Metaprogramming for Program Analyzers

Guannan Wei

with Oliver Bračevac, Shangyin Tan, Yuxuan Chen, and Tiark Rompf

August 2020, PurPL Retreat

guannanwei@purdue.edu
https://continuation.passing.style

What is metaprogramming?

- Metaprogramming treats other programs as data objects.

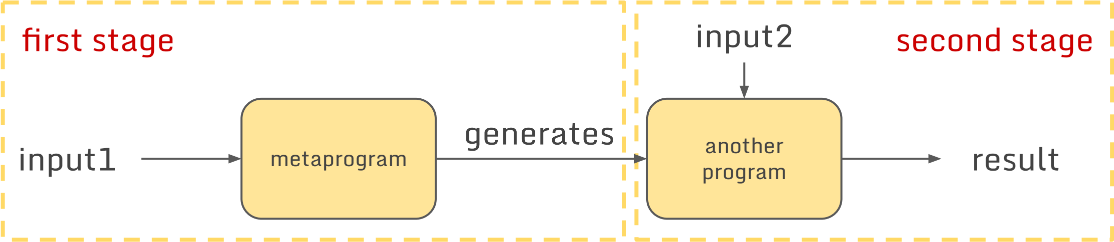
- In general, metaprograms *analyze*, *interpret*, *transform* and *generate* other programs.

What is metaprogramming?

- Metaprogramming treats other programs as data objects.
- In general, metaprograms *analyze*, *interpret*, *transform* and *generate* other programs.
- Generative metaprogramming:
  macros, templates, multi-stage programming (the LMS framework) ...

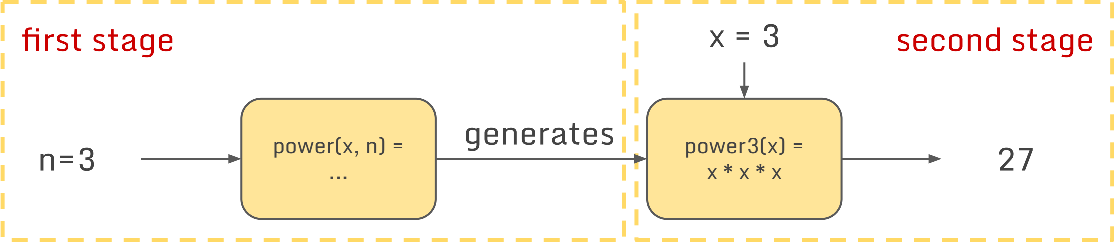
What is metaprogramming?

- Metaprogramming treats other programs as data objects.
- In general, metaprograms *analyze*, *interpret*, *transform* and *generate* other programs.
- Generative metaprogramming:
  macros, templates, multi-stage programming (the LMS framework) ...



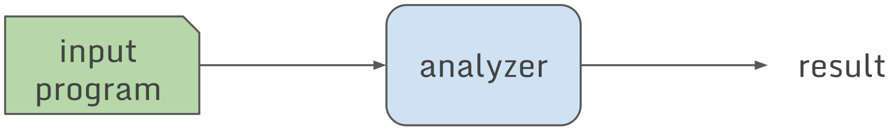
What is metaprogramming?

- Metaprogramming treats other programs as data objects.
- In general, metaprograms *analyze*, *interpret*, *transform* and *generate* other programs.
- Generative metaprogramming:
  macros, templates, multi-stage programming (the LMS framework) ...



What is a program analyzer?

- A (static) program analyzer computes runtime behaviors/properties of a program without running it.
- A program analyzer is also a metaprogram -- in the sense that itself is a program and it analyzes another program.

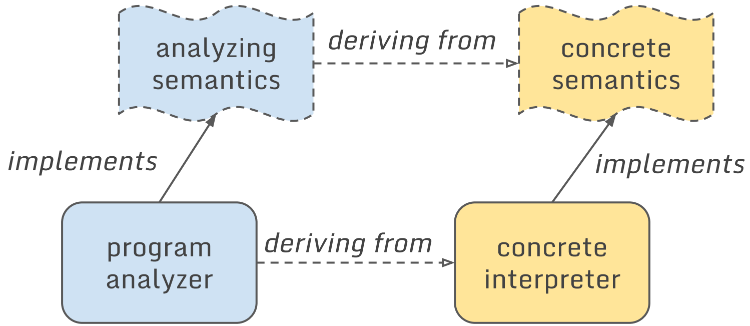


e.g.:
- termination
- control flow
- data flow
- numerical bounds
- validity of assertions
- ...

