On Optimally Combining Static and Dynamic Analyses For Intensional Program Properties

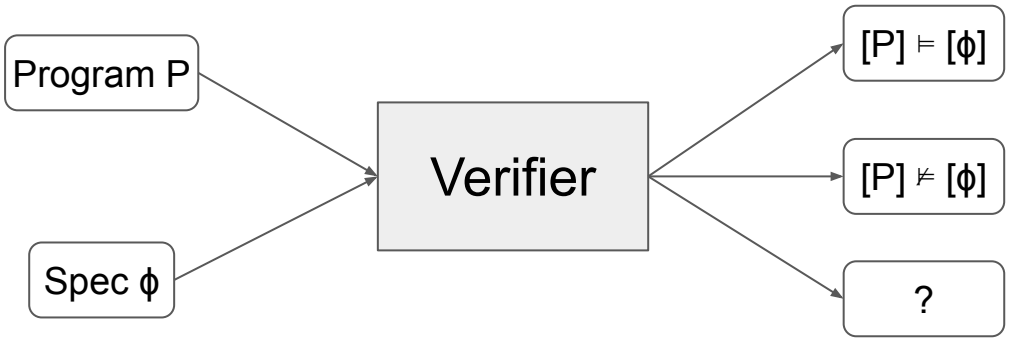
Ravi Mangal David Devecsery Alessandro Orso

Georgia Tech

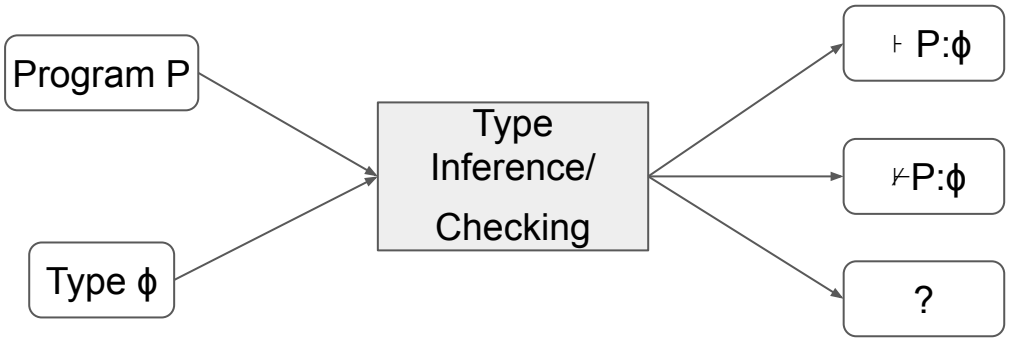
Motivation

Recipe for designing automated, efficient, sound, and complete program verification tools

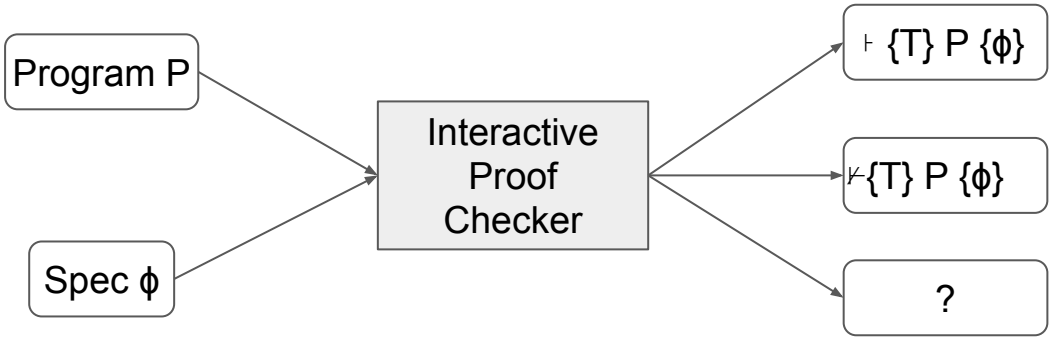
What is program verification?



