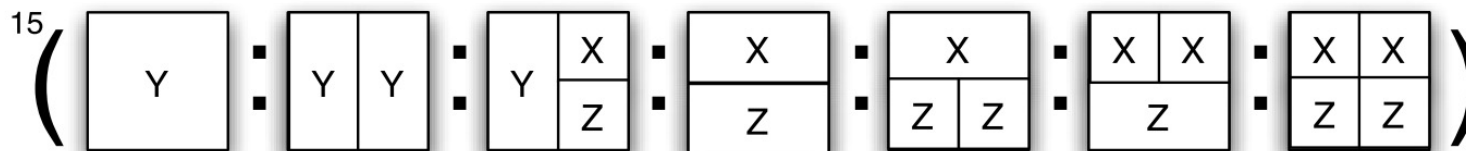
Wait, ...
did we just solve an NP problem
in polynomial time?



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 $O(|P| \times K)$ time

For one instruction in P :

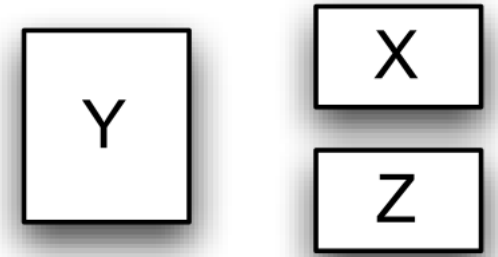
- Application of a rule to an area: $O(1)$
- A puzzle solver $O(1)$ rules on each area of a board
- Execution of a puzzle solver on a board with K areas takes $O(K)$ time

Solving type 0 puzzles

	Board	Kinds of Pieces
Type-0	<div>0</div> <div><div><div></div><div></div></div><div><div></div><div></div></div></div> <div>...</div> <div><div>K-1</div><div><div></div><div></div></div><div><div></div><div></div></div></div>	<div><div>Y</div></div> <div><div>X</div><div>Z</div></div>
Type-1	<div><div><div></div><div></div><div></div><div></div></div><div>...</div><div><div></div><div></div><div></div><div></div></div></div>	<div><div>Y</div></div> <div><div>X</div><div>Z</div></div> <div><div>Y</div></div> <div><div>X</div><div>Z</div></div>
Type-2	<div><div><div></div><div></div><div></div><div></div><div></div><div></div></div><div>...</div><div><div></div><div></div><div></div><div></div><div></div><div></div></div></div>	<div><div>Y</div></div> <div><div>Y</div></div> <div><div>Y</div></div> <div><div>X</div><div>Z</div></div> <div><div>X</div><div>Z</div></div> <div><div>X</div><div>Z</div></div>

