Formal Verification of Just-in-Time Compilation

Aurèle Barrière

A means to execute a program, interleaving execution and optimization.
Just-in-Time compilers (JITs) have been popularized for executing *dynamic* languages.

A means to execute a program, interleaving execution and optimization.
Just-in-Time compilers (JITs) have been popularized for executing *dynamic* languages.

Modern web browsers use JITs to execute the JavaScript programs they find on the web.

A means to execute a program, interleaving execution and optimization.
Just-in-Time compilers (JITs) have been popularized for executing *dynamic* languages.

Modern web browsers use JITs to execute the JavaScript programs they find on the web.

Other JIT use cases: eBPF in the Linux Kernel, Python, Java . . .

A means to execute a program, interleaving execution and optimization.
Just-in-Time compilers (JITs) have been popularized for executing *dynamic* languages.

Modern web browsers use JITs to execute the JavaScript programs they find on the web.

Other JIT use cases: eBPF in the Linux Kernel, Python, Java …

Billions of people use JITs every day.

Bugs lead to vulnerabilities

May 2019: Coinbase company workers receive an email with a link.
Opening the link in Firefox installs malware on company computers.
Zero-day vulnerability (exploited before the bug was found and fixed).

Bugs lead to vulnerabilities

May 2019: Coinbase company workers receive an email with a link.
Opening the link in Firefox installs malware on company computers.
Zero-day vulnerability (exploited before the bug was found and fixed).

Google's Project Zero team reports 25 Zero-day exploits in web browsers in 2021, and 11 in 2022.

- **August 2019:** exploits in WebKit, the engine used in Safari.
- **January 2021**: exploits in V8, used in Google Chrome.
- **November 2022**: exploits in the Java standard library.

Bugs lead to vulnerabilities

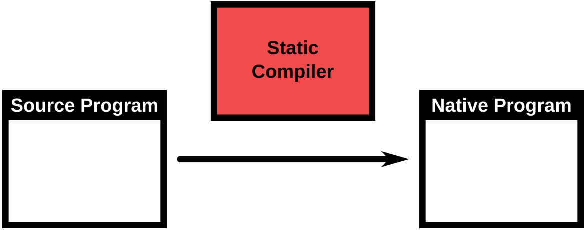
May 2019: Coinbase company workers receive an email with a link.
Opening the link in Firefox installs malware on company computers.
Zero-day vulnerability (exploited before the bug was found and fixed).

Google's Project Zero team reports 25 Zero-day exploits in web browsers in 2021, and 11 in 2022.

- **August 2019:** exploits in WebKit, the engine used in Safari.
- **January 2021**: exploits in V8, used in Google Chrome.
- **November 2022**: exploits in the Java standard library.

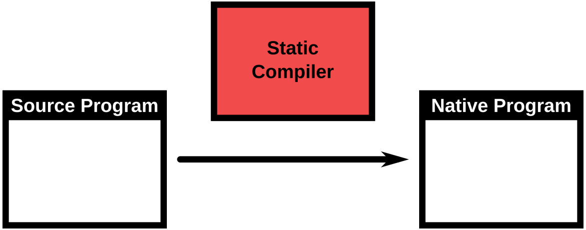
Bugs in JITs

These vulnerabilities rely on **bugs** in **Just-in-Time compilers** (JITs), and allow an attacker to execute malicious code.



Static compilation

Transforming a source program into an equivalent native program.
The resulting program can then be executed independently.
To trust this execution: trust both the source program and the compiler.



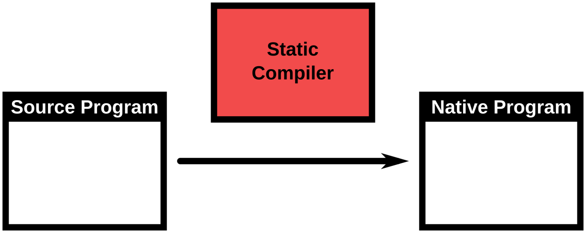
Static compilation

Transforming a source program into an equivalent native program.
The resulting program can then be executed independently.
To trust this execution: trust both the source program and the compiler.

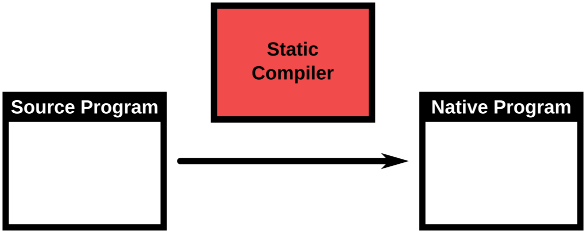
Despite testing, static compilers still contain bugs

PLDI 2011: Hundreds of bugs found in GCC and LLVM [Yang et al. 2011].

