Optimization

NI(E)-GEN, Spring 2023

https://courses.fit.cvut.cz/NI-GEN

ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE

FIT

Why?

- minimize overheads of the execution process (calls, jumps, etc.)
- minimize code size (beter Icache locality, smaller code size?)
- cache already computed values where possible (faster, smaller code)
- rewrite common sequences into more performant variants
- parallelization

What For?

- speed – the go-to variant
- size – where program size is a constraint (embedded)
- energy efficiency – very desired but next to impossible to quantify

Why not?

- Slow is Good
  - optimizations often include the side-channel attack surface
  - code can be deliberately made slow (even randomly slow) to prevent those

- Slow is Predictable
  - Instruction latencies can be used as explicit timing method (embedded)

- Slow is Robust
  - Compilers often include extra checks, asserts, or memory hints in non-optimized code for easier debugging

Why not?

- Debugging
  - Code layout means some breakpoints will not work
  - Optimized values mean limited introspection
  - Inlining changes stack layout
  - Advanced optimizations make the code look absolutely cryptic

- Compiler Errors
  - Sadly, optimizations are known to introduce subtle bugs
  - Turning them off for problematic parts of code often results in correct compilation

Basic Concepts

- Analysis
  - Passes that do not change the code, but gather semantic information about it
- Optimizations
  - Code changing passes, may invalidate optimizations
- Normalization
  - Often optimizations are done for “no good reason”, but they make the code more regular which in turns makes rest of the process easier
- Undefined Behavior
  - makes compiler happy – exact semantics does not have to be observed in undefined behavior situations, which allows for more optimization opportunities

Data-flow Optimizations

- Common Subexpression Elimination (CSE)
  - Reuses previously calculated identical values
- Global Value Numbering (GVN)
  - SSA-based version of CSE
- Strength Reduction
  - Replaces costly operations with identical but less expensive ones

Data-flow Optimizations

- Constant Folding & Propagation
  - Replaces variables with constants
- Bounds Check Elimination
  - Removes automatic bounds checking where possible

Code Removal

- Dead Instruction Elimination (DIE)
  - Removes instruction w/o sideeffects and no uses
- Dead Store Elimination (DSE)
  - Removes store of a never-read value (language dependent)
- Dead Code Elimination (DCE)
  - Removes unreachable code blocks

Function Call Optimizations

- Inlining – extremely useful
  - Removes function call overhead
  - But often more importantly increases the optimizer’s context

- Tail Call Optimization
  - requirement for functional languages

Loop Optimizations

- most execution time spent in loops, which makes loop optimizations the most important
- main themes are removal of loop overhead via various transformations (relatively easy)
- and enabling parallelism (very hard)

Loop Optimizations

- Loop unrolling
  - Reduces loop overhead by lowering loop iterations by copying its body

- Loop Inversion
  - Replaces while loop with guard and do-while

- Loop Interchange
  - Swaps nested loops for better cache locality

Loop Optimizations

- Loop Splitting & Peeling
  - single loop split into multiple loops iterating over disjoint contiguous areas of the previous range
  - improves cache locality, highlights potential parallelism
- Loop Unswitching
  - conditionals in the loop that will be the same for the whole loop are moved out of the loop and the rest of the loop is duplicated
- Loop Reversal
  - reverse the order of the loop. This does not change performance of the program, but rather enables other opimizations by potentially reducing dependencies

Loop Optimizations

- Loop Fission
  - breaks loop into multiple loops with distinct bodies (helps reference locality for the smaller bodies)
- Loop Fusion
  - combine multiple loops iterating the same amount of times / over same data as long as they do not reference each other
- Loop Invariant Code Motion
  - code that does not rely on the induction variable can be scheduled before the loop
  - this is only true if the loop is executed at least once, i.e. ideal for do-whole loops.

