Polynomial Loop Recognition in Traces
(tool paper)

Alain Ketterlin

/ Camus / ICube / ICPS

IMPACT 2025: January 22, 2025

- *Nested Loop Recognition* (NLR)
  - takes a trace as input
  - outputs one or more affine loops
  - such that the loops produce the trace

- *Nested Loop Recognition* (NLR)
    - takes a trace as input
    - outputs one or more affine loops
    - such that the loops produce the trace
- NLR has been used for
    - trace compression
    - memory address prediction
    - also on parallel (MPI) traces
    - dynamic optimization
    - assisted static analysis

- *Nested Loop Recognition* (NLR)
  - takes a trace as input
  - outputs one or more affine loops
  - such that the loops produce the trace
- NLR has been used for
  - trace compression
  - memory address prediction
  - also on parallel (MPI) traces
  - dynamic optimization
  - assisted static analysis

- Goal: find integer polynomials wherever NLR has integer affine functions
  - for increased expressive power in general
  - to capture any kind of accumulation (e.g., ranks)
  - (as a natural next step)

→ Polynomial Loop Recognition (PLR)

- the input is made of tagged vectors of numbers

```
- val A, 10
```

- the input is made of tagged vectors of numbers
- shift-reduce strategy

→ incoming data is *shifted* to a stack

```
- val A, 10
- val B, 100
```

- the input is made of tagged vectors of numbers
- shift-reduce strategy

→ incoming data is *shifted* to a stack

```
- val A, 10
- val B, 100
- val B, 110
```

- the input is made of tagged vectors of numbers
- shift-reduce strategy

→ incoming data is *shifted* to a stack

- sometimes, reductions happen:
  - a new loop is formed

```
- val A, 10
- val B, 100
- val B, 110
- val B, 120
```
(loop)

- the input is made of tagged vectors of numbers
- shift-reduce strategy

→ incoming data is *shifted* to a stack

- sometimes, reductions happen:
  - a new loop is formed

```
- val A, 10
- for j=0 to 3
      val B, 100+10*j
```

- vector elements and loop bounds are affine functions of counters in scope
- the stack holds a mixture of vectors and loops

- the input is made of tagged vectors of numbers
- shift-reduce strategy

→ incoming data is *shifted* to a stack

- sometimes, reductions happen:
  - a new loop is formed
  - an existing loop gets a new iteration
- vector elements and loop bounds are affine functions of counters in scope
- the stack holds a mixture of vectors and loops

```
- val A, 10
- for j=0 to 3
      val B, 100+10*j
- val B, 130 (iter)
```

- the input is made of tagged vectors of numbers
- shift-reduce strategy

→ incoming data is *shifted* to a stack

- sometimes, reductions happen:
  - a new loop is formed
  - an existing loop gets a new iteration
- vector elements and loop bounds are affine functions of counters in scope
- the stack holds a mixture of vectors and loops

```
- val A, 10
- for j=0 to 4
      val B, 100+10*j
```

- the input is made of tagged vectors of numbers
- shift-reduce strategy

→ incoming data is *shifted* to a stack

- sometimes, reductions happen:
  - a new loop is formed
  - an existing loop gets a new iteration
- vector elements and loop bounds are affine functions of counters in scope
- the stack holds a mixture of vectors and loops

```
- val A, 10
- for j=0 to 4
      val B, 100+10*j
- val B, 140 (iter)
```

- the input is made of tagged vectors of numbers
- shift-reduce strategy

→ incoming data is *shifted* to a stack

- sometimes, reductions happen:
  - a new loop is formed
  - an existing loop gets a new iteration
- vector elements and loop bounds are affine functions of counters in scope
- the stack holds a mixture of vectors and loops

```
- val A, 10
- for j=0 to 5
    val B, 100+10*j
```

