ScalarEvolution and Loop Optimization

Loop Optimization in LLVM

- Relatively young
- Canonicalization
  - Make optimizations simpler
  - Make optimizations more general

Loop Canonicalizations

- Natural Loops
- Loop Rotation
- LoopSimplify form
- LCSSA form
  - “Loop-Closed SSA”
  - Identifies loop exit values

Loop Rotation

C code: `for (i = 0; i < n; ++i)`

Loop Rotation

C code: `for (i = 0; i < n; ++i)`

Typical Lowering:

```
i = 0;
while (true) {
  if (i >= n) break;
  ...
  ++i;
}
```

Loop Rotation

C code: `for (i = 0; i < n; ++i)`

Typical Lowering:

Trip Count = n + 1

```
        i = 0;
        while (true) {
Branch    if (i >= n) break;
          ...
          ++i;
Branch  }
```

Loop Rotation

C code: `for (i = 0; i < n; ++i)`

Typical Lowering:

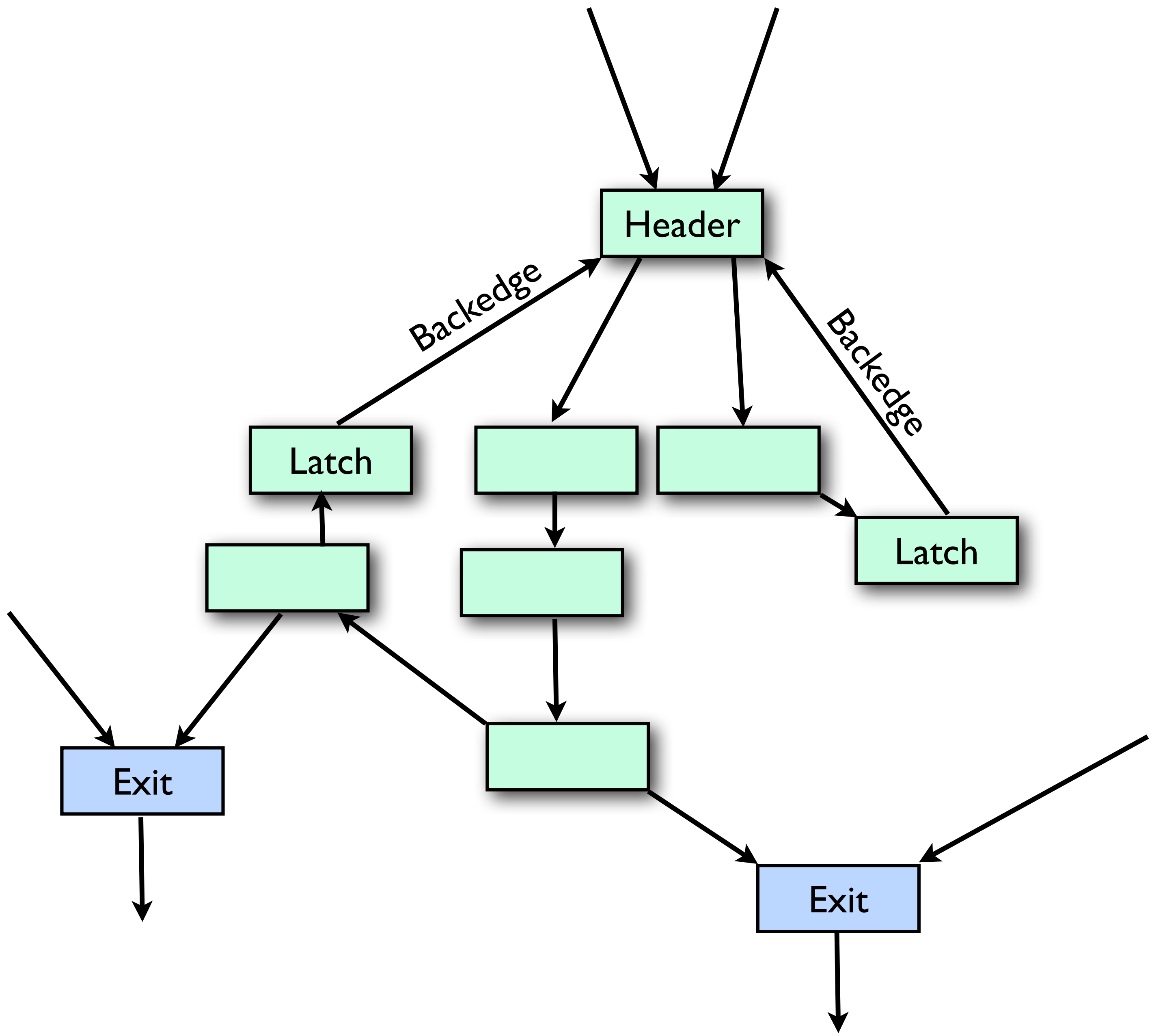
Trip Count = n + 1

```
        i = 0;
        while (true) {
Branch    if (i >= n) break;
          ...
          ++i;
Branch  }
```

