Metaprogramming Program Analyzers

Guannan Wei

ANTIQUE Seminar - 25/10/2025

About Me

- PhD, Purdue University, 2023
  - Dissertation: Metaprogramming Program Analyzers

- Incoming assistant professor at Tufts University (Boston)

Metaprogramming Program Analyzers

- Goal: improving the construction and performance of program analyzers
  - Inherent complexity: undecidability, NP-complete/hard, abstract domains, etc.
  - ***Accidental complexity***: languages/compiler, implementation choice, data structures, etc.

Metaprogramming Program Analyzers

- Goal: improving the construction and performance of program analyzers
  - Inherent complexity: undecidability, NP-complete/hard, abstract domains, etc.
  - ***Accidental complexity***: languages/compiler, implementation choice, data structures, etc.

- Metaprogramming Program Analyzers :
  Applying program generation/transformation to the implementation of analyzers

Metaprogramming Program Analyzers

- Goal: improving the construction and performance of program analyzers
  - Inherent complexity: undecidability, NP-complete/hard, abstract domains, etc.
  - ***Accidental complexity***: languages/compiler, implementation choice, data structures, etc.

- Metaprogramming Program Analyzers :
  Applying program generation/transformation to the implementation of analyzers
  - 1. Derivation of big-step abstract interpreters from small-step ones
  - 2. Compilation of analyses (control-flow analysis, symbolic execution)

Bridging the gap between small-step and big-step abstract interpreters

Abstract interpretation:
An approach to build sound static analysis from a concrete semantics.

Can we build an abstract interpreter from another abstract interpreter?

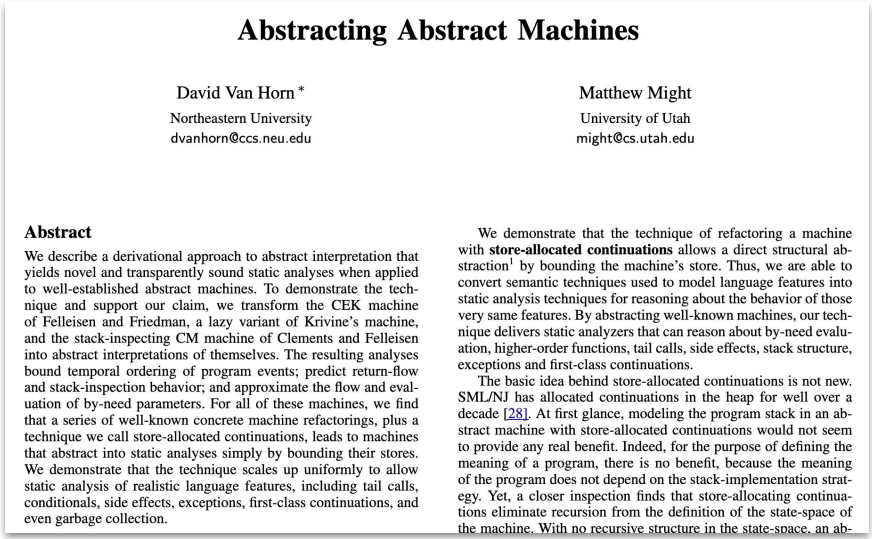
Bridging the gap between small-step and big-step abstract interpreters

Abstract Abstract Machines
[ICFP 10]

Abstract Machines
(CEK/CESK)

A recipe to derive *small-step* abstract interpreters from concrete state machines.

Popular approach in analyzing functional higher-order programs; rooted in control-flow analysis (CFA).



Abstracting Abstract Machines

David Van Horn *
Northeastern University
dvanhorn@ccs.neu.edu

Matthew Might
University of Utah
might@cs.utah.edu

Abstract

We describe a derivational approach to abstract interpretation that yields novel and transparently sound static analyses when applied to well-established abstract machines. To demonstrate the technique and support our claim, we transform the CEK machine of Felleisen and Friedman, a lazy variant of Krivine's machine, and the stack-inspecting CM machine of Clements and Felleisen into abstract interpretations of themselves. The resulting analyses bound temporal ordering of program events; predict return-flow and stack-inspection behavior; and approximate the flow and evaluation of by-need parameters. For all of these machines, we find that a series of well-known concrete machine refactorings, plus a technique we call store-allocated continuations, leads to machines that abstract into static analyses simply by bounding their stores. We demonstrate that the technique scales up uniformly to allow static analysis of realistic language features, including tail calls, conditionals, side effects, exceptions, first-class continuations, and even garbage collection.

We demonstrate that the technique of refactoring a machine with **store-allocated continuations** allows a direct structural abstraction[1] by bounding the machine's store. Thus, we are able to convert semantic techniques used to model language features into static analysis techniques for reasoning about the behavior of those very same features. By abstracting well-known machines, our technique delivers static analyzers that can reason about by-need evaluation, higher-order functions, tail calls, side effects, stack structure, exceptions and first-class continuations.

The basic idea behind store-allocated continuations is not new. SML/NJ has allocated continuations in the heap for well over a decade [28]. At first glance, modeling the program stack in an abstract machine with store-allocated continuations would not seem to provide any real benefit. Indeed, for the purpose of defining the meaning of a program, there is no benefit, because the meaning of the program does not depend on the stack-implementation strategy. Yet, a closer inspection finds that store-allocating continuations eliminate recursion from the definition of the state-space of the machine. With no recursive structure in the state-space, an ab-

Originally at ICFP 2010; later CACM 2011;
ICFP Most Influential Paper at ICFP 2020

Bridging the gap between small-step and big-step abstract interpreters

Abstract Abstract Machines
[ICFP 10]

Abstract Machines
(CEK/CESK)

A recipe to derive *small-step* abstract interpreters from concrete state machines.

Popular approach in analyzing functional higher-order programs; rooted in control-flow analysis (CFA).

finite state space:

$\texttt{State}^\# := \langle \texttt{Expr}, \texttt{Env}^\#, \texttt{Store}^\#, \texttt{Kont}^\# \rangle$

nondeterministic state transition:

$\texttt{State}^\# \to \texttt{Set}[\texttt{State}^\#]$

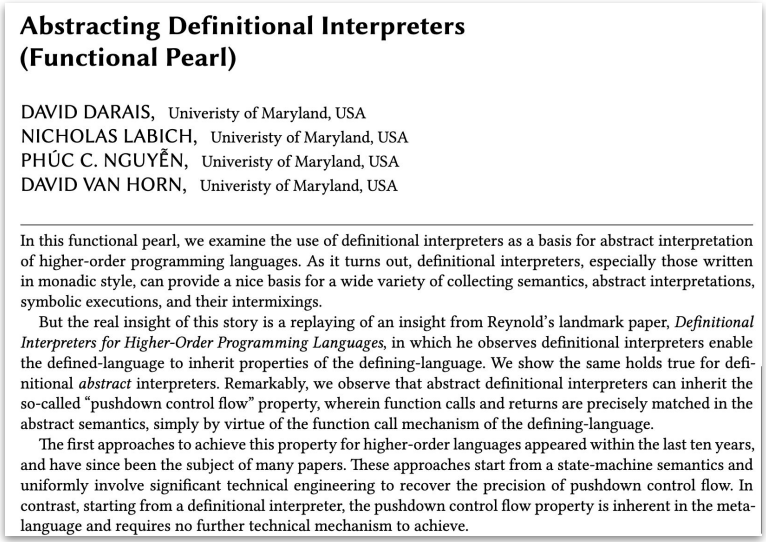
Bridging the gap between small-step and big-step abstract interpreters

Abstract Abstract Machines
[ICFP 10]

A recipe to derive *small-step* abstract interpreters from concrete state machines.

A *big-step*, *compositional*, monadic abstract interpreter.

Abstract Definitional Interpreters
[ICFP 17]



Abstracting Definitional Interpreters (Functional Pearl)

DAVID DARAIS, Univeristy of Maryland, USA
NICHOLAS LABICH, Univeristy of Maryland, USA
PHÚC C. NGUYỄN, Univeristy of Maryland, USA
DAVID VAN HORN, Univeristy of Maryland, USA

In this functional pearl, we examine the use of definitional interpreters as a basis for abstract interpretation of higher-order programming languages. As it turns out, definitional interpreters, especially those written in monadic style, can provide a nice basis for a wide variety of collecting semantics, abstract interpretations, symbolic executions, and their intermixings.

But the real insight of this story is a replaying of an insight from Reynold's landmark paper, *Definitional Interpreters for Higher-Order Programming Languages*, in which he observes definitional interpreters enable the defined-language to inherit properties of the defining-language. We show the same holds true for definitional *abstract* interpreters. Remarkably, we observe that abstract definitional interpreters can inherit the so-called "pushdown control flow" property, wherein function calls and returns are precisely matched in the abstract semantics, simply by virtue of the function call mechanism of the defining-language.

The first approaches to achieve this property for higher-order languages appeared within the last ten years, and have since been the subject of many papers. These approaches start from a state-machine semantics and uniformly involve significant technical engineering to recover the precision of pushdown control flow. In contrast, starting from a definitional interpreter, the pushdown control flow property is inherent in the meta-language and requires no further technical mechanism to achieve.

Abstract Machines
(CEK/CESK)

Definitional Interpreters

Bridging the gap between small-step and big-step abstract interpreters

Abstract Abstract Machines
[ICFP 10]

A recipe to derive *small-step* abstract interpreters from concrete state machines.

A *big-step*, *compositional*, monadic abstract interpreter.

Abstract Definitional Interpreters
[ICFP 17]

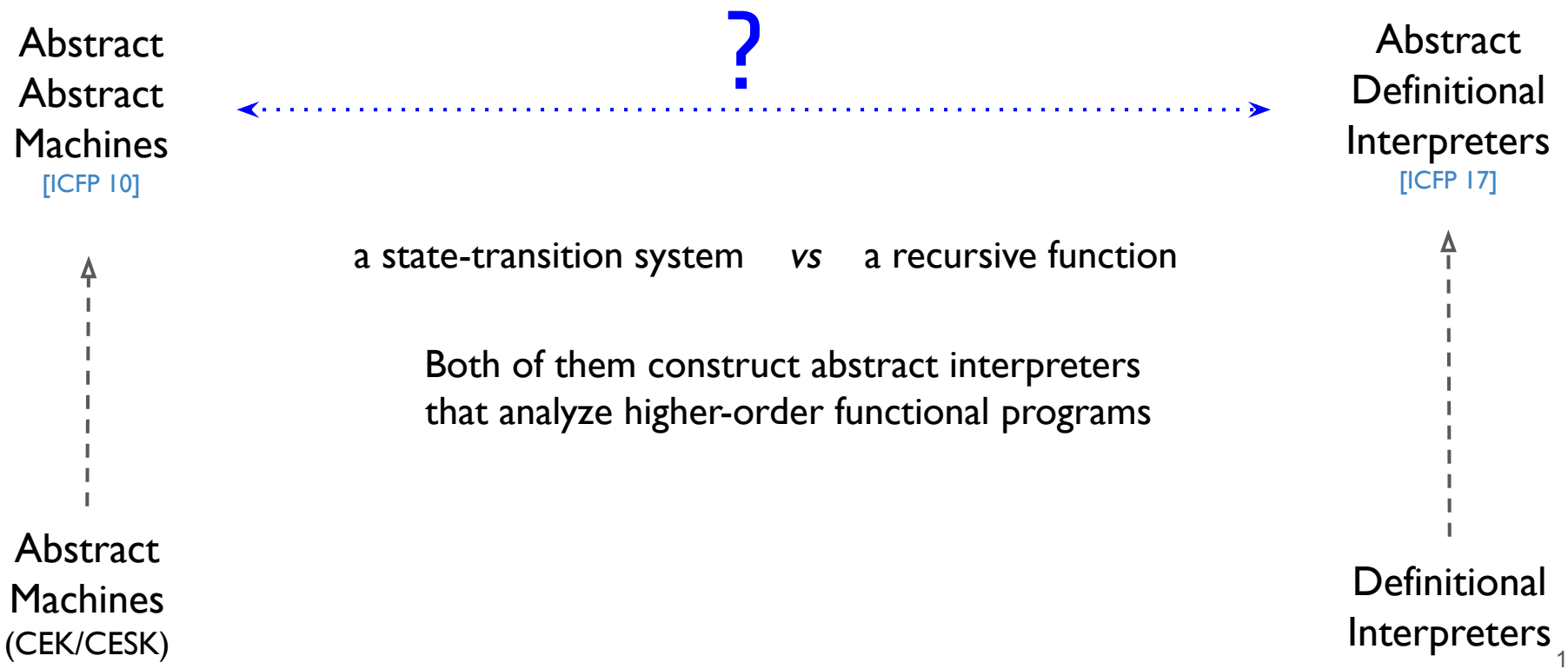
a state-transition system *vs* a recursive function

Both of them construct abstract interpreters that analyze higher-order functional programs

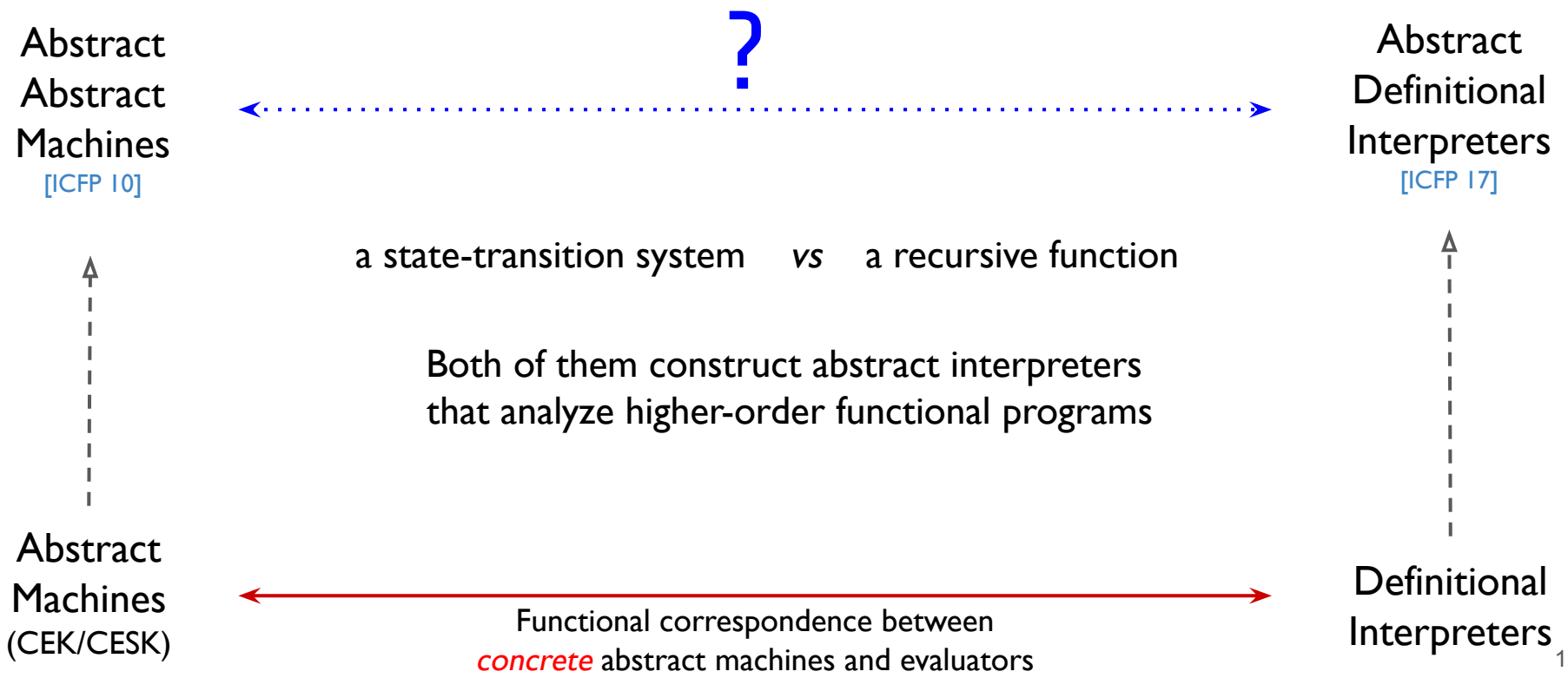
Abstract Machines
(CEK/CESK)

