Scalar Evolution - Demystified

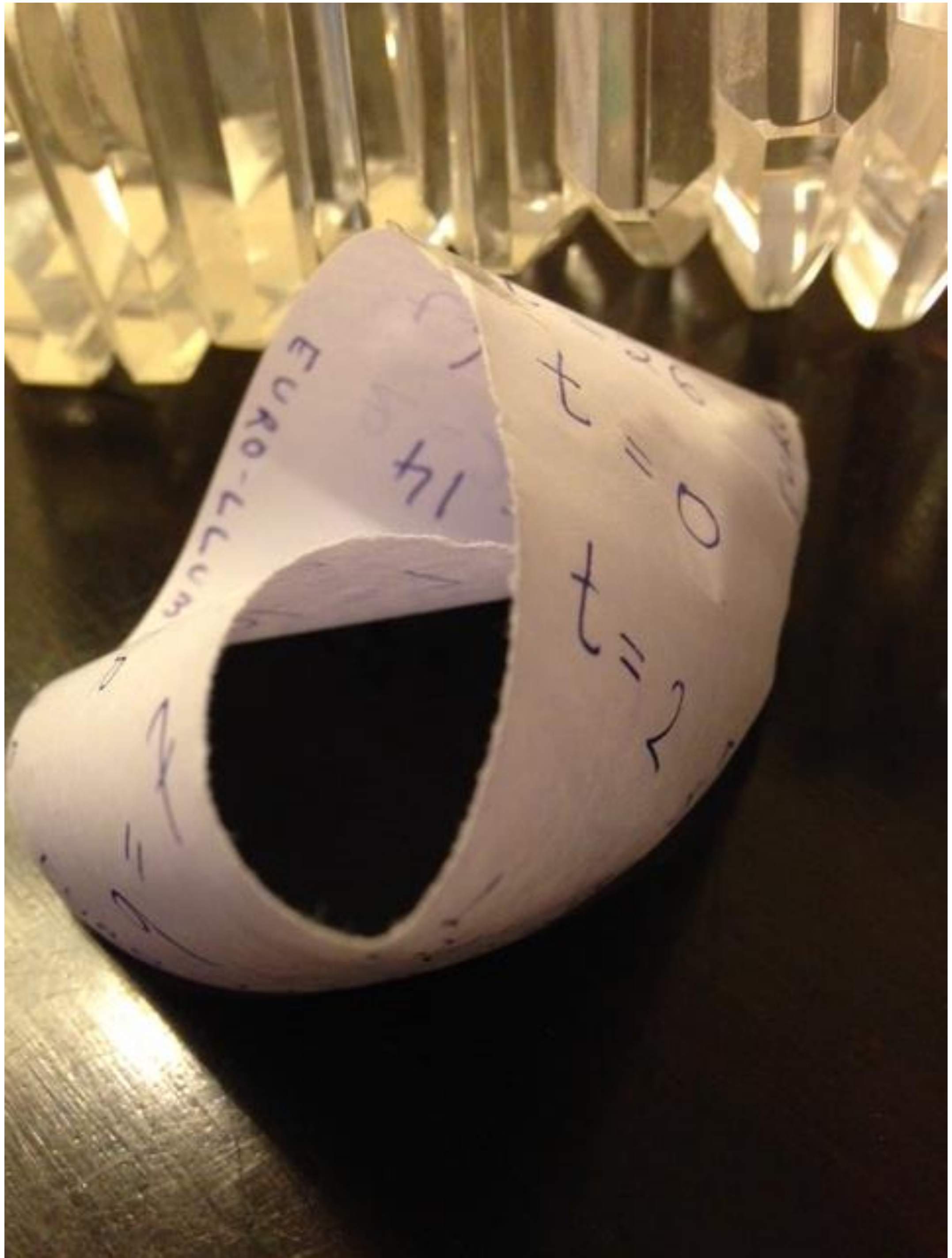
Javed Absar (javed.absar@arm.com)

Contents

- Introduction
- Mathematical Framework
- Scalar Evolution (SCEV) Implementation in LLVM
- SCEV as a Service
- Some Additional Topics
- Conclusion

Introduction – Scalar Evolution

arm



Scalar Evolution:
Change in the Value of Scalar Variables
Over Iterations of the Loop

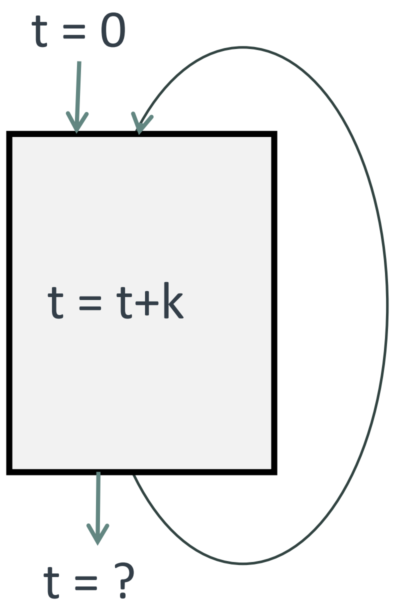
- unknown compiler engineer

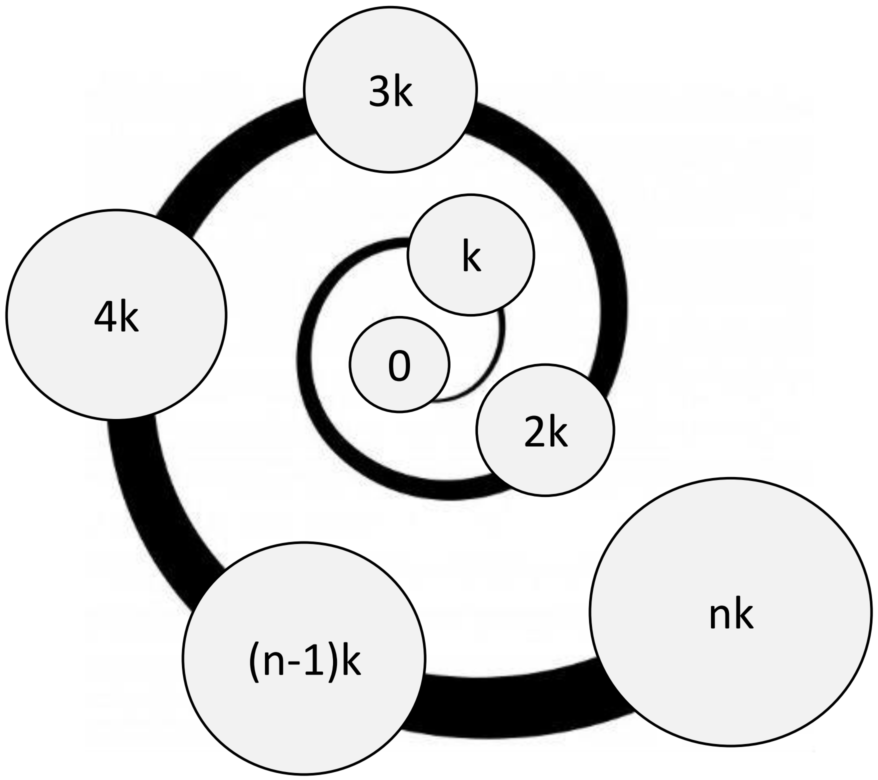
Introduction – Scalar Evolution

- Powerful symbolic technique
- LLVM SCEV - Practical implementation
- Passes using SCEV
  - Loop strength reduction (LSR)
  - Induction Variable Simplify (IndVars)
  - Loop Vectorizer, SLP Vectorizer, Load Store Vectorizer, Re-associate nary expr
  - Loop Access Analysis, Dependence Analysis, SCEV-AA

