Advanced
T⚙pics
in
C⚙mpilers

Loop

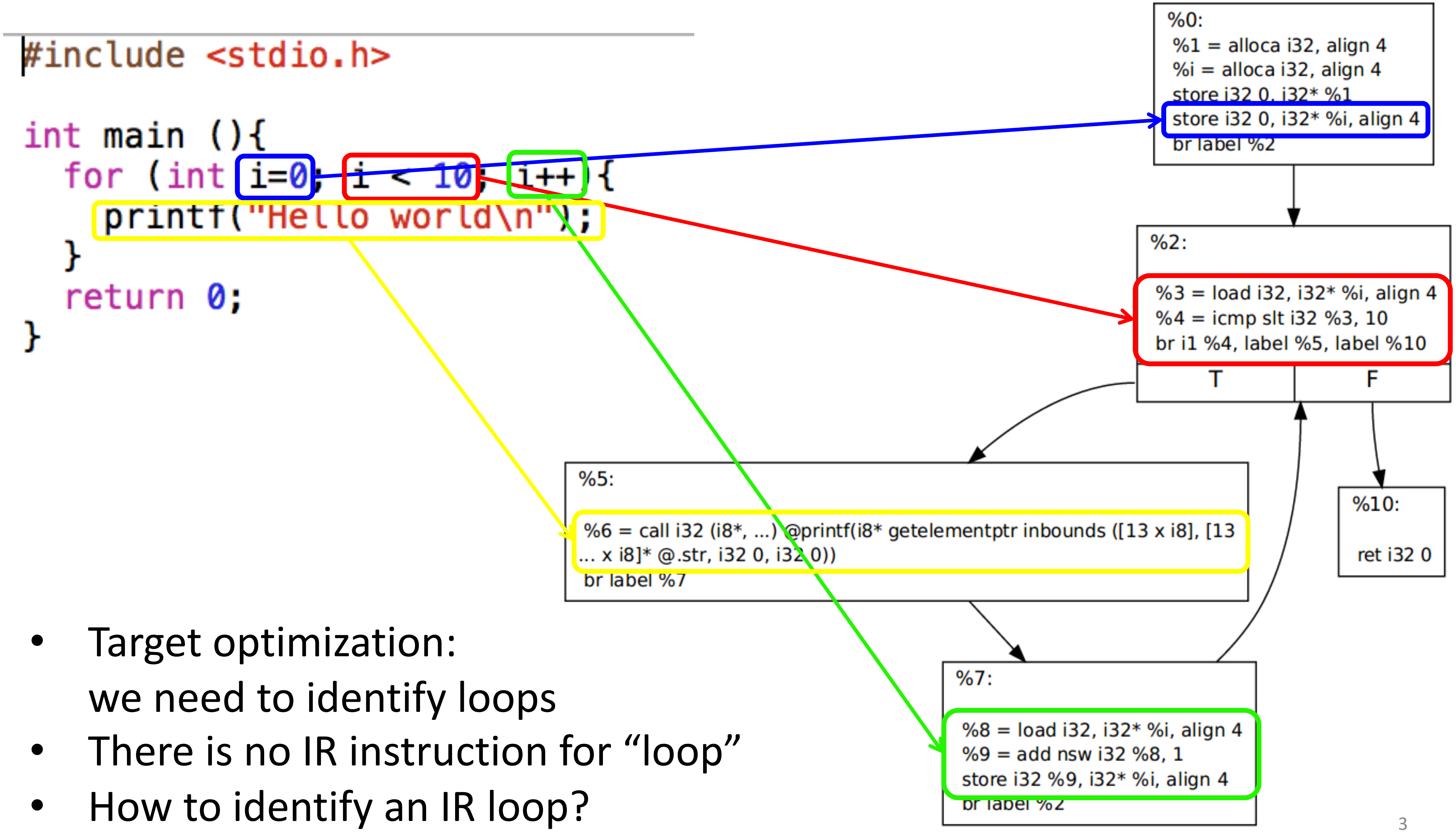
Simone Campanoni
simone.campanoni@northwestern.edu

Outline

- Loops in LLVM (from C⚙de analysis *and* transf⚙rmation)

- A loop in NOELLE

- Abstractions for a single loop in NOELLE



```c
#include <stdio.h>

int main (){
  for (int i=0; i < 10; i++){
    printf("Hello world\n");
  }
  return 0;
}
```

```
%0:
 %1 = alloca i32, align 4
 %i = alloca i32, align 4
 store i32 0, i32* %1
 store i32 0, i32* %i, align 4
 br label %2
```

```
%2:
 %3 = load i32, i32* %i, align 4
 %4 = icmp slt i32 %3, 10
 br i1 %4, label %5, label %10
```

```
%5:
 %6 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([13 x i8], [13
... x i8]* @.str, i32 0, i32 0))
 br label %7
```

```
%7:
 %8 = load i32, i32* %i, align 4
 %9 = add nsw i32 %8, 1
 store i32 %9, i32* %i, align 4
 br label %2
```

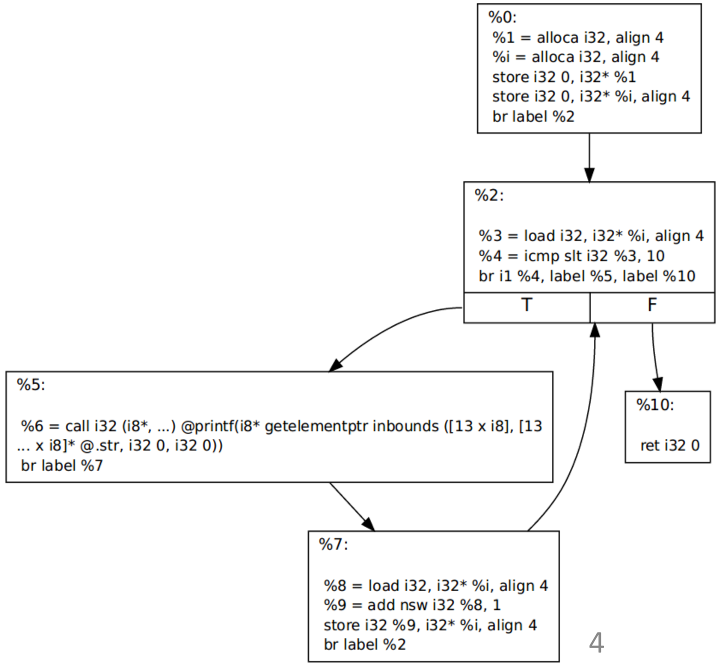
```
%10:
 ret i32 0
```

CFG for 'main' function

- Target optimization:
  we need to identify loops
- There is no IR instruction for “loop”
- How to identify an IR loop?

Loops in IR

- Loop identification control flow analysis:
  - Input: Control-Flow-Graph
  - Output: loops in CFG
  - Not sensitive to input syntax: a uniform treatment for all loops
- Define a loop in graph terms (natural loop)
- Properties of a natural loop
  - Single entry point
  - Edges must form at least a cycle in CFG



```
%0:
 %1 = alloca i32, align 4
 %i = alloca i32, align 4
 store i32 0, i32* %1
 store i32 0, i32* %i, align 4
 br label %2
```

```
%2:

 %3 = load i32, i32* %i, align 4
 %4 = icmp slt i32 %3, 10
 br i1 %4, label %5, label %10
```

```
%5:

 %6 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([13 x i8], [13
... x i8]* @.str, i32 0, i32 0))
 br label %7
```

```
%10:

 ret i32 0
```

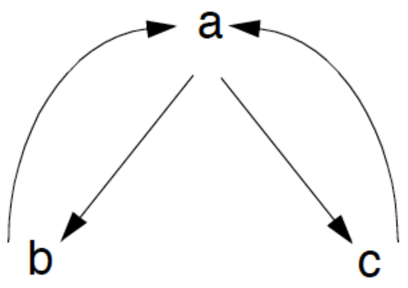
```
%7:

 %8 = load i32, i32* %i, align 4
 %9 = add nsw i32 %8, 1
 store i32 %9, i32* %i, align 4
 br label %2
```

CFG for 'main' function

Identify inner loops

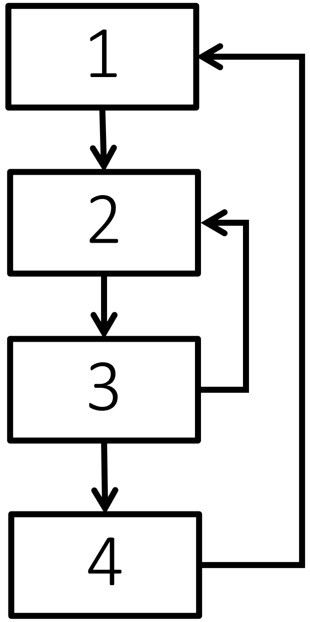
- If two natural loops do not have the same header
  - They are either disjoint, or
  - One is entirely contained (nested within) the other
    - Outer loop, inner loop
    - Loop nesting relation: loop nesting tree
- What about if two loops share the same header?

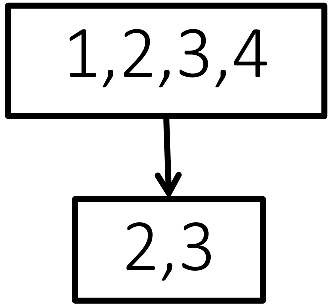
```
while (a: i < 10){
    b: if (i == 5) continue;
    c: ...
}
```



Loop nesting tree

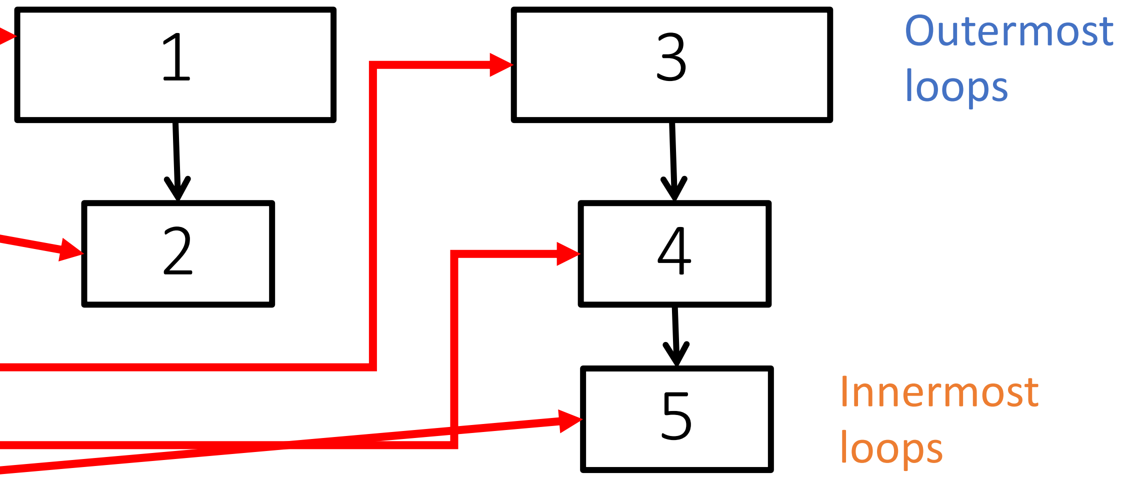
- **Loop-nest tree**: each node represents the blocks of a loop, and parent nodes are enclosing loops.
- The leaves of the tree are the inner-most loops.



