Building your own compositional static analyzer with Infer.AI

Sam Blackshear, Dino Distefano, Jules Villard
Facebook
Roadmap

1 | Infer.AI architecture

2 | Building intraprocedural analyzers

3 | Building compositional interprocedural analyzers
Need scalable, incremental tools that are easy to extend
Need scalable incremental tools that are easy to extend

millions of lines of code
Need scalable, incremental tools that are easy to extend.

- Millions of lines of code
- 100K commits/week
Need scalable, incremental tools that are easy to extend

Millions of lines of code
100K commits/week
Small team of analysis experts
Recipe for a scalable/extensible analyzer

Procedure Summary

Frontend

Scheduler + results database

Analyzer Plugins

Procedure Summary

Program
Recipe for a scalable/extensible analyzer

Don't want to change

Scheduler + results database

Analyzer Plugins

Procedure Summary
Recipe for a scalable/extensible analyzer

Extensibility should live here

Languages

Bug Types

Procedure Summary

Analyzer Plugins

Scheduler + results database

Analyses

Analyses
Intraprocedural static analyzers are interpreters.
Intraprocedural static analyzers are interpreters

State\textsubscript{IN} \rightarrow \text{Interpreter} \rightarrow \text{Instructions} \rightarrow \text{State\textsubscript{OUT}}
Monolithic interpreters are hard to extend
Monolithic interpreters are hard to extend

New bug types

State\textsuperscript{IN}

Interpreter

Instructions

State\textsuperscript{OUT}
Monolithic interpreters are hard to extend

New analyses

New bug types

Interpreter

State_{IN}

State_{OUT}

Instructions
Monolithic interpreters are hard to extend

New analyses

New bug types

New languages

Interpreter

State\textsubscript{IN}

State\textsubscript{OUT}

Instructions
Separating instructions and commands

- if (e) { ...
- while (e) { ...
- try { ...
- x = y
- x = call m()
- x.f = y
- x = y.f
Separating instructions and commands

Instructions

if (e) { ...
while (e) { ...
try { ...
x = y
x = call m()
x.f = y
x = y.f

Command
Separating instructions and commands
Splitting the interpreter

State\textsubscript{IN} \quad \text{Command interpreter} \quad \text{State}\textsubscript{OUT} \quad \text{Command}

\begin{align*}
0101001 \\
0000001 \\
11111011 \\
11100100 \\
11010111 \\
01000011
\end{align*}
Splitting the interpreter

State\textsubscript{IN} → Command interpreter → Command

Command interpreter → State\textsubscript{OUT} → Control interpreter

CFG → Control interpreter

Command
Splitting the interpreter

State\textsubscript{IN} → Command interpreter → Command

Command interpreter → State\textsubscript{OUT} → State\textsubscript{OUT}

CFG → Control interpreter → State\textsubscript{OUT}
Splitting the interpreter
Generalizing to multiple paths

STATE
if(...) {
  command 1;
  STATE1
} else {
  command 2;
  STATE2
}
[???] command 3;
Generalizing to multiple paths

```java
STATE
if(...) {
    command 1;
    STATE_1
} else {
    command 2;
    STATE_2
}
[???]
command 3;
```
Generalizing to multiple paths

```c
STATE
if(...) {
    command 1;
    STATE1
} else {
    command 2;
    STATE2
}
[???]
cmmand 3;
```
Generalizing to multiple paths

```c
if(...) {
    command 1;
    STATE1
} else {
    command 2;
    STATE2
}

[???] command 3;
```
Putting it all together

Command interpreter

New bug types

State\textsubscript{IN}

DOMAIN

0 1 0 1 0 0 1
0 0 0 0 0 0 1
1 1 1 1 0 1 1
0 1 0 0 0 0 0
1 0 1 0 1 1
0 1 0 0 0 1

State\textsubscript{OUT}

DOMAIN

0 1 0 1 0 0 1
0 0 0 0 0 0 1
1 1 1 1 0 1 1
0 1 0 0 0 0 0
1 0 1 0 1 1
0 1 0 0 0 1

New analyses

CFG

Control interpreter

Command
Putting it all together

Command interpreter

New bug types

STATE

DOMAIN

New analyses

CFG

Control interpreter

STATE

DOMAIN

New bug types

STATE

DOMAIN

State

Out

Command

New languages?
Recipe for an scalable/extensible analyzer
Recipe for an scalable/extensible analyzer

Program -> Frontend -> Analyzer Plugins

Scheduler + results database

Procedure Summary
Frontend

Infer Intermediate Language

- Load
- Store
- Call
- Assume
Roadmap

1 | Infer.AI architecture

2 | Building intraprocedural analyzers

3 | Building compositional interprocedural analyzers
Roadmap

2. Building intraprocedural analyzers
- Domains and domain combinators
- Transfer functions
- Control-flow graphs
- Putting it all together
Extensible analysis architecture

Program → Frontend → Scheduler + results database

Analyzer Plugins

Procedure Summary
Extensible analysis architecture
Extensible analysis architecture

DOMIAN

0101001
0000001
1111011
0101001
0100001

StateIN

Transfer Functions

DOMIAN

0101001
0000001
1111011
0101000
1101011
0100001

StateOUT

Command

CFG

Abstract Interpreter
Extensible analysis architecture

Interpreter

Transfer Functions

Command

CFG

Abstract Interpreter
Abstract domains are simple (AbstractDomain.ml)

```ocaml
module type S = sig
  type astate

  (** the partial order induced by join *)
  val (<=) : lhs:astate -> rhs:astate -> bool

  val join : astate -> astate -> astate

  val widen : prev:astate -> next:astate -> num_iters:int -> astate

  val pp : F.formatter -> astate -> unit
end
```
Built-in domains: booleans

```plaintext
(** Boolean domain ordered by p || ~q. Useful when you want a boolean that's true only when it's true in both conditional branches. *)
module BooleanAnd : S with type astate = bool

(** Boolean domain ordered by ~p || q. Useful when you want a boolean that's true only when it's true in one conditional branch. *)
module BooleanOr : S with type astate = bool
```
Built-in domains: booleans

- Boolean domains

```plaintext
(** Boolean domain ordered by p || ~q. Useful when you want a boolean that's true only when it's true in both conditional branches. *)
module BooleanAnd : S with type astate = bool

(** Boolean domain ordered by ~p || q. Useful when you want a boolean that's true only when it's true in one conditional branch. *)
module BooleanOr : S with type astate = bool
```
Built-in domains: access paths

\[ x \in \text{Var} \]

\[ f \in \text{Fld} \]

\[
\begin{align*}
\text{AP} &::= x \mid \text{AP} . f \mid \text{AP} [e] \mid \text{AP} * \\
\text{e} &\in \hat{\text{Exp}} ::= \text{AP} \mid ... 
\end{align*}
\]

[Jones and Muchnick POPL '79 Flow analysis and optimization of LISP-like structures]
Built-in domains: access paths

\[ x \in \text{Var} \]
\[ f \in \text{Fld} \]
\[ AP ::= x \mid AP \cdot f \mid AP [e] \mid AP * \]
\[ e \in \text{Exp} ::= AP \mid ... \]