Introduction – Scalar Evolution

```
void foo(int *a, int n, int k) {
  int t = 0;
  for (int i = 0; i < n; i++)
     t = t + k;
  *a = t;
}
```





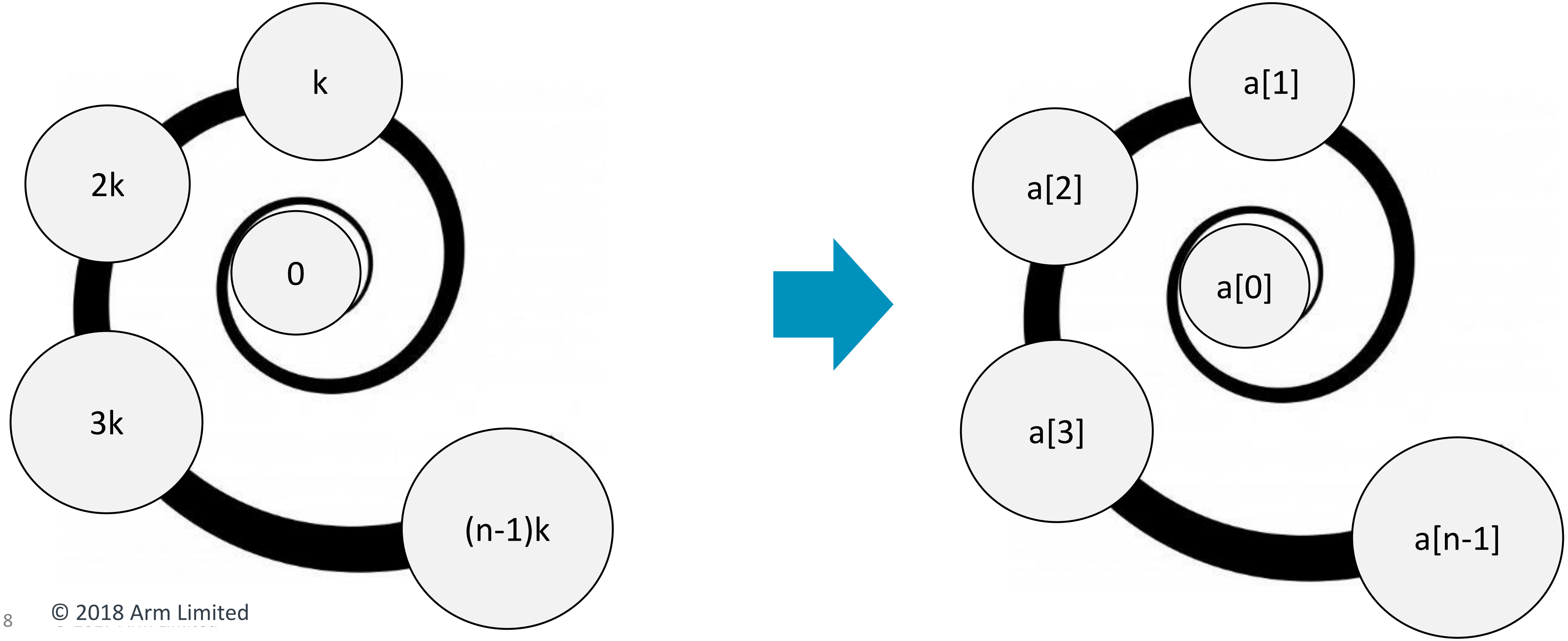
Introduction – Scalar Evolution

```
1. for.body:
2.   %i = phi i32 [ %inc, %for.body ], [ 0, %for.body.preheader ]
3.   %t = phi i32 [ %tk, %for.body ], [ 0, %for.body.preheader ]
4.   %tk = add nsw i32 %t, %k
5.   %inc = add nuw nsw i32 %i, 1
6.   %cmp = icmp slt i32 %inc, %n
7.   br i1 %cmp, label %for.body, label %for.cond.cleanup
…
for.cond.cleanup:
  %tk.final = phi i32 [ 0, %entry ], [ %tk, %for.body]
8.  store i32 %tk.final, i32* %a
…
```

```
*** IR Dump After Induction Variable Simplification ***
…
1’. for.cond.cleanup.loopexit:
2’.   %tk.final = mul i32 %n, %k
3’. store i32 %tk.final, i32* %a
```

Introduction – Scalar Evolution

```
int foo(int *a, int n, int k) {
  for (int i = 0; i < n; i++)
    a[i] = i*k;
}
```



Introduction – Scalar Evolution

```
int foo(int *a, int n, int k) {
  for (int i = 0; i < n; i++)
    a[i] = i*k;
}
```

```
1. *** IR Dump After Canonicalize natural loops ***
2. for.body:
3.   %i = phi i32 [ %inc, %for.body ], [ 0, %for.body.preheader ]
4.   %ik = mul nsw i32 %i, %k
5.  %arrayidx = getelementptr inbounds i32, i32* %a, i32 %i
6.   store i32 %ik, i32* %arrayidx
7.   …
```

```
1'. *** IR Dump After Loop Strength Reduction ***
2'. for.body:
3'.   %IV_IK = phi i32 [ 0, %for.body.preheader ], [ %IV_IK_plus_K, %for.body ]
4'.   store i32 %IV_IK, i32* %lsr.iv
5'.   %lsr.iv.next = add i32 %lsr.iv7, -1
6'.   %IV_IK_plus_K = add i32 %IV_IK, %k
```


arm

Mathematical Framework - Chain of Recurrences

arm

Induction Variable

Basic Induction Variable (BIV):

- Increases or decreases by a constant on each iteration of the loop

Generalized Induction Variable (GIV)

- Update value is not constant
- Dependent on other BIVs/GIVs (linear, non-linear), multiple update

Chain of Recurrences

Formalism to analyse expressions in BIV and GIV

Express as Recurrences

Factorial

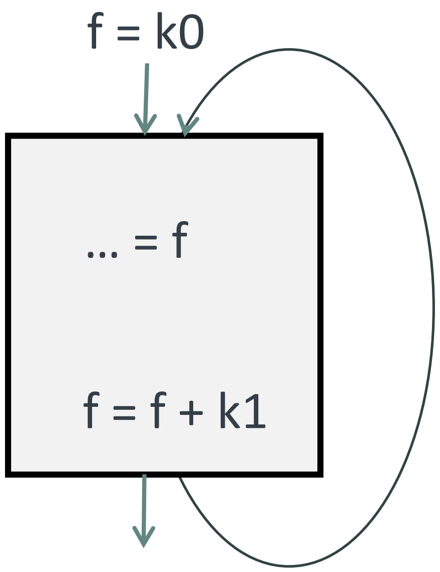
n! = 1 x 2 x … n $\Leftrightarrow$ n! = (n-1)! x n

$$f(n) = \prod_{k=1}^{n} k\,. \quad \Leftrightarrow \quad f(n) = f(n-1) * n$$

Basic Recurrences

k_0, k_1 are loop-invariants

```
int f = k0;
for (int i = 0; i < n; i++) {
   … = f ;
   f = f + k1;
}
```