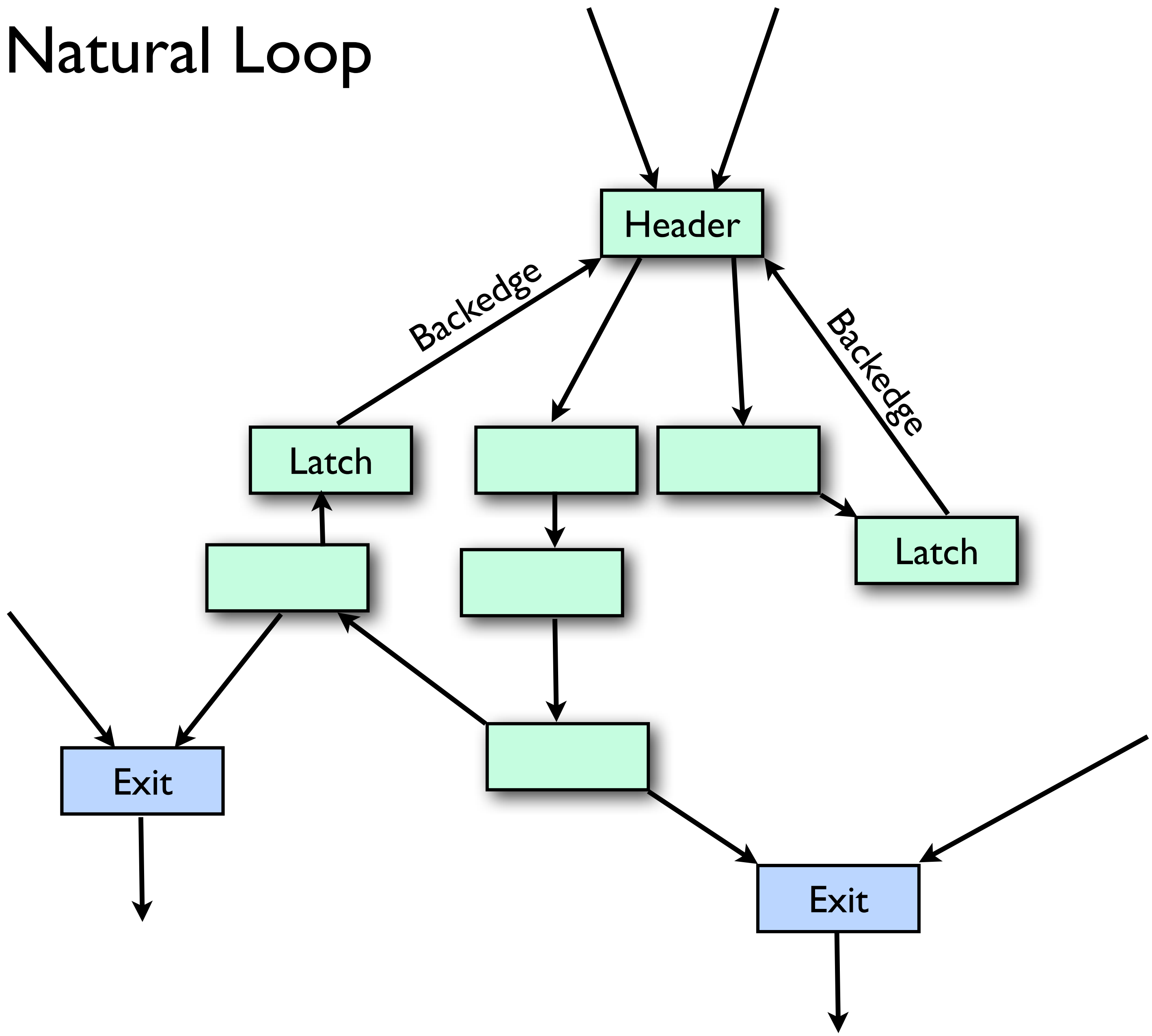
After Rotation:

Trip Count = n

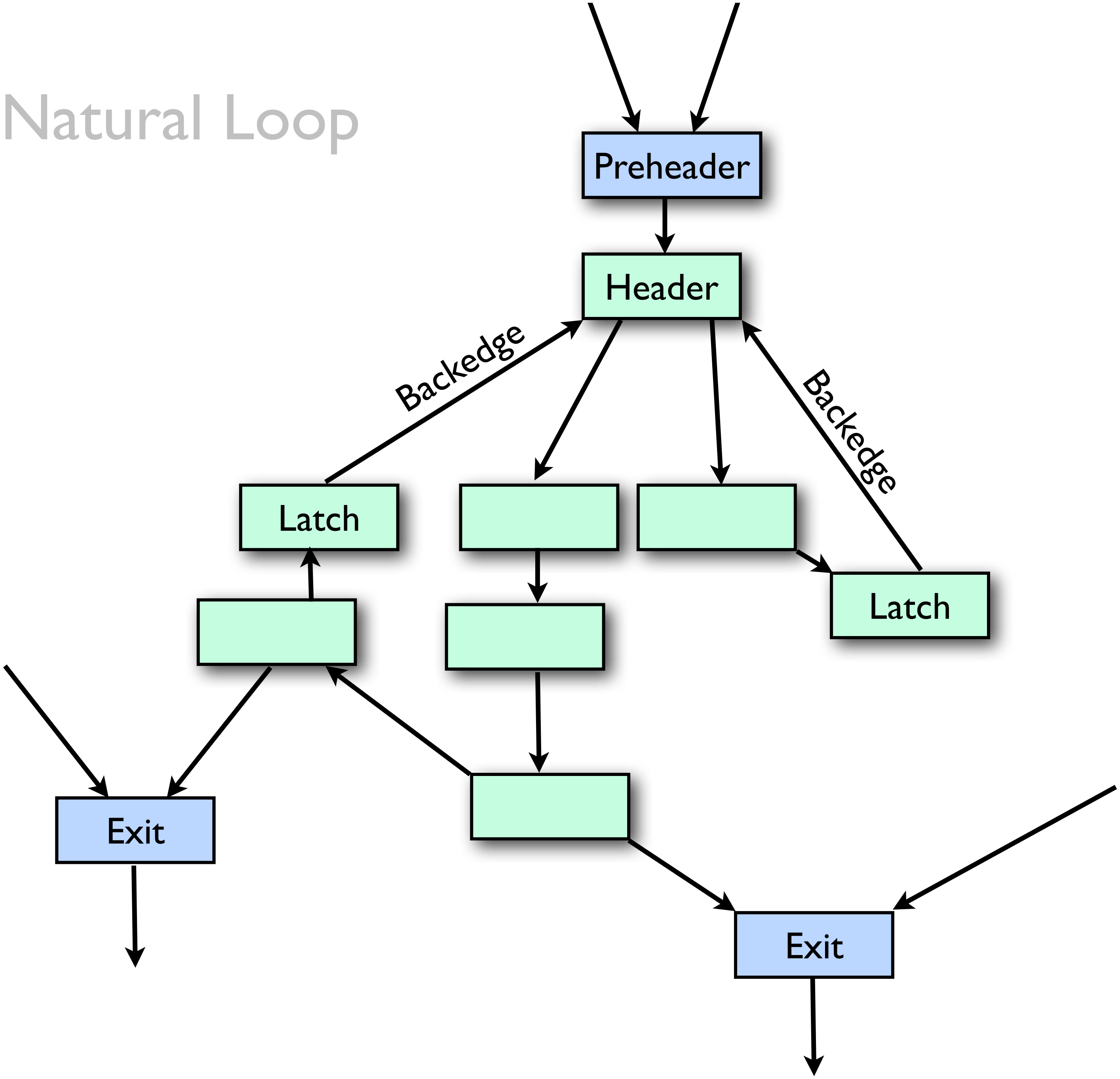
```
        i = n;
Branch  if (i > 0) {
          do {
            ...
            ++i;
Branch    } while (i < n);
        }
```