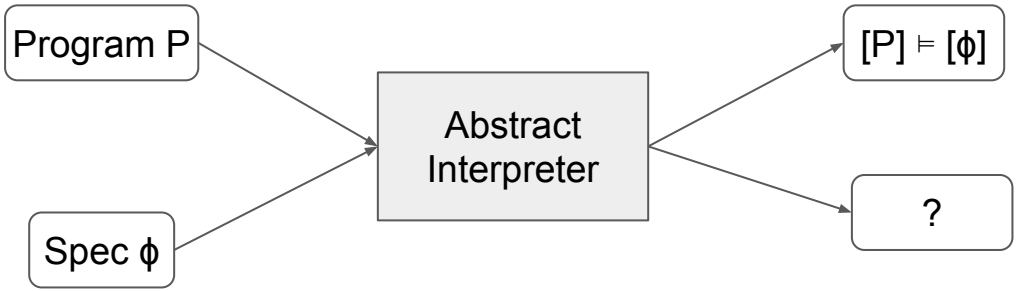
Static Verifiers - Type Systems



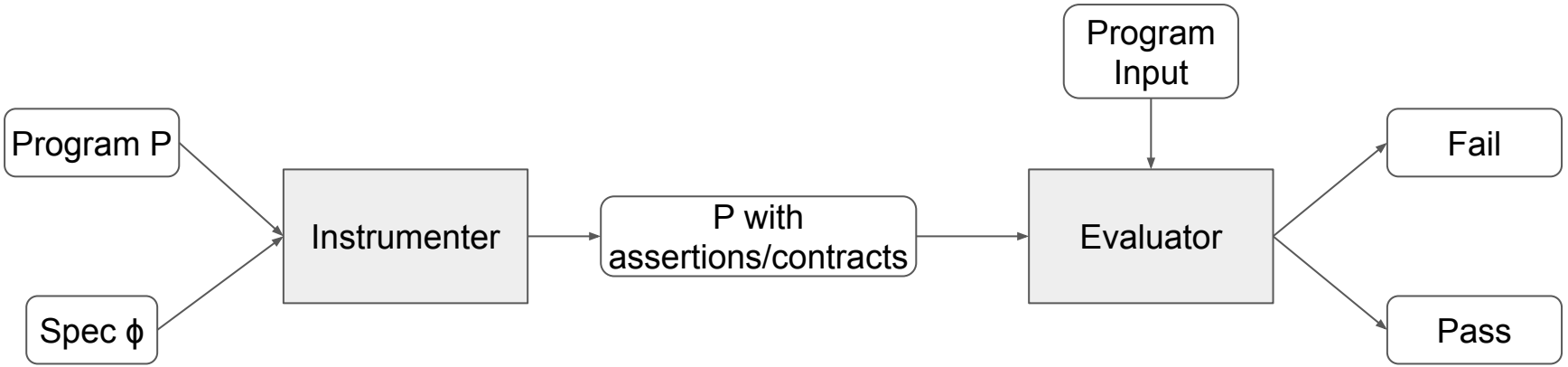
Static Verifiers - Program Logics



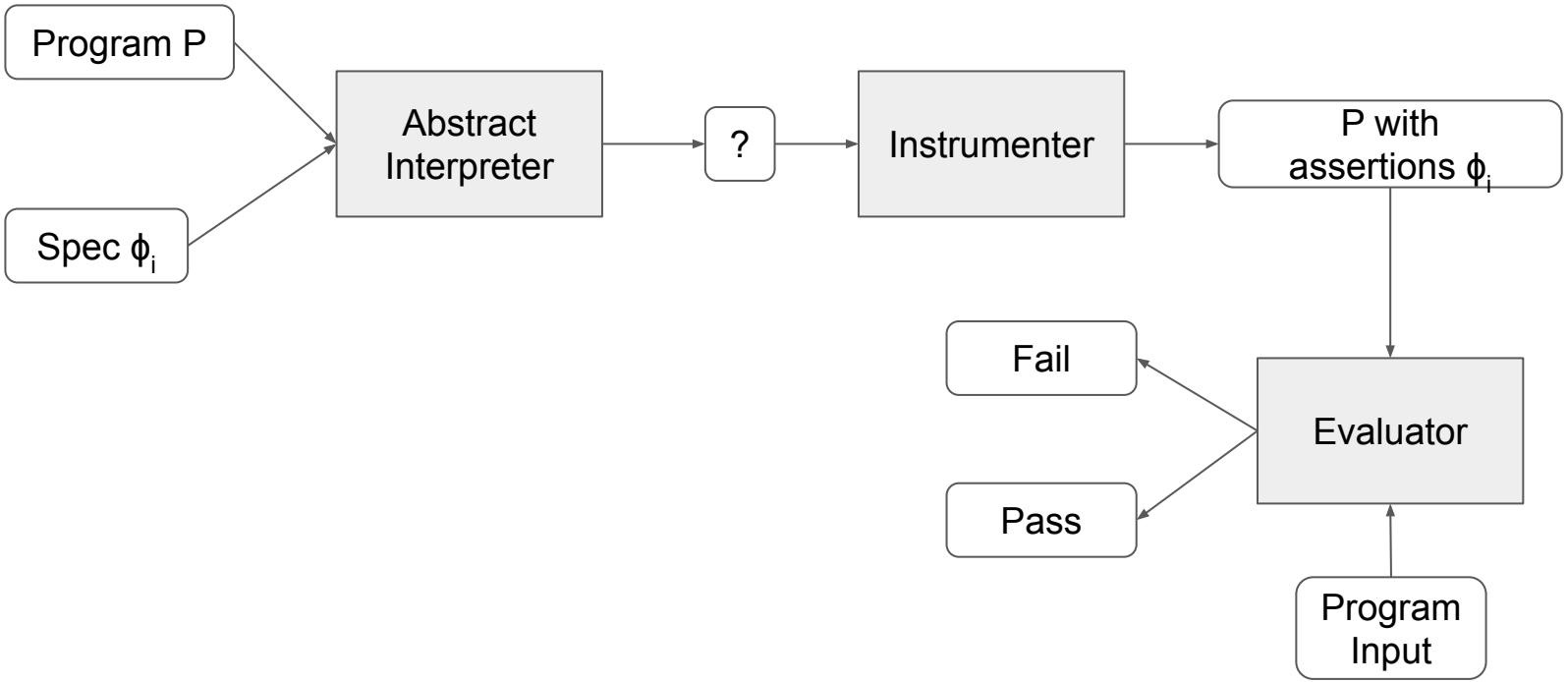
Static Verifiers - Program Analyses



Dynamic Verifiers



Let’s combine!



Are we done?

Yes, and no :(

- Yes, for ‘extensional’ or ‘local’ properties
  - Properties about the results of computations
  - Can be answered by observing the current program state

- No, for ‘intensional’ properties
  - Properties about how computations execute

Intensional Properties

- Track how computations execute
- Examples - taint tracking, datarace checking, program complexity, etc.
- Instrumented semantics - Augment the states of standard semantics with instrumentation data
  - Ex. <P,S> ⇝ <P’,S’> modified to <P,S,I> ⇝ <P’,S’,I’>

What is the problem?

- Combined verification only removes runtime assertions but not additional instrumentation
  - High overheads!

- No existing general framework for combining static and dynamic verifiers with a focus on efficiency

Idea!

- Remove “all” dynamic instrumentation “associated” with statically proven assertions

- Challenges:
  - How to communicate information from static to dynamic verifier?
  - How to formally define the notions of “all” and “associated” in the statement above?
  - How to automatically discover “all associated” instrumentation for a given assertion?