Definitional Interpreters

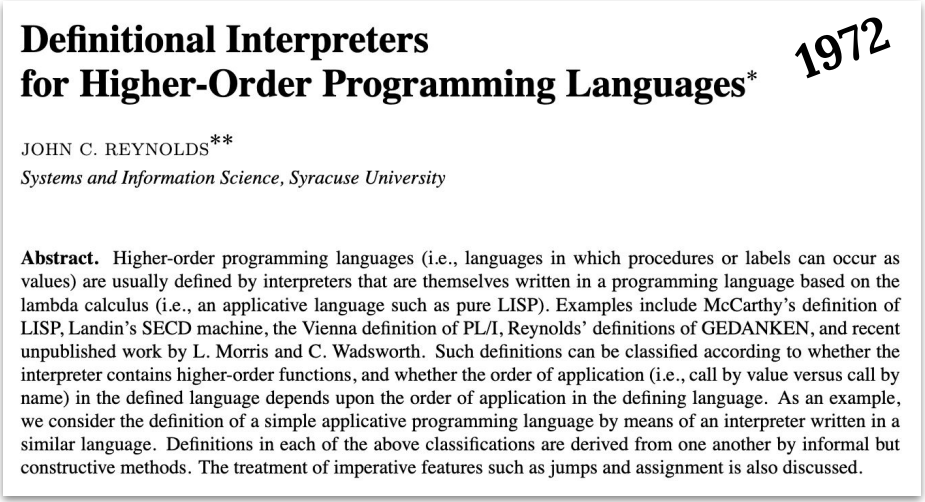
Bridging the gap between small-step and big-step abstract interpreters



Bridging the gap between small-step and big-step abstract interpreters



Bridging the gap between small-step and big-step ~~abstract~~ interpreters



Definitional Interpreters for Higher-Order Programming Languages*

1972

JOHN C. REYNOLDS**
Systems and Information Science, Syracuse University

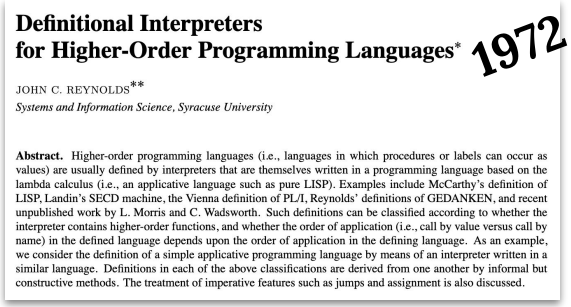
Abstract. Higher-order programming languages (i.e., languages in which procedures or labels can occur as values) are usually defined by interpreters that are themselves written in a programming language based on the lambda calculus (i.e., an applicative language such as pure LISP). Examples include McCarthy's definition of LISP, Landin's SECD machine, the Vienna definition of PL/I, Reynolds' definitions of GEDANKEN, and recent unpublished work by L. Morris and C. Wadsworth. Such definitions can be classified according to whether the interpreter contains higher-order functions, and whether the order of application (i.e., call by value versus call by name) in the defined language depends upon the order of application in the defining language. As an example, we consider the definition of a simple applicative programming language by means of an interpreter written in a similar language. Definitions in each of the above classifications are derived from one another by informal but constructive methods. The treatment of imperative features such as jumps and assignment is also discussed.

Abstract Machines (CEK/CESK)

Functional correspondence between *concrete* abstract machines and evaluators

Definitional Interpreters

Bridging the gap between small-step and big-step ~~abstract~~ interpreters



Definitional Interpreters for Higher-Order Programming Languages* 1972

JOHN C. REYNOLDS**
Systems and Information Science, Syracuse University

Abstract. Higher-order programming languages (i.e., languages in which procedures or labels can occur as values) are usually defined by interpreters that are themselves written in a programming language based on the lambda calculus (i.e., an applicative language such as pure LISP). Examples include McCarthy's definition of LISP, Landin's SECD machine, the Vienna definition of PL/I, Reynolds' definitions of GEDANKEN, and recent unpublished work by L. Morris and C. Wadsworth. Such definitions can be classified according to whether the interpreter contains higher-order functions, and whether the order of application (i.e., call by value versus call by name) in the defined language depends upon the order of application in the defining language. As an example, we consider the definition of a simple applicative programming language by means of an interpreter written in a similar language. Definitions in each of the above classifications are derived from one another by informal but constructive methods. The treatment of imperative features such as jumps and assignment is also discussed.

Order-of-application dependence:	Use of higher-order functions: yes	no
yes	direct interpreter for GEDANKEN	McCarthy's definition of LISP
no	Morris-Wadsworth method	SECD machine, Vienna definition

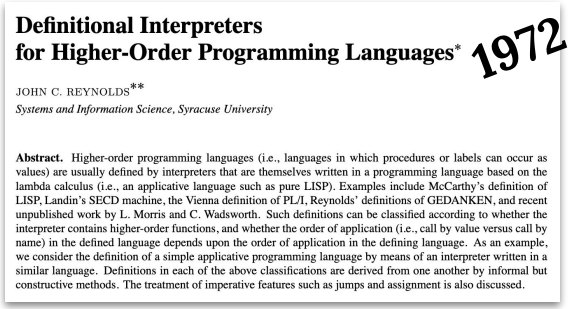
Abstract Machines (CEK/CESK)

Definitional Interpreters

Functional correspondence between *concrete* abstract machines and evaluators

Bridging the gap between small-step and big-step ~~abstract~~ interpreters

*de*functionalization: transform higher-order functions to first-order data types with their dispatching functions (e.g., closure conversion).



Definitional Interpreters for Higher-Order Programming Languages* 1972

JOHN C. REYNOLDS**
Systems and Information Science, Syracuse University

Abstract. Higher-order programming languages (i.e., languages in which procedures or labels can occur as values) are usually defined by interpreters that are themselves written in a programming language based on the lambda calculus (i.e., an applicative language such as pure LISP). Examples include McCarthy's definition of LISP, Landin's SECD machine, the Vienna definition of PL/I, Reynolds' definitions of GEDANKEN, and recent unpublished work by L. Morris and C. Wadsworth. Such definitions can be classified according to whether the interpreter contains higher-order functions, and whether the order of application (i.e., call by value versus call by name) in the defined language depends upon the order of application in the defining language. As an example, we consider the definition of a simple applicative programming language by means of an interpreter written in a similar language. Definitions in each of the above classifications are derived from one another by informal but constructive methods. The treatment of imperative features such as jumps and assignment is also discussed.

Order-of-application dependence:	Use of higher-order functions: yes → no	
yes	direct interpreter for GEDANKEN	McCarthy's definition of LISP
no	Morris-Wadsworth method	SECD machine, Vienna definition

Abstract Machines (CEK/CESK) ⟷ Definitional Interpreters

Functional correspondence between *concrete* abstract machines and evaluators

Bridging the gap between small-step and big-step ~~abstract~~ interpreters

*de*functionalization: transform higher-order functions to first-order data types with their dispatching functions (e.g., closure conversion).

*re*functionalization: the inverse of defunctionalization [Danvy et al.].

Order-of-application dependence:	Use of higher-order functions: yes ⇄ no	
yes	direct interpreter for GEDANKEN	McCarthy's definition of LISP
no	Morris-Wadsworth method	SECD machine, Vienna definition

Definitional Interpreters for Higher-Order Programming Languages* 1972

JOHN C. REYNOLDS**
Systems and Information Science, Syracuse University

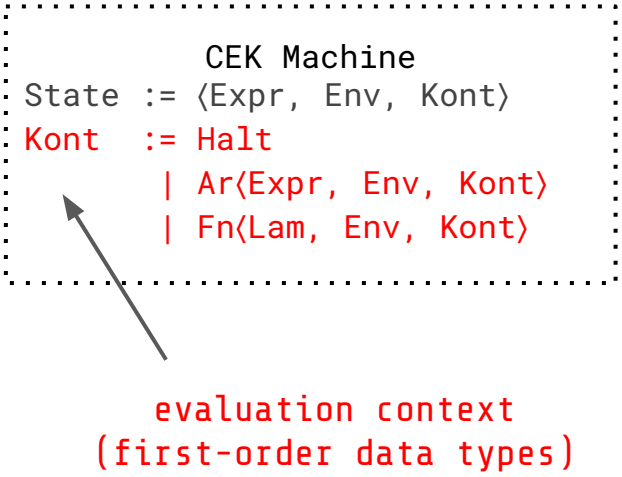
Abstract. Higher-order programming languages (i.e., languages in which procedures or labels can occur as values) are usually defined by interpreters that are themselves written in a programming language based on the lambda calculus (i.e., an applicative language such as pure LISP). Examples include McCarthy's definition of LISP, Landin's SECD machine, the Vienna definition of PL/I, Reynolds' definitions of GEDANKEN, and recent unpublished work by L. Morris and C. Wadsworth. Such definitions can be classified according to whether the interpreter contains higher-order functions, and whether the order of application (i.e., call by value versus call by name) in the defined language depends upon the order of application in the defining language. As an example, we consider the definition of a simple applicative programming language by means of an interpreter written in a similar language. Definitions in each of the above classifications are derived from one another by informal but constructive methods. The treatment of imperative features such as jumps and assignment is also discussed.

Abstract Machines (CEK/CESK) ⟷ Definitional Interpreters

Functional correspondence between *concrete* abstract machines and evaluators

Bridging the gap between small-step and big-step ~~abstract~~ interpreters

- Example: refunctionalizing a CEK machine yields an interpreter in continuation-passing style.



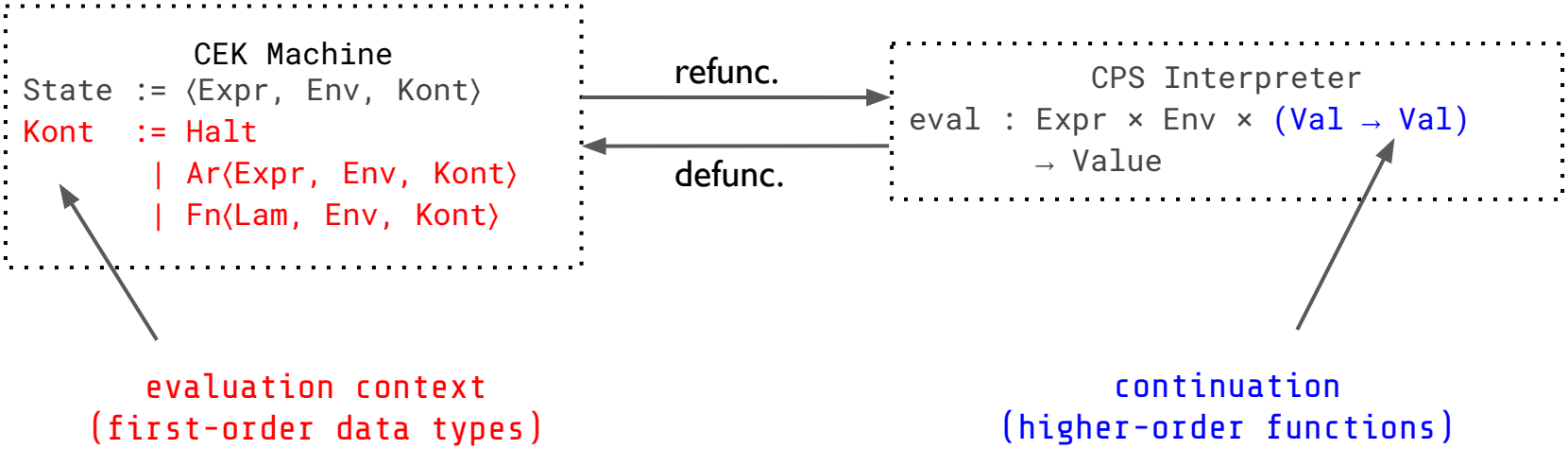
Abstract Machines (CEK/CESK)

Functional correspondence between *concrete* abstract machines and evaluators

Definitional Interpreters

Bridging the gap between small-step and big-step ~~abstract~~ interpreters

- Example: refunctionalizing a CEK machine yields an interpreter in continuation-passing style.



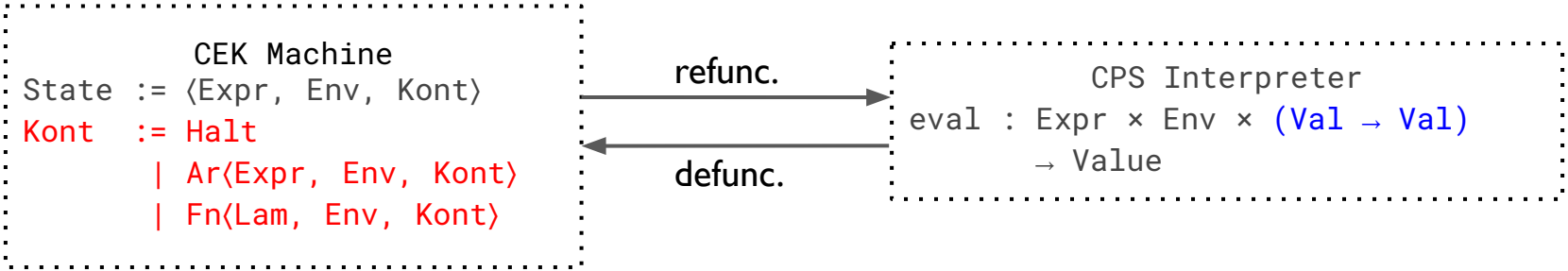
Abstract Machines (CEK/CESK)

Functional correspondence between *concrete* abstract machines and evaluators

Definitional Interpreters

Bridging the gap between small-step and big-step ~~abstract~~ interpreters

- Example: refunctionalizing a CEK machine yields an interpreter in continuation-passing style.



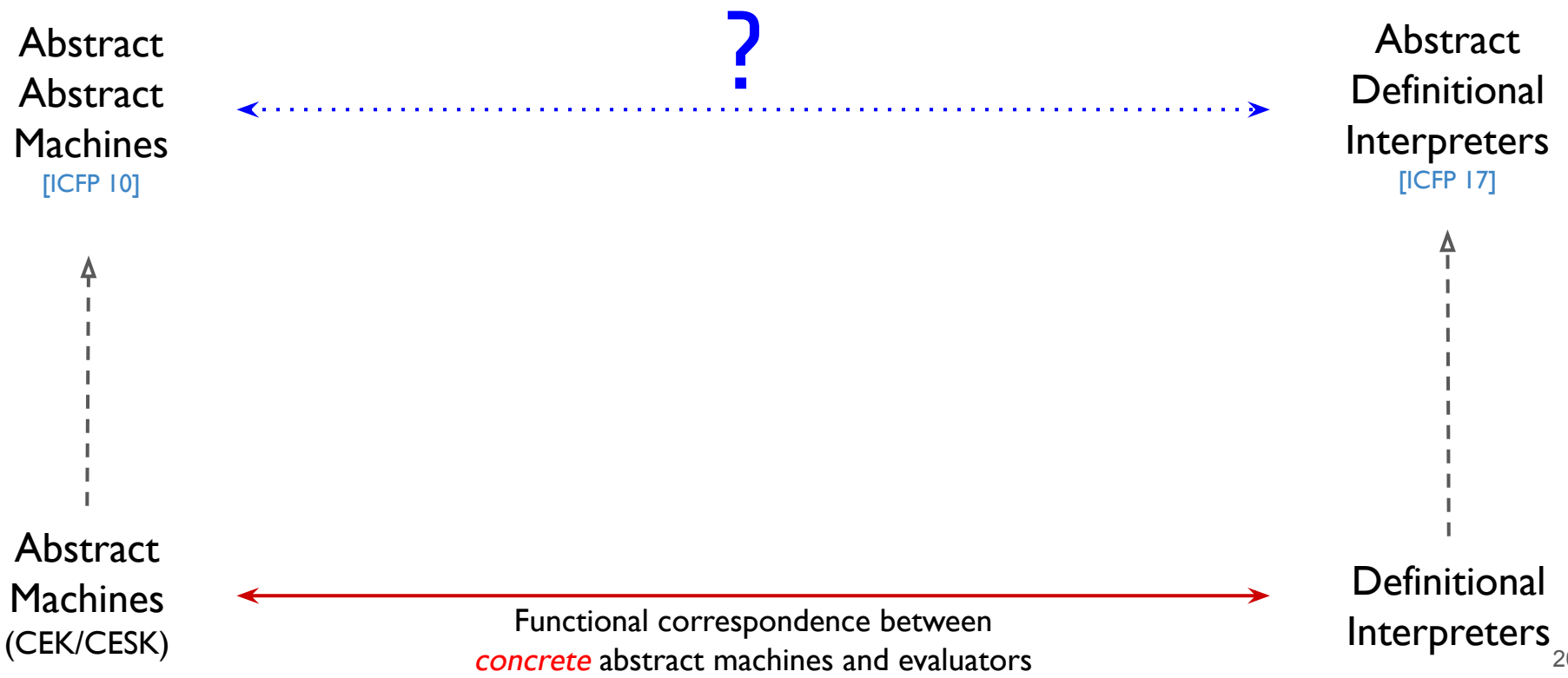
- *refunctionalized* evaluation contexts = *higher-order* continuations
- *defunctionalized* continuations = *first-order* evaluation contexts

