Efficiency, Precision, Simplicity, and Generality in Interprocedural Data Flow Analysis: Resurrecting the Classical Call Strings Method

Uday P. Khedker and Bageshri Karkare

Department of Computer Science and Engineering,
Indian Institute of Technology, Bombay

Dec 2007

Efficiency, Precision, Simplicity, and Generality in Interprocedural Data Flow Analysis:
Resurrecting the Classical Call Strings Method

Uday P. Khedker and Bageshri Karkare

Department of Computer Science and Engineering,
Indian Institute of Technology, Bombay

Dec 2007

Outline

- Issues in interprocedural analysis
- The classical call strings approach to interprocedural data flow analysis
- The proposed variant of call strings approach
- Empirical Results
- Conclusions

A clarification:

Data flow analysis has nothing to do with data flow architectures!

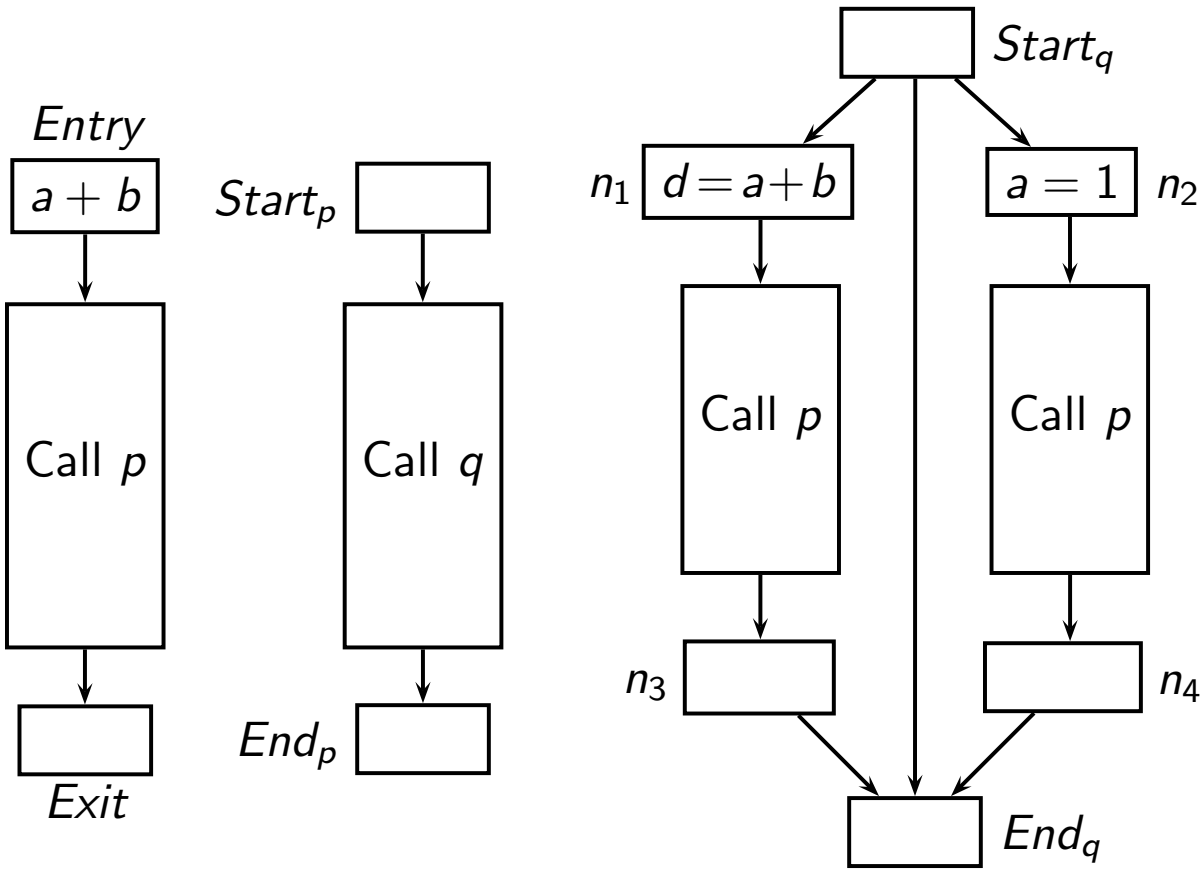
Part 2

Issues in Interprocedural Analysis

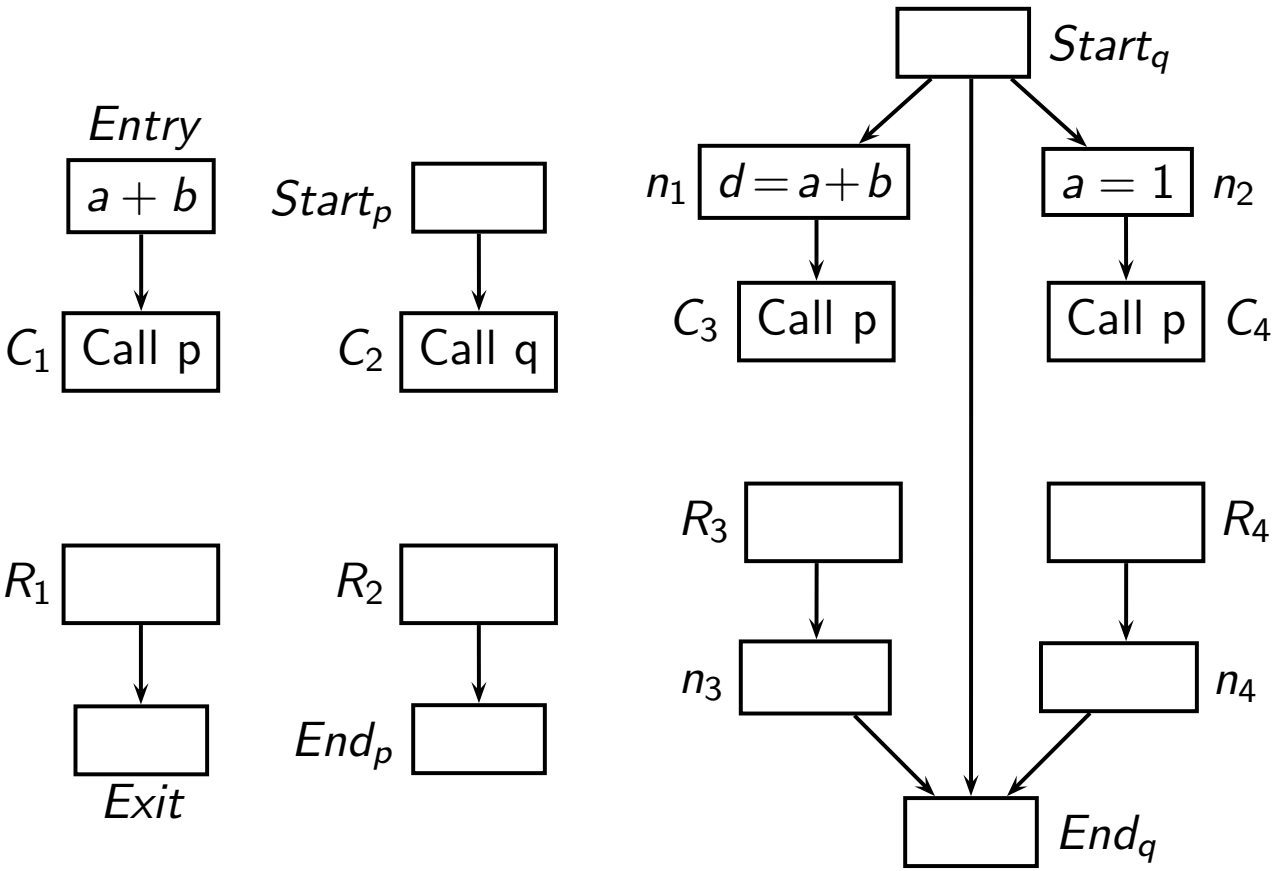
Interprocedural Analysis: Overview

- Extends the scope of data flow analysis across procedure boundaries
  Incorporates the effects of
  - procedure calls in the caller procedures, and
  - calling contexts in the callee procedures.
- Approaches :
  - Generic : Call strings approach, functional approach.
  - Problem specific : Alias analysis, Points-to analysis, Partial redundancy elimination, Constant propagation

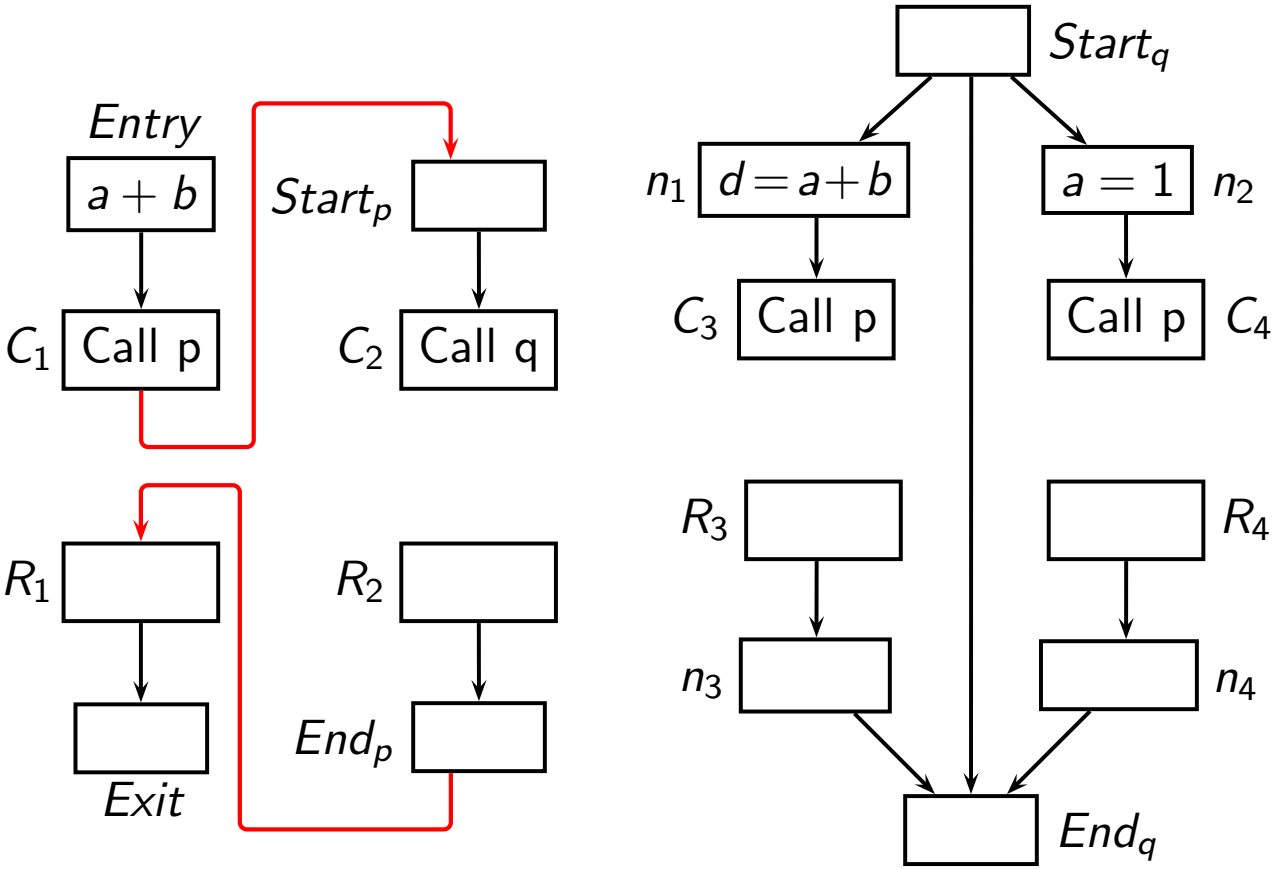
Defining Interprocedural Context for Static Analysis



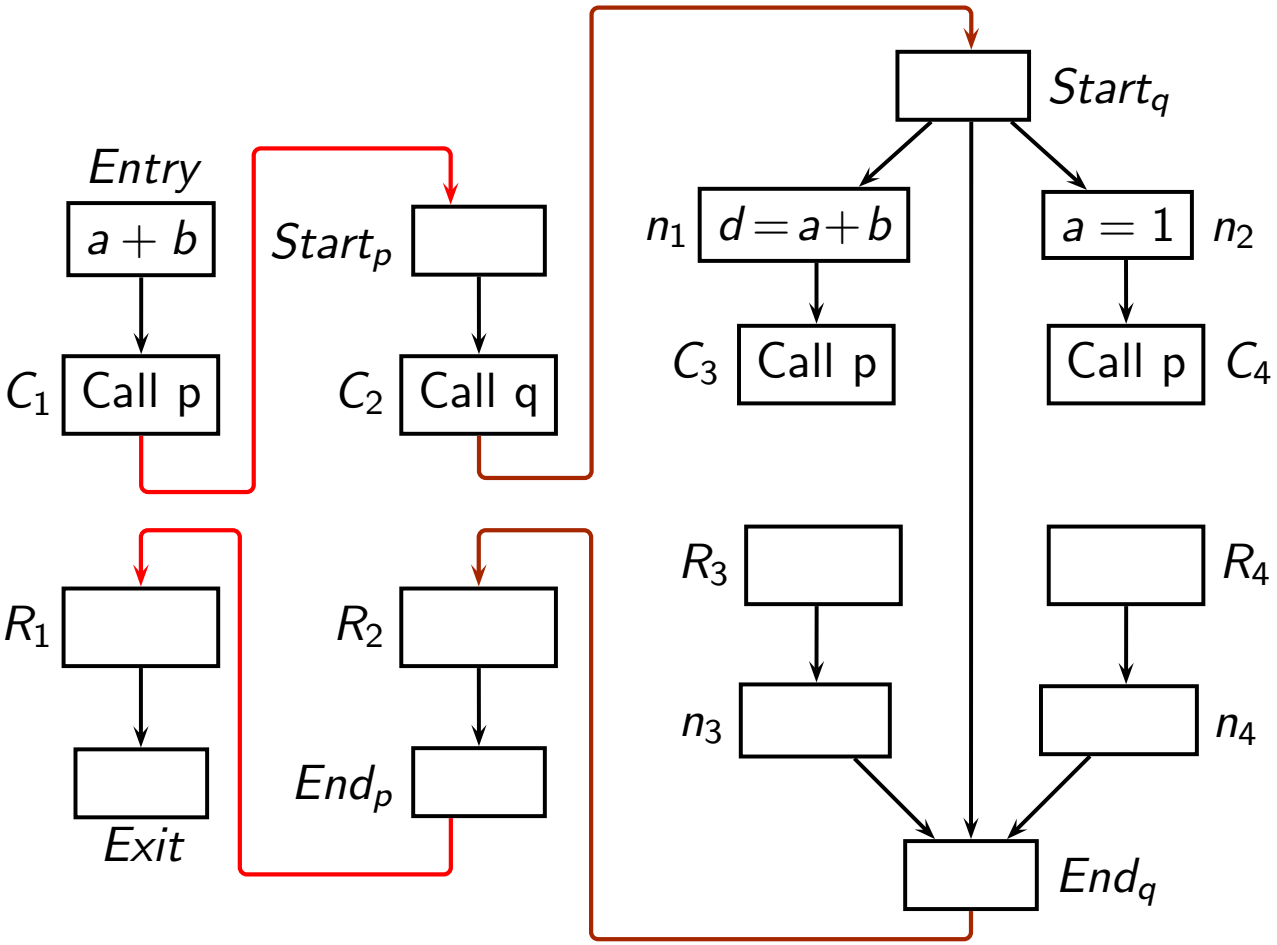
Defining Interprocedural Context for Static Analysis



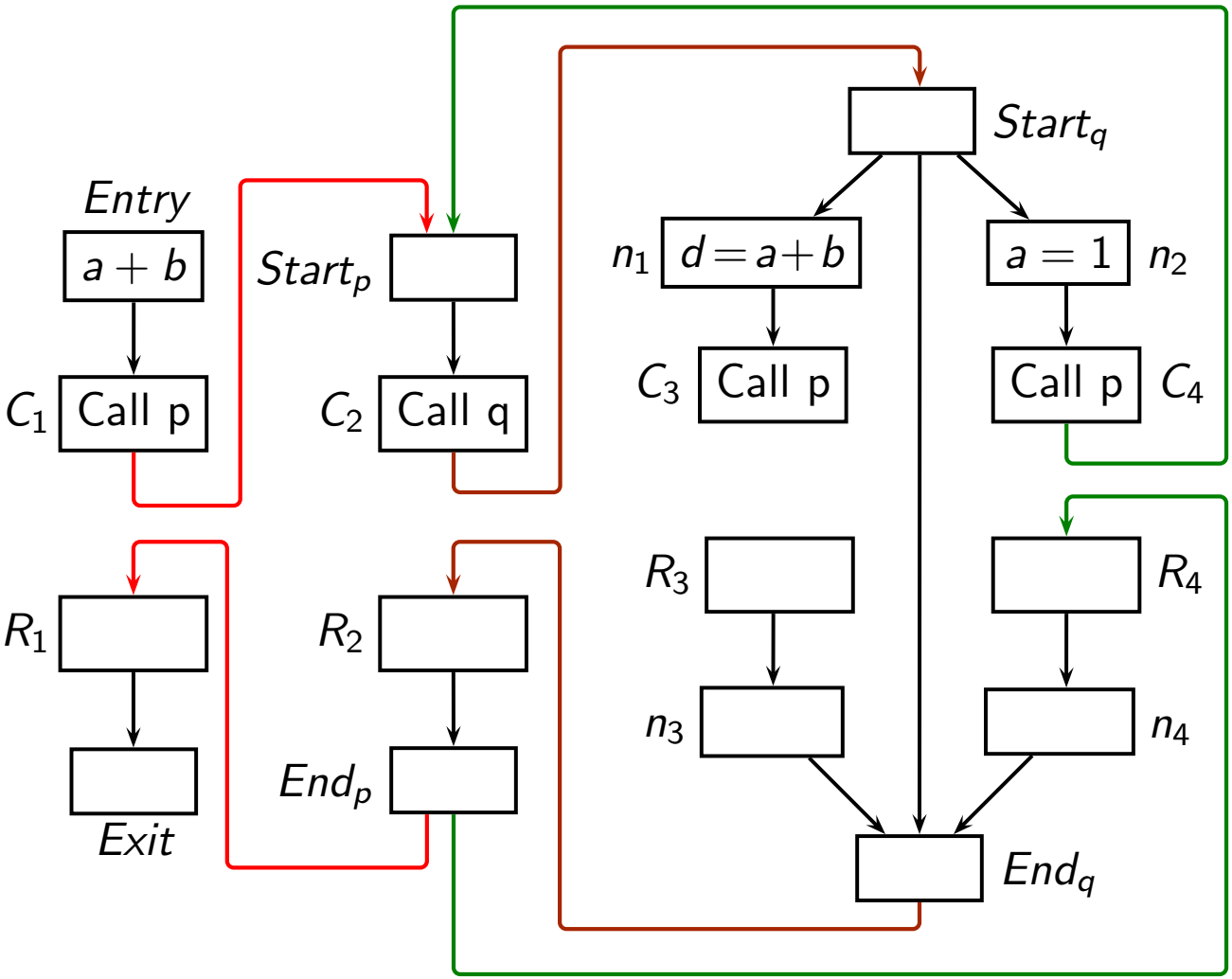
Defining Interprocedural Context for Static Analysis



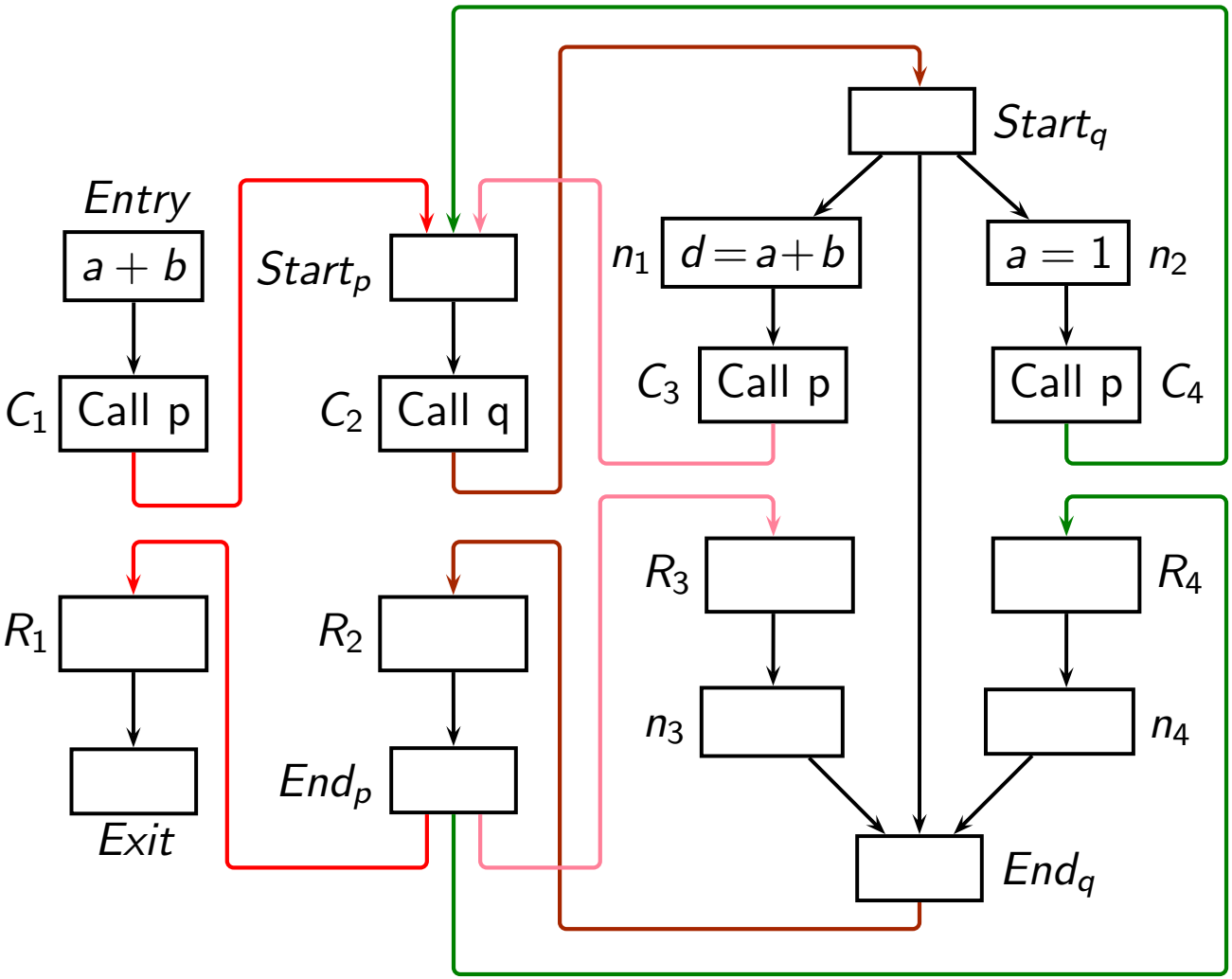
Defining Interprocedural Context for Static Analysis



