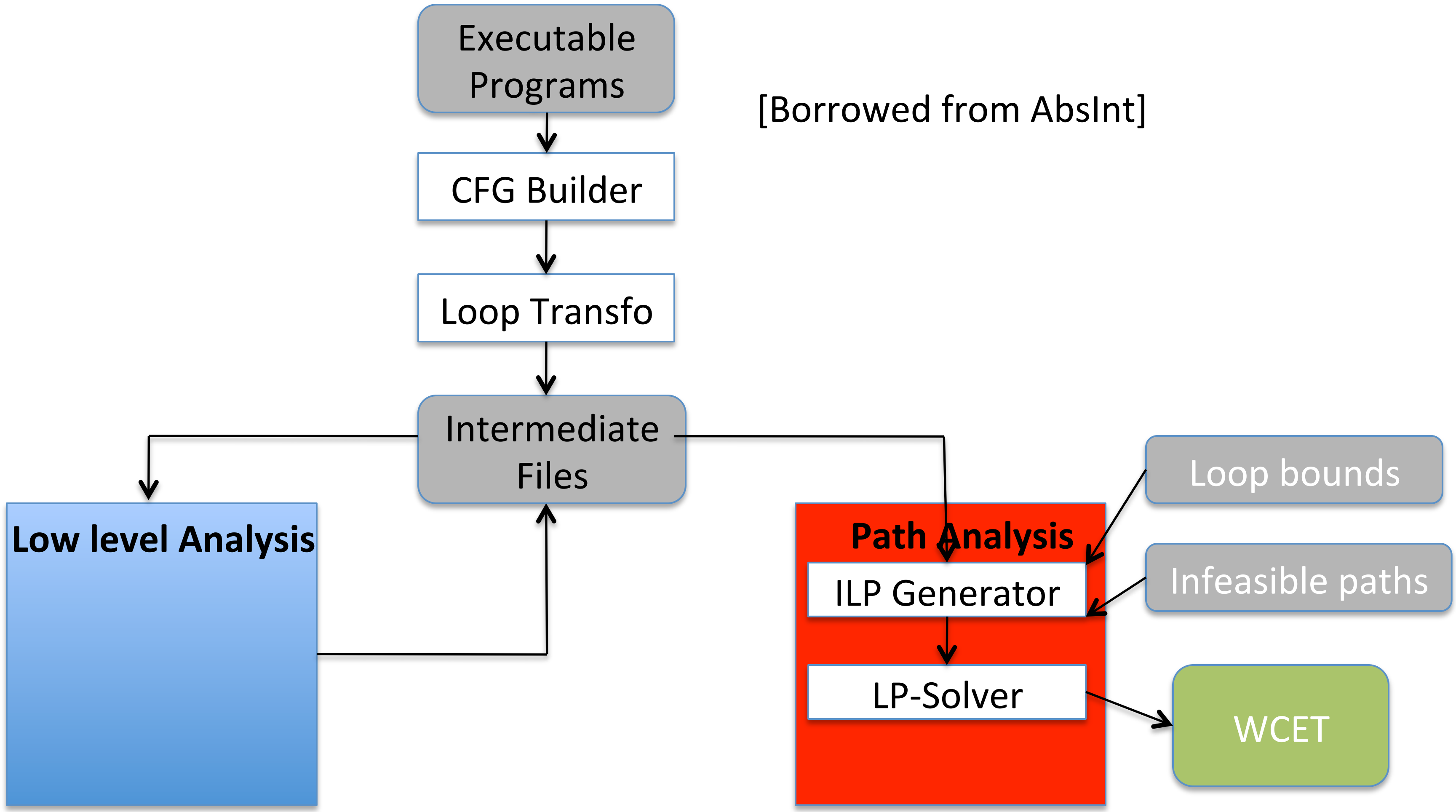
Symbolic Simulation on Complicated Loops for WCET Path Analysis