Abstract Machines (CEK/CESK)

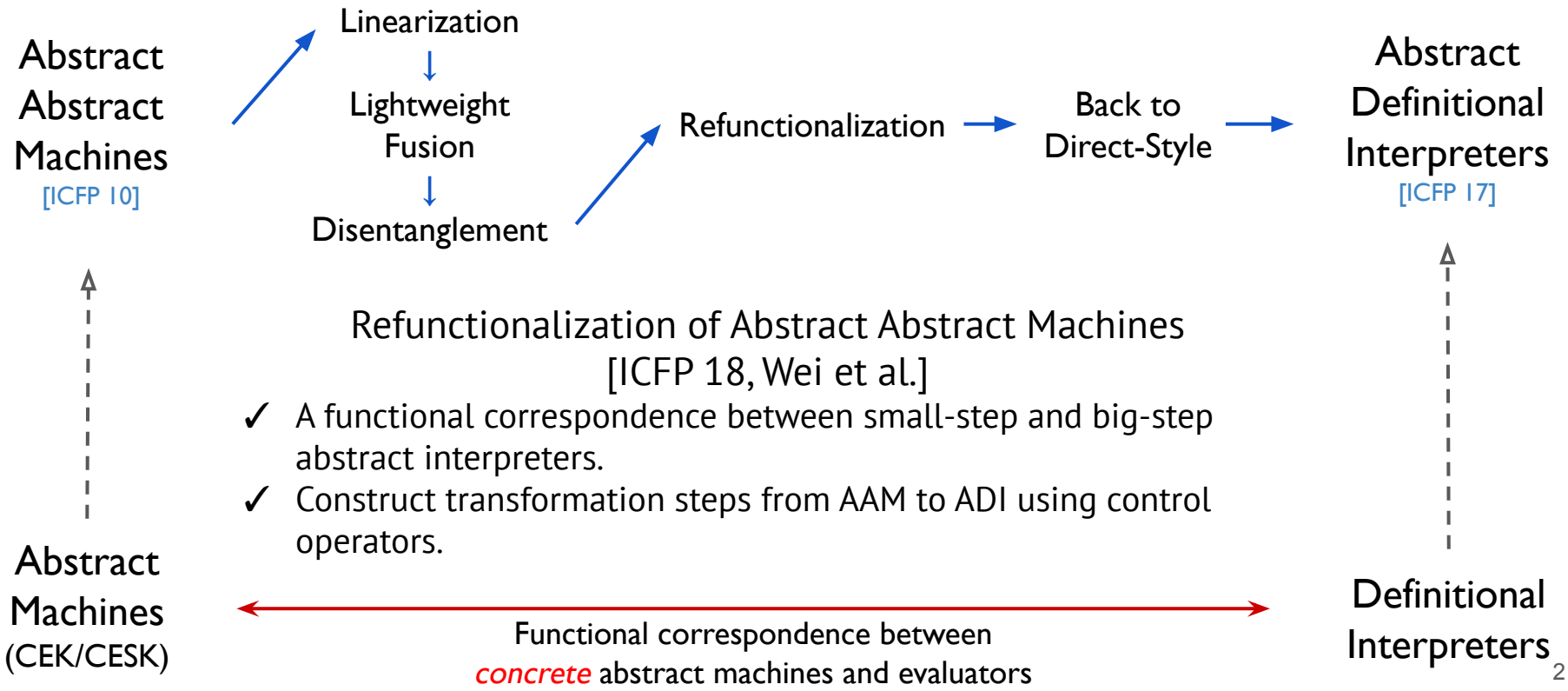
Definitional Interpreters

Functional correspondence between *concrete* abstract machines and evaluators

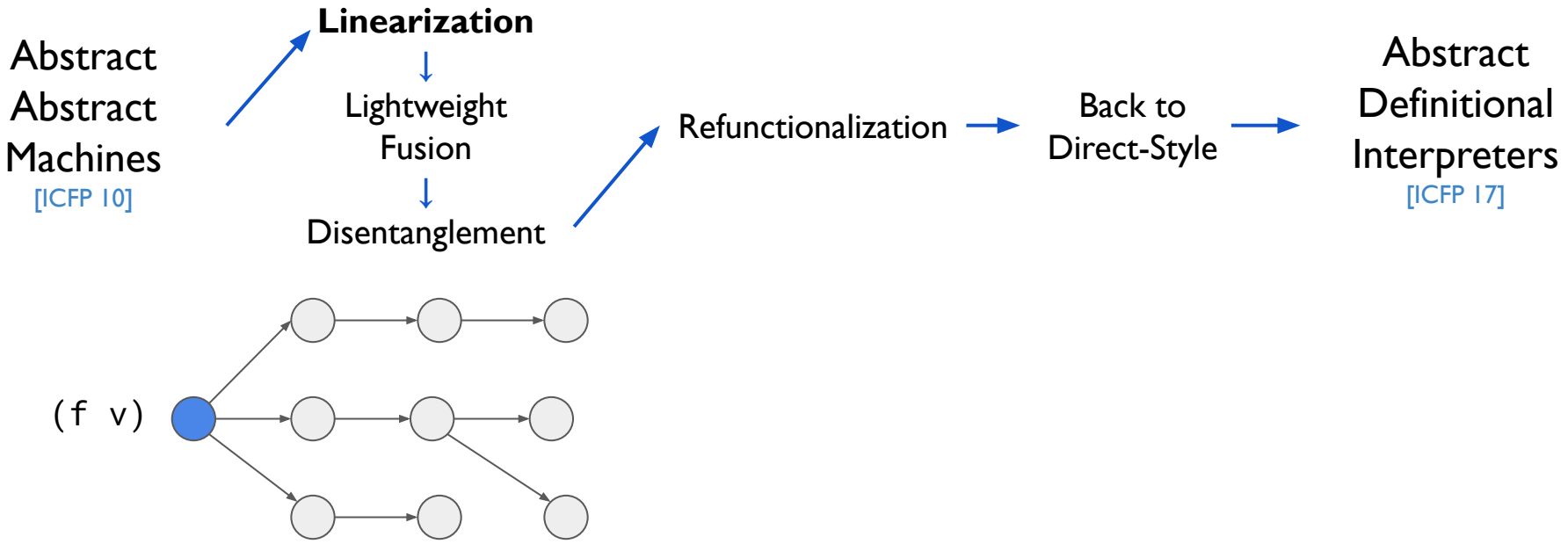
Bridging the gap between small-step and big-step abstract interpreters



Bridging the gap between small-step and big-step abstract interpreters



Bridging the gap between small-step and big-step abstract interpreters



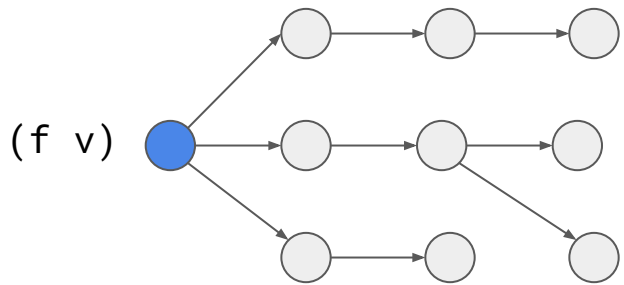

```
State# = ⟨Expr, Env#, Store#, Kont⟩
step: State# → P(State#)
```

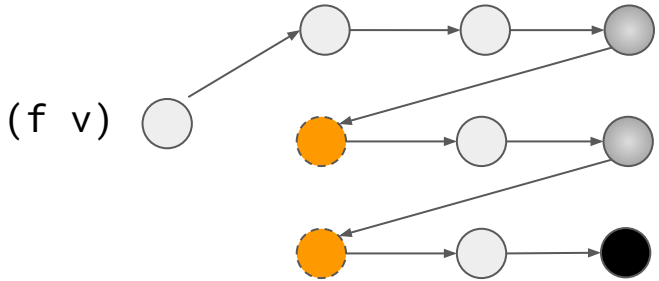
Bridging the gap between small-step and big-step abstract interpreters

Abstract Abstract Machines
[ICFP 10]

Linearization
↓
Lightweight Fusion
↓
Disentanglement

Refunctionalization → Back to Direct-Style → Abstract Definitional Interpreters
[ICFP 17]

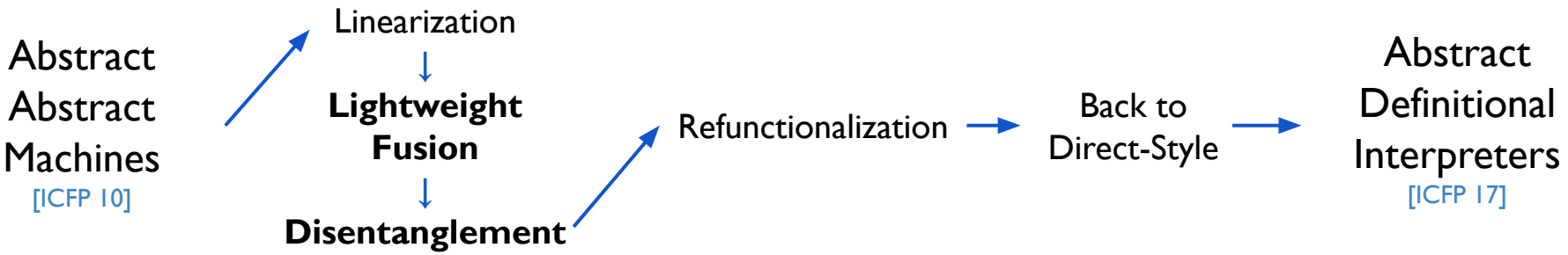




`State# = ⟨Expr, Env#, Store#, Kont⟩`
`step: State# → P(State#)`

`NDState# = ⟨Expr, Env#, Store#, Kont, MKont⟩`
`step: NDState# → NDState#`

Bridging the gap between small-step and big-step abstract interpreters



Further tweak the form of AAM and expose continuations explicitly:

- ***Fusion***: merges the step function and the driver loop function into one, so the abstract interpreter is a single, recursive function.
- ***Disentanglement***: lifts the code that dispatches those two data types representing continuations to be top-level functions.

Bridging the gap between small-step and big-step abstract interpreters

- Types of the first-order dispatching functions:

```
State : ⟨Expr, Env#, Store#, Kont, MKont⟩
continue  : State × Cache ⇒ Cache
mcontinue : State × Cache ⇒ Cache
```

- Types of the higher-order continuations:

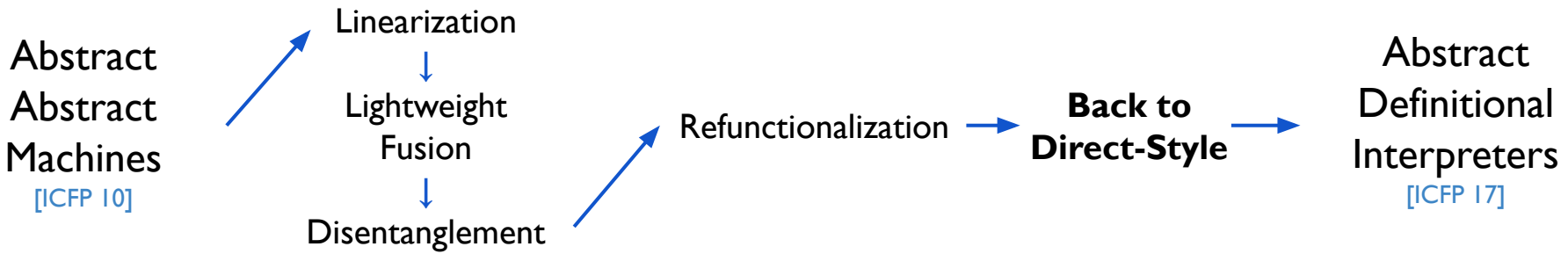
```
State : ⟨Expr, Env#, Store#, ~~Kont, MKont~~⟩
type Cont  = (State × Cache × MCont) ⇒ Cache
type MCont = (State × Cache) ⇒ Cache
```

Refunctionalization to 2CPS

```
def aeval(state: State, seen: Cache, k: Cont, mk: MCont): Cache = {
  e match {
    case Let(x, App(f, ae), e) if isAtomic(f) && isAtomic(ae) ⇒
      val closures = atomicEval(f, ρ, σ).toList
      val Clos(Lam(v, body), c_ρ) = closures.head
      val α = alloc(v);                    val new_ρ = c_ρ + (v ↦ α)
      val argvs = atomicEval(ae, ρ, σ);    val new_σ = σ.join(α ↦ argvs)
      val new_k: Cont = ...
       // A HO function takes result of App and then evaluates e
      val new_mk: MCont = ...
      // A HO function iterates over the target closures
      aeval(State(body, new_ρ, new_σ), new_seen, new_k, new_mk)

    case ae if isAtomic(ae) ⇒ k(state, new_seen, mk)
  }
}
```

Bridging the gap between small-step and big-step abstract interpreters



- We have obtained a *re*functionalized AAM in CPS.
- From extended CPS to direct-style, three choices:
  - Use explicit side-effects and assignments.
  - Use monads [Darais et al. ICFP 17].
  - *Use delimited control operators (shift/reset).*
    - shift to capture the continuation
    - reset to set the boundary

Back to Direct-Style

```
def aeval(state: State, seen: Cache): (State, Cache) @cps[Cache] = {
  ...
  e match {
    case Let(x, App(f, ae), e) if isAtomic(f) && isAtomic(ae) ⇒
      val closures = atomicEval(f, ρ, σ).toList
      val (Clos(Lam(v, body), c_ρ), c_seen) = choices(closures, new_seen)
      val v_α = alloc(v);  val new_ρ = c_ρ + (v ↦ v_α)
      val new_σ = σ.join(v_α ↦ atomicEval(ae, ρ, σ))
      val (bd_state, bd_seen) = aeval(State(body, new_ρ, new_σ), c_seen)
      val State(bd_ae, bd_ρ, bd_σ) = bd_state
      val x_α = alloc(x);  val new_ρ_* = ρ + (x ↦ x_α)
      val new_σ_* = bd_σ.join(x_α ↦ atomicEval(bd_ae, bd_ρ, bd_σ))
      aeval(State(e, new_ρ_*, new_σ_*), bd_seen)
    case ae if isAtomic(ae) ⇒ (state, new_seen)
  }
}
```

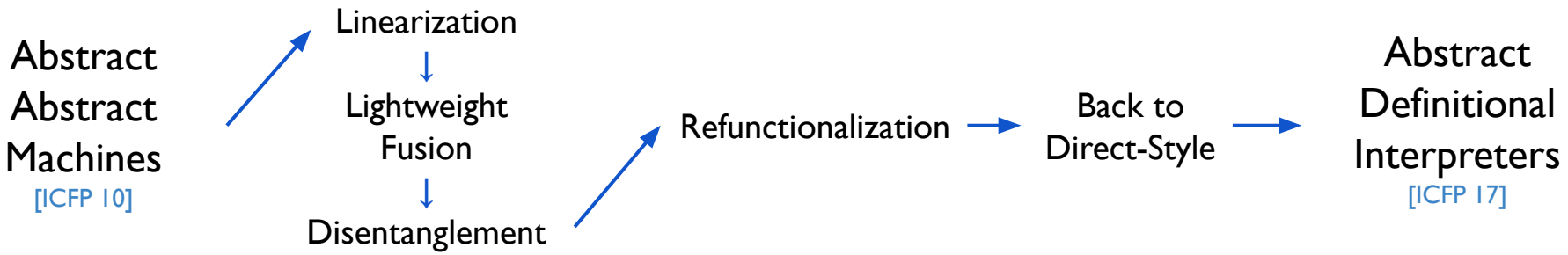
Back to Direct-Style

```
def aeval(state: State, seen: Cache): (State, Cache) @cps[Cache] = {
  ...
  e match {
    case Let(x, App(f, ae), e) if isAtomic(f) && isAtomic(ae) ⇒
      val closures = atomicEval(f, ρ, σ).toList
      val (Clos(Lam(v, body), c_ρ), c_seen) = choices(closures, new_seen)
      val v_α = alloc(v);  val new_ρ = c_ρ + (v ↦ v_α)
      val new_σ = σ.join(v_α ↦ atomicEval(ae, ρ, σ))
      val (bd_state, bd_seen) = aeval(State(body, new_ρ, new_σ), c_seen)
      val State(bd_ae, bd_ρ, bd_σ) = bd_state
      val x_α = alloc(x);  val new_ρ_* = ρ + (x ↦ x_α)
      val new_σ_* = bd_σ.join(x_α ↦ atomicEval(bd_ae, bd_ρ, bd_σ))
      aeval(State(e, new_ρ_*, new_σ_*), bd_seen)
    case ae if isAtomic(ae) ⇒ (state, new_seen)
  }
}
```

Get a closure of f, nondeterministically.

choices uses shift to capture the continuation, implicitly.

