Interprocedural analyses and optimizations

Costs of procedure calls

- Up until now, we treated calls conservatively:
  - make the flow function for call nodes return top
  - start iterative analysis with incoming edge of the CFG set to top
  - This leads to less precise results: “lost-precision” cost

- Calls also incur a direct runtime cost
  - cost of call, return, argument & result passing, stack frame maintainance
  - “direct runtime” cost

Addressing costs of procedure calls

- Technique 1: try to get rid of calls, using inlining and other techniques

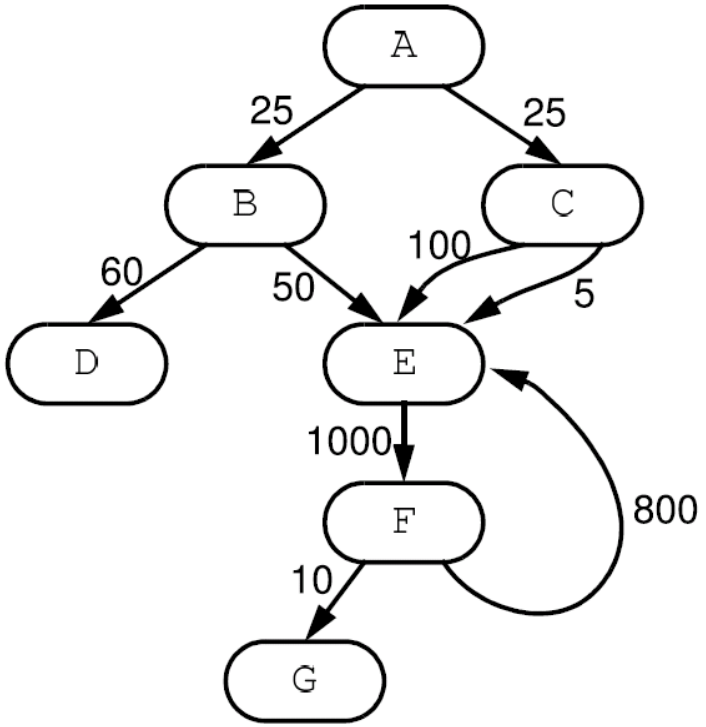
- Technique 2: interprocedural analysis, for calls that are left

Inlining

- Replace call with body of callee
- Turn parameter- and result-passing into assignments
  - do copy prop to eliminate copies
- Manage variable scoping correctly
  - rename variables where appropriate

Program representation for inlining

- Call graph
  - nodes are procedures
  - edges are calls, labelled by invocation counts/frequency
- Hard cases for builing call graph
  - calls to/from external routines
  - calls through pointers, function values, messages
- Where in the compiler should inlining be performed?



Inlining pros and cons (discussion)

Inlining pros and cons

- Pros
  - eliminate overhead of call/return sequence
  - eliminate overhead of passing args & returning results
  - can optimize callee in context of caller and vice versa
- Cons
  - can increase compiled code space requirements
  - can slow down compilation
  - recursion?
- Virtual inlining: simulate inlining during analysis of caller, but don’t actually perform the inlining

Which calls to inline (discussion)

- What affects the decision as to which calls to inline?

Which calls to inline

- What affects the decision as to which calls to inline?
  - size of caller and callee (easy to compute size before inlining, but what about size after inlining?)
  - frequency of call (static estimates or dynamic profiles)
  - call sites where callee benefits most from optimization (not clear how to quantify)
  - programmer annotations (if so, annotate procedure or call site? Also, should the compiler really listen to the programmer?)

Inlining heuristics

- Strategy 1: superficial analysis
  - examine source code of callee to estimate space costs, use this to determine when to inline
  - doesn’t account for post-inlining optimizations
- How can we do better?

Inlining heuristics

- Strategy 2: deep analysis
  - perform inlining
  - perform post-inlining analysis/optimizations
  - estimate benefits from opts, and measure code space after opts
  - undo inlining if costs exceed benefits
  - better accounts for post-inlining effects
  - much more expensive in compile-time
- How can we do better?

Inlining heuristics

- Strategy 3: amortized version of 2

  [Dean & Chambers 94]

  – perform strategy 2: an inlining “trial”
  – record cost/benefit trade-offs in persistent database
  – reuse previous cost/benefit results for “similar” call sites

Inlining heuristics

- Strategy 4: use machine learning techniques
- For example, use genetic algorithms to evolve heuristics for inlining
  - fitness is evaluated on how well the heuristics do on a set of benchmarks
  - cross-populate and mutate heuristics
- Can work surprisingly well to derive various heuristics for compilers

Another way to remove procedure calls

```
int f(...) {
    if (...) return g(...);
    ...
    return h(i(....), j(...));
}
```

Tail call eliminiation

- Tail call: last thing before return is a call
  - callee returns, then caller immediately returns
- Can splice out one stack frame creation and destruction by jumping to callee rather than calling
  - callee reuses caller's stack frame & return address
  - callee will return directly to caller's caller
  - effect on debugging?

Tail recursion elimination

- If last operation is self-recursive call, what does tail call elimination do?

Tail recursion elimination

- If last operation is self-recursive call, what does tail call elimination do?
- Transforms recursion into loop: tail recursion elimination
  - common optimization in compilers for functional languages
  - required by some language specifications, eg Scheme
  - turns stack space usage from O(n) to O(1)

Addressing costs of procedure calls

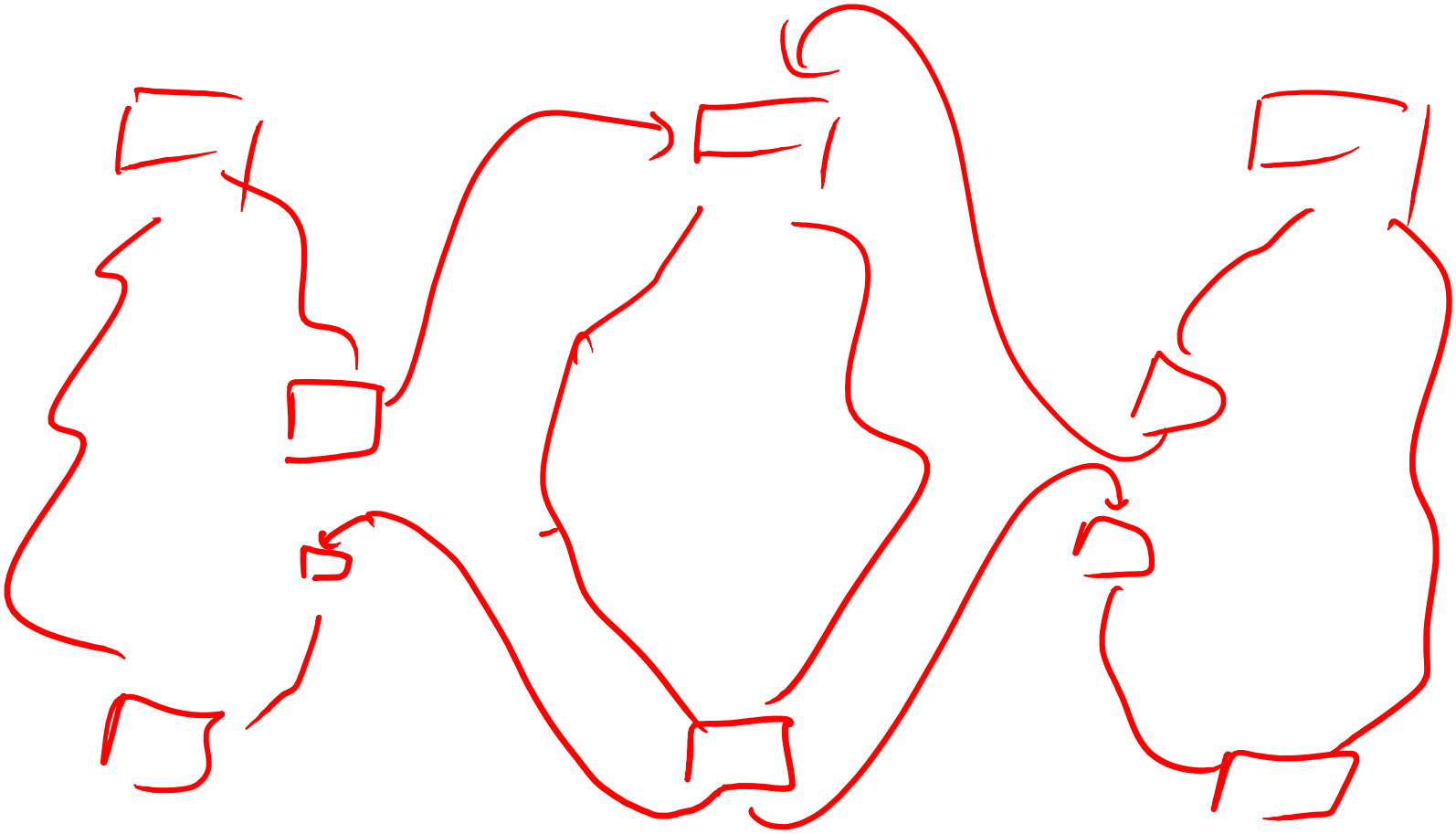
- Technique 1: try to get rid of calls, using inlining and other techniques