Duc-Hiep Chu and Joxan Jaffar
National University of Singapore

Natural Modularization of Static WCET Analysis

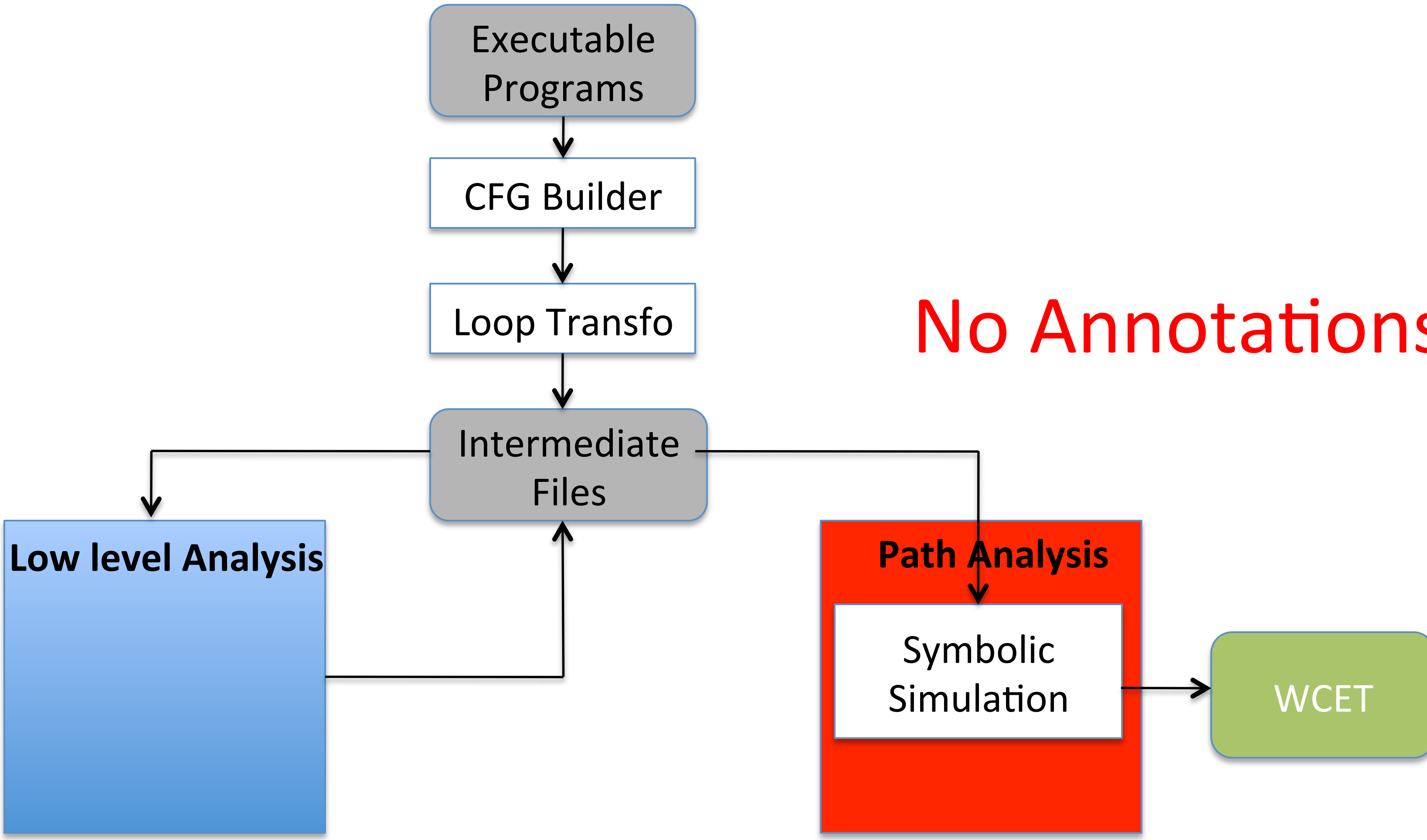


[Borrowed from AbsInt]

Path Analysis using ILP

- Simple and elegant
- Manual: users to provide loop/recursion bounds and additional constraints to exclude infeasible paths
  - Information used is not verified
  - This task is not always trivial
    - Can be error-prone
    - Users might not know of such information

Our Target



No Annotations!

Challenge 1: Complicated Loops

- Some patterns for complicated loops:
  - Triangular loops
  - Down-sampling
  - Amortized loops
  - No closed form (but terminating)
- They challenge the aggregation process
- Two options:
  - Unrolling: accurate but not scalable in general
  - Loop Abstraction (e.g. loop invariant or fixed-point computation): more scalable but not accurate

Challenge 2: Infeasible Paths

- Good detection of infeasible paths concerns path-sensitivity
- In theory, intractably many infeasible paths
  - Providing annotations for them is not plausible
- In ILP practice
  - Hard to come up with annotations for infeasible paths which stretch over loops and nested loops