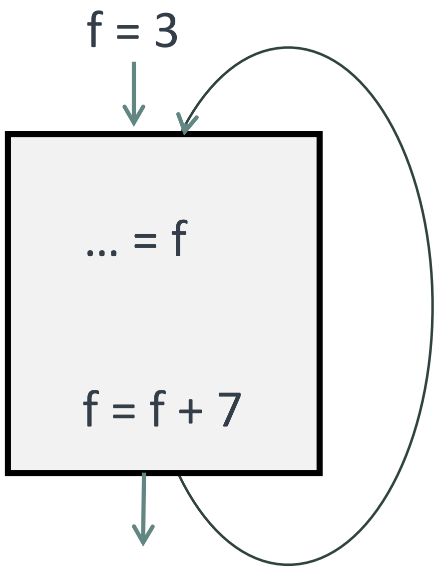
$$f(i) = \begin{cases} k_0 & \text{if } i = 0 \\ f(i-1) + k_1 & \text{if } i > 0 \end{cases}$$

$$basic\ recurrence = \{k_0, +, k_1\}_i$$

Basic Recurrences

$$BR = \{3, +, 7\}_i$$

Generates: 3 10 17 24 31 …




```
int f = 3;
for (int i = 0; i < n; i++) {
   … = f ;
   f = f + 7;
}
```

$$f(i) = \begin{cases} 3 & \text{if } i = 0 \\ f(i-1) + 7 & \text{if } i > 0 \end{cases}$$

Basic Recurrence – Example

```
int foo(int *a, int n, int k){
  for (int i = 0; i < n; i++)
    a[i] = i*k;
}
```

```
$ opt -analyze -scalar-evolution foo.ll
```

```
1. Printing analysis 'Scalar Evolution Analysis' for function 'foo':
2. Classifying expressions for: @foo
3. …
4. %mul = mul nsw i32 %i, %k
5.  -->  {0,+,%k}<%for.body>  Exits: ((-1 + %n) * %k)
6. …
```

Chain Recurrences

$$CR = \underbrace{\{1, +, \underbrace{\{3, +, 7\}}_{f_1(i)}\}}_{f_0(i)}$$

$$f_1(i) = \left\{\begin{array}{ll} 3 & \text{if } i = 0 \\ f_1(i-1) + 7 & \text{if } i > 0 \end{array}\right\}$$

$$f_0(i) = \left\{\begin{array}{ll} 1 & \text{if } i = 0 \\ f_0(i-1) + f_1(i-1) & \text{if } i > 0 \end{array}\right\}$$

$$CR = \{1, +, \{3, +, 7\}\} \Leftrightarrow \{1,+,3,+,7\}$$

Chain Recurrences

```
for ( int x = 0; x < n; x++)
   p[x] = x*x*x  + 2*x*x + 3*x + 7;
```

x	0	1	2	3	4	5
p(x)	7	13	29	61	115	197
Δ	-	6	16	32	54	82
Δ^2	-	-	10	16	22	28
Δ^3	-	-	-	6	6	6

$$CR = \{7, +, 6, +, 10, +, 6\}$$

Chain Recurrences

$$CR = \{7, +, 6, +, 10, +, 6\}$$

$f_0(i)$	$f_1(i)$	$f_2(i)$	$f_3(i)$
7 +6	6 +10	10 +6	6
13 +16	16 +16	16 +6	6
29 +32	32 +22	22 +6	6
61	54	28	6

	$f_0(0)$	$f_0(1)$	$f_0(2)$	$f_0(3)$
$p(x) =$	7	13	29	61

Chain of Recurrences

```
void foo(int *p, int n){
  for( int x = 0; x < n; x++)
       p[x] = x*x*x  + 2*x*x + 3*x + 7;
}
```

IV Chain#0 Head: (store i32 %add6, i32* %arrayidx.) **SCEV={7,+,6,+,10,+,6}<%for.body>**

```
void foo(int *p, int n) {
   int t0 = 7;
   int t1 = 6;
   int t2 = 10;
   for(int x = 0; x < n; x++) {
     p[x] = t0;
     t0 = t0 + t1;
     t1 = t1 + t2;
     t2 = t2 + 6;
     //p[x] = x*x*x  + 2*x*x + 3*x + 7;
   }
}
```

Chain of Recurrences - Synopsis

```
for (i=0; i< n; i++)
   … = i*k;
```

$\{0,+,\%k\}_i$

SCEV Analysis

SCEV Rewriting Rules and Implementation in LLVM

SCEV Rewriting/Folding

```
void foo(int *a, short k, int p, int n) {
  for (int i = 0; i < n; i++)
    a[i] = (k+i)*p;
}
```

```
 %i = phi i32 [ 0, %entry], [ %inc, %for.body ]
.
.
.

%inc = add nuw nsw i32 %i, 1
```

%i = phi i32 [%inc, %for.body], [0, %entry]

--> {0,+,1}<nuw><nsw><%for.body

SCEV Rewriting/Folding

i: $\{7, +, 3\}$	j: $\{1, +, 1\}$	k=i+j
7	1	8
10	2	12
13	3	16

$$k = i + j = \{7 + 1, +, 3 + 1\} = \{8, +4\}$$

$$\boxed{\{e, +, f\} + \{g, +, h\} \quad \Rightarrow \quad \{e + g, +, f + h\}}$$

SCEV Rewriting/Folding

Expression		Rewrite	Example		
$G + \{e,+,f\}$	$\Rightarrow$	$\{G+e,+,f\}$	$12 + \{7,+,3\}$	$\Rightarrow$	$\{19,+,3\}$
$G * \{e,+,f\}$	$\Rightarrow$	$\{G*e,+,G*f\}$	$12 * \{7,+,3\}$	$\Rightarrow$	$\{84,+,36\}$
$\{e,+,f\} + \{g,+,h\}$	$\Rightarrow$	$\{e+g,+,f+h\}$	$\{7,+,3\} + \{1,+,1\}$	$\Rightarrow$	$\{8,+,4\}$
$\{e,+,f\} * \{g,+,h\}$	$\Rightarrow$	$\left\{\begin{matrix} e*g,+, \\ e*h+f*g+f*h, \\ +,2*f*h \end{matrix}\right\}$	$\{0,+,1\} * \{0,+,1\}$	$\Rightarrow$	$\{0,+,1,+,2\}$

