A Refactoring Tool for the Safe Parallelization of Array-Centric For-Loops

Christoph Amrein, Luc Bläser

Ziel

- Sichere Parallelisierung
  - *For* zu *Parallel.For*
- Gestützt durch schnelle und einfache Datenflussanalyse
  - Alternative zu GCD / Polyeder

```
for(int i = 0; i < 10; i++) {
  // ...
}
```

```
Parallel.For(0, 10, i => {
  // ...
});
```

Array Access Intersections

Schleife ist parallelisierbar

```
long[] factorials = new long[] {
  15, 15, 18, 18, 20, 20
};

int upperBound = factorials.Length / 2;
for(int i = 0; i < upperBound; i++) {
  long current = factorials[1 + i * 2];
  long factorial = 1;

  for(int j = 1; j <= current; j++) {
    factorial = factorial * j;
  }

  factorials[1 + i * 2] = factorial;
}
```

Schleife ist nicht parallelisierbar

```
int[] sum = new int[] {
  1, 2, 3, 4, 5, 6
};

for(int i = 1; i < sum.Length; i++) {
  sum[i] = sum[i - 1] + sum[i];
}
```

Loop Dependence Analyse: Übersicht

- DFA wie Instanzen des Monotone Frameworks

```
int[] shared = new int[20];
for(int i = 0; i < 10; ++i) {
  int p = i + 2;
  if(i % 2 == 0) {
    shared[p] = i;
  } else {
    shared[p] = -i;
  }
}
```

Loop Dependence Analyse: Übersicht

- DFA wie Instanzen des Monotone Frameworks

1. Transformiere zu SSA

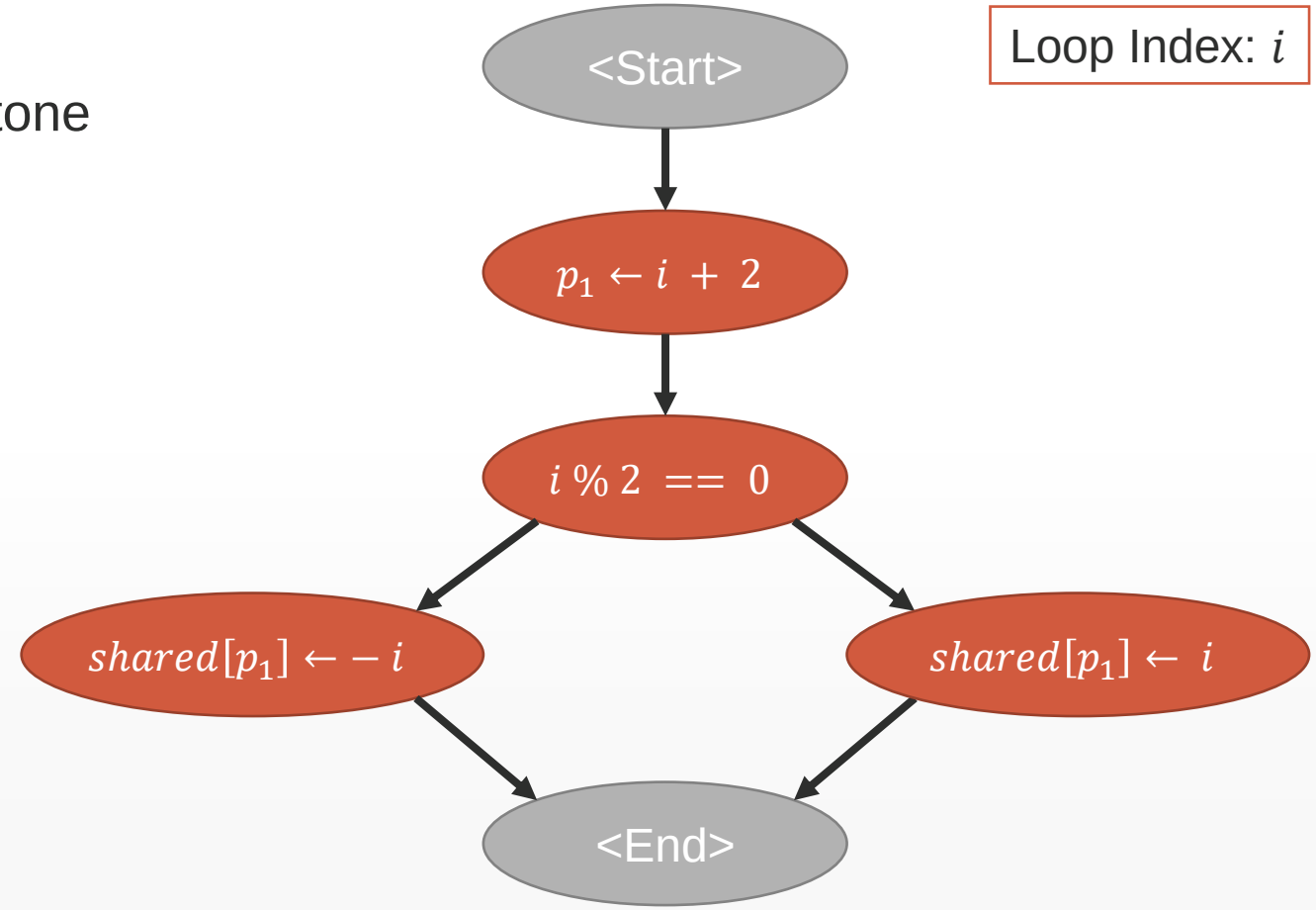
$p_1 \leftarrow i + 2$
`if` $i\%2 == 0$ `goto` 1
$shared[p_1] \leftarrow -i$
`goto` 2
`label` 1
$shared[p_1] \leftarrow i$
`label` 2