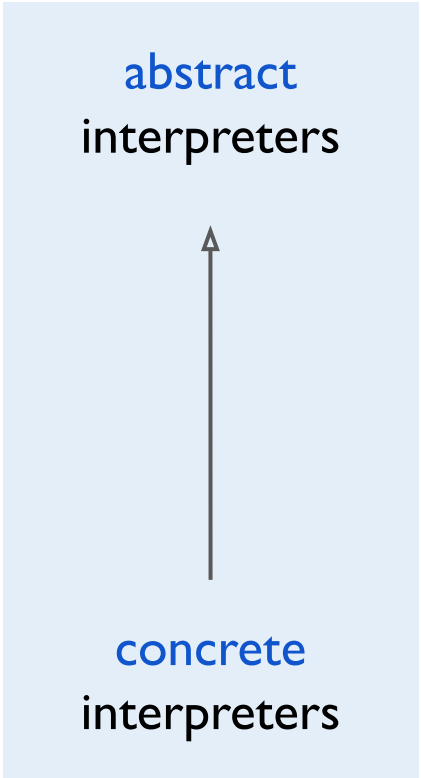
Bridging the gap between small-step and big-step abstract interpreters



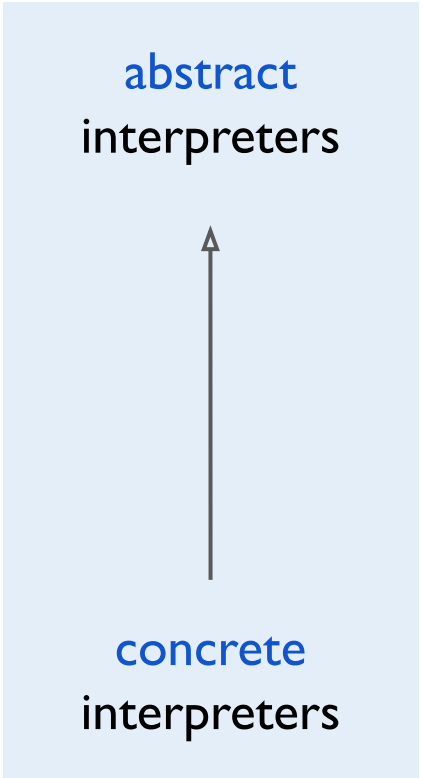
Applying transformations to the meta-constructs of the analyzer:
functional correspondence between
abstract semantic artifacts by refunctionalization [ICFP 18, Wei et al.]

The analysis results are equivalent modulo the representation of continuations.

Performant Program Analysis by Compilation



Performant Program Analysis by Compilation



Easy to build with high-level abstractions (e.g. AAM/ADI), but inherently slow.

- inspecting program AST/IR
- dispatching the semantics
- recursion/loop at meta-level
- abstractions (monads, etc.)
- …

Performant Program Analysis by Compilation

abstract
interpreters

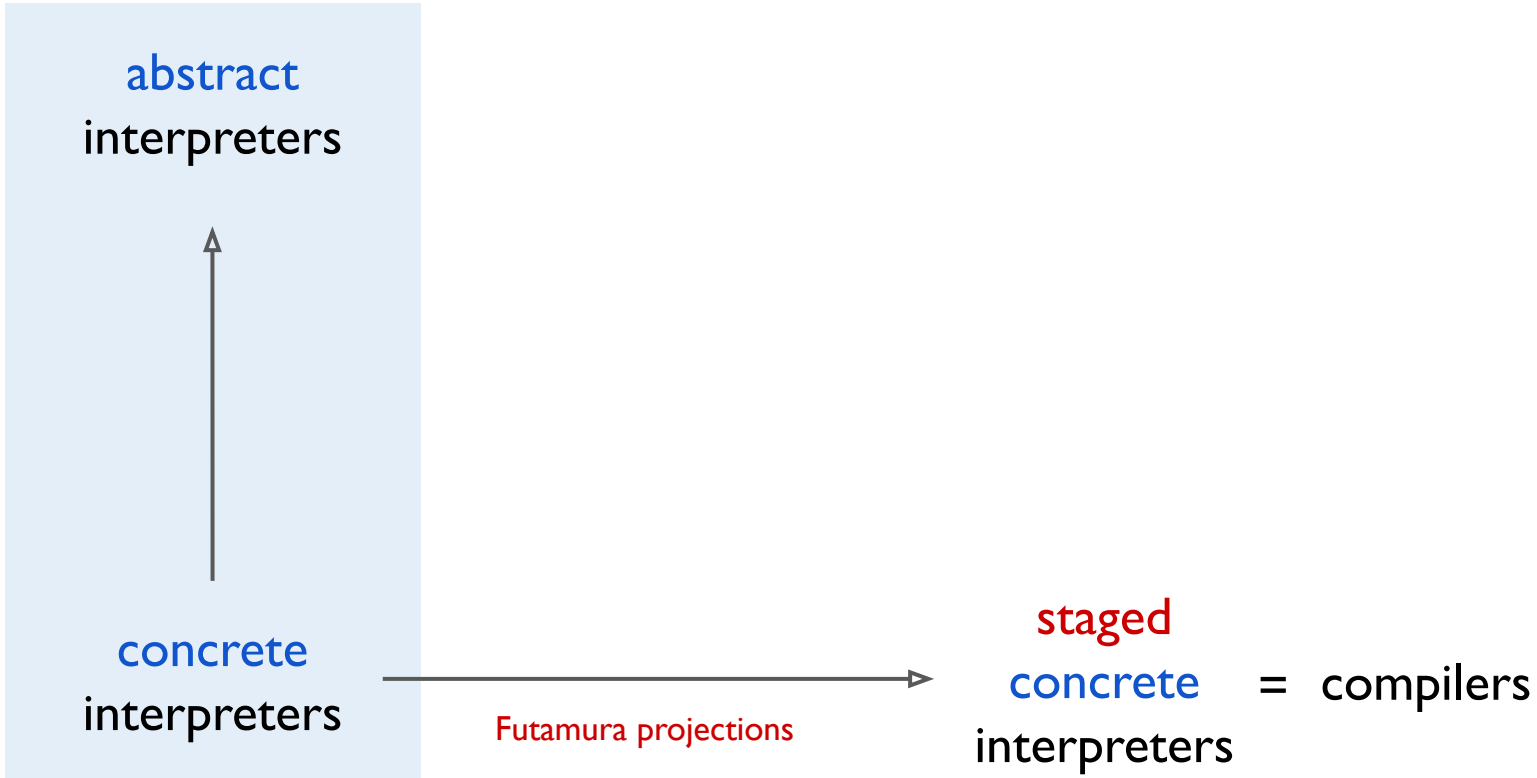
concrete
interpreters

Easy to build with high-level abstractions (e.g. AAM/ADI), but inherently slow.

A few data points:
symbolic interpreter performance
compared to native execution

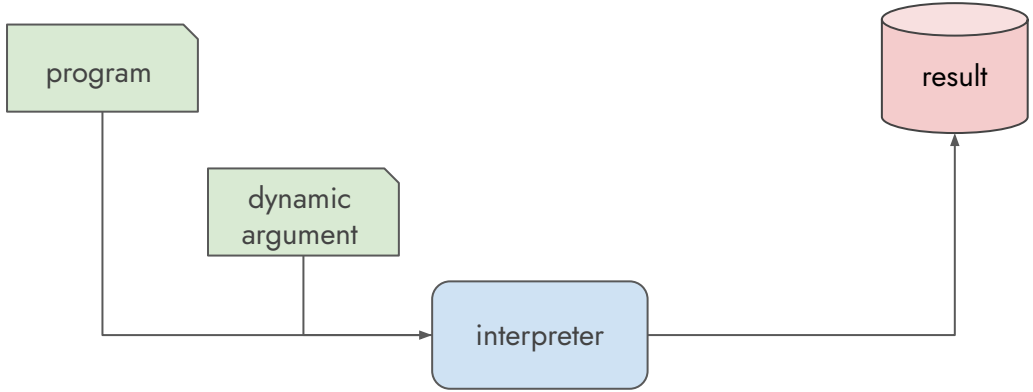
KLEE (C++)	3,000x slower
angr (Python)	321,000x slower

Performant Program Analysis by Compilation



Performant Program Analysis by Compilation

abstract
interpreters



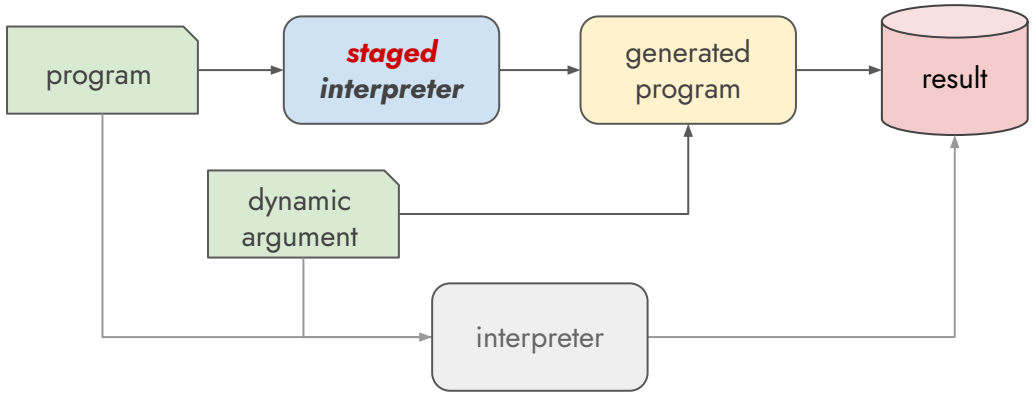
concrete
interpreters

Futamura projections

staged
concrete
interpreters
= compilers

Performant Program Analysis by Compilation

abstract
interpreters

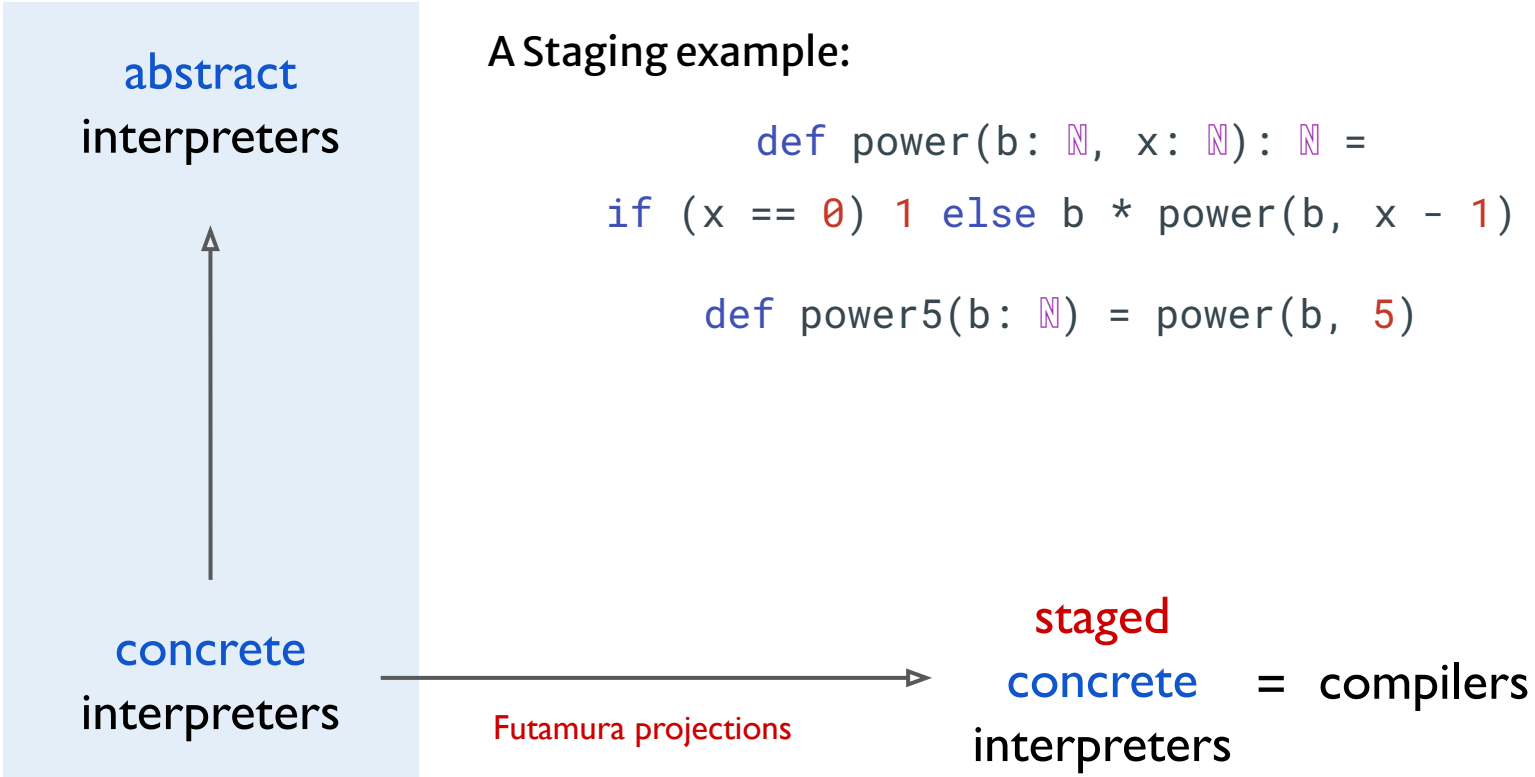


concrete
interpreters

Futamura projections

staged
concrete = compilers
interpreters

Performant Program Analysis by Compilation

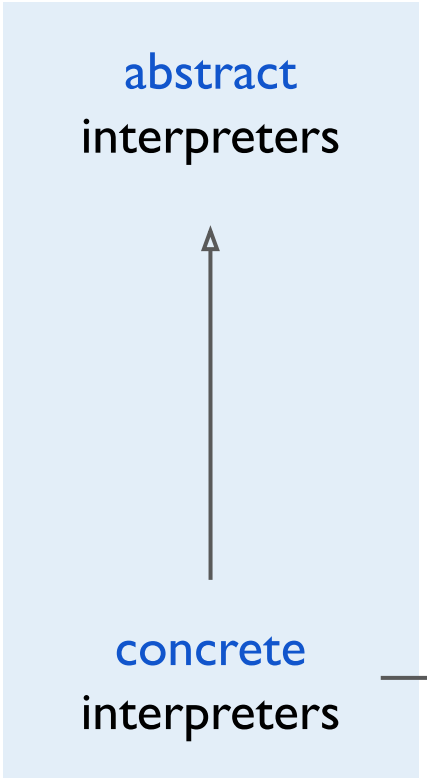


A Staging example:

```
def power(b: ℕ, x: ℕ): ℕ =
if (x == 0) 1 else b * power(b, x - 1)

def power5(b: ℕ) = power(b, 5)
```