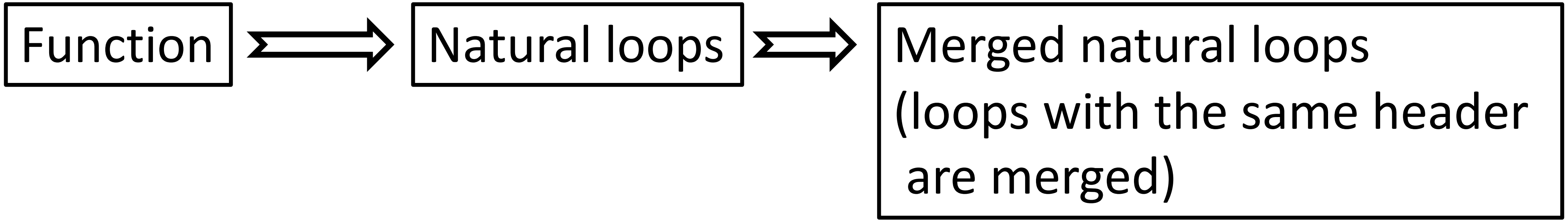


Loop nesting forest

```
void myFunction (){
1: while (…){
2:    while (…){ … }
    }
    …
3: for (…){
4:    do {
5:        while(…) {…}
        } while (…)
    }
}
```



Loops in LLVM



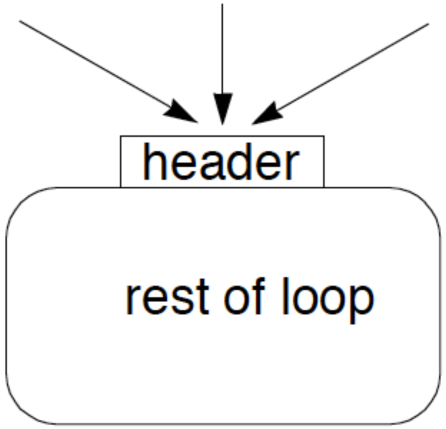
Function

Natural loops

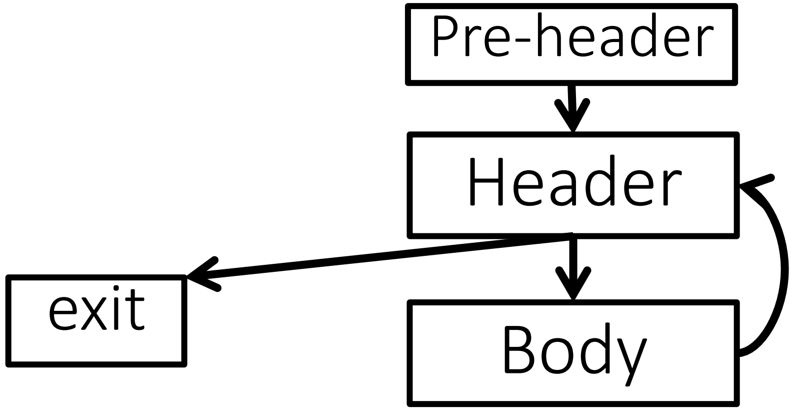
Merged natural loops
(loops with the same header
are merged)

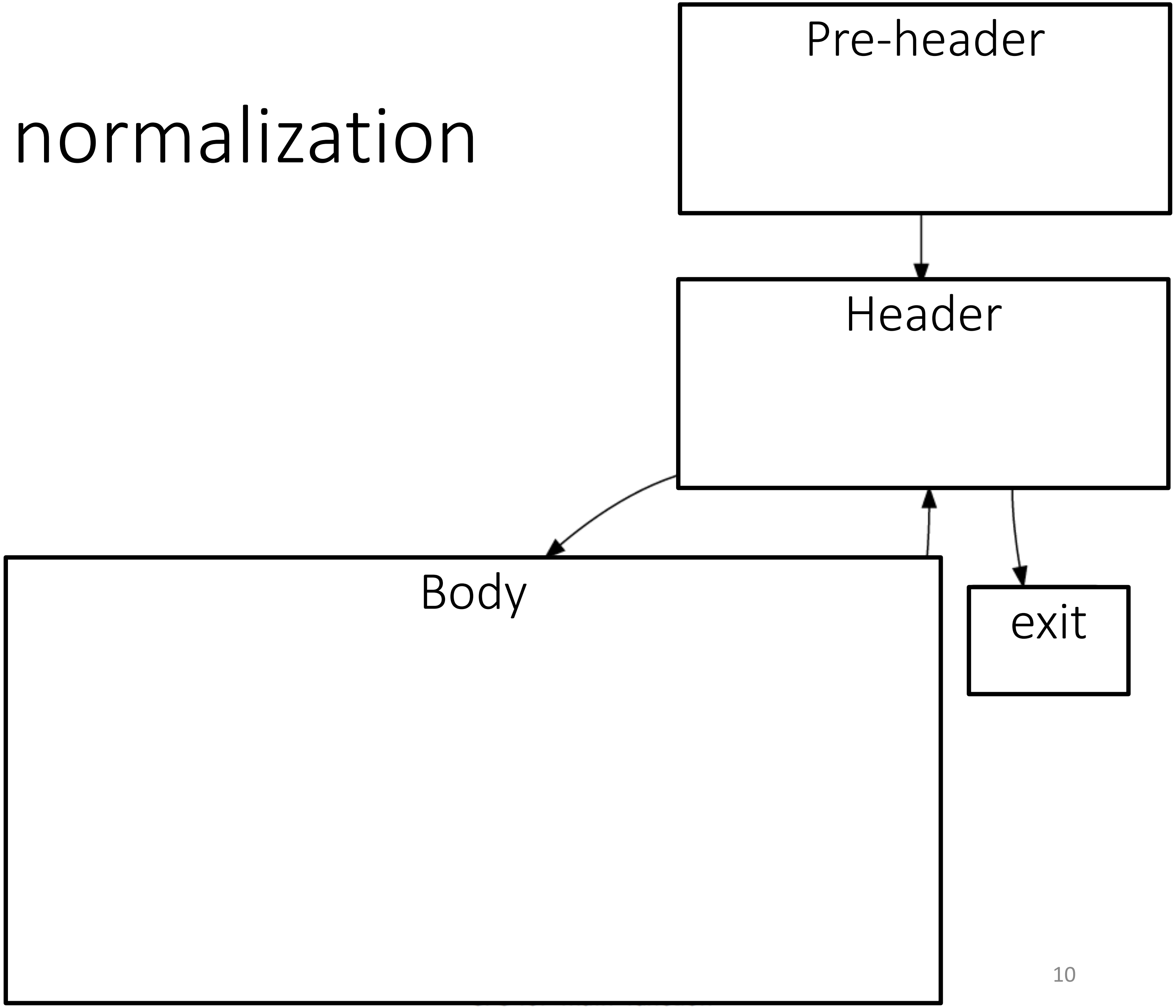
First loop normalization: adding a pre-header

- Optimizations often require code to be executed once before the loop
- Create a pre-header basic block for every loop



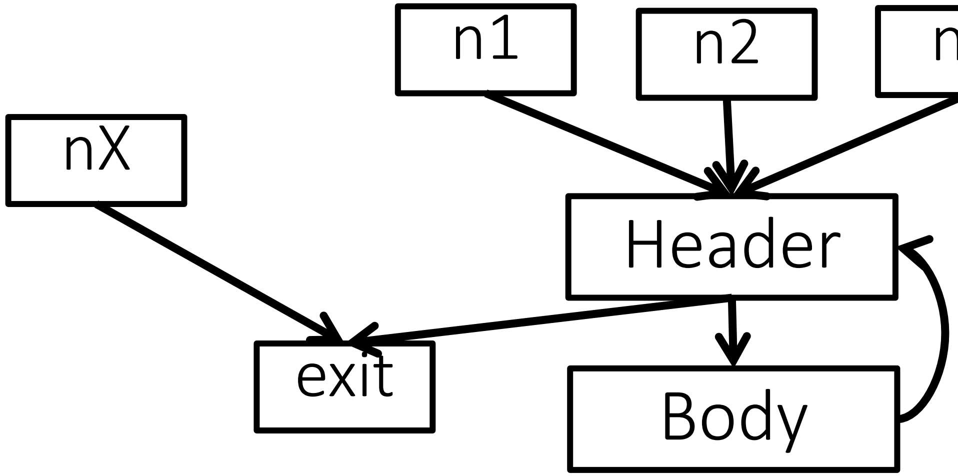
Common loop normalization





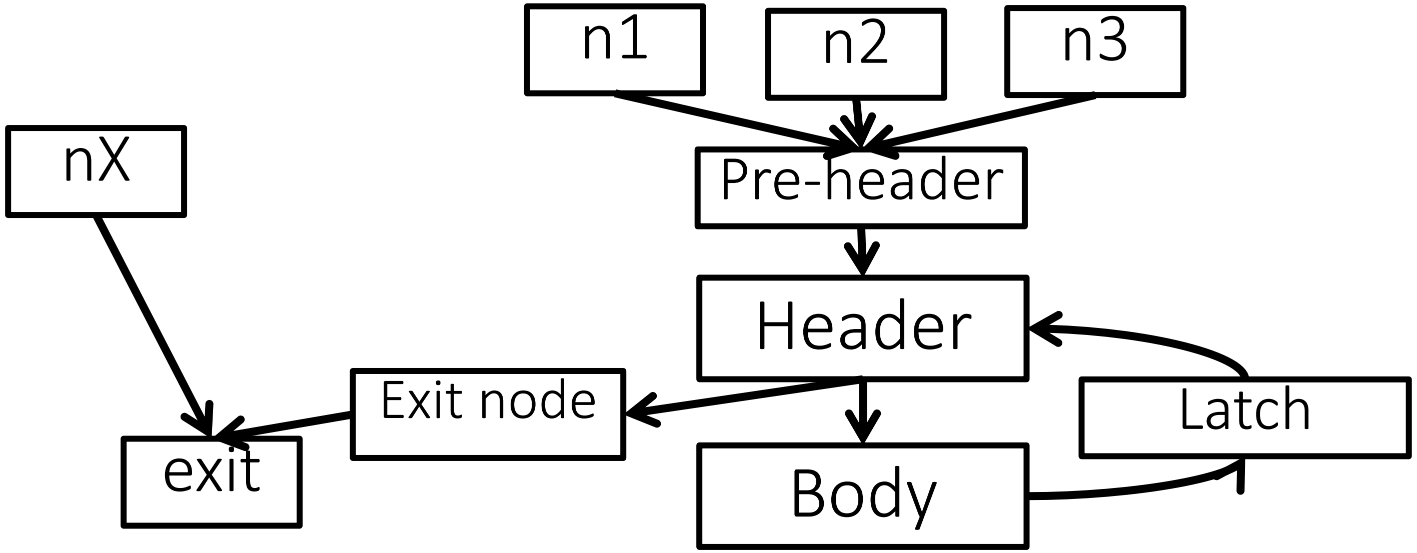
Loop normalization in LLVM

- The loop-simplify pass normalize natural loops
- Output of loop-simplify:



Loop normalization in LLVM

- The loop-simplify pass normalize natural loops
- Output of loop-simplify:
  - **Pre-header**: the only predecessor of the header
  - **Latch**: single node executed just before starting a new loop iteration
  - **Exit node**: ensures it is dominated by the header

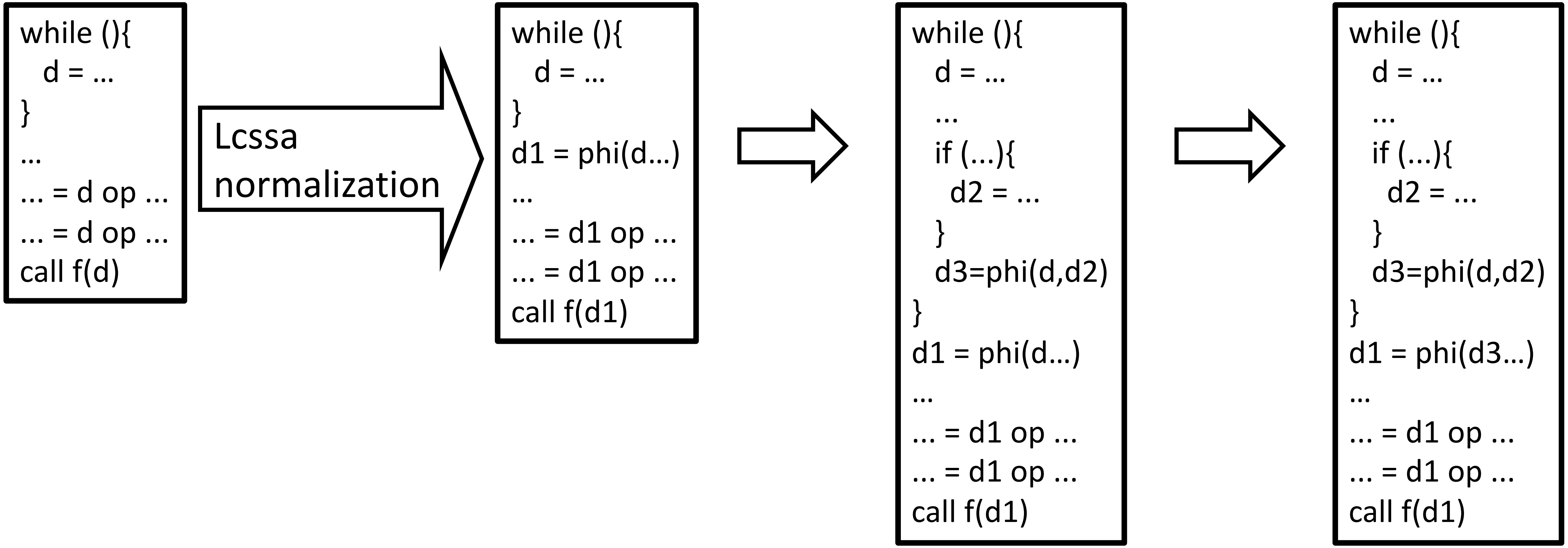


Further normalizations in LLVM

- Loop representation can be further normalized:
  - *loop-simplify* normalize the shape of the loop
  - What about definitions in a loop?

- Problem: updating code in loop might require to update code outside loops for keeping SSA
  - Loop-closed SSA form: no var defined in loop is used outside of that loop
  - lcssa insert phi instruction at loop boundaries
    for variables defined in the body of a loop and used outside that loop

Loop pass example



```
while (){
   d = …
}
…
… = d op …
… = d op …
call f(d)
```

Lcssa normalization

```
while (){
   d = …
}
d1 = phi(d…)
…
… = d1 op …
… = d1 op …
call f(d1)
```

```
while (){
   d = …
   …
   if (…){
     d2 = …
   }
   d3=phi(d,d2)
}
d1 = phi(d…)
…
… = d1 op …
… = d1 op …
call f(d1)
```

```
while (){
   d = …
   …
   if (…){
     d2 = …
   }
   d3=phi(d,d2)
}
d1 = phi(d3…)
…
… = d1 op …
… = d1 op …
call f(d1)
```

Loop-closed SSA-form

Outline

- Loops in LLVM (from Code analysis *and* transformation)
- A loop in NOELLE
- Abstractions for a single loop in NOELLE

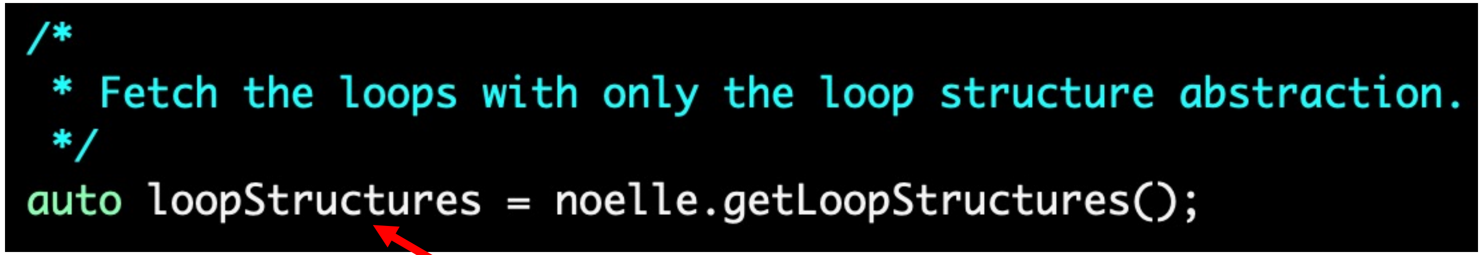
NOELLE

- All loops in NOELLE are normalized as canonical and in LCSSA form at all time

- Before invoking NOELLE to any IR file, you must normalize that IR
  - noelle-norm:
    normalizations required by NOELLE
  - noelle-simplification:
    normalizations required by NOELLE +
    fast optimizations that are needed most of the time
    (e.g., dead code elimination)

Introduction to NOELLE compilation/optimization...

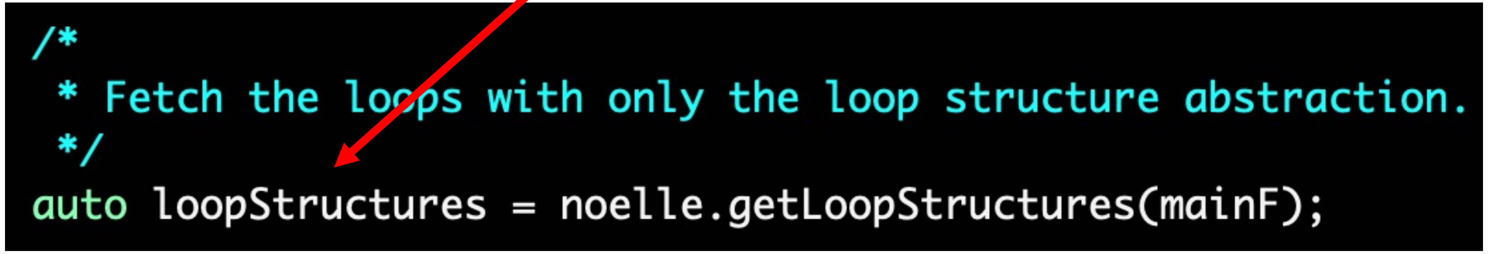
Get all program loops with NOELLE



```
/*
 * Fetch the loops with only the loop structure abstraction.
 */
auto loopStructures = noelle.getLoopStructures();
```

Container of objects (one per loop) that describe loops.

Each one is an instance of arcana::noelle::LoopStructure



```
/*
 * Fetch the loops with only the loop structure abstraction.
 */
auto loopStructures = noelle.getLoopStructures(mainF);
```

Freeing memory

- As for all other abstractions NOELLE provides,
  it is the caller of the NOELLE's API that generates LoopStructure
  that is responsible to free their memory
  whenever they are no longer needed

- To free memory of an instance myLoop of LoopStructure
  (or any other abstraction provided by NOELLE): delete myLoop

- NOELLE provides no support to check (and update)
  the validity of LoopStructure after changing the IR
  (since the creation of LoopStructure)

Re-computing LoopStructure

Imagine the following situation:

1. You asked NOELLE to create LoopStructure and
2. You modified the IR after having computed LoopStructure and
3. You still need to invoke the API of LoopStructure and
4. You don’t know whether LoopStructure is valid or not, then

recompute LoopStructure (e.g., with noelle-fixedpoint)

Outline

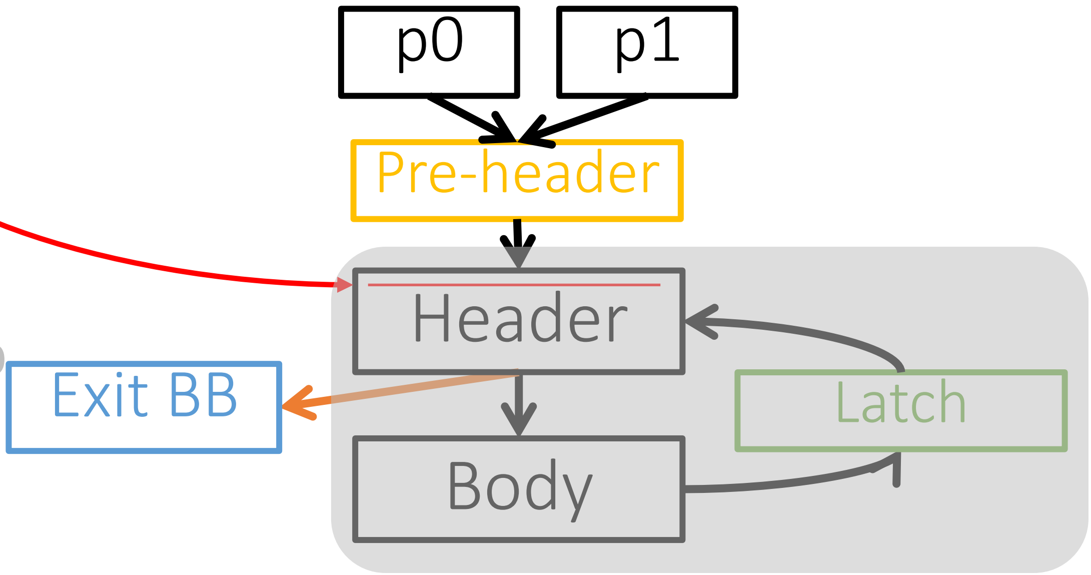
- Loops in LLVM (from Code analysis *and* transformation)