Loop Index: i

Loop Dependence Analyse: Übersicht

- DFA wie Instanzen des Monotone Frameworks

1. Transformiere zu SSA
2. Erzeuge CFG

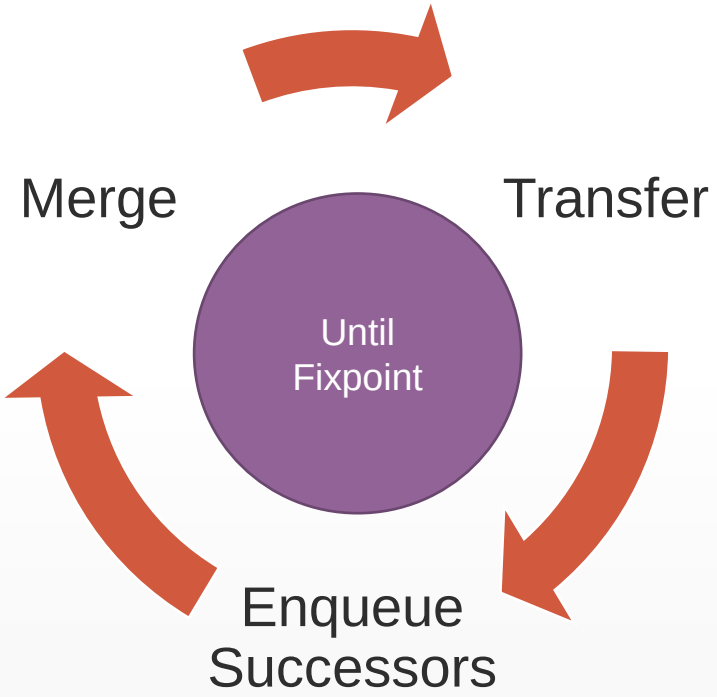


Loop Index: i

Loop Dependence Analyse: Übersicht

- DFA wie Instanzen des Monotone Frameworks

1. Transformiere zu SSA
2. Erzeuge CFG
3. Wende DFA an



Loop Dependence Analyse: Übersicht

- DFA wie Instanzen des Monotone Frameworks

1. Transformiere zu SSA
2. Erzeuge CFG
3. Wende DFA an
4. Interpretiere die Resultate

```
int[] shared = new int[12];
for(int i = 0; i < 10; ++i) {
  int p = i + 2;
  if(i % 2 == 0) {
    shared[p] = i;
  } else {
    shared[p] = -i;
  }
}
```

Überschneidende Zugriffe auf das Array «shared»?