Static compilation theory is well established.



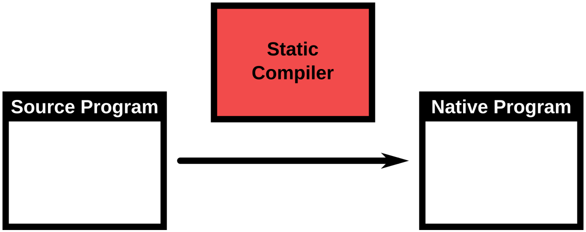
Static compilation theory is well established.



Program Semantics

We can formally define the behavior of a program, then prove the correctness of code transformations.

Static compilation theory is well established.



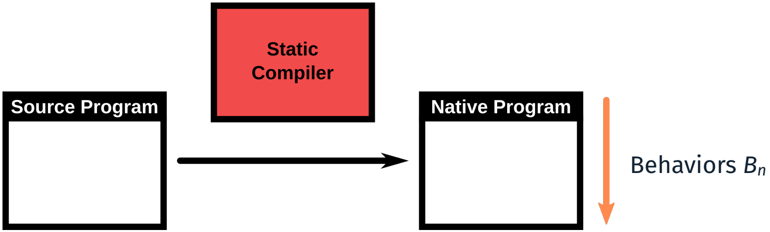
Program Semantics

We can formally define the behavior of a program, then prove the correctness of code transformations.

Formally Verified Static Compilers

CompCert [Leroy 2006], CakeML [Kumar et al. 2014], VeLLVM [Zhao et al. 2012].
We can **prove** that a compiler **preserves** the behavior of the program it compiles.

Static compilation theory is well established.



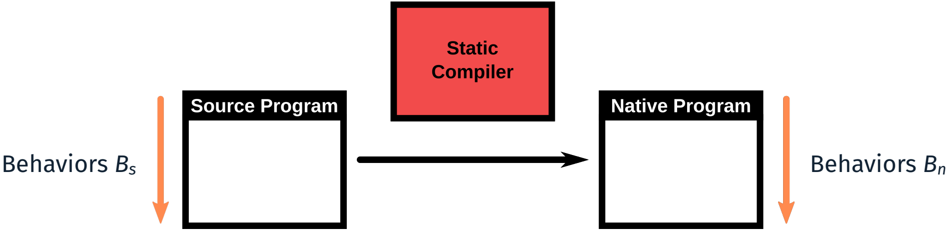
Program Semantics

We can formally define the behavior of a program, then prove the correctness of code transformations.

Formally Verified Static Compilers

CompCert [Leroy 2006], CakeML [Kumar et al. 2014], VeLLVM [Zhao et al. 2012].
We can **prove** that a compiler **preserves** the behavior of the program it compiles.

Static compilation theory is well established.



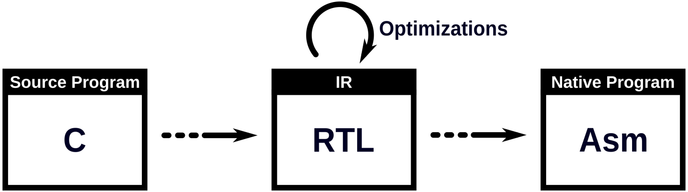
Program Semantics

We can formally define the behavior of a program, then prove the correctness of code transformations.

Formally Verified Static Compilers

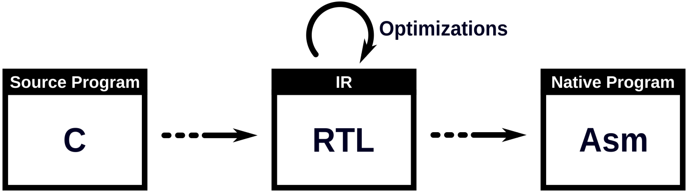
CompCert [Leroy 2006], CakeML [Kumar et al. 2014], VeLLVM [Zhao et al. 2012].
We can **prove** that a compiler **preserves** the behavior of the program it compiles. $B_n \subseteq B_s$

From C to assembly (x86, ARM, PowerPC, RiscV). 9 Intermediate Representations (IRs).



Software-proof codesign: developed in Coq, verified in Coq.
Executable: can be extracted as an OCaml program.
Modular proof: each transformation proved independently.

From C to assembly (x86, ARM, PowerPC, RiscV). 9 Intermediate Representations (IRs).



Software-proof codesign: developed in Coq, verified in Coq.
Executable: can be extracted as an OCaml program.
Modular proof: each transformation proved independently.