arm


FOLDING SCEV EXPRESSIONS

SCEV Rewriting/Folding

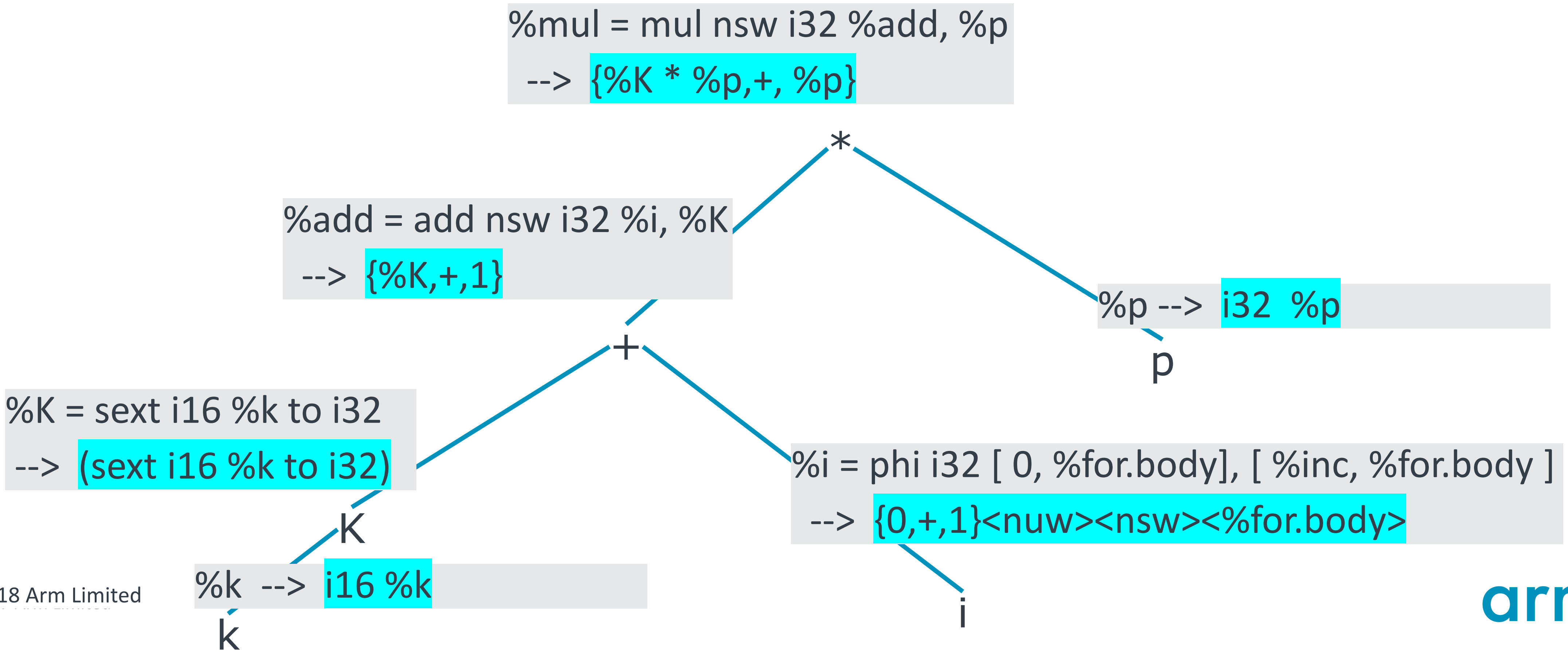
```
void foo(int *a, short k, int p, int n) {
  for (int i = 0; i < n; i++)
    a[i] = (k+i)*p;
}
```

```
for.body.lr.ph:                                   ; preds = %entry
  %K = sext i16 %k to i32
  br label %for.body

for.body:                                         ; preds = %for.body, %for.body.lr.ph
  %i = phi i32 [ 0, %for.body.lr.ph ], [ %inc, %for.body ]
  %add = add nsw i32 %i, %K
  %mul = mul nsw i32 %add, %p
  %arrayidx = getelementptr inbounds i32, i32* %a, i32 %i
  store i32 %mul, i32* %arrayidx
  %inc = add nuw nsw i32 %i, 1
  %exitcond = icmp eq i32 %inc, %n
  br i1 %exitcond, label %for.cond.cleanup, label %for.body
```

SCEV Rewriting/Folding

```
void foo(int *a, short k, int p, int n) {
  for (int i = 0; i < n; i++)
    a[i] = (k+i)*p;
}
```



arm

Rewriting Example

$p(x) = x^3 + 2x^2 + 3x + 7$

$= \{0,+,1\}^3 + 2 * \{0,+,1\}^2 + 3 * \{0,+,1\} + 7$

= {0,+,1,+,6,+,6} + 2*{0,+,1,+,2} + {0,+,3} + 7

= {0,+,1,+,6,+,6} + {0,+,2,+,4} + {7,+,3}

= {0,+,1,+,6,+,6} + {0,+,2,+,4} + {7,+,3}

= {0,+,3,+,10,+,6} + {7,+,3}

= {7,+,6,+10,+,6}

SCEV Rewriting/Folding

$$(i+1)^2 = i^2 + 2i + 1$$
$$\Rightarrow\ (i+1)^2 - i^2 - 2i = 1$$

```
void foo(int *a) {
  for (int i = 0; i < 100; i++)
    a[i] = (i+1)*(i+1) - i*i - 2*i;
}
```

```
*** IR Dump After Loop-Closed SSA Form Pass ***
for.body:                                          ; preds = %entry, %for.body
  %i = phi i32 [ 0, %entry ], [ %add, %for.body ]
  %add = add nuw nsw i32 %i, 1                     ; %add = i+1
  %mul = mul nsw i32 %add, %add                    ; %mul = (i+1)*(i+1)
  %mul314 = add nuw i32 %i, 2                      ; %mul314 = i+2
  %0 = mul i32 %i, %mul314                         ; %0 = i*(i+2) = i*i + 2*i
  %sub4 = sub i32 %mul, %0                         ; %sub4 = (i+1)*(i+1) - i*i - 2*i
  %arrayidx = getelementptr inbounds i32, i32* %a, i32 %i
  store i32 %sub4, i32* %arrayidx, align 4, !tbaa !3
  %cmp = icmp ult i32 %add, 100
  br i1 %cmp, label %for.body, label %for.cond.cleanup
}
```

SCEV Rewriting/Folding

```
2. %add = add nuw nsw i32 %i, 1                ; %add = i+1
3. %mul = mul nsw i32 %add, %add               ; %mul = (i+1)*(i+1)
4. %mul314 = add nuw i32 %i, 2                 ; %mul314 = i+2
5. %0 = mul i32 %i, %mul314                    ; %0 = i*(i+2) = i*i + 2*i
6. %sub4 = sub i32 %mul, %0                    ; %sub4 = (i+1)*(i+1) - i*i - 2*i
```

```
1. %i = phi i32 [ 0, %entry ], [ %add, %for.body ] => scev = ({0,+,1}<nuw><nsw><%for.body>)
2. %add = add nuw nsw i32 %i, 1 => scev = ({1,+,1}<nuw><nsw><%for.body>)
3. %mul = mul nsw i32 %add, %add => scev = ({1,+,3,+,2}<%for.body>)
4. %mul314 = add nuw i32 %i, 2 => scev = ({2,+,1}<nuw><nsw><%for.body>)
5. %0 = mul i32 %i, %mul314 => scev = ({0,+,3,+,2}<%for.body>)
6.  %sub4 = sub i32 %mul, %0 => scev = (1)
```

SCEV Rewriting/Folding

```
void foo(int *a) {
  for (int i = 0; i < 100; i++)
    a[i] = (i+1)*(i+1) - i*i - 2*i;  // equals 1
}
```

```
1. *** IR Dump After Induction Variable Simplification ***
2. for.body:
3.   %i = phi i32 [ 0, %entry ], [ %add, %for.body ]
4.   %add = add nuw nsw i32 %i, 1
 5.  %arrayidx = getelementptr inbounds i32, i32* %a, i32 %i
 6.  store i32 1, i32* %arrayidx
 7.  %exitcond = icmp ne i32 %add, 100
  8. br i1 %exitcond, label %for.body, label %for.cond.cleanup
```

SCEV Rewriting/Folding

```
void foo(char a[100][100], char b[100][100], int p, int k) {
  int i, j;
  for (i = 0; i < 100; i++ )
    for (j = 0; j < 10; j++ )
       a[i][p*j+k] = b[i][j*j];
}
```

```
$ opt -analyze -scalar-evolution foo.ll
```

```
 %i = phi i32 [ 0, %entry ], [ %inc9, %for.i.loop.inc ]
  -->  {0,+,1}<nuw><nsw><%for.i.loop.header> U: [0,100) S: [0,100)          Exits: 99

 %j = phi i32 [ 0, %for.i.loop.header ], [ %inc, %for.j.loopbody ]
  -->  {0,+,1}<nuw><nsw><%for.j.loopbody> U: [0,10) S: [0,10)          Exits: 9

 %index_b = getelementptr inbounds [100 x i8], [100 x i8]* %b, i32 %i, i32 %mul
  -->  {{%b,+,100}<nsw><%for.i.loop.header>,+,1,+,2}<%for.j.loopbody>

%index_a = getelementptr inbounds [100 x i8], [100 x i8]* %a, i32 %i, i32 %add
  -->  {{(%k + %a)<nsw>,+,100}<nw><%for.i.loop.header>,+,%p}<nw><%for.j.loopbody>
```