What is a program analyzer?

- A (static) program analyzer computes runtime behaviors/properties of a program without running it.
- A program analyzer is also a metaprogram -- in the sense that itself is a program and it analyzes another program.

- A *semantic* view of program analyses and analyzers:
    - program analyzers approximately simulate the concrete execution

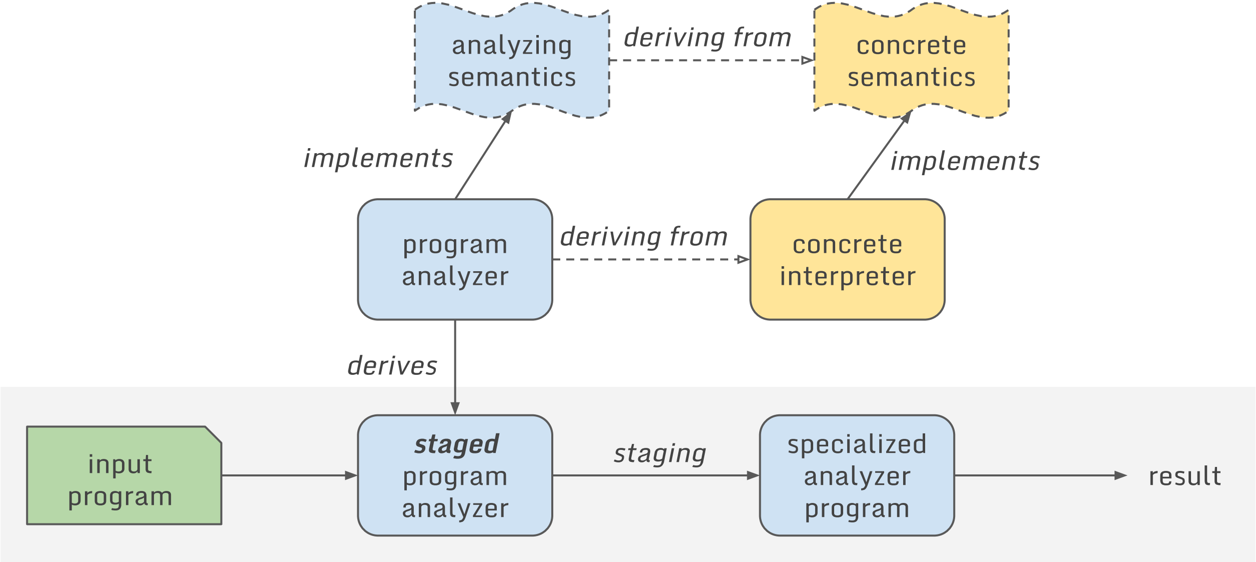


When building program analyzers, what can metaprogramming abstractions do?

- Multi-stage programming + functional programming can improve the construction, performance, and flexibility of program analyzers.

When building program analyzers, what can metaprogramming abstractions do?

- Multi-stage programming + functional programming can improve the construction, performance, and flexibility of program analyzers.



Our Recent Study

- Abstract interpreters and control-flow analysis for functional languages.

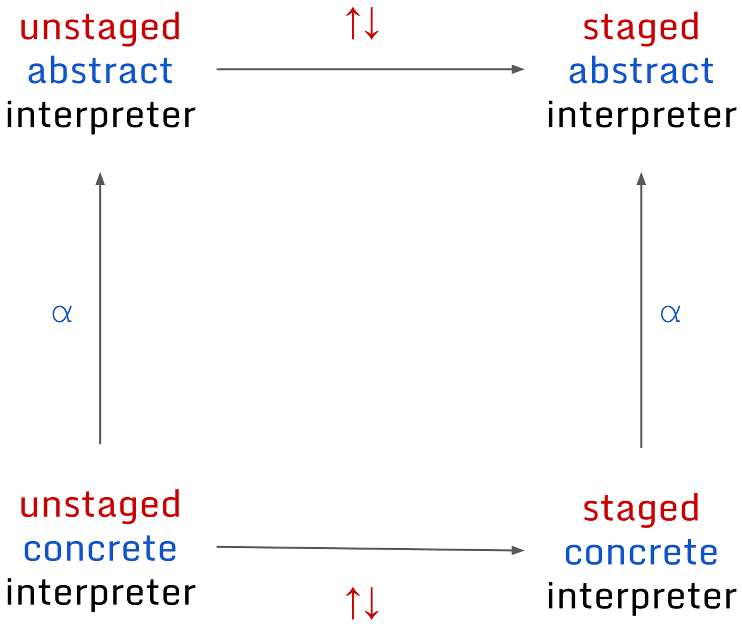
  Staged Abstract Interpreters (OOPSLA 2019)
  Guannan Wei, Yuxuan Chen, and Tiark Rompf