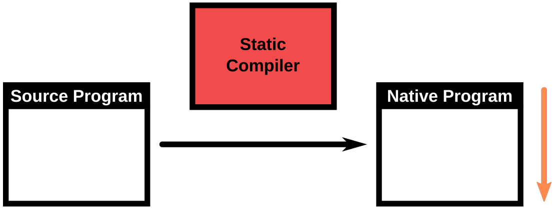
Formal Verification brings substantial guarantees

PLDI 2011: No bugs found in the formally verified part of CompCert [Yang et al. 2011].

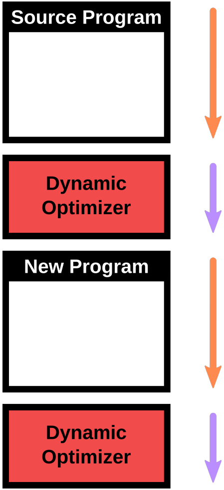
What about Just-in-Time Compilation ?

In essence, using a JIT means **interleaving** execution and optimization.

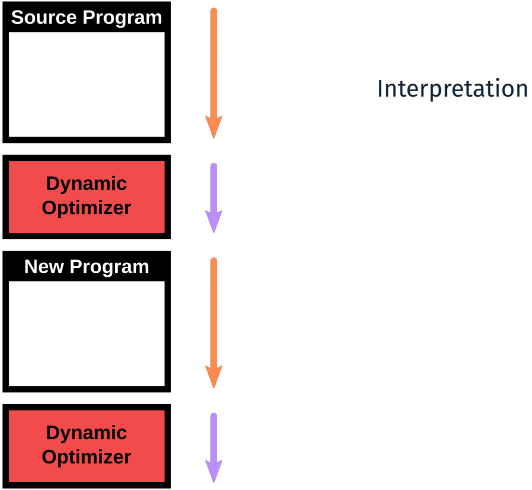
In essence, using a JIT means **interleaving** execution and optimization.



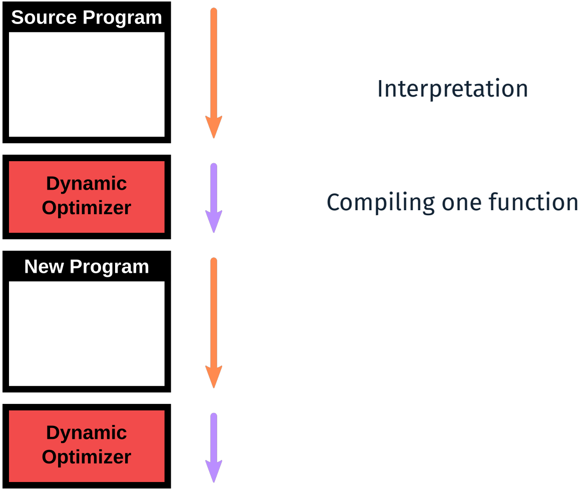
In essence, using a JIT means **interleaving** execution and optimization.



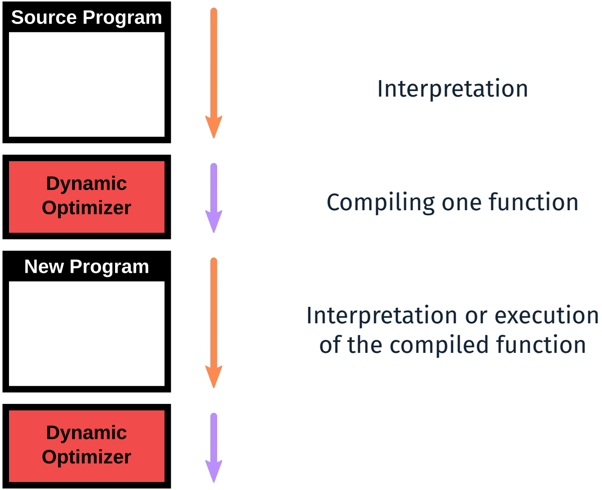
In essence, using a JIT means **interleaving** execution and optimization.



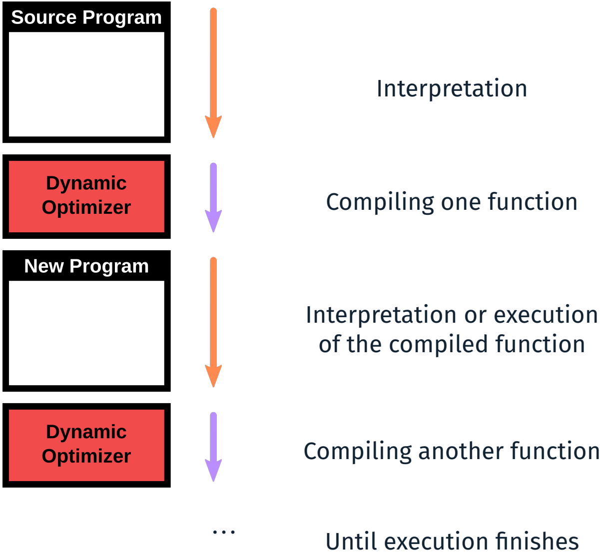
In essence, using a JIT means **interleaving** execution and optimization.