Our Approach

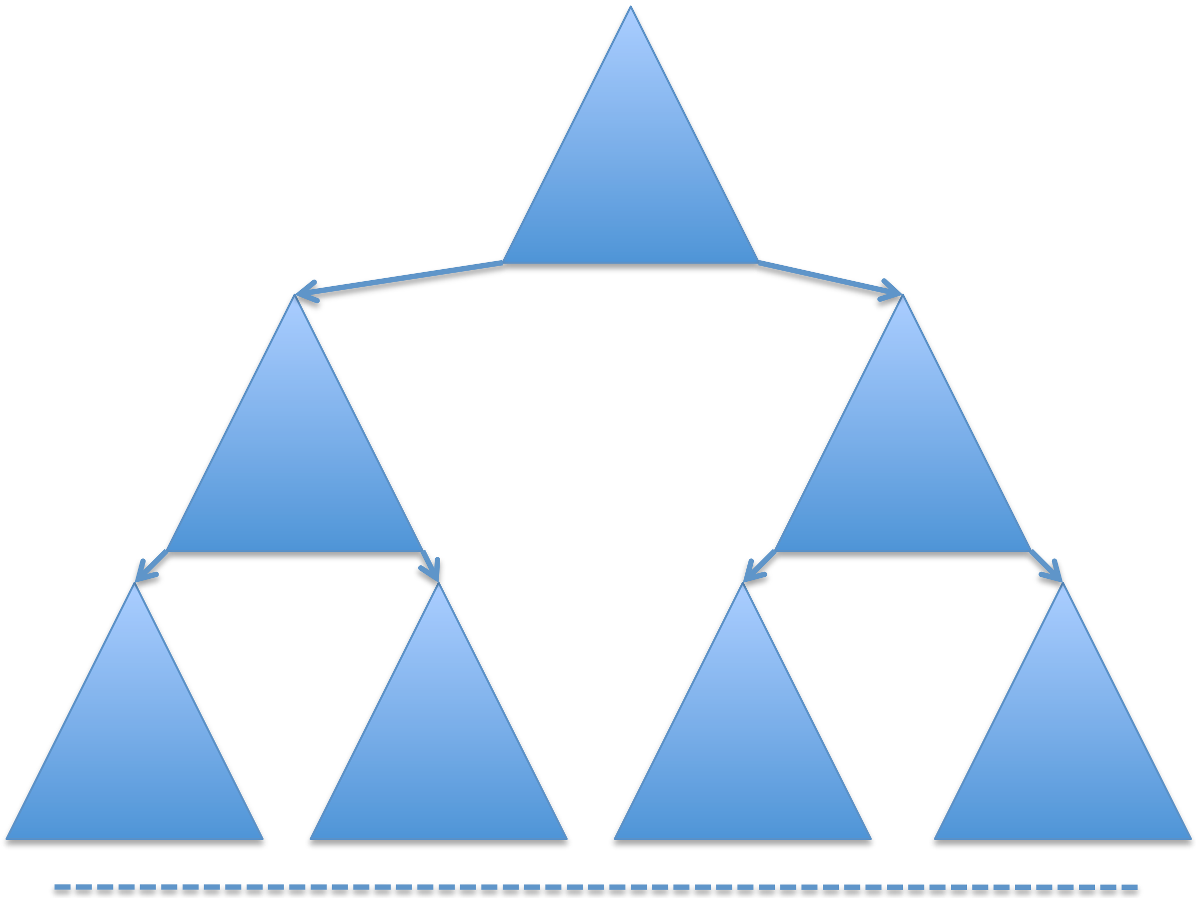
- Symbolic simulation as a brute-force method
  - Loops are unrolled
  - We attempt path-sensitivity
  - Similar to running a program but we are proving it
  - Can be widely applied to different programs and problems
- Question: how to make this scalable? In general, symbolic simulation is:
  - At least proportional to the execution of the WCET path
  - Very expensive as

  Estimated #states = 2 ^ #states_per_average_ground_run
- In short, we need to deal with the state explosion problem of the symbolic tree in an ANALYSIS problem
- Empirically, we overcome both issues mentioned above

Our Approach

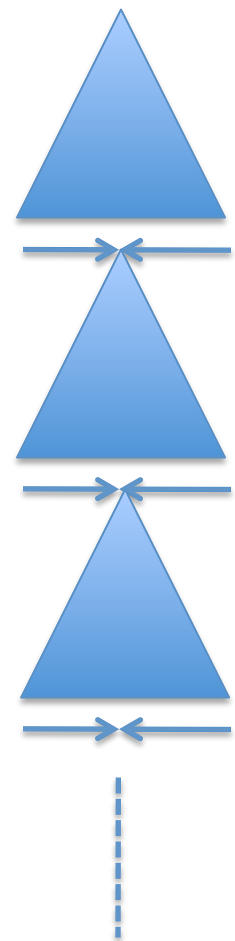
- Iteration Abstraction
  - Path merging (as in [Lundqvist99] and [Gustaffson05])
  - We only perform at the end of each loop body
  - We use polyhedral domain
- Compounded Summarization with Interpolation
  - We are summary-based
  - Interpolants tell us when we can safely reuse
  - Compounded both horizontally and vertically
- Witness Path
  - Witness path conditions tell us when we can precisely reuse (i.e. strengthen the interpolant)