Examples:
\[ x \quad x.f \quad x[i].g \quad x.f.g \]
Built-in domains: access paths

\( x \in \text{Var} \)
\( f \in \text{Fld} \)

\[
\text{AP} ::= x \mid \text{AP} \cdot f \mid \text{AP} [e] \mid \text{AP} *
\]
\( e \in \hat{\text{Exp}} ::= \text{AP} \mid ... \)

- **Examples:**
  \( x \quad x.f \quad x[i].g \quad x.f.g \)

- **Concretization:** all addresses that may be read via given path at current program point

[Jones and Muchnick POPL '79 Flow analysis and optimization of LISP-like structures]
Built-in domains: access paths

\[ x \in Var \]
\[ f \in Fld \]
\[ AP ::= x \mid AP \cdot f \mid AP[e] \mid AP \ast \]
\[ e \in Exp ::= AP \mid \ldots \]

- Excellent domain for prototyping; simple, very close to concrete syntax
- Hard to handle aliasing well. Any two access paths can alias if the types of the last accesses are compatible:

\[ \text{type}(ap_1) <: \text{type}(ap_2) \lor \text{type}(ap_2) <: \text{type}(ap_1) \]

[Jones and Muchnick POPL '79 Flow analysis and optimization of LISP-like structures]
Built-in domains: access paths (AccessPath.ml)

```ocaml
module Raw : sig
    (** root var, and a list of accesses. closest to the root var is first that is, x.f.g is represented as (x, [f; g]) *)
    type t = base * access list [@@deriving compare]
end
```

```ocaml
type t =
| Abstracted of Raw.t (** abstraction of heap reachable from an access path, e.g. x.f* *)
| Exact of Raw.t (** precise representation of an access path, e.g. x.f.g *)
```
Built-in domains: access paths (AccessPath.ml)

- AccessPath.Raw.t (no length bounding)

```ocaml
module Raw : sig
  (** root var, and a list of accesses. closest to the root var is first that is, x.f.g is represented as (x, [f; g]) *)
type t = base * access list [@deriving compare]
```

```ocaml
type t =
  | Abstracted of Raw.t (** abstraction of heap reachable from an access path, e.g. x.f* *)
  | Exact of Raw.t (** precise representation of an access path, e.g. x.f.g *)
```
Built-in domains: access paths (AccessPath.ml)

- **AccessPath.Raw.t** (no length bounding)
  ```ml
  module Raw : sig
  (** root var, and a list of accesses. closest to the root var is first that is, x.f.g is represented as (x, [f; g]) *)
  type t = base * access list [@deriving compare]
  ```

- **AccessPath.t** (with length bounding)
  ```ml
  type t =
  | Abstracted of Raw.t (** abstraction of heap reachable from an access path, e.g. x.f* *)
  | Exact of Raw.t (** precise representation of an access path, e.g. x.f.g *)
  ```
Built-in domains: access paths (AccessPath.ml)

- **AccessPath.Raw.t** (no length bounding)

```
module Raw : sig
  (** root var, and a list of accesses. closest to the root var is first that is, x.f.g is represented as (x, [f; g]) *)
  type t = base * access list [@deriving compare]
```

- **AccessPath.t** (with length bounding)

```
type t =
  | Abstracted of Raw.t (** abstraction of heap reachable from an access path, e.g. x.f * )
  | Exact of Raw.t (** precise representation of an access path, e.g. x.f.g * )
```

- **AccessPathDomains.Set** (add-only set of paths w/ normalization)
Built-in domains: access tree
Built-in domains: access tree

- Trie where nodes are bases (at level 0) or accesses (at level $n > 0$)
Built-in domains: access tree

- Trie where nodes are bases (at level 0) or accesses (at level n > 0)

- Sparse representation of set of access paths, fast membership queries and....
Built-in domains: access tree

- Trie where nodes are bases (at level 0) or accesses (at level n > 0)

- E.g., \{ x.f, x.f.g, x.h^* \} =

- Sparse representation of set of access paths, fast membership queries and...
Built-in domains: access tree

```
module Make (TraceDomain : AbstractDomain.WithBottom) : S
```

```
  x
  /|
  / |
T0  T1
  /|
  / |
  f  h
  /|
  / |
  g  *  
```
Built-in domains: access tree

- Can associate abstract value with each node + look it up fast
Built-in domains: access tree

- Can associate abstract value with each node and look it up fast

Used in taint analysis to remember execution history for each memory location
Domain combinators facilitate building new domains

```plaintext
module FiniteSet (Element : PrettyPrintable.PrintableOrderedType)

module InvertedSet

module Map (Key : PrettyPrintable.PrintableOrderedType) (ValueDomain : S)

module InvertedMap
```
Domain combinators facilitate building new domains

- Powerset domains

```
module FiniteSet (Element : PrettyPrintable.PrintableOrderedType)
```

```
module InvertedSet
```

```
module Map (Key : PrettyPrintable.PrintableOrderedType) (ValueDomain : S)
```

```
module InvertedMap
```
Domain combinators facilitate building new domains

- **Powerset domains**

  ```
  module FiniteSet (Element : PrettyPrintable.PrintableOrderedType)
  module InvertedSet
  ```

- **Map domains**

  ```
  module Map (Key : PrettyPrintable.PrintableOrderedType) (ValueDomain : S)
  module InvertedMap
  ```
Domain combinators facilitate building new domains

(** Lift a pre-domain to a domain

module BottomLifted (Domain : S)

module TopLifted (Domain : S)

(** Cartesian product of two domains. *)

module Pair (Domain1 : S) (Domain2 : S) : S
Domain combinators facilitate building new domains

- Introducing dummy top/bottom values

```plaintext
(** Lift a pre-domain to a domain
module BottomLifted (Domain : S)

module TopLifted (Domain : S)

(** Cartesian product of two domains. *)
module Pair (Domain1 : S) (Domain2 : S) : S
```
Domain combinators facilitate building new domains

- Introducing dummy top/bottom values

```haskell
(** Lift a pre-domain to a domain
module BottomLifted (Domain : S)
```

```haskell
module TopLifted (Domain : S)
```

- Cartesian product

```haskell
(** Cartesian product of two domains. *)
module Pair (Domain1 : S) (Domain2 : S) : S
```
Control flow graphs (CFGs)
Control flow graphs (CFGs)

- Cfg module (Cfg.ml) is a collection of CFGs for every procedure in a file
Control flow graphs (CFGs)

- Cfg module (Cfg.ml) is a collection of CFGs for every procedure in a file
- ProcCfg module limits view to a single procedure (almost always what you want)
CFGs: customize view of control-flow (ProcCfg.ml)

(** Forward CFG with no exceptional control-flow *)
module Normal : S with type t = Procdesc.t

(** Forward CFG with exceptional control-flow *)
module Exceptional : S with type t = Procdesc.t

(** Wrapper that reverses the direction of the CFG *)
module Backward (Base : S) : S with type t = Base.t

module OneInstrPerNode (Base : S
CFGs: customize view of control-flow (ProcCfg.ml)