- A loop in NOELLE

- Abstractions for a single loop in NOELLE

Loop abstractions in NOELLE

- We saw one abstraction so far: LoopStructure
- LoopStructure describes structural aspects of a loop
  - Entry instruction
  - Exit basic blocks, exit edges
  - Latches
  - Pre-header
  - Successor of the Header within the loop
  - Set of basic blocks that compose the loop
  - Nesting level
  - An ID
- LoopStructure is a little more than LLVM's Loop



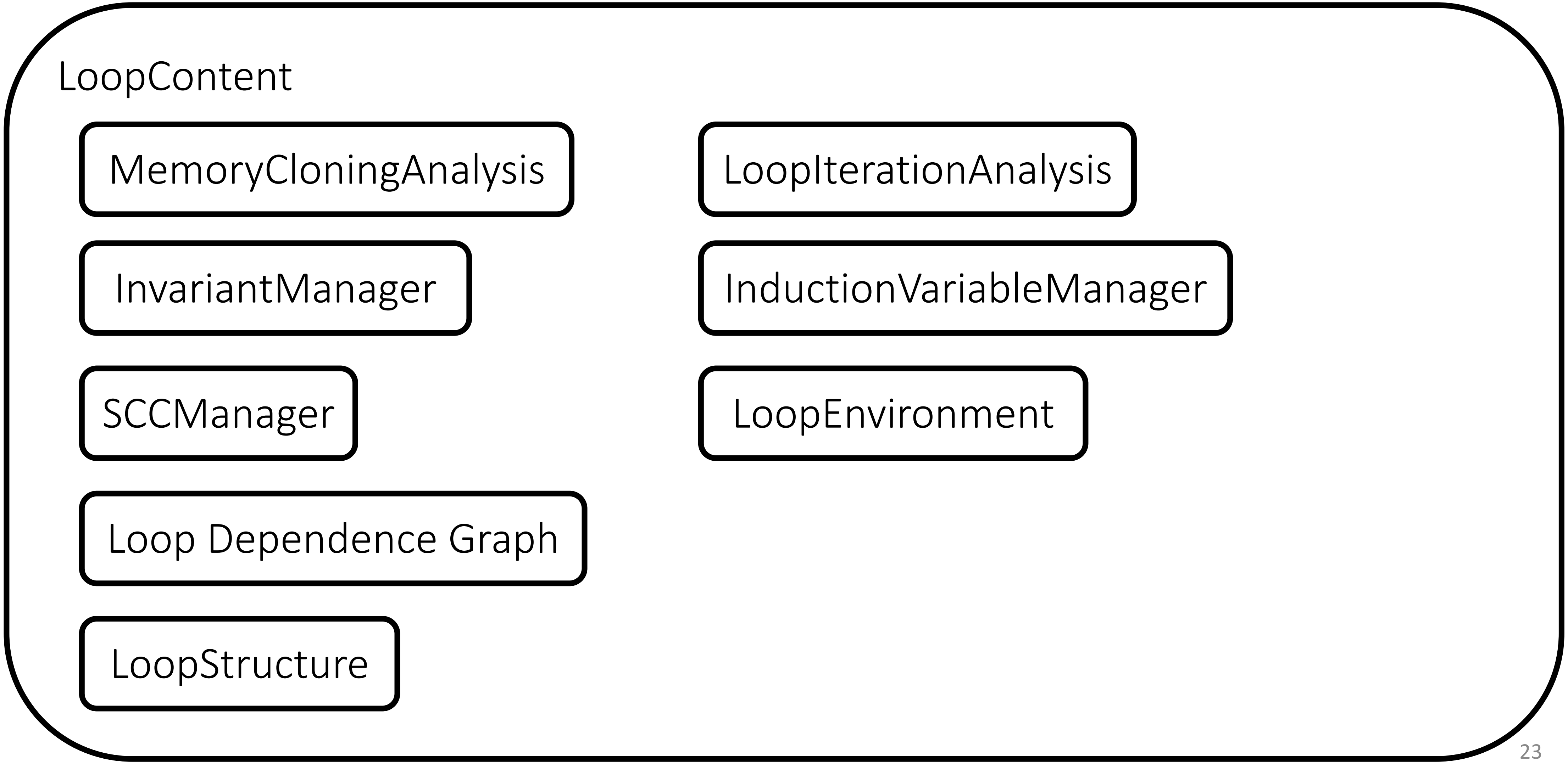
Loop abstractions in NOELLE

When you study an important loop (e.g., a hot loop),
we often need more information about it going beyond
its structure. For example:

- What are the induction variables of a loop?
- What are the invariants of a loop?
- What is the dependence graph of this loop?
  (i.e., loop dependence graph --- LDG)
- What is the SCCDAG of the loop dependence graph of this loop?

To capture all information of a loop: arcana::noelle::LoopContent

Loop abstractions in NOELLE



LoopContent

MemoryCloningAnalysis

InvariantManager

SCCManager

Loop Dependence Graph

LoopStructure

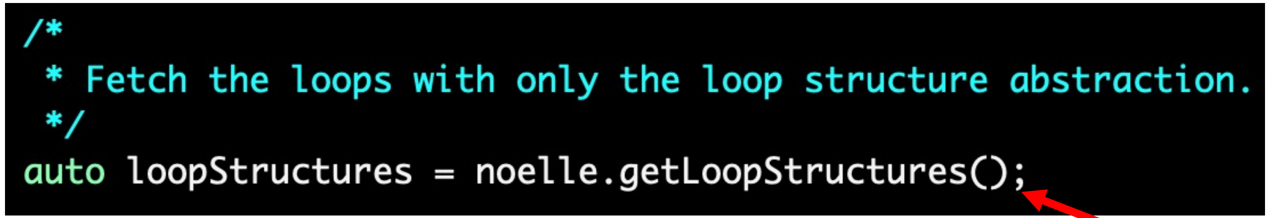
LoopIterationAnalysis

InductionVariableManager

LoopEnvironment

LoopContent

- In NOELLE:
  LoopStructure is the simplest abstraction that describes a loop

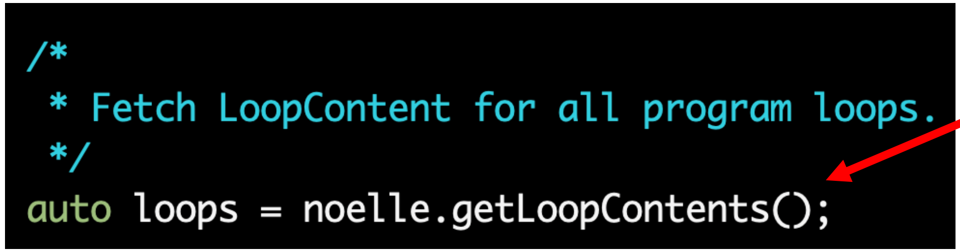


```
/*
 * Fetch the loops with only the loop structure abstraction.
 */
auto loopStructures = noelle.getLoopStructures();
```

You should get all loop structures of a program (relatively low complexity) and only fetch LoopContent for loops you decide to target

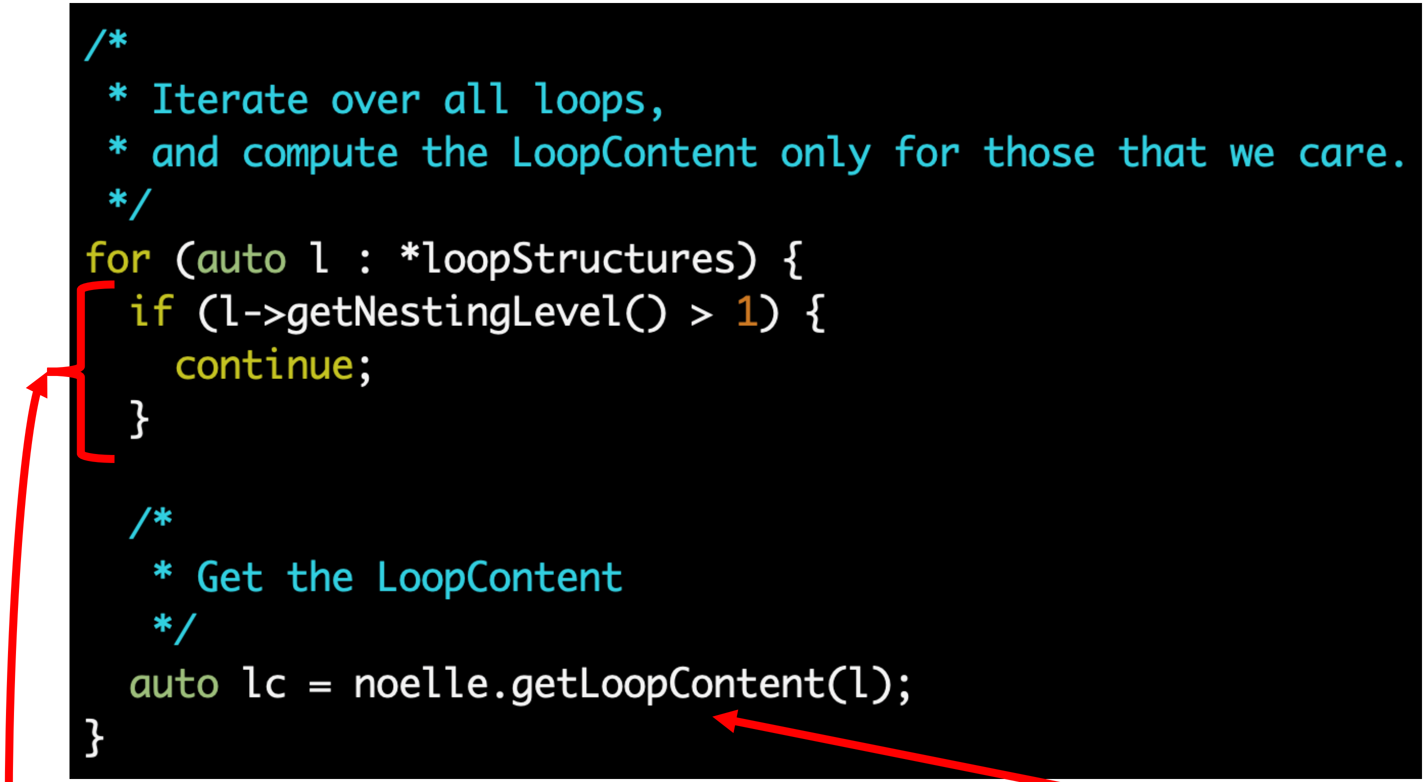
- In NOELLE:
  LoopContent is the abstraction that describes a loop
  with the highest amount of information available in NOELLE