Solving type 0 puzzles: algorithm

- Place all Y-pieces on the board



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

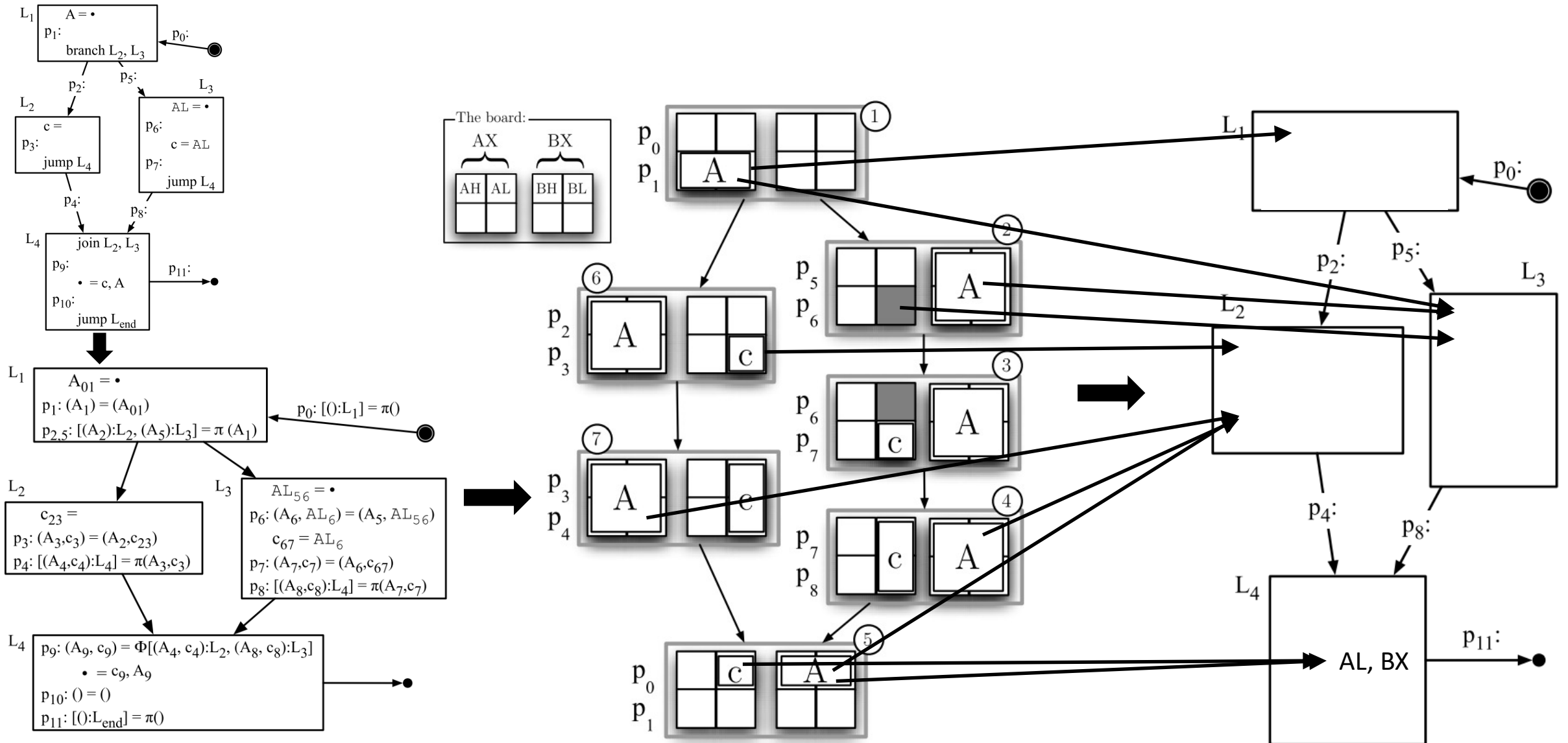
Spilling

- If the algorithm to solve a puzzle 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