Speculative Optimizations

- not much used in ahead-of-time compilers
- extremely important for JITs and dynamic languages

- devirtualization
- non-null assumption
- type assumption
- value assumption

Partial Evaluation

```
int add(int i, int j) {
    return i + j;
}

int y = 0;
for (int x = 0; x < VERY_LARGE_N; ++x)
    y += add(1, x);
```

Partial Evaluation

```
int add(int i, int j) {
    return i + j;
}

int y = 0;
for (int x = 0; x < VERY_LARGE_N; ++x) {
    int i = 1;
    int j = x;
    y += i + j;
}
```

Partial Evaluation

```
int add(int i, int j) {
    return i + j;
}

int y = 0;
for (int x = 0; x < VERY_LARGE_N; ++x) {
    int i = 1;
    int j = x;
    y += 1 + j;
}
```

Partial Evaluation

```
int add(int i, int j) {
    return i + j;
}

int y = 0;
for (int x = 0; x < VERY_LARGE_N; ++x) {
    int i = 1;
    int j = x;
    y += 1 + x;
}
```

Partial Evaluation

```
int y = 0;
for (int x = 0; x < VERY_LARGE_N; ++x)
    y += 1 + x;
```

Partial Evaluation

```
int add(int i, int j) {
    return i + j;
}

int y = 0;
for (int x = 0; x < VERY_LARGE_N; ++x)
    y += add(x, 1);
```

Partial Evaluation

```
int add(int i, int j) {
    return i + j;
}

int y = 0;
λ inc = add(1);
for (int x = 0; x < VERY_LARGE_N; ++x)
    y += inc(x);
```

Partial Evaluation

```
#include <functional>
using namespace std::placeholders;

int add(int i, int j) {
    return i + j;
}

int y = 0;
auto inc = std::bind(add, 1, _2);
for (int x = 0; x < VERY_LARGE_N; ++x)
    y += inc(x);
```

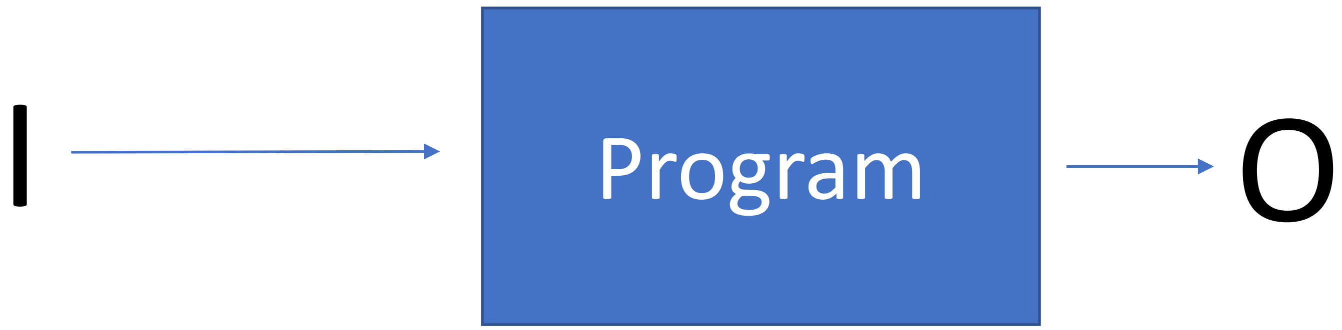
Partial Evaluation

- generalization of the optimization by specialization
- creates new version of the function where some of its arguments are bound to constants, while others remain
- this increases the context for the optimizer in the specialized version