- the input is made of tagged vectors of numbers
- shift-reduce strategy

→ incoming data is *shifted* to a stack

- sometimes, reductions happen:
  - a new loop is formed
  - an existing loop gets a new iteration
- vector elements and loop bounds are affine functions of counters in scope
- the stack holds a mixture of vectors and loops

```
- val A, 10
- for j=0 to 5
      val B, 100+10*j
- val A, 20
```

- the input is made of tagged vectors of numbers
- shift-reduce strategy

→ incoming data is *shifted* to a stack

- sometimes, reductions happen:
  - a new loop is formed
  - an existing loop gets a new iteration
- vector elements and loop bounds are affine functions of counters in scope
- the stack holds a mixture of vectors and loops

```
- val A, 10
- for j=0 to 5
      val B, 100+10*j
- val A, 20
- val B, 200
```

- the input is made of tagged vectors of numbers
- shift-reduce strategy

→ incoming data is *shifted* to a stack

- sometimes, reductions happen:
  - a new loop is formed
  - an existing loop gets a new iteration
- vector elements and loop bounds are affine functions of counters in scope
- the stack holds a mixture of vectors and loops

```
- val A, 10
- for j=0 to 5
      val B, 100+10*j
- val A, 20
- val B, 200
- ...
```

- the input is made of tagged vectors of numbers
- shift-reduce strategy

→ incoming data is *shifted* to a stack

- sometimes, reductions happen:
  - a new loop is formed
  - an existing loop gets a new iteration
- vector elements and loop bounds are affine functions of counters in scope
- the stack holds a mixture of vectors and loops

```
- val A, 10
- for j=0 to 5
      val B, 100+10*j
- val A, 20
- for j=0 to 15
      val B, 200+10*j
```

- the input is made of tagged vectors of numbers
- shift-reduce strategy

→ incoming data is *shifted* to a stack

- sometimes, reductions happen:
  - a new loop is formed
  - an existing loop gets a new iteration
- vector elements and loop bounds are affine functions of counters in scope
- the stack holds a mixture of vectors and loops

```
- val A, 10
- for j=0 to 5
      val B, 100+10*j
- val A, 20
- for j=0 to 15
      val B, 200+10*j
- val A, 30
```

- the input is made of tagged vectors of numbers
- shift-reduce strategy

→ incoming data is *shifted* to a stack

- sometimes, reductions happen:
  - a new loop is formed
  - an existing loop gets a new iteration
- vector elements and loop bounds are affine functions of counters in scope
- the stack holds a mixture of vectors and loops

```
- val A, 10
- for j=0 to 5
      val B, 100+10*j
- val A, 20
- for j=0 to 15
      val B, 200+10*j
- val A, 30
- ...
```

- the input is made of tagged vectors of numbers
- shift-reduce strategy

→ incoming data is *shifted* to a stack

- sometimes, reductions happen:
  - a new loop is formed
  - an existing loop gets a new iteration
- vector elements and loop bounds are affine functions of counters in scope
- the stack holds a mixture of vectors and loops

```
- val A, 10
- for j=0 to 5
      val B, 100+10*j
- val A, 20
- for j=0 to 15
      val B, 200+10*j
- val A, 30
- for j=0 to 25 (loop!)
      val B, 300+10*j
```

- the input is made of tagged vectors of numbers
- shift-reduce strategy

→ incoming data is *shifted* to a stack

- sometimes, reductions happen:
  - a new loop is formed
  - an existing loop gets a new iteration
- vector elements and loop bounds are affine functions of counters in scope
- the stack holds a mixture of vectors and loops

```
- for i=0 to 3
    for j=0 to 5+10*i
        val B, 100+100*i+10*j
```

- the input is made of tagged vectors of numbers
- shift-reduce strategy

→ incoming data is *shifted* to a stack

- sometimes, reductions happen:
  - a new loop is formed
  - an existing loop gets a new iteration
- vector elements and loop bounds are affine functions of counters in scope
- the stack holds a mixture of vectors and loops

- incremental (the stack holds the current model)
- greedy (reduce as soon as possible)