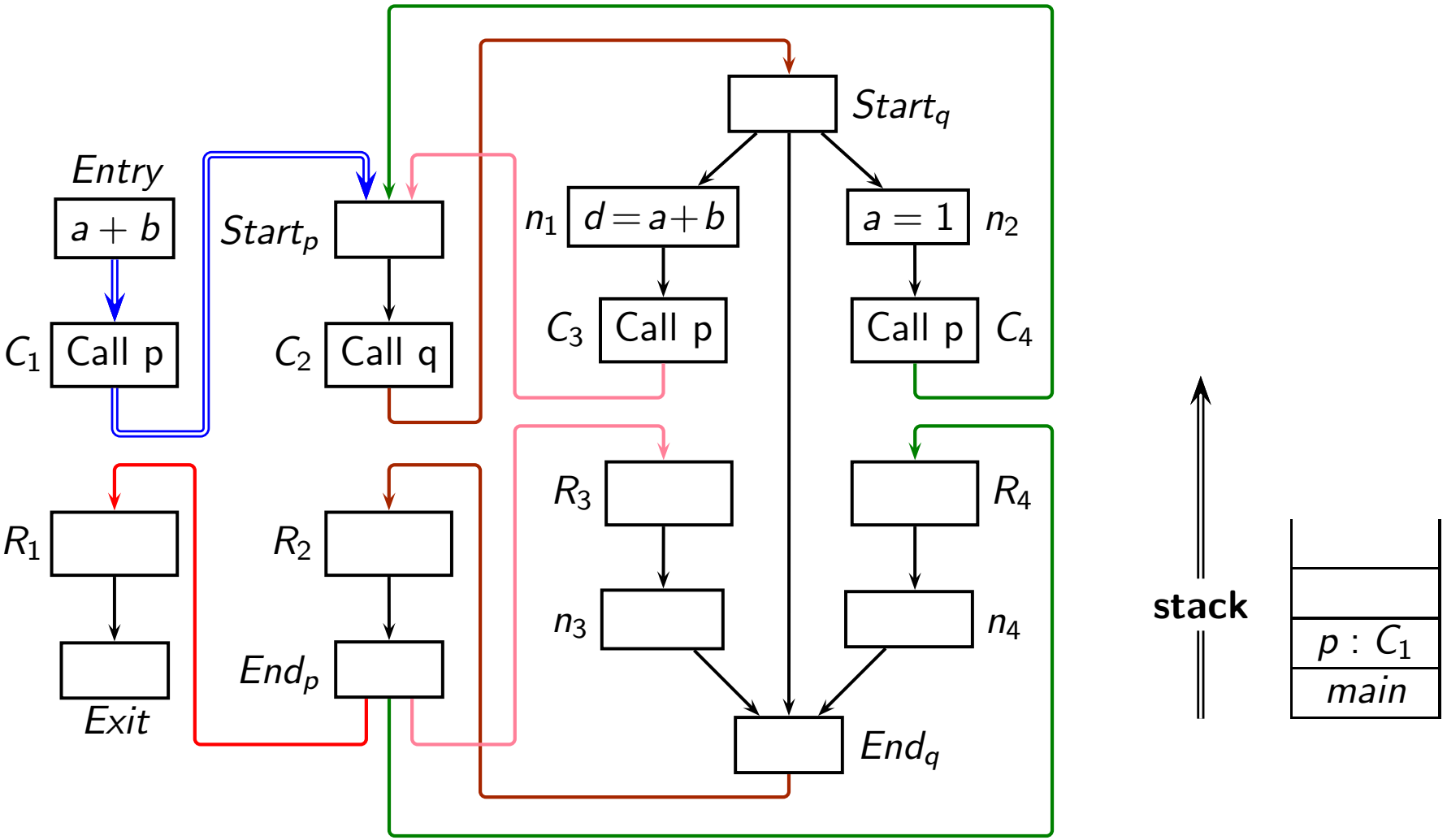
Defining Interprocedural Context for Static Analysis



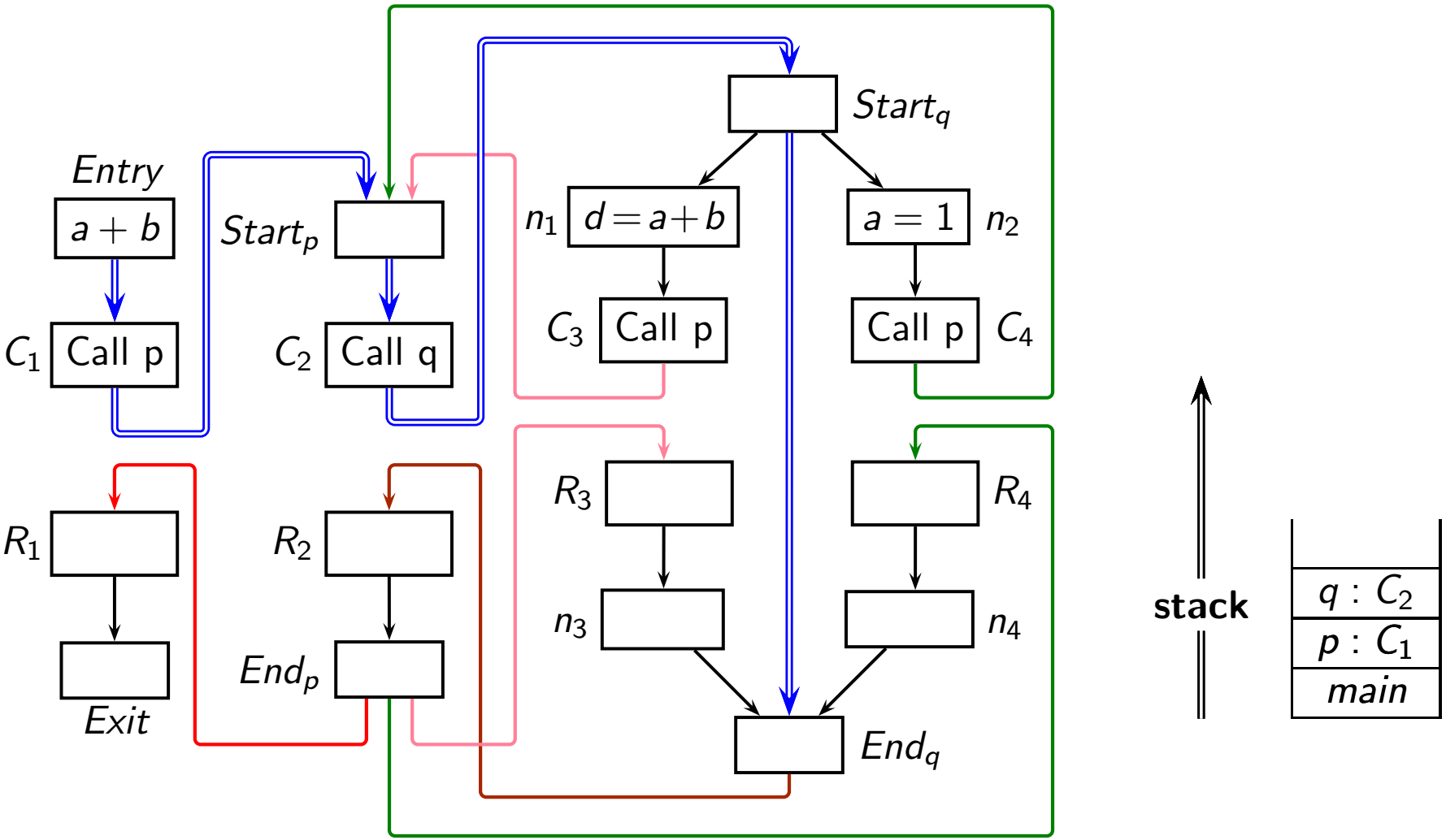
Defining Interprocedural Context for Static Analysis



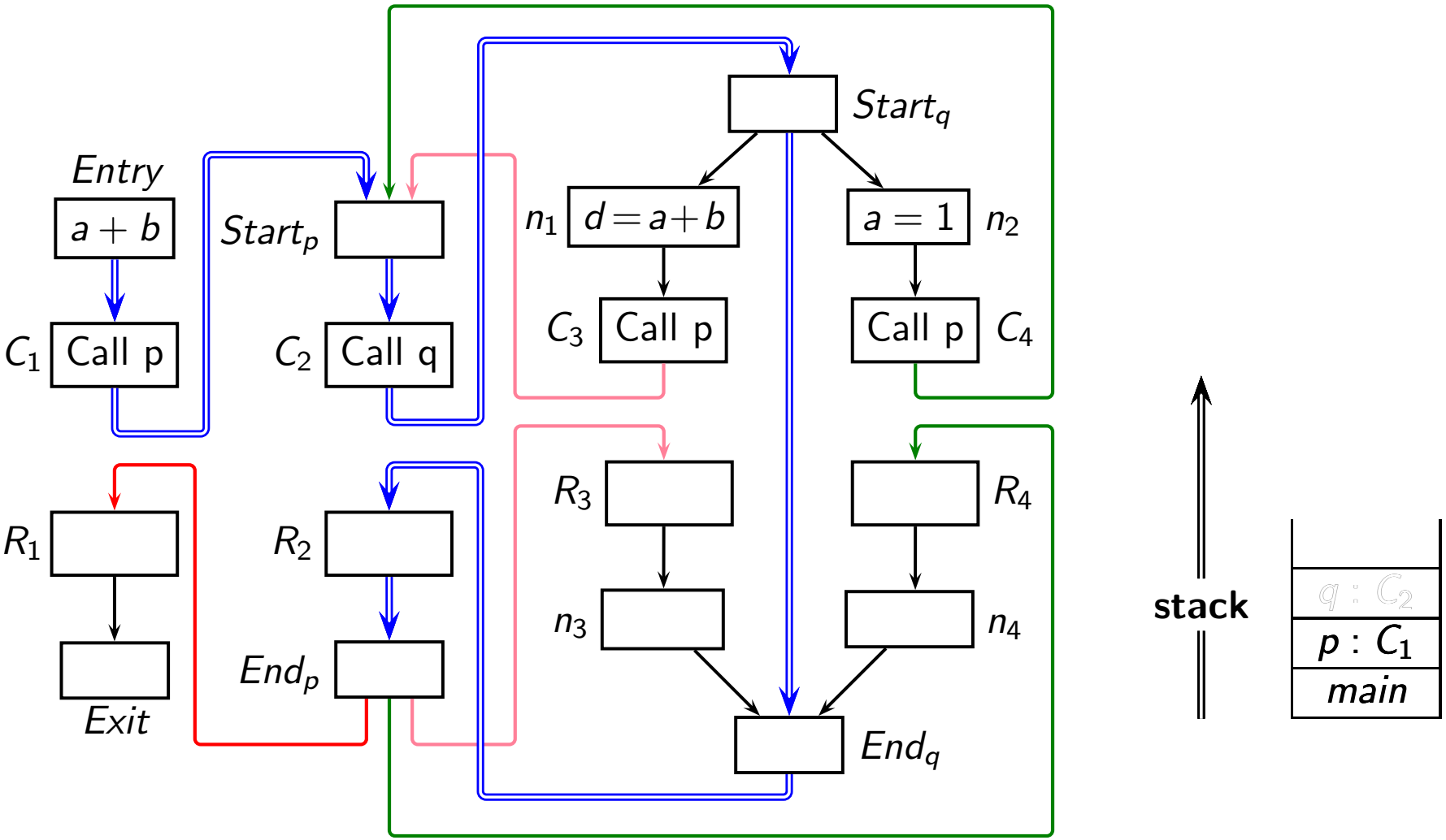
Defining Interprocedural Context for Static Analysis



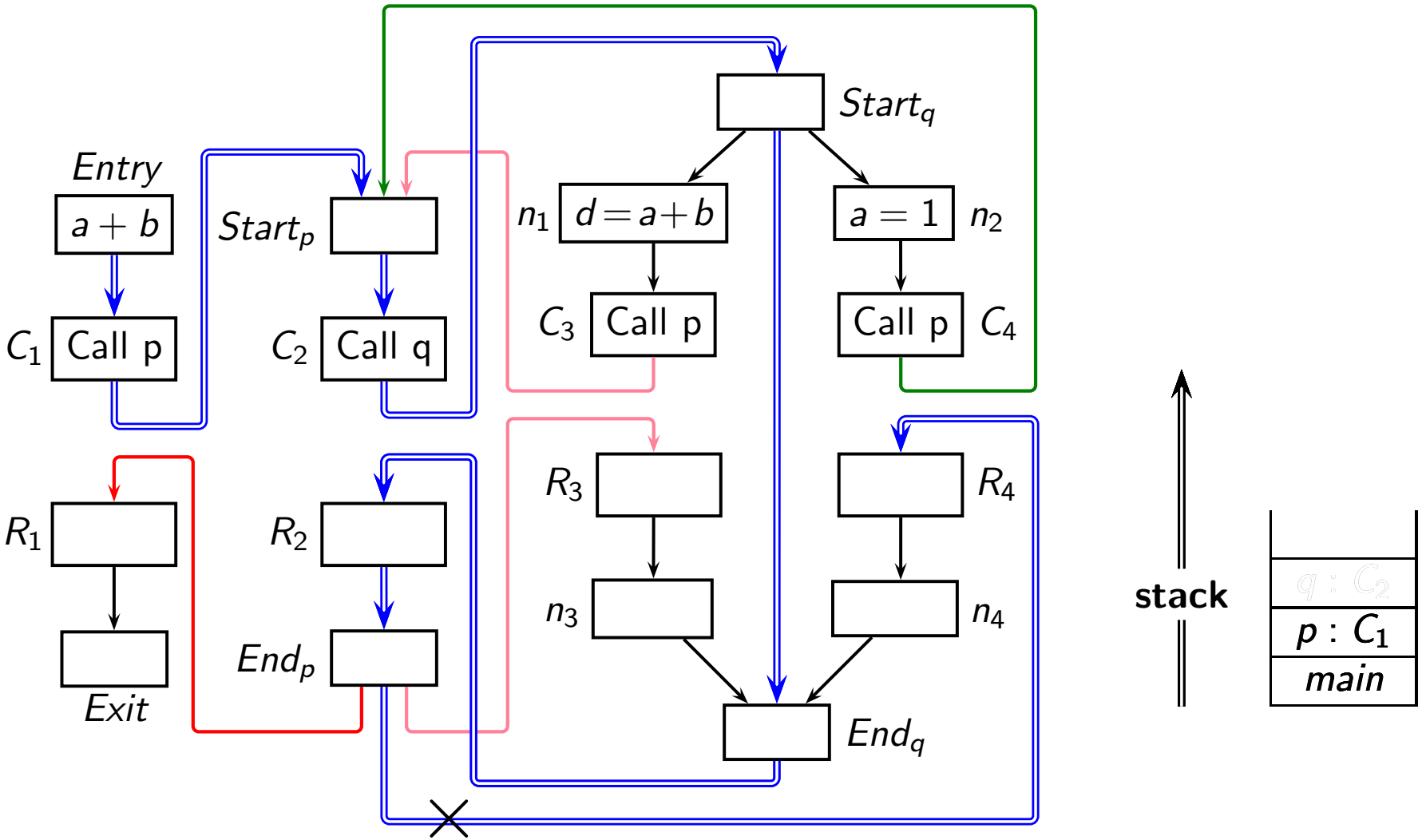
Defining Interprocedural Context for Static Analysis



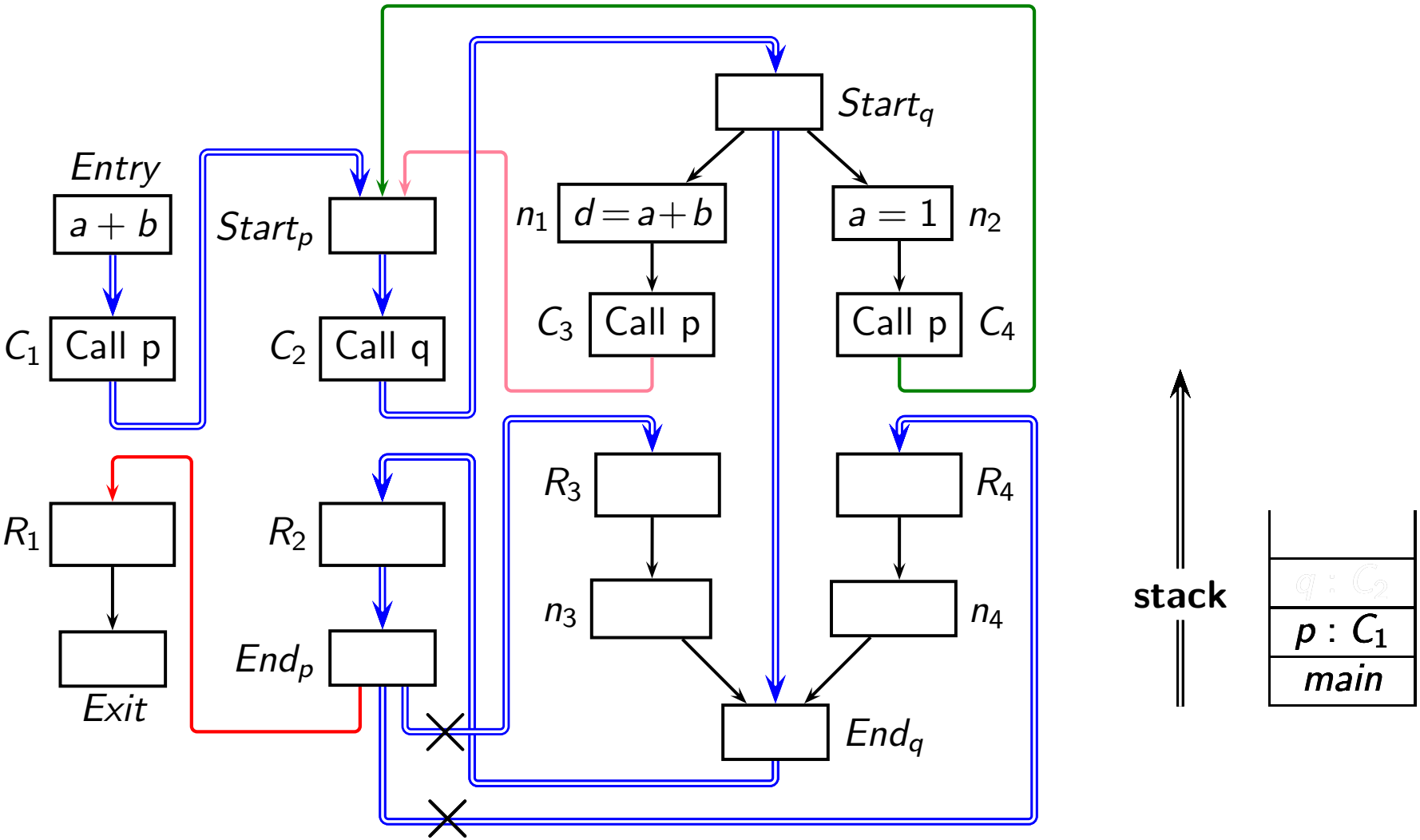
Defining Interprocedural Context for Static Analysis