Performant Program Analysis by Compilation



A Staging example:

```
def power(b: Rep[ℕ], x: ℕ): Rep[ℕ] =
if (x == 0) 1 else b * power(b, x - 1)

def power5(b: Rep[ℕ]) = power(b, 5)
```

specialization/code generation

```
def power5(b: ℕ) = b * b * b * b * b
```

Futamura projections

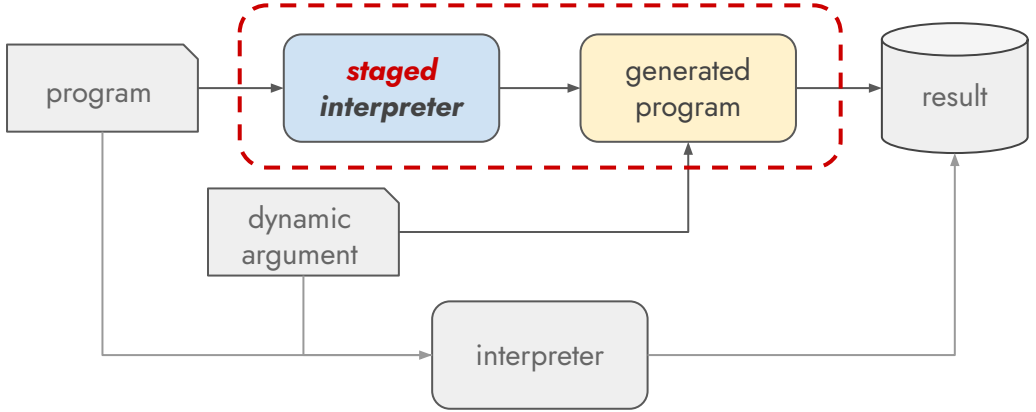
staged concrete interpreters = compilers

Performant Program Analysis by Compilation

abstract
interpreters

concrete
interpreters

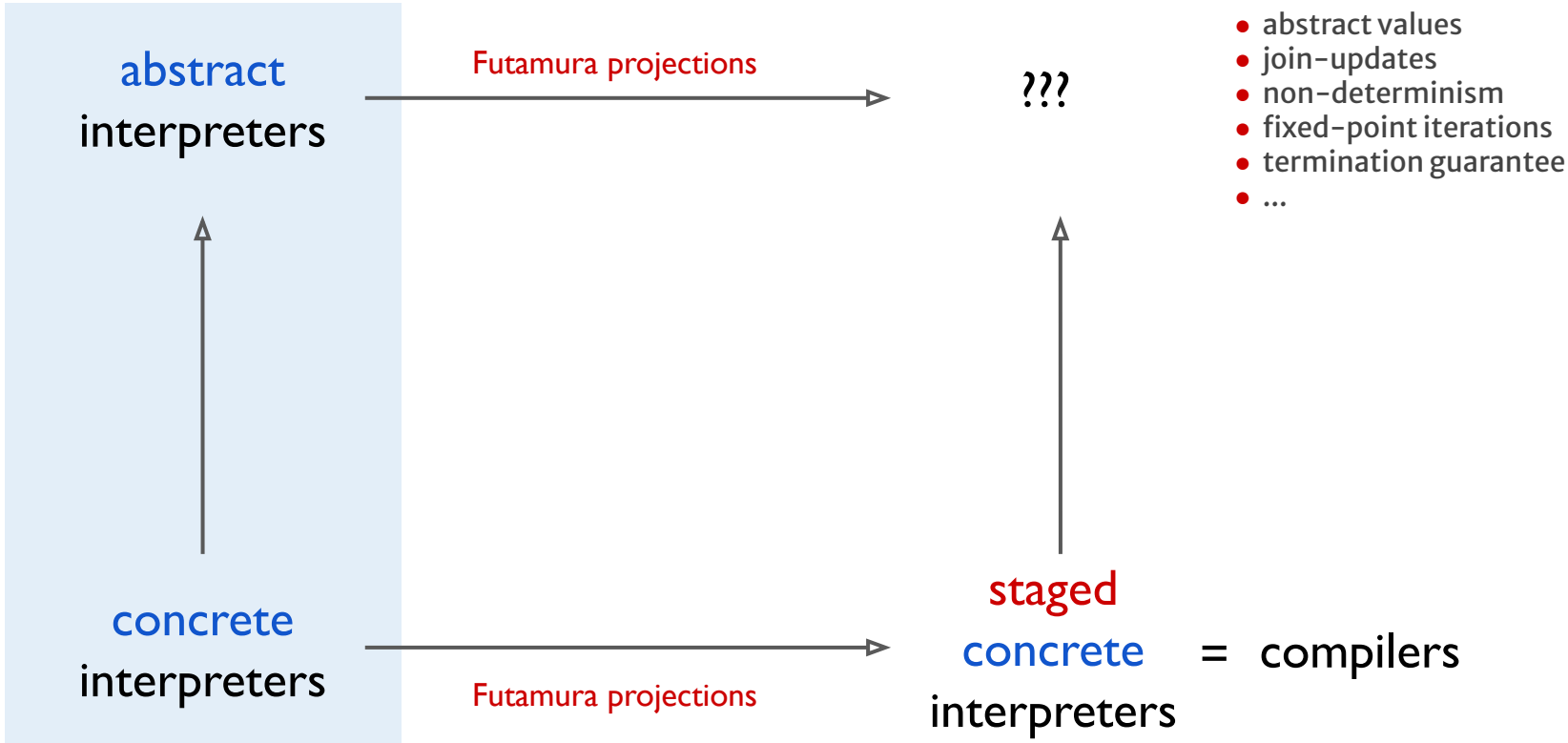
1st Futamura projection:
Compilation removes interpretation overhead



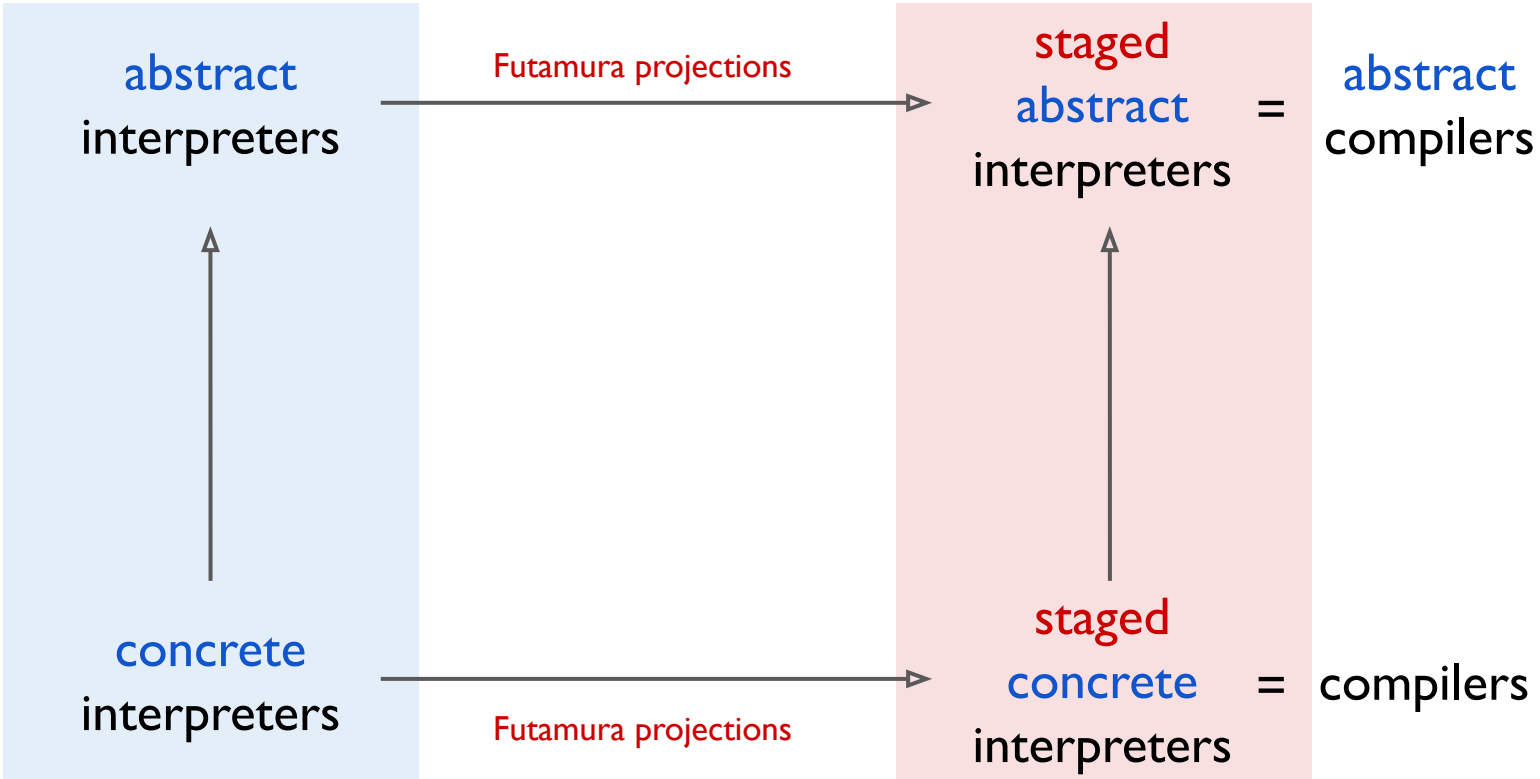
Futamura projections

staged
concrete
interpreters
= compilers

Performant Program Analysis by Compilation



Performant Program Analysis by Compilation



Performant Program Analysis by Compilation

abst
interp

= abstract compilers

conc
interpreters

Futamura projections

= compilers

interpreters

Staged Abstract Interpreters

Fast and Modular Whole-Program Analysis via Meta-programming

OOPSLA 19

GUANNAN WEI, YUXUAN CHEN, and TIARK ROMPF, Purdue University, USA

It is well known that a staged interpreter is a compiler: specializing an interpreter to a given program produces an equivalent executable that runs faster. This connection is known as the first Futamura projection. It is even more widely known that an abstract interpreter is a program analyzer: tweaking an interpreter to run on abstract domains produces a sound static analysis. What happens when we combine these two ideas, and apply specialization to an abstract interpreter?

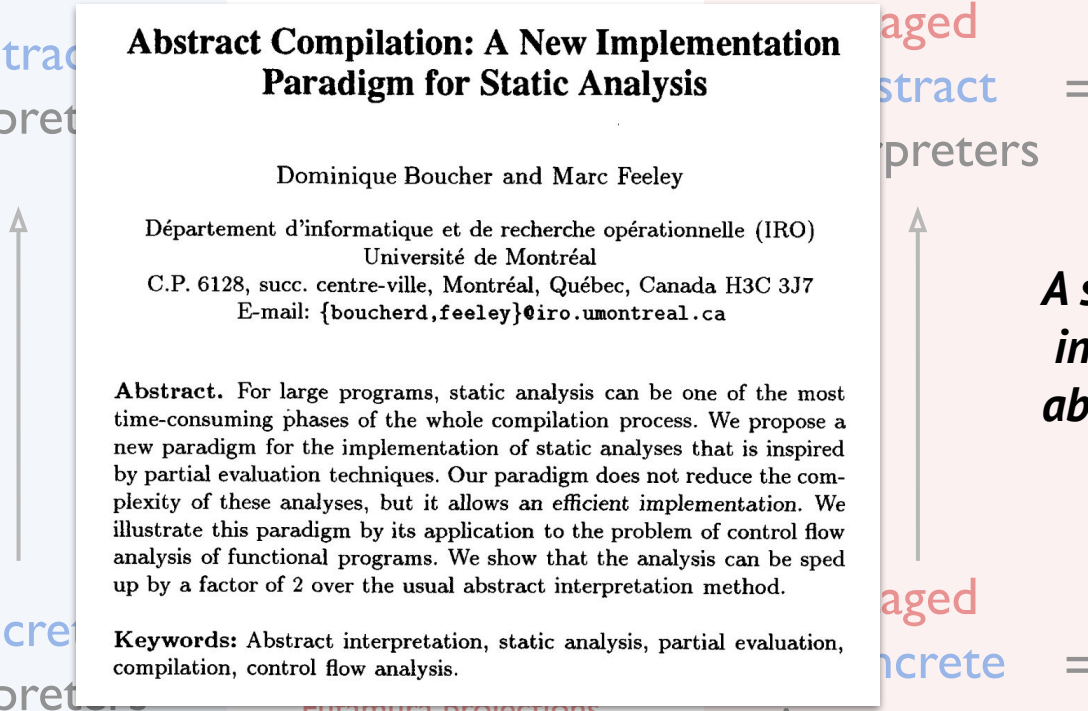
In this paper, we present a unifying framework that naturally extends the first Futamura projection from concrete interpreters to abstract interpreters. Our approach derives a sound staged abstract interpreter based on a generic interpreter with type-level binding-time abstractions and monadic abstractions. By using different instantiations of these abstractions, the generic interpreter can flexibly behave in one of four modes: as an unstaged concrete interpreter, a staged concrete interpreter, an unstaged abstract interpreter, or a staged abstract interpreter. As an example of abstraction without regret, the overhead of these abstraction layers is eliminated in the generated code after staging. We show that staging abstract interpreters is practical and useful to optimize static analysis, all while requiring less engineering effort and without compromising soundness. We conduct three case studies, including a comparison with Boucher and Feeley's abstract compilation, applications to various control-flow analyses, and a demonstration for modular analysis. We also empirically evaluate the effect of staging on the execution time. The experiment shows an order of magnitude speedup with staging for control-flow analyses.

A staged abstract interpreter is an abstract compiler

Performant Program Analysis by Compilation

abstrac
interpret

aged
stract = abstract
preters compilers



Abstract Compilation: A New Implementation Paradigm for Static Analysis

Dominique Boucher and Marc Feeley

Département d'informatique et de recherche opérationnelle (IRO)
Université de Montréal
C.P. 6128, succ. centre-ville, Montréal, Québec, Canada H3C 3J7
E-mail: {boucherd,feeley}@iro.umontreal.ca

Abstract. For large programs, static analysis can be one of the most time-consuming phases of the whole compilation process. We propose a new paradigm for the implementation of static analyses that is inspired by partial evaluation techniques. Our paradigm does not reduce the complexity of these analyses, but it allows an efficient implementation. We illustrate this paradigm by its application to the problem of control flow analysis of functional programs. We show that the analysis can be sped up by a factor of 2 over the usual abstract interpretation method.

Keywords: Abstract interpretation, static analysis, partial evaluation, compilation, control flow analysis.