- Technique 2: interprocedural analysis, for calls that are left

Interprocedural analysis

- Extend intraprocedural analyses to work across calls
- Doesn’t increase code size
- But, doesn’t eliminate direct runtime costs of call
- And it may not be as effective as inlining at cutting the “precision cost” of procedure calls

A simple approach (discussion)



A simple approach

- Given call graph and CFGs of procedures, create a single CFG (control flow super-graph) by:
  - connecting call sites to entry nodes of callees (entries become merges)
  - connecting return nodes of callees back to calls (returns become splits)
- Cons:
  - speed?
  - separate compilation?
  - imprecision due to “unrealizable paths”

Another approach: summaries (discussion)

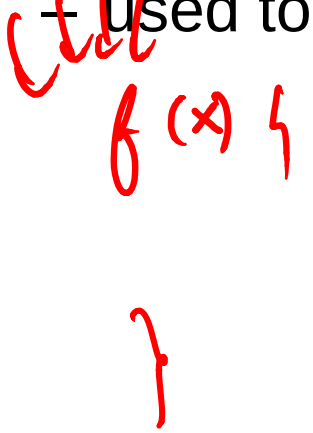
Code examples for discussion

```
global a;
a := 5;
f(...);
b := a + 10;
```

```
global a;
global b;

f(p) {
  *p := 0;
}

g() {
   a := 5;
   f(&a);
   b := a + 10;
}

h() {
   a := 5;
   f(&b);
   b := a + 10;
}
```

Another approach: summaries

- Compute summary info for each procedure
- Callee summary: summarizes effect/results of callee procedures for callers
  - used to implement the flow function for a call node
- Caller summaries: summarizes context of all callers for callee procedure
  - used to start analysis of a procedure



Examples of summaries

MOD
- the set of variables possibly modified by a call to a proc

USE
- the set of variables possibly read by a call to a proc

MOD-BEFORE-USE
- the set of variables definitely modified before use

LIVE-RESULT
- whether result may be live in caller

CONST-ARGS
- the constant values of those formals that are constant

CONST-RESULT
- the constant result of a procedure, if it's a constant

ARGS-MAY-POINT-TO
- may-point-to info for formal parameters

RESULT-MAY-POINT-TO
- may-point-to info for the result

PURE
- a pure, terminating function, without side-effects

Issues with summaries

- Level of “context” sensitivity:
  - For example, one summary that summarizes the entire procedure for all call sites
  - Or, one summary for each call site (getting close to the precision of inlining)
  - Or ...
- Various levels of captured information
  - as small as a single bit
  - as large as the whole source code for callee/callers
- How does separate compilation work?

How to compute summaries

- Using iterative analysis
- Keep the current solution in a map from procs to summaries
- Keep a worklist of procedures to process
- Pick a proc from the worklist, compute its summary using intraprocedural analysis and the current summaries for all other nodes
- If summary has changed, add callers/callees to the worklist for callee/caller summaries

How to compute callee summaries

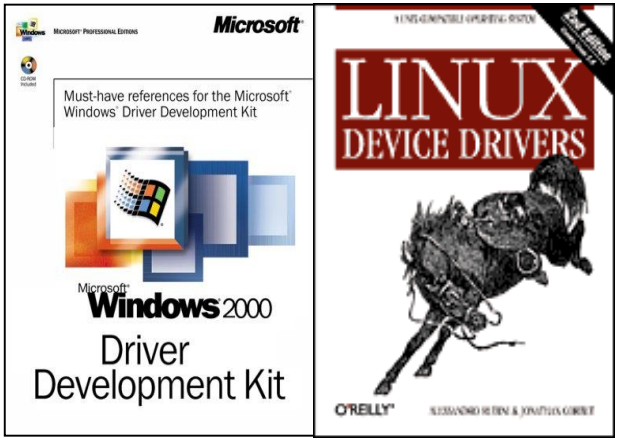
```
let m: map from proc to computed summary
let worklist: work list of procs

for each proc p in call graph do
   m(p) := ⊥

for each proc p do
   worklist.add(p)

while (worklist.empty.not) do
   let p := worklist.remove_any;
   // compute summary using intraproc analysis
   // and current summaries m
   let summary := compute_summary(p,m);
   if (m(p) ≠ summary)
      m(p) := summary;
      for each caller c of p
         worklist.add(c)
```

Examples

- Let’s see how this works on some examples

- We’ll use an analysis for program verification as a running example

Protocol checking



Interface usage rules in documentation

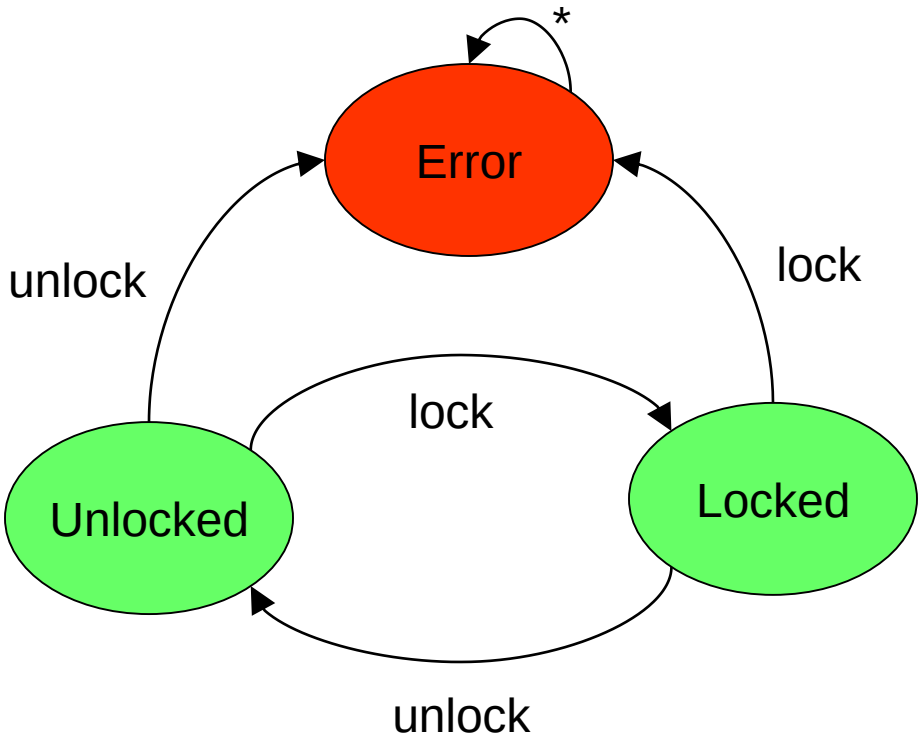
- Order of operations, data access
- Resource management
- Incomplete, wordy, not checked