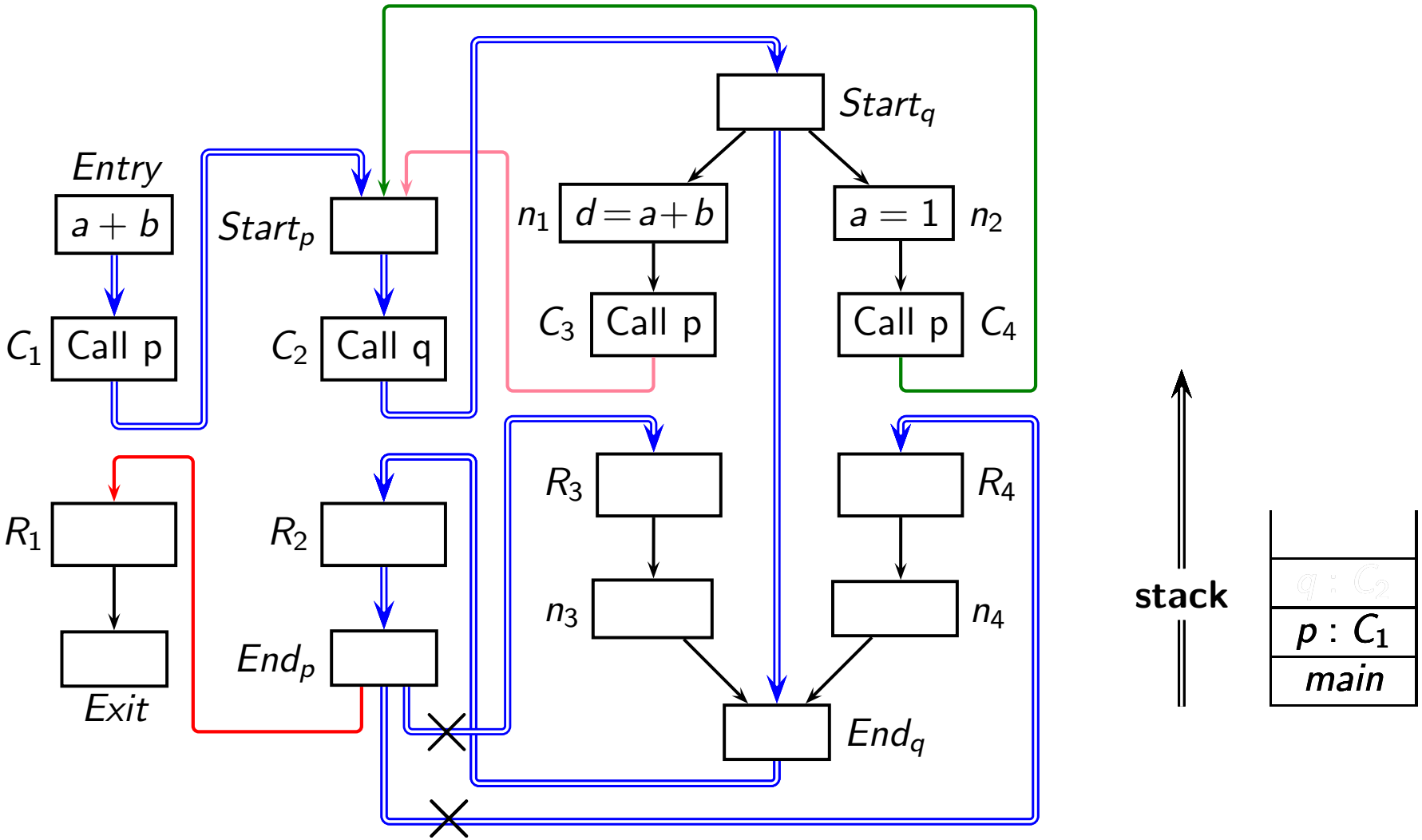
Defining Interprocedural Context for Static Analysis



Defining Interprocedural Context for Static Analysis



Defining Interprocedural Context for Static Analysis



Context is defined by stack snapshot ⇒ Unbounded number of contexts

Safety, Precision, and Efficiency of Data Flow Analysis

- Data flow analysis uses static representation of programs to compute summary information along paths

Safety, Precision, and Efficiency of Data Flow Analysis

A path which represents legal control flow

- Data flow analysis uses static representation of programs to compute summary information along paths
- *Ensuring Safety*. All valid paths must be covered

Safety, Precision, and Efficiency of Data Flow Analysis

A path which represents legal control flow

- Data flow analysis uses static representation of programs to compute summary information along paths
- *Ensuring Safety*. All valid paths must be covered
- *Ensuring Precision*. Only valid paths should be covered.

Safety, Precision, and Efficiency of Data Flow Analysis

A path which represents legal control flow

- Data flow analysis uses static representation of programs to compute summary information along paths
- *Ensuring Safety*. All valid paths must be covered
- *Ensuring Precision*. Only valid paths should be covered.
- *Ensuring Efficiency*. Only relevant valid paths should be covered.

A path which yields information that affects the summary information.

Flow and Context Sensitivity

- Flow sensitive analysis:
  Considers intraprocedurally valid paths
- Context sensitive analysis:
  Considers interprocedurally valid paths

Flow and Context Sensitivity

- Flow sensitive analysis:
  Considers intraprocedurally valid paths
- Context sensitive analysis:
  Considers interprocedurally valid paths

For maximum statically attainable precision, analysis must be both flow and context sensitive.

Part 3

Classical Call Strings Approach

Call Strings Approach [Sharir and Pnueli 1981]

Most general, flow and context sensitive method

- Remember call history
  Information should be propagated *back* to the correct point

- Call string at a program point:
  - Sequence of *unfinished calls* reaching that point
  - Starting from the *Entry*

  A snap-shot of call stack in terms of call sites

Call Strings Approach [Sharir and Pnueli 1981]

Most general, flow and context sensitive method

- Remember call history
  Information should be propagated *back* to the correct point

- Call string at a program point:
  - Sequence of *unfinished calls* reaching that point
  - Starting from the *Entry*

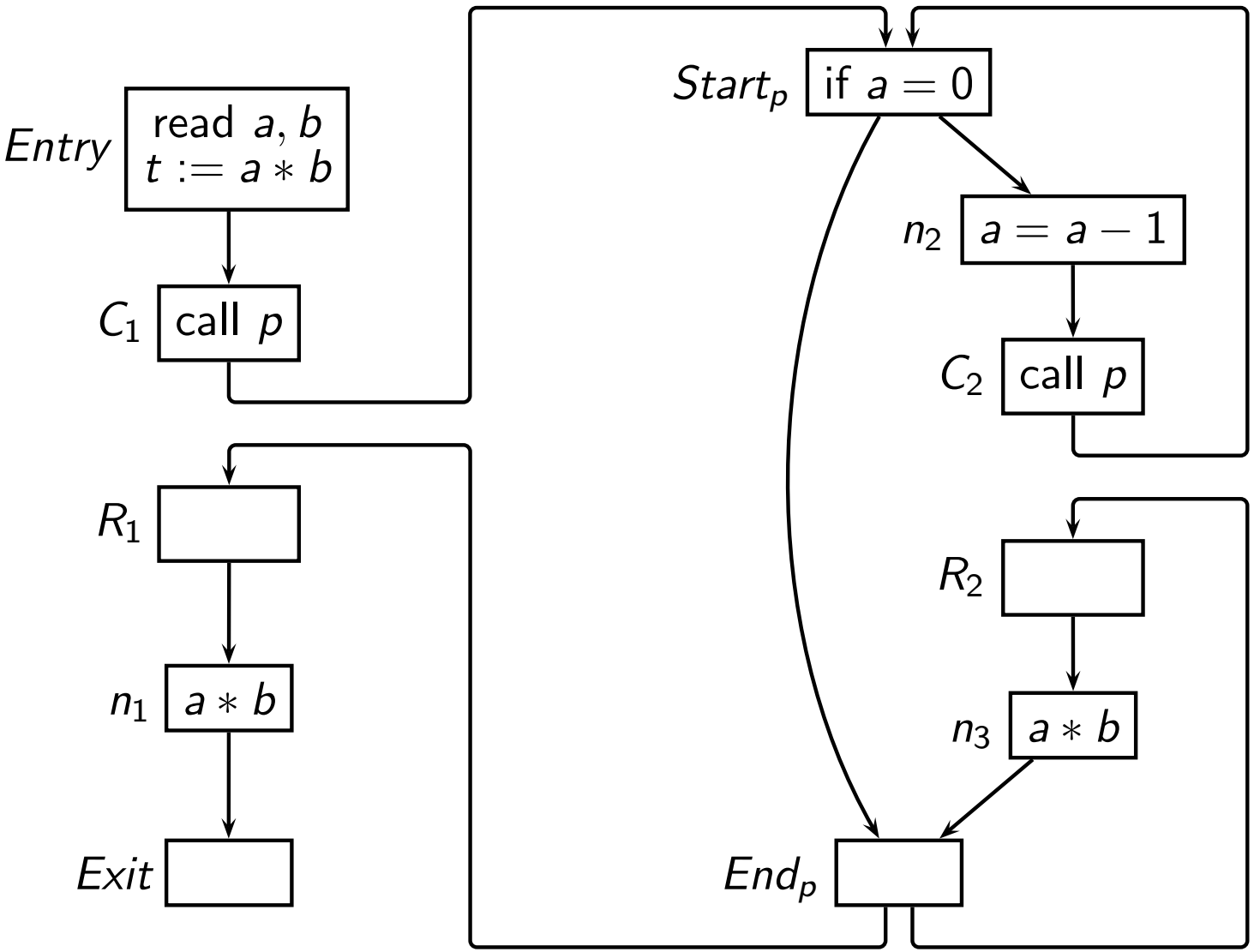
  A snap-shot of call stack in terms of call sites

Efficiency ??

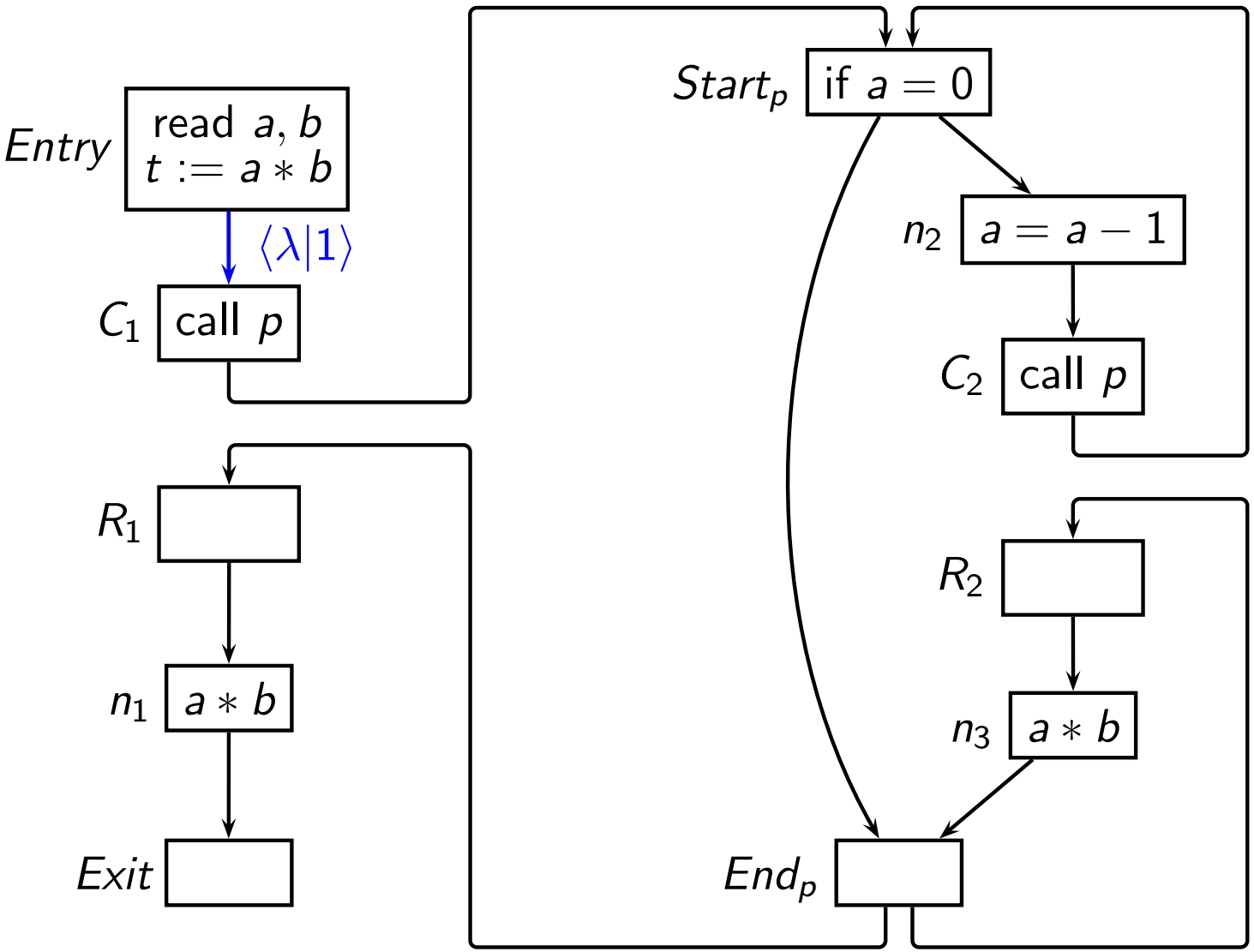
Interprocedural Data Flow Analysis Using Call Strings

- Data flow information at a program point
  $\langle$Call String : σ, Data Flow Value : $d\rangle$
- Flow functions
  - Intraprocedural edges: Manipulate data flow value d
  - Interprocedural edges: Manipulate call string σ
    - Call edge $C_i \longrightarrow s_p$ (i.e. call site c_i calling procedure p)
      append c_i to every σ reaching c_i.
    - Return edge $e_p \longrightarrow R_i$ (i.e. p returning the control to call site c_i)
      if the last call site is c_i, remove it and propagate the data flow value

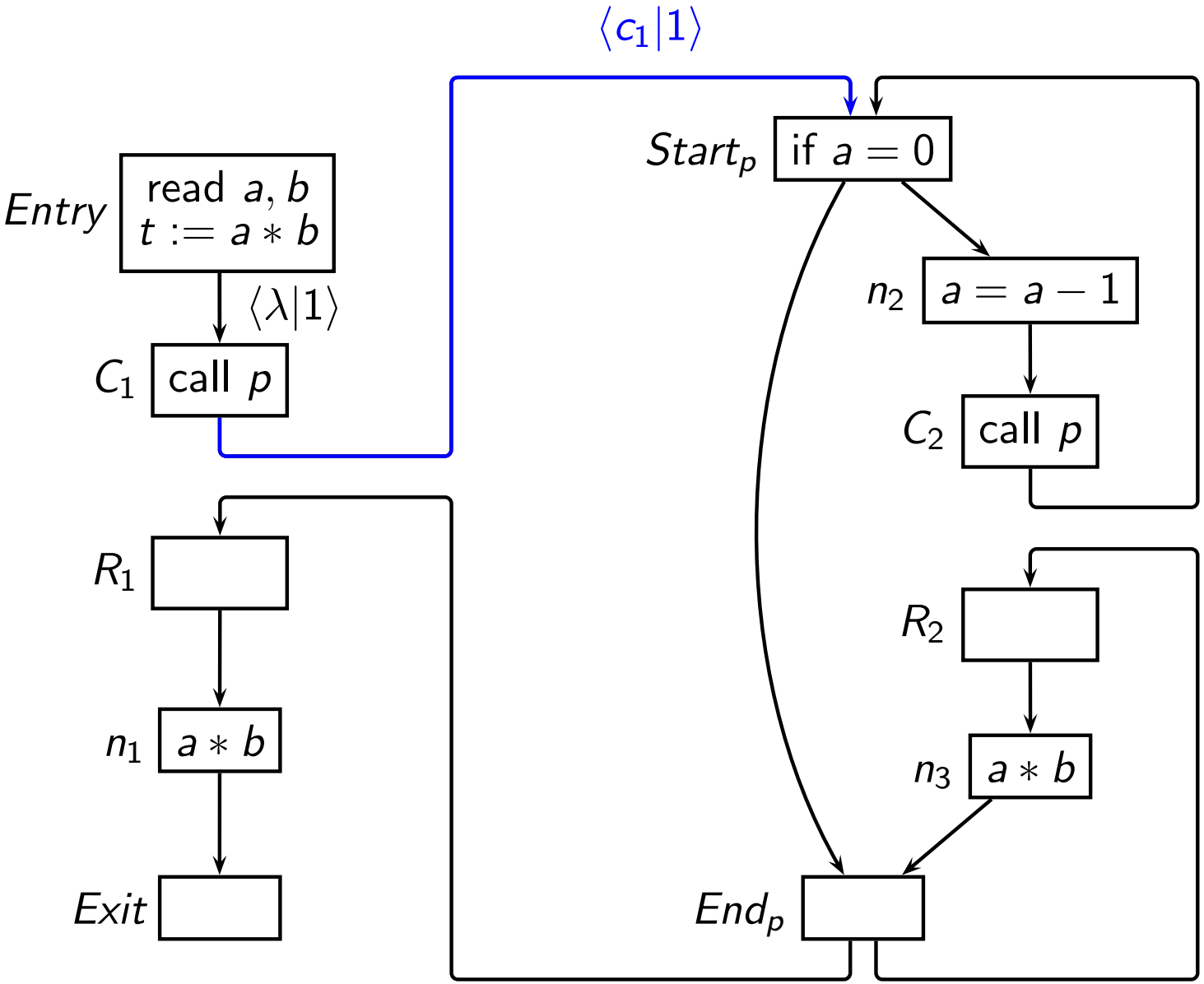
Available Expressions Analysis Using Call Strings Approach



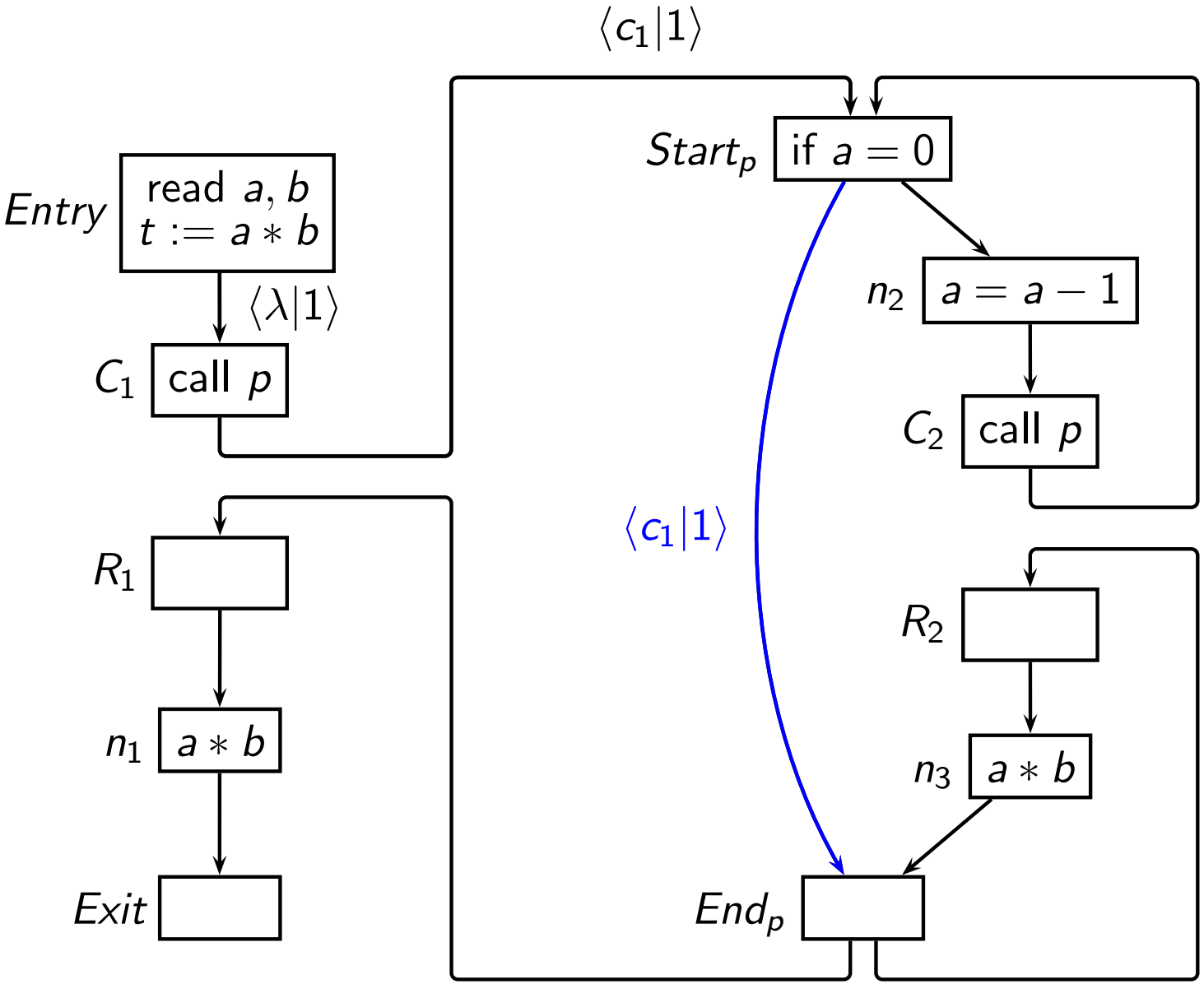
Available Expressions Analysis Using Call Strings Approach