- With/without exceptional edges

```ml
(** Forward CFG with no exceptional control-flow *)
module Normal : S with type t = Procdesc.t

(** Forward CFG with exceptional control-flow *)
module Exceptional : S with type t = Procdesc.t

(** Wrapper that reverses the direction of the CFG *)
module Backward (Base : S) : S with type t = Base.t

module OneInstrPerNode (Base : S
```
CFGs: customize view of control-flow (ProcCfg.ml)

- With/without exceptional edges

```ocaml
(** Forward CFG with no exceptional control-flow *)
module Normal : S with type t = Procdesc.t

(** Forward CFG with exceptional control-flow *)
module Exceptional : S with type t = Procdesc.t
```

- Backward analysis

```ocaml
(** Wrapper that reverses the direction of the CFG *)
module Backward (Base : S) : S with type t = Base.t

module OneInstrPerNode (Base : S
```
CFGs: customize view of control-flow (ProcCfg.ml)

- With/without exceptional edges

```ocaml
(* Forward CFG with no exceptional control-flow *)
module Normal : S with type t = Procdesc.t

(* Forward CFG with exceptional control-flow *)
module Exceptional : S with type t = Procdesc.t
```

- Backward analysis

```ocaml
(* Wrapper that reverses the direction of the CFG *)
module Backward (Base : S) : S with type t = Base.t
```

- Changing granularity of blocks

```ocaml
module OneInstrPerNode (Base : S)
```
Transfer functions (TransferFunctions.ml)

```ocaml
module type S = sig
  module CFG : ProcCfg.S

(** abstract domain whose state we propagate *)
module Domain : AbstractDomain.S

(** read-only extra state (results of previous analyses, globals, etc.) *)
type extras

(** type of the instructions the transfer functions operate on *)
type instr

(** {A} instr {A'}. [node] is the node of the current instruction *)
val exec_instr : Domain.astate -> extras ProcData.t -> CFG.node -> instr -> Domain.astate
end

module type MakeSIL = functor (C : ProcCfg.S) -> sig
  include (SIL with module CFG = C)
end

module type MakeHIL = functor (C : ProcCfg.S) -> sig
  include (HIL with module CFG = C)
end
```
Putting it all together: simple liveness analysis
(Liveness.ml)
Putting it all together: simple liveness analysis (Liveness.ml)

```ml
module TransferFunctions (CFG : ProcCfg.S) = struct

module CFG = CFG

module Domain = AbstractDomainFINITESET(Var)

let extras = ProcData.no_extras

let exec_instr astate _ = function
  | Sil.Load (lhs_id, rhs_exp, _, _) ->
    Domain.remove (Var.of_id lhs_id) astate
    |> exp_add_live rhs_exp
  | Sil.Store (Lvar lhs_pvar, _, rhs_exp, _) ->
    let astate' =
      if Pvar.is_global lhs_pvar
      then astate (* never kill globals *)
      else Domain.remove (Var.of_pvar lhs_pvar) astate in
    exp_add_live rhs_exp astate'

module Analyzer =
```
Putting it all together: simple liveness analysis

(Liveness.ml)

```ocaml
module TransferFunctions (CFG : ProcCfg.S) = struct
    module CFG = CFG
    module Domain = AbstractDomain.FiniteSet(Var)
    type extras = ProcData.no_extras

    let exec_instr astate _ _ = function
        | Sil.Load (lhs_id, rhs_exp, _, _) ->
            Domain.remove (Var.of_id lhs_id) astate
            |> exp_add_live rhs_exp
        | Sil.Store (Lvar lhs_pvar, _, rhs_exp, _) ->
            let astate' =
                if Pvar.is_global lhs_pvar
                then astate (* never kill globals *)
                else Domain.remove (Var.of_pvar lhs_pvar) astate in
            exp_add_live rhs_exp astate'

    module Analyzer =
```
Analyzing procedures (AbstractInterpreter.ml)

(** compute and return invariant map for the given procedure starting from [initial] *)
val exec_pdesc :
  TransferFunctionsextras ProcData.t -> initial:TransferFunctions.Domain.asterate -> invariant_map

(** compute and return the postcondition for the given procedure starting from [initial]. If [debug] is true, print html debugging output. *)
val compute_post :
  ?debug:bool ->
  TransferFunctionsextras ProcData.t ->
  initial:TransferFunctions.Domain.asterate ->
  TransferFunctions.Domain.asterate option
Analyzing procedures (AbstractInterpreter.ml)

- Get invariant map from node id -> abstract state

```ml
val exec_pdesc: TransferFunctionsextras ProcData.t \rightarrow initial:TransferFunctionsDomain.astroate \rightarrow invariant_map
```

```ml
val compute_post: ?debug:bool \rightarrow TransferFunctionsextras ProcData.t \rightarrow initial:TransferFunctionsDomain.astroate \rightarrow TransferFunctionsDomain.astroate option
```
Analyzing procedures (AbstractInterpreter.ml)

- Get invariant map from node id -> abstract state

```ml
(val exec_pdesc :
 TransferFunctions.extras ProcData.t -> initial:TransferFunctions.Domain.astate -> invariant_map)
```

- Just grab the postcondition

```ml
(val compute_post :
 ?debug:bool ->
 TransferFunctions.extras ProcData.t ->
 initial:TransferFunctions.Domain.astate ->
 TransferFunctions.Domain.astate option)
```
Hooking up your checker (RegisterCheckers.ml)

```ml
module Analyzer = AbstractInterpreter.Make (CFG) (TransferFunctions)

let analyze_procedure { Callbacks.proc_desc; tenv; } =
  let post = Analyzer.compute_post ~initial:Domain.initial (ProcData.make proc_desc tenv) in
  report post

let checkers = [
  "annotation reachability", Config.annotation_reachability,
  [Procedure AnnotationReachability.checker, Config.Java];
  "bi abduction", Config.bi abduction,
  [Procedure Interproc.analyze_procedure, Config.Clang;
   Procedure Interproc.analyze_procedure, Config.Java];
  "your checker name", Config.y our_checker_CLI_flag,
  [ (* your checker entrypoint, your supported languages *) ]
];
```
Hooking up your checker (RegisterCheckers.ml)

- Define entrypoint for analyzing single procedure

```ocaml
module Analyzer = AbstractInterpreter.Make (CFG) (TransferFunctions)

let analyze_procedure { Callbacks.proc_desc; tenv; } =
  let post = Analyzer.compute_post ~initial:Domain.initial (ProcData.make proc_desc tenv) in
  report post

let checkers = [
  "annotation reachability", Config.annotation_reachability,
  [Procedure AnnotationReachability.checker, Config.Java];
  "biabduction", Config.biabduction,
  [Procedure Interproc.analyze_procedure, Config.Clang;
  Procedure Interproc.analyze_procedure, Config.Java];
  "your checker name", Config.your_checker_CLI_flag,
  [ (* your checker entrypoint, your supported languages *) ]
];
```
Hooking up your checker (RegisterCheckers.ml)