SCEV Expression – Interface

SCEV Analysis APIs

- getSCEV(Value)
- getAddExpr(Ops)
- getMulExpr(Ops)

SCEV – expression classes

- SCEVConstant, SCEVCastExpr, SCEVAddExpr, SCEVMulExpr, SCEVDivExpr, **SCEVAddRecExpr**, SCEVUnknown

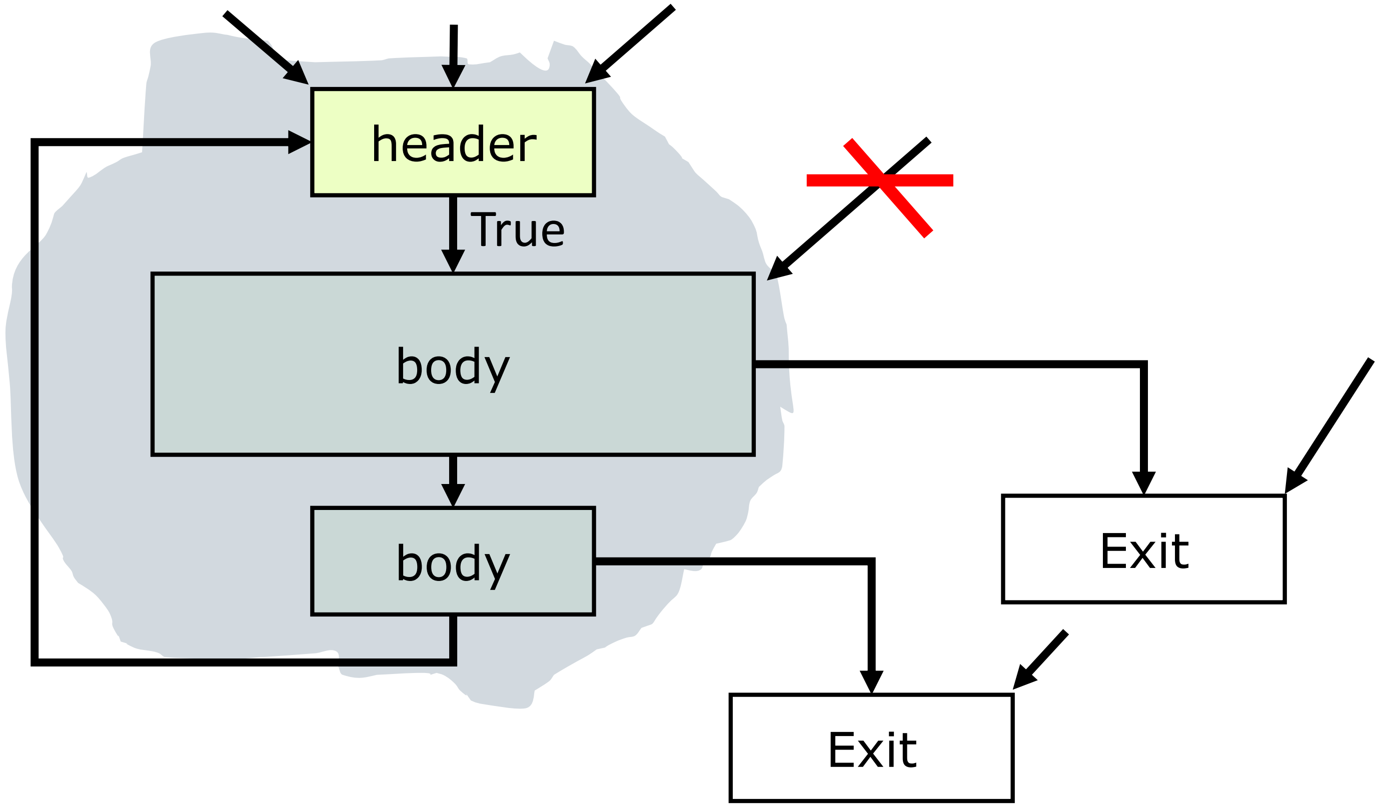
Normal Form- Can compare two SCEV pointers for equivalence

Loops and Loop Optimizations using SCEV

arm

Natural Loop

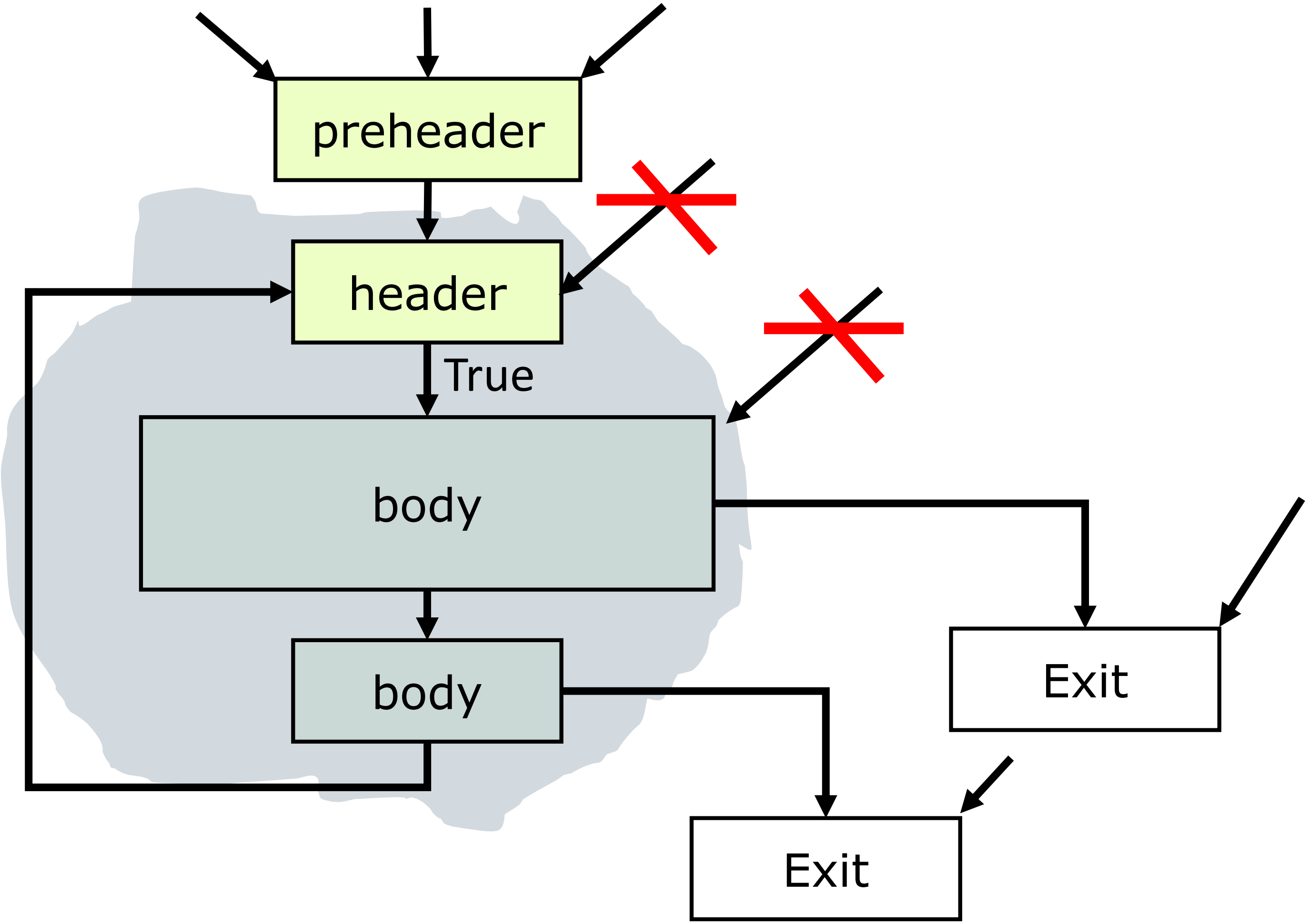
Natural Loop – exactly one unique entry point