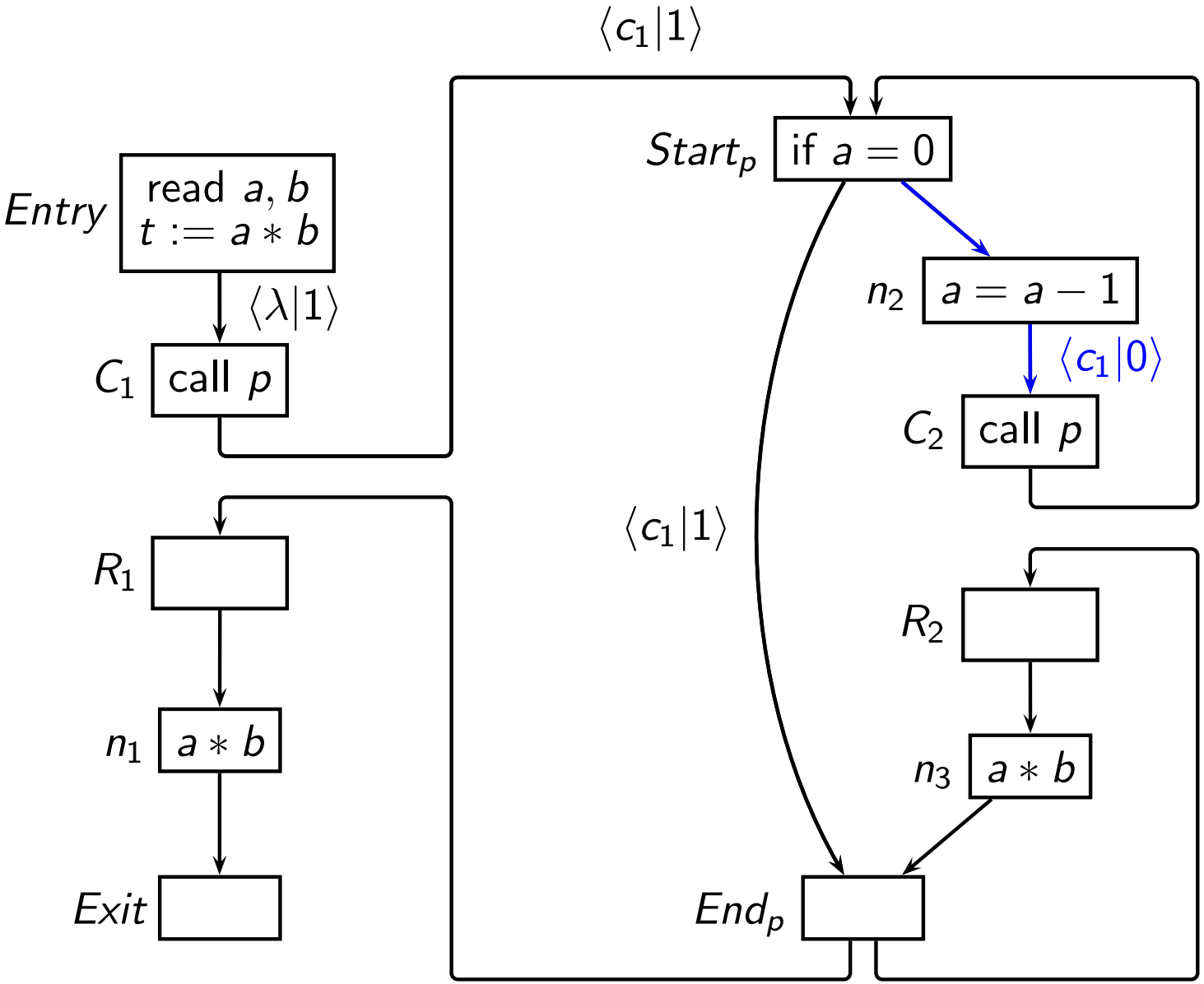
Available Expressions Analysis Using Call Strings Approach



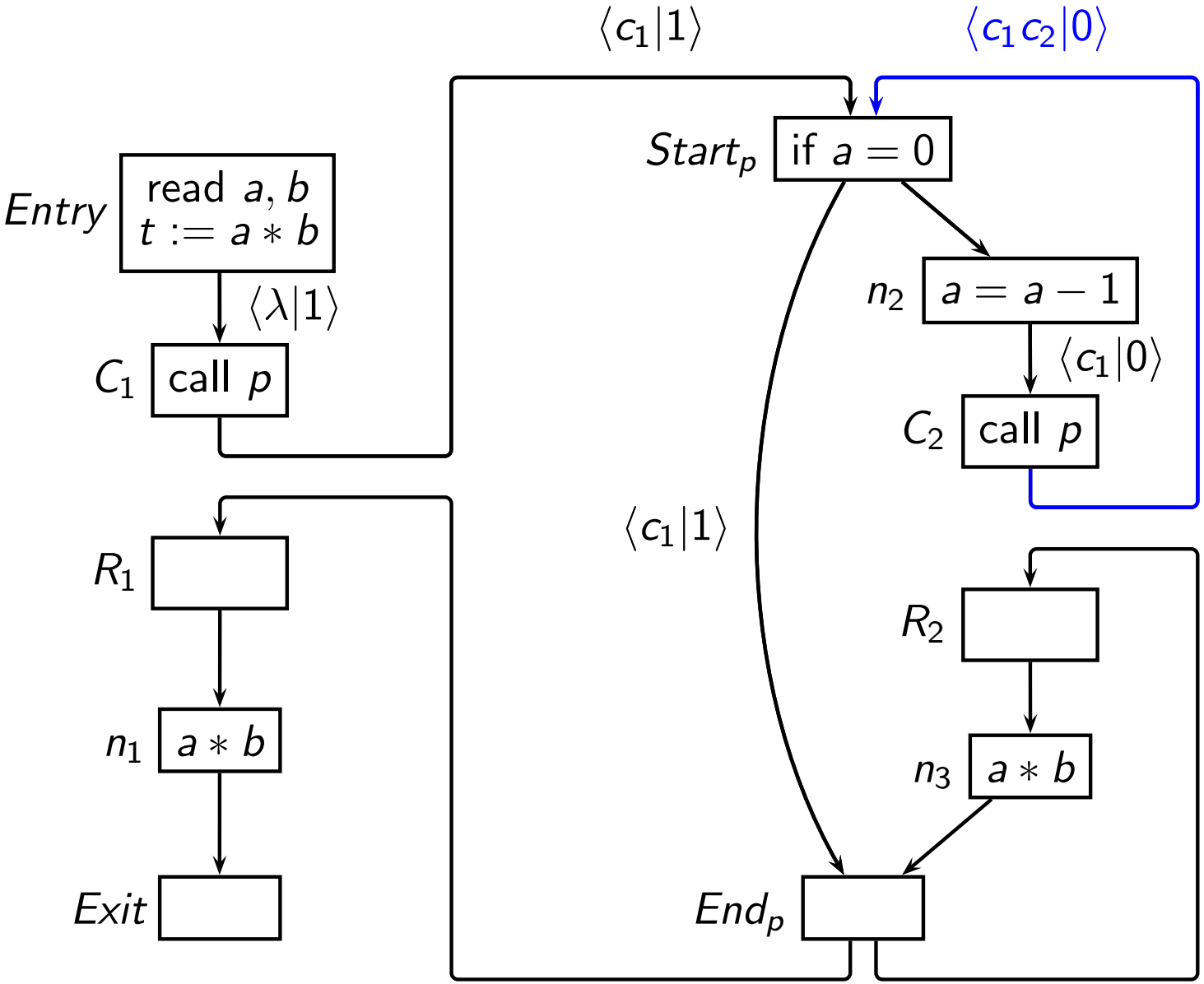
Available Expressions Analysis Using Call Strings Approach



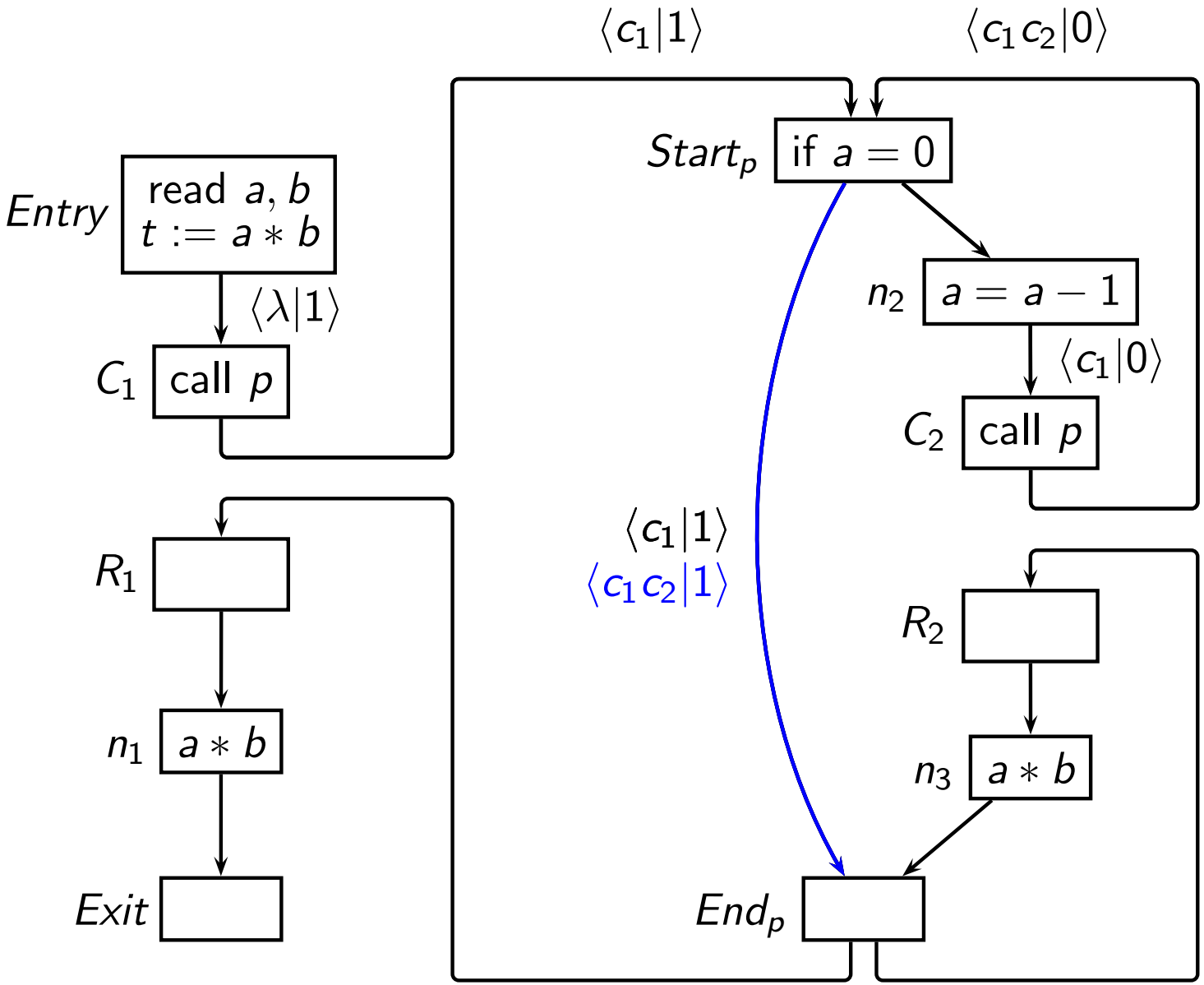
Available Expressions Analysis Using Call Strings Approach



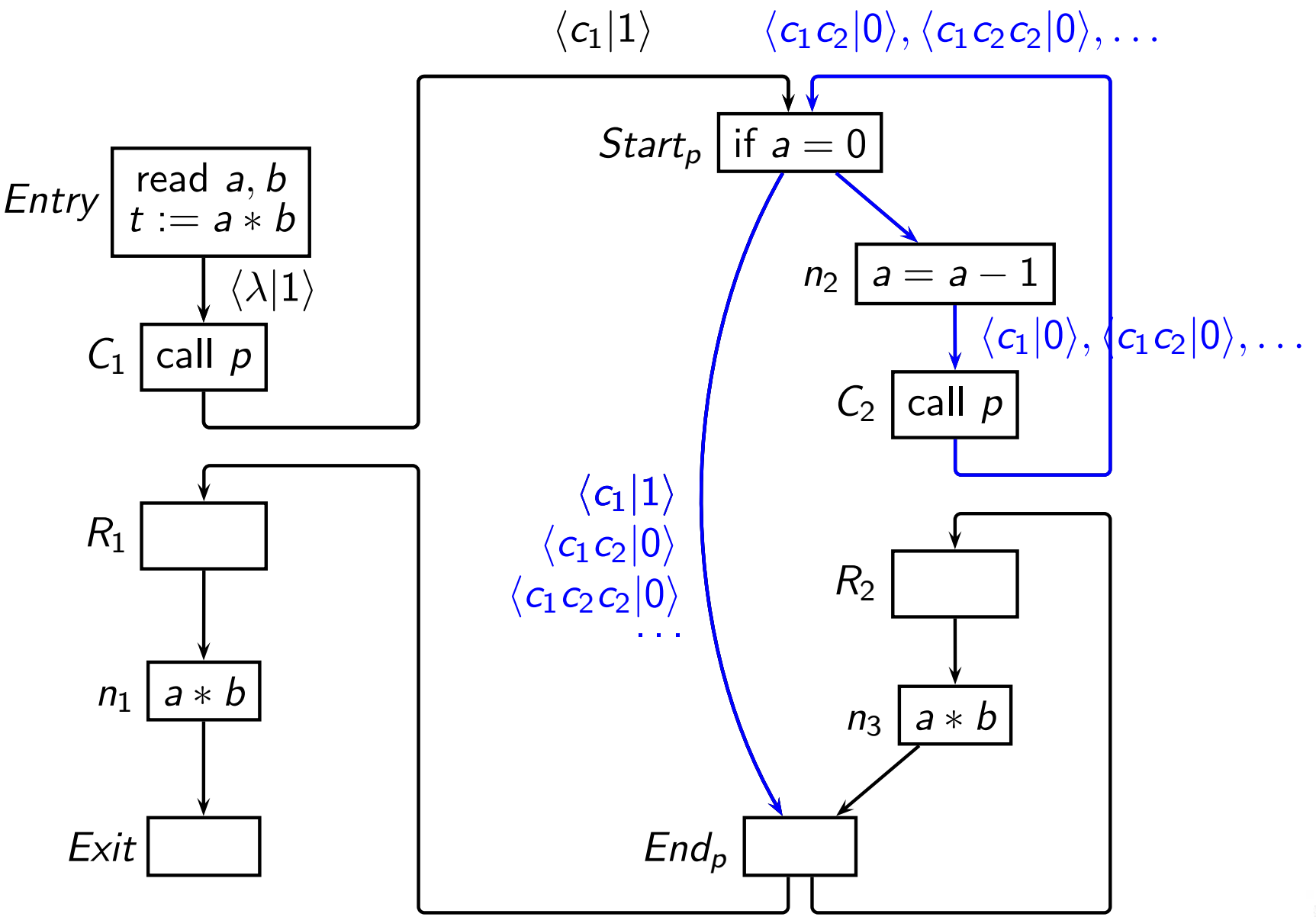
Available Expressions Analysis Using Call Strings Approach



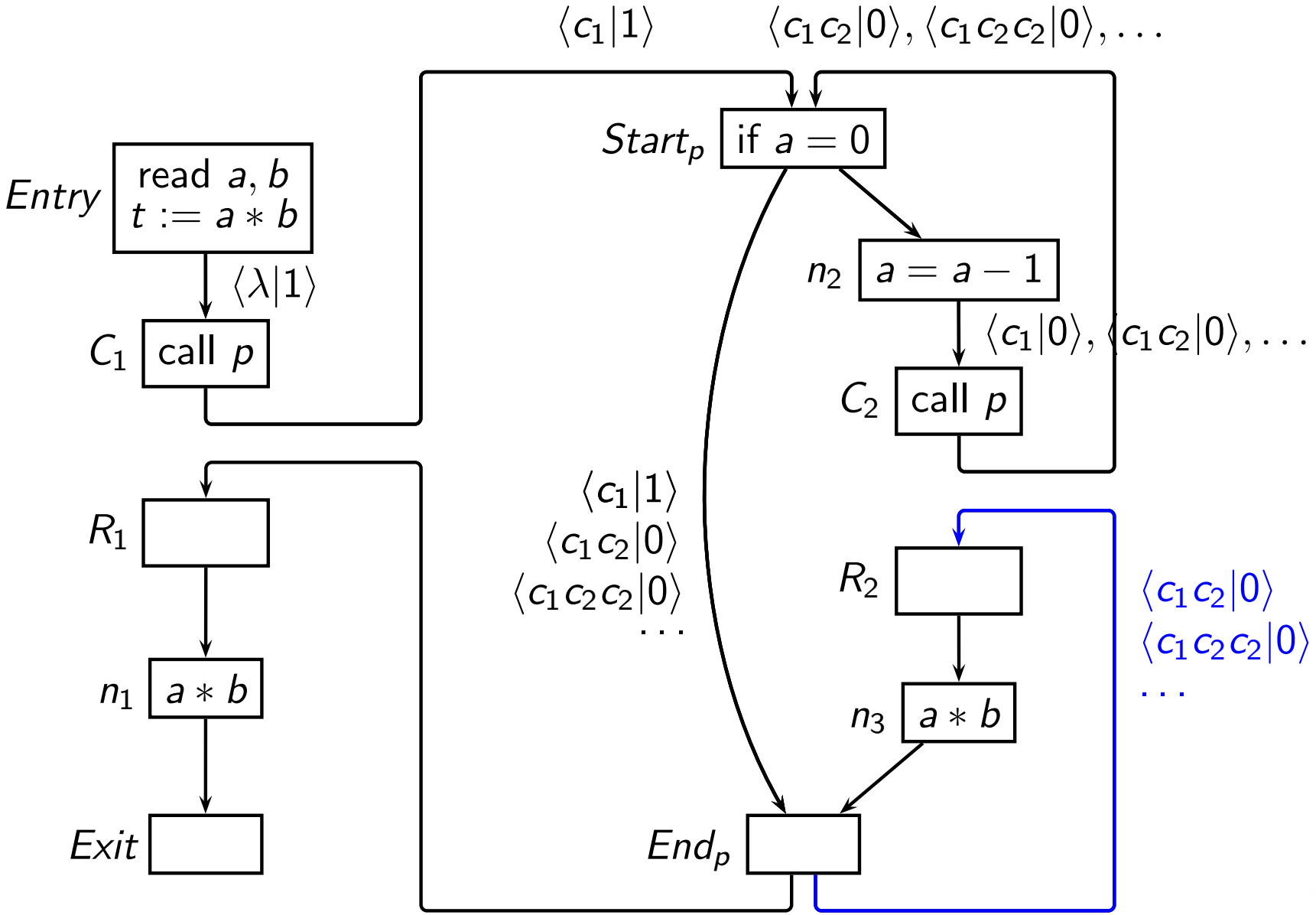
Available Expressions Analysis Using Call Strings Approach



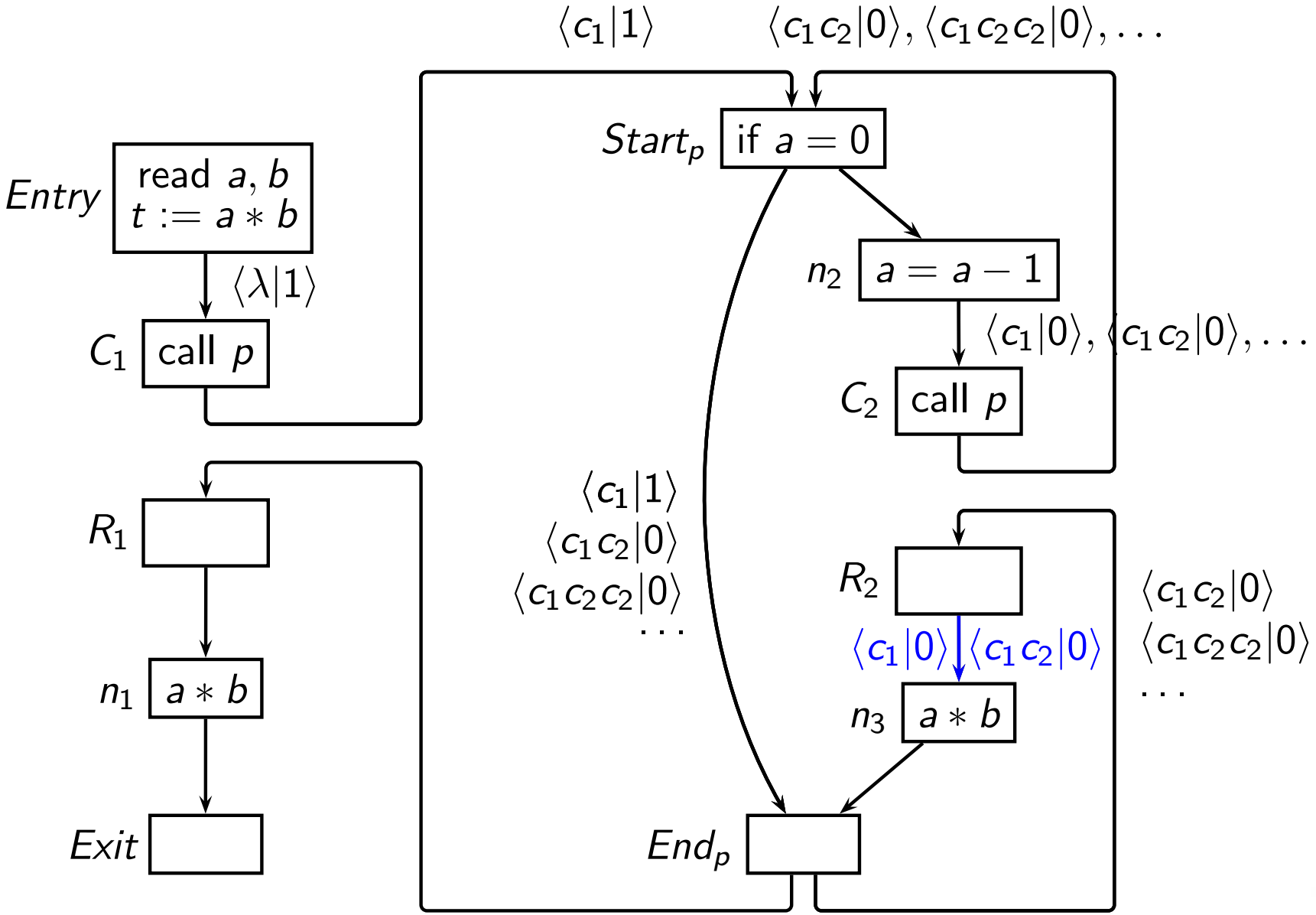
Available Expressions Analysis Using Call Strings Approach