Loop Dependence Analyse: Transferfunktion

$p_1 \leftarrow i + 2$

- Verwendet Menge von vordefinierten Regeln für Expressions
- Konservativ
  - Informationstransfer nur wenn garantiert

Loop Dependence Analyse: Transferfunktion

$p_1 \leftarrow i + 2$

i	+	2

Loop Dependence Analyse: Transferfunktion

$$p_1 \leftarrow i + 2$$

i	+	2
Variable	Addition	Konstante

Loop Dependence Analyse: Transferfunktion

$p_1 \leftarrow i + 2$

i	+	2
Variable	Addition	Konstante
$\{\mid, +, \not\simeq 0\}$		$\{\nmid, +, \not\simeq 0\}$

$+$: Positive, $-$: Negative, $\not\simeq 0$: At-most-once-zero,
$\mid$: Loop dependent, $\nmid$: Not loop dependent

Loop Dependence Analyse: Transferfunktion

$p_1 \leftarrow i + 2$

i	+	2
Variable	Addition	Konstante
$\{\mid, +, \not\simeq 0\}$		$\{\nmid, +, \not\simeq 0\}$
Transferregeln für Addition		

$+$: Positive, $-$: Negative, $\not\simeq 0$: At-most-once-zero,
$\mid$: Loop dependent, $\nmid$: Not loop dependent

Loop Dependence Analyse: Transferfunktion

$p_1 \leftarrow i + 2$

i	+	2
Variable	Addition	Konstante
$\{\mid, +, \not\simeq 0\}$		$\{\nmid, +, \not\simeq 0\}$
Transferregeln für Addition		

Transferregeln für Addition

Links	Rechts	Transferiert
$\{\mid\}$	$\{\nmid\}$	$\mid$
$\{\mid, +\}$	$\{+\}$	$\mid$
…		
$\{\nmid\}$	$\{\nmid\}$	$\nmid$
$\{+\}$	$\{+\}$	$+$
$\{-\}$	$\{-\}$	$-$
$\{+\}$	$\{+\}$	$\not\simeq 0$
$\{-\}$	$\{-\}$	$\not\simeq 0$
…		

$+$: Positive, $-$: Negative, $\not\simeq 0$: At-most-once-zero,
$\mid$: Loop dependent, $\nmid$: Not loop dependent

Loop Dependence Analyse: Transferfunktion

$p_1 \leftarrow i + 2$

i	+	2
Variable	Addition	Konstante
$\{\mid, +, \not\approx 0\}$		$\{\nmid, +, \not\approx 0\}$
Transferregeln für Addition		
$\{\mid, +, \not\approx 0\}$		

Transferregeln für Addition		
Links	Rechts	Transferiert
$\{\mid\}$	$\{\nmid\}$	$\mid$
$\{\mid, +\}$	$\{+\}$	$\mid$
…		
$\{\nmid\}$	$\{\nmid\}$	$\nmid$
$\{+\}$	$\{+\}$	$+$
$\{-\}$	$\{-\}$	$-$
$\{+\}$	$\{+\}$	$\not\approx 0$
$\{-\}$	$\{-\}$	$\not\approx 0$
…		

$+$: Positive, $-$: Negative, $\not\approx 0$: At-most-once-zero,
$\mid$: Loop dependent, $\nmid$: Not loop dependent

Loop Dependence Analyse: Resultate

Drei Checks für den Element Accessor sind notwendig:

1. Hat «Loop Dependent» Eigenschaft
   $\{\|, +, \neq 0\}$
2. Hat bei jedem Zugriff denselben Wert (p ist immer p_1)
3. Wird immer für die gleiche Dimension verwendet

```
int[] shared = new int[12];
for(int i = 0; i < 10; ++i) {
  int p = i + 2;
  if(i % 2 == 0) {
    shared[p] = i;
  } else {
    shared[p] = -i;
  }
}
```

Überschneidende Zugriffe auf das Array «shared»?
Nein, Loop ist parallelisierbar ✓

Demo

Schlussfolgerung: Parallel Refactoring Tool

- Schnelle Analyse
- Alternative zu Polyhedral und GCD
- Einfaches Modell
- Erweiterbar

Christoph Amrein
christoph.amrein@hsr.ch

Luc Bläser
luc.Blaeser@hsr.ch
concurrency.ch

Vielen Dank

Quellcode und Implementation sind verfügbar unter:
https://github.com/camrein/RefactorToParallel

Bonus

Implementation

- Visual Studio Integration
- Konvertiert C# Code zu einem Drei-Adress-Code
- Wendet Copy Propagation und Common Subexpression Elimination an
- Unterstützt Zugriff auf unterschiedliche Arrays
  - Alias Analysis innerhalb des Drei-Adress-Codes
  - Logische Beweisführung von Aliasen ausserhalb des Loops