Violated rules ⇒ crashes

- Failed runtime checks
- Unreliable software

FSM protocols

- These protocols can often be expressed as FSMs
- For example: lock protocol



FSM protocols

- Alphabet of FSM are actions that affect the state of the FSM
- Often leave error state implicit
- These FSMs can get pretty big for realistic kernel protocols

FSM protocol checking

- Goal: make sure that FSM does not enter error state
- Lattice:

FSM protocol checking

- Goal: make sure that FSM does not enter error state
- Lattice: $(L, \bot, \top, \sqsubseteq, \sqcap, \sqcup)$

$(2^{\{u,l,e\}}, \emptyset, \{u,l,e\}, \cap, \cup)$

Lock protocol example

```
main() {
    g();
    f();
    lock;
    unlock;
}
```

```
f() {
    h();
    if (...) {
        main();
    }
}
```

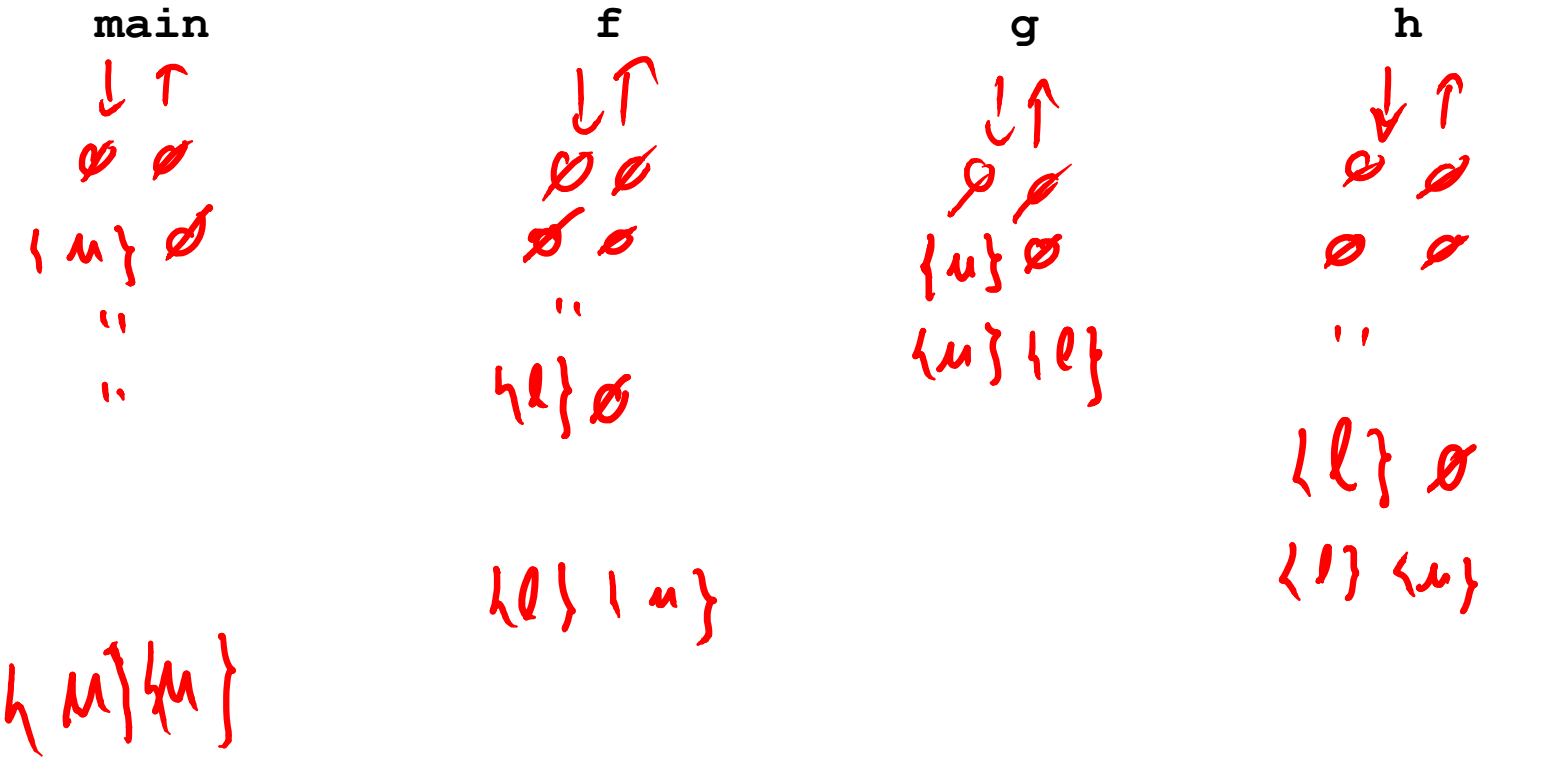
```
g() {
    lock;
}

h() {
    unlock;
}
```

Lock protocol example

```
main() {
  g();
  f();
  lock;
  unlock;
}
```

```
f() {
  h();
  if (...) { main(); }
}
```

```
g() { lock; }

h() { unlock; }
```

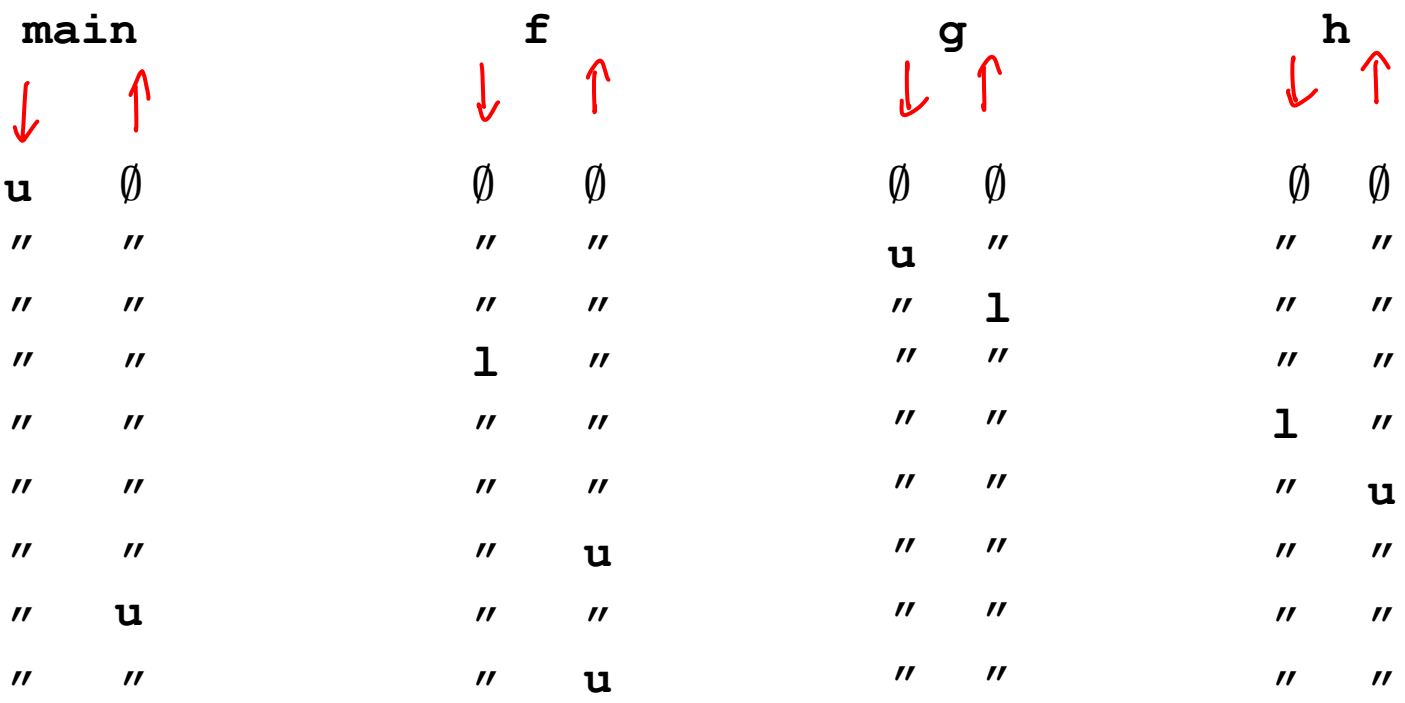


Lock protocol example

```
main() {
   g();
   f();
   lock;
   unlock;
}
```

```
f() {
   h();
   if (...) { main(); }
}
```

```
g() { lock; }

h() { unlock; }
```