- Define entrypoint for analyzing single procedure

```ocaml
module Analyzer = AbstractInterpreter.Make (CFG) (TransferFunctions)

let analyze_procedure { Callbacks.proc_desc; tenv; } =
  let post = Analyzer.compute_post ~initial:Domain.initial (ProcData.make proc_desc tenv) in
  report post
```

- Add entrypoint to RegisterCheckers module

```ocaml
let checkers = [
  "annotation reachability", Config.annotation_reachability,
  [Procedure AnnotationReachability.checker, Config.Java];
  "biabduction", Config.biabduction,
  [Procedure Interproc.analyze_procedure, Config.Clang;
    Procedure Interproc.analyze_procedure, Config.Java];
  "your checker name", Config.your_checker_CLI_flag,
  [ (* your checker entrypoint, your supported languages * ) ];
]`
Roadmap

1 | Infer.AI architecture

2 | Building intraprocedural analyzers

3 | Building compositional interprocedural analyzers
Roadmap

- Summaries
- Bottom-up modular/compositional analysis
- Real-world case study: thread-safety analysis
- Designing compositional domains

3 Building compositional interprocedural analyzers
Bottom up modular/compositional analysis

- Compute call graph, do topological sort
- Analyze each procedure once using reverse postorder scheduling
- Break call cycles by iterating to fixed point
Bottom up modular/compositional analysis

- Compute call graph, do topological sort
- Analyze each procedure once using reverse postorder scheduling
- Break call cycles by iterating to fixed point
Bottom up modular/compositional analysis

- Compute call graph, do topological sort
- Analyze each procedure once using reverse postorder scheduling
- Break call cycles by iterating to fixed point
Bottom up modular/compositional analysis

- Compute call graph, do topological sort
- Analyze each procedure once using reverse postorder scheduling
- Break call cycles by iterating to fixed point
Bottom up modular/compositional analysis

- Compute call graph, do topological sort
- Analyze each procedure once using reverse postorder scheduling
- Break call cycles by iterating to fixed point
Bottom up modular/compositional analysis

- Compute call graph, do topological sort
- Analyze each procedure once using reverse postorder scheduling
- Break call cycles by iterating to fixed point
Bottom up modular/compositional analysis

- Compute call graph, do topological sort
- Analyze each procedure once using reverse postorder scheduling
- Break call cycles by iterating to fixed point
Why modular + compositional definitions

Modular: analyze one procedure (+ deps) at a time
Why modular + compositional definitions

Modular: analyze one procedure (+ deps) at a time

Compositional: summary for a procedure can be used in all calling contexts
Why modular + compositional definitions

Modular: analyze one procedure (+ deps) at a time
   No global view

Compositional: summary for a procedure can be used in all calling contexts
Why modular + compositional definitions

Modular: analyze one procedure (+ deps) at a time
No global view

Compositional: summary for a procedure can be used in all calling contexts
Never need to reanalyze procedure in new context
Why modular + compositional matters

- Scalable: linear in the number of procedures
- Incremental: easy to transition from-scratch analysis
  -> diff analysis
- Extensible: for new analysis, just need new domain + transfer functions
Constraints of bottom-up analysis

- Will have summary for callee P6
- But don't know anything about callers P2, P3
- Need to compute summary usable in any calling context
Constraints of bottom-up analysis

- Will have summary for callee P6
- But don't know anything about callers P2, P3
- Need to compute summary usable in any calling context
Compositionality and modularity challenges

1. How do we combine the callee summary with the current state? (compositionality)

2. How do we represent state from the caller during analysis? (modularity)
Brief detour into related work: modular/compositional analysis

- "Symbolic relational separate analysis", introduced in [Cousot and Cousot Static determination of dynamic properties of recursive procedures IFIP '77, Modular static program analysis CC '02]
Brief detour into related work: modular/compositional analysis

- Lots of papers use this approach for one kind of analysis or another (too many to list here, just chase reverse refs of Cousot paper)
- But few general guidelines for designing modular/compositional domains...
Brief detour into related work: modular/compositional analysis

- **Generating Precise and Concise Procedure Summaries** [Yorsh et al. POPL '08] shows how to design domains yielding summaries that compose efficiently and precisely.
- Complex domains assume existence of global points-to analysis...
Brief detour into related work: modular/compositional analysis

- Infer.AI doesn't impose any structure on summaries or provide automatic summary instantiation

- Makes it easy to experiment with different ideas

- Informal tips on domain/summary design later in talk
Interprocedural analysis: defining summaries (Specs.ml)

type payload =
{
  preposts : NormSpec.t list option; (** list of specs *)
  typestate : unit TypeState.t option; (** final typesta
  annot_map: AnnotReachabilityDomain.astate option; (**
  crashcontext_frame: Stacktree_j.stacktree option;
  (** Procedure location and blame_range info for crash
  quandary : QuandarySummary.t option;
  resources : ResourceLeakDomain.summary option;
  siof : SiofDomain.astate option;
  threadsafety : ThreadSafetyDomain.summary option;
  buffer_overrun : BufferOverrunDomain.Summary.t option;
  (* Your summary here *)
}

module Summary = Summary.Make (struct
  type payload = ThreadSafetyDomain.summary

  let update_payload post (summary : Specs.summary) =
    { summary with payload = { summary.payload with threadsafety = Some post } }

  let read_payload (summary : Specs.summary) =
    summary.payload.threadsafety
end)
Interprocedural analysis: defining summaries (Specs.ml)

- Add your summary type to master summary "payload"

```ocaml
type payload =
{
  preposts : NormSpec.t list option; (** list of specs *
  typestate : unit TypeState.t option; (** final typest
  annot_map: AnnotReachabilityDomain.astate option; (**
  crashcontext_frame: Stacktree_j.stacktree option;
  (** Procedure location and blame_range info for crash
  quandary : QuandarySummary.t option;
  resources : ResourceLeakDomain.summary option;
  siof : SiofDomain.astate option;
  threadsafety : ThreadSafetyDomain.summary option;
  buffer_overrun : BufferOverrunDomain.Summary.t option;
  (* Your summary here *)
}
```

```ocaml
module Summary = Summary.Make (struct
  type payload = ThreadSafetyDomain.summary

  let update_payload post (summary : Specs.summary) =
    { summary with payload = { summary.payload with threadsafety = Some post }}

  let read_payload (summary : Specs.summary) =
    summary.payload.threadsafety
end)
```
Interprocedural analysis: defining summaries (Specs.ml)

- Add your summary type to master summary "payload"

```ocaml
type payload =
{
  preposts : NormSpec.t list option; (** list of specs */
typestate : unit TypeState.t option; (** final typesta
annot_map: AnnotReachabilityDomain.astate option; (**
crashcontext_frame : Stacktree_j.stacktree option;
(** Procedure location and blame_range info for crash
quandary : QuandarySummary.t option;
resources : ResourceLeakDomain.summary option;
siof : SiofDomain.astate option;
threadsafty : ThreadSafetyDomain.summary option;
buffer_overrun : BufferOverrunDomain.Summary.t option;
(* Your summary here *)
}
```

- Define helper module for updating/reading payload with your summary