Naïve Simulation Does Not Scale



Iteration Abstraction

Multiple contexts are merged into one

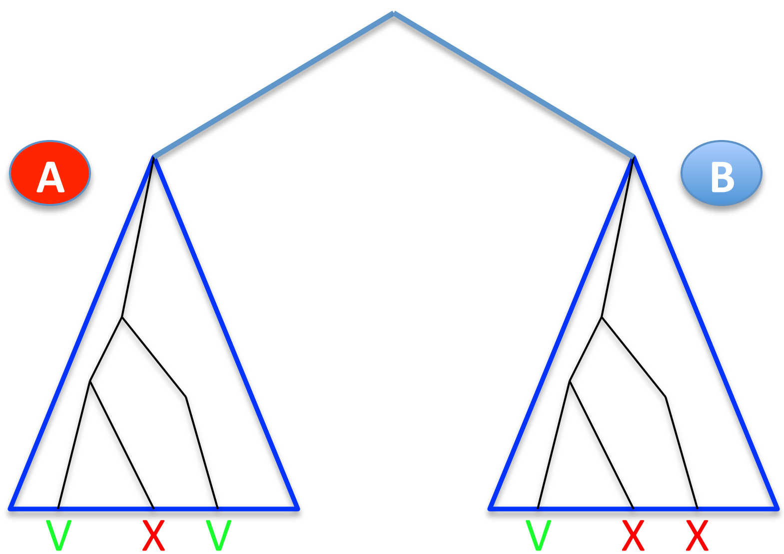


Iteration Abstraction

- Similar to abstract execution[Gustafsson05]
  - They used interval domain
  - We use polyhedral domain (convex hull)
    - First introduced to program analysis by [P. Cousot and N. Halbwachs, POPL'78]
- In general, we might lose information due to abstraction
- Fortunately, most variables affecting control flows of the program are transformed linearly
- Unresolved problems:
  - The depth of the tree is still the depth of the longest path
  - # paths are still exponential wrt # branches outside loops

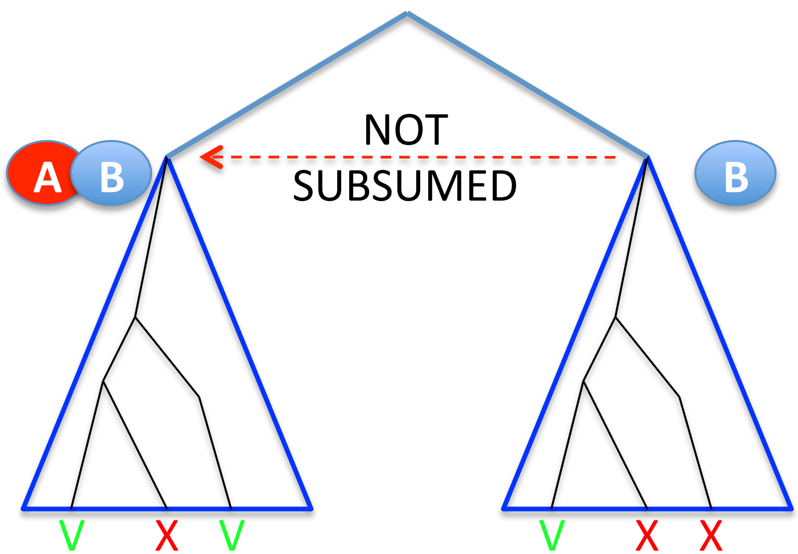
Summarization with Interpolation

A and B are sibling sub-trees (same program point, different context)



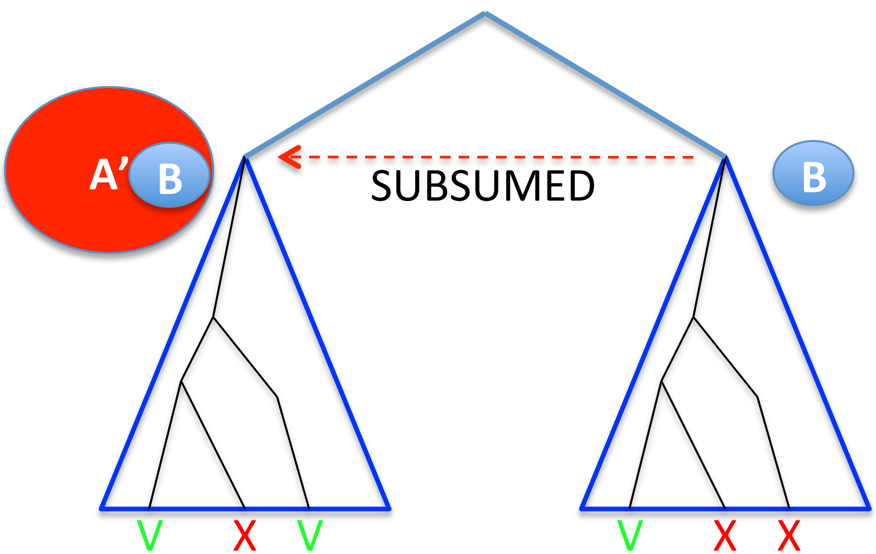
Summarization with Interpolation

A and B are sibling sub-trees (same program point, different context)