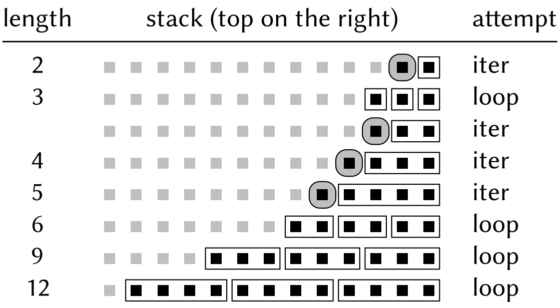
```
- for i=0 to 3
      for j=0 to 5+10*i
          val B, 100+100*i+10*j
- ...
```

Two reduction operations:

- form a *new loop* (from 3 blocks)
- recognize a *new iteration* (for an existing loop)

Search the stack:

- on increasingly long segments
- considering blocks of up to K items



(here with $K = 4$)

When

- 2+1 isomorphic blocks (syntactic criterion)
- with constants in arithmetic progression (numeric criterion)

```
          [. . .]
j=0  - val 25
     - for i = 0 to 15 { val 13 + 7i; }
j=1  - val 49
     - for i = 0 to 27 { val 19 + 7i; }
j=2  - val 73
     - for i = 0 to 39 { val 25 + 7i; }
```

Then form a new loop

Note:

- constants interpolated into affine functions
- coefficients of existing variables must match

```
          [. . .]
⇒ - for j = 0 to 3
j     val 25 + 24j
      for i = 0 to 15 + 12j { val 13 + 6j + 7i; }
```

When

- a loop on the stack, followed by
- its extrapolated next iteration

```
  [. . .]
- for j = 0 to 3
  j [ val 25 + 24j
    [ for i = 0 to 15 + 12j { val 13 + 6j + 7i; }
j=3
- val 97
- for i = 0 to 51 { val 31 + 7i; }
```

Then, increment upper bound of the loop, drop the rest

```
  [. . .]
- for j = 0 to 4
  j [ val 25 + 24j
    [ for i = 0 to 15 + 12j { val 13 + 6j + 7i; }
```

When

- a loop on the stack, followed by
- its extrapolated next iteration

```
 [. . .]
- for j = 0 to 3
  j  val 25 + 24j
     for i = 0 to 15 + 12j { val 13 + 6j + 7i; }
j=3
- val 97
- for i = 0 to 51 { val 31 + 7i; }
```

Then, increment upper bound of the loop, drop the rest

```
 [. . .]
- for j = 0 to 4
  j  val 25 + 24j
     for i = 0 to 15 + 12j { val 13 + 6j + 7i; }
```

(actually slightly more complex, because sub-loops may vanish for some iterations)

Roadmap:

- What exactly is an integer polynomial?
  - → not exactly what we thought they were…
- Interpolation and loop formation?
  - → any efficient way?
- (Recognizing iteration?)
  - → very little change expected here
- Search strategy?
  - → how much of the stack must be searched

Binomial powers

$$x^{\underline{k}|} \triangleq \binom{x}{k} = \frac{x \cdot (x-1) \cdots (x-k+1)}{k!}$$

Integer polynomials

$$p(x) = a_0 + a_1 x^{\underline{1}|} + \cdots + a_n x^{\underline{n}|} \qquad (a_i \in \mathbb{Z})$$

e.g., $7 + 3x^{\underline{1}|} + 5x^{\underline{2}|} = 7 - \frac{1}{2}x^1 + \frac{5}{2}x^2$

Binomial powers

$$x^{\underline{k}|} \triangleq \binom{x}{k} = \frac{x \cdot (x-1) \cdots (x-k+1)}{k!}$$

Interpolation of successive values

$$\begin{aligned} v_0 &= p(0) = a_0 + a_1 \cdot 0^{\underline{1}|} + a_2 \cdot 0^{\underline{2}|} + \dots \\ v_1 &= p(1) = a_0 + a_1 \cdot 1^{\underline{1}|} + a_2 \cdot 1^{\underline{2}|} + \dots \\ v_2 &= p(2) = a_0 + a_1 \cdot 2^{\underline{1}|} + a_2 \cdot 2^{\underline{2}|} + \dots \end{aligned}$$

...