Evaluation

- Der Prototyp wurde auf 20 Open Source Projekten evaluiert
  - 223 for-loops analysiert
  - 199 for-loops sind parallelisierbar
  - 193 for-loops erkannt
- Nur geringfügig sind umfangreichere Analysen notwendig

```
for(int i = 0; i < buffer.Length/3; i++) {
  buffer[i * 3] = /* ... */;
  buffer[i * 3 + 1] = /* ... */;
  buffer[i * 3 + 2] = /* ... */;
}
```

Loop Dependence Analysis: Not Parallelizable

```
 1: int[] sum = new int[] {
 2:  1, 2, 3, 4, 5, 6
 3: };
 4:
 5: for(int i = 1; i < sum.Length; i++) {
 6:  int p = i - 1;
 7:  int value = sum[p];
 8:  p = i;
 9:  sum[p] = value + sum[p];
10: }
```

- Verwendung von zwei unterschiedlichen Definitionen von p
  - Definition von Zeile 6 wird auf Zeile 7 verwendet
  - Definition von Zeile 8 wird auf Zeile 9 verwendet

```
    // ...
 5: for(int i = 1; i < sum.Length; i++) {
 6:  int p = i - 1;
 7:  sum[i] = sum[p] + sum[i];
 8: }
```

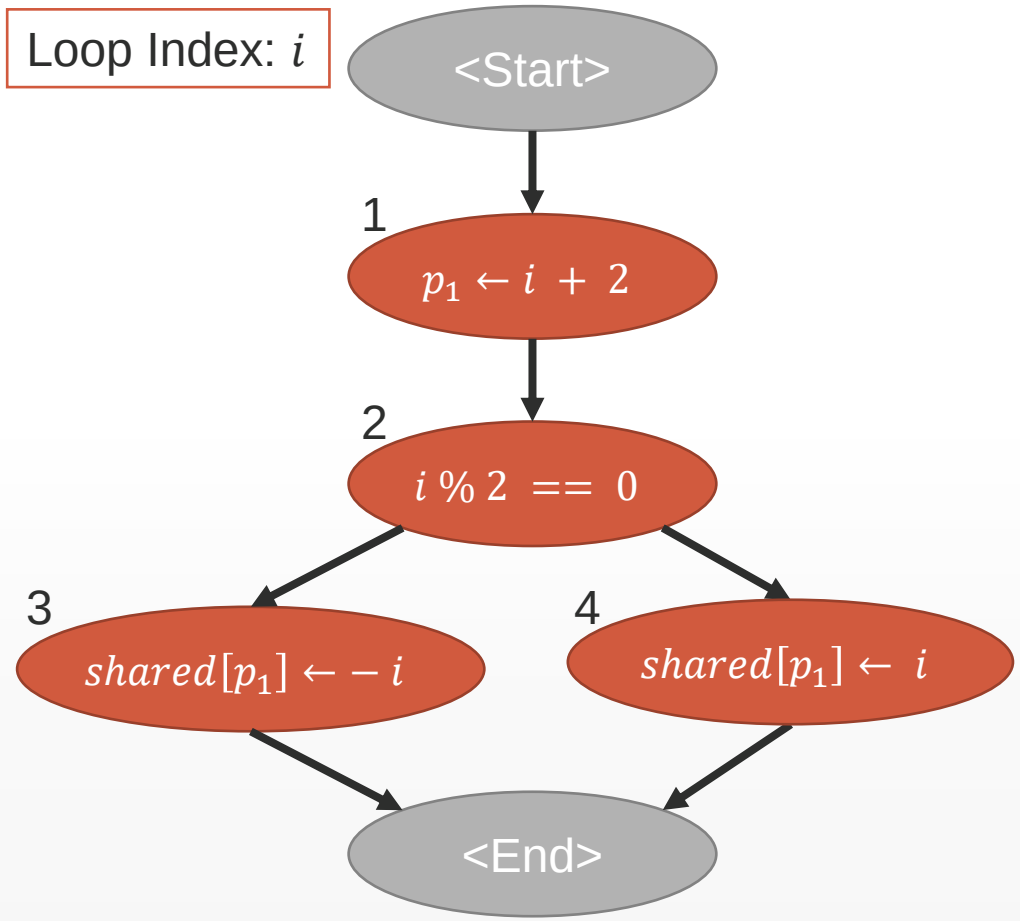
- Verwendung von zwei unterschiedlichen Variablen
  - p und i werden auf Zeile 7 verwendet