Significantly more expensive than



```
/*
 * Fetch LoopContent for all program loops.
 */
auto loops = noelle.getLoopContents();
```

From LoopStructure to LoopContent

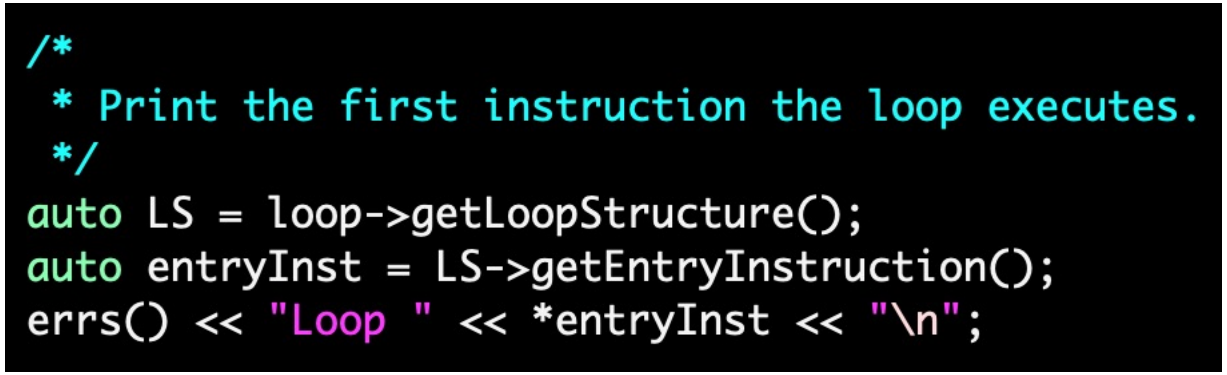


```
/*
 * Iterate over all loops,
 * and compute the LoopContent only for those that we care.
 */
for (auto l : *loopStructures) {
  if (l->getNestingLevel() > 1) {
    continue;
  }

  /*
   * Get the LoopContent
   */
  auto lc = noelle.getLoopContent(l);
}
```

Whatever filter you want to implement to skip loops you don’t care

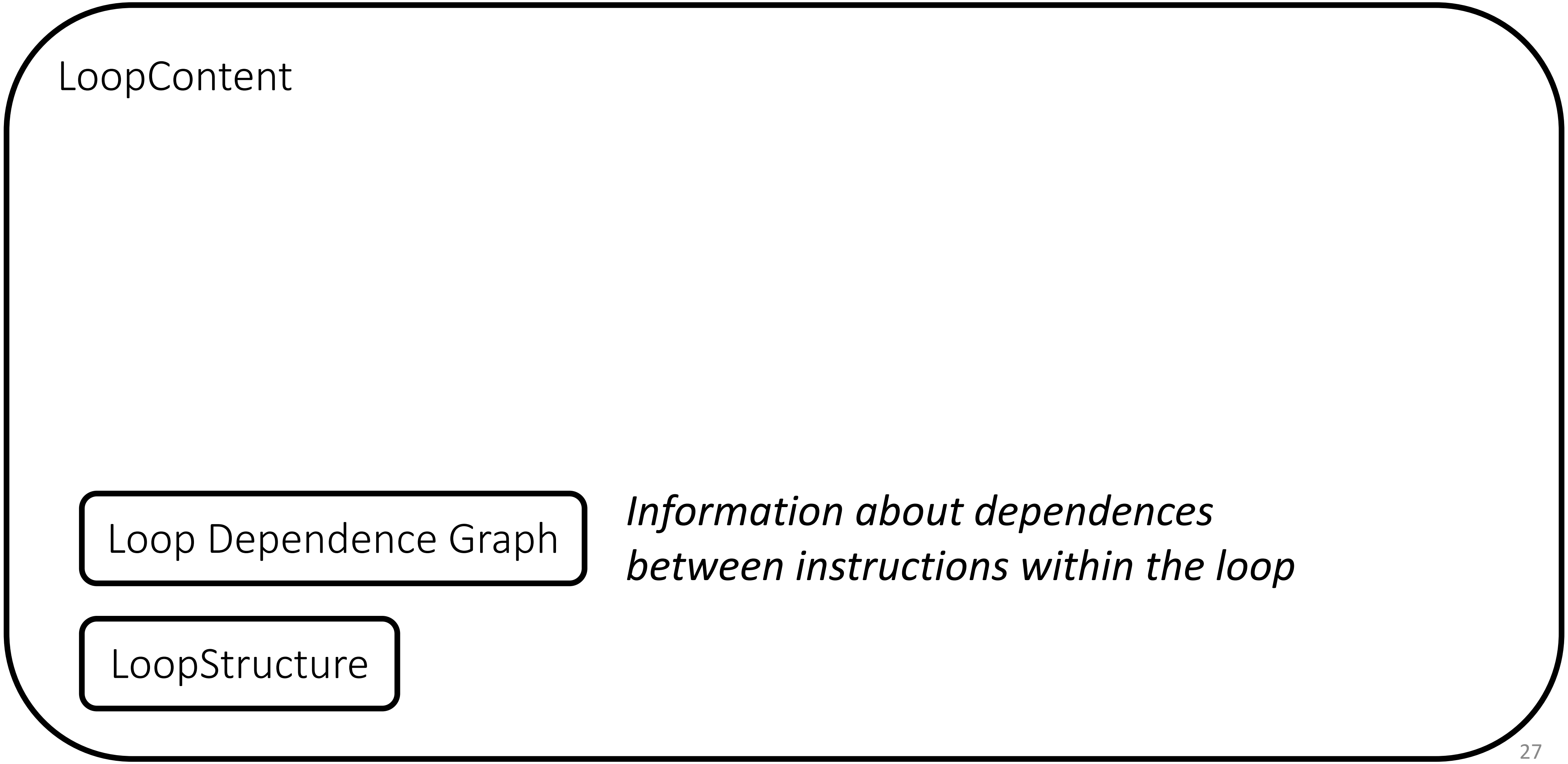
It creates a new LoopStructure to include in lc

From LoopContent to LoopStructure



```
/*
 * Print the first instruction the loop executes.
 */
auto LS = loop->getLoopStructure();
auto entryInst = LS->getEntryInstruction();
errs() << "Loop " << *entryInst << "\n";
```

Abstractions related to loops in NOELLE



LoopContent

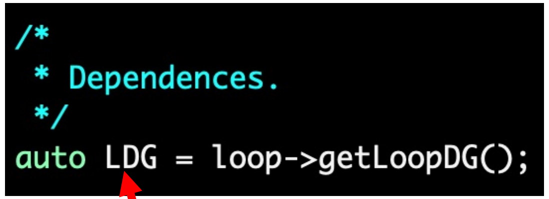
Loop Dependence Graph

Information about dependences between instructions within the loop

LoopStructure

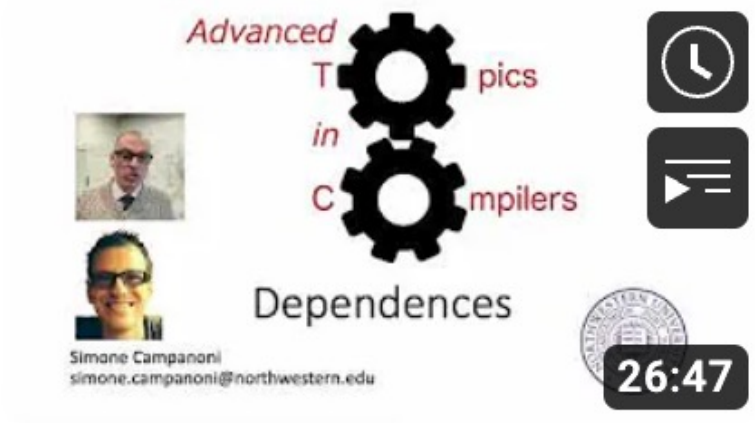
From LoopContent
to Loop Dependence Graph

- Loop dependence Graph



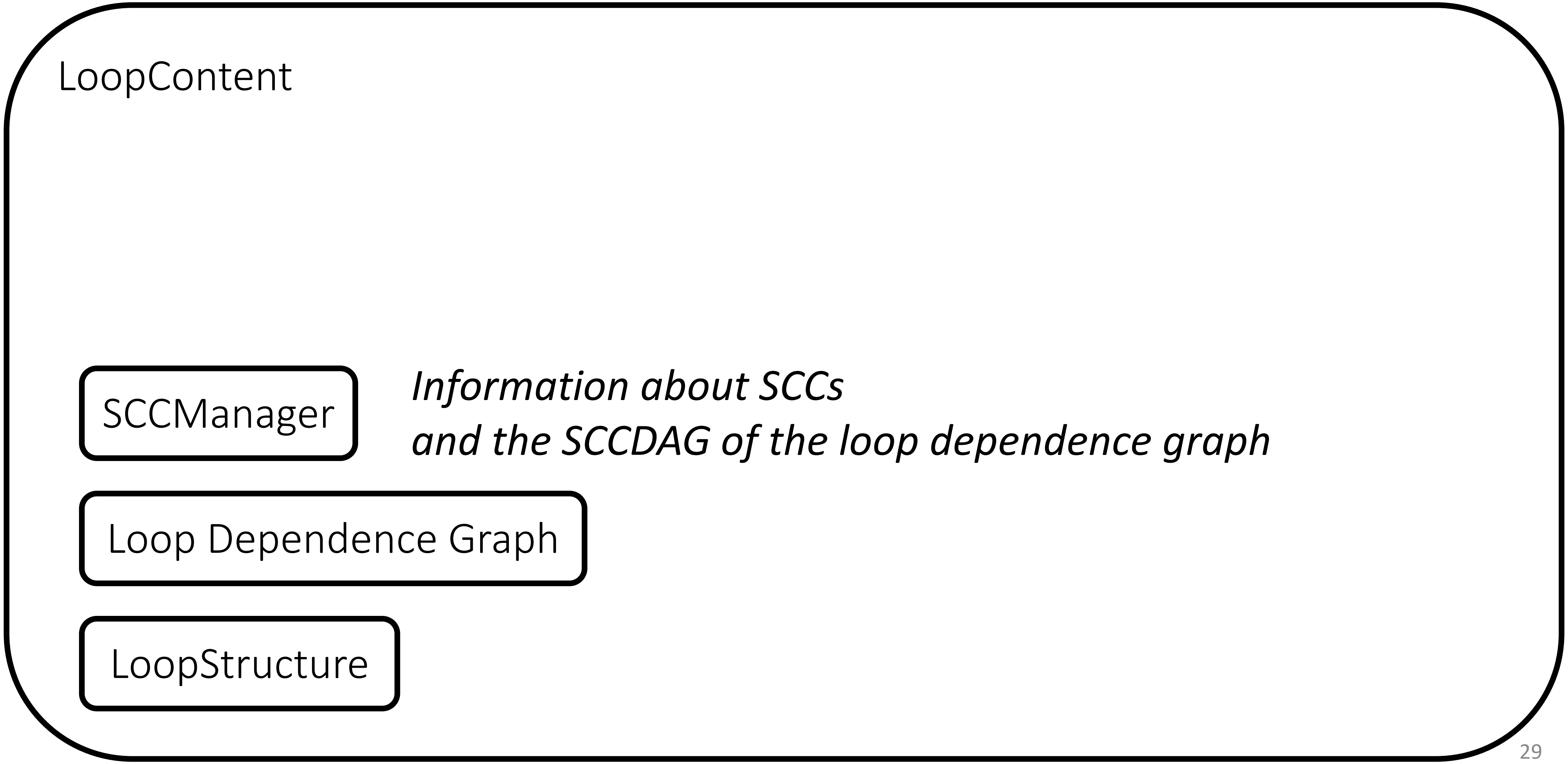
```
/*
 * Dependences.
 */
auto LDG = loop->getLoopDG();
```