(because $i^{\underline{k}|} = 0$ when $k > i$; $i^{\underline{i}|} = 1$; $i^{\underline{0}|} = 1$)

$\longrightarrow$ always a unique integer solution

Integer polynomials

$$p(x) = a_0 + a_1 x^{\underline{1}|} + \dots + a_n x^{\underline{n}|} \qquad (a_i \in \mathbb{Z})$$

e.g., $7 + 3x^{\underline{1}|} + 5x^{\underline{2}|} = 7 - \frac{1}{2}x^1 + \frac{5}{2}x^2$

Solutions

either $\begin{cases} a_0 = v_0, \\ a_i = v_i - \sum_{j=0}^{i-1} a_j \cdot i^{\underline{j}|} \quad (0 < i \leq n) \end{cases}$

or $a_i = \displaystyle\sum_{j=0}^{i} (-1)^{\underline{i-j}|} \cdot i^{\underline{j}|} \cdot v_j$

Finite difference (at any order)

$$\Delta f(x) = f(x+1) - f(x) \qquad \text{and then} \qquad \Delta^{(0)} f = f, \quad \Delta^{(d+1)} f = \Delta\left(\Delta^{(d)} f\right) \quad (d \geq 0)$$

For binomial powers and polynomials

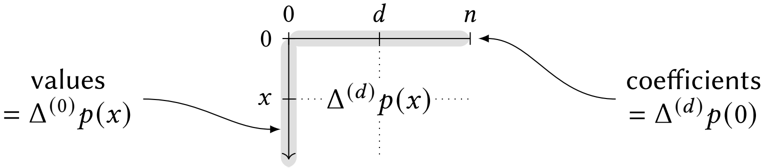
$$\Delta x^{\underline{k+1}} = x^{\underline{k}} \qquad \text{and then} \quad \Delta(a_0 + a_1 x^{\underline{1}} + \cdots + a_n x^{\underline{n}}) = a_1 + \cdots + a_n x^{\underline{n-1}}$$

$$\text{at any order} \quad \Delta^{(d)}\left(\sum_{i=0}^{n} a_i \cdot x^{\underline{i}}\right) = \sum_{i=d}^{n} a_i \cdot x^{\underline{i-d}} \qquad \Rightarrow \Delta^{(d)} p(0) = a_d$$

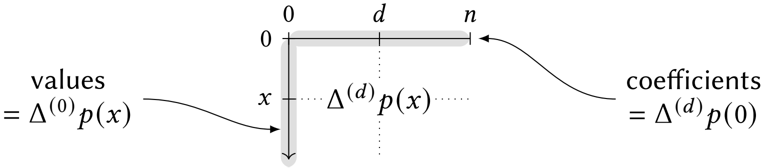
e.g.,

$$\begin{aligned} p(x) &= 7 + 3x^{\underline{1}} + 5x^{\underline{2}} \\ \Delta^{(1)} p(x) &= \phantom{7 + {}} 3 \phantom{x^{\underline{1}}} + 5x^{\underline{1}} \\ \Delta^{(2)} p(x) &= \phantom{7 + 3x^{\underline{1}} + {}} 5 \end{aligned}$$

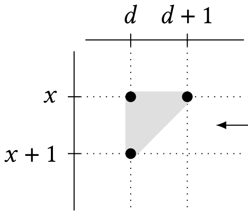
Considering all finite differences simultaneously



Considering all finite differences simultaneously



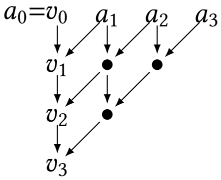
Local differentiation relation



$$\begin{array}{ccc} \Delta^{(d)} p(x) & + & \Delta^{(d+1)} p(x) \\ \shortparallel & & \\ \Delta^{(d)} p(x+1) & & \end{array}$$