- Approach:
  - “Parameterize” static and dynamic verifiers
  - Notions of “all” and “associated” defined with respect to this parameterization

A taint analysis example

```
void main() {
    A o1 = src1();
    B o2 = src2();
    A o3 = foo(o1);
    B o4 = bar(o2);
    assert(o3 not tainted);
    sink1(o3);
    assert(o4 not tainted);
    sink2(o4);
}
```

```
A foo(A a) {
    .
    x = san(a);
    .
    return x;
}
```

```
B foo(B b) {
    .
    .
    return y;
}
```

An interlude: Some definitions

$\text{eval} : \text{State} \longrightarrow \text{State}$ (evaluator for the language)

$\text{abs\_int} : \text{Abs\_State} \longrightarrow \text{Abs\_State}$ (abstract interpreter)

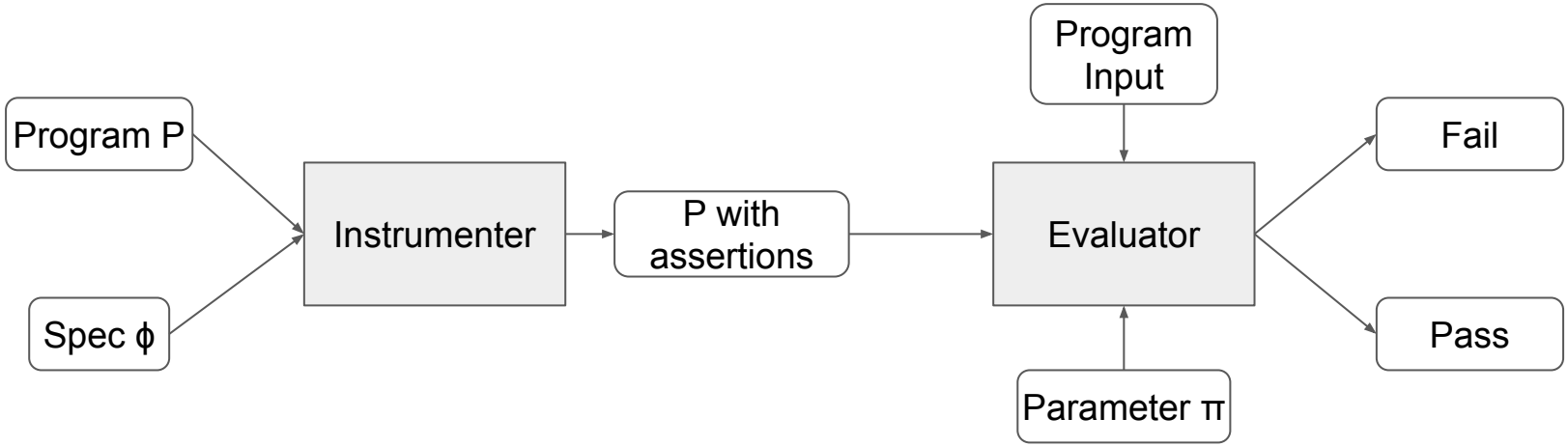
$\alpha : 2^{\text{State}} \longrightarrow \text{Abs\_State}$ (abstraction function)

$\alpha\{\text{eval}(s) | s \in X\} \sqsubseteq \text{abs\_int}(\alpha(X))$ (relationship between eval and abs_int)

$\text{erase} : \text{State} \longrightarrow \text{Uninstrumented\_State}$

$\text{extract} : \text{State} \longrightarrow \text{Instrumentation\_Data}$

Parametric Verifiers - Dynamic

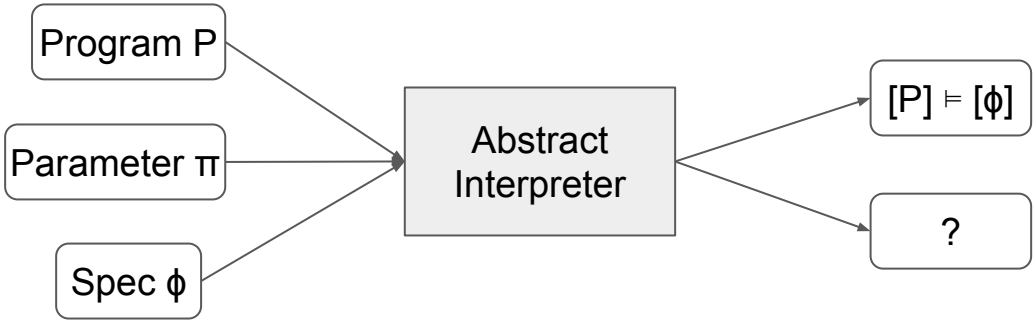


$eval_\pi$: Parameter x State ⟶ State (parametric evaluator for the language)