Another lock protocol example

```
main() {
    g();
    f();
    lock;
    unlock;
}
```

```
f() {
    g();
    if (...) {
        main();
    }
}
```

```
g() {
    if(isLocked()) {
        unlock;
    }
    else {
        lock;
    }
}
```

Another lock protocol example

```
main() {
   g();
   f();
   lock;
   unlock;
}
```

```
f() {
   g();
   if (...) { main(); }
}
```

```
g() {
   if(isLocked()) {
      unlock;
   } else { lock; }
}
```

main		f		g	
↓	↑	↓	↑	↓	↑
u	∅	∅	∅	∅	∅
〃	〃	〃	〃	u	〃
〃	〃	〃	〃	〃	l
〃	〃	l	〃	〃	〃
{u, l}		l	l	{u, l}	{u, l}

Another lock protocol example

```
main() {
   g();
   f();
   lock;
   unlock;
}
```

```
f() {
   g();
   if (...) { main(); }
}
```

```
g() {
   if(isLocked()) {
      unlock;
   } else { lock; }
}
```

main		f		g	
u	∅	∅	∅	∅	∅
〃	〃	〃	〃	u	〃
〃	〃	〃	〃	〃	l
〃	〃	l	〃	〃	〃
〃	〃	〃	〃	{u,l}	〃
〃	〃	〃	〃	〃	{u,l}
〃	〃	{u,l}	{u,l}	〃	〃
{u,l}	{u,e}	〃	〃	〃	〃

What went wrong?

What went wrong?

- We merged info from two call sites of g()
- Solution: summaries that keep different contexts separate
- What is a context?

Approach #1 to context-sensitivity

- Keep information for different call sites separate

- In this case: context is the call site from which the procedure is called

Example again

```
main() {
  g();
  f();
    lock;
    unlock;
}
```

```
f() {
  g();
    if (...) {
}
```

```
main(); }
```

```
g() {
    if(isLocked()) {
        unlock;
    } else { lock; }
}
```

main f g

Example again

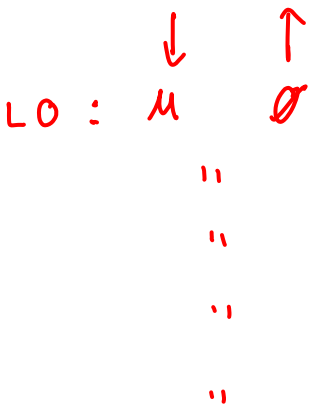
```
main() {
  g();
  f();
  lock;
  unlock;
}
```

```
f() {
  g();
  if (...) { main(); }
}
```

```
g() {
  if(isLocked()) {
    unlock;
  } else { lock; }
}
```

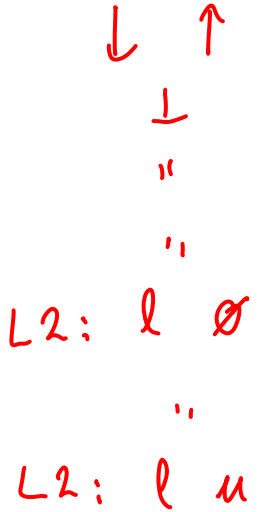
L0 L1 L2 L3 L4

main

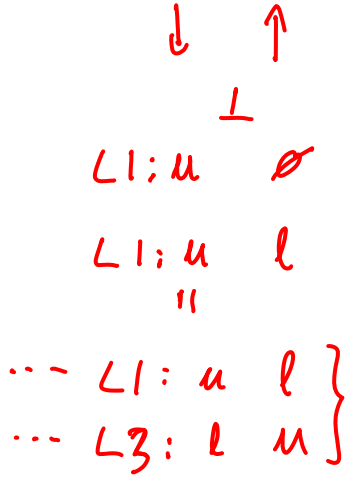


L0 u u
L4 u u

f



g



How should we change the example?

- How should we change our example to break our context sensitivity strategy?

```
main() {
   g();
   f();
   lock;
   unlock;
}
```

```
f() {
   g();
   if (...) {
      main();
   }
}
```

```
g() {
   if(isLocked()) {
      unlock;
   }
   else {
      lock;
   }
}
```

Answer

```
main() {
   h();
   f();
   lock;
   unlock;
}
```

```
f() {
   h();
   if (...) {
      main();
   }
}
```

```
h() { g() }

g() {
   if(isLocked()) {
      unlock;
   }
   else {
      lock;
   }
}
```

In general

- Our first attempt was to make the context be the immediate call site
- Previous example shows that we may need 2 levels of the stack
  - the context for an analysis of function f is: call site L_1 where f was called from AND call site L_2 where f's caller was called from
- Can generalize to k levels
  - k-length call strings approach of Sharir and Pnueli
  - Shiver's k-CFA

Approach #2 to context-sensitivity

Approach #2 to context-sensitivity

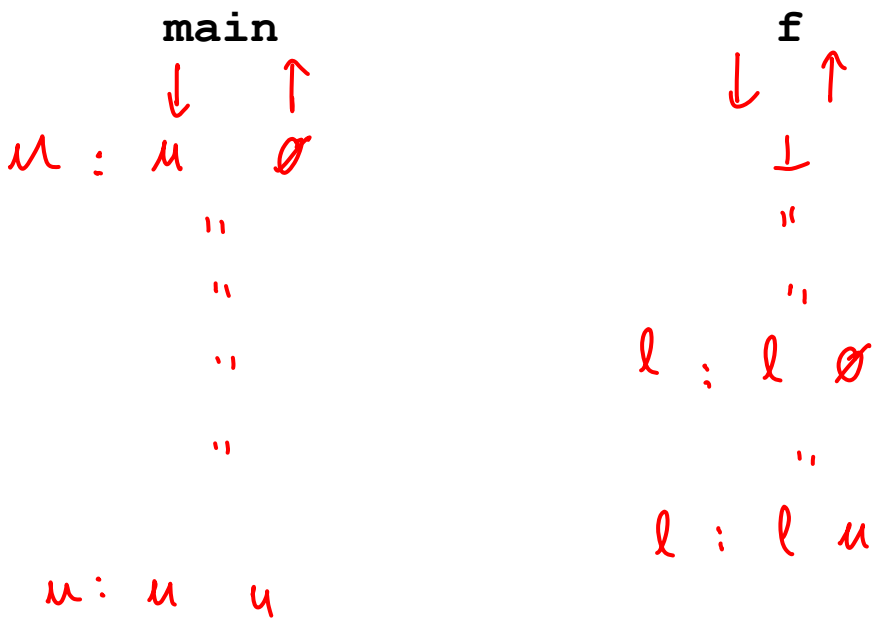
- Use dataflow information at call site as the context, not the call site itself

Using dataflow info as context

```
main() {
    g();
    f();
    lock;
    unlock;
}
```

```
f() {
    g();
    if (...) { main(); }
}
```

```
g() {
    if(isLocked()) {
        unlock;
    } else { lock; }
}
```