Enumeration

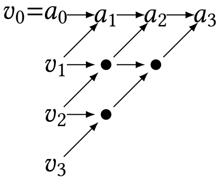
$$\Delta^{(d)}p(x+1) = \Delta^{(d)}p(x) + \Delta^{(d+1)}p(x)$$




```
// t is [a_0, . . . ,a_n]
for (i=1; i<=n; i++)
    for (j=n; j>=i; j--)
        t[j] += t[j-1];
// t is [v_0, . . . ,v_n]
```

Interpolation

$$\Delta^{(d+1)}p(x) = \Delta^{(d)}p(x+1) - \Delta^{(d)}p(x)$$

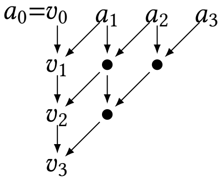



```
// t is [v_0, . . . ,v_n]
for (i=1; i<=n; i++)
    for (j=n; j>=i; j--)
        t[j] -= t[j-1];
// t is [a_0, . . . ,a_n]
```

(exactly $(n+1)^{\underline{2}}$ additions or subtractions)

Enumeration

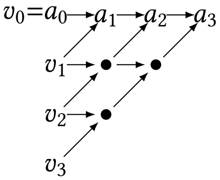
$$\Delta^{(d)}p(x+1) = \Delta^{(d)}p(x) + \Delta^{(d+1)}p(x)$$




```
// t is [a_0,...,a_n]
for (i=1; i<=n; i++)
    for (j=n; j>=i; j--)
        t[j] += t[j-1];
// t is [v_0,...,v_n]
```

Interpolation

$$\Delta^{(d+1)}p(x) = \Delta^{(d)}p(x+1) - \Delta^{(d)}p(x)$$




```
// t is [v_0,...,v_n]
for (i=1; i<=n; i++)
    for (j=n; j>=i; j--)
        t[j] -= t[j-1];
// t is [a_0,...,a_n]
```

(exactly $(n+1)^{\underline{2}}$ additions or subtractions)

- Goal: recognize polynomials wherever NLR recognizes affine functions
- Loops are arbitrarily nested
  → multivariate polynomials in all variables in scope

- First adjustment: when forming a new loop, all numbers are interpolated

```
j=0 [- for i = 0 to ... { val 13 + 5i; }
j=1 [- for i = 0 to ... { val 19 + 7i; }
j=2 [- for i = 0 to ... { val 25 + 9i; }

          [...] { val 13 + 6j + (5+2j)i; }
```

→ introduces "non-linear" terms (here 2·j·i)

- Second adjustment: consider more blocks, allow higher-degree polynomials

```
j=0 [- for i = 0 to ... { val 13 + 7i; }
j=1 [- for i = 0 to ... { val 19 + 7i; }
j=2 [- for i = 0 to ... { val 27 + 7i; }
j=3 [- for i = 0 to ... { val 37 + 7i; }
```

13	6	2	0
19	8	2	
27	10		
37			

`[...] { val` $13 + 6j^{\underline{1}} + 2j^{\underline{2}}$ `+ 7i; }`

Rule: $n+2$ blocks & degree at most $n \longrightarrow$ form a new loop
(intuition: a model must be smaller than the data it covers)

$\longrightarrow$ a new parameter D bounds the degree
(the algorithm will not consider segments with more than $D+2$ blocks)

- Recognizing new iterations does not require significant change
(only more arithmetic)

- Two independent parameters to bound complexity:
  - K (syntactic)
  - D (numeric)
- Enumerating attempts:

  for every segment length ℓ
  for every degree d between 0 and D
  if $d+2$ evenly divides ℓ and $\frac{\ell}{d+2} \leq K$
  attempt to form a new loop
- Somewhat coherent:
  - for a given block size: attempt degree d if lower degrees have failed
  - for a given degree: attempt size k is shorter blocks have failed

length	stack (top on the right)	attempt
2		loop ($d = 0$)
		iter
3		loop ($d = 1$)
		iter
4		loop ($d = 0$)
		loop ($d = 2$)
		iter
5		loop ($d = 3$)
		iter
6		loop ($d = 0$)
		loop ($d = 1$)
8		loop ($d = 0$)
		loop ($d = 2$)
[...]		
15		loop ($d = 3$)
16		loop ($d = 2$)
20		loop ($d = 3$)