```ocaml
module Summary = Summary.Make (struct
  type payload = ThreadSafetyDomain.summary

  let update_payload post (summary : Specs.summary) =
  { summary with payload = { summary.payload with threadsafety = Some post } }

  let read_payload (summary : Specs.summary) =
  summary.payload.threadsafety
end)
```
Interprocedural analysis: storing summaries

```
let checker { summary; proc_desc; tenv; } : Specs.summary =
match Analyzer.compute_post (ProcData.make_default proc_data tenv) ~initial with
| Some post ->
  report post: Summary.update_summary (convert_to_summary post) master_summary
```
Interprocedural analysis: storing summaries

1. Convert postcondition to a summary (can be same)
Interprocedural analysis: storing summaries

1. Convert postcondition to a summary (can be same)
2. Call Summary.update_summary
Interprocedural analysis: using summaries

```haskell
match instr with
  | Call (return_opt, Direct callee_procname, actuals, _, _) ->
    begin
      match Summary.read_summary callee_procname with
        | Some summary ->
          (* Looked up the summary for callee_procname... do something with it *)
        | None ->
          (* No summary for callee_procname; it's native code or missing for some reason *)
    end
```
Interprocedural analysis: using summaries

- In transfer functions, just grab summary and use it
Roadmap

- Summaries
- Bottom-up modular/compositional analysis
- **Real-world case study: thread-safety analysis**
- Designing compositional domains

3 | Building compositional interprocedural analyzers
Who wants concurrency analysis?

Litho: A declarative UI framework for Android

GET STARTED  LEARN MORE  TUTORIAL
Who wants concurrency analysis?

Litho: A declarative UI framework for Android

Asynchronous layout

Litho can measure and layout your UI ahead of time without blocking the UI thread. By decoupling its layout system from the traditional Android View system, Litho can drop the UI thread constraint imposed by Android.
Litho: framework for building Android UI

Litho Component

- Fetch data
- Measure/Layout
- Draw

Talk to network
Determine size and position
Render and attach
Improve performance by moving layout to background

UI thread

Background thread(s)

Fetch data
Measure/Layout
Draw
Improve performance by moving layout to background

UI thread

Background thread(s)

Measure/Layout step needs to be thread-safe
Requirements for thread-safety analysis

Interprocedural
Requirements for thread-safety analysis

Interprocedural

Low annotation burden
Requirements for thread-safety analysis

Interprocedural

Low annotation burden

Modular

Compositional

Clang 5 documentation

THREAD SAFETY ANALYSIS

Acquiring and releasing locks:

- LOCKABLE
- EXCLUSIVE_LOCK_FUNCTION, SHARED_LOCK_FUNCTION
- EXCLUSIVE_TRYLOCK_FUNCTION, SHARED_TRYLOCK_FUNCTION
- UNLOCK_FUNCTION

Guarded data:

- GUARDED_BY, PT_GUARDED_BY

Guarded methods:

- EXCLUSIVE_LOCKS_REQUIRED, SHARED_LOCKS_REQUIRED
- LOCKS_EXCLUDED

Deadlock detection:

- ACQUIRED_BEFORE, ACQUIRED_AFTER

And a few misc. hacks...
How to trigger analysis: just add @ThreadSafe

```java
@ThreadSafe // checks all methods, subclasses
class A {
    void foo(B b) {
        b.m(); // all callees checked too
    }
}
```
How to trigger analysis: just add `@ThreadSafe`

```java
@ThreadSafe // checks all methods, subclasses
class A {
    void foo(B b) {
        b.m(); // all callees checked too
    }
}

class C {
    Obj mField;

    @ThreadSafe // checks method and all callees
    synchronized void bar() { mField = ... }

    void baz() { mField = ... } // also checked, will warn
}
```
How to trigger analysis: just add `@ThreadSafe`

```java
@ThreadSafe // checks all methods, subclasses
class A {
    void foo(B b) {
        b.m(); // all callees checked too
    }
}

class C {
    Obj mField;

    @ThreadSafe // checks method and all callees
    synchronized void bar() { mField = ... }

    void baz() { mField = ... } // also checked, will warn
}

@ThreadSafe(enableChecks = false) class D {} // won't warn
```
Infer thread-safety analysis: what should it do?

Find data races:
- two simultaneous accesses to the same memory location where at least one is a write.
Report data races with two warning types

- Write outside sync
- Memory
- Unprotected write warning (self-race)
Report data races with two warning types

- **Write outside sync**
  - Memory
  - Unprotected write warning (self-race)

- **Read/Write race**
  - Read
  - Write
  - Memory
  - Read/write race warning
Minimum viable analysis

- Analysis triggered by @ThreadSafe annotation
- Assume all non-private methods in a single @ThreadSafe class can run in parallel
- Report full call stack to any field accessed outside of synchronization
How does it work?

1. Stack trace to access
2. Lock(s) held
3. Current thread
4. Ownership info
Aggregate summaries for class and report

class C {
    public void m1() { ... }
    public void m2() { ... }
    private void m3() { ... }
}
Aggregate summaries for class and report

```java
class C {
    public void m1() { ... }
    public void m2() { ... }
    private void m3() { ... }
}
```

Report when:
- reachable from non-private method
- can find conflicting access(es)
Start with a very simple domain

Need to track:
- Name, location of accessed field. Use access paths
- Locks. Use boolean for "must be held"
- Threads. Use boolean for "on main thread"
Computing summaries: simple intraprocedural case

```java
private void setF(Obj o) {
    o.f = ... // line 1
}
summ: { (o.f, 1) }
```
Computing summaries: simple intraprocedural case

```java
private void setF(Obj o) {
    o.f = ... // line 1
}
summ: { (o.f, 1) }
```

```java
void setFWithSync(Obj o) {
    synchronized(o) {
        lockHeld
        o.f = ...;
    }
}
summ: { }
```
Applying summaries

```java
private void setF(Obj o) {
    o.f = ... // line 1
}
summ: { (o.f, 1, _) }
```

```java
private void callSetF(Obj x) {
    x.g = ... // line 2
    { (x.g, 2, _) }
    setF(x); // summ: { (o.f, 1, setF) }
    { (x.g, 2, _) } |_| project(summ, x) }
} 
summ: { (x.g, 2, _), (x.f, 1, setF) }
```
Applying summaries

```java
private void setF(Obj o) {
    o.f = ... // line 1
}
summ: { (o.f, 1, _) }
```

```java
private void callSetF(Obj x) {
    x.g = ... // line 2
    { (x.g, 2, _) }
    setF(x); // summ: { (o.f, 1, setF) }
    { (x.g, 2, _) } |_| project(summ, x) }
}
summ: { (x.g, 2, _), (x.f, 1, setF) }
```

*project* binds callee formals to caller actuals
Applying summaries with join loses call stack

```java
private void setF(Obj o) {
    o.f = ... // line 1
}
summ: { (o.f, 1, _) }
```

```java
private void callSetF(Obj x) {
    x.g = ... // line 1
    setF(x); // summ: { (o.f, 1, setF) }
    someOtherFunction1()
}
summ: { (x.f, 1, setF), (x.g, 2, callSetF) }
```
Applying summaries with join loses call stack