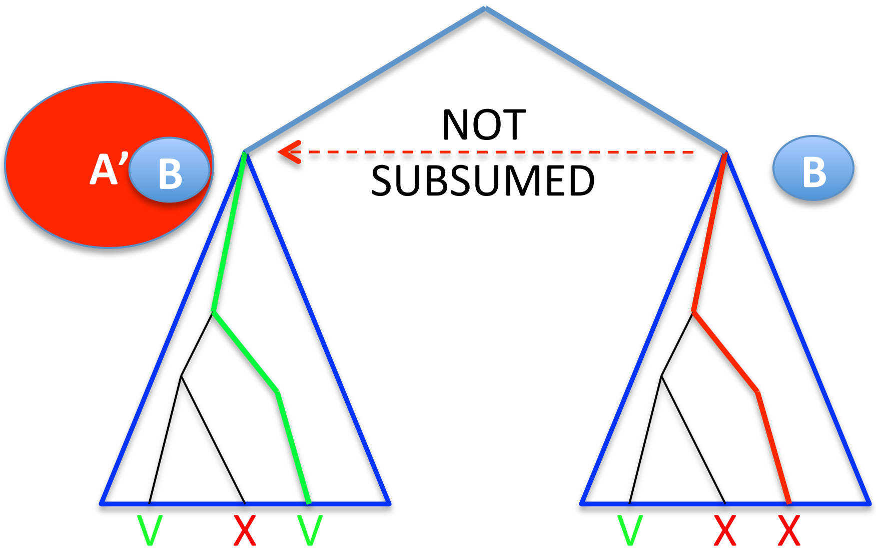
Summarization with Interpolation

A and B are sibling sub-trees (same program point, different context)



Generalize A (to A') while preserving infeasibility: B has no more feasible paths than A

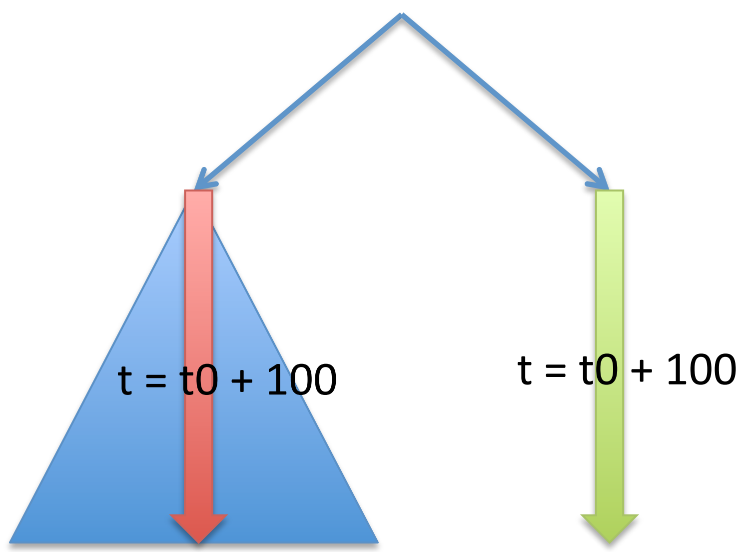
Witness Paths



- Witness path depicting best found solution for sub-tree A
- Mirror path in sibling sub-tree B
- Though B can safely re-use the analysis of A, best path of A is in fact infeasible in B

Breadth-wise Reuse of Summarization

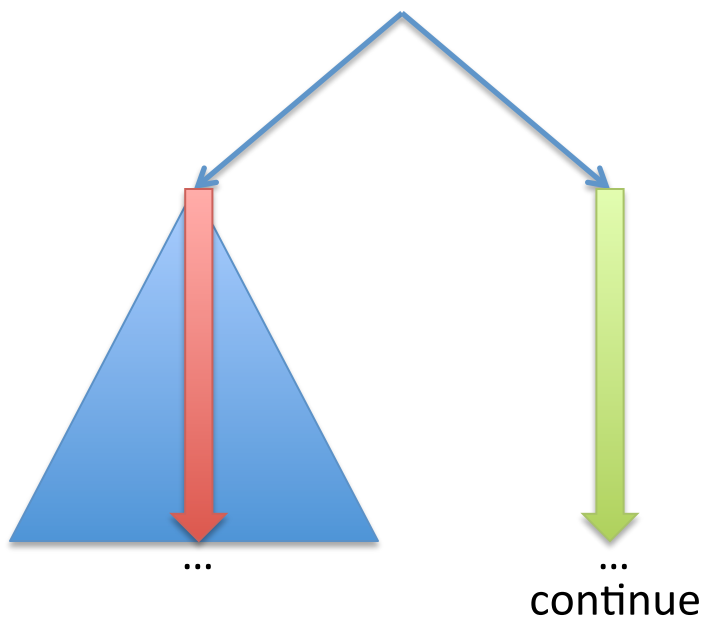
- Use the summarization to produce the solution