- Symbolic execution engines for imperative languages.

  Compiling Symbolic Execution with Staging and Algebraic Effects
  (Conditionally accepted, OOPSLA 2020)
  Guannan Wei, Oliver Bracevac, and Tiark Rompf

Staged Abstract Interpreter

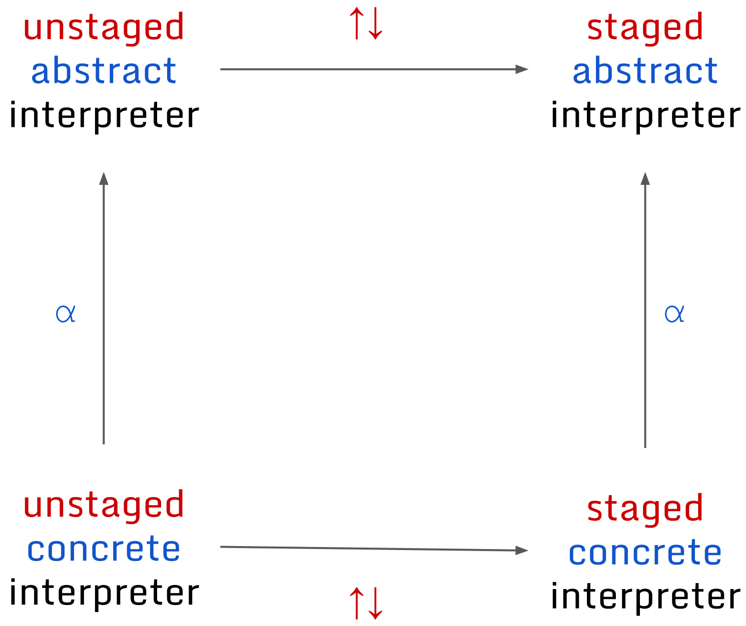
- Constructed a generic monadic interpreter that abstracts over *value domains* and *binding-times*.



Staged Abstract Interpreter

- Constructed a generic monadic interpreter that abstracts over *value domains* and *binding-times*.

```
def eval(ev: EvalFun)(e: Expr): Ans =
  e match {
    case Var(x) ⇒ for {
      ρ ← ask_env
      σ ← get_store
    } yield get(σ, ρ, x)
    case Lam(x, e) ⇒ for {
      ρ ← ask_env
    } yield close(ev)(Lam(x, e), ρ)
    case App(e1, e2) ⇒ for {
      v1 ← ev(e1)
      v2 ← ev(e2)
      rt ← ap_clo(ev)(v1, v2)
    } yield rt
    ...
  }
```



Staged Abstract Interpreter - Key Result

- Code sharing with concrete interpreter brings more confidence of correctness.
- The staged abstract interpreter is also a ***compiler*** that takes program as input and generates low-level code according to the abstract semantics.
- The generated code is modular and reusable, and has no interpretation and monadic overhead.

- Performance evaluation on control-flow analysis of a subset of Scheme
  - 0-CFA with/without store widening: on average ~10 times faster compared with the unstaged analyzer