An artificial example

conventional counter names

```
for i0 = 0 to 10
  val 7 + 3*i0 + 5*i0~2
  for i1 = 0 to 8 + 1*i0~2
    val 3 + 35*i0~2 + 11*i1 + 5*i0~2*i1 + 7*i0*i1~2
```

binomial power

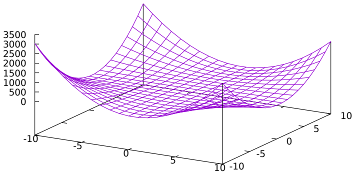
An artificial example

```
for i0 = 0 to 10
  val 7 + 3*i0 + 5*i0~2
  for i1 = 0 to 8 + 1*i0~2
    val 3 + 35*i0~2 + 11*i1 + 5*i0~2*i1 + 7*i0*i1~2
```

conventional counter names

binomial power

$x^{\underline{2}|}y^{\underline{2}|}$ over $[-10, 10]^2$



3025
2475
1980
1540
...

```
for i0 = 0 to 21
  for i1 = 0 to 21
    val 3025 - 550*i0 + 55*i0~2
        - 550*i1 + 100*i0*i1 - 10*i0~2*i1
        + 55*i1~2 - 10*i0*i1~2 + 1*i0~2*i1~2
```

$= (\mathtt{i0} - 10)^{\underline{2}|} \cdot (\mathtt{i1} - 10)^{\underline{2}|}$

normalized ranges

fully expanded form

Question: can PLR help understand memory accesses?

- An unspecified kernel:
  - with 3 parameters N, M, P
  - instrumented to trace memory accesses
- NLR/PLR single-execution model ($N = 10, M = 15, P = 20$):

```
for i0 = 0 to 10
  for i1 = 0 to 20
    for i2 = 0 to 15
      val 0x560edc3692a0 + 120*i0 + 8*i2
      val 0x560edc3699b0 + 8*i1 + 160*i2
    val 0x560edc36a0c0 + 160*i0 + 8*i1
```

$\longrightarrow$ arrays?

- Explore parameter space, build a concatenated trace:

```
for (N=10; N<15; N++)
    for (M=10; M<15; M++)
        for (P=10; P<15; P++)
            kernel (N, M, P, ...);
```

- PLR output:

```
for i0 = 0 to 5
  for i1 = 0 to 5
    for i2 = 0 to 5
      for i3 = 0 to 10 + 1*i0
        for i4 = 0 to 10 + 1*i2
          for i5 = 0 to 10 + 1*i1
            val 0x[...]92a0 + 80*i3 + 8*i1*i3 + 8*i5
            val 0x[...]99b0 + 8*i4 + 80*i5 + 8*i2*i5
          val 0x[...]a0c0 + 80*i3 + 8*i2*i3 + 8*i4
```

parameter space

parameterized kernel

- Analysis: array delinearization

address	major index range	minor index range	array?
i3*(i1+10)+i5	i3 ∈ [0,i0+10)	i5 ∈ [0,i1+10)	(i0+10)×(i1+10) $(\equiv N \times M)$
i5*(i2+10)+i4	i5 ∈ [0,i1+10)	i4 ∈ [0,i2+10)	(i1+10)×(i2+10) $(\equiv M \times P)$
i3*(i2+10)+i4	i3 ∈ [0,i0+10)	i4 ∈ [0,i2+10)	(i0+10)×(i2+10) $(\equiv N \times P)$

Question: is PLR able to recognize ranking polynomials?