Transfer functions

- Our pairs of summaries look like functions from input information to output information
- We call these transfer functions
- Complete transfer functions
  - contain entries for all possible incoming dataflow information
- Partial transfer functions
  - contain only some entries, and continually refine during analysis

Top-down vs. bottom-up

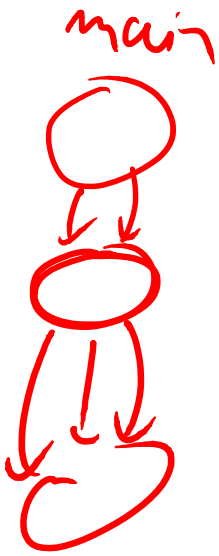
- We've always run our interproc analysis top down: from main, down into procs
- For data-based context sensitivity, can also run the analysis bottom-up
  - analyze a proc in all possibly contexts
  - if domain is distributive, only need to analyze singleton sets

Bottom-up example

```
main() {
   g();
   f();
   lock;
   unlock;
}
```

```
f() {
   g();
   if (...) { main(); }
}
```

```
g() {
   if(isLocked()) {
      unlock;
   } else { lock; }
}
```



main	f	g
⊥	⊥	⊥
"	"	u → l l → u
"	u → l l → u	"
u → u l → e	"	"
"	u → {l,e} l → u	"
u → u l → e		

Top-down vs. bottom-up

- What are the tradeoffs?

Top-down vs. bottom-up

- What are the tradeoffs?
  - In top-down, only analyze procs in the context that occur during analysis, whereas in bottom-up, may do useless work analyzing proc in a data context never used during analysis
  - However, top-down requires analyzing a given function at several points in time that are far away from each other. If the entire program can't fit in RAM, this will lead to unnecessary swapping. On the other hand, can do bottom-up as one pass over the call-graph, one SCC at a time. Once a proc is analyzed, it never needs to be reloaded in memory.
  - top-down better suited for infinite domains

In class exercise

```
    main() {
L1:   f()
    }

    f() {
      if(Unlocked()) {
        lock;
L2:     f();
      } else {
        unlock;
      }
    }
```

In class exercise

```
    main() {
L1:   f()
    }

    f() {
      if(Unlocked()) {
        lock;
L2:     f();
      } else {
        unlock;
      }
    }
```

In class exercise

```
    main() {
L1:   f()
    }

    f() {
      if(Unlocked()) {
        lock;
L2:     f();
      } else {
        unlock;
      }
    }
```

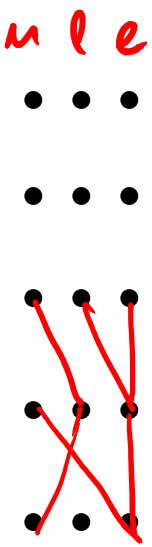
main		f	
↓	↑	↓	↑
u	∅	∅	∅
〃	〃	u	〃
〃	〃	u,l	〃
〃	〃	u,l	u
〃	u	u,l	u

Reps Horwitz and Sagiv 95 (RHS)

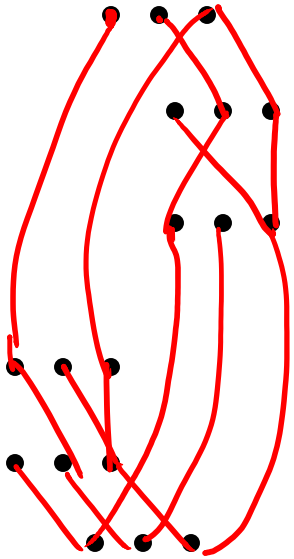
- Another approach to context-sensitive interprocedural analysis
- Express the problem as a graph reachability query
- Works for distributive problems

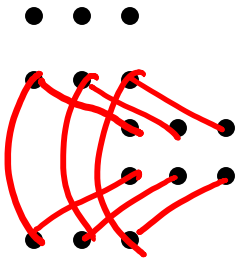
Reps Horwitz and Sagiv 95 (RHS)

```
main() {
    g();
    f();
    lock;
    unlock;
}
```



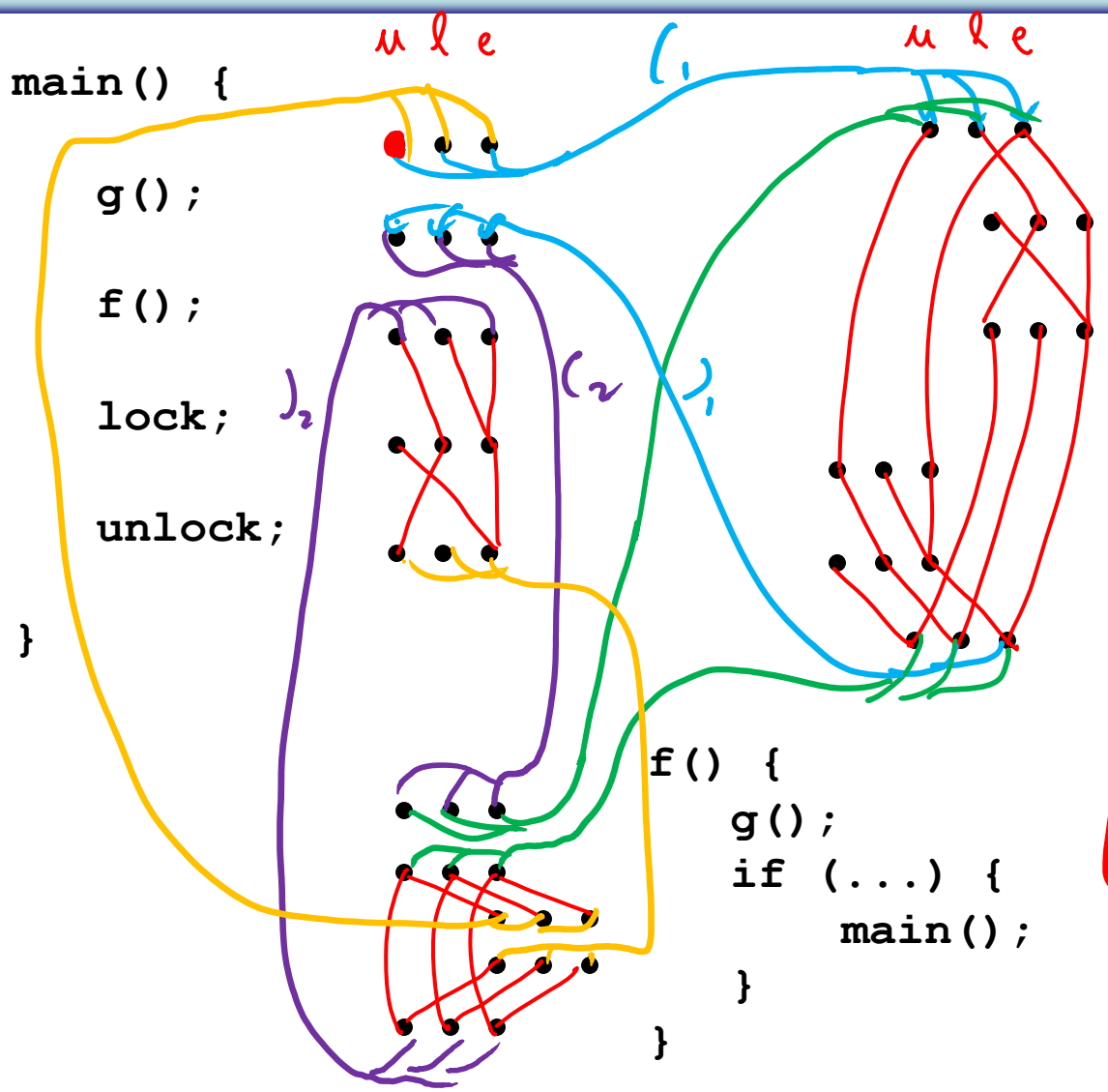

```
g() {
    if(isLocked()) {
        unlock;
    } else {
        lock;
    }
}
```





```
f() {
    g();
    if (...) {
        main();
    }
}
```