In essence, using a JIT means **interleaving** execution and optimization.

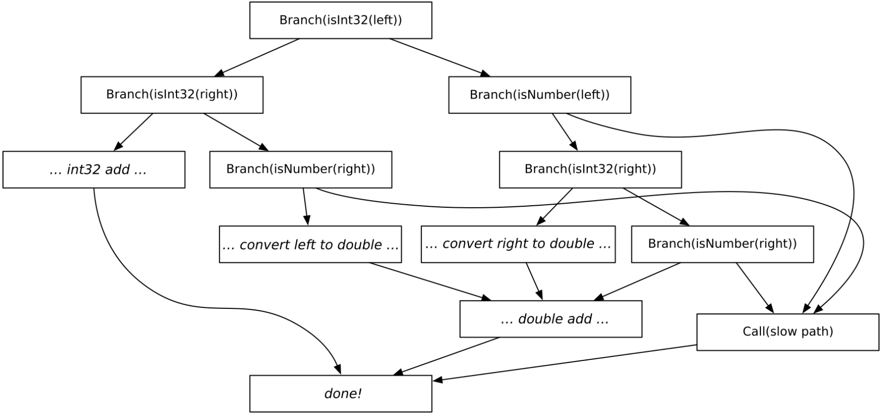


In essence, using a JIT means **interleaving** execution and optimization.



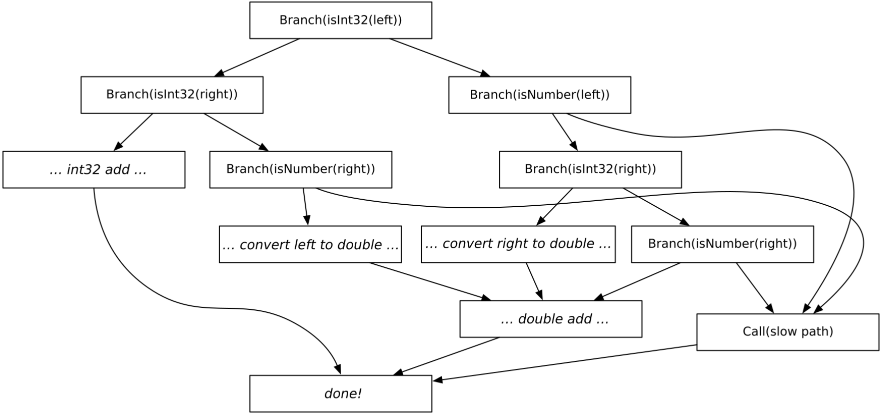
Executing operation `left + right`, an example from JavaScriptCore [WebKit 2020].

Executing operation `left + right`, an example from JavaScriptCore [WebKit 2020].



Difficult to compile. Restricts optimizations.

Executing operation `left + right`, an example from JavaScriptCore [WebKit 2020].



Difficult to compile. Restricts optimizations.

Type Speculation

If we can speculate on the types of `left` and `right`, the graph is reduced to a single node.

Execution Timeline

Interpretation of `f`

Execution Stack

Interpreter: `f`

Program

```
Function f():
 while(...):
  g()

Function g():
 g1
 g2
```

Execution Timeline

Interpretation of `f`

Interpretation of `g`

Execution Stack

Interpreter: `f`
Interpreter: `g`

Program

```
Function f():
 while(...):
  g()

Function g():
 g1
 g2
```

Execution Timeline

Interpretation of `f`

Interpretation of `g`

Optimization of `g`

Execution Stack

Interpreter: `f`

Optimizing Compiler

Program

```
Function f():
 while (...):
  g()

Function g():
 g1
 g2

Function g_x86():
 g1
 Speculation (x is int32)
 g2'
```

Execution Timeline

Interpretation of `f`

Interpretation of `g`

Optimization of `g`

Call to `g_x86`

Execution Stack

Interpreter: `f`

Native: `g_x86`

Program

```
Function f():
 while (...):
  g()

Function g():
 g1
 g2

Function g_x86():
 g1
 Speculation (x is int32)
 g2'
```