The condition for reuse is determined by interpolation and witness paths

Reuse of Summarization

- The leaves of the sub-tree need not be terminal



We need cut-off points and continuation contexts

Reuse of Summarization

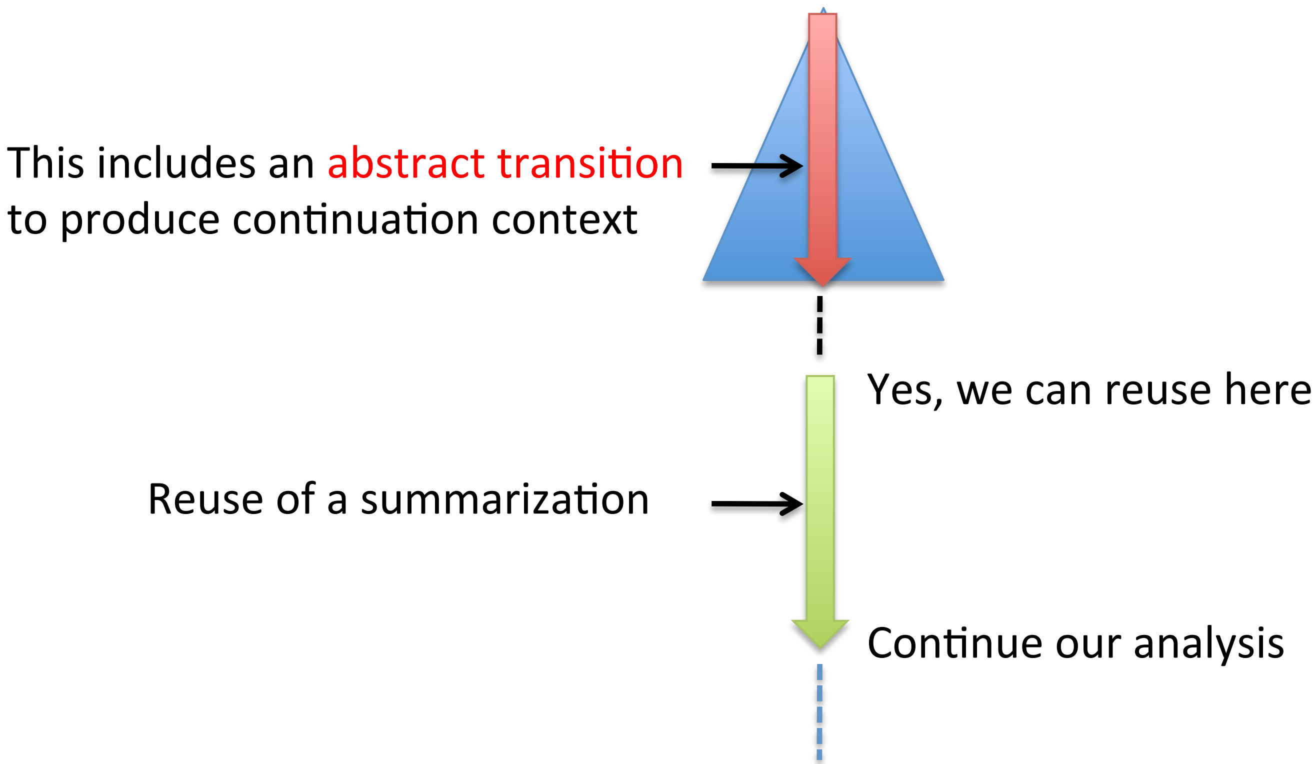
- To produce continuation context, we require the notion of Abstract Transformer
  - Gives an (abstract) input-output relationship for a finite sub-tree
  - Natural cut-off points:
    - Ending point of loop body
    - Ending point of function body
  - Again we compute it using hulling in polyhedral domain