Loop Dependence Analysis: Not Parallelizable

```
1: int[] sum = new int[] {
2:   0, 1, 2, 3, 4, 5, 6
3: };
4:
5: for(int i = 1; i < sum.Length; i++) {
6:   int r = 0;
7:   sum[r] += sum[i];
8: }
```

- Verwendung einer Variable ohne «Loop Dependence» Eigenschaft
  - Variable r wird auf Zeile 7 verwendet
- Verwendung von zwei unterschiedlichen Variablen
  - r und i werden auf Zeile 7 verwendet

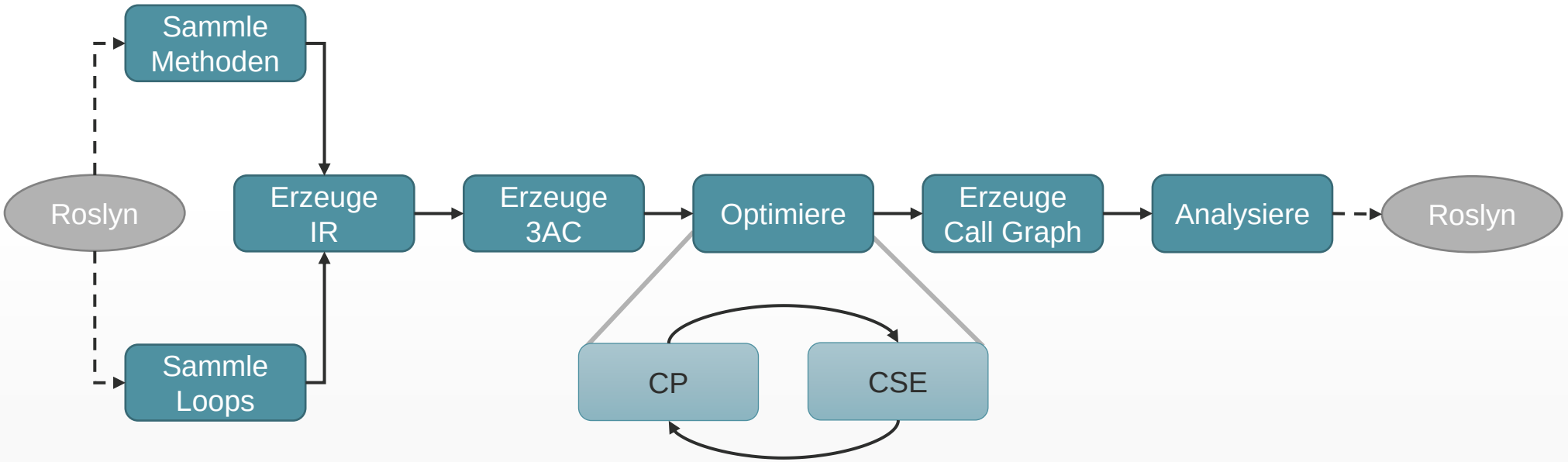
Loop Dependence Analyse: DFA Lösung



ℓ	$Entry(\ell)$	$Exit(\ell)$
1	$\{i\|, i+, i \neq 0\}$	$\{i\|, i+, i \neq 0, p_1\|, p_1+, p_1 \neq 0\}$
2	$\{i\|, i+, i \neq 0, p_1\|, p_1+, p_1 \neq 0\}$	$\{i\|, i+, i \neq 0, p_1\|, p_1+, p_1 \neq 0\}$
3	$\{i\|, i+, i \neq 0, p_1\|, p_1+, p_1 \neq 0\}$	$\{i\|, i+, i \neq 0, p_1\|, p_1+, p_1 \neq 0\}$
4	$\{i\|, i+, i \neq 0, p_1\|, p_1+, p_1 \neq 0\}$	$\{i\|, i+, i \neq 0, p_1\|, p_1+, p_1 \neq 0\}$

Nur Blöcke innerhalb des Loops werden analysiert.
Array Elemente werden nicht berücksichtigt.