- cholesky (polybench v3)

```
for (i=0; i<n; ++i) {
S1:   x = A[i][i];
      for (j=0; j<=i-1; ++j)
S2:       x -= A[i][j] * A[i][j];
S3:   p[i] = 1.0 / sqrt(x);
      for (j=i+1; j<n; ++j) {
S4:       x = A[i][j];
          for (k=0; k<=i-1; ++k)
S5:           x -= A[j][k] * A[i][k];
S6:       A[j][i] = x * p[i];
}     }
```

- trace (tagged) memory accesses
- run with $n = 256 \rightarrow 5{,}658{,}112$ entries
- post-processing: add
  - global sequence number
  - local (per-access) sequence number
- final trace:

```
S1  0x7ffec37b92a0 0 0
S3  0x7ffec38392b0 1 0
S4  0x7ffec37b92a1 2 0
S6a 0x7ffec38392b0 3 0
S6b 0x7ffec37b93a0 4 0
S4  0x7ffec37b92a2 5 1
S6a 0x7ffec38392b0 6 1
[...]
```

```
for i0 = 0 to 256
  val S1 , 0x7ffec37b92a0 + 257*i0 ,                       // tag , memory address
      767*i0 + 506*i0~2 - 4*i0~3 , 1*i0                    // global rank , local rank
  for i1 = 0 to 1*i0
    val S2 , 0x7ffec37b92a0 + 256*i0 + 1*i1 ,
        1 + 767*i0 + 506*i0~2 - 4*i0~3 + 1*i1 , 1*i0~2 + 1*i1
  val S3 , 0x7ffec38392b0 + 1*i0 ,
      1 + 768*i0 + 506*i0~2 - 4*i0~3 , 1*i0
  for i1 = 0 to 255 - 1*i0
    val S4 , 0x7ffec37b92a1 + 257*i0 + 1*i1 ,
        2 + 768*i0 + 506*i0~2 - 4*i0~3 + 3*i1 + 2*i0*i1 , 255*i0 - 1*i0~2 + 1*i1
    for i2 = 0 to 1*i0
      val S5a , 0x7ffec37b93a0 + 256*i0 + 256*i1 + 1*i2 ,
          3 + 768*i0 + 506*i0~2 - 4*i0~3 + 3*i1 + 2*i0*i1 + 2*i2 , 254*i0~2 - 2*i0~3 + 1*i0*i1 + 1*i2
      val S5b , 0x7ffec37b92a0 + 256*i0 + 1*i2 ,
          4 + 768*i0 + 506*i0~2 - 4*i0~3 + 3*i1 + 2*i0*i1 + 2*i2 , 254*i0~2 - 2*i0~3 + 1*i0*i1 + 1*i2
    val S6a , 0x7ffec38392b0 + 1*i0 ,
        3 + 770*i0 + 506*i0~2 - 4*i0~3 + 3*i1 + 2*i0*i1 , 255*i0 - 1*i0~2 + 1*i1
    val S6b , 0x7ffec37b93a0 + 257*i0 + 256*i1 ,
        4 + 770*i0 + 506*i0~2 - 4*i0~3 + 3*i1 + 2*i0*i1 , 255*i0 - 1*i0~2 + 1*i1
```

- Variants: remove some fields (among *Tag*, *Address*, *Global*, *Local*)
  → similar output as long as one of *Tag* or *Address* is included
- Using only ranks:
  → interleaved monotonously increasing counters

- Variants: remove some fields (among *Tag*, *Address*, *Global*, *Local*)
  - → similar output as long as one of *Tag* or *Address* is included
- Using only ranks:
  - → interleaved monotonously increasing counters
- Output:

```
for i0 = 0 to 5
  val 0 , 1*i0
for i0 = 0 to 254
  for i1 = 0 to 3
    val 1 + 1*i0 , 5 + 3*i0 + 1*i1
val 1 , 767
[...]
for i0 = 0 to 252
  ... (same loop as in Figure 1) ...
```