Execution Timeline

Interpretation of f

Interpretation of g

Optimization of g

Call to g_x86

Deoptimization to g

Execution Stack

Interpreter: f
Native: g_x86

Speculation fails

Program

```
Function f():
 while (...):
  g()

Function g():
 g1
 g2

Function g_x86():
 g1
 Speculation (x is int32)
 g2'
```

Executing a program with a JIT with speculative optimizations

Execution Timeline

Interpretation of `f`

Interpretation of `g`

Optimization of `g`

Call to `g_x86`

Deoptimization to `g`

Execution Stack

Interpreter: `f`
~~Native: `g_x86`~~
Interpreter: `g`

Speculation fails

On-stack replacement

Program

```
Function f():
 while (...):
  g()

Function g():
 g1
 g2

Function g_x86():
 g1
 Speculation (x is int32)
 g2'
```

Execution Timeline

Interpretation of `f`

Interpretation of `g`

Optimization of `g`

Call to `g_x86`

Deoptimization to `g`

Execution Stack

Execution Stack	
Interpreter: `f`	
~~Native: `g_x86`~~	Speculation fails
Interpreter: `g`	On-stack replacement

Program

```
Function f():
 while (...):
  g()

Function g():
 g1
 g2

Function g_x86():
 g1
 Speculation (x is int32)
 g2'
```

Deoptimization requires the JIT to synthesize interpreter stackframes in the middle of a function.

JIT-specific techniques like speculation, on-stack replacement are scarcely formalized.

JIT-specific techniques like speculation, on-stack replacement are scarcely formalized.

Related Work in Formalizing JITs

- [Myreen 2010]: Verified non-optimizing x86 JIT.
- [Guo et al. 2011]: Soundness of trace optimizations in JITs.
- [Brown et al. 2020]: DSL to write verified range analyses in JITs.
- [Flückiger et al. 2018]: **Sourir**, an IR with speculation, with correctness proofs of speculative optimizations.

In this thesis, we investigate the feasability of developing formally verified JITs.

Dynamic Optimizations: JITs interleave optimizations with the program execution.

Speculative Optimizations: JITs insert and manipulate speculations in their programs.

Impure Components: Some JITs components are difficult to write in Coq (*e.g.* executing native code).

Static Compiler Reuse: Verified JITs should reuse formally verified compilers.

Dynamic Optimizations: JITs interleave optimizations with the program execution.
Our approach: adapt the simulation proof methodology of CompCert.

Speculative Optimizations: JITs insert and manipulate speculations in their programs.

Impure Components: Some JITs components are difficult to write in Coq (*e.g.* executing native code).

Static Compiler Reuse: Verified JITs should reuse formally verified compilers.

Dynamic Optimizations: JITs interleave optimizations with the program execution.
Our approach: adapt the simulation proof methodology of CompCert.

Speculative Optimizations: JITs insert and manipulate speculations in their programs.
Our approach: define semantics for speculative instructions, verify code transformations manipulating them.

Impure Components: Some JITs components are difficult to write in Coq (*e.g.* executing native code).

Static Compiler Reuse: Verified JITs should reuse formally verified compilers.

Dynamic Optimizations: JITs interleave optimizations with the program execution.
Our approach: adapt the simulation proof methodology of CompCert.

Speculative Optimizations: JITs insert and manipulate speculations in their programs.
Our approach: define semantics for speculative instructions, verify code transformations manipulating them.

Impure Components: Some JITs components are difficult to write in Coq (*e.g.* executing native code).
Our approach: a free monadic encoding of the JIT with a pure specification of impure primitives.

Static Compiler Reuse: Verified JITs should reuse formally verified compilers.

Dynamic Optimizations: JITs interleave optimizations with the program execution.
Our approach: adapt the simulation proof methodology of CompCert.

Speculative Optimizations: JITs insert and manipulate speculations in their programs.
Our approach: define semantics for speculative instructions, verify code transformations manipulating them.

Impure Components: Some JITs components are difficult to write in Coq (*e.g.* executing native code).
Our approach: a free monadic encoding of the JIT with a pure specification of impure primitives.

Static Compiler Reuse: Verified JITs should reuse formally verified compilers.
Our approach: integrate the CompCert backend and its proof to generate native code. Semantics for interleaving native code execution with other JIT components.

Dynamic Optimizations: JITs interleave optimizations with the program execution.
Our approach: adapt the simulation proof methodology of CompCert. B)

Speculative Optimizations: JITs insert and manipulate speculations in their programs.
Our approach: define semantics for speculative instructions, verify code transformations manipulating them. C)

Impure Components: Some JITs components are difficult to write in Coq (*e.g.* executing native code).
Our approach: a free monadic encoding of the JIT with a pure specification of impure primitives.