Plugin: Analyse Schritte



Plugin: Interne Code Konvertierung

```
1  for(int i = 1; i < 10; i++)
2  {
3    a[i + 1] += 1;
4  }
```

C#

```
1   DECL i
2   i = 1
3   LABEL #0
4   IF i < 10 JUMP #1
5   JUMP #2
6   LABEL #1
7   a[i + 1] = a[i + 1] + 1
8   LABEL #3
9   i = i + 1
10  JUMP #0
11  LABEL #2
```

IR

```
1   DECL i
2   i = 1
3   LABEL #0
4   IF i < 10 JUMP #1
5   JUMP #2
6   LABEL #1
7   DECL $temp_0
8   $temp_0 = i + 1
9   DECL $temp_1
10  $temp_1 = i + 1
11  DECL $temp_2
12  $temp_2 = a[$temp_1]
13  a[$temp_0] = $temp_2 + 1
14  LABEL #3
15  i = i + 1
16  JUMP #0
17  LABEL #2
```

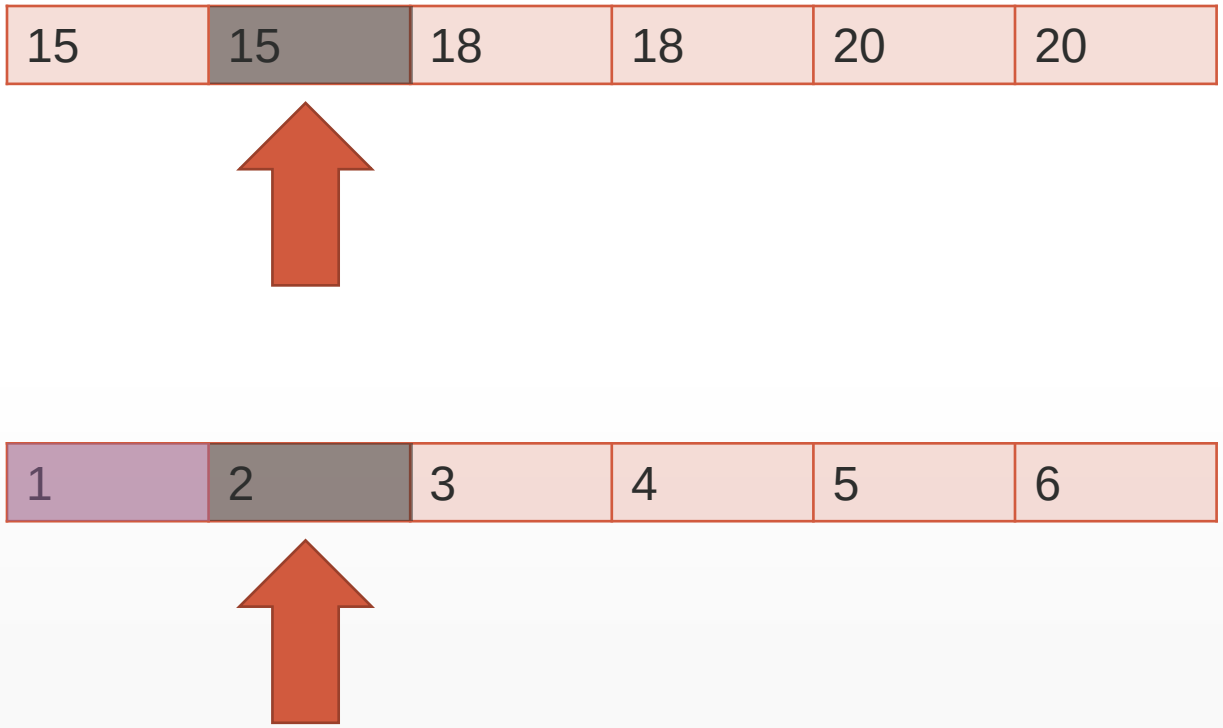
3AC

Plugin: Analyse Verbesserungen

- Identifikation von Verzweigungen welche für den Loop Konstant sind
  - Z.B. Bedingungen welche jede Iteration zum selben Resultat evaluieren
  - Jede Iteration hat denselben Ausführungspfad
- Teile den Loop in kleinere Teile auf (z.B. mit Slicing) und analysiere jeden Teil separat

```
 1: int[] shared = new int[] {
 2:  0, 1, 2, 3, 4, 5, 6, 7
 3: };
 4: bool externalState = /* ... */;
 5:
 6: for(int i = 0; i < 4; i++) {
 7:  if(externalState) {
 8:   shared[i * 2 + 1] += 1;
 9:  } else {
10:   shared[i] *= 2;
11:  }
12: }
```

Array Access Intersections



Array Access Intersections