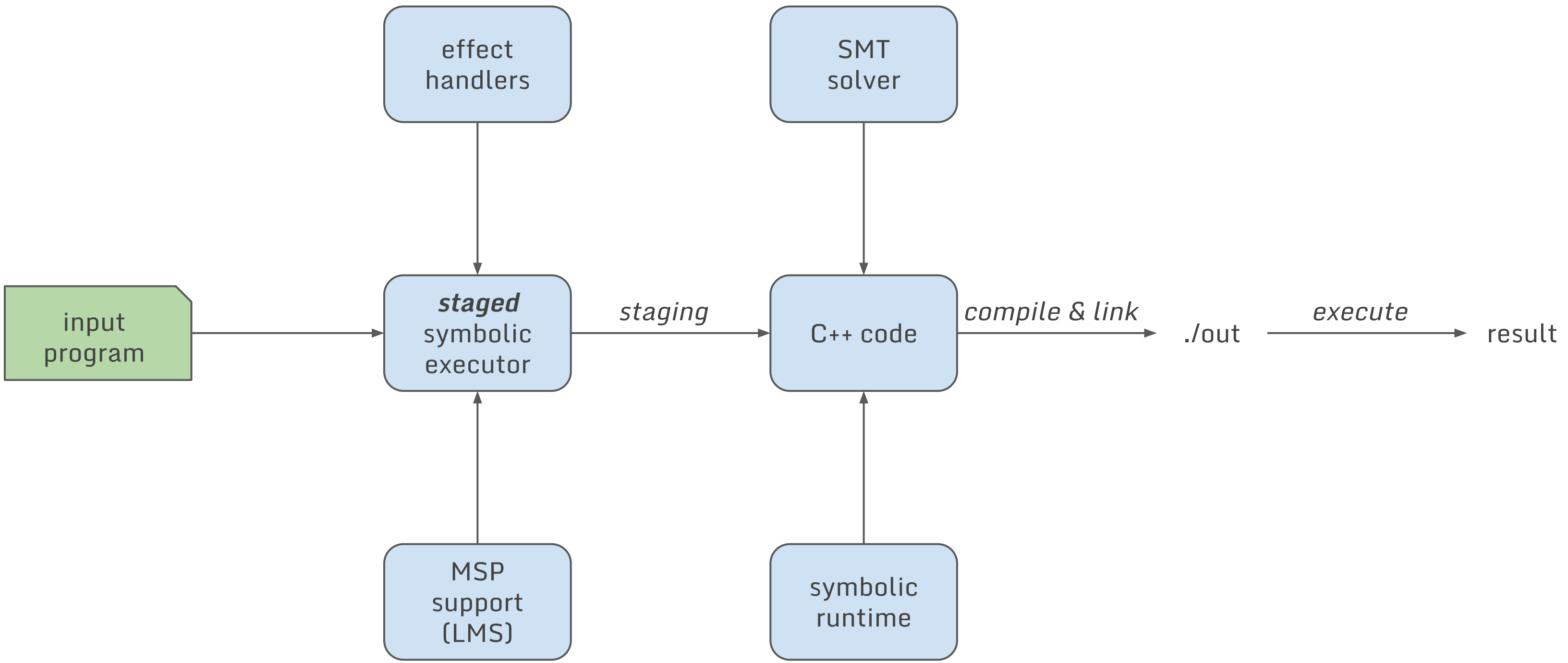
Compiling Symbolic Execution

- Applying staging to symbolic execution engine

Compiling Symbolic Execution

- Applying staging to symbolic execution engine, and moreover, we
  - improve efficiency by generating/staging to C++ code
  - integrate with SMT solver APIs directly in the generated code
  - use *algebraic effects and handlers* to abstract over nondeterminism behavior, which gives us more flexibility over path selection strategy

Compiling Symbolic Execution



Compiling Symbolic Execution - Key Result

- Using algebraic effects and effect handlers enable flexibly interprets the nondeterminism effects -- resulting in different path exploration strategies
  - depth-first, breath-first, random, fair random sampling, etc.

- Build a prototype for a subset of LLVM IR
- Performance evaluation on micro benchmarks
  - The generated code (C++) run 3~20x faster than unstaged counterpart (Scala)
  - The generated code (C++) outperforms ~17%~60% than KLEE (interpretation, C++)
  - Still a lot of room to improve!

When building program analyzers, what can metaprogramming abstractions do?

- Multi-stage programming + functional programming can improve the construction, performance, and flexibility of program analyzers.

- Recipe:
  - Deriving the analyzer from concrete definitional interpreters,
    expressing the analyzing semantics with high fidelity and confidence of correctness.
  - Turning the analyzer to a compiler via staging and the 1st Futamura projection,
    generating low-level code and eliminating the interpretation overhead.
  - Using type/effect system to model and abstract over the behavior of analyzers,
    improving the modularity and flexibility of analyzers.