Reps Horwitz and Sagiv 95 (RHS)



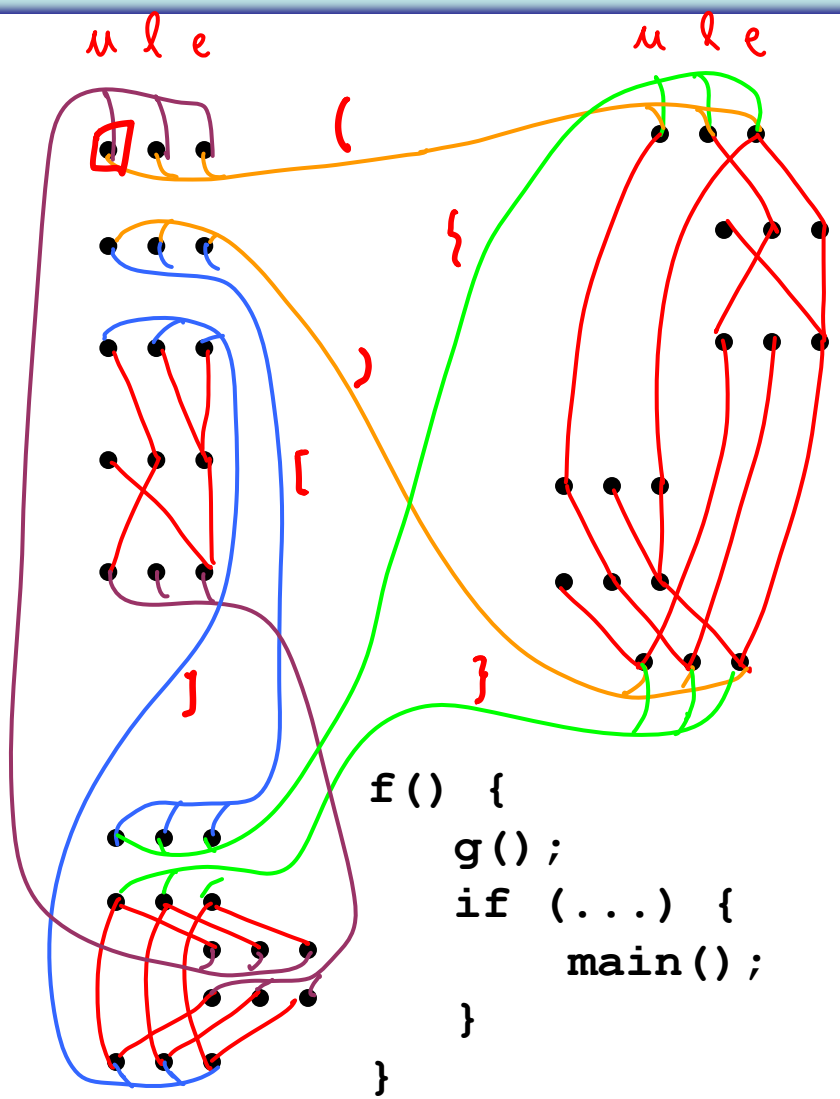
$L ::= \varepsilon \mid (_1 L)_1 \mid (_2 L)_2$

LL

$\varepsilon \quad (_1)_1 \quad (_1(_2(_2)_2)_1 \quad (_1)_2$

Reps Horwitz and Sagiv 95 (RHS)

```
main() {
    g();
    f();
    lock;
    unlock;
}
```




```
f() {
    g();
    if (...) {
        main();
    }
}
```

```
g() {
    if(isLocked()) {
        unlock;
    } else {
        lock;
    }
}
```

L = (_i L)_i | ε | LL

Procedure specialization

- Interprocedural analysis is great for callers
- But for the callee, information is still merged

```
main() {
    x := new A(...);
    y := x.g();
    y.f();

    x := new A(...);
    y := x.g();
    y.f();

    x := new B(...);
    y:= x.g();
    y.f();
}
```

Type-of(x) = Type-of(ret)

```
// g too large to inline
g(x) {
    x.f();
    // lots of code
    return x;
}

// but want to inline f
f(x@A) { ... }
f(x@B) { ... }
```

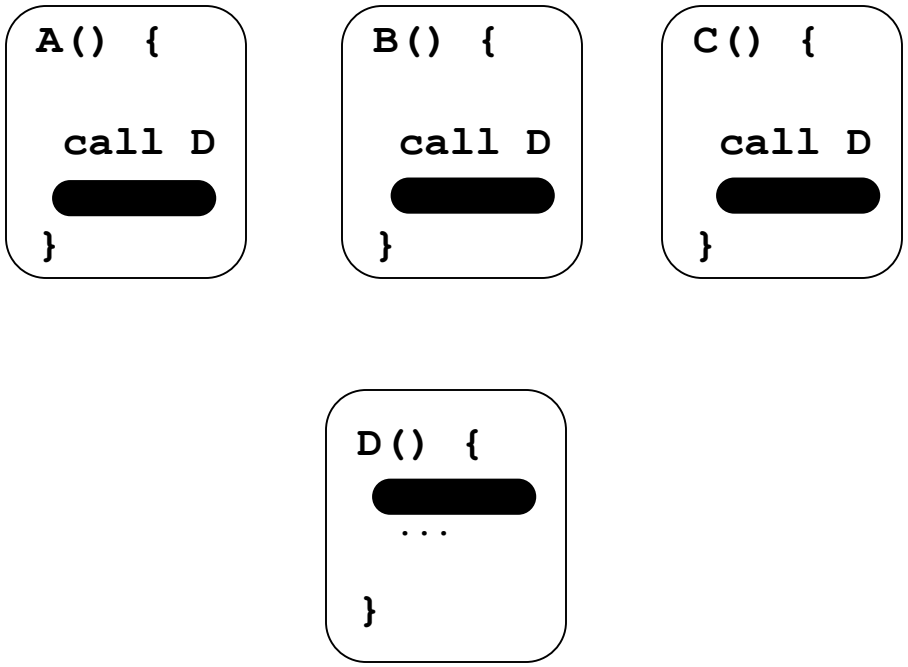
Procedure specialization

- Specialize g for each dataflow information
- “In between” inlining and context-sensitve interproc