  E.g. <1> if (*) x++; else x += 2; <2>

  Abstract transformer $\Delta = x + 1 \le x' \le x + 2$

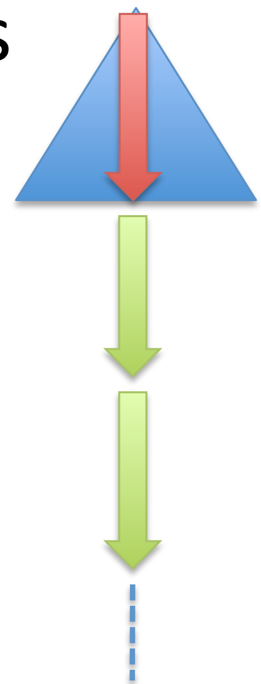
Depth-wise Reuse of Summarization

- Reuse is not just for sibling



Depth-wise Reuse of Summarization

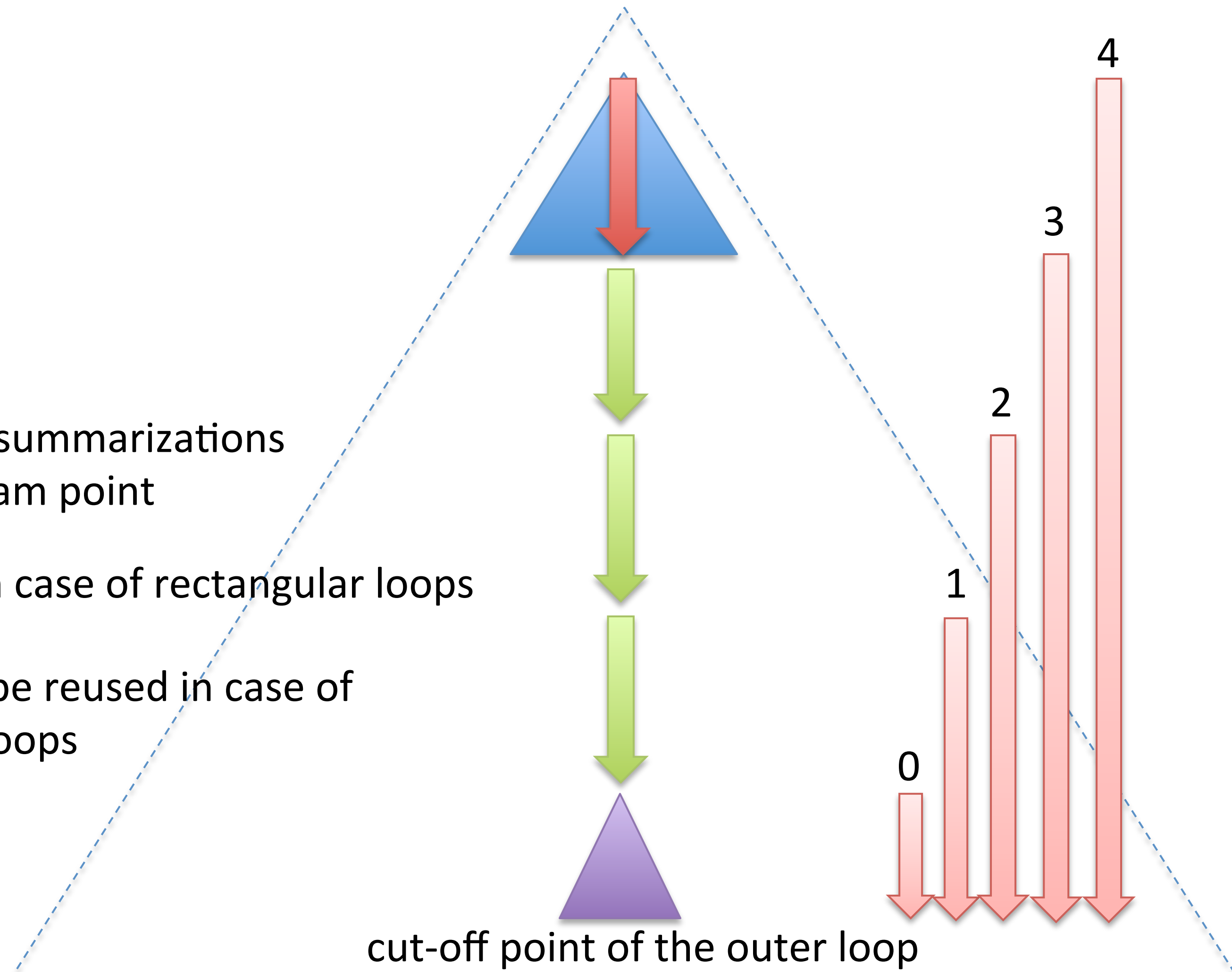
- Very often, the analysis tree for an un-nested loop looks like this



Depth-wise Loop Compression

- We just showed the benefits of abstracting and summarizing each iteration of a loop
- How about summarizing the whole loop?
  - It benefits when dealing with nested loops

Depth-wise Loop Compression



A serialization of summarizations
for a single program point

4 will be reused in case of rectangular loops

0,1,2,3 will likely be reused in case of
non-rectangular loops

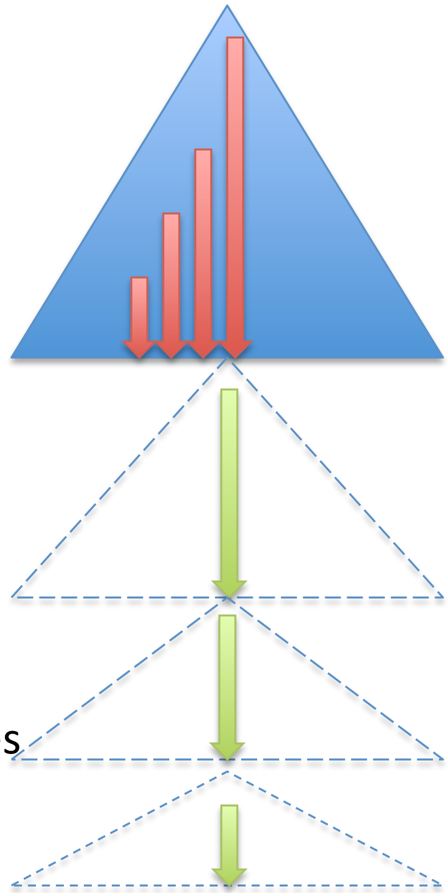
Depth-wise Loop Compression

This is the case for `bubblesort`
(a classic example for triangular loop)