Instance of the class arcana::noelle::PDG


Advanced Topics in Compilers
Dependences
Simone Campanoni
simone.campanoni@northwestern.edu
26:47

Dependences with NOELLE

Abstractions related to loops in NOELLE



LoopContent

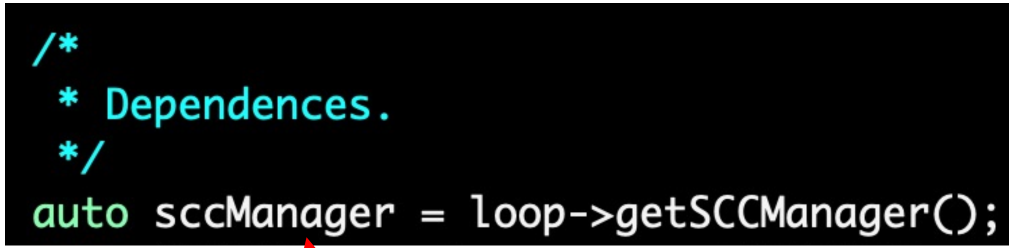
SCCManager

Information about SCCs
and the SCCDAG of the loop dependence graph

Loop Dependence Graph

LoopStructure

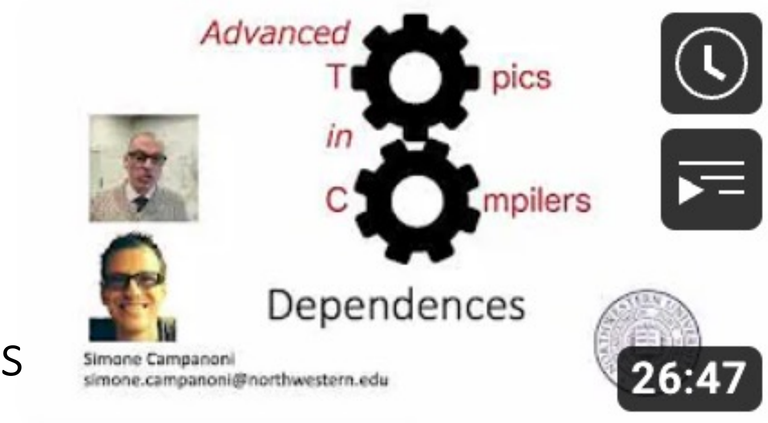
From LoopContent to SCCManager



```
/*
 * Dependences.
 */
auto sccManager = loop->getSCCManager();
```

Instance of the class arcana::noelle::SCCDAGAttrs

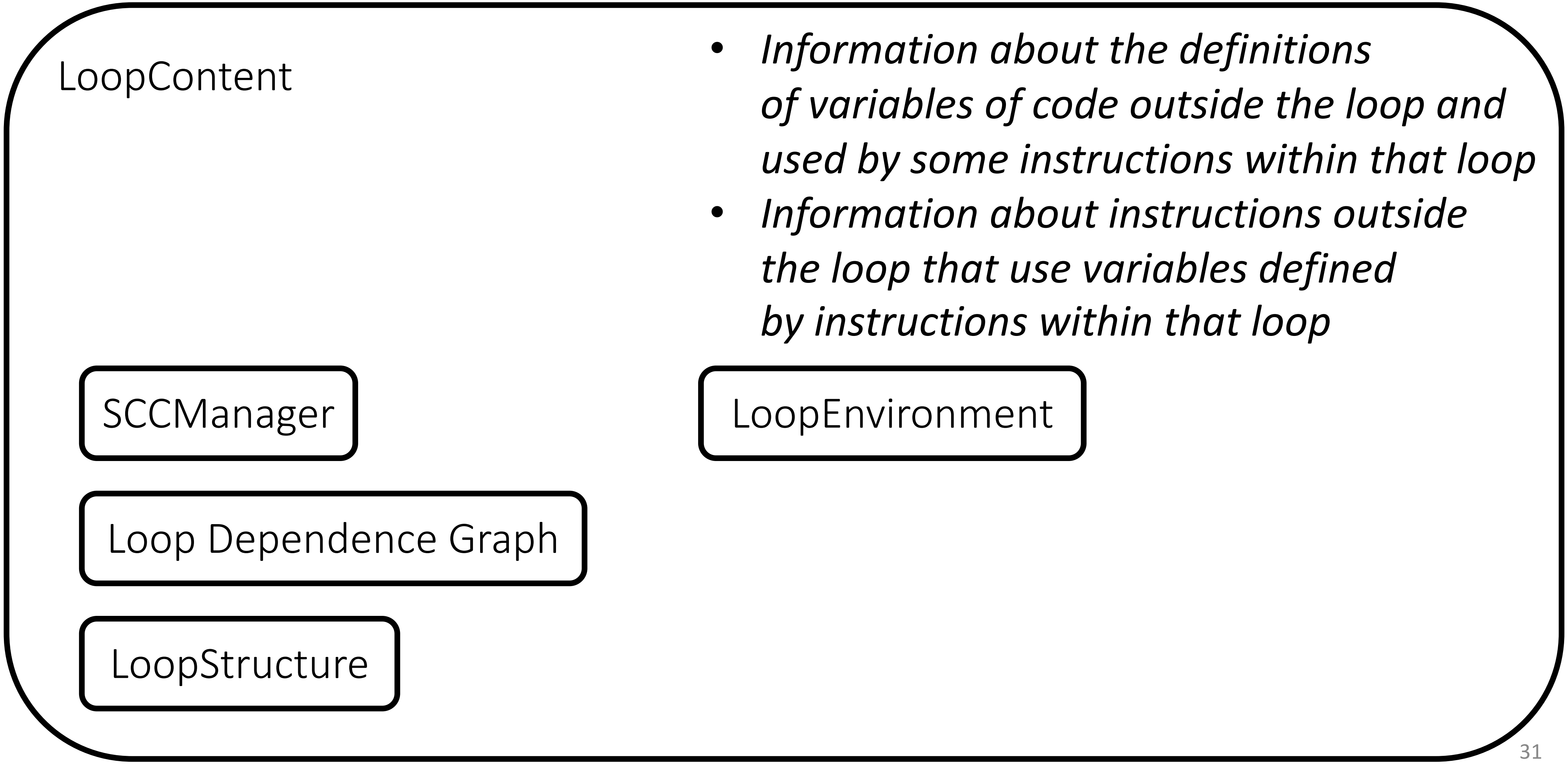
Instance of the class arcana::noelle::SCCDAG



Dependences with NOELLE

(For more information about arcana::noelle::SCCDAGAttrs*,*
please check out the tutorial dedicated to it)

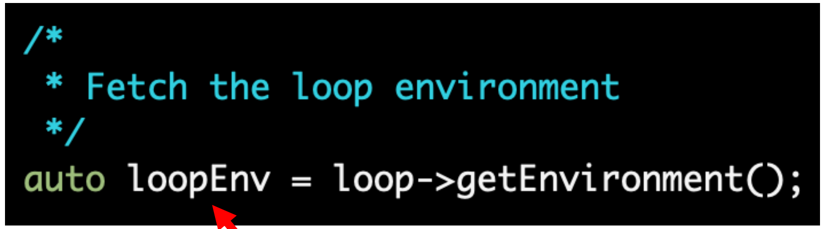
Abstractions related to loops in NOELLE



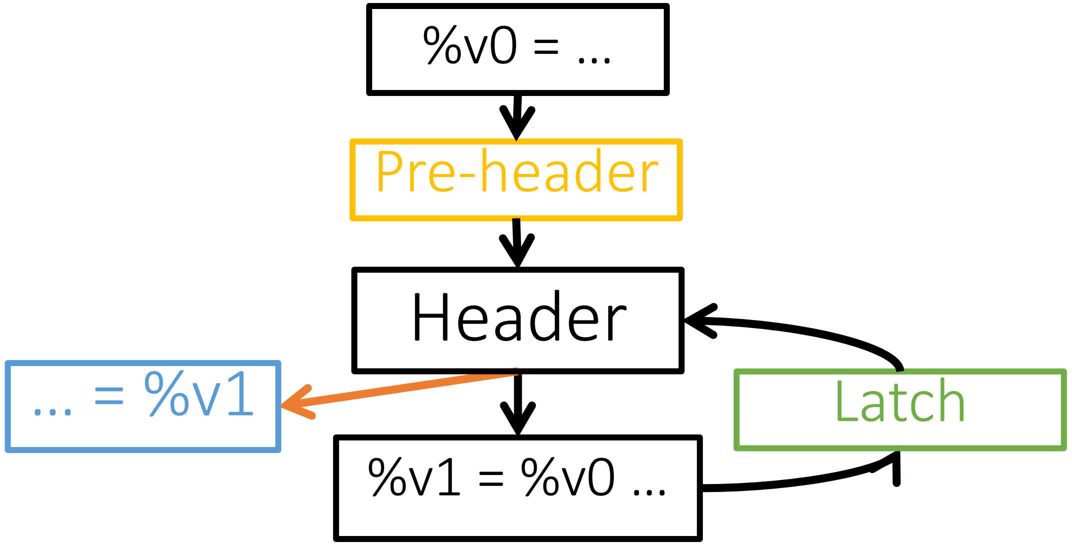
LoopContent

- *Information about the definitions of variables of code outside the loop and used by some instructions within that loop*
- *Information about instructions outside the loop that use variables defined by instructions within that loop*