arm

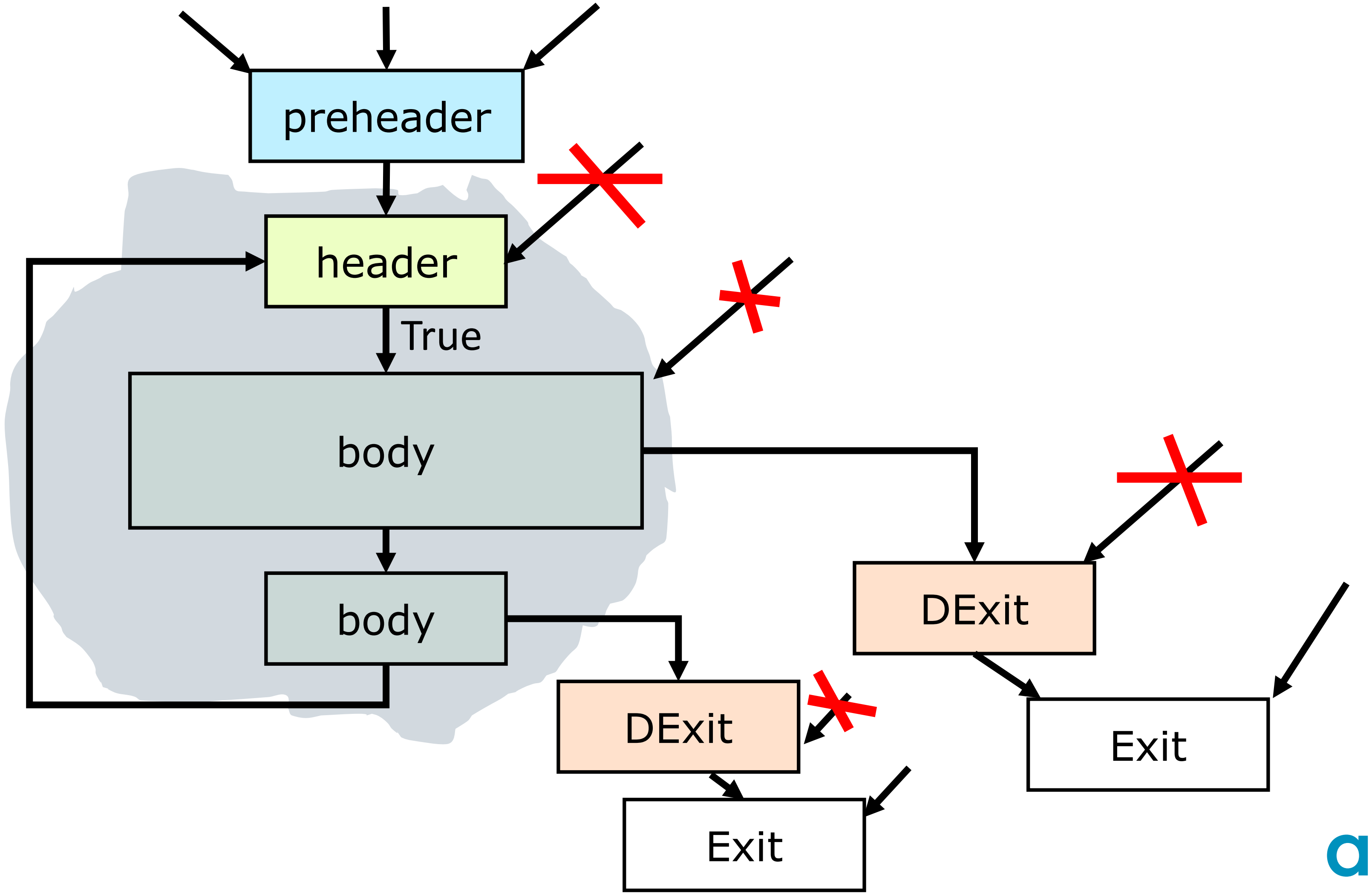
Canonical Loop

1. Pre-header



Canonical Loop

1. Pre-header
2. Dedicated Exit
3. Single Backedge



SCEV As a Service – Loop Strength Reduce, Vectorizer, Loop Access Analysis

arm

SCEV User - Loop Strength Reduce (LSR)

Hoist loop-invariant computations outside loop

Replace multiply with add

Loop Strength Reduce

Check Loop Form

Collect Chains

Collect Types and Factors

Generate Formulas

Solve and Implement

Loop Strength Reduction

```
void foo(int *a, int n, int c, int k) {
  for (int i = 3; i < n; i++) {
    a[c*i] =  c*i+k;
  }
}
```

```
for.body:                                        ; preds = %entry, %for.body
  %i = phi i32 [ %inc, %for.body ], [ 3, %entry ]
  %ci = mul nsw i32 %i, %c
  %ci_plus_k = add nsw i32 %ci, %k
  %arrayidx = getelementptr inbounds i32, i32* %a, i32 %ci
  store i32 %ci_plus_k, i32* %arrayidx, align 4, !tbaa !3
  %inc = add nuw nsw i32 %i, 1
  %exitcond = icmp eq i32 %inc, %n
  br i1 %exitcond, label %for.cond.cleanup, label %for.body
}
```

LSR. Collect Chains

```
void foo(int *a, int n, int c, int k) {
  for (int i = 3; i < n; i++) {
    a[c*i] =  c*i+k;
  }
}
```

```
store i32 %ci_plus_k, i32* %arrayidx
```

```
%ci_plus_k = add nsw i32 %ci, %k
--> scev(%ci_plus_k) = {(3*%c+%k),+,%c}
```

```
%arrayidx = getelementptr inbounds i32, i32* %a, i32 %ci
-->SCEV(%arrayidx) = {(12*%c +%a),+,(4*%c)}
```

```
%ci = mul nsw i32 %i, %c
  -->  SCEV(%ci) = {(3*%c),+,%c}
```

```
%i = phi i32 [ %inc, %for.body ], [ 3, %entry ]
  -->  scev(%i) = {3,+,1}
```

LSR. Collect Chains

```
void foo(int *a, int n, int c, int k) {
  for (int i = 3; i < n; i++) {
    a[c*i] =  c*i+k;
  }
}
```

```
Collecting IV Chains.
IV Chain#0 Head: (  store i32 %ci_plus_k, i32* %arrayidx, align 4, !tbaa !3)
                 IV={((3 * %c) + %k),+,%c}<nw><%for.body>

IV Chain#1 Head: (  store i32 %ci_plus_k, i32* %arrayidx, align 4, !tbaa !3)
                 IV={((12 * %c) + %a)<nsw>,+,(4 * %c)}<nsw><%for.body>

IV Chain#2 Head: (  %exitcond = icmp eq i32 %inc, %n)
                 IV={4,+,1}<nuw><nsw><%for.body>
```