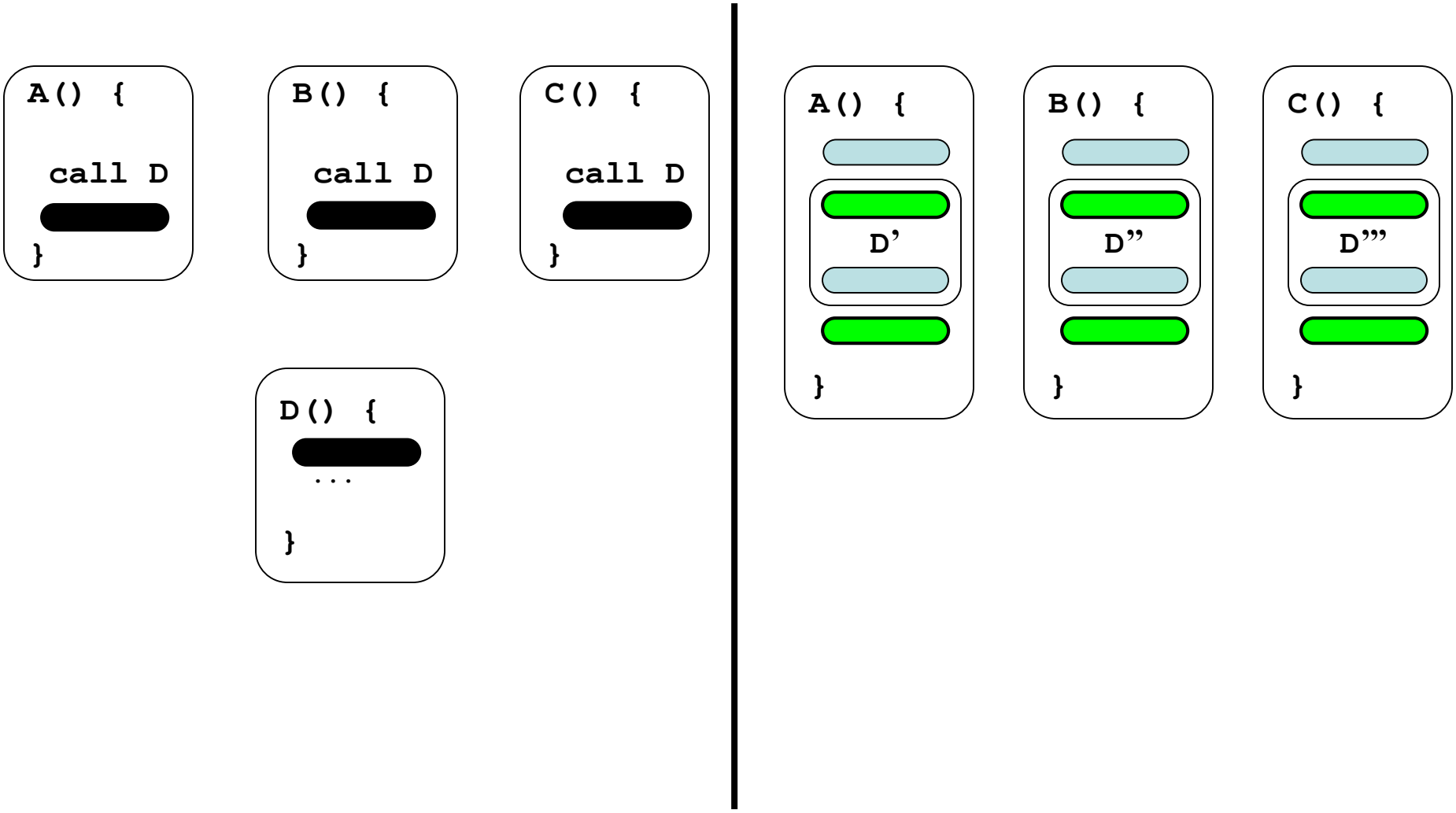
```
main() {
    x := new A(...);
    y := x.g1();
    y.f();

    x := new A(...);
    y := x.g1();
    y.f();

    x := new B(...);
    y:= x.g2();
    y.f();
}
```

```
g1(x) {
    x.f(); // can now inline
    // lots of code
    return x;
}
g2(x) {
    x.f(); // can now inline
    // lots of code
    return x;
}

// but want to inline f
f(x@A) { ... }
f(x@B) { ... }
```