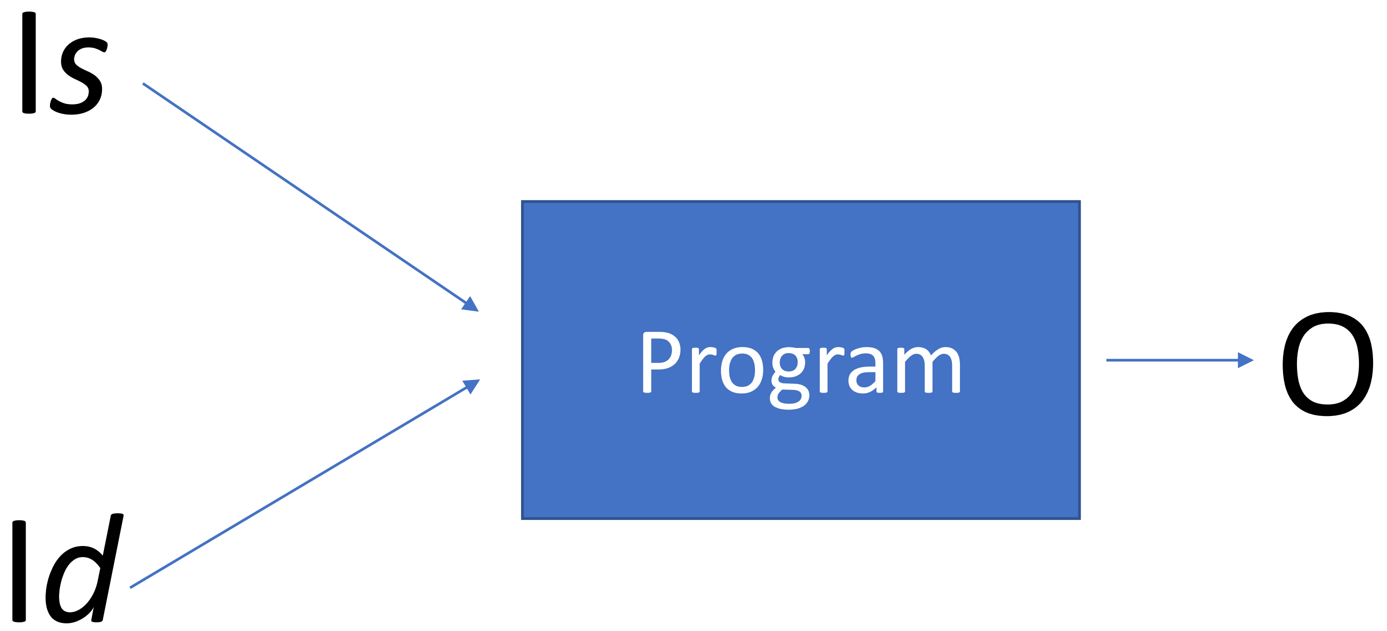
Partial Evaluation

- generalization of the optimization by specialization
- creates new version of the function where some of its arguments are bound to constants, while others remain
- this increases the context for the optimizer in the specialized version
- can be generalized further to whole programs