Static Compiler Reuse: Verified JITs should reuse formally verified compilers.
Our approach: integrate the CompCert backend and its proof to generate native code. Semantics for interleaving native code execution with other JIT components.

In this presentation:

- A) Our JIT Design
- B) Dynamic Optimizations
- C) Speculative Optimizations

Designing a Formally Verified JIT in Coq

CompCert Theorem

If we compile a program whose behaviors are free of errors, then any behavior of the compiled program is a behavior of the source program.

```
Theorem transf_c_program_is_refinement:
  ∀ p tp, transf_c_program p = OK tp → (* compilation of p produced tp *)
  (∀ beh,program_behaves (source_sem p) beh → not_wrong beh) → (* p has no wrong behaviors*)
  (∀ beh,program_behaves (asm_sem tp) beh → program_behaves (source_sem p) beh).
  (* every behavior of tp is a behavior of p *)
```

CompCert Theorem

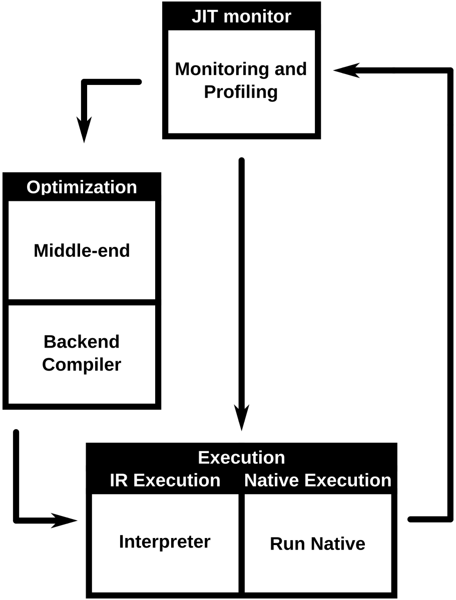
If we compile a program whose behaviors are free of errors, then any behavior of the compiled program is a behavior of the source program.

```
Theorem transf_c_program_is_refinement:
  ∀ p tp, transf_c_program p = OK tp → (* compilation of p produced tp *)
  (∀ beh,program_behaves (source_sem p) beh → not_wrong beh) → (* p has no wrong behaviors*)
  (∀ beh,program_behaves (asm_sem tp) beh → program_behaves (source_sem p) beh).
  (* every behavior of tp is a behavior of p *)
```

JIT Theorem

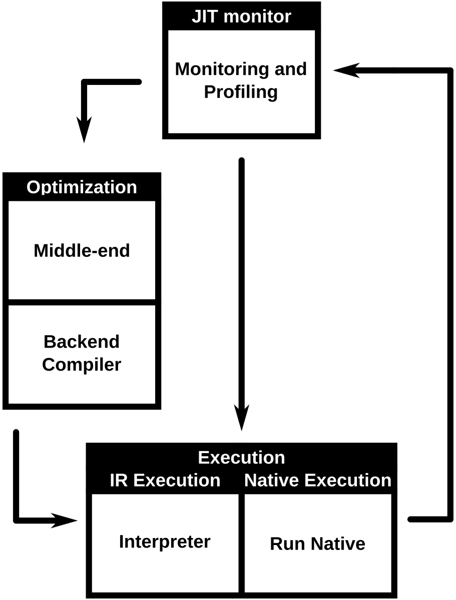
If the semantics (`source_sem`) of the program is free of errors, then any behavior of the JIT on that program (`jit_sem`) is a behavior of the program.

```
Theorem jit_behavior_refinement:
  ∀ p,
  (∀ beh,program_behaves (source_sem p) beh → not_wrong beh) → (* p has no wrong behaviors*)
  (∀ beh,program_behaves (jit_sem p) beh → program_behaves (source_sem p) beh).
  (* every behavior of the JIT executing p is a behavior of p *)
```



JIT loop

The **monitor** chooses execution or optimization. **Profiling**: inspects execution and suggests speculations and optimizations.

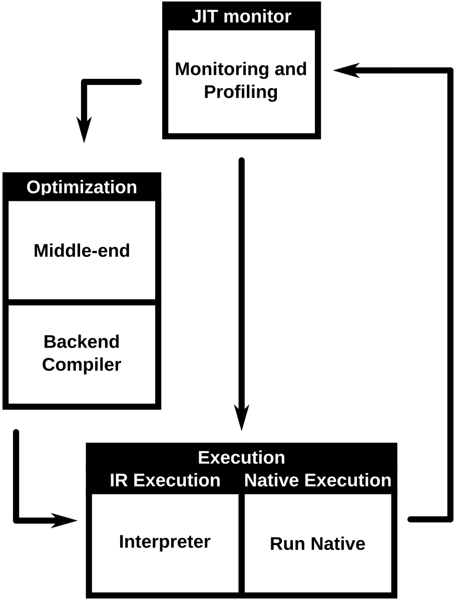


Middle-end Optimizer

From the IR to the IR.
Inserts speculation.