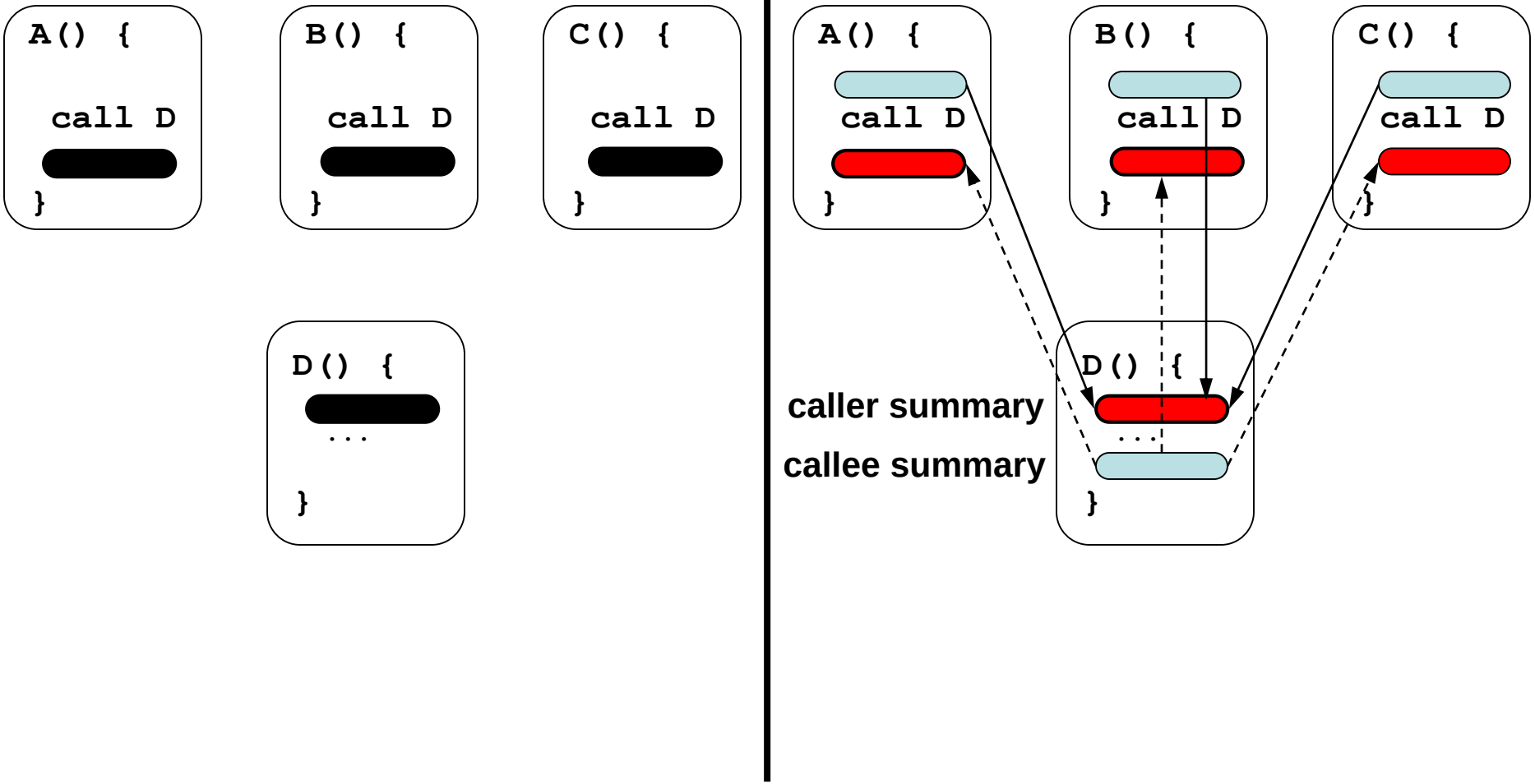
Recap using pictures



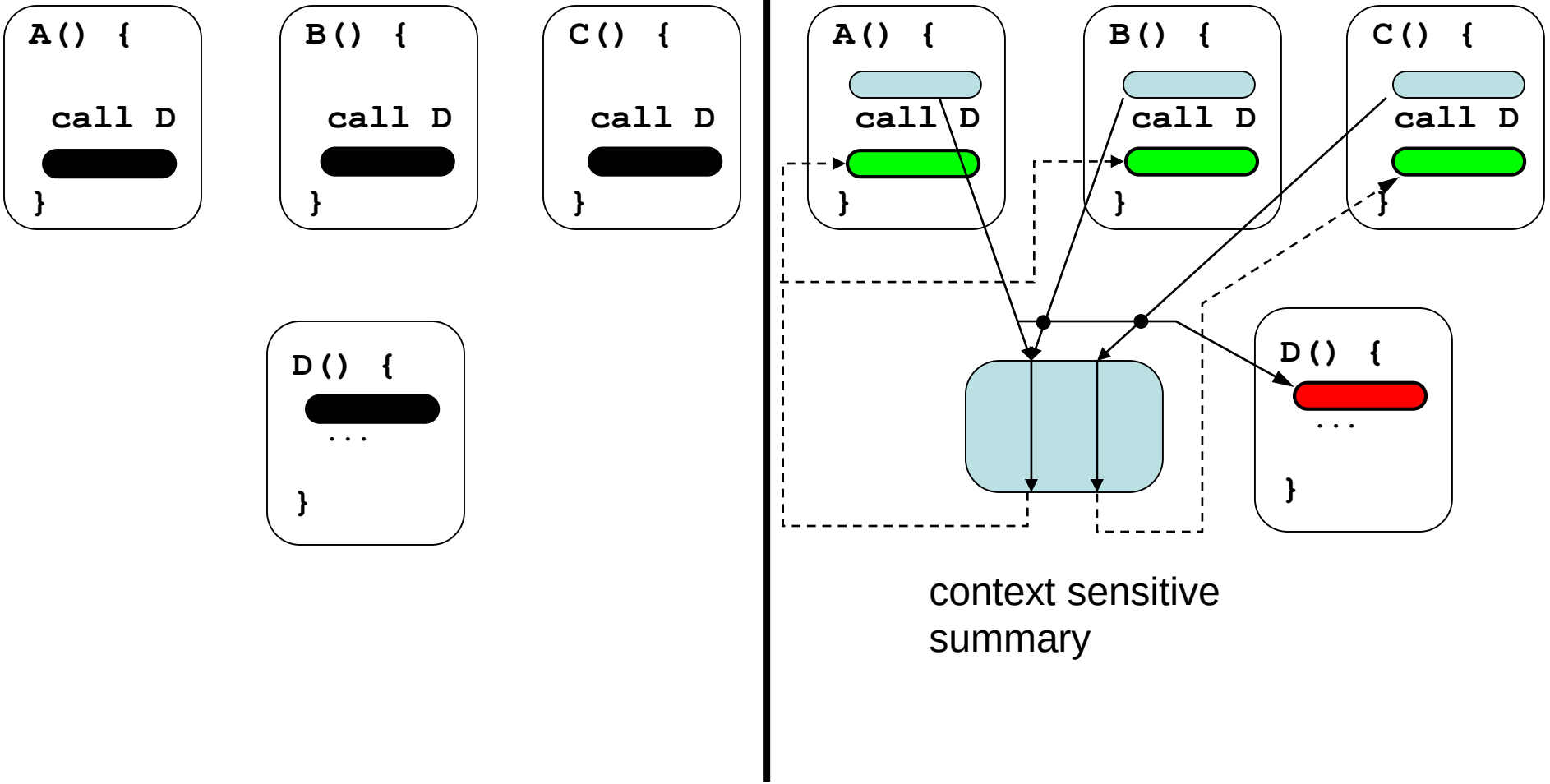
Inlining



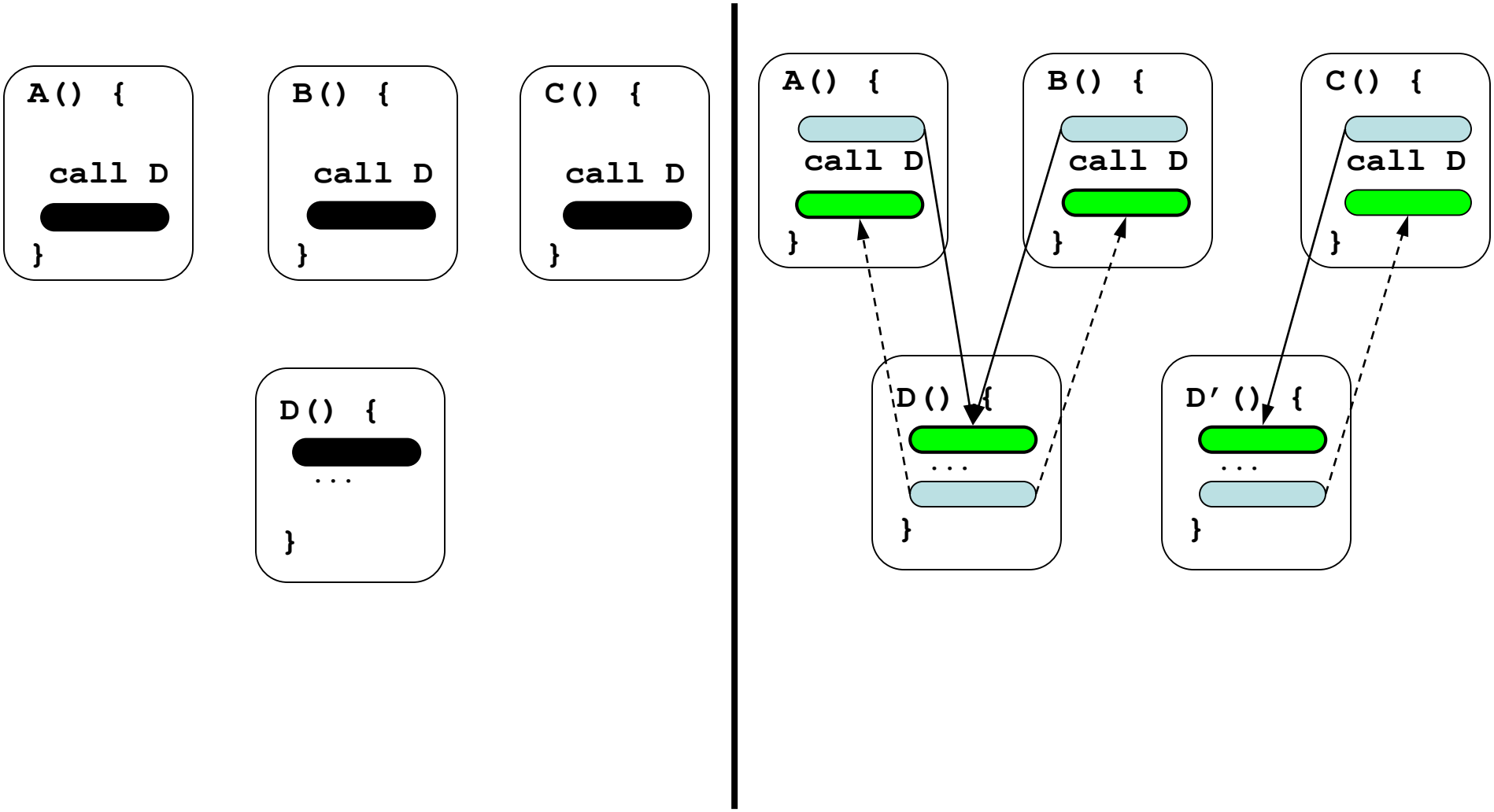
Context-insensitive summary



Context sensitive summary



Procedure Specialization



Comparison

	Caller precision	Callee precision	Code bloat
Inlining			
context-insensitive interproc			
Context sensitive interproc			
Specialization			

Comparison

	Caller precision	Callee precision	code bloat
Inlining	☺, because contexts are kept separate	☺, because contexts are kept separate	☹ may be large if we want to get the best precision
context-insensitive interproc	😐, because contexts are merged	😐, because contexts are merged	☺ none
Context sensitive interproc	☺, because of context sensitive summaries	😐, because contexts are still merged when optimizing callees	☺ none
Specialization	☺, contexts are kept separate	☺, contexts are kept separate	😐 Some, less than inlining

Summary on how to optimize function calls

- Inlining
- Tail call optimizations
- Interprocedural analysis using summaries
  - context sensitive
  - context insensitive
- Specialization

Cutting edge research

- Making interprocedural analysis scalable
- Optimizing first order function calls
- Making inlining effective in the presence of dynamic dispatching and class loading