SAS 1996

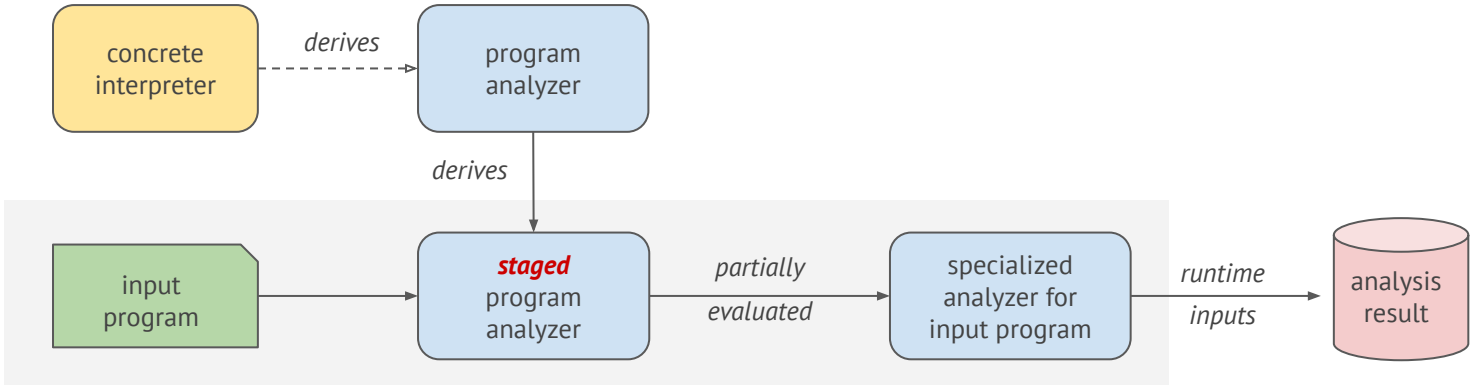
A staged abstract interpreter is an abstract compiler

concre
interpret

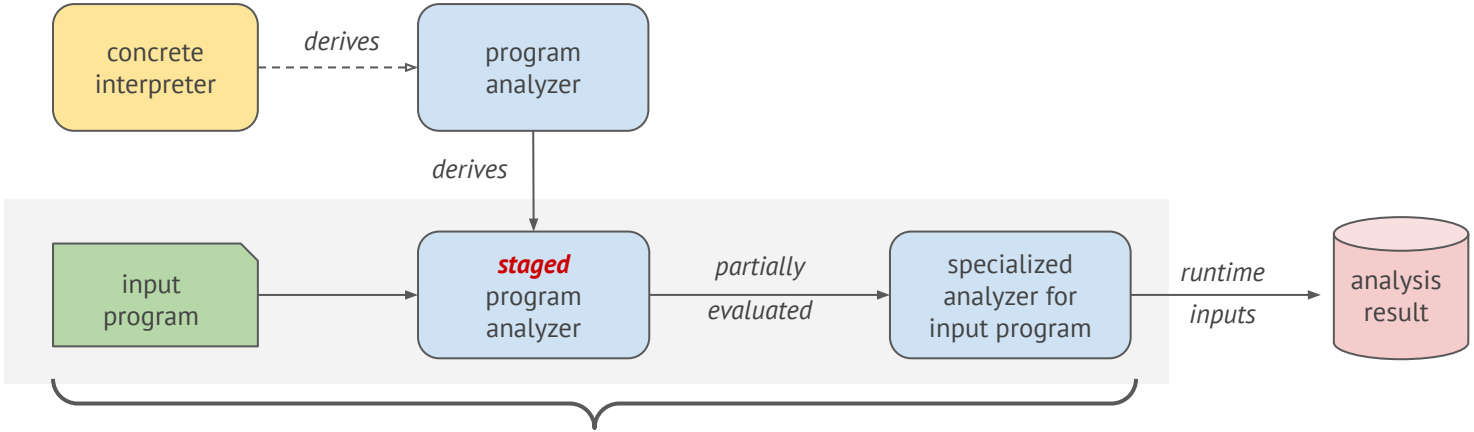
aged
ncrete = compilers
interpreters

Futamura projections

Performant Program Analysis by Compilation



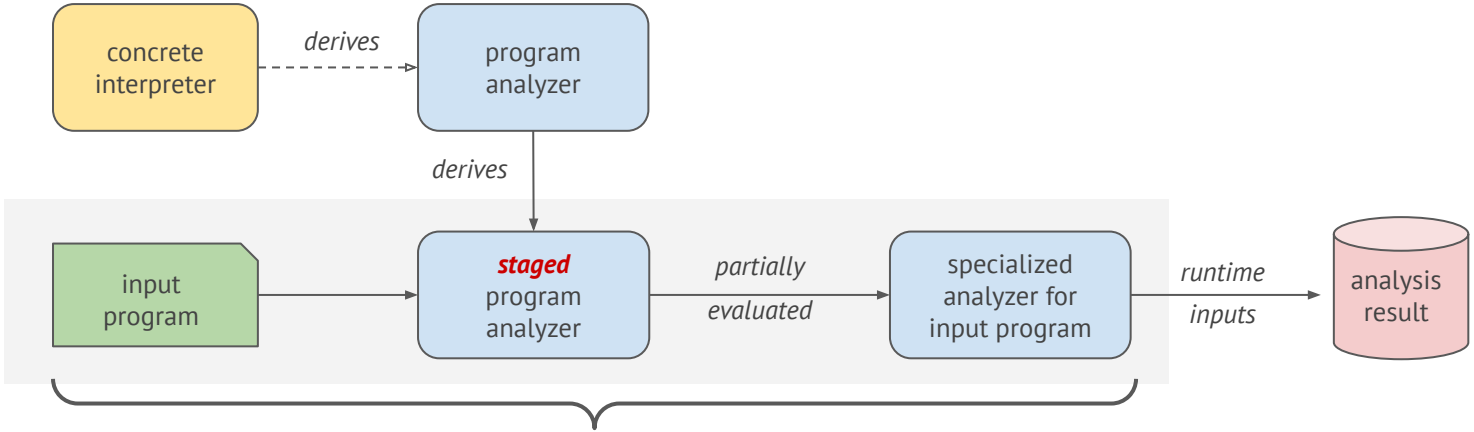
Performant Program Analysis by Compilation



the 1st Futamura projection

- ✓ Abstract interpreters for higher-order program analysis (CFA) [OOPSLA 19]
- ✓ Symbolic execution with algebraic effects/handlers [OOPSLA 20, FSE Demo 21]
- ✓ Parallel symbolic execution by generating CPS code [ICSE 23]

Performant Program Analysis by Compilation

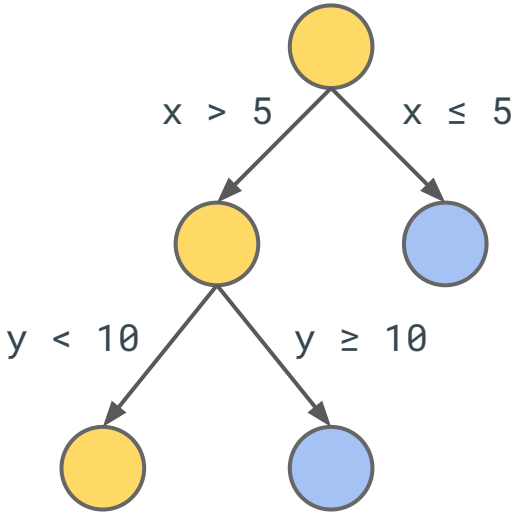


the 1st Futamura projection

- ✓ Abstract interpreters for higher-order program analysis (CFA) [OOPSLA 19]
- ✓ Symbolic execution with algebraic effects/handlers [OOPSLA 20, FSE Demo 21]
- ✓ **Parallel symbolic execution by generating CPS code [ICSE 23]**

Symbolic Execution

```
x = user_input()
y = user_input()
if (x > 5) {
    if (y < 10) {
        ... /* path 1 */
    } else {
        ... /* path 2 */
    }
} else {
    ... /* path 3 */
}
```



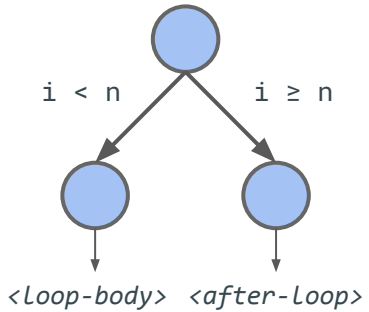
solver(x > 5 ∧ y < 10) = { x = 6, y = 9 }

Path Explosion, Worse

```
n = user_input() // i.e. symbolic
while (i < n) {
    <loop-body>
}
<after-loop>
```

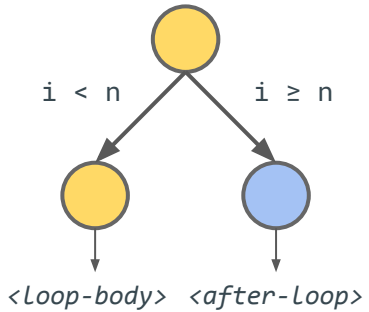
Path Explosion, Worse

```
n = user_input() // i.e. symbolic
while (i < n) {
    <loop-body>
}
<after-loop>
```



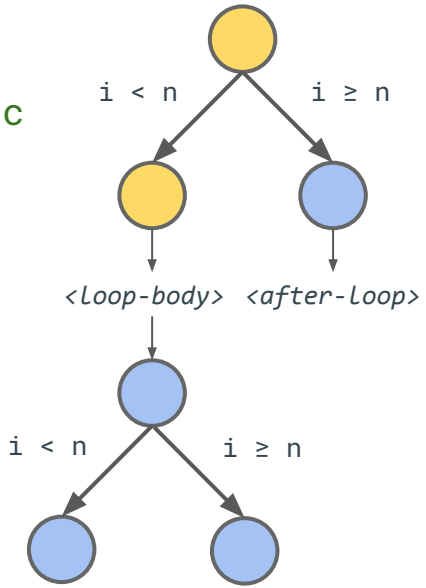
Path Explosion, Worse

```
n = user_input() // i.e. symbolic
while (i < n) {
    <loop-body>
}
<after-loop>
```



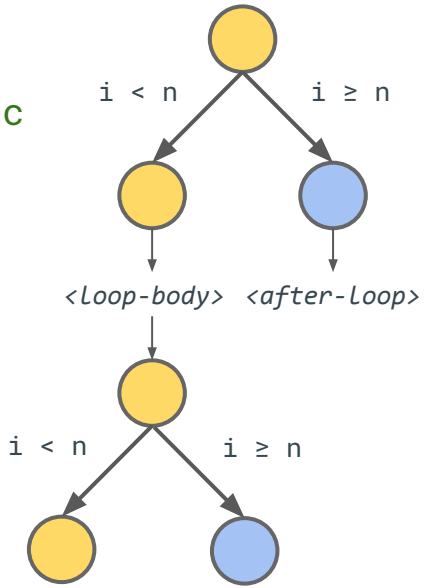
Path Explosion, Worse

```
n = user_input() // i.e. symbolic
while (i < n) {
    <loop-body>
}
<after-loop>
```



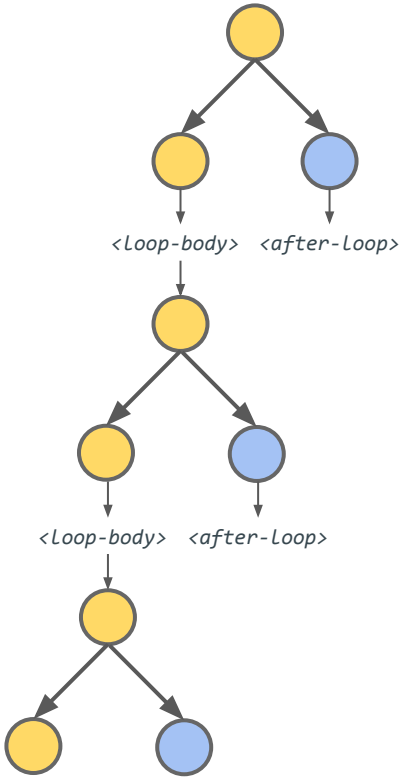
Path Explosion, Worse

```
n = user_input() // i.e. symbolic
while (i < n) {
    <loop-body>
}
<after-loop>
```



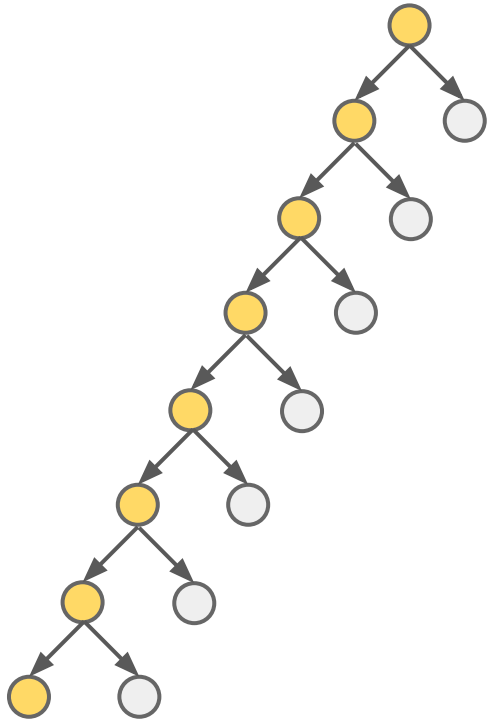
Path Explosion, Worse

```
n = user_input() // i.e. symbolic
while (i < n) {
    <loop-body>
}
<after-loop>
```



Path Explosion, Worse

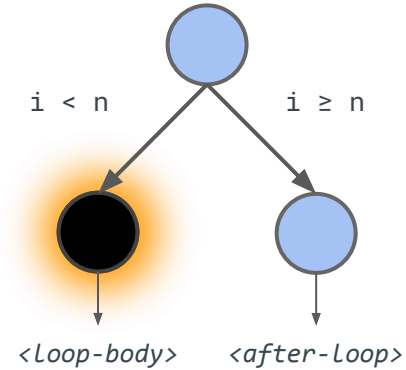
```
n = user_input() // i.e. symbolic
while (i < n) {
    <loop-body>
}
<after-loop>
```



. . .

Path Explosion, Worse

```
n = user_input() // i.e. symbolic
while (i < n) {
    <loop-body>
}
<after-loop>
```

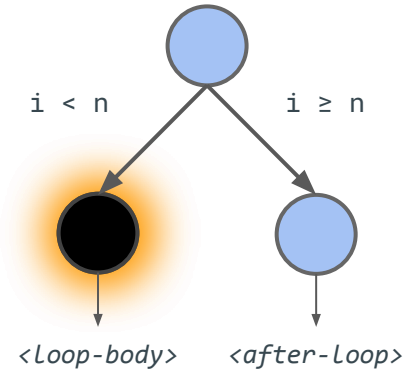


Problem: once running into the black hole,

we cannot effectively explore other parts of the program

Escaping the Black Hole

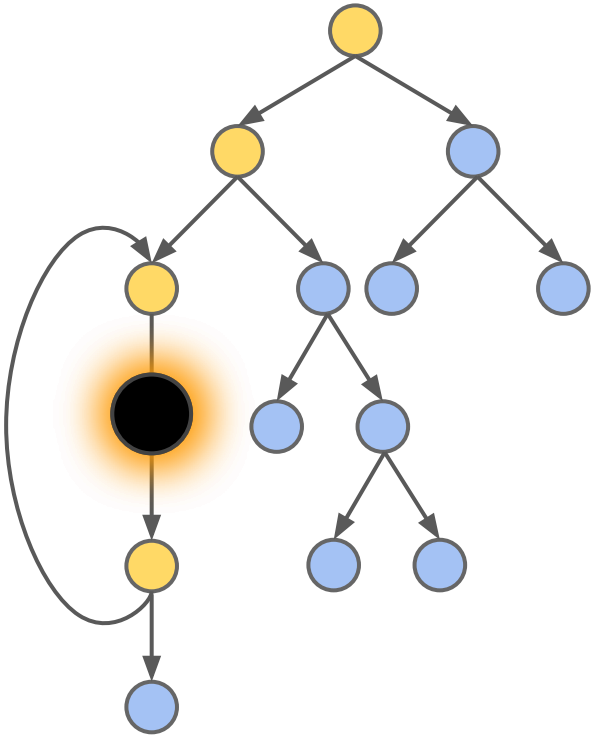
```
n = user_input() // i.e. symbolic
while (i < n) {
    <loop-body>
}
<after-loop>
```



Traditional wisdom: deploys clever path selection heuristics (e.g. in KLEE)

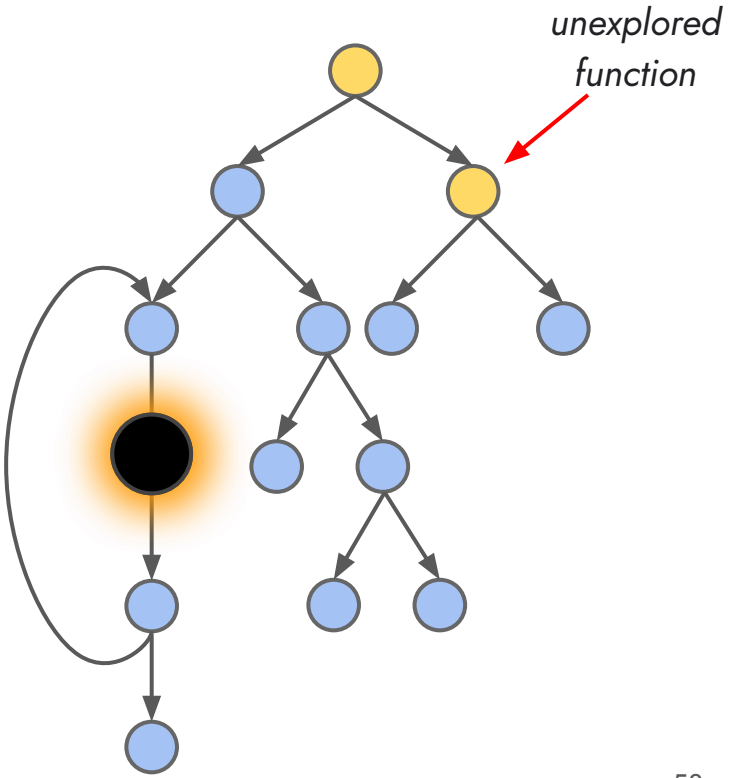
Escaping the Black Hole

- random state/path selection
- coverage-guided heuristics
- …



Escaping the Black Hole

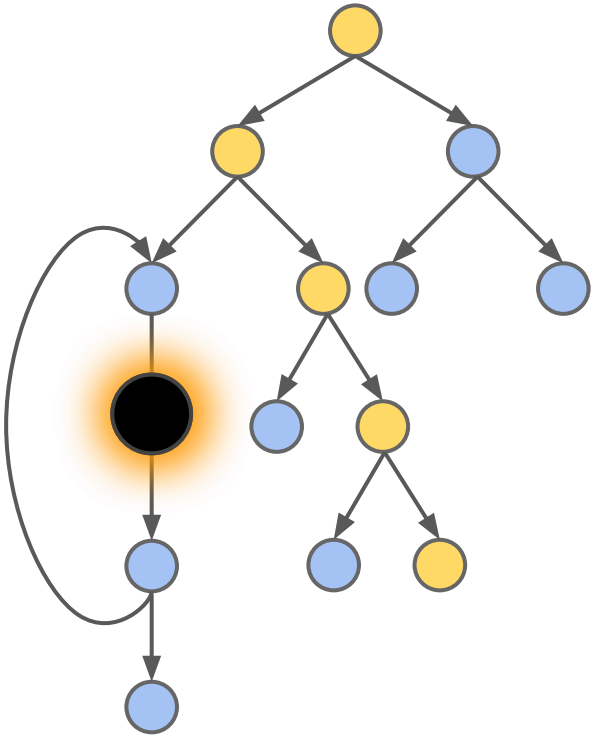
- random state/path selection
- *coverage-guided heuristics*
- …



Escaping the Black Hole

- random state/path selection
- coverage-guided heuristics
- …

Deploying path selection strategies needs the ability to *pause* and *resume* the execution of paths.