Header
Backedge
Backedge
Latch
Latch
Exit
Exit

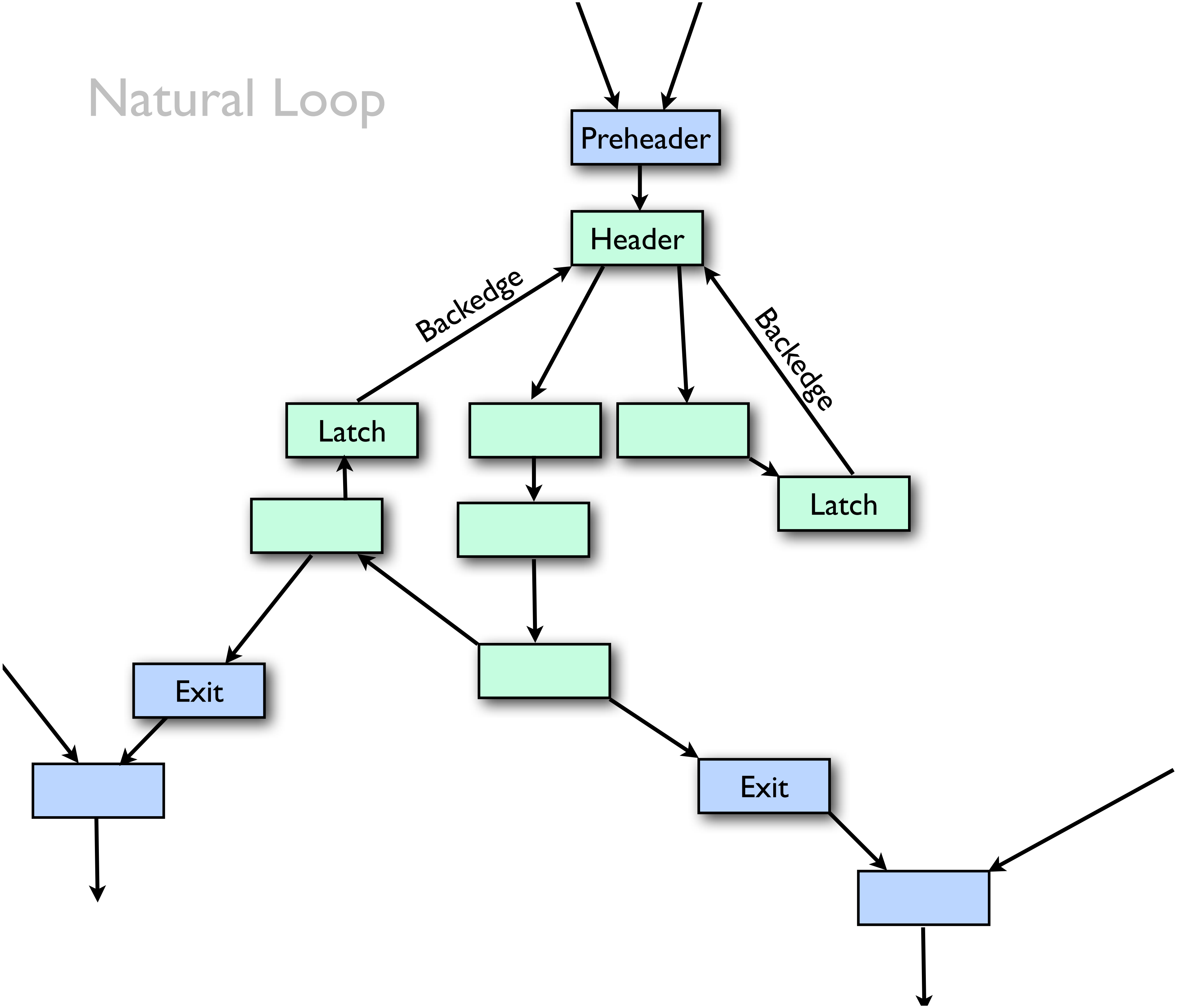
Natural Loop



Natural Loop

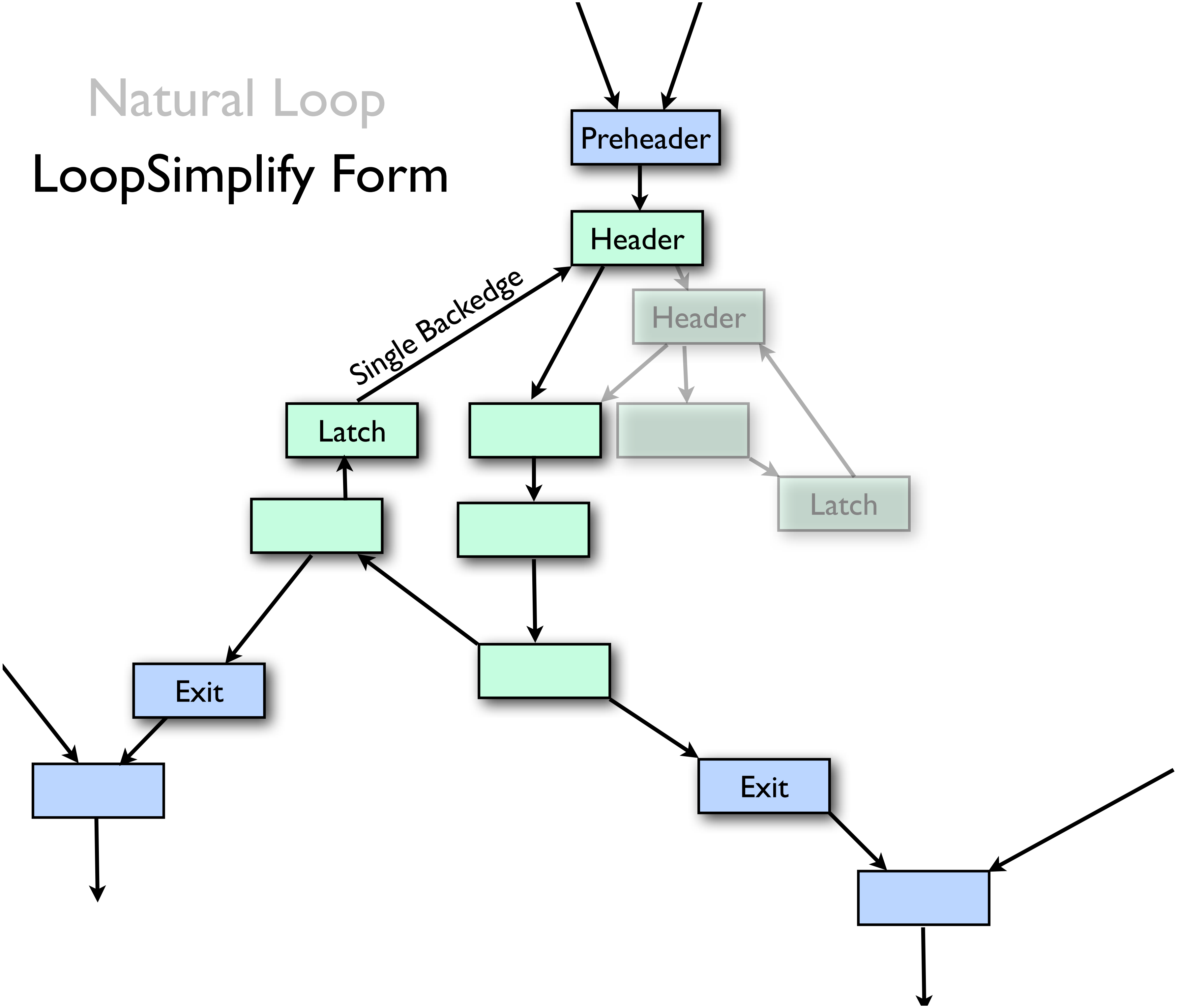


Natural Loop

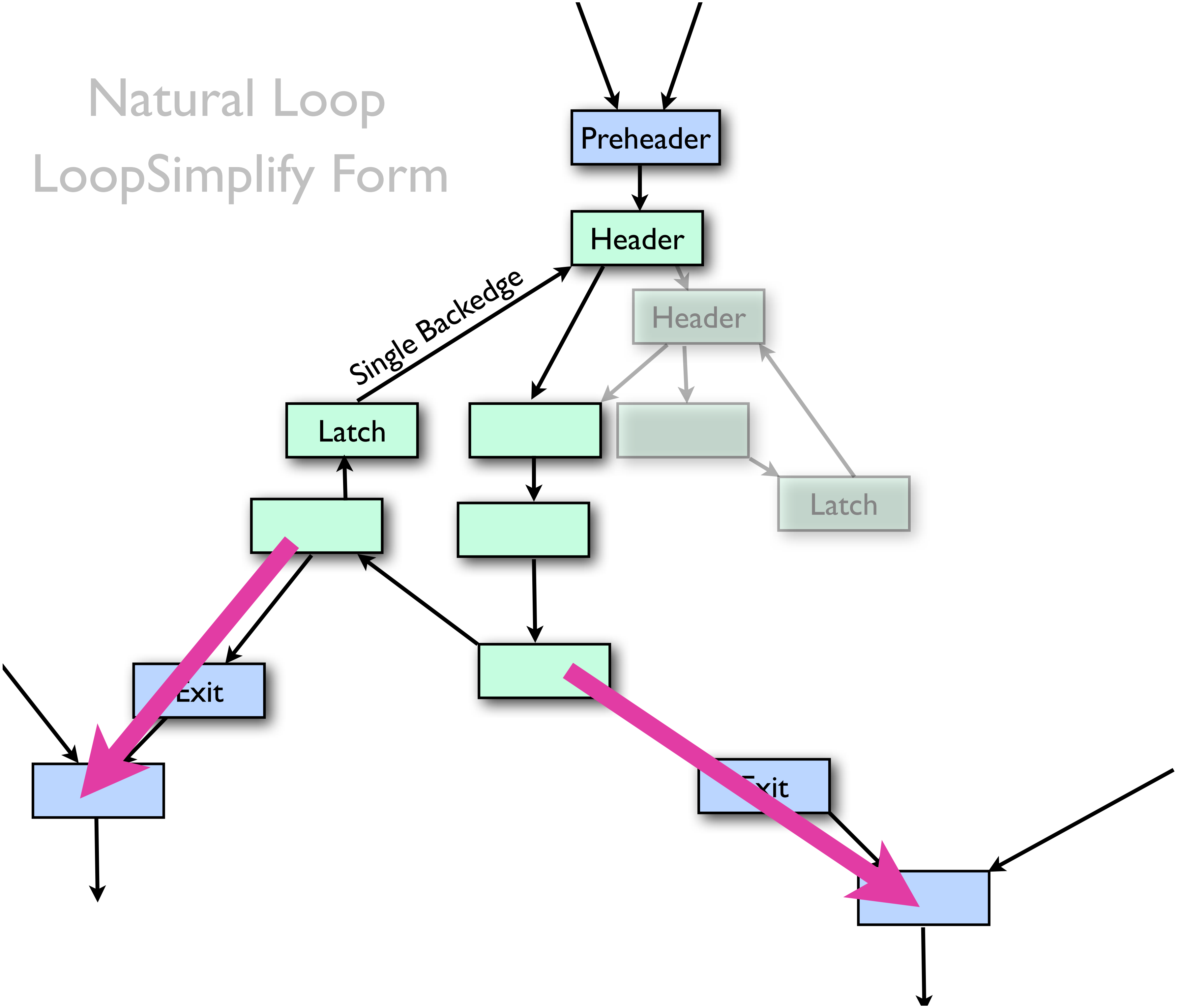


Natural Loop

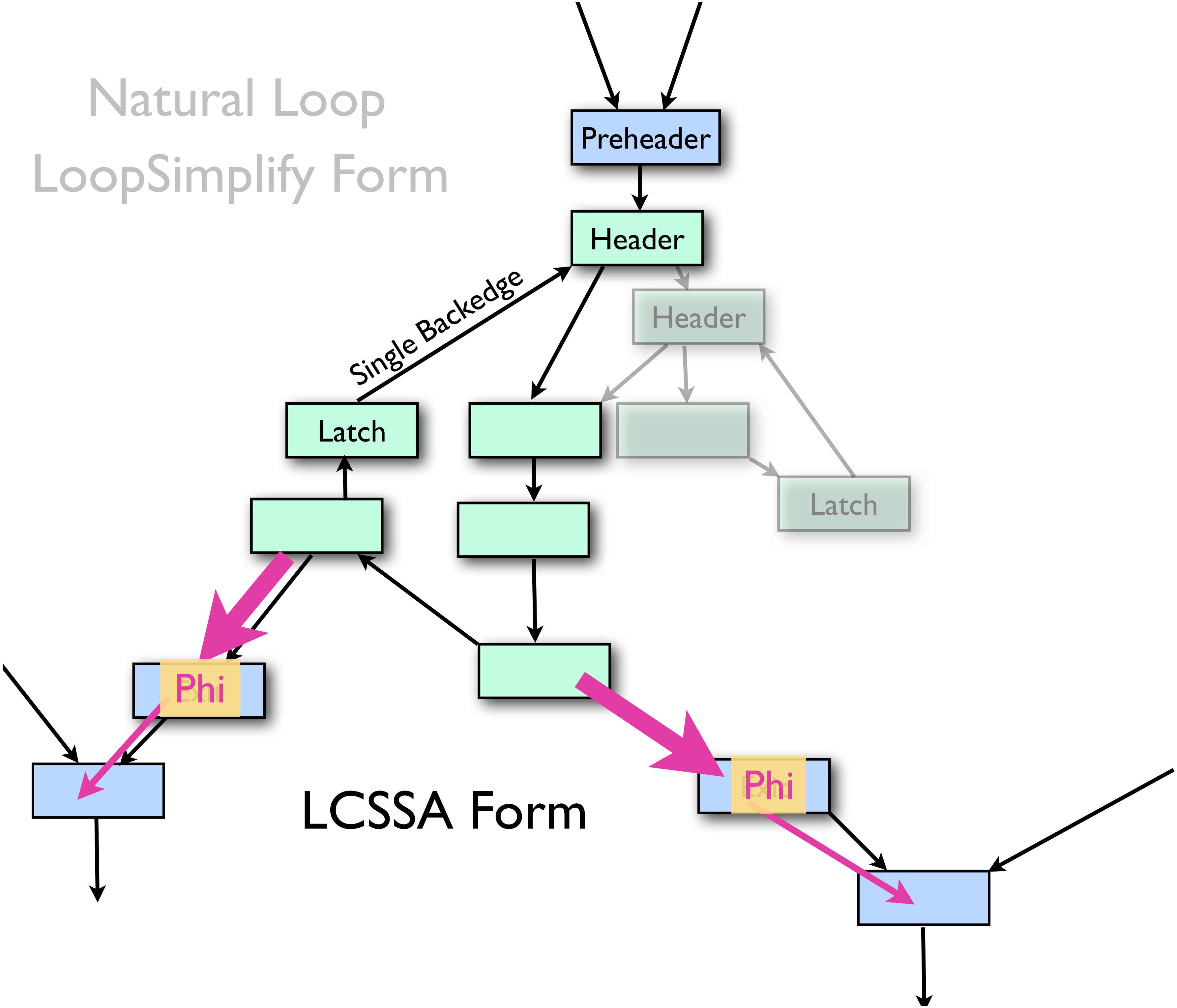
LoopSimplify Form



Natural Loop
LoopSimplify Form



Natural Loop
LoopSimplify Form



Loops in LLVM IR

```
header:
  %i = phi i64 [ 0, %preheader ],
               [ %next, %backedge ]
  ...
  %p = getelementptr @A, 0, %i
  %a = load float* %p
  ...
latch:
  %next = add i64 %i, 1
  %cmp = icmp slt %next, %N
  br i1 %cmp, label %header, label %exit
```

Loops in LLVM IR

```
header:
  %i = phi i64 [ 0, %preheader ],
               [ %next, %backedge ]
  ...
  %p = getelementptr @A, 0, %i