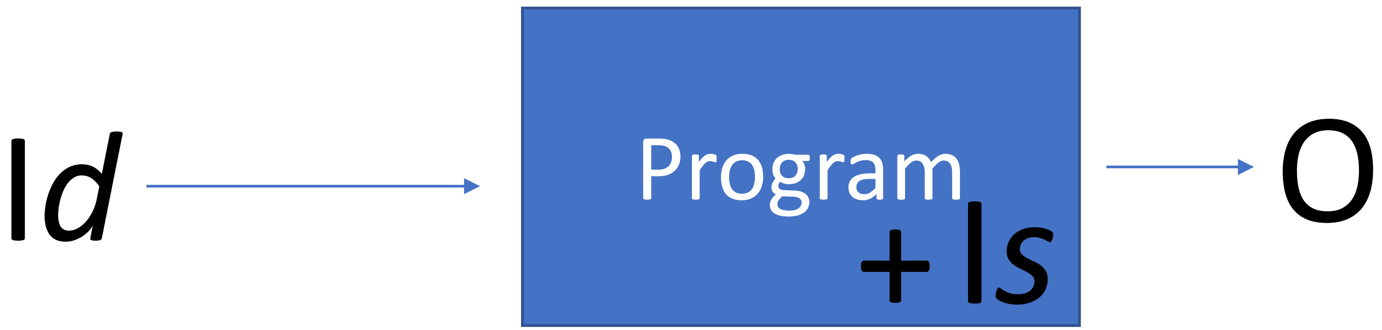
Partial Evaluation



Partial Evaluation

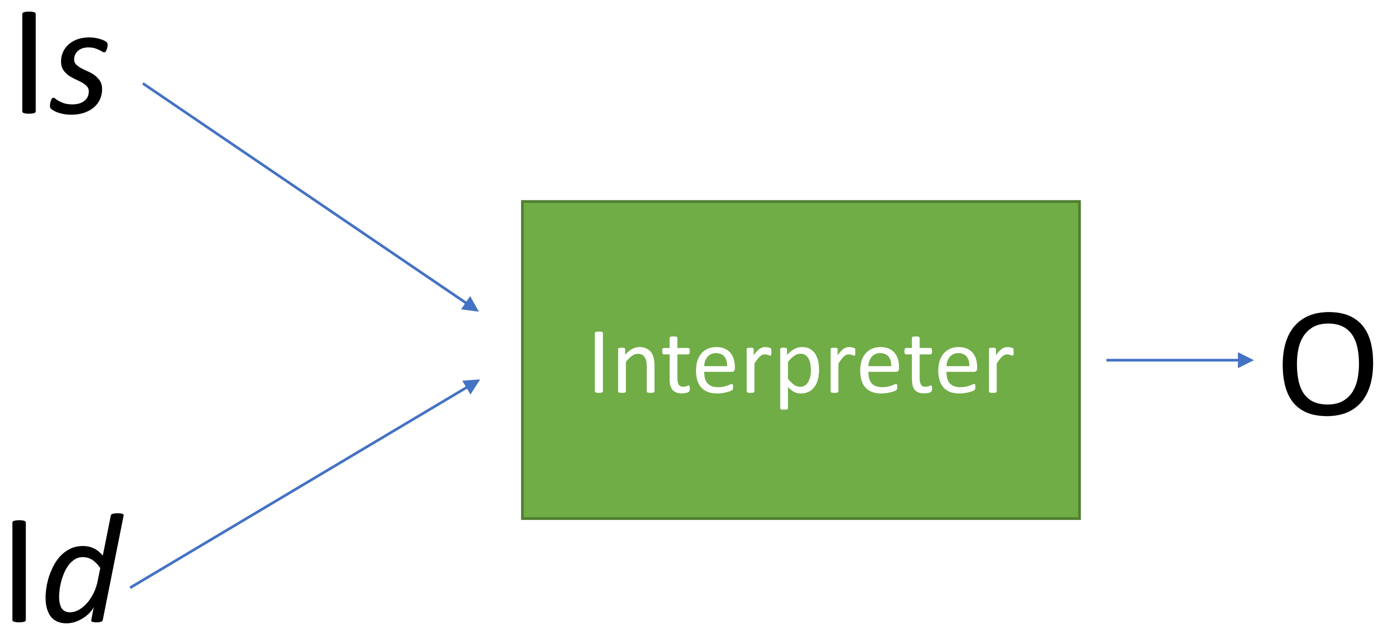


Partial Evaluation

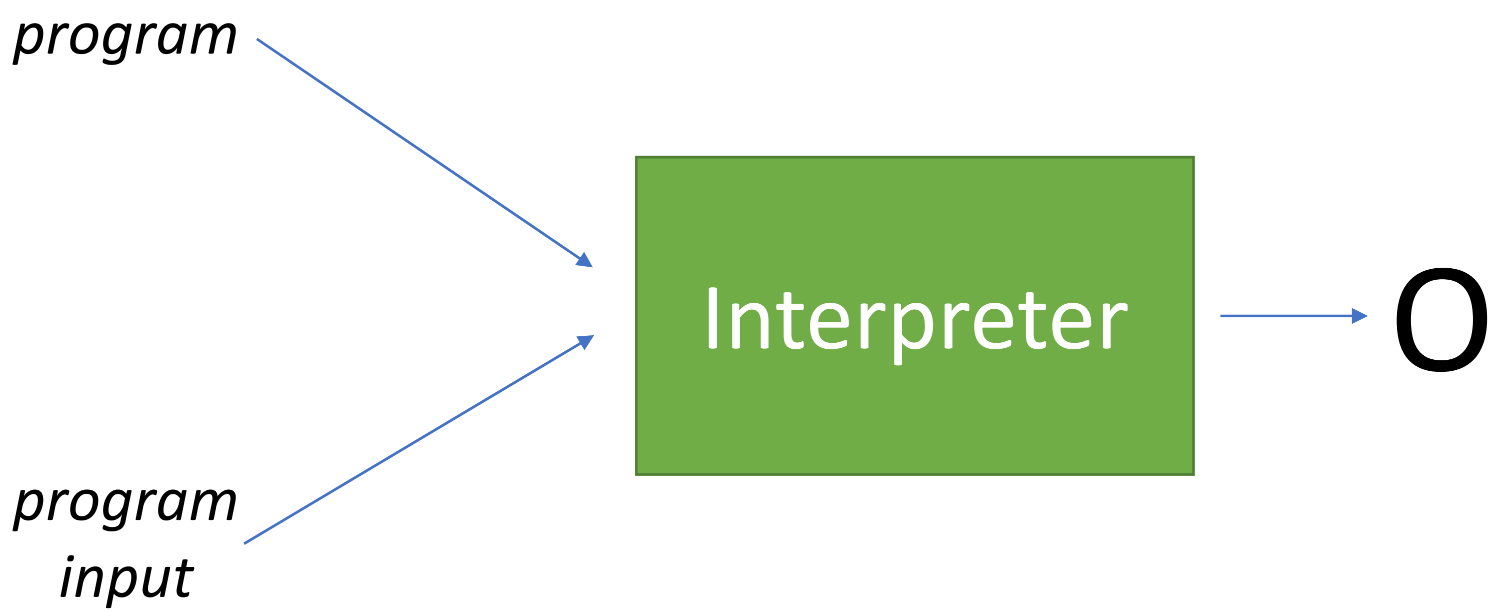


Futamura Projections

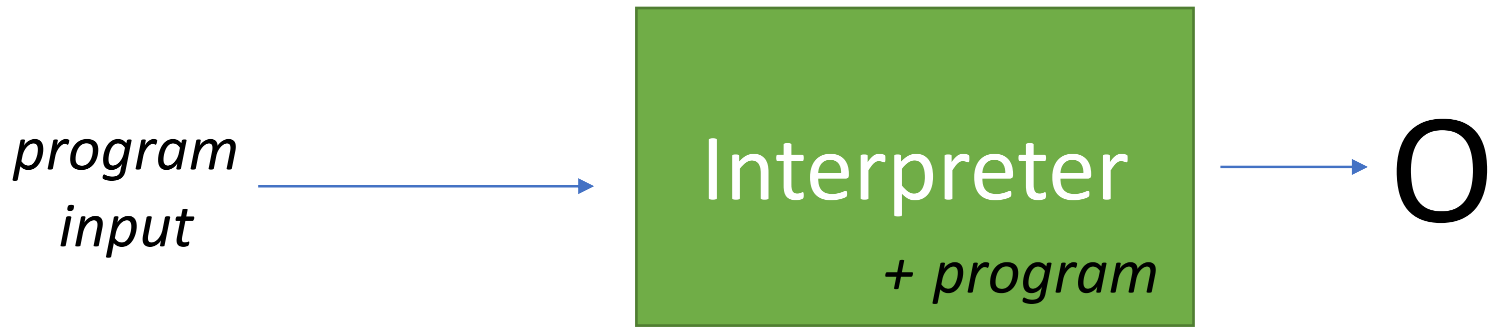
Futamura Projections



Futamura Projections