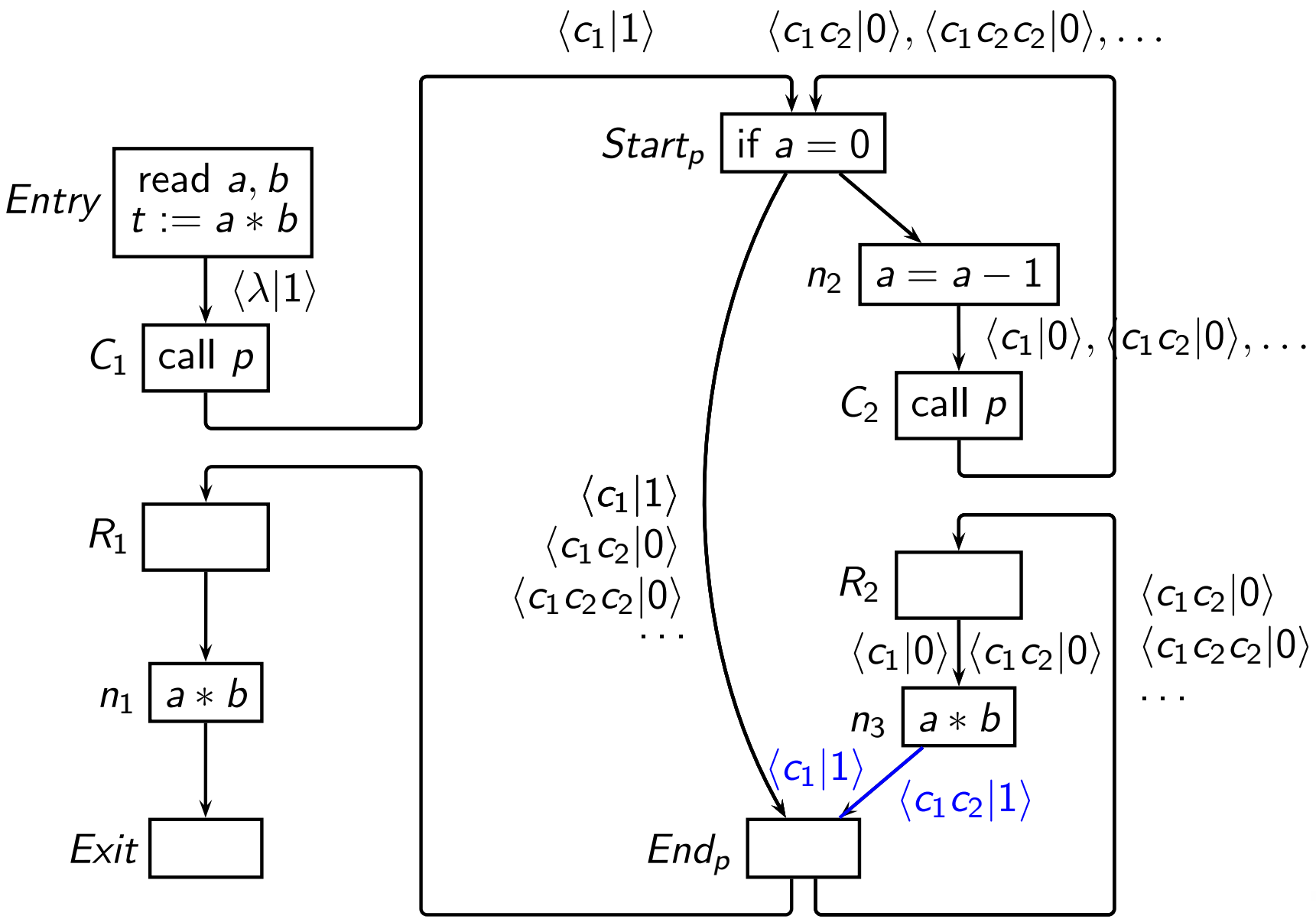
Available Expressions Analysis Using Call Strings Approach



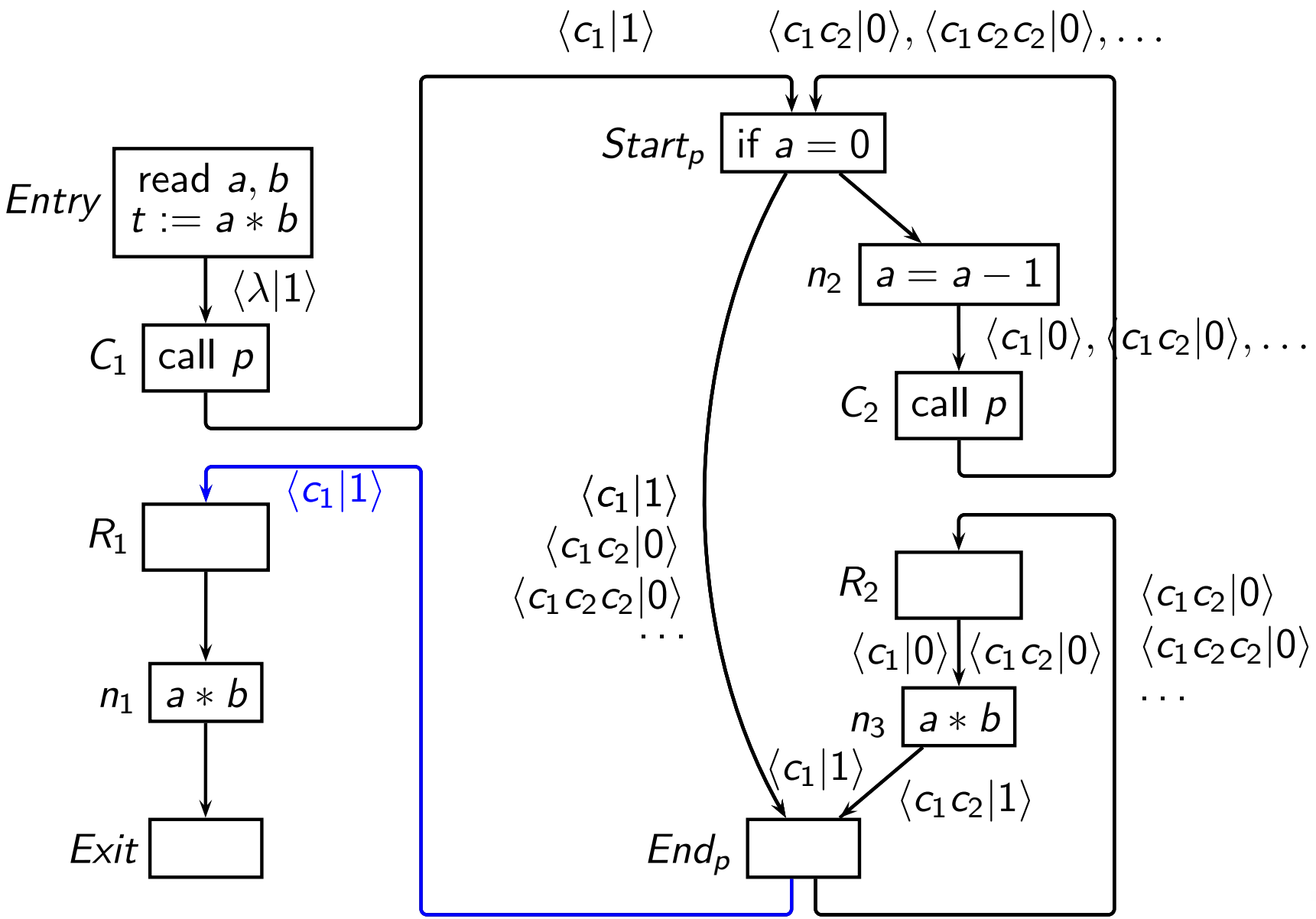
Available Expressions Analysis Using Call Strings Approach



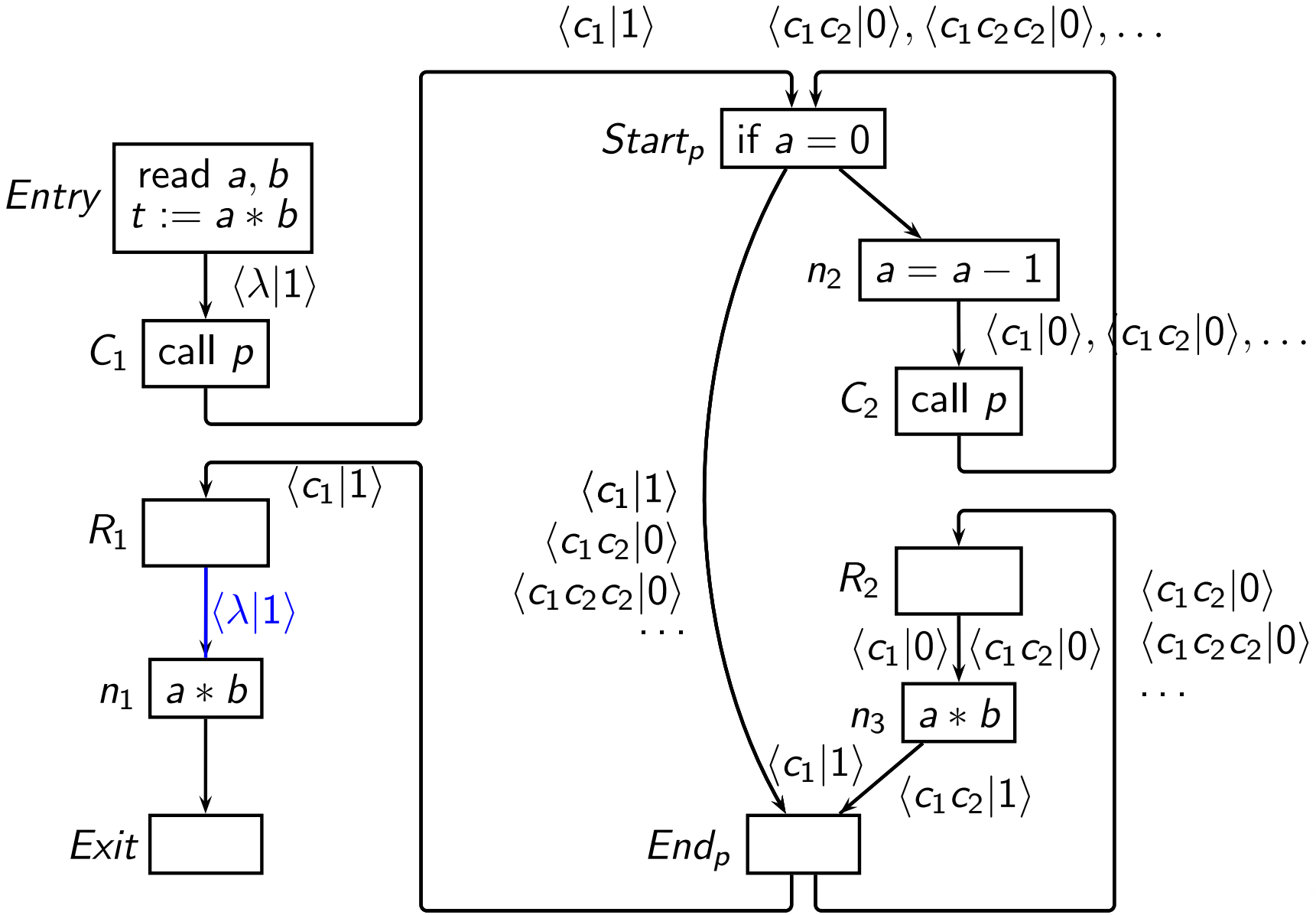
Available Expressions Analysis Using Call Strings Approach



Available Expressions Analysis Using Call Strings Approach



Available Expressions Analysis Using Call Strings Approach



Terminating Call String Construction

- For non-recursive programs: Number of call strings is finite
- For recursive programs: Number of call strings could be infinite

  Fortunately, the problem is decidable for finite lattices.
  - All call strings with the length
    - $K \cdot (|L| + 1)^2$ for general bounded frameworks
      (L is the overall lattice of data flow values)
    - $K \cdot (|\widehat{L}| + 1)^2$ for separable bounded frameworks
      (L is the component lattice for an entity)
    - $K \cdot 3$ for bit vector frameworks

    must be constructed

  $\Rightarrow$ Large number of long call strings

Intuition Behind the Classical Call String Length

- Consider call and return nodes C_i and R_i
- Let the value at C_i x_i and the value at R_i be z_i.

Intuition Behind the Classical Call String Length

- Consider call and return nodes C_i and R_i
- Let the value at C_i x_i and the value at R_i be z_i.
- Both x_i and z_i are in $L \cup \{undef\}$

Intuition Behind the Classical Call String Length

- Consider call and return nodes C_i and R_i
- Let the value at C_i x_i and the value at R_i be z_i.
- Both x_i and z_i are in $L \cup \{undef\}$
- The number of possibilities for x_i and z_i is $(|L| + 1)$

Intuition Behind the Classical Call String Length

- Consider call and return nodes C_i and R_i
- Let the value at C_i x_i and the value at R_i be z_i.
- Both x_i and z_i are in $L \cup \{undef\}$
- The number of possibilities for x_i and z_i is $(|L| + 1)$
- Thus c_i can appear $(|L| + 1)^2$ times in a call string

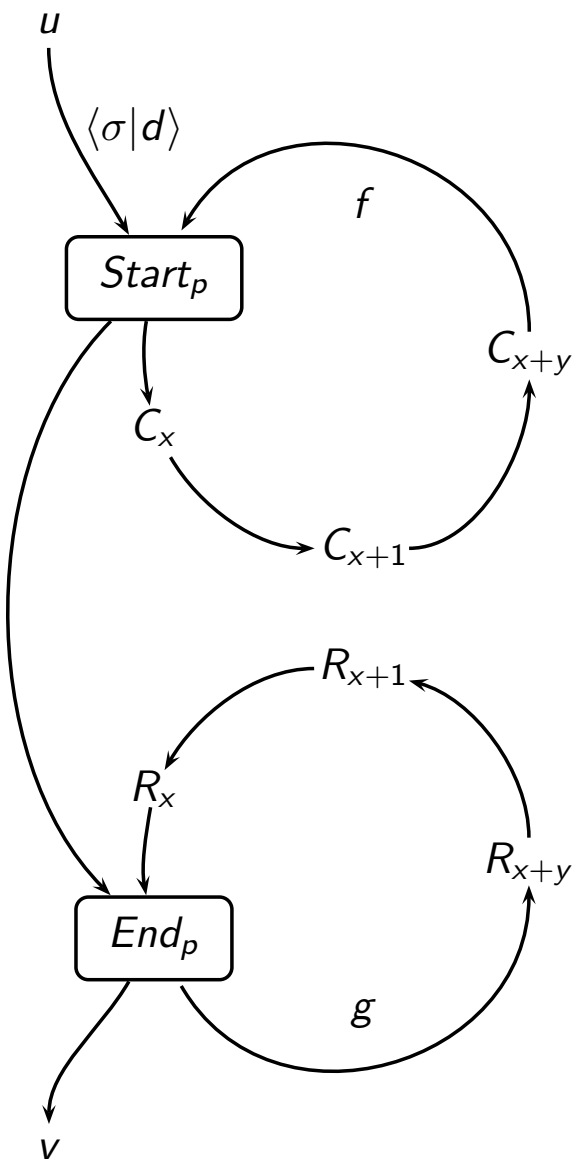
Intuition Behind the Classical Call String Length

- Consider call and return nodes C_i and R_i
- Let the value at C_i x_i and the value at R_i be z_i.
- Both x_i and z_i are in $L \cup \{undef\}$
- The number of possibilities for x_i and z_i is $(|L| + 1)$
- Thus c_i can appear $(|L| + 1)^2$ times in a call string
- Since there are K call sites, it is sufficient to construct call strings of length $K \cdot (|L| + 1)^2$