  %a = load float* %p   <---- ?
  ...
latch:
  %next = add i64 %i, 1
  %cmp = icmp slt %next, %N
  br i1 %cmp, label %header, label %exit
```

Loops in LLVM IR

```
header:
  %i = phi i64 [ 0, %preheader ],
               [ %next, %backedge ]
  ...
  %p = getelementptr @A, 0, %i
  %a = load float* %p
  ...
latch:
  %next = add i64 %i, 1
  %cmp = icmp slt %next, %N
  br i1 %cmp, label %header, label %exit
```

{@A,+,sizeof(float)}<%header>

Start — Stride — Loop

ScalarEvolution

- An analysis Pass
- Understand loop-oriented expressions, “scalars” whose values may evolve as loops iterate
- Map from Value to SCEV
- Loop trip-count analysis

How does LLVM use ScalarEvolution today?

- IndVarSimplify (IndVars)
  - prepare loops for advanced optimizations
  - expose trip counts
  - promote induction variables
  - rewrite exit values
- LoopStrengthReduce (LSR)
  - prepare loops for efficient execution

IndVars vs. LSR

```
for (i = 0; i < n; i += 2)
  ... = p[i];
```

IndVars vs. LSR

```
for (i = 0; i < n; i += 2)
  ... = p[i];
```

Indvars sets up a canonical induction variable:

```
for (i = 0; i != n; ++i)
  ... = p[i*2];
```

IndVars vs. LSR

```
for (i = 0; i < n; i += 2)
  ... = p[i];
```

Indvars sets up a canonical induction variable:

```
for (i = 0; i != n; ++i)
  ... = p[i*2];
```

LSR eliminates the multiplication in the loop:

```
for (i = 0; i != n; i += 2)
  ... = p[i];
```

Tripcount Analysis

- Induction Variable analysis using SSA
- “Backedge-Taken Count”
  - may be an arbitrary expression
- New tools: nsw, nuw, inbounds
  - `for (i = a; i < b; i += c)`
    - `(b-a)/c` ?
    - what if i is an “int” on a 64-bit target?

What’s a SCEV?

- “SCalar EVolution” expression
- + * / sext zext trunc smax umax
- Constant, Sizeof, Alignof
- Unknown Value
- Add Recurrences (AddRecs)

What's a SCEV?

- "SCalar EVolution" expression
- + * / sext zext trunc smax umax
- Constant, Sizeof, Alignof
- Unknown Value
- Add Recurrences (AddRecs)

A simple example

```
define void @foo(i64 %a, i64 %b, i64 %c) {
  %t0 = add i64 %b, %a
  %t1 = add i64 %t0, 7
  %t2 = add i64 %t1, %c
  ret i64 %t2
}
```

A simple example

```
define void @foo(i64 %a, i64 %b, i64 %c) {
  %t0 = add i64 %b, %a
  %t1 = add i64 %t0, 7
  %t2 = add i64 %t1, %c
  ret i64 %t2
}
```