LSR. Collect Fixups and Formula

```c
void foo(int *a, int n, int c, int k) {
  for (int i = 3; i < n; i++) {
    a[c*i] =  c*i+k;
  }
}
```

```
reg({((12 * %c) + %a)<nsw>,+,(4 * %c)}<nsw><%for.body>)

LSR Use: Kind=Address of i32 in addrspace(0), Offsets={0}, widest fixup type: i32*
    reg({((12 * %c) + %a)<nsw>,+,(4 * %c)}<nsw><%for.body>)
    reg((12 * %c)) + 1*reg({%a,+,(4 * %c)}<%for.body>)
    reg((12 * %c)) + reg(%a) + 1*reg({0,+,(4 * %c)}<%for.body>)
    reg(%a) + 1*reg({(12 * %c),+,(4 * %c)}<nsw><%for.body>)
    reg(((12 * %c) + %a)<nsw>) + 1*reg({0,+,(4 * %c)}<%for.body>)
    -1*reg({((-12 * %c) + (-1 * %a)),+,(-4 * %c)}<nw><%for.body>)
    reg((12 * %c)) + -1*reg({(-1 * %a),+,(-4 * %c)}<%for.body>)
    reg((12 * %c)) + reg(%a) + 4*reg({0,+,%c}<%for.body>)
    reg((12 * %c)) + reg(%a) + -4*reg({0,+,(-1 * %c)}<%for.body>)
    reg((12 * %c)) + reg(%a) + -1*reg({0,+,(-4 * %c)}<%for.body>)
    reg(%a) + 4*reg({(3 * %c),+,%c}<nsw><%for.body>)
    reg(%a) + -4*reg({(-3 * %c),+,(-1 * %c)}<%for.body>)
    reg(%a) + -1*reg({(-12 * %c),+,(-4 * %c)}<nw><%for.body>)
    reg(((12 * %c) + %a)<nsw>) + 4*reg({0,+,%c}<%for.body>)
    reg(((12 * %c) + %a)<nsw>) + -4*reg({0,+,(-1 * %c)}<%for.body>)
    reg(((12 * %c) + %a)<nsw>) + -1*reg({0,+,(-4 * %c)}<%for.body>)
  LSR Use: Kind=Basic, Offsets={0}, widest fixup type: i32
```

LSR. Solve – Choose Formula

```
void foo(int *a, int n, int c, int k) {
  for (int i = 3; i < n; i++) {
    a[c*i] =  c*i+k;
  }
}
```

```
Chosen solution requires 3 instructions 5 regs, with addrec cost 2, plus 1 base add, plus 2 setup cost:

  1. LSR Use: Kind=ICmpZero, Offsets={0}, widest fixup type: i32
     reg({(-3 + %n),+,-1}<nw><%for.body>)

  2. LSR Use: Kind=Address of i32 in addrspace(0), Offsets={0}, widest fixup type: i32*
     reg(%a) + 4*reg({(3 * %c),+,%c}<nsw><%for.body>)

  3. LSR Use: Kind=Basic, Offsets={0}, widest fixup type: i32
     reg(%k) + 1*reg({(3 * %c),+,%c}<nsw><%for.body>)
```

LSR. Implement

```
void foo(int *a, int n, int c, int k) {
  for (int i = 3; i < n; i++) {
    a[c*i] =  c*i+k;
  }
}
```

Chosen solution requires 3 instructions 5 regs, with addrec cost 2, plus 1 base add, plus 2 setup cost:
reg(%a) + 4*reg({(3 * %c),+,%c}<nsw><%for.body>)

```
for.body.preheader:                              ; preds = %entry
  %0 = add i32 %n, -3
  %1 = mul i32 %c, 3
  br label %for.body

for.body:                                    ; preds = %for.body.preheader, %for.body
  %lsr.iv1 = phi i32 [ %1, %for.body.preheader ], [ %lsr.iv.next2, %for.body ]
  %lsr.iv = phi i32 [ %0, %for.body.preheader ], [ %lsr.iv.next, %for.body ]
  %2 = add i32 %k, %lsr.iv1
  %scevgep = getelementptr i32, i32* %a, i32 %lsr.iv1
  store i32 %2, i32* %scevgep
  %lsr.iv.next = add i32 %lsr.iv, -1
  %lsr.iv.next2 = add i32 %lsr.iv1, %c
  %exitcond = icmp eq i32 %lsr.iv.next, 0
  br i1 %exitcond, label %for.cond.cleanup.loopexit, label %for.body
}
```

SCEV User - Dependence Analysis

Strong SIV Test:

Dependence($A[c_1+stride*i]$, $A[c_2+stride*i]$)

Dependence-distance: $d = i' - i = (c_1-c_2)/stride$ (Eq. 1)

```
void foo(int *A, int n) {
  for (int i = 0; i < 100; i++)
    A[2*i+1] = A[2*i];
}
```

```
testing subscript 0, SIV
    src = {0,+,2}<nuw><nsw><%for.body>
    dst = {1,+,2}<nuw><nsw><%for.body>
        Strong SIV test
            Stride = 2, i32
            C1 = 0, i32
            C2 = 1, i32
            Delta = -1, i32
               .
da analyze - none!
```

```
1. bool strongSIVtest(const SCEV *Stride, const SCEV *c1,
                                      const SCEV *c2, ...) const {
  ...
2.  const SCEV *Delta = SE->getMinusSCEV(c1, c2);

3. // Can we compute distance?
4. if (isa<SCEVConstant>(Delta) &&
      isa<SCEVConstant>(Stride)) {  ....
  }
```

SCEV User - Vectorizers

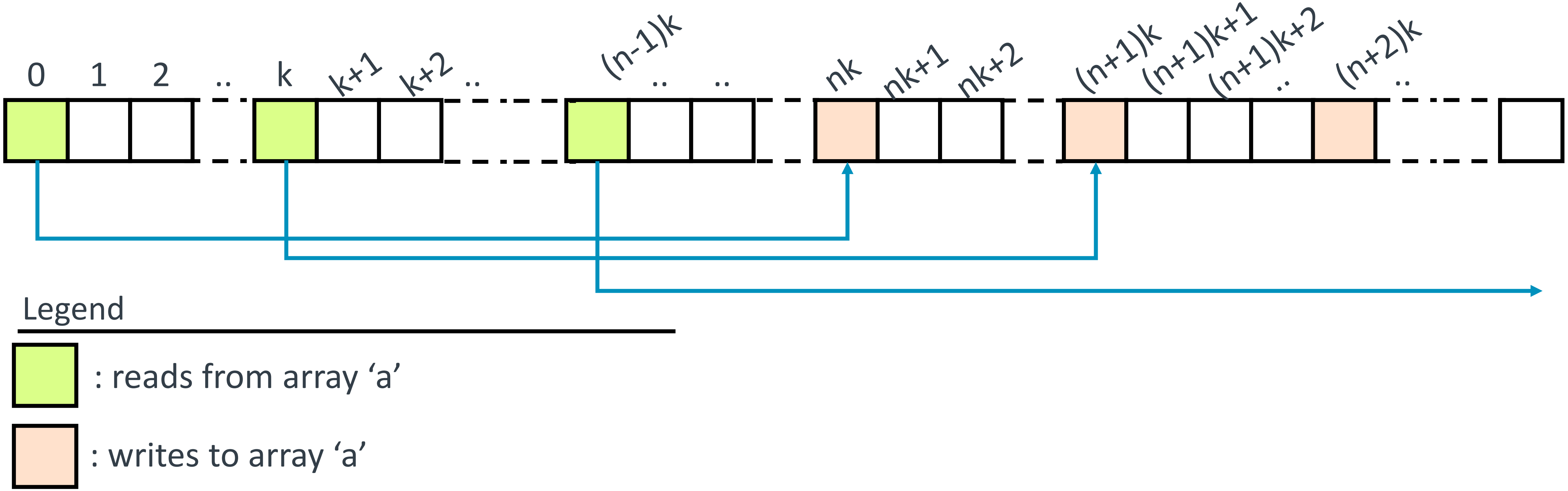
Vectorizers – Loop Vectorizer, SLP, Load-Store Vectorizer