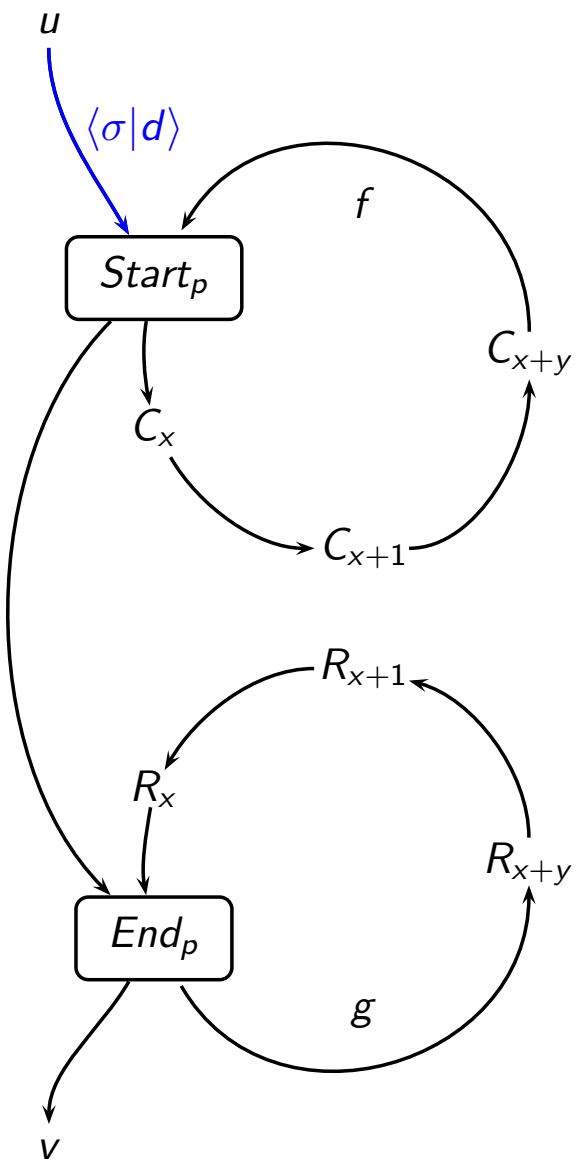
Part 4

The Proposed Variant

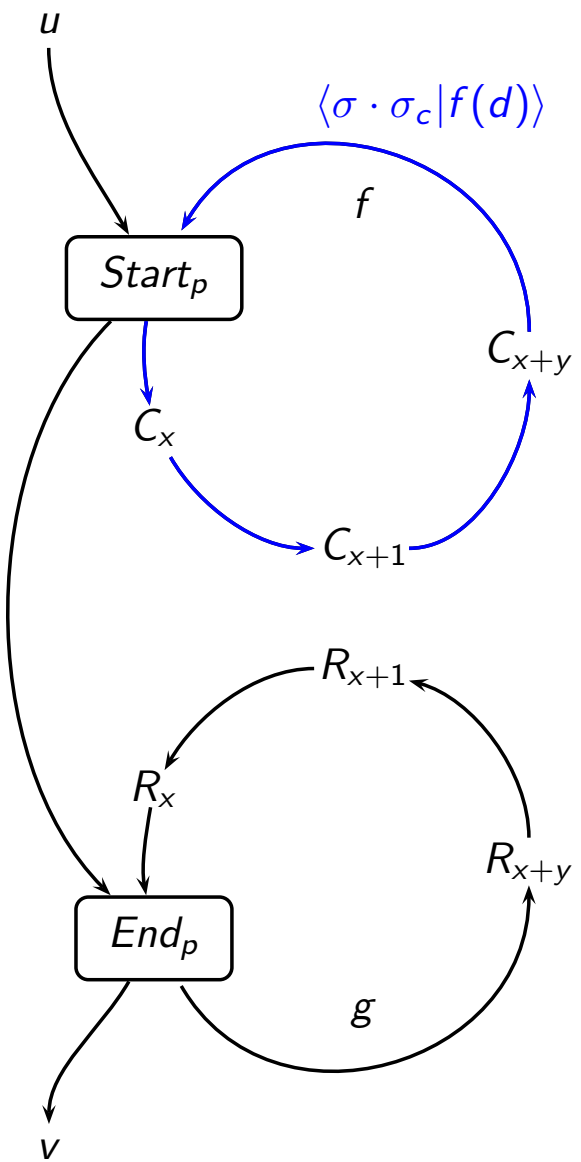
Call Strings for Recursive Contexts



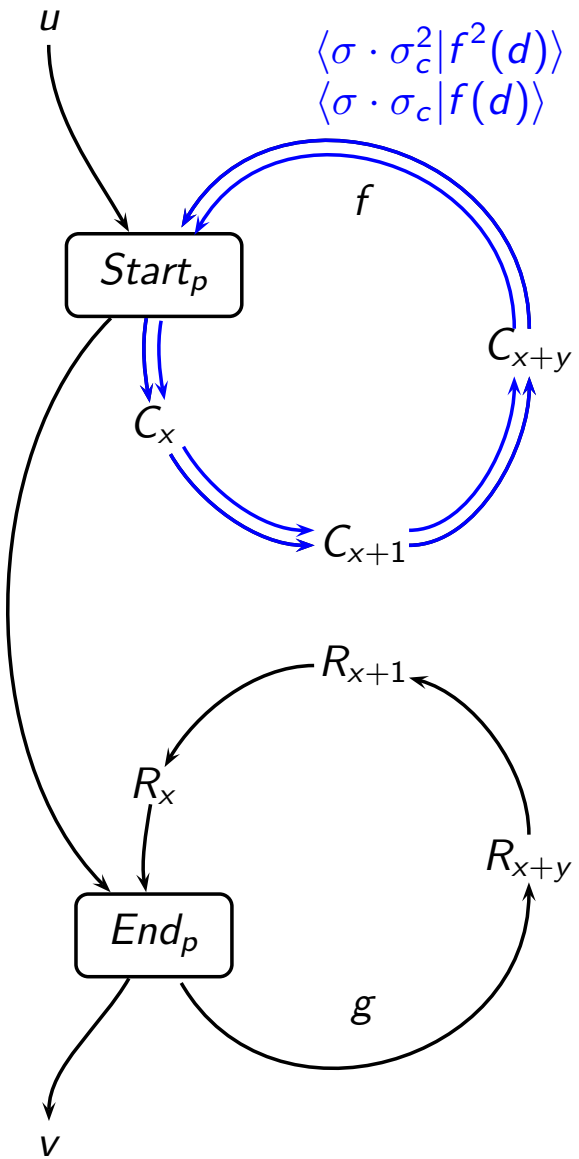
Call Strings for Recursive Contexts



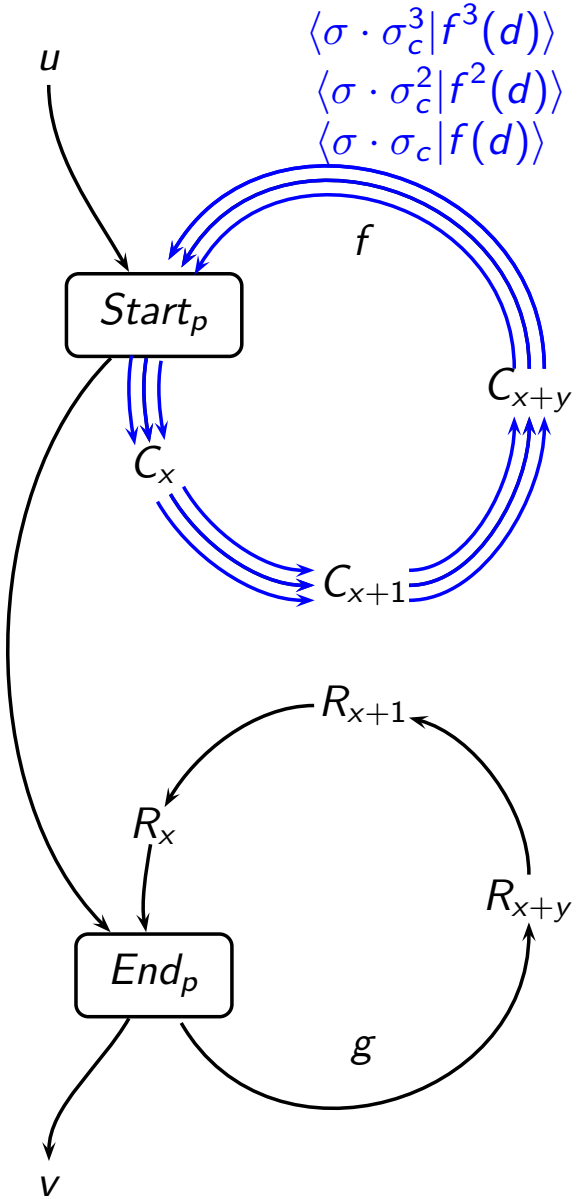
Call Strings for Recursive Contexts



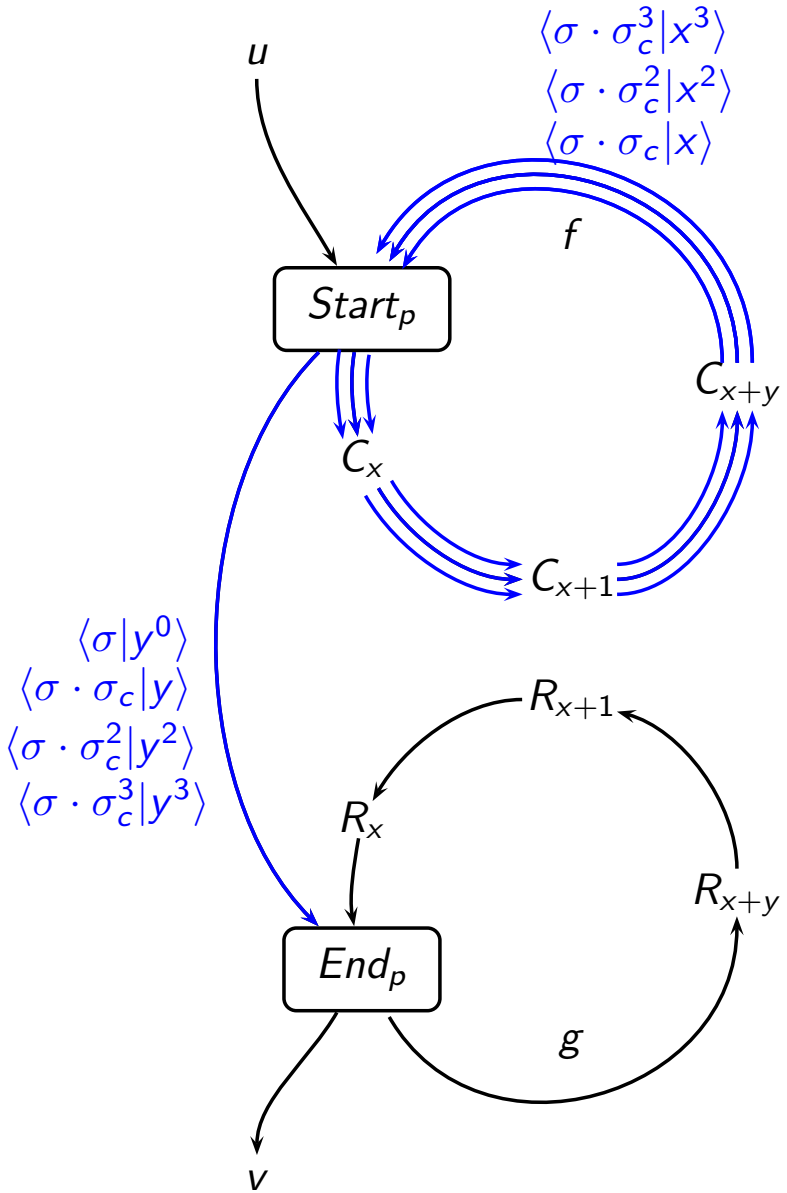
Call Strings for Recursive Contexts



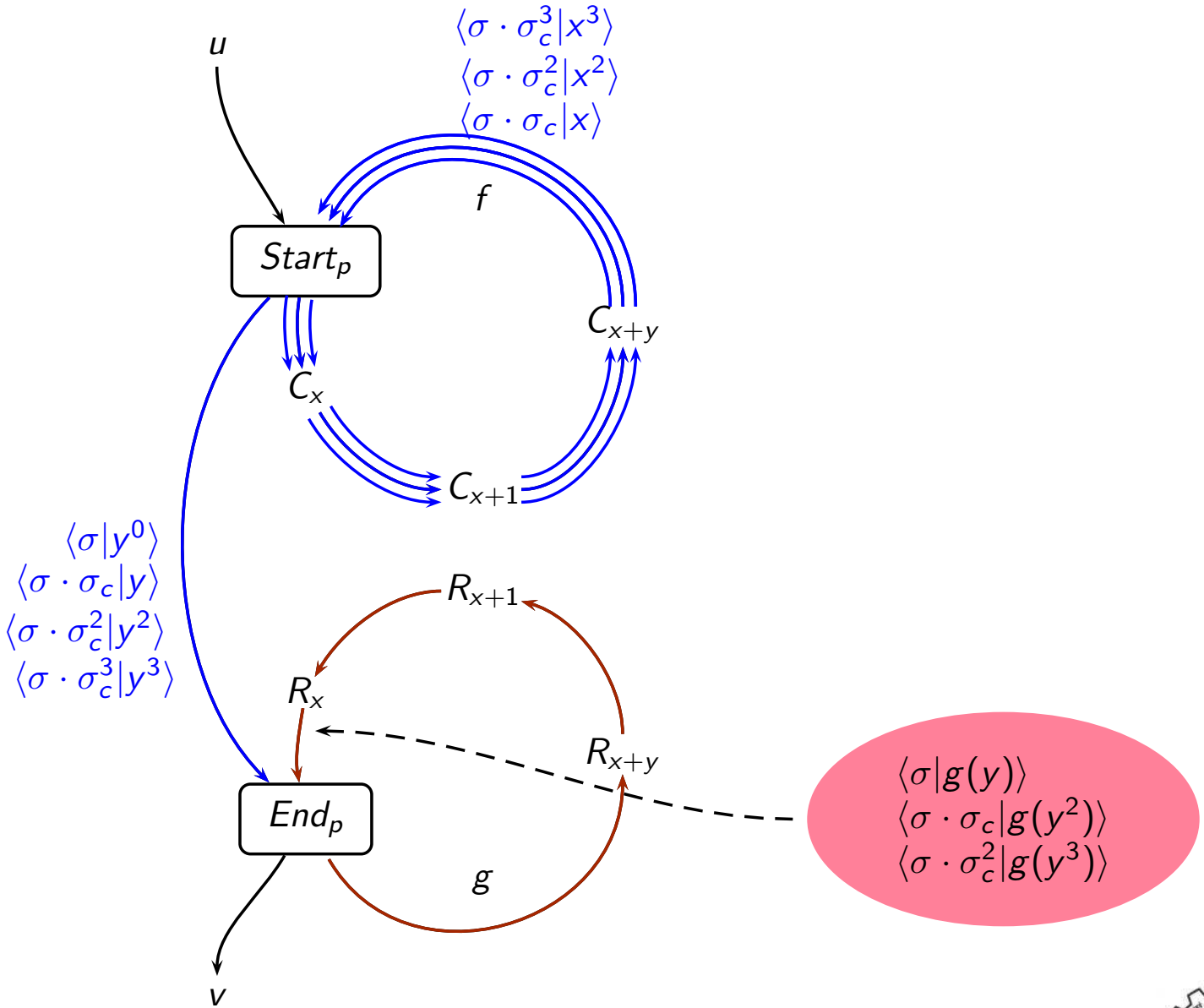
Call Strings for Recursive Contexts



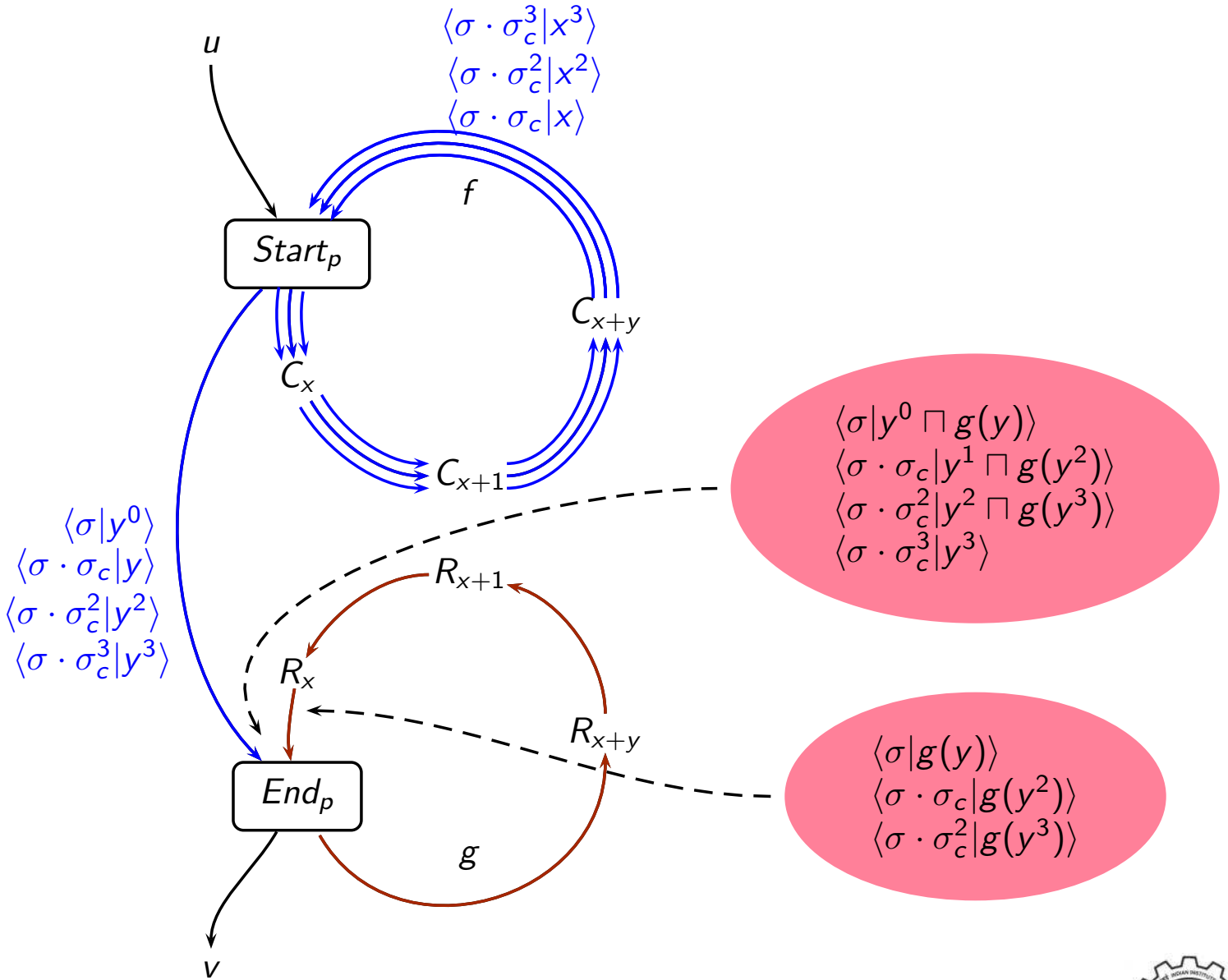
Call Strings for Recursive Contexts



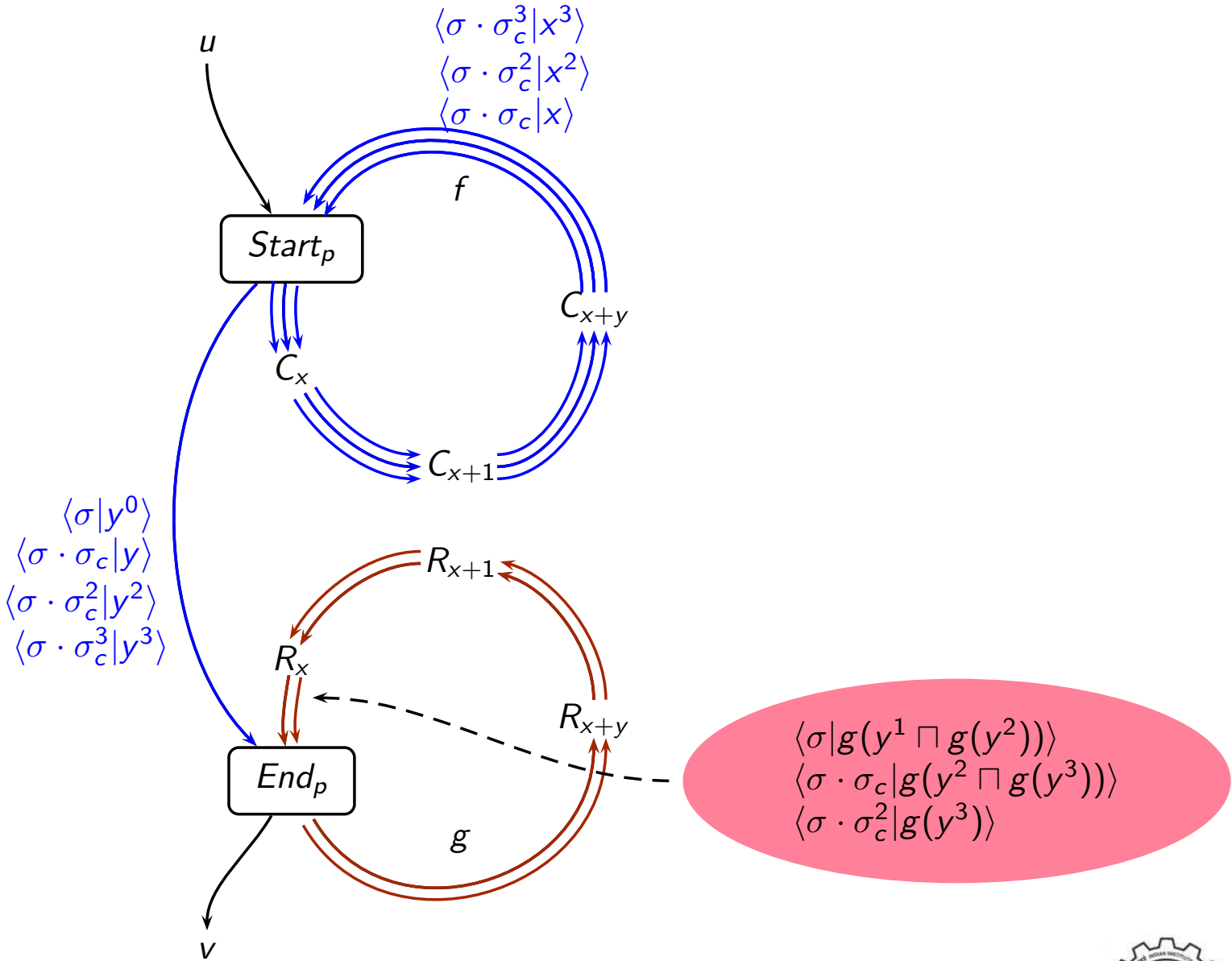
Call Strings for Recursive Contexts