→ really (not that) bad

Background on NLR

Integer Polynomial Interpolation

Polynomial Loop Recognition

Examples

Final Remarks

- Polynomial loop recognition in traces, with a few caveats

`for j ... val` $\boxed{1 + 2j^{\underline{1}} + 3j^{\underline{2}} + 4j^{\underline{3}} + 5j^{\underline{4}} + 0j^{\underline{5}}}$

`for j ... val` $\boxed{2 + 4j^{\underline{1}} + 0j^{\underline{2}}} + \boxed{5j^{\underline{3}} + 7j^{\underline{4}} + 0j^{\underline{5}}}$ would be broken prematurely

`for j ... val` $\boxed{3 + 6j^{\underline{1}} - 3j^{\underline{2}} + 6j^{\underline{3}} + 9j^{\underline{4}} + 0j^{\underline{5}}}$

- Polynomial loop recognition in traces, with a few caveats

  `for j ... val` $\boxed{1 + 2j^{\underline{1}} + 3j^{\underline{2}} + 4j^{\underline{3}} + 5j^{\underline{4}} + 0j^{\underline{5}}}$

  `for j ... val` $\boxed{2 + 4j^{\underline{1}} + 0j^{\underline{2}}} + \boxed{5j^{\underline{3}} + 7j^{\underline{4}} + 0j^{\underline{5}}}$ would be broken prematurely

  `for j ... val` $\boxed{3 + 6j^{\underline{1}} - 3j^{\underline{2}} + 6j^{\underline{3}} + 9j^{\underline{4}} + 0j^{\underline{5}}}$

- Binomial powers & integer polynomials are crucial enablers (again)
  - last year: integration & counting;
  - this year: differentiation & interpolation

- Polynomial loop recognition in traces, with a few caveats

`for j ... val` $\boxed{1 + 2j^{\underline{1}\rfloor} + 3j^{\underline{2}\rfloor} + 4j^{\underline{3}\rfloor} + 5j^{\underline{4}\rfloor} + 0j^{\underline{5}\rfloor}}$

`for j ... val` $\boxed{2 + 4j^{\underline{1}\rfloor} + 0j^{\underline{2}\rfloor}} + \boxed{5j^{\underline{3}\rfloor} + 7j^{\underline{4}\rfloor} + 0j^{\underline{5}\rfloor}}$ would be broken prematurely

`for j ... val` $\boxed{3 + 6j^{\underline{1}\rfloor} - 3j^{\underline{2}\rfloor} + 6j^{\underline{3}\rfloor} + 9j^{\underline{4}\rfloor} + 0j^{\underline{5}\rfloor}}$

- Binomial powers & integer polynomials are crucial enablers (again)
  - last year: integration & counting;
  - this year: differentiation & interpolation

- Polynomial loops? Is this a thing?
  - \+ counts, ranks, all forms of accumulation
  - – polynomial bounds: never seen one, never written one

Final Remarks

- Polynomial loop recognition in traces, with a few caveats

  $\texttt{for j ... val}\ \boxed{1 + 2j^{\underline{1}} + 3j^{\underline{2}} + 4j^{\underline{3}} + 5j^{\underline{4}} + 0j^{\underline{5}}}$

  $\texttt{for j ... val}\ \boxed{2 + 4j^{\underline{1}} + {\color{red}0}j^{\underline{2}}} + \boxed{5j^{\underline{3}} + 7j^{\underline{4}} + 0j^{\underline{5}}}$ would be broken prematurely

  $\texttt{for j ... val}\ \boxed{3 + 6j^{\underline{1}} - 3j^{\underline{2}} + 6j^{\underline{3}} + 9j^{\underline{4}} + 0j^{\underline{5}}}$

- Binomial powers & integer polynomials are crucial enablers (again)
  - last year: integration & counting;
  - this year: differentiation & interpolation

- Polynomial loops? Is this a thing?
  + counts, ranks, all forms of accumulation
  - polynomial bounds: never seen one, never written one
  + *may* be useful for analysis; e.g.,

    every affine (polynomial) loop *nest* has
    an equivalent polynomial *perfect* loop