Use SCEV for

- Induction variable (step loop-invariant)
- Trip count
- Loop Access Analysis

Loop Access Analysis

```
void foo(char *a, unsigned n, unsigned k) {
  for (unsigned i = 0; i < n; i++)
    a[k*i+n*k] = a[k*i];
}
```



Loop Access Analysis

```
void foo(char *a, unsigned n, unsigned k) {
  for (unsigned i = 0; i < n; i++)
    a[k*i+n*k] = a[k*i];
}
```

READS: SCEV(a[k*i]) = {%a,+,%k}
WRITES: SCEV(a[k*i+n*k) = {%n * %k + %a,+,%k}

- Are strides same?
- If strides are same, loop-invariant. are they – constant (1,2,3,…)? symbolic?

1. LAA: Replacing SCEV: {((%n * %k) + %a)<nsw>,+,%k} by: {((%n + %a),+,1}
2. EXTRA-DEBUG:: isSafeDependenceDistance
3. .....BackedgeTakenCount = (-1 + %n)
4. ….SE.getMinusSCEV(%n, (-1 + %n)) = 1
5. Total Dependences: None

SCEV assumption:
Equal predicate: %k == 1

Additional Topics - Miscellaneous SCEV

Trip Count

```
void foo(int *a, int c) {
  for (int i = 0; i < 100; i++)
    a[i] = c;
}
```

entry:

```
br label %for.body
```

For.body:

```
%i.04 = phi i32 [ 0, %entry ], [ %inc, %for.body ]
 %arrayidx = getelementptr inbounds i32, i32* %a, i32 %i.04
 store i32 %c, i32* %arrayidx, align 4, !tbaa !3
 %inc = add nuw nsw i32 %i.04, 1
 %exitcond = icmp eq i32 %inc, 100
 br i1 %exitcond, label %for.cond.cleanup, label %for.body
```

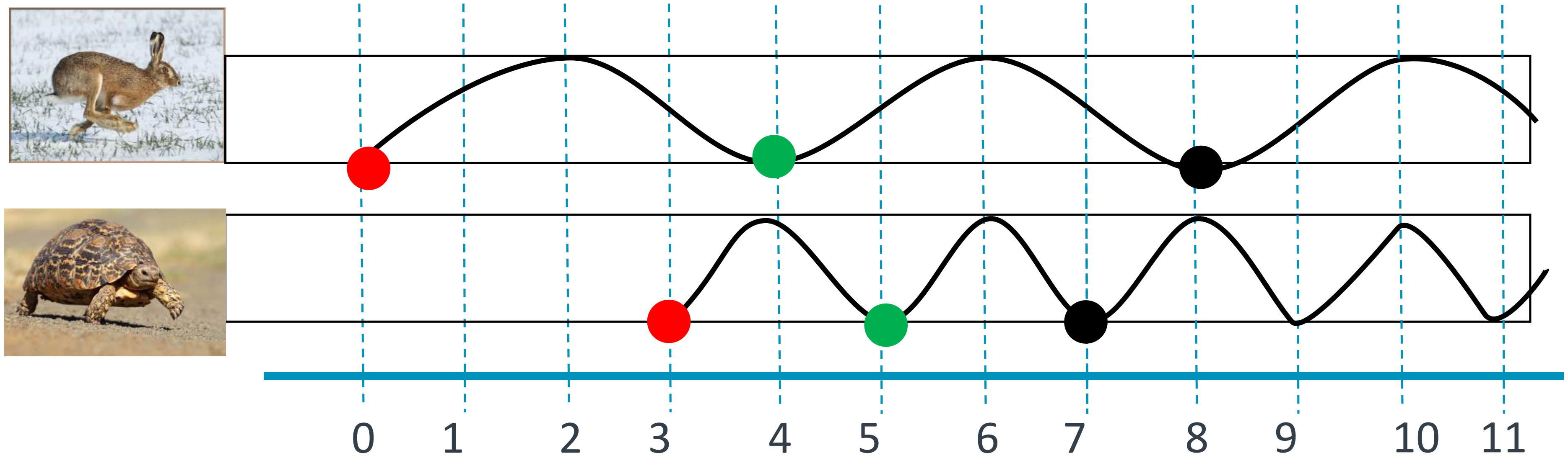
99

for.cond.cleanup:

```
ret void
```

```
$ opt -analyze -scalar-evolution foo.ll
Determining loop execution counts for: @foo
Loop %for.body: backedge-taken count is 99
```

Trip Count




```
int foo(int *a, int n, int c) {
  int hare , tortoise, step;
  for (hare = 0, tortoise = 3, step = 0; hare < tortoise ; hare+=4, tortoise+=2, step+=1);
  return step;
}
```

```
%tortoise.010 = phi i32 [ 3, %entry ], [ %add1, %for.inc ]
 -->  {3,+,2}<nuw><nsw><%for.inc>
.
Loop %for.inc: Unpredictable backedge-taken count.
```

Multiply Recurrence

```
void foo(int *a) {
  unsigned i = 0;
  for (unsigned bit = 1; bit < 0x10000; bit = 2*bit) {
    a[i] = a[i] & bit; i++;
  }
}
```

```
// SCEV(bit) → {1, *, 2}
```

```
 %i= phi i32 [ 0, %entry ], [ %inc, %for.body ]
  -->  {0,+,1}<nuw><nsw><%for.body>
.
 %inc = add nuw nsw i32 %i, 1
  -->  {1,+,1}<nuw><nsw><%for.body>

%mul = shl i32 %bit.010, 1
  -->  (2 * %bit.010)
```

Conclusion

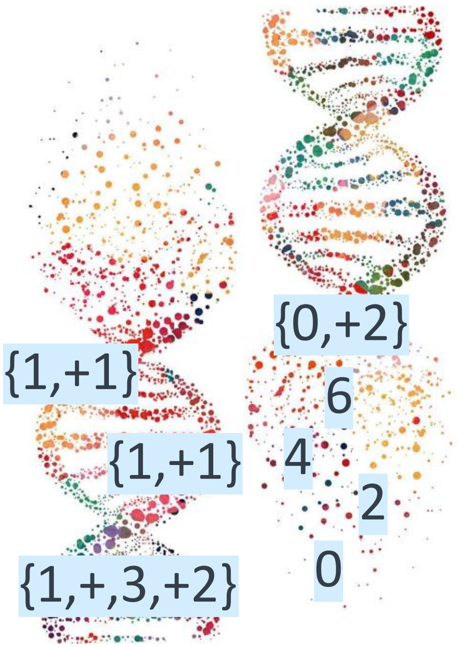
In this tutorial we learnt –

Construction of SCEV expressions

Simplification of SCEV expressions (rewriting rules)

How passes make use of LLVM SCEV as a service

Limitations of SCEV and LLVM SCEV



Scalar Evolution:
Change in the Value of
Scalar Variables
Over Iterations of the Loop

Creates and Simplifies Recurrences for ‘Expressions involving Induction Variables’

- unknown compiler engineer

Thank You!
Danke!
Merci!
谢谢!
ありがとう!
Gracias!
Kiitos!
감사합니다
धन्यवाद