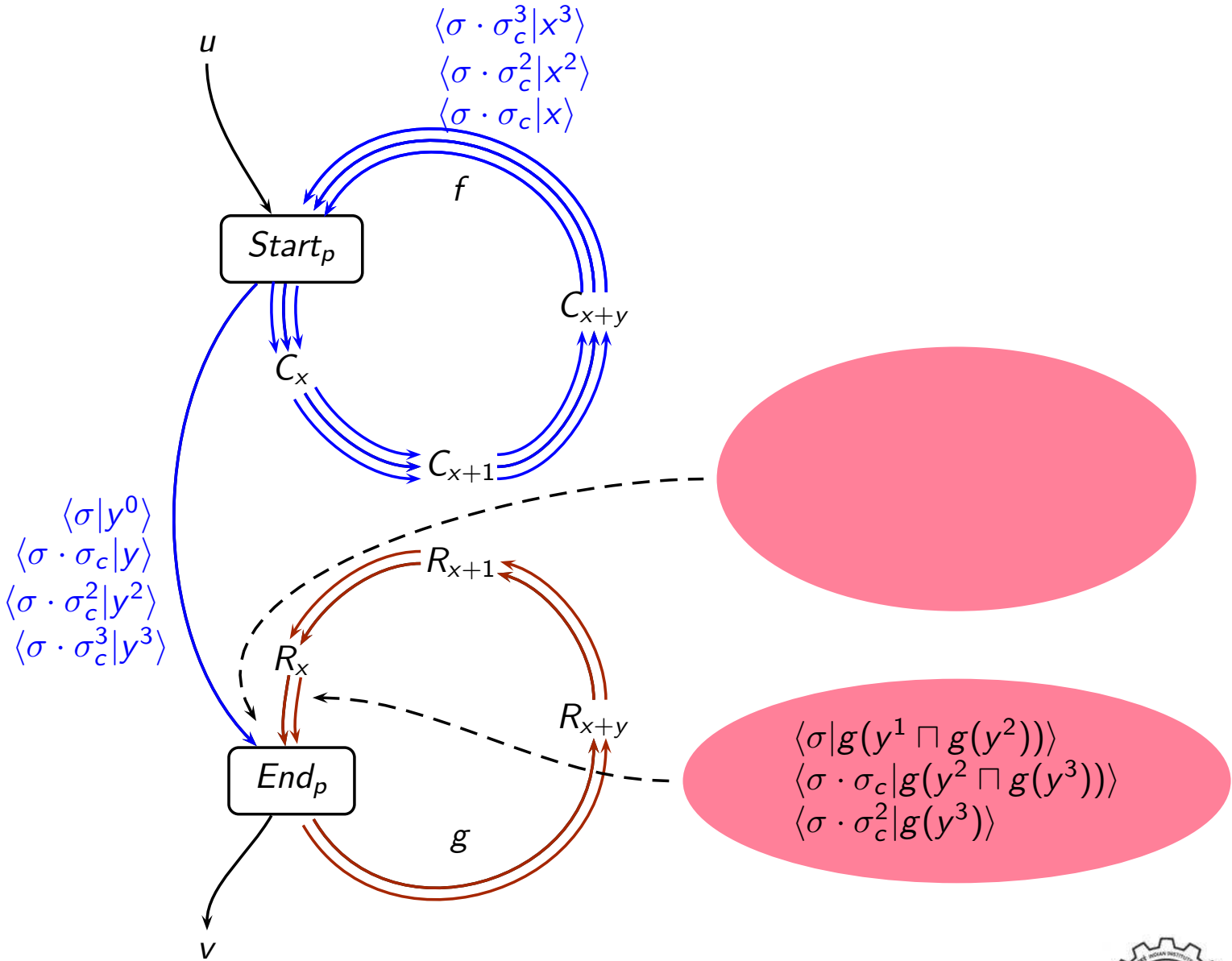
Call Strings for Recursive Contexts



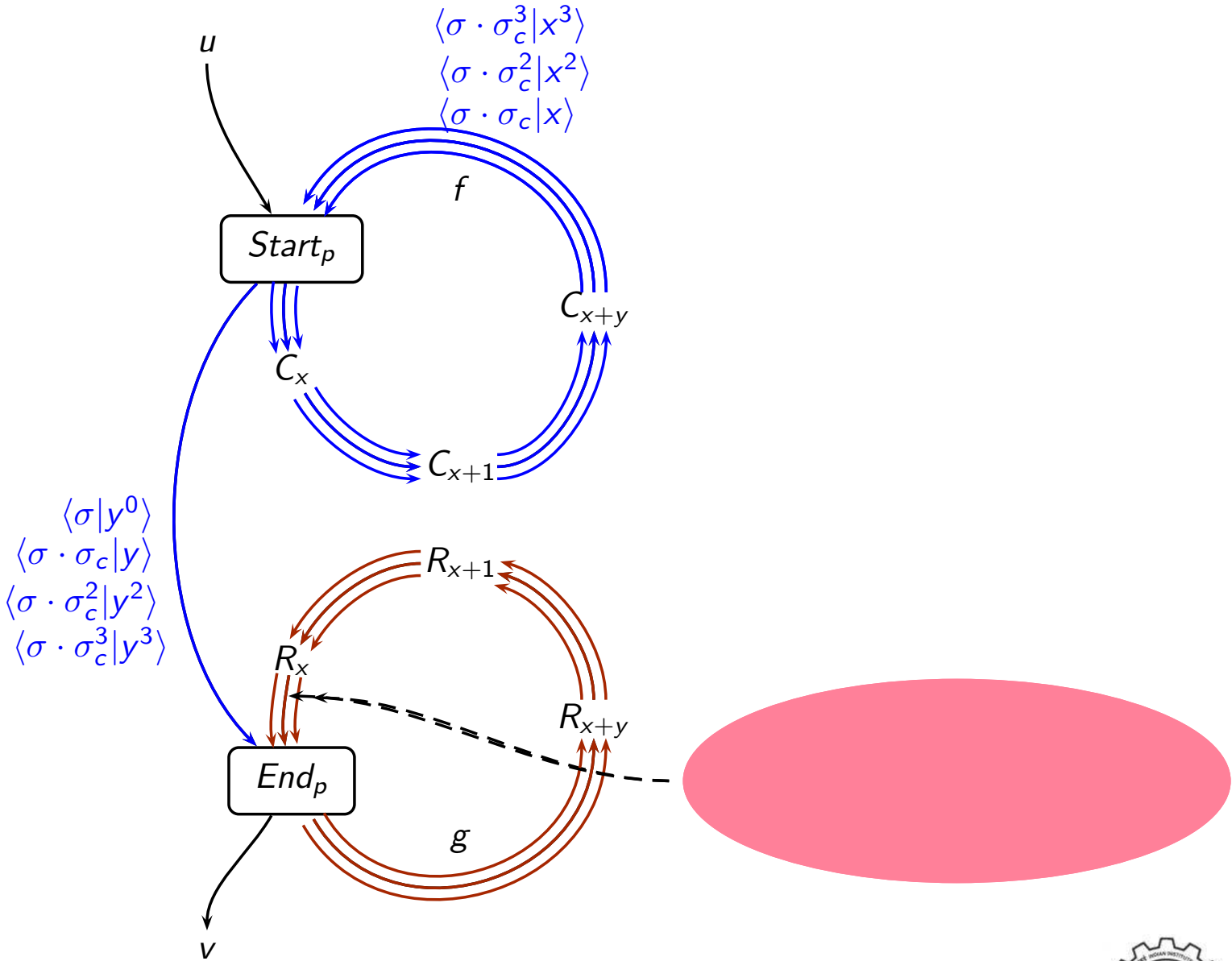
Call Strings for Recursive Contexts



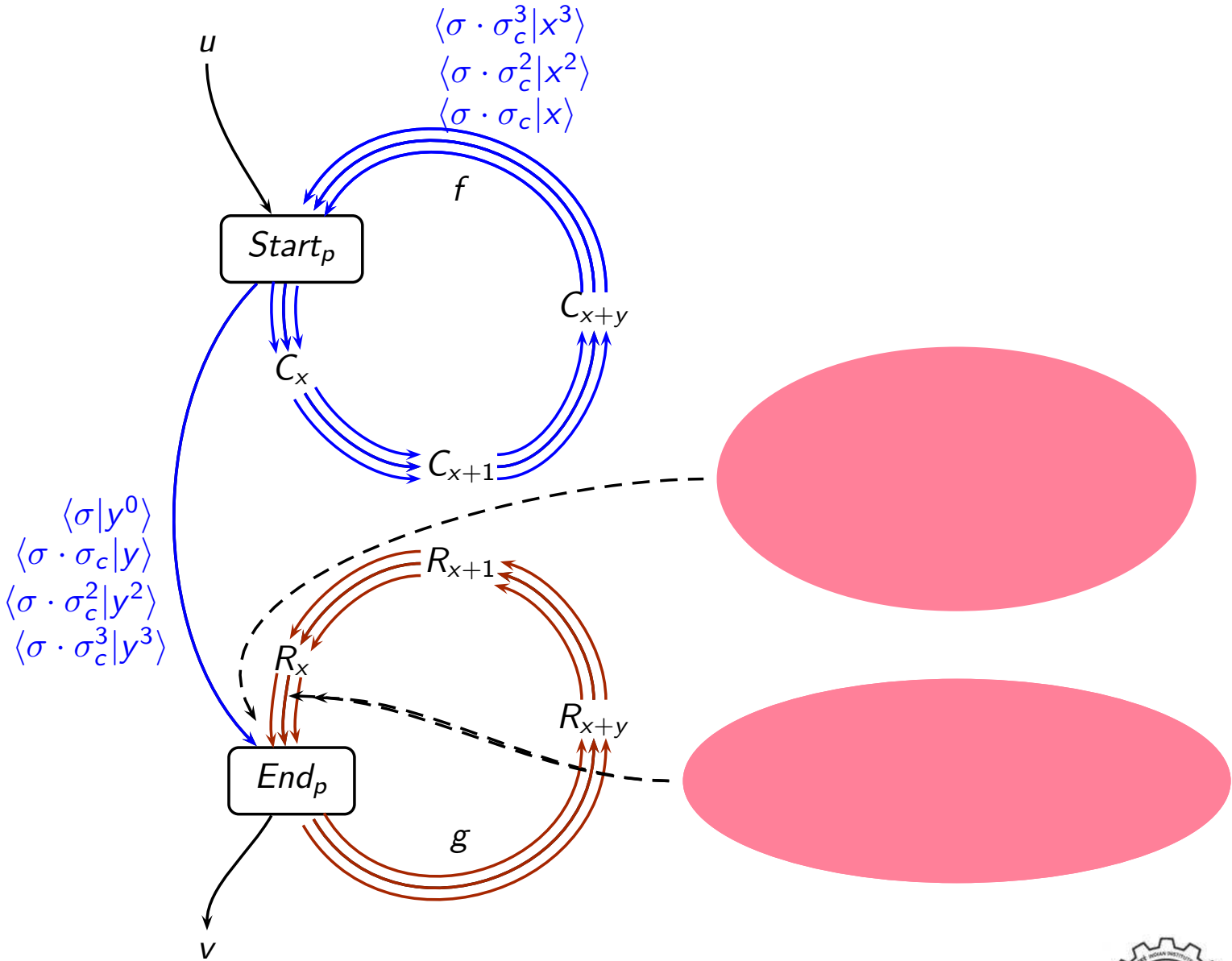
Call Strings for Recursive Contexts



Call Strings for Recursive Contexts



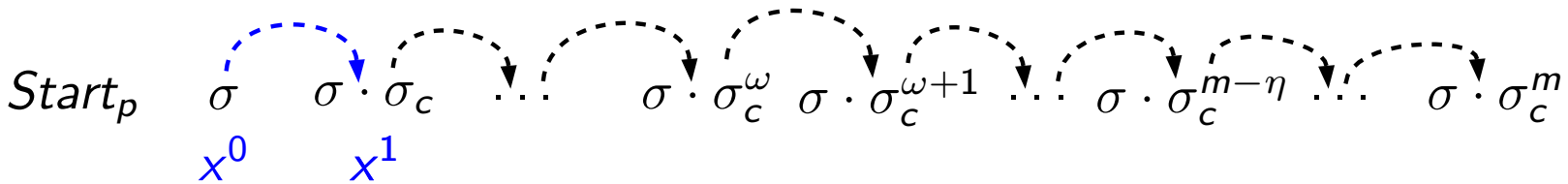
Call Strings for Recursive Contexts



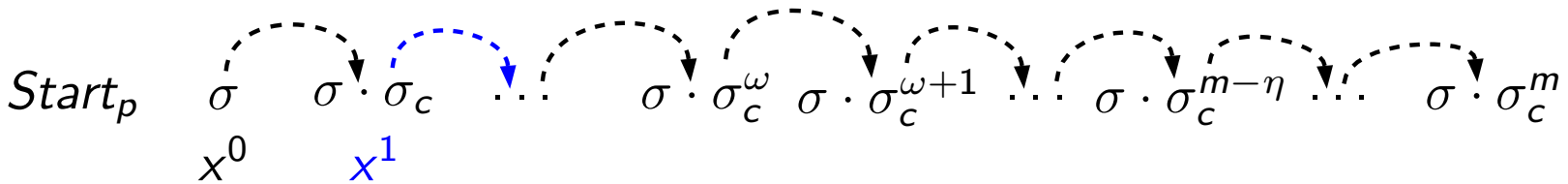
The Moral of the Story

- In the cyclic call sequence, the computation begins from the first call string and influences successive call strings.
- In the cyclic return sequence, the computation begins from the last call string and influences the preceding call strings.