Symbolic Execution

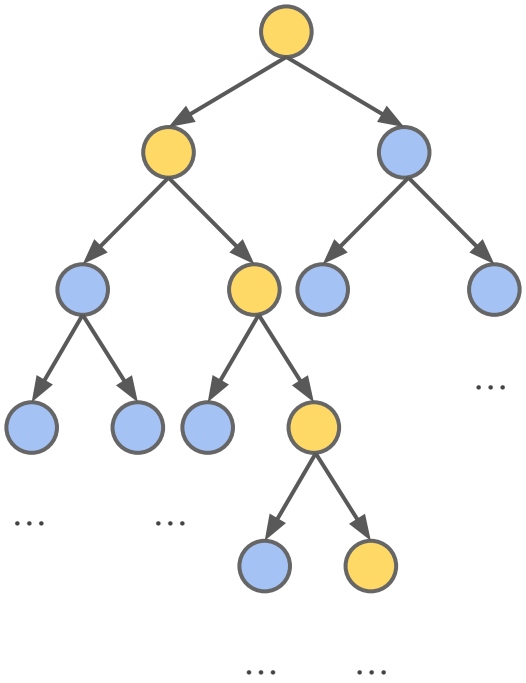
Concrete Execution

1 path

vs

Symbolic Execution

exponential number of independent paths



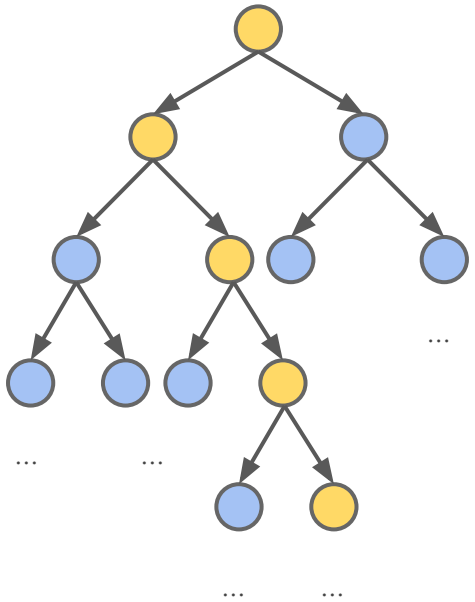
Goal: deriving compilers from high-level symbolic interpreters to generate efficient and effective symbolic-execution code.

Compiling Parallel SymExec with Continuation

💡 **Insight**

Viewing symbolic semantics as cooperative concurrency

- When forking, two paths execute concurrently
- Execution of forked path cooperates with a scheduler



Compiling Parallel SymExec with Continuation

💡 **Insight**

Viewing symbolic semantics as cooperative concurrency

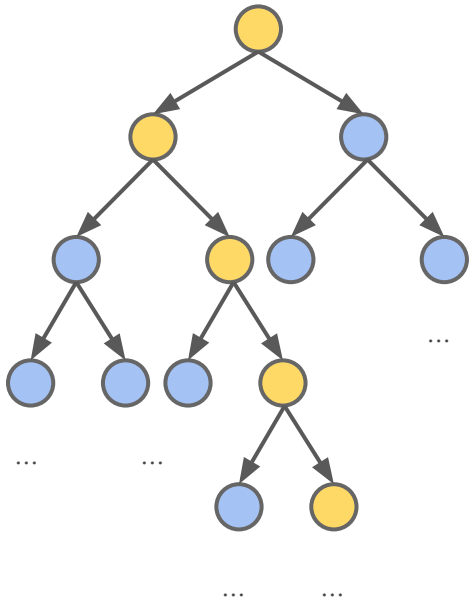
- When forking, two paths execute concurrently
- Execution of forked path cooperates with a scheduler

✓ **Solution**

Compiling to continuation-passing style by exposing control in the first-stage symbolic interpreter

✓ **GenSym**

An optimizing parallel symbolic-execution compiler

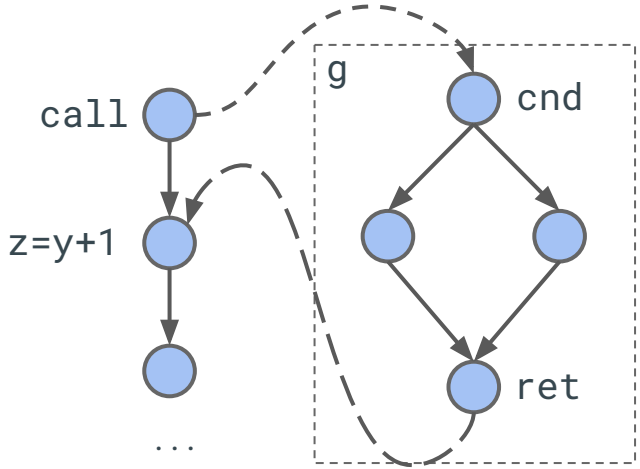


Compiling Parallel SymExec with Continuation

Represent the rest of the execution as a function k *in the generated code*

```
y = g()
z = y + 1
...
```

```
def g() =
  if (sym_cnd) {
    x = 42
  } else {
    x = 100
  }
  return x
```



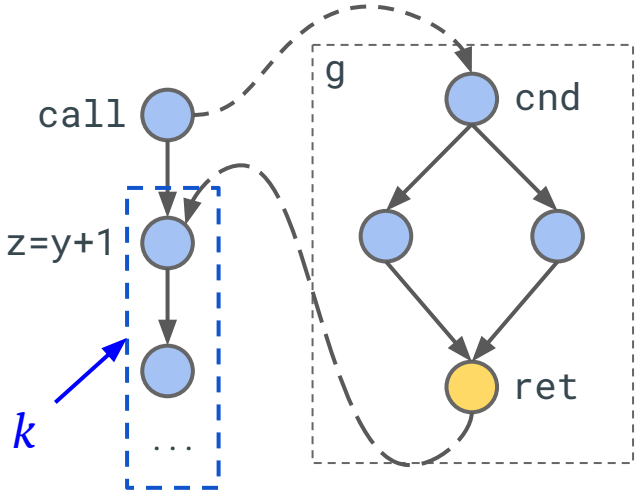
static control-flow graph

Compiling Parallel SymExec with Continuation

Represent the rest of the execution as a function *k* *in the generated code*

```
y = g()
z = y + 1
...
```

```
def g() =
  if (sym_cnd) {
    x = 42
  } else {
    x = 100
  }
  return x
```



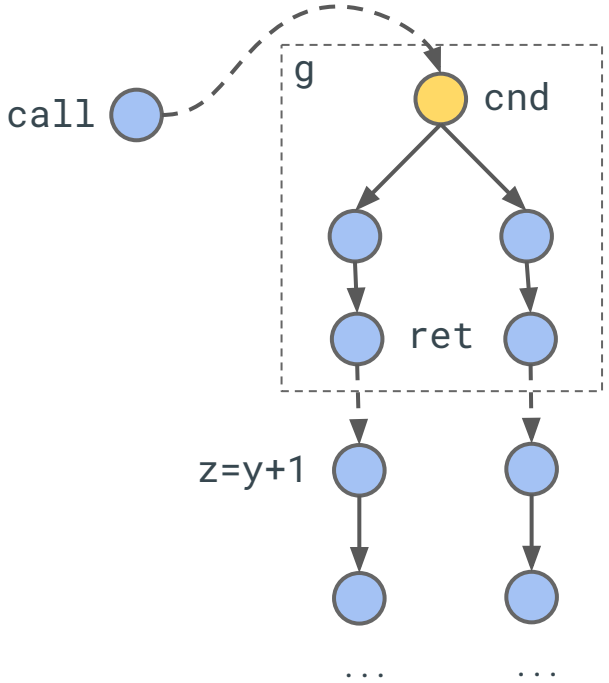
static control-flow graph

Compiling Parallel SymExec with Continuation

Represent the rest of the execution as a function k *in the generated code*

```
y = g()
z = y + 1
...
```

```
def g() =
  if (sym_cnd) {
    x = 42
  } else {
    x = 100
  }
  return x
```



Invoke and fork

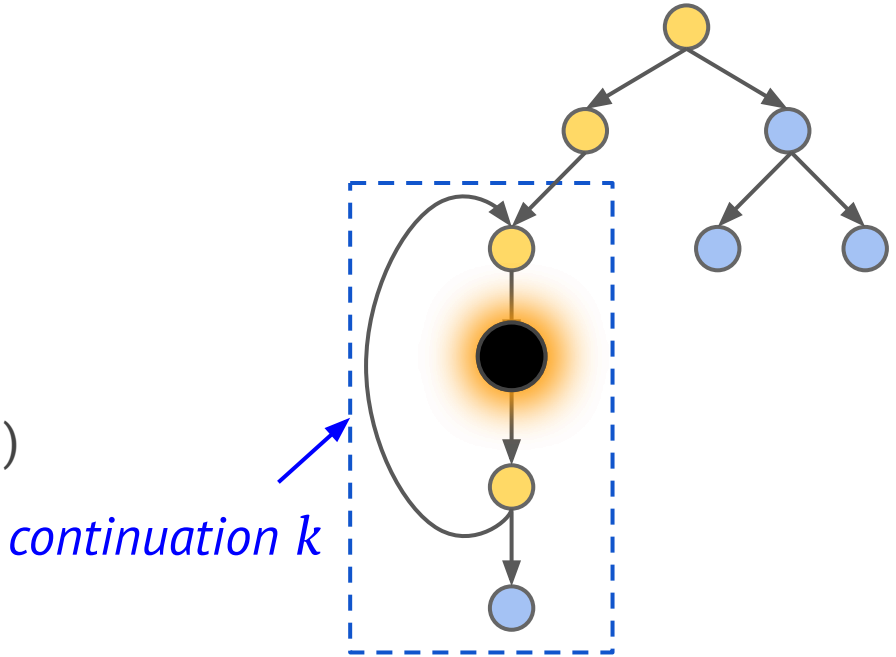
k(s1); k(s2)

Compiling Parallel SymExec with Continuation

```
n = user_input()
while (i < n) {
    <loop-body>
}
<after-loop>
```

Save and pause

```
scheduler.put(() => k(s))
```

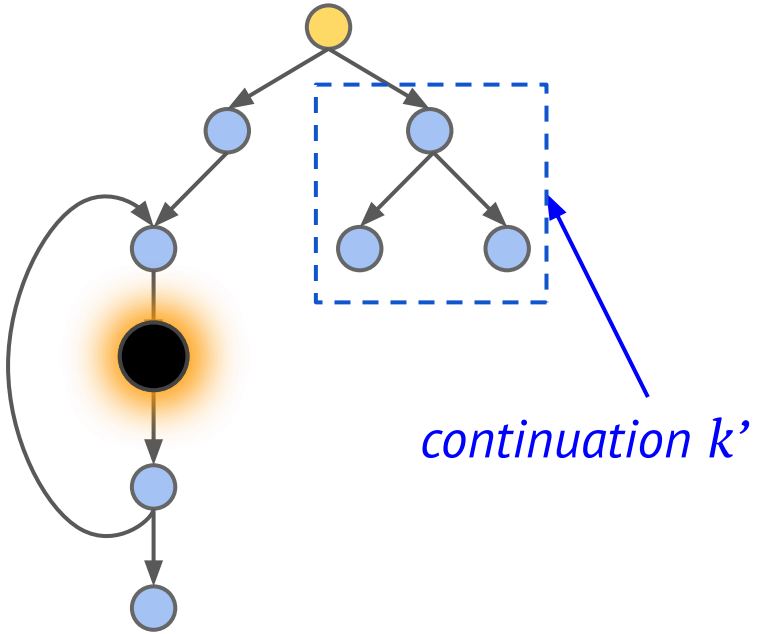


Compiling Parallel SymExec with Continuation

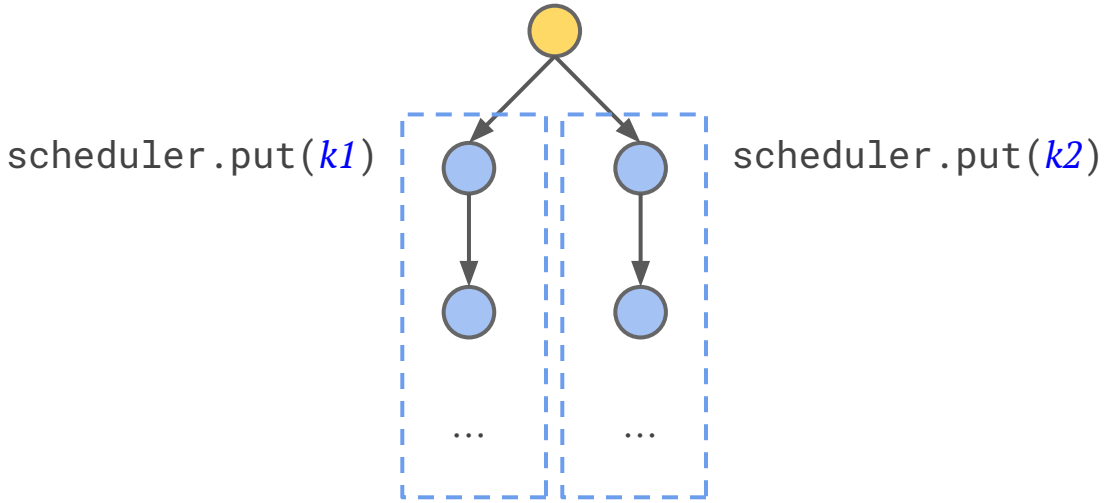
```
n = user_input()
while (i < n) {
    <loop-body>
}
<after-loop>
```

Dispatch and resume

k' = scheduler.get(); *k'*()