We discover the whole triangle by just (fully) exploring the first iteration of the outer loop

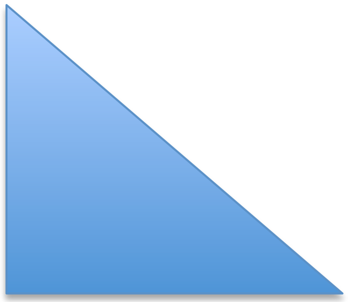
The number inner loop's iterations being explored is just linear (Note: only one is fully explored, while the rest are partially explored)

This separates us from other simulation techniques

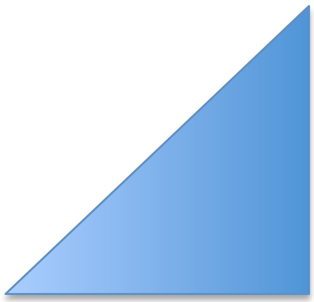


Triangular Loop

- We have done well for this type of triangle

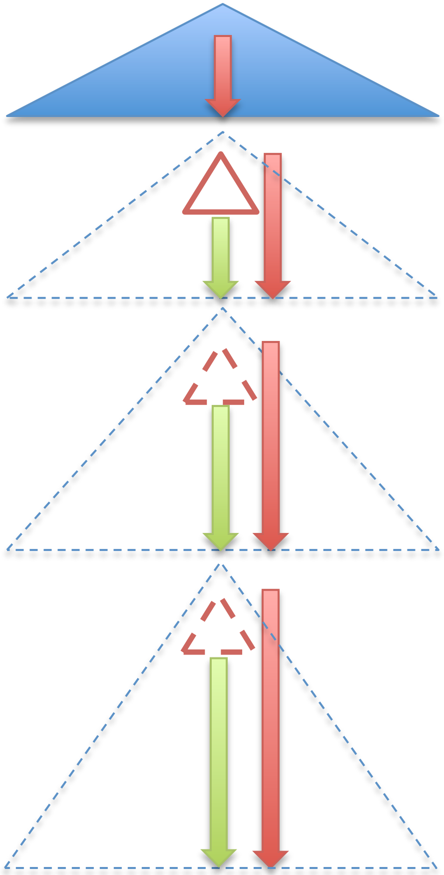


- How about this? (e.g. `insertsort`)



Triangular Loop

It is still linear



Experimental Results

Benchmark	Size Parameter	Complexity	WCET	States	Time(ms)	Exact?	
						Manual	Proof
bubblesort	n=25 n=50 n=100	O(n^2)	1648 6423 25348	135 260 510	233 701 2438	Y	N
expint	NA	-	859	519	8247	Y	Y
fft1	n=8 n=16 n=32 n=64	O(nlogn)	181 379 791 1661	111 176 287 495	446 927 2197 6818	Y	Y
fir	NA	-	760	108	387	Y	Y
insertsort	n=25 n=50 n=100	O(n^2)	1120 4120 15745	159 309 609	387 1504 7542	Y	N
j_complex	NA	-	534	165	491	N	N
ns	n=5 n=10 n=20	O(n^4)	2655 35555 522105	63 103 183	59 116 344	Y	Y
nsichneu	NA	-	281	334	15542	Y	N
ud	NA	-	819	487	1137	Y	Y

Experimental Results

Benchmark	Size Parameter	Complexity	WCET	States	Time(ms)	Exact?	
						Manual	Auto
amortized	n=50 n=100 n=200	O(n)	394 792 1590	95 186 339	287 1035 4057	Y	Y
two_shapes	n=50 n=100 n=200	O(n^2)	2199 8149 31299	259 509 1009	497 3235 19839	Y	Y
non_deter	n=25 n=50 n=100	O(n^2)	3904 15304 60604	129 242 467	59 116 344	Y	Y
tcas	NA	-	99	6020	15925	Y	Y

Exactness

- Meaning?
  - It's the best a path analyzer can do
  - Implication: want a better bound? improve our low-level analysis
- Proof?
  - Sometimes it is achievable

Proof of Exactness

- Case 1: Single-path programs
  - Power of the abstract domain and/or the theorem prover plays an important role
- Case 2: Multi-path programs
  - The solver is complete wrt the witness condition of the worst-case path and
  - The worst-case path involves no "destructive merges"
    - No loop or no path merging due to loop
    - There are path merging, but they are not lossy ([Thakur08])

Conclusion

- Fully automated WCET path analysis
  - The bound is proved safe wrt to what the low-level analysis component has produced
- The complexity of the analysis can be asymptotically better than a ground run
- Many times, we get exact bound, even for programs with complicated loops
  - Sometimes we have a proof of exactness

Thank you!

Question?