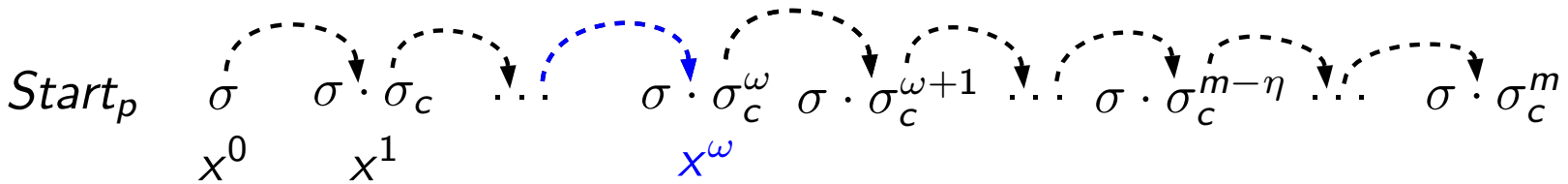
Bounding the Call String Length



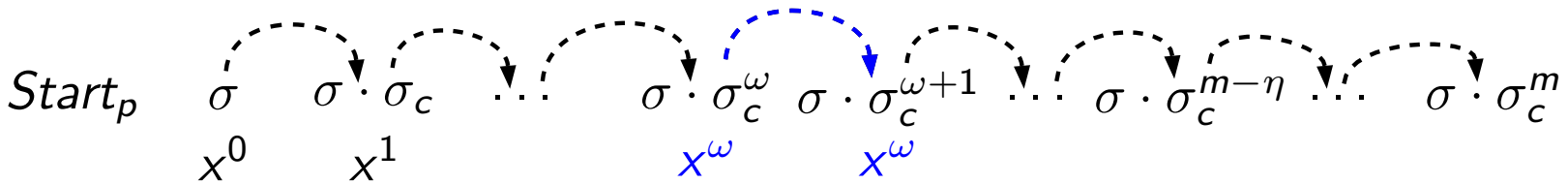
Bounding the Call String Length



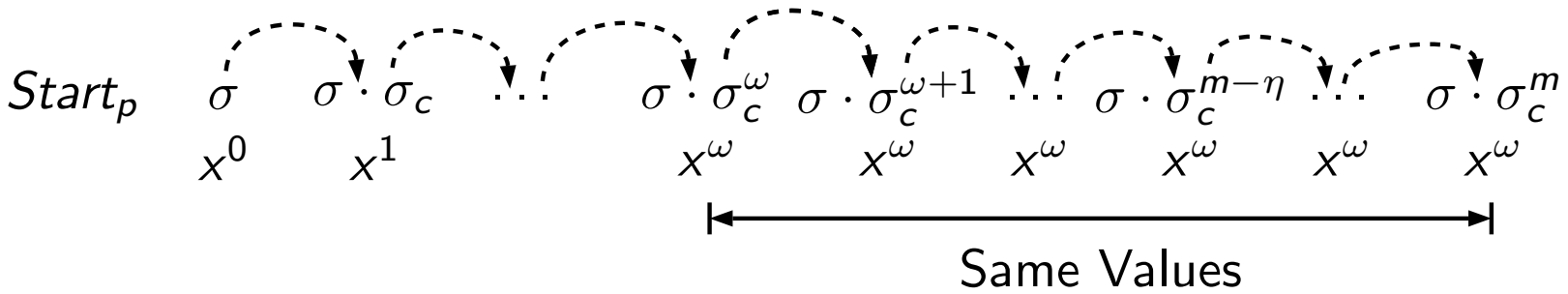
Bounding the Call String Length



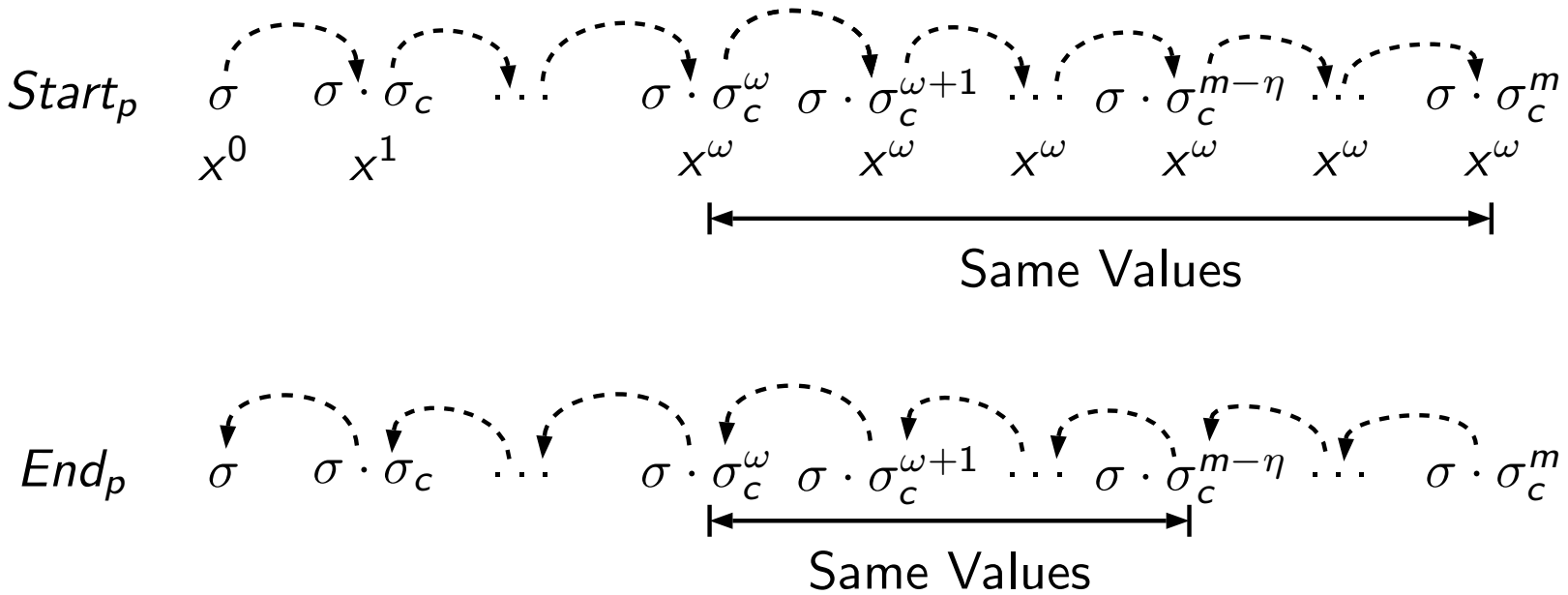
Bounding the Call String Length



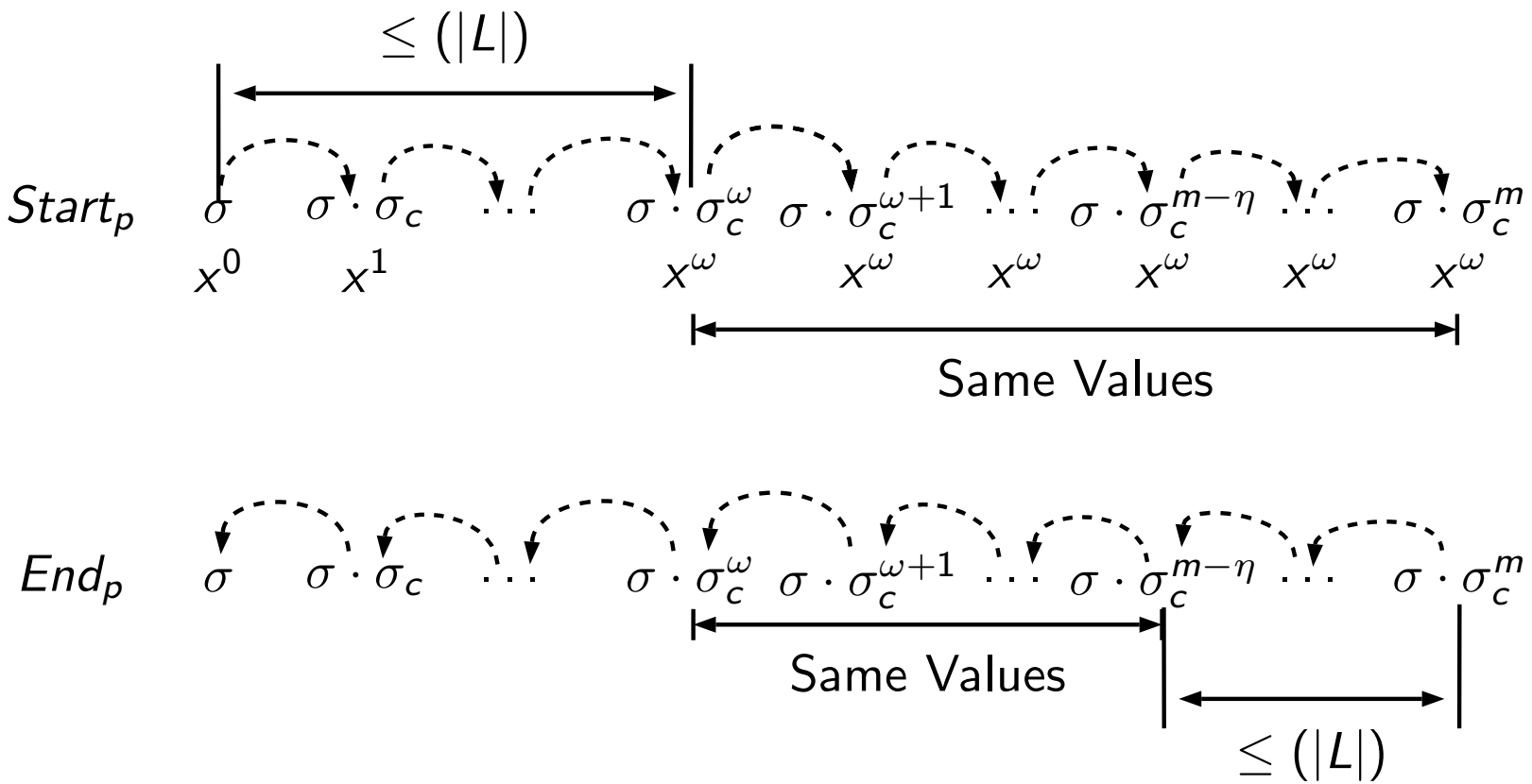
Bounding the Call String Length



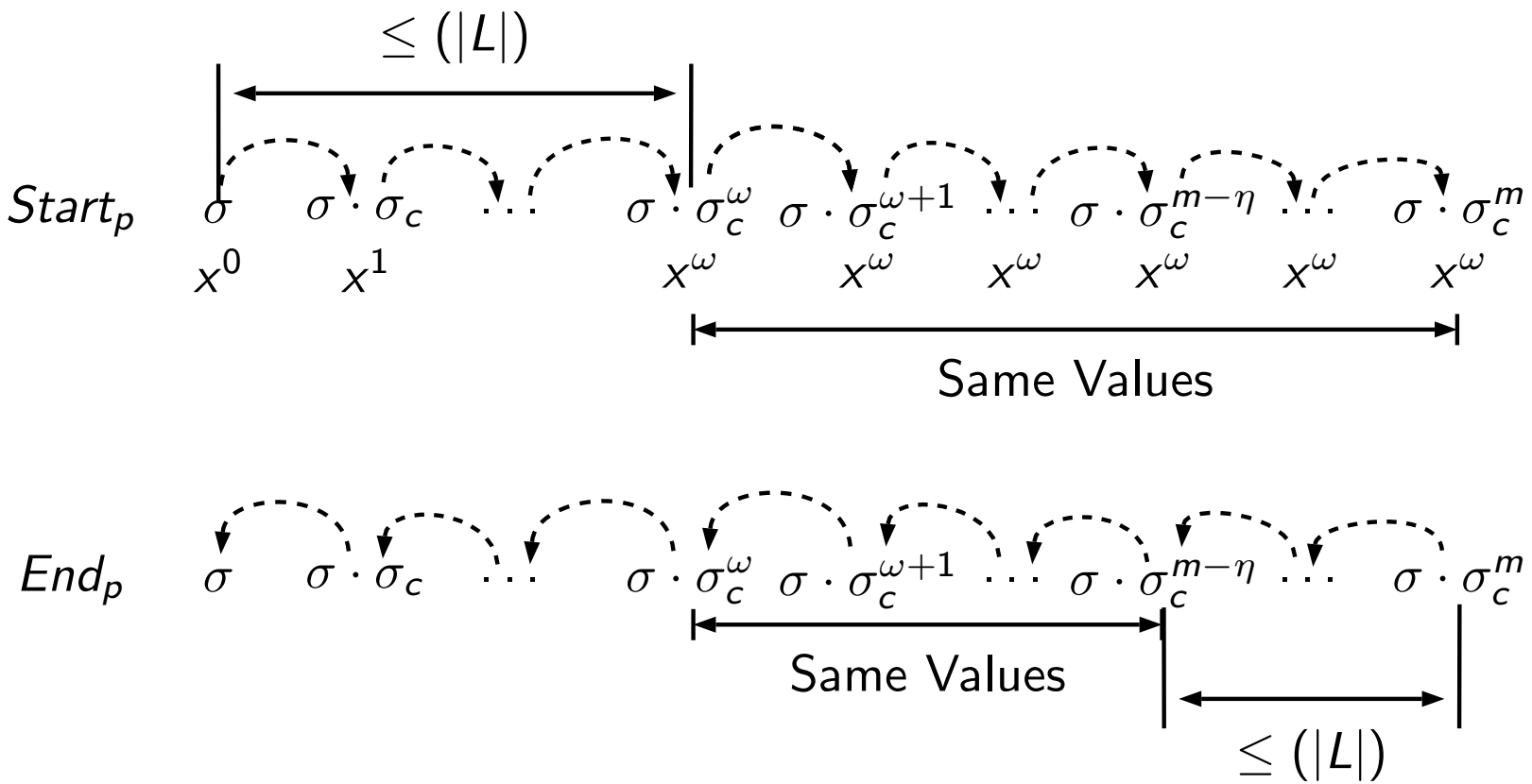
Bounding the Call String Length



Bounding the Call String Length

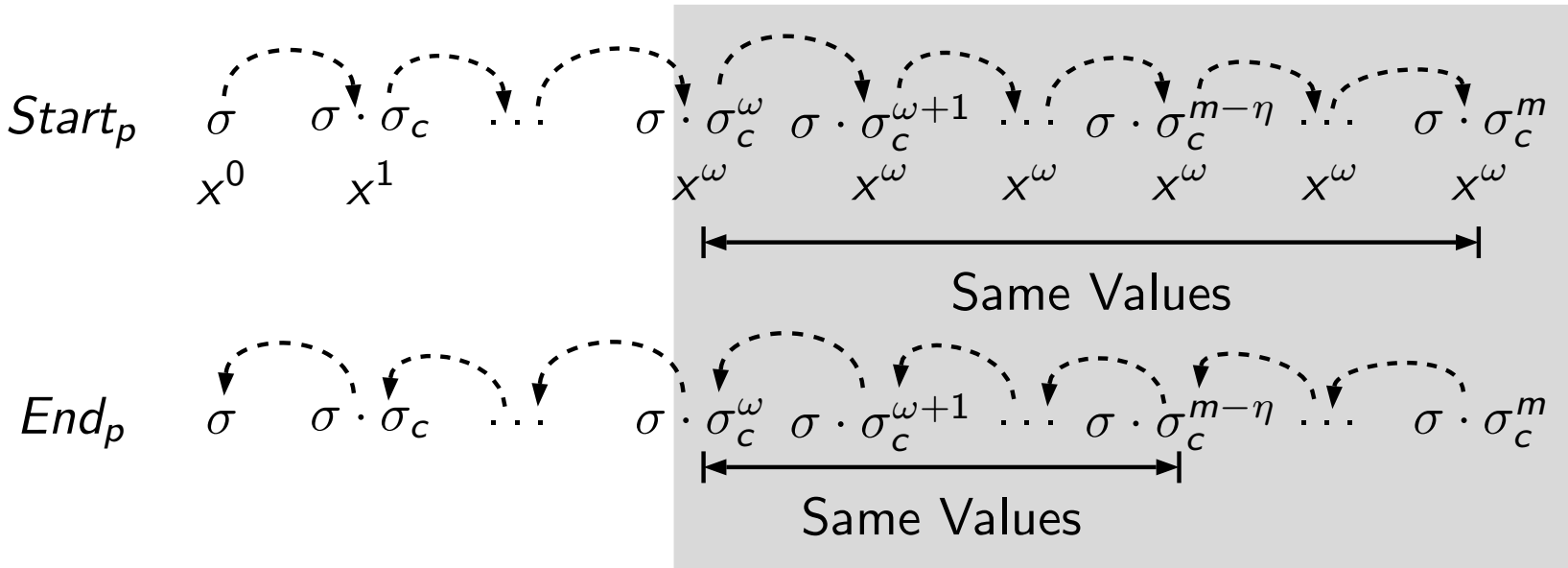


Bounding the Call String Length

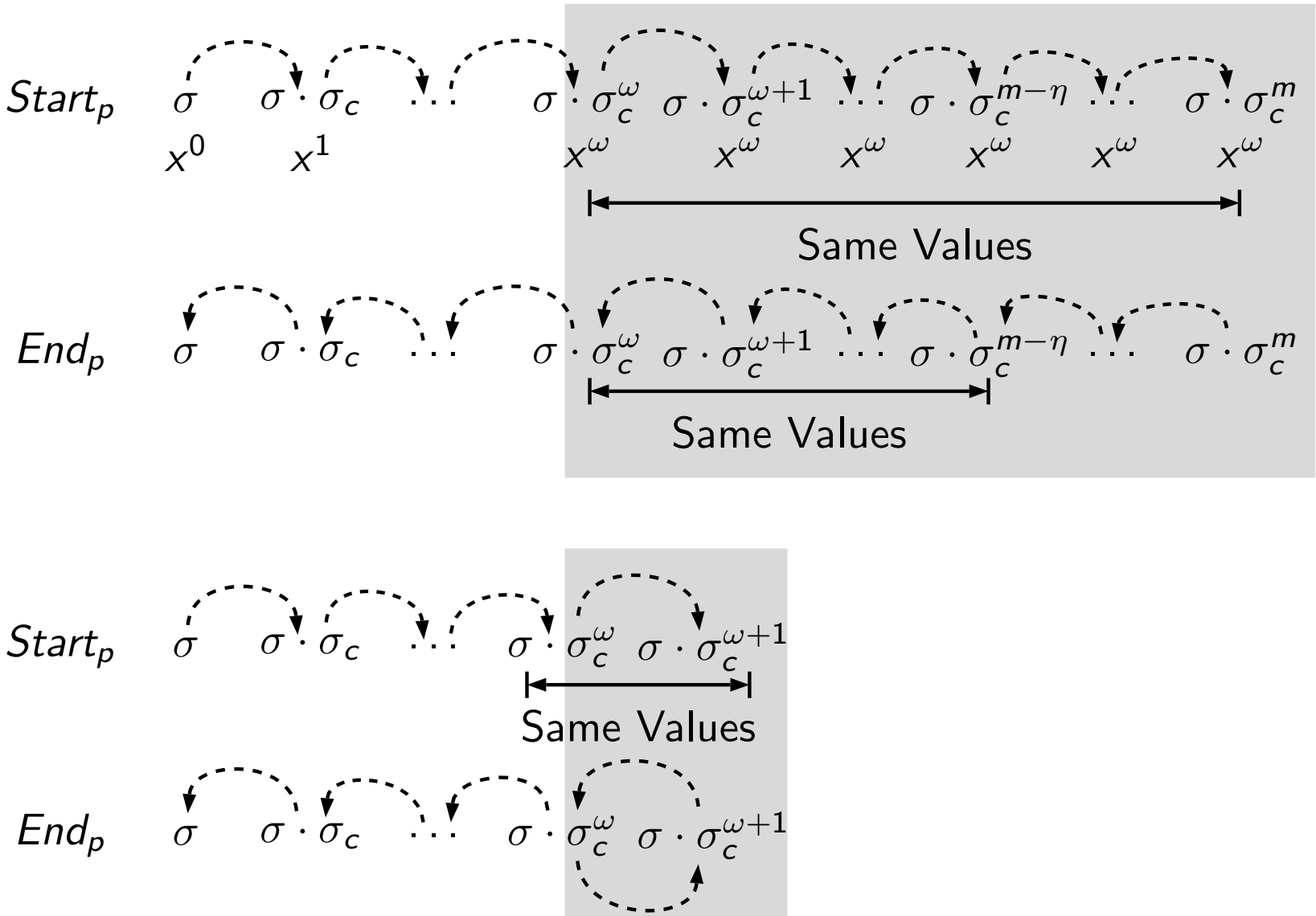


- m must be $>= 2 \cdot |L|$
- Since there could be K call sites in σ_c, call strings of length $K \cdot 2 \cdot |L|$ are sufficient.

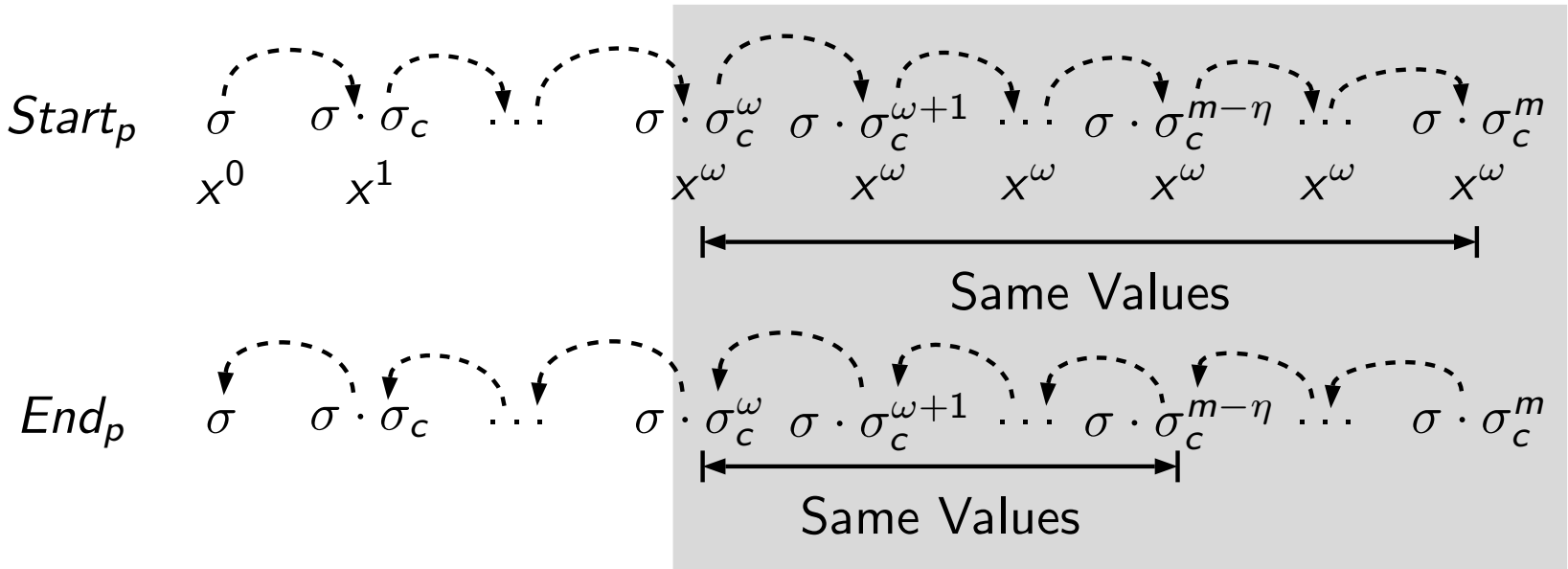
Using Data Flow Values to Bound Call Strings

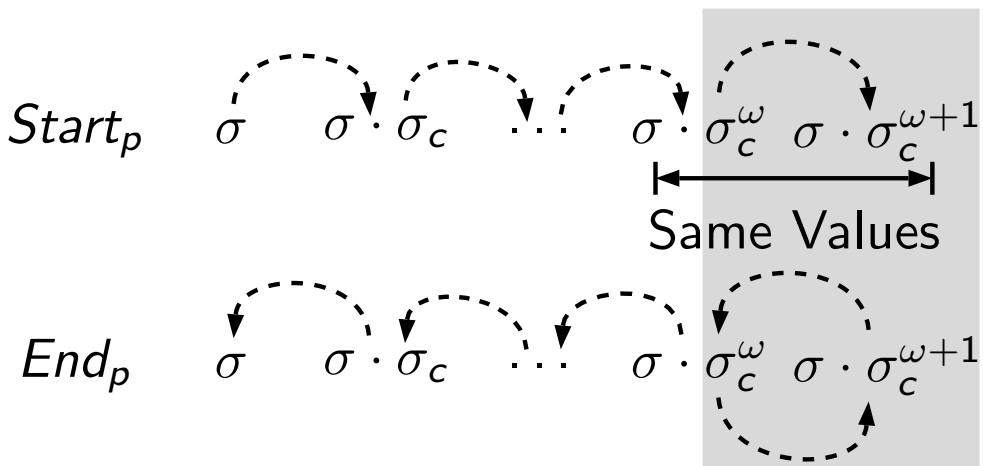


Using Data Flow Values to Bound Call Strings



Using Data Flow Values to Bound Call Strings





$\omega \leq |L| \Rightarrow$

- at most $|L| + 1$ occurrences of σ_c.
- call string length is at most $K \cdot (|L| + 1)$

The Final Algorithm

- Use exactly the same method with this small change:

The Final Algorithm

- Use exactly the same method with this small change:
  - discard redundant call strings at the start of every procedure, and

The Final Algorithm

- Use exactly the same method with this small change:
  - discard redundant call strings at the start of every procedure, and
  - simulate regeneration of call strings at the end of every procedure.

The Final Algorithm

- Use exactly the same method with this small change:
  - discard redundant call strings at the start of every procedure, and
  - simulate regeneration of call strings at the end of every procedure.
- Intuition:

The Final Algorithm

- Use exactly the same method with this small change:
  - discard redundant call strings at the start of every procedure, and
  - simulate regeneration of call strings at the end of every procedure.
- Intuition:
  - If σ_1 and σ_2 have equal values at $Start_p$,

The Final Algorithm

- Use exactly the same method with this small change:
  - discard redundant call strings at the start of every procedure, and
  - simulate regeneration of call strings at the end of every procedure.
- Intuition:
  - If σ_1 and σ_2 have equal values at $Start_p$,
  - Then, since σ_1 and σ_2 are transformed in the same manner by traversing the same set of paths,

The Final Algorithm

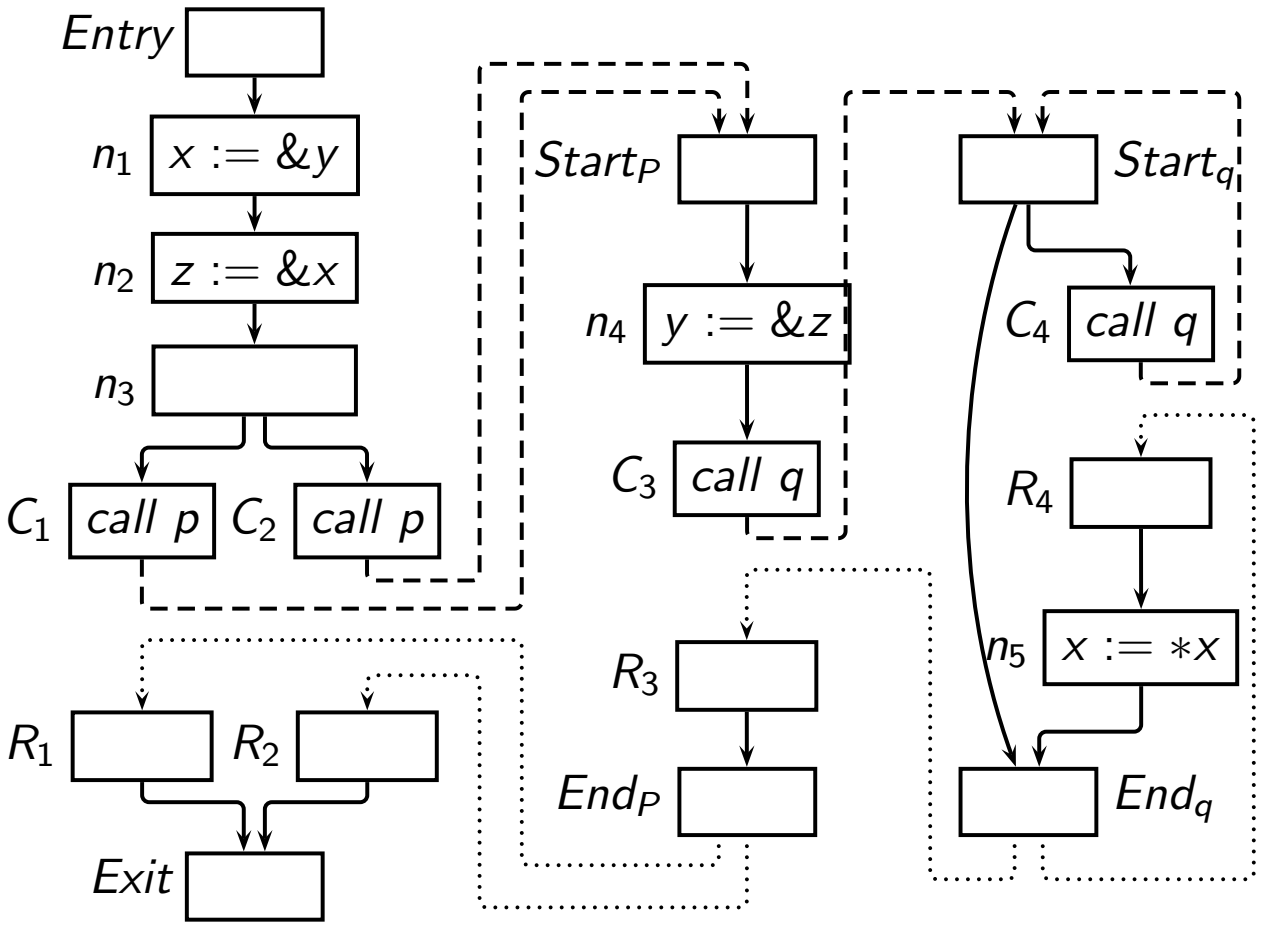
- Use exactly the same method with this small change:
  - discard redundant call strings at the start of every procedure, and
  - simulate regeneration of call strings at the end of every procedure.
- Intuition:
  - If σ_1 and σ_2 have equal values at $Start_p$,
  - Then, since σ_1 and σ_2 are transformed in the same manner by traversing the same set of paths,
  - The values associated with them will also be transformed in the same manner and will continue to remain equal at End_p.

Implementing the Required Change

Let $shortest(\sigma, u)$ denote the shortest call string which has the same value as σ at u.

$$
\begin{aligned}
represent(\langle \sigma, d \rangle, Start_p) &= \langle shortest(\sigma, Start_p), d \rangle \\
regenerate(\langle \sigma, d \rangle, End_p) &= \{\langle \sigma', d \rangle \mid \sigma \text{ and } \sigma' \text{ have the same value at } Start_p\}
\end{aligned}
$$

Points-To Analysis



- $|L| = 512$, $K = 3$.
- The classical method requires 789507 call strings
- We need 5 call strings!

Approximate Version

- For framework with infinite lattices, a fixed point for cyclic call sequence may not exist.
- Use a demand driven approach:
  After a dynamically definable limit, start merging the values and associate them with the last call string.
  Assumption: Height of the lattice is finite.

Part 5

Empirical Measurements

Reaching Definitions Analysis in GCC 4.0

Program	LoC	$\#F$	$\#C$	$3K$ length bound				Proposed Approach		
				K	$\#CS$	Max	Time	$\#CS$	Max	Time
hanoi	33	2	4	4	100000+	99922	3973 $\times 10^3$	8	7	2.37
bit_gray	53	5	11	7	100000+	31374	2705 $\times 10^3$	17	6	3.83
analyzer	288	14	20	2	21	4	20.33	21	4	1.39
distray	331	9	21	6	96	28	322.41	22	4	1.11
mason	350	9	13	8	100000+	22143	432 $\times 10^3$	14	4	0.43
fourinarow	676	17	45	5	510	158	397.76	46	7	1.86
sim	1146	13	45	8	100000+	33546	1427 $\times 10^3$	211	105	234.16
181_mcf	1299	17	24	6	32789	32767	484 $\times 10^3$	41	11	5.15
256_bzip2	3320	63	198	7	492	63	258.33	406	34	200.19

- LoC is the number of lines of code,
- $\#F$ is the number of procedures,
- $\#C$ is the number of call sites,
- $\#CS$ is the number of call strings
- Max denotes the maximum number of call strings reaching any node.
- Analysis time is in milliseconds.

Part 6

Conclusions

A Summary of Contributions

- Clearly identified the exact set of call strings required.

A Summary of Contributions

- Clearly identified the exact set of call strings required.
- Value based termination of call string construction. No need to construct call strings upto a fixed length.

A Summary of Contributions

- Clearly identified the exact set of call strings required.
- Value based termination of call string construction. No need to construct call strings upto a fixed length.
- Only as many call strings are constructed as are required.

A Summary of Contributions

- Clearly identified the exact set of call strings required.
- Value based termination of call string construction. No need to construct call strings upto a fixed length.
- Only as many call strings are constructed as are required.
- Significant reduction in space and time.

A Summary of Contributions

- Clearly identified the exact set of call strings required.
- Value based termination of call string construction. No need to construct call strings upto a fixed length.
- Only as many call strings are constructed as are required.
- Significant reduction in space and time.
- Worst case call string length becomes linear in the size of the lattice instead of the original quadratic.

A Summary of Contributions

- Clearly identified the exact set of call strings required.
- Value based termination of call string construction. No need to construct call strings upto a fixed length.
- Only as many call strings are constructed as are required.
- Significant reduction in space and time.
- Worst case call string length becomes linear in the size of the lattice instead of the original quadratic.

All this is achieved by a simple change without compromising on the precision, simplicity, and generality of the classical method.

Conclusions

- Compromising on precision may not be necessary for efficiency.
- Separating the necessary information from redundant information is much more significant.
- A precise modelling of the process of analysis is often an eye opener.

Future Work

- Implementing for points-to analysis.
- Modelling parameters and return values.
- Applying it to Heap Reference Analysis [TOPLAS, Nov 2007]
- Machine independent optimizer generator in GCC.

Reference

- To appear in CC 2008.

Acknowledgement

- Implementation was carried out by Seema Ravandale.

Last but not the least ...

Thank You!