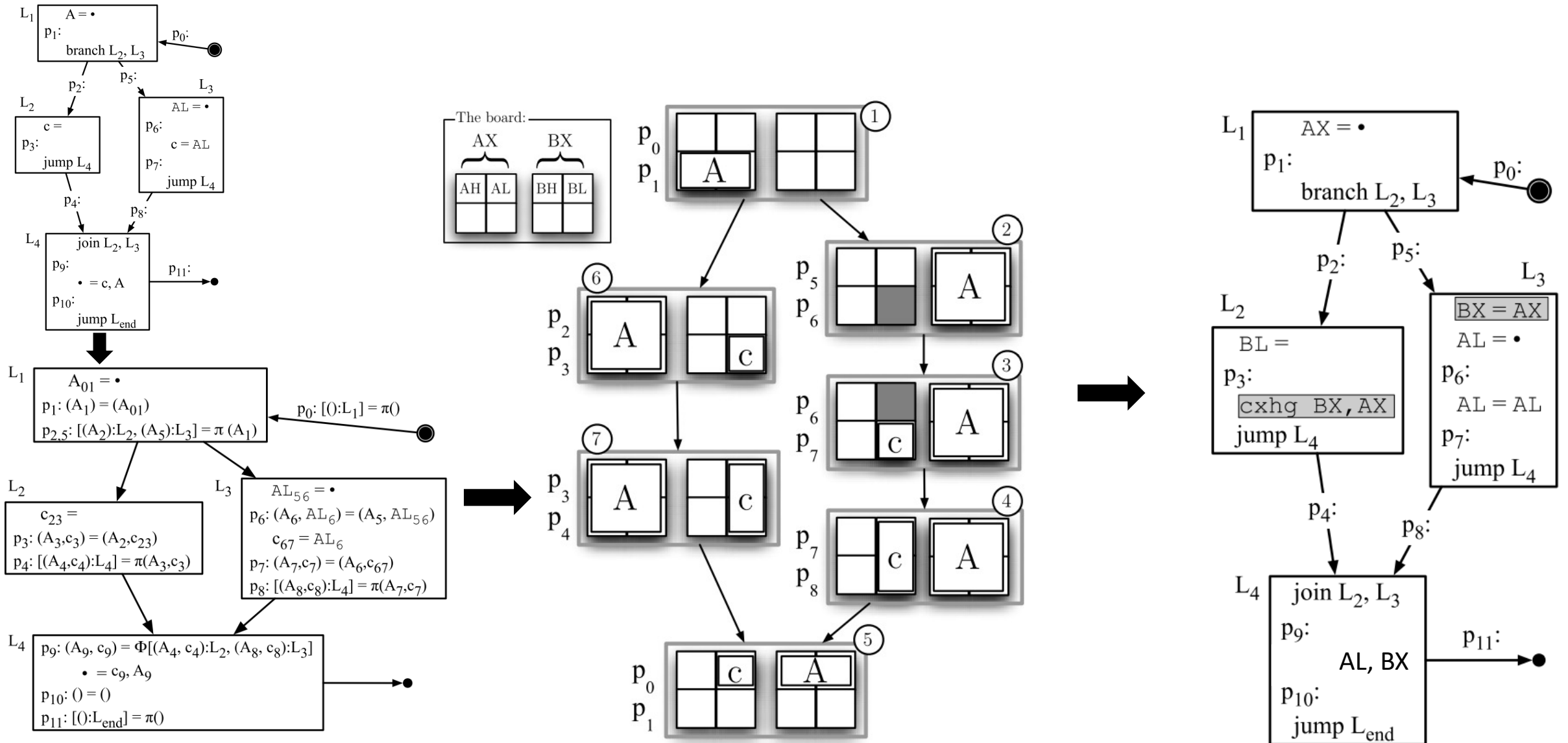
Outline

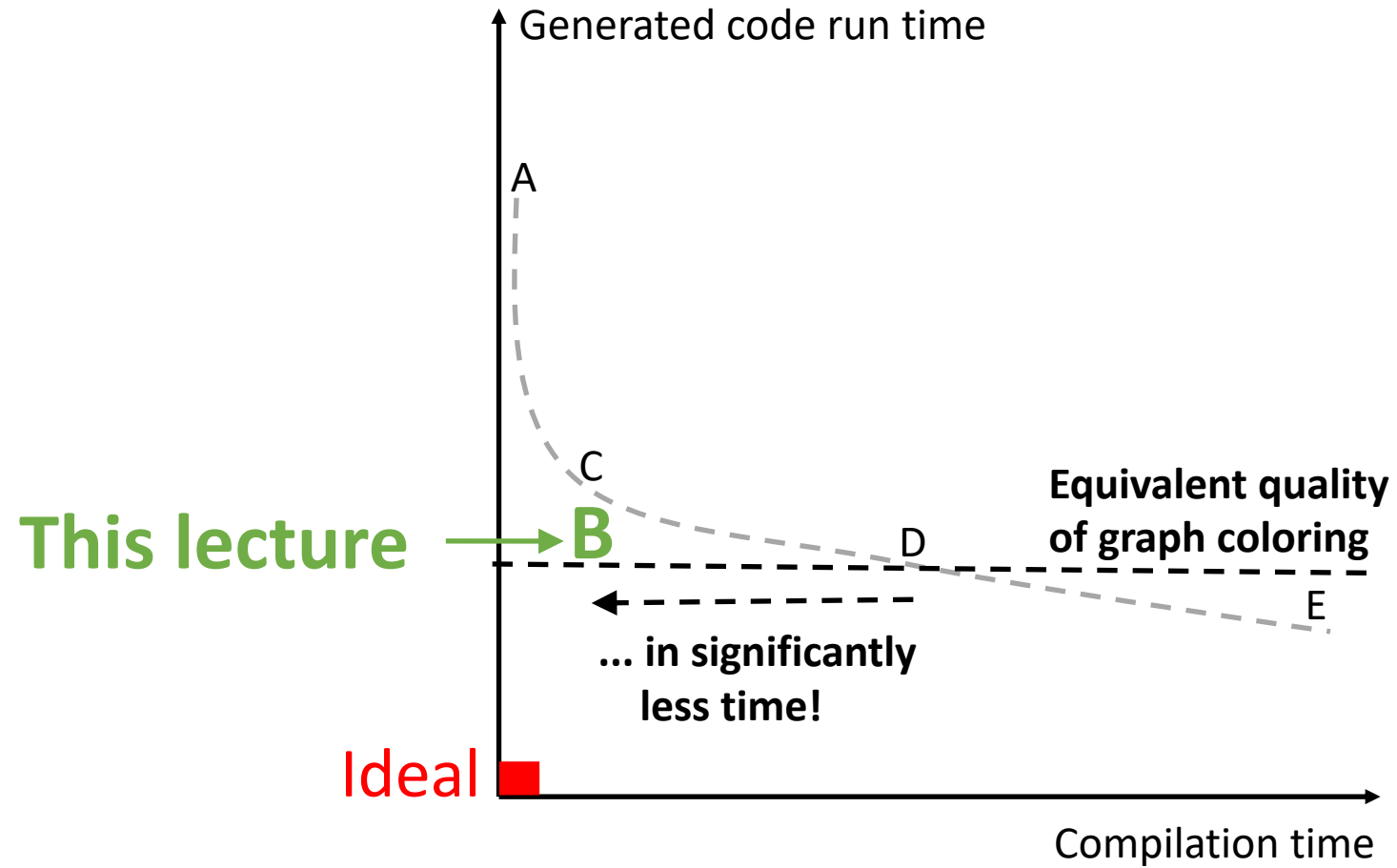
- Register allocation abstractions
- From a program to a collection of puzzles
- Solve puzzles
- From solved puzzles to assembly code

From solved puzzles to assembly code



From solved puzzles to assembly code





Thank you!

Always have faith in your ability

Success will come your way eventually

Best of luck!