$\forall\, s,\pi.\quad (erase(eval_\pi(\pi,s)) = erase(eval(s)))\ \vee\ (eval_\pi(\pi,s) = fail_i)$

$\forall\, \pi_1,\pi_2.\quad \pi_1 \preccurlyeq \pi_2 \Rightarrow \exists s,i,j.(\ (eval_\pi(\pi_1,s) = fail_i) \Rightarrow (eval_\pi(\pi_2,s) = fail_j))$

Parametric Verifiers - Static

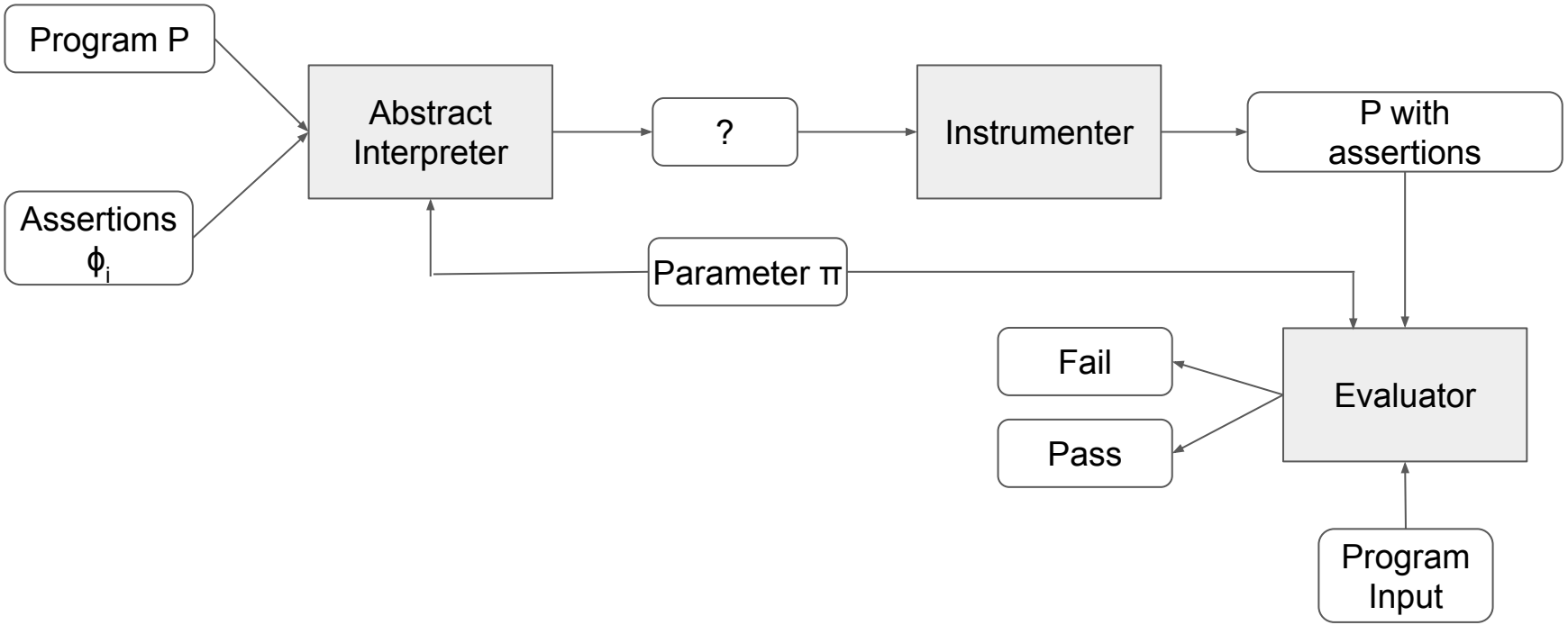


abs_int_π: Parameter x Abs_State ⟶ Abs_State (parametric abstract interpreter)