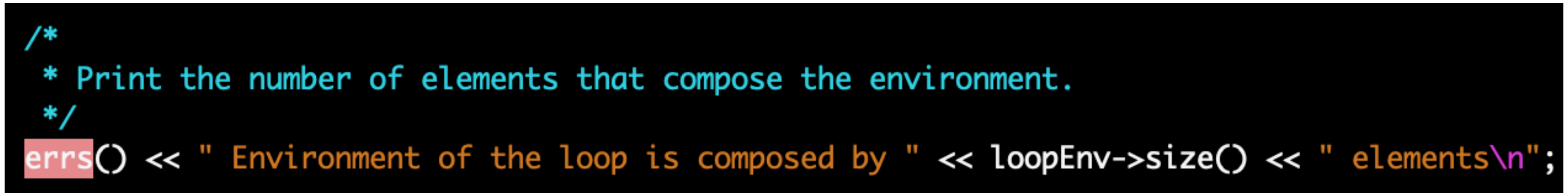
From LoopContent to LoopEnvironment



```
/*
 * Fetch the loop environment
 */
auto loopEnv = loop->getEnvironment();
```



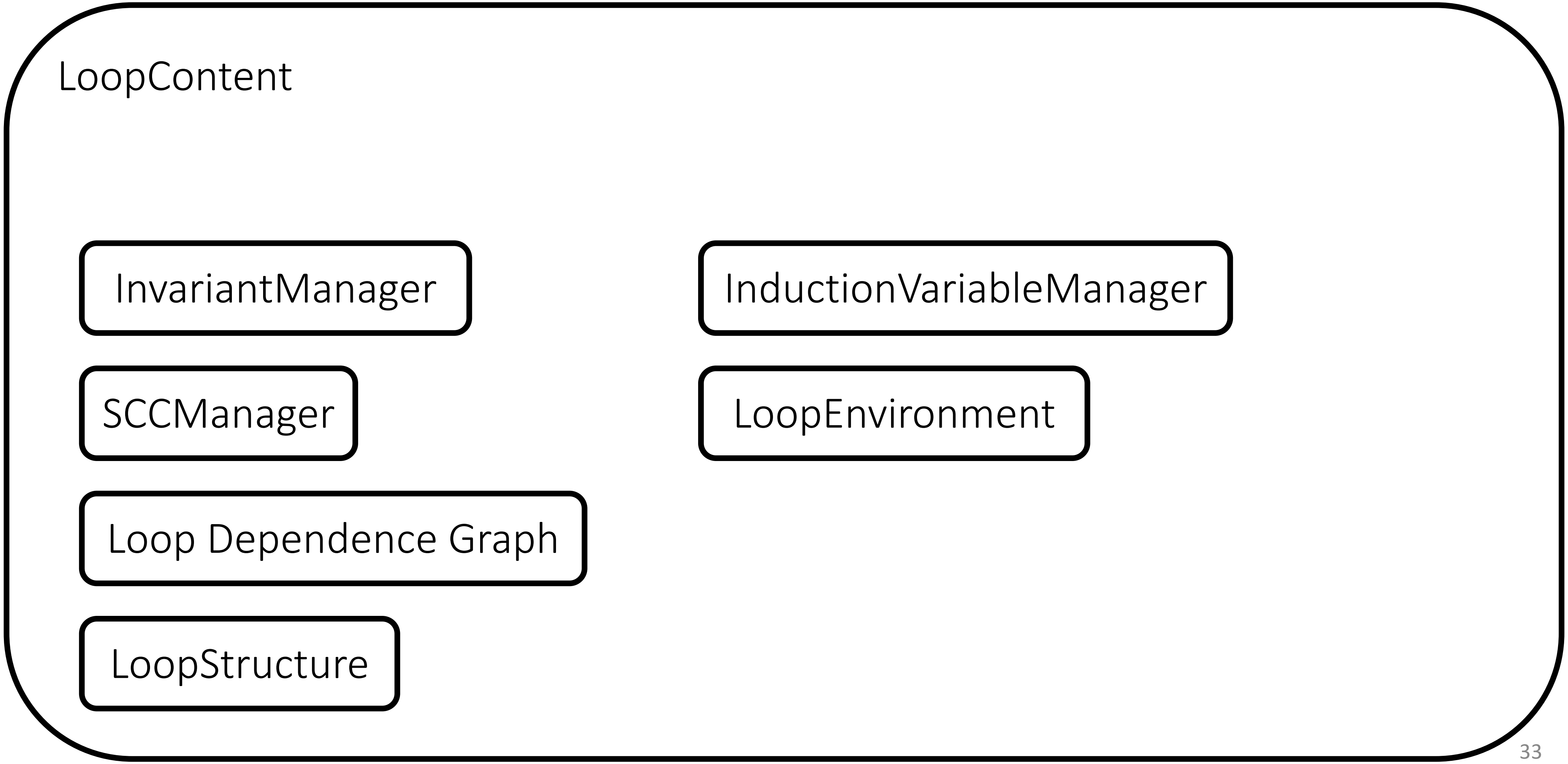
Instance of the class arcana::noelle::LoopEnvironment



```
/*
 * Print the number of elements that compose the environment.
 */
errs() << " Environment of the loop is composed by " << loopEnv->size() << " elements\n";
```

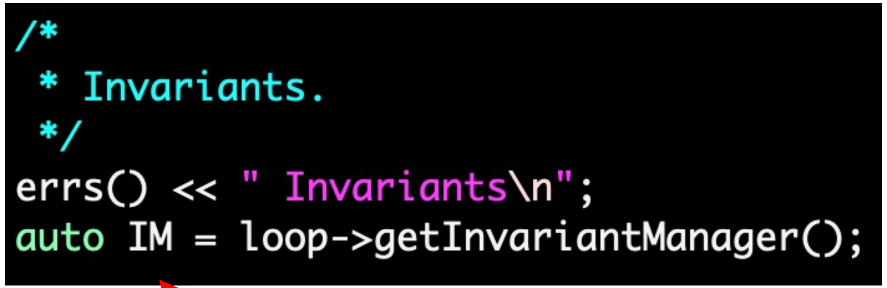
(For more information about arcana::noelle::LoopEnvironment*,*
please check out the tutorial dedicated to it)

Abstractions related to loops in NOELLE



From LoopContent
to the invariant and IV managers

- InvariantManager

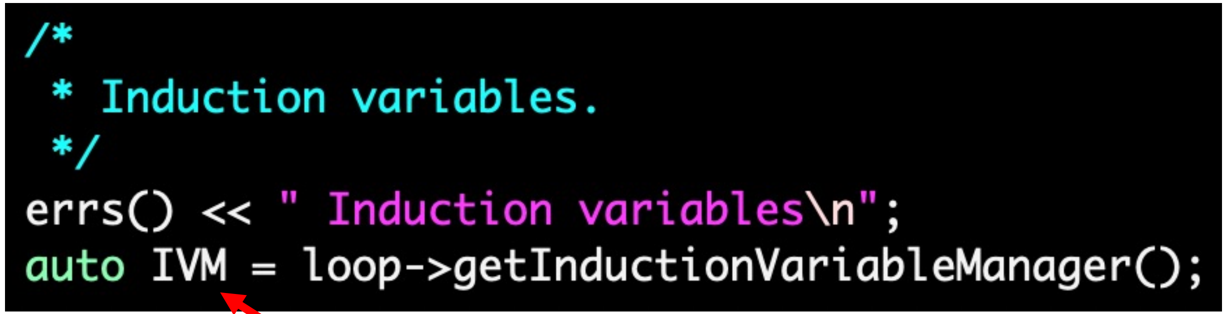


```
/*
 * Invariants.
 */
errs() << " Invariants\n";
auto IM = loop->getInvariantManager();
```

(For more information about arcana::noelle::InvariantManager*, please check out the tutorial dedicated to it)*

Instance of the class arcana::noelle::InvariantManager

- InductionVariableManager

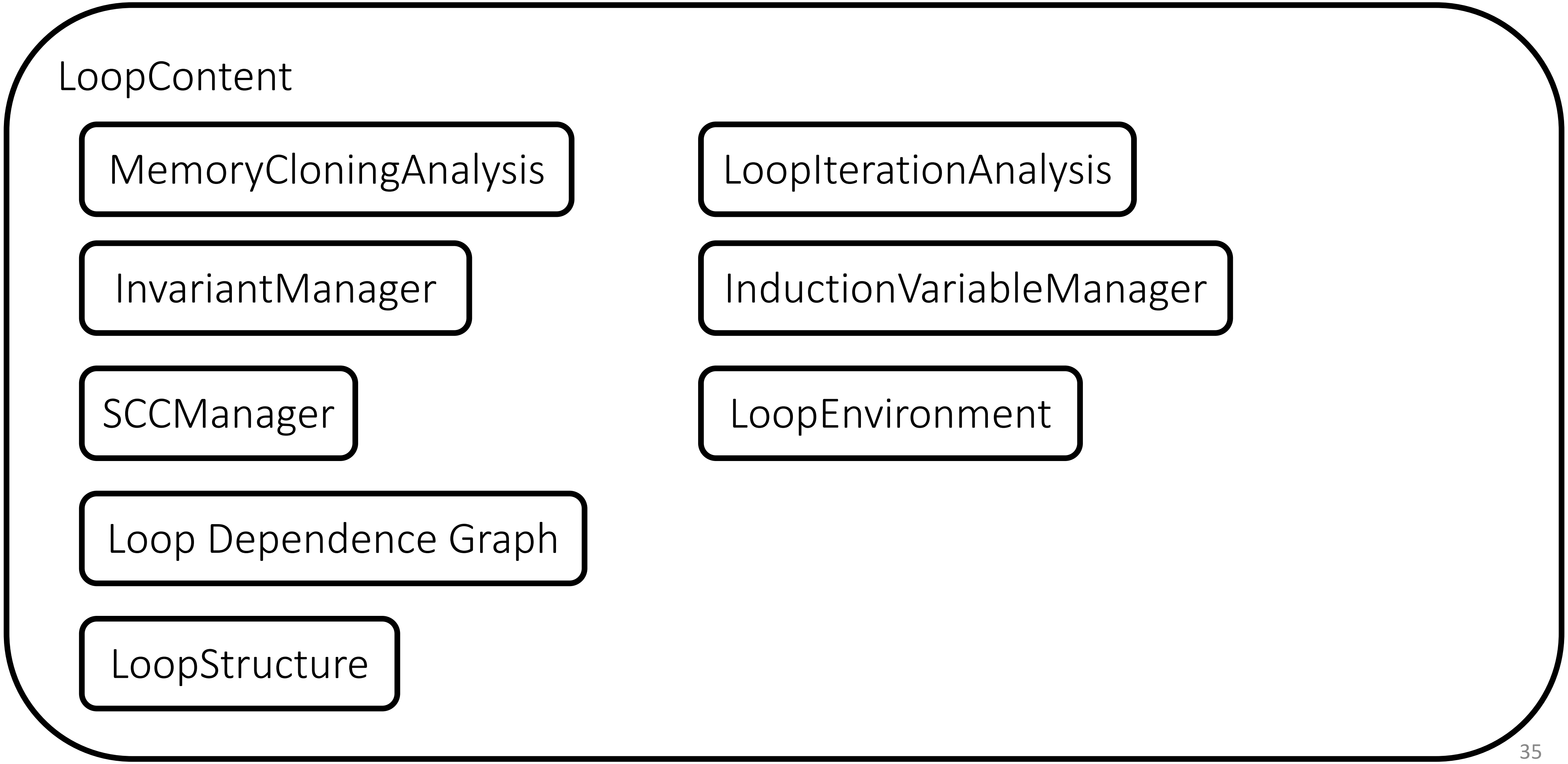


```
/*
 * Induction variables.
 */
errs() << " Induction variables\n";
auto IVM = loop->getInductionVariableManager();
```