Compiling Parallel SymExec with Continuation



Compiling Parallel SymExec with Continuation

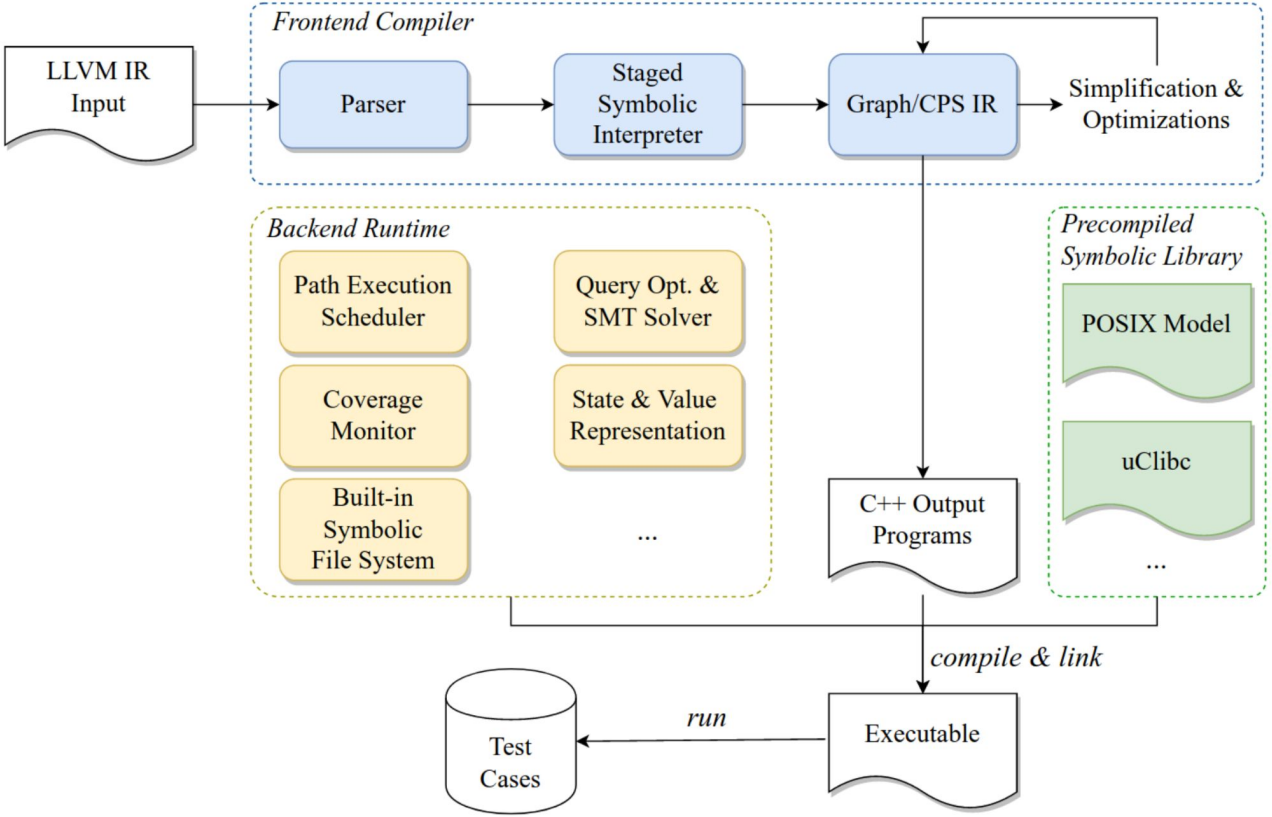
represent the rest of execution as a function *k in the generated code*

- *invoke and fork*
  `k(s1); k(s2)`
- *save and pause*
  `scheduler.put(() => k(s))`
- *dispatch and resume*
  `k = scheduler.get(); k()`
- *dispatch in parallel*

Compiling Parallel SymExec with Continuation

- **GenSym**
  - Handles a large subset of LLVM IR
  - Simulates path forking and switching using continuations
  - Written in Scala/LMS and generates C++ code

Compiling Parallel SymExec with Continuation



An Example

```
 4 define i32 @power(i32 x, i32 n) {
 5   b0: cmp = icmp eq i32 n, 0
 6       br i1 cmp, label b2, label b1
 7   b1: subv = sub i32 n, 1
 8       retv = call i32 @power(i32 x, i32 subv)
 9       mulv = mul i32 x, retv
10       br label b2
11   b2: r = phi i32 [mulv, b1], [1, b0]
12       ret i32 r
13 }
```

LLVM source (power func)

```
15 using Cont = function<void(S, V)>;  // the continuation type
16 void power_b0(S s, List[V] args, Cont k) { // compiled block ‘b0‘
17   s.assign_seq(List("x", "n"), args);
18   V cmp = args[1] == IntV(0L, 32);
19   s.set_cur_block("b0"); // record the current block (for phi node)
20   if (cmp.is_conc()) {   // concrete branch condition
21     if (cmp.value() == 1) power_b1(s, k); // jump to block ‘b1‘
22     else power_b2(s, k);                  // jump to block ‘b2‘
23   } else { // symbolic branch condition
24     // yield the control to scheduler and add two new tasks to schedule
25     schedule(task(s, cmp, power_b2, k), task(s.fork(), !cmp, power_b1, k));
26   }
27 }
28 void power_b1(S s0, Cont k) { // compiled block ‘b1‘
29   V subv = s0.lookup("n") - IntV(1, 32);
30   List[V] args = List(s0.lookup("x"), subv);
31   // recursively call ‘power‘ with a new continuation
32   power_b0(s0, args, [=](S s1, V retv) {
33     s1.assign("mulv", retv * s1.lookup("x"));
34     s1.set_cur_block("b1");
35     power_b2(s1, k);
36   });
37 }
38 void power_b2(S s, Cont k) { // compiled block ‘b2‘
39   V r = s.last_block() == "b1" ? s.lookup("mulv") : IntV(1L, 32);
40   k(s, r);
41 }
```

GenSym generated code

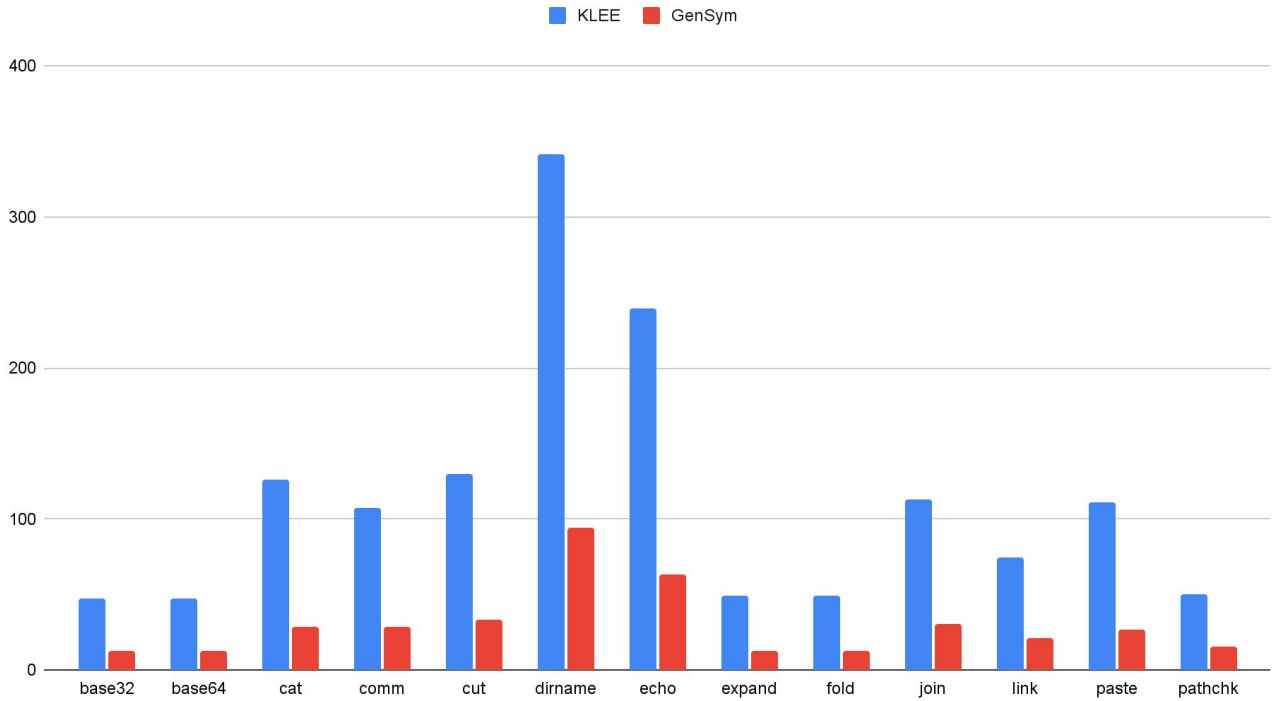
Compiling Parallel SymExec with Continuation

- **GenSym**
  - Handles a large subset of LLVM IR
  - Simulates path forking and switching using continuations
  - Written in Scala/LMS and generates C++ code

- **Benchmarks**
  - A subset of GNU Coreutils (using POSIX file system and uClibc library)
  - Average program size 28k LOC of LLVM IR

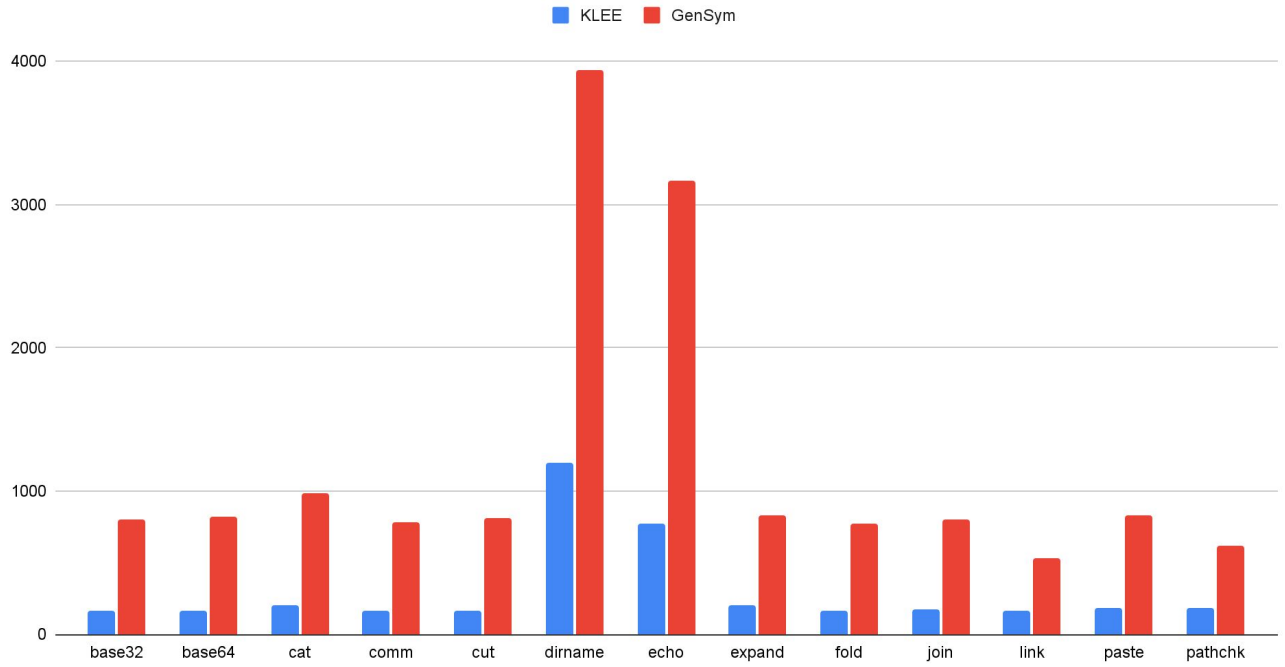
- **Performance evaluation**
  - Compared with KLEE, a state-of-the-art symbolic interpreter

Compiling Parallel SymExec with Continuation



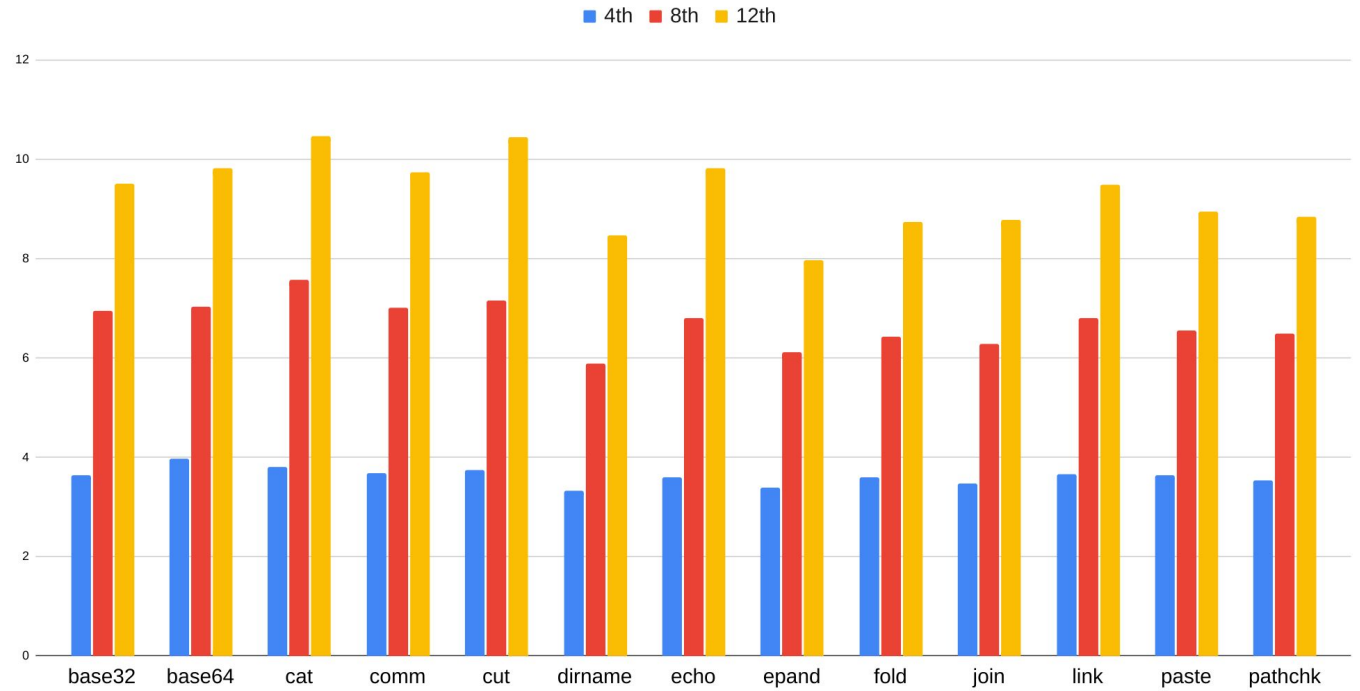
Single-thread pure execution time (sec)
excluding solver: **4x faster on avg**

Compiling Parallel SymExec with Continuation



Single-thread path throughput
(paths per second) of 1-hour running: **4.3x on avg**.

Compiling Parallel SymExec with Continuation



Speedups using more threads

4th - 3.6x
8th - 6.7x
12th - 9.3x

Conclusion & Future Work

- Metaprogramming can help the construction/performance of program analyzers
  - Bridge the gap between static analysis spec. and implementations
  - Apply to wide range of program analyses
  - Refunctionalization/defunctionalization, partial evaluation & staging, CPS transformation, code generation, etc.

Conclusion & Future Work

- Metaprogramming can help the construction/performance of program analyzers
  - Bridge the gap between static analysis spec. and implementations
  - Apply to wide range of program analyses
  - Refunctionalization/defunctionalization, partial evaluation & staging, CPS transformation, code generation, etc.
- Future work
  - Staging relational & numerical abstract domains
  - Compositional program analysis
  - Declarative specifications of abstract interpretation to efficient implementations
  - Theoretical foundation

Conclusion & Future Work

Thank you!

- Metaprogramming can help the construction/performance of program analyzers
  - Bridge the gap between static analysis spec. and implementations
  - Apply to wide range of program analyses
  - Refunctionalization/defunctionalization, partial evaluation & staging, CPS transformation, code generation, etc.
- Future work
  - Staging relational & numerical abstract domains
  - Compositional program analysis
  - Declarative specifications of abstract interpretation to efficient implementations
  - Theoretical foundation