$\forall X,\pi.\quad \alpha\{eval_\pi(\pi,s) | s \in X\} \sqsubseteq abs\_int_\pi(\pi,\alpha(X))$

$\forall \pi_1,\pi_2.\quad \pi_1 \preccurlyeq \pi_2 \quad \Rightarrow \quad fails(abs\_int_\pi(\pi_1,init)) \subseteq fails(abs\_int_\pi(\pi_2,init))$

Let’s combine, again!



Minimal Parameter Problem (MPP)

- Find minimal parameter π such that,

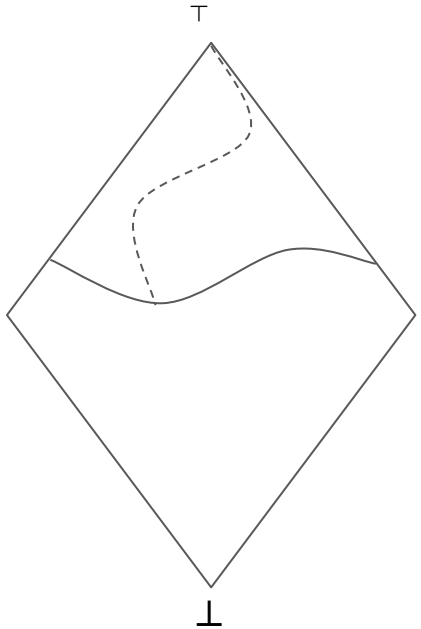
$$\forall\, s,i.\ (\mathrm{eval}_{\pi}(\top,s) = \mathrm{fail}_{i}) \ \Rightarrow\ (\mathrm{eval}_{\pi}(\pi,s) = \mathrm{fail}_{i})$$

- Finding the minimal parameter is equivalent to removing “all dynamic” instrumentation “associated” with statically proven assertions!

- MPP is not tractable, but we can approximately solve it by exploiting the relationship between our parametric static and dynamic verifiers

A simple parameter search algorithm

Parameter Space



Analysis designer’s responsibilities

- Parameterize static and dynamic verifiers with a shared parametric notion

- Ensure:
  - $\alpha\{\text{eval}(s) | s \in X\} \sqsubseteq \text{abs\_int}(\alpha(X))$
  - $\forall\, s,\pi. \quad (\text{erase}(\text{eval}_\pi(\pi,s)) = \text{erase}(\text{eval}(s))) \quad \vee \quad (\text{eval}_\pi(\pi,s) = \text{fail}_i)$
  - $\forall\, \pi_1,\pi_2. \quad \pi_1 \preccurlyeq \pi_2 \quad \Rightarrow \quad \exists s,i,j.(\ (\text{eval}_\pi(\pi_1,s) = \text{fail}_i) \ \Rightarrow\ (\text{eval}_\pi(\pi_2,s) = \text{fail}_j))$
  - $\forall\, X,\pi. \quad \alpha\{\text{eval}_\pi(\pi,s) | s \in X\} \sqsubseteq \text{abs\_int}_\pi(\pi,\alpha(X))$
  - $\forall\, \pi_1,\pi_2. \quad \pi_1 \preccurlyeq \pi_2 \quad \Rightarrow \quad \text{fails}(\text{abs\_int}_\pi(\pi_1,\text{init})) \subseteq \text{fails}(\text{abs\_int}_\pi(\pi_2,\text{init}))$

- And then,

Can we improve?

- Improve parameter search:
  - Exploit the structure of static verifier proofs
  - Leverage existing work on finding best abstractions

- Further reduce dynamic verifier overheads
  - Make optimistic/speculative assumptions statically to prove more assertions
  - Check these assumptions at runtime
  - Reduced instrumentation due to more proven assertions vs Overhead of checking assumptions

- Connections to gradual typing and hybrid typing (Please help!)

Thank You!