(7 + %a + %b +%c)

```c
double *
bar(double a[10][10], long b, long c) {
  return &a[b * 3 + 7][c + 5];
}
```

```llvm
define double*
@bar([10 x double]* %a, i64 %b, i64 %c) {
  %bx3   = mul i64 %b, 3
  %bx3a7 = add i64 %bx3, 7
  %ca5   = add i64 %c, 5
  %z     = getelementptr [10 x double]* %a,
                         i64 %bx3a7,
                         i64 %ca5
  ret double* %z
}
```

```
double *
bar(double a[10][10], long b, long c) {
  return &a[b * 3 + 7][c + 5];
}
```

```
define double*
@bar([10 x double]* %a, i64 %b, i64 %c) {
  %bx3   = mul i64 %b, 3
  %bx3a7 = add i64 %bx3, 7
  %ca5   = add i64 %c, 5
  %z     = getelementptr [10 x double]* %a,
                          i64 %bx3a7,
                          i64 %ca5
  ret double* %z
}
```

(600 + (8 * %c) + (240 * %b) + %a)

Add Recurrences

```
{@A,+,sizeof(float)}<%loop>
```

- Based on Bachmann, Wang, and Zima's "Chains of Recurrences" ("chrecs")
- Lots of room for exploration

Add Recurrences

```
{@A,+,sizeof(float)}<%loop>
```

Start — Stride — Loop

- Based on Bachmann, Wang, and Zima's "Chains of Recurrences" ("chrecs")
- Lots of room for exploration

```
void foo(long n, double *p) {
  for (long i = 0; i < n; ++i)

    p[i] = 0.0;
}
```

```
void foo(long n, double *p) {
  for (long i = 0; i < n; ++i)

    p[i] = 0.0;
}
```

As a SCEV:

{%p,+,8}<%for.body>

Optionally, without TargetData:

{%p,+,sizeof(double)}<%for.body>

```
void foo(long n, long j, char *p) {
  for (long i = 0; i < n; ++i)

    p[i + j + bar()] = 0.0;

}
```

```
void foo(long n, long j, char *p) {
   for (long i = 0; i < n; ++i)

      p[i + j + bar()] = 0.0;

}
```

({(%j + %p),+,1}<%for.body> + %call)

```
void foo(long n, long j, char *p) {
  for (long i = 0; i < n; ++i)

    p[i + j + bar()] = 0.0;

}
```

({(%j + %p),+,1}<%for.body> + %call)

- AddRec operands are always loop-invariant

Nested AddRecs

```
%a = getelementptr
       [3 x [3 x double]]* %p,
       %i, %j, %k
```

```
{{{%p,+,72}<%L0>,+,24}<%L1>,+,8}<%L2>
```

Nested AddRecs

```
%a = getelementptr
     [3 x [3 x double]]* %p,
     %i, %j, %k
```

```
{{{%p,+,72}<%L0>    ,+,24}<%L1>    ,+,8}<%L2>
```

Outer Loop | Middle Loop | Inner Loop

Expression Canonicalization

- Goals:
  - Uniquify
  - Simplify
  - Put Add Recurrences on the outside
- Subtract by adding -1*x

Future uses for ScalarEvolution

- SCEV AliasAnalysis
- Loop dependence analysis
- Software prefetch insertion
- Array bounds-check elimination
- ...

Dependence Analysis on SCEVs

- What's missing before this can start?
  - shape analysis
  - given a nest of AddRecs, break out a base and indices for each array dimension
- Why?
  - GEPs are abstracted away
  - multidimensional VLAs and hand-linearized code "just work"

Non-linear recurrences

```
for (i=0; i<n; ++i)
        j += i
```

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

Non-linear recurrences

```
for (i=0; i<n; ++i)
      j += i
```

{0,+,0,+,1}<L>

=

{0,+, {0,+,1}<L> }<L>

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

Non-linear recurrences

```
for (i=0; i<n; ++i)
        j += i
```

{0,+,0,+,1}<L>

=

{0,+, {0,+,1}<L> }<L>

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

{0,+,1,+,2}<L>

=

{0,+, {1,+,2}<L> }<L>

Non-linear recurrences

```
for (i=0; i<n; ++i)
      j += i
```

```
{0,+,0,+,1}<L>
      =
{0,+, {0,+,1}<L> }<L>
```

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

```
{0,+,1,+,2}<L>
      =
{0,+, {1,+,2}<L> }<L>
```

- ScalarEvolution can solve polynomial recurrences in some cases
- There's lots more to explore here

Design questions

- ScalarEvolution is essentially a value-constraints analysis.
- Should it grow to be able to analyze floating-point values too? Vector values?
- Or should there be a separate value constraints analysis Pass instead?

CallbackVH fun

- ScalarEvolution is a FunctionPass today
- Keep the map<Value *, SCEV *> current
- Automatic notification for Value deletion.
- Automatic notification for Value modifications?
- Could it be an ImmutablePass?
- Other passes could use this too

the end.