```java
private void setF(Obj o) {
    o.f = ... // line 1
}
summ: { (o.f, 1, _) }

private void callSetF(Obj x) {
    x.g = ... // line 1
    setF(x); // summ: { (o.f, 1, setF) }
    someOtherFunction1()
}
summ: { (x.f, 1, setF), (x.g, 2, callSetF) }

@ThreadSafe public void reportHere(Obj y) {
    callSetF(y); // summ: { (x.f, 1, setF), ... }
    someOtherFunction2()
}
summ: { (y.f, 1, setF), (y.g, 2, callSetF) }
```
Applying summaries with join loses call stack

```java
private void setF(Obj o) {
    o.f = ... // line 1
}
summ: { (o.f, 1, _) }

private void callSetF(Obj x) {
    x.g = ... // line 1
    setF(x); // summ: { (o.f, 1, setF) }
    someOtherFunction1()
}
summ: { (x.f, 1, setF), (x.g, 2, callSetF) }

@ThreadSafe public void reportHere(Obj y) {
    callSetF(y); // summ: { (x.f, 1, setF), ... }
    someOtherFunction2()
}
summ: { (y.f, 1, setF), (y.g, 2, callSetF) }

Can't recover call stack!
private void setF(Obj o) {
    o.f = ... // line 1
} 
summ: { (o.f, [(1, _)]) }

private void callSetF(Obj x) {
    setF(x); // line 2
    summ: { (o.f, [(1, _)]) }
    { } |_| (2, _) :: project(summ, x)
    someOtherFunction1();
} 
summ: { (x.f, [(2, _) :: (1, setF)]) }
private void setF(Obj o) {
    o.f = ... // line 1
}
summ: { (o.f, [(1, _)]) }

private void callSetF(Obj x) {
    setF(x); // line 2 summ: { (o.f, [(1, _)]) }
    { } |_| (2, _) :: project(summ, x)
    someOtherFunction1();
}
summ: { (x.f, [(2, _) :: (1, setF)]) }

public void publicMethod(Obj y) {
    callSetF(y); // line 3
    someOtherFunction2();
}
summ: { (y.f, [(3, _) :: (2, callSetF) :: (1, setF)]) }
Explicit call stack tracking bloats summaries

```java
private void setF(Obj o) {
    o.f = ... // line 1
    o.g = ...
}
summ: { (o.f, [(1, _)]),
         o.g, [(2, _)] }
```
Explicit call stack tracking bloats summaries

```java
private void setF(Obj o) {
  o.f = ... // line 1
  o.g = ...
}
summ: { (o.f, [(1, _)]),
         o.g, [(2, _)] }
```

```java
private void callSetF(Obj x) {
  setF(x); // line 2
  someOtherFunction1();
}
summ: { (x.f, [(2, _) :: (1, setF)]),
        (x.g, [(2, _) :: (2, setF)])
```
Explicit call stack tracking bloats summaries

private void setF(Obj o) {
  o.f = ... // line 1
  o.g = ...
}
summ: { (o.f, [(1, _)]),
         o.g, [(2, _)] }

private void callSetF(Obj x) {
  setF(x); // line 2
  someOtherFunction1();
}
summ: { (x.f, [(2, _) :: (1, setF)],
         (x.g, [(2, _) :: (2, setF)}

public void publicMethod(Obj y) {
  callSetF(y); // line 3
  someOtherFunction2();
}
summ: { (y.f, [(3, _) :: (2, callSetF) :: (1, setF)],
         (y.g, [(3, _) :: (2, callSetF) :: (2, setF)] }
Visualization of summary size explosion
Visualization of summary size explosion
Visualization of summary size explosion

$P_{\text{MAIN}}$

1 + 2(1) = 3
Visualization of summary size explosion

\[ 1 + 2(1 + 1 + 3) = 10 \]

\[ 1 + 2(1) = 3 \]
Visualization of summary size explosion

1 + 2(1 + 1 + 3) = 10

1 + 2(3) = 7

1 + 2(1 + 1 + 3) = 10

1 + 2(3) = 7

1 + 2(1) = 3
Visualization of summary size explosion

\[ 1 + 2(10) = 20 \]

\[ 1 + 2(1 + 1 + 3) = 10 \]

\[ 1 + 2(1 + 1 + 3) = 10 \]

\[ 1 + 2(3) = 7 \]

\[ 1 + 2(10) = 20 \]

\[ 1 + 2(3) = 7 \]

\[ 1 + 2(1 + 1 + 3) = 10 \]

\[ 1 + 2(1) = 3 \]
Visualization of summary size explosion

1 + 2(10) = 20

1 + 2(1 + 1 + 3) = 10

1 + 2(1 + 1 + 3) = 10

1 + 2(20 + 7) = 55

1 + 2(3) = 7

1 + 2(1) = 3
Solution: track last call that leads to access OOS

private void setF(Obj o) {
  o.f = ... // line 1
  o.g = ...
}

summ: { o.f, (1, _),
        o.g, (2, _) }
private void setF(Obj o) {
  o.f = ... // line 1
  o.g = ...
}
summ: { o.f, (1, _),
         o.g, (2, _) }

private void callSetF(Obj o) {
  setF(o); // line 2
  someOtherFunction1();
}
summ: { (o.f, (2, setF),
         (o.g, (2, setF)}

Solution: track last call that leads to access OOS
private void setF(Obj o) {
    o.f = ... // line 1
    o.g = ...
}
summ: { o.f, (1, _),
        o.g, (2, _) }

private void callSetF(Obj o) {
    setF(o); // line 2
    someOtherFunction1();
} 
summ: { (o.f, (2, setF),
        (o.g, (2, setF)}

public void publicMethod(Obj o) {
    callSetF(o); // line 3
    someOtherFunction2();
} 
summ: { (o.f, (3, callSetF),
        (o.g, (3, callSetF) }
private void setF(Obj o) {
  o.f = ... // line 1
  o.g = ...
} 
summ: { o.f, (1, _),
        o.g, (2, _) }

private void callSetF(Obj o) {
  setF(o); // line 2
  someOtherFunction1();
} 
summ: { (o.f, (2, setF),
        (o.g, (2, setF)}

public void publicMethod(Obj o) {
  callSetF(o); // line 3
  someOtherFunction2();
} 
summ: { (o.f, (3, callSetF),
        (o.g, (3, callSetF)}
Solution: track last call that leads to access OOS

Recover call stack by unrolling summaries when reporting
1. How do we combine the callee summary with the current state? (compositionality)

2. How do we represent state from the caller during analysis? (modularity)
Mutating owned objects leads to false positives

Obj local = new Obj();
local.f = ... // safe write
global.g = ... // unsafe write
Mutating owned objects leads to false positives

```java
Obj local = new Obj();
local.f = ... // safe write

global.g = ... // unsafe write
```

```java
Obj objFactory() {
    return new Obj();
}

Obj local = objFactory();
local.f = ... // safe write
```
Mutating owned objects leads to false positives

```java
Obj local = new Obj();
local.f = ... // safe write
global.g = ... // unsafe write

Obj objFactory() {
    return new Obj();
}

Obj local = objFactory();
local.f = ... // safe write
```