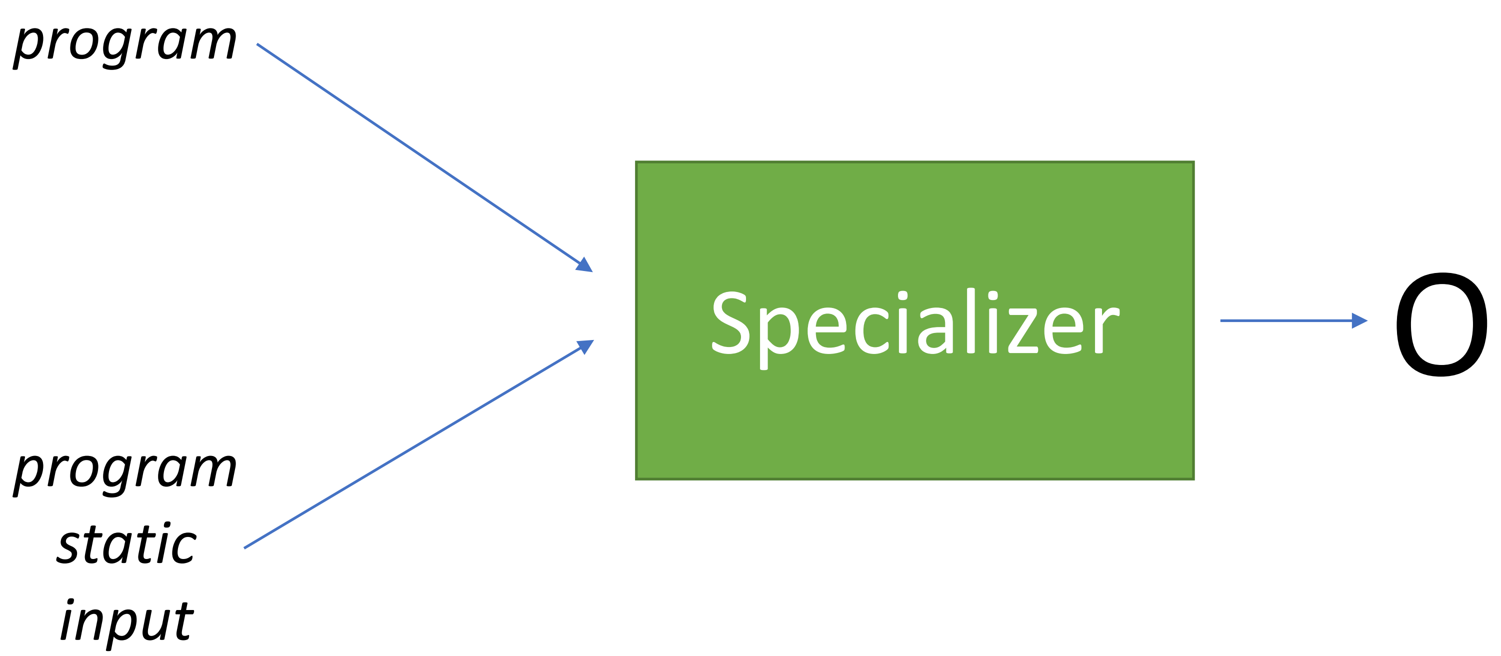
Futamura Projections



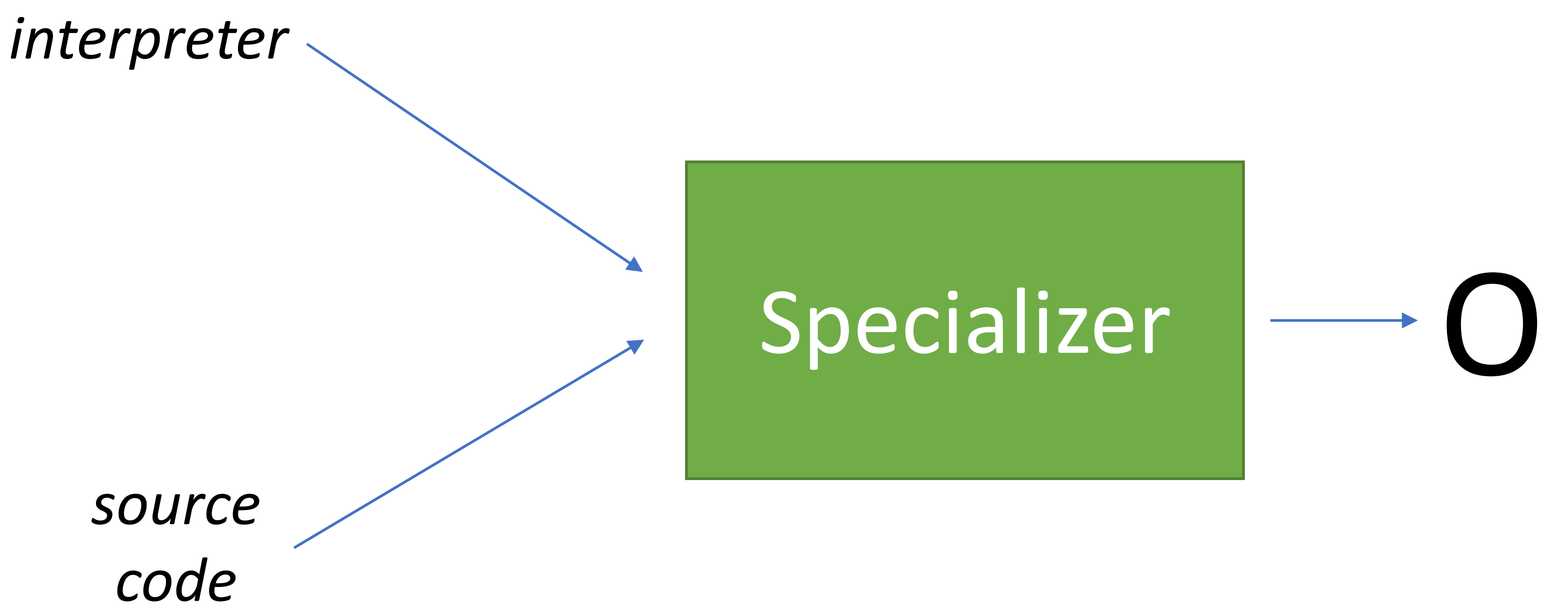
Futamura Projections

Futamura Projections (2)

Futamura Projections (2)



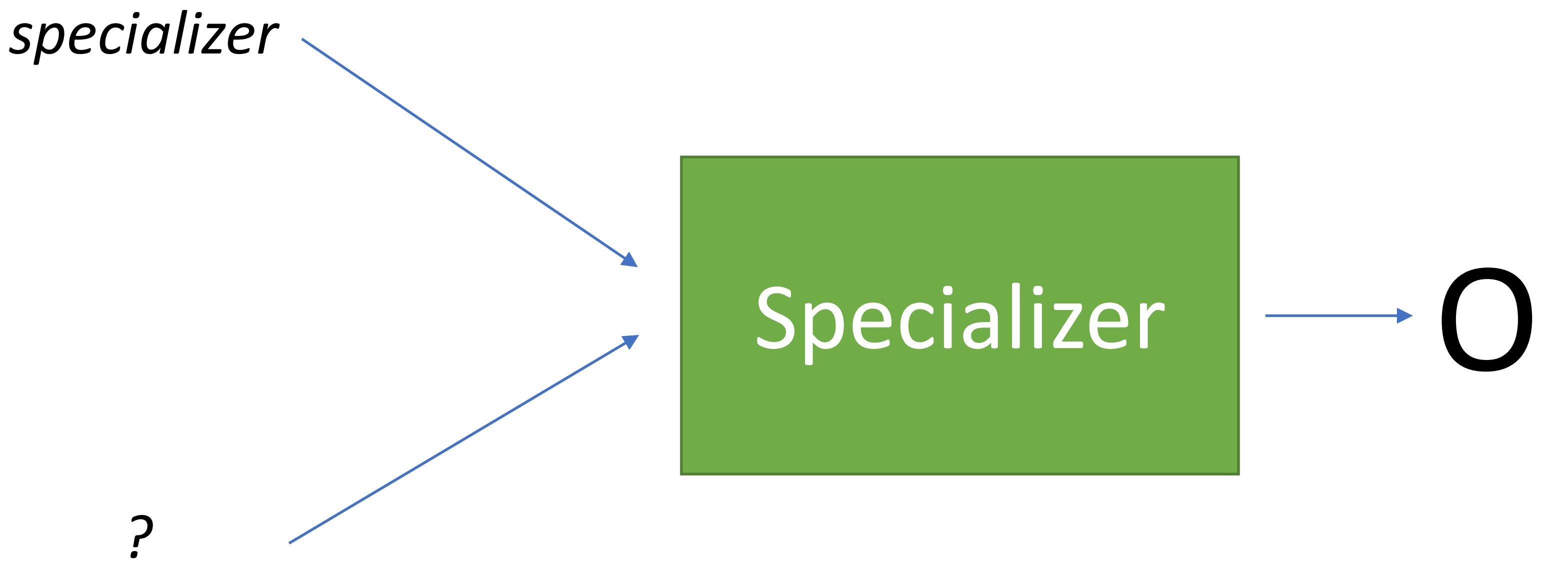
Futamura Projections (2)



Futamura Projections (2)

Futamura Projections (2)

Futamura Projections (3)



Futamura Projections

I) specializing interpreter to source code yields executable

II) specializing specializer for an interpreter (as used in I) yields a compiler

III) specializing specialier for itself (as used in II) yields compiler generator that given an interpreter produces a compiler

Bootstrapping a compiler

- a compiler is bootstrapped (self-hosted) when it can compile itself
- increases confidence in the compiler
- allows for greater control of the compilation and optimization process
- requires the first version of the compiler to be written in a different language
- is a costly process