Backend Compiler

Generates native code.
Use the CompCert backend from RTL to x86.



Interpreter
Interpret the IR code.

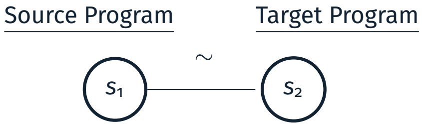
Native Code Execution
Run the generated code.

Returns to the monitor on function calls, function returns or deoptimizations.

Correctness of a JIT with dynamic optimizations

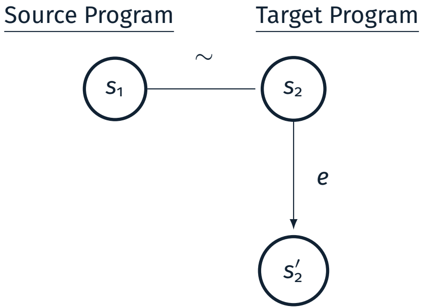
Backward Simulation in CompCert: (simplified)

- design an invariant $\sim$, a relation between semantic states.
- show that each semantic step of the target program is matched with steps of the source program preserving the invariant and the observable events e.



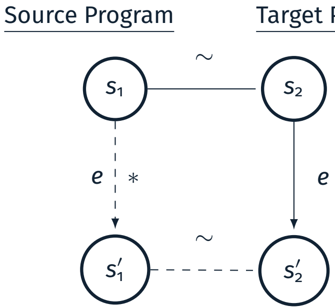
Backward Simulation in CompCert: (simplified)

- design an invariant $\sim$, a relation between semantic states.
- show that each semantic step of the target program is matched with steps of the source program preserving the invariant and the observable events e.



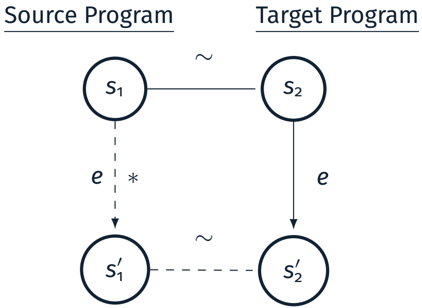
Backward Simulation in CompCert: (simplified)

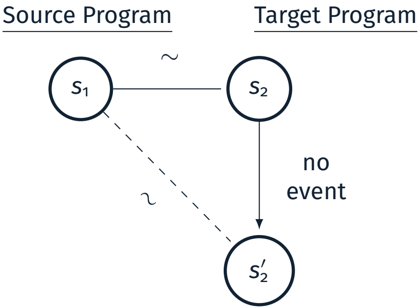
- design an invariant $\sim$, a relation between semantic states.
- show that each semantic step of the target program is matched with steps of the source program preserving the invariant and the observable events e.



Backward Simulation in CompCert: (simplified)

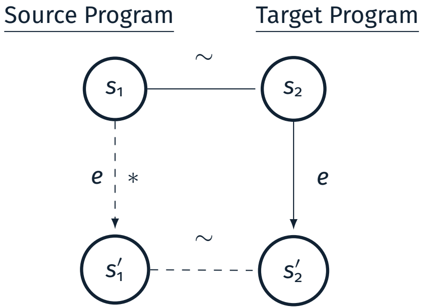
- design an invariant $\sim$, a relation between semantic states.
- show that each semantic step of the target program is matched with steps of the source program preserving the invariant and the observable events *e*.

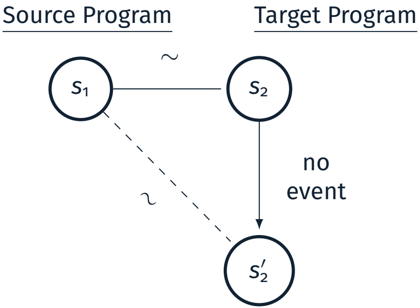




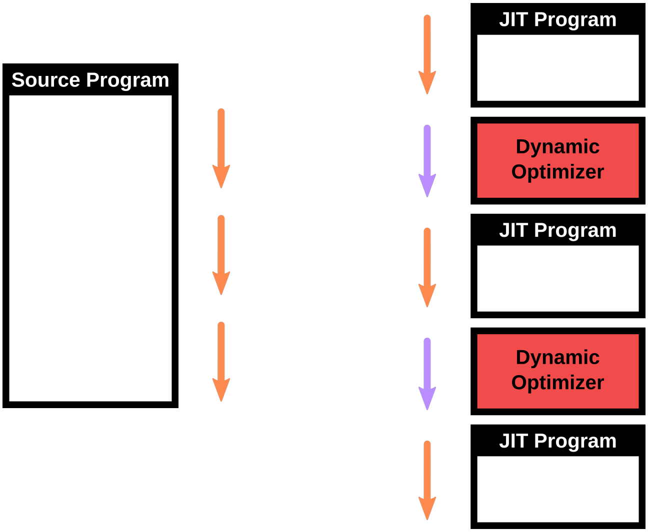
Backward Simulation in CompCert: (simplified)