False positives
Mutating owned objects leads to false positives

Obj local = new Obj();
local.f = ... // safe write
global.g = ... // unsafe write

Obj objFactory() {
  return new Obj();
}
Obj local = objFactory();
local.f = ... // safe write
Ownership can be conditional

```java
private void writeF(Obj a) {
    a.f = ...
}
Obj o = new Obj();
writeF(o); // safe
```
Ownership can be conditional

private void writeF(Obj a) {
    a.f = ...
}

Obj o = new Obj();
writeF(o); // safe

Builder setX(X x) {
    this.x = x;
    return this;
}

new Builder().setX(x).setY(y); // safe

global.set(X).f = 7; // not safe
Ownership can be conditional

```java
private void writeF(Obj a) {
  a.f = ...
}

Obj o = new Obj();
writeF(o); // safe
```

False positives

```java
Builder setX(X x) {
  this.x = x;
  return this;
}

new Builder().setX(x).setY(y); // safe
global.set(X).f = 7; // not safe
```
Ownership can be conditional

```java
private void writeF(Obj a) {
    a.f = ...;
}
Obj o = new Obj();
writeF(o); // safe
```

Safe if formal is owned by caller

```java
Builder setX(X x) {
    this.x = x;
    return this;
}
new Builder().setX(x).setY(y); // safe
```

False positives

```
new Builder().setX(x).setY(y); // safe
```

Returns ownership if formal is owned by caller

```
global.set(X).f = 7; // not safe
```
Track owned locals + owned return value

Obj local = new Obj();
owned(local), {}
local.f = ... // safe write
global.g = ... // unsafe write
owned(local), { (g, 3) }
Track owned locals + owned return value

Obj local = new Obj();
owned(local), {}
local.f = ... // safe write
global.g = ... // unsafe write
owned(local), { (g, 3) }

Obj objFactory() {
  return new Obj();
}

Obj local = objFactory();
owned(local)
local.f = ... // safe write
private void writeF(Obj a) {
    a.f = ... 
}
summ: { (a.f, 1) if ¬owned(a) }

Obj o = new Obj();
owned(o)
writeF(o);
owned(o) |_| project(summ, o)
owned(o) ^ { (a.f, 1) if ¬owned(o) }
owned(o) ^ {}
Need to track ownership in summaries

Builder setX(X x) {
    this.x = x;
    return this;
}

summ: { (this.x if ¬owned(this) } ^
        owned(ret) if owned(this)

owned(a)
Builder b = a.setX(x);

owned(a) ^ project(summ, b, a, x)

owned(a) ^ owned(b) if owned(a)
    ^ { (this.x if ¬owned(a) } 

owned(b) if owned(a)

owned(a) ^ owned(b) ^ {} 

b.setY(y); // safe by similar reasoning
Thread-safety analysis makes conversion faster/safer

- 100+ Litho components moved to background layout with very few crashes
- Analysis enabled for all Litho component diffs
- 300+ thread-safety regressions caught/fixed on diffs
Minimum viable analysis -> formalism + sound tool

- Boolean lock abstraction -> infer permissions associated with locks/threads (collaboration with UCL)
- Access paths -> separation logic
- Proof of soundness
- Transfer formalism into tool
Roadmap

- Summaries
- Bottom-up modular/compositional analysis
- Real-world case study: thread-safety analysis
- **Designing compositional domains**

3. Building compositional interprocedural analyzers
Compositionality and modularity challenges

1. How do we represent state from the caller during analysis? (modularity)
2. How do we combine the callee summary with the current state? (compositionality)
Modularity: representing state from the caller

\( x, y \in \text{Var} \)

\( e \in \text{Exp} ::= x \mid ... \)

\( c \in \text{Cmd} ::= e_1 = e_2 \mid y = \text{call } p(\bar{x}) \)
Modularity: representing state from the caller

\[ x, y \in Var \]

\[ e \in Exp ::= x \mid ... \]

\[ c \in Cmd ::= e_1 = e_2 \mid y = \text{call } p(\bar{x}) \]

\[ \hat{Val} ::= \hat{x} \mid FP(x) \]

Add ghost variable for "footprint" value
read from environment
Modularity: representing state from the caller

\[ \hat{Val} ::= \hat{x} \mid FP(x) \]

Add ghost variable for "footprint" value read from environment

\[
y \notin \text{dom}(\hat{\sigma}) \quad \hat{\sigma}' = \text{update}(x, \hat{\sigma}, FP(y)) \\
\{ \hat{\sigma} \} \quad x = y \quad \{ \hat{\sigma}' \}
\]
Modularity: representing state from the caller

\[ \hat{V}al ::= \hat{x} \mid FP(x) \]

Add ghost variable for "footprint" value read from environment

When we read a variable that isn't defined, introduce ghost variable

\[
y \notin \text{dom}(\hat{\sigma}) \quad \hat{\sigma}' = \text{update}(x, \hat{\sigma}, FP(y))
\]

\[
\{\hat{\sigma}\} \quad x = y \quad \{\hat{\sigma}'\}
\]
Modularity: representing state from the caller

\[ \hat{V}al ::= \hat{x} \mid FP(x) \]

Add ghost variable for "footprint" value read from environment

When we read a variable that isn't defined, introduce ghost variable

\[
y \notin \text{dom}(\hat{\sigma}) \quad \hat{\sigma}' = \text{update}(x, \hat{\sigma}, FP(y)) \quad \{ \hat{\sigma}' \}
\]

Easiest implementation: \[ \hat{\sigma}[\hat{x} \mapsto FP(y)] \]
Modularity: representing state from the caller

- Summaries are parameterized by footprint values
- Generic: fully context-insensitive, but each caller can fill in context when applying the summary
Modularity: representing state from the caller

- Summaries are parameterized by footprint values
- Generic: fully context-insensitive, but each caller can fill in context when applying the summary

```java
private void writeF(Obj a) {
    a.f = ...
}
summ: { (a.f, 1) if ¬owned(a) } =~
λ a. if owned(a) {} else { (a.f, 1) }
```
Modularity: representing state from the caller

\[ y \notin \text{dom}(\hat{\sigma}) \quad \hat{\sigma}' = \text{update}(x, \hat{\sigma}, FP(y)) \]

\{\hat{\sigma}\} \quad x = y \quad \{\hat{\sigma}'\}

- Use for formals, globals, field/array reads from env
- Used in bi-abduction analysis [Compositional shape analysis by means of bi-abduction, Calcagno et al. JACM '11]
- Useful in subsequent Infer analyses: thread-safety, Quandary taint analysis, ...
Compositionality and modularity challenges