(For more information about arcana::noelle::InductionVariableManager*, please check out the tutorial dedicated to it)*

Instance of the class arcana::noelle::InductionVariableManager

Abstractions related to loops in NOELLE



LoopContent

MemoryCloningAnalysis

InvariantManager

SCCManager

Loop Dependence Graph

LoopStructure

LoopIterationAnalysis

InductionVariableManager

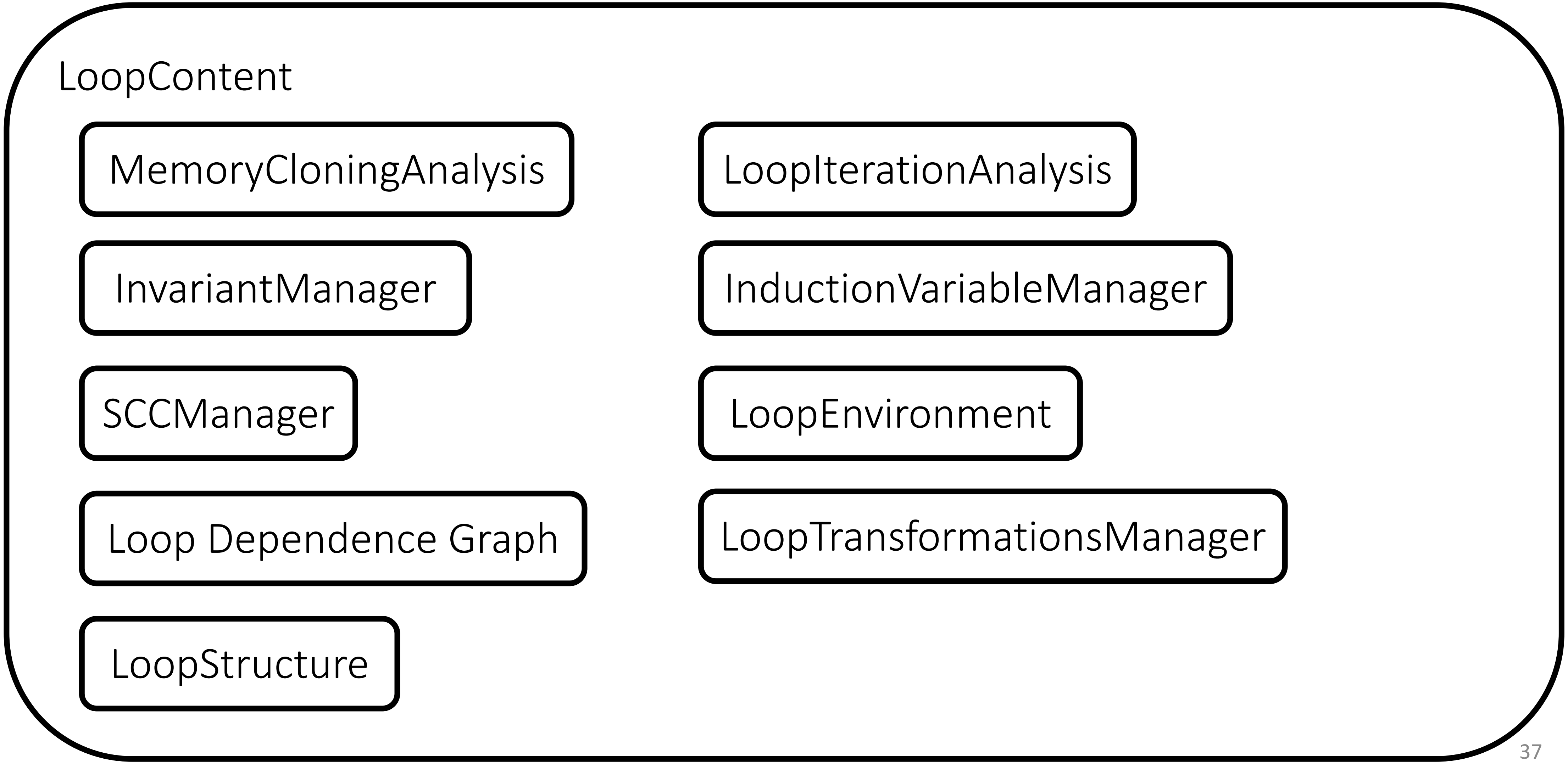
LoopEnvironment

From LoopContent
to the loop-specific analyses

- auto mca = loop->getMemoryCloningAnalysis();

- auto ita = loop->getLoopIterationSpaceAnalysis();

Abstractions related to loops in NOELLE



LoopContent

MemoryCloningAnalysis

InvariantManager

SCCManager

Loop Dependence Graph

LoopStructure

LoopIterationAnalysis

InductionVariableManager

LoopEnvironment

LoopTransformationsManager

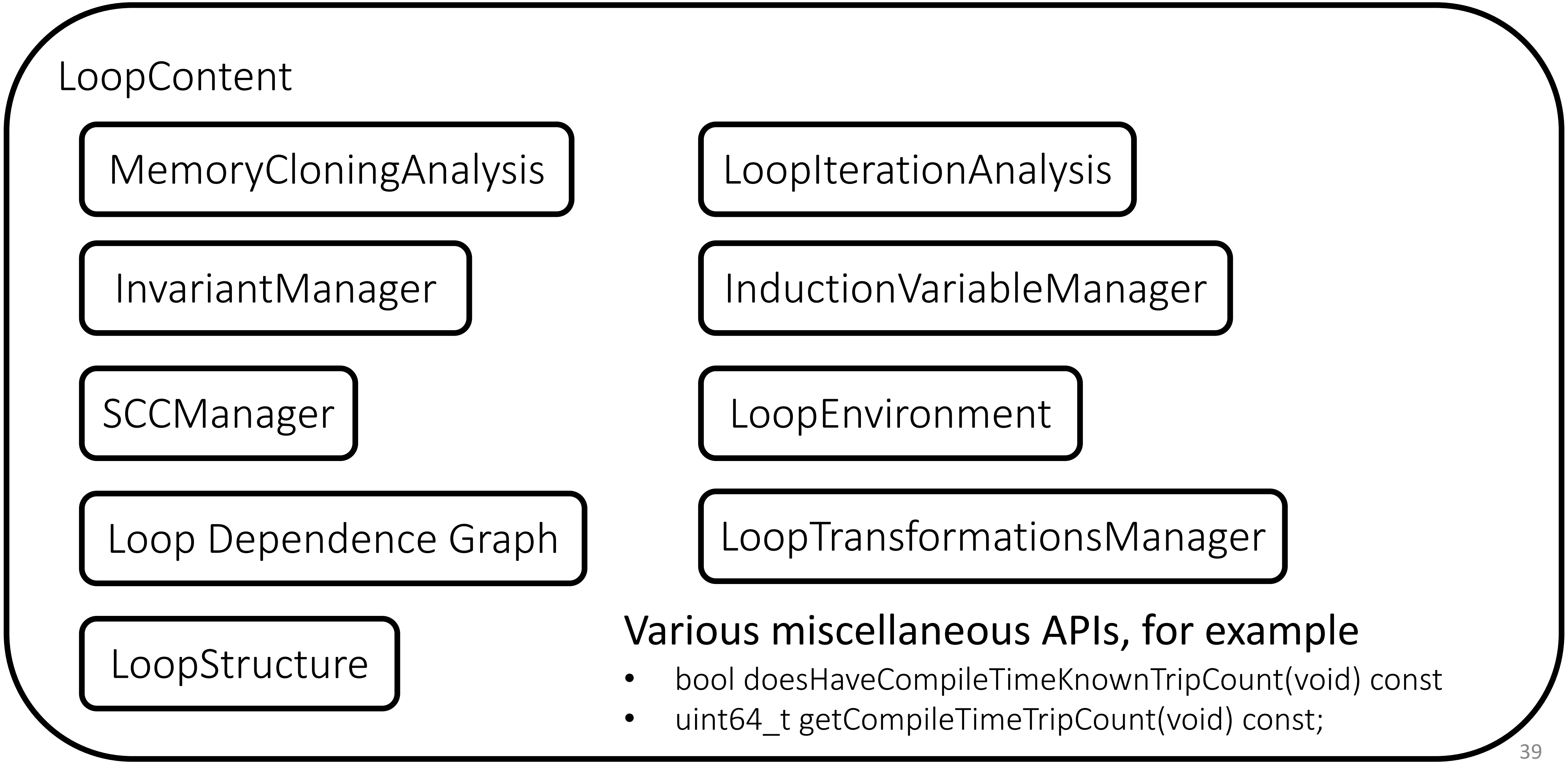
From LoopContent
to LoopTransformationsManager

```
LoopTransformationsManager *ltm = loop->getLoopTransformationsManager();

uint32_t c = ltm->getMaximumNumberOfCores();

ltm->isTransformationEnabled(Transformation::LOOP_DISTRIBUTION_ID);
```

↓

noelle/core/Transformations.hpp

Abstractions related to loops in NOELLE



LoopContent

- MemoryCloningAnalysis
- InvariantManager
- SCCManager
- Loop Dependence Graph
- LoopStructure
- LoopIterationAnalysis
- InductionVariableManager
- LoopEnvironment
- LoopTransformationsManager

Various miscellaneous APIs, for example

- bool doesHaveCompileTimeKnownTripCount(void) const
- uint64_t getCompileTimeTripCount(void) const;

Always have faith in your ability

Success will come your way eventually

Best of luck!