1. How do we represent state from the caller during analysis? (modularity)

2. How do we combine the callee summary with the current state? (compositionality)
Compositionality: combining callee state with current state

\[ \hat{\sigma}_p : \text{summary for procedure } p \]

\[ \hat{\sigma}'_p = \text{project}(\bar{x}, y, \hat{\sigma}, \hat{\sigma}_p) \]

\[ \hat{\sigma}' = \hat{\sigma} \oplus \hat{\sigma}'_p \]

\[ \{ \hat{\sigma} \} \quad y = \text{call } p(\bar{x}) \quad \{ \hat{\sigma}' \} \]
Compositionality: combining callee state with current state

\( \hat{\sigma}_p : \text{summary for procedure } p \)

Replace footprint variables in summary with actuals
Bind return value from summary to return variable

\[
\hat{\sigma}'_p = \text{project}(\vec{x}, y, \hat{\sigma}, \hat{\sigma}_p) \quad \hat{\sigma}' = \hat{\sigma} \oplus \hat{\sigma}'_p
\]

\[
\{ \hat{\sigma} \} \quad y = \text{call } p(\vec{x}) \quad \{ \hat{\sigma}' \}
\]
Compositionality: combining callee state with current state

\[ \hat{\sigma}_p' = \text{project}(\bar{x}, y, \hat{\sigma}, \hat{\sigma}_p) \]
\[ \{\hat{\sigma}\} \quad y = \text{call} \; p(\bar{x}) \quad \{\hat{\sigma}'\} \]

\[ \hat{\sigma}' = \hat{\sigma} \oplus \hat{\sigma}_p' \]
Compositionality: combining callee state with current state

- Join for weak updates
- Append for traces
- Domain-specific operator for strong updates...

\[
\hat{\sigma}'_p = \text{project}(\vec{x}, y, \hat{\sigma}, \hat{\sigma}_p) \quad \hat{\sigma}' = \hat{\sigma} \oplus \hat{\sigma}'_p
\]

\[
\{ \hat{\sigma} \} \quad y = \text{call } p(\vec{x}) \quad \{ \hat{\sigma}' \}
\]
Example: interprocedural allocation counting

\[ \hat{\sigma} \in \mathbb{Nat} \cup \{ \top \} \]

Overapproximate number of allocated heap cells

\[ \{ \hat{\sigma} \} x = \text{malloc}(\ldots) \{ \hat{\sigma} + 1 \} \]
Example: interprocedural allocation counting

\[ \hat{\sigma} \in \text{Nat} \cup \{T\} \]
Example: interprocedural allocation counting

\[ \hat{\sigma} \in Nat \cup \{\top\} \]

\[ \text{project}(\vec{x}, y, \hat{\sigma}, \hat{\sigma}_p) = \hat{\sigma}_p \]
Example: interprocedural allocation counting

\[ \hat{\sigma} \in Nat \cup \{\top\} \]

\[ \text{project}(\vec{x}, y, \hat{\sigma}, \hat{\sigma}_p) = \hat{\sigma}_p \]

\[ \hat{\sigma} \oplus \hat{\sigma}_p = +\top \]
Example: interprocedural allocation counting

\[ \hat{\sigma} \in \text{Nat} \cup \{ \top \} \]

\[ \text{project}(\vec{x}, y, \hat{\sigma}, \hat{\sigma}_p) = \hat{\sigma}_p \]

\[ \hat{\sigma} \oplus \hat{\sigma}_p = + \top \]

We don't care about caller state or strong updates w.r.t callee. Easy.
Example: interprocedural escape analysis

\[ \hat{Val} ::= \hat{x} \mid FP(x) \]

\[ \hat{\sigma} \subseteq 2^{\hat{Val}} \]
Example: interprocedural escape analysis

\[ \hat{V}al ::= \hat{x} \mid FP(x) \]

\[ \hat{\sigma} \subseteq 2^{\hat{V}al} \]

Set of local variables holding addresses that may escape scope of current function
Example: interprocedural escape analysis

$$Val ::= \hat{x} \mid FP(x)$$

$$\hat{\sigma} \subseteq 2^{Val}$$

Set of local variables holding addresses that may escape scope of current function

<table>
<thead>
<tr>
<th>y is local</th>
<th>y is formal</th>
</tr>
</thead>
<tbody>
<tr>
<td>{\hat{\sigma}} x.f = y {\hat{\sigma} \cup {\hat{y}}}</td>
<td>{\hat{\sigma}} x.f = y {\hat{\sigma} \cup {FP(y)}}</td>
</tr>
</tbody>
</table>
Example: interprocedural escape analysis

\[ Val ::= \hat{x} \mid FP(x) \]

\[ \hat{\sigma} \subseteq 2^{\hat{Val}} \]

\[
\text{project}(\bar{x}, y, \hat{\sigma}, \hat{\sigma}_p) = \\
\bigcup_{x_i} \{\hat{x}_i\} \text{ if } FP(x_i) \in \hat{\sigma}_p \land x_i \text{ is local} \\
x_i \{FP(x_i)\} \text{ if } FP(x_i) \in \hat{\sigma}_p \land x_i \text{ is formal} \\
\{\} \text{ otherwise}
\]
Example: interprocedural escape analysis

\[
\hat{\text{Val}} ::= \hat{x} \mid FP(x)
\]

\[
\hat{\sigma} \subseteq 2^{\hat{\text{Val}}}
\]

\[
\text{project}(\vec{x}, y, \hat{\sigma}, \hat{\sigma}_p) =
\bigcup \{\hat{x}_i\} \text{ if } FP(x_i) \in \hat{\sigma}_p \land x_i \text{ is local}
\]

\[
x_i \{FP(x_i)\} \text{ if } FP(x_i) \in \hat{\sigma}_p \land x_i \text{ is formal}
\]

\[
\emptyset \text{ otherwise}
\]

\[
\hat{\sigma} \oplus \hat{\sigma}_p = \bigcup
\]
Incrementalizing modular + compositional analyses is easy

- Each summary is a function of its instructions + callee summaries
- Simple change propagation algorithm over call graph works
- Can piggyback on incremental build systems for free distributed cache
From-scratch analysis
From-scratch analysis
From-scratch analysis
From-scratch analysis
From-scratch analysis
From-scratch analysis
From-scratch analysis
From-scratch analysis

Go bottom-up, compute summary for all procedures.

Report all bugs found.
Incremental analysis: full

Change P2, P3

If P3 changes, need to re-analyze P1

If P1 or P2 changes, need to re-analyze PMain
Incremental analysis: full

Change P2, P3

Re-analyze P2, P3

If P3 changes, need to re-analyze P1

If P1 or P2 changes, need to re-analyze PMain
Incremental analysis: changed code only

Change P2, P3
Re-analyze P2, P3
Can stop there if we only care about reporting errors in P2, P3
Why modular + compositional matters

- Scalable: linear in the number of procedures
- Incremental: easy to transition from-scratch analysis
  -> diff analysis
- Extensible: for new analysis, just need new domain + transfer functions
Conclusion: try out your analysis ideas in Infer

- Frontends for Java, C, C++, Obj-C
- Framework for writing modular/compositional interprocedural analyses
- Your analyses can make real programmers happy

fbinfer.com/docs/absint-framework.html
Lab exercise: building your own compositional analyzer

github.com/facebook/infer/infer/src/labs/lab.md