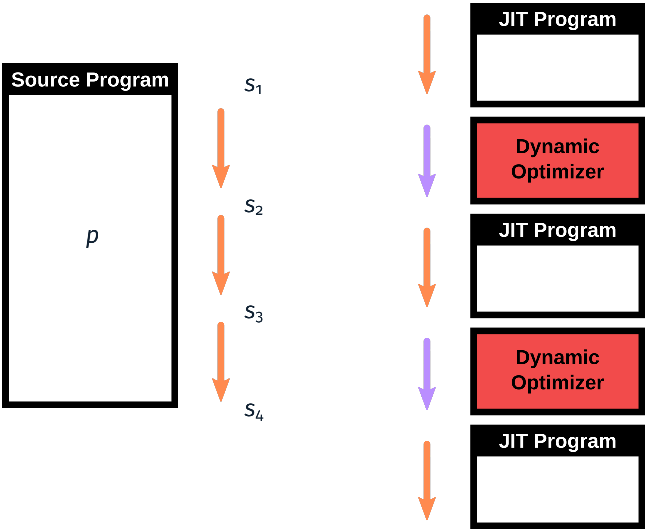
- design an invariant $\sim$, a relation between semantic states.
- show that each semantic step of the target program is matched with steps of the source program preserving the invariant and the observable events e.



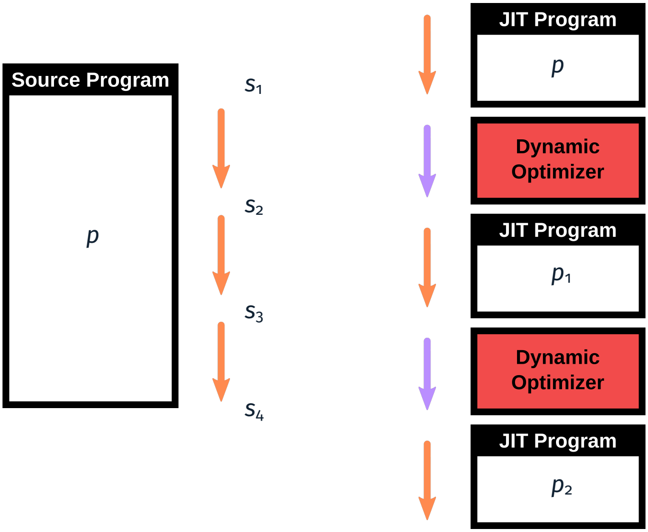


Simulations are **composable**.


Source Program
JIT Program
Dynamic Optimizer
JIT Program
Dynamic Optimizer
JIT Program

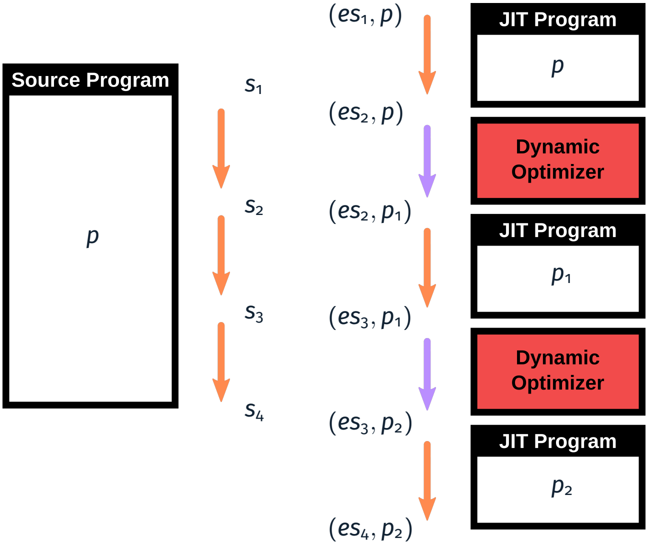


Source Execution:
Fixed program p.
Semantic states of p: s_1, s_2, s_3, s_4.



Source Execution:
Fixed program p.
Semantic states of p: s_1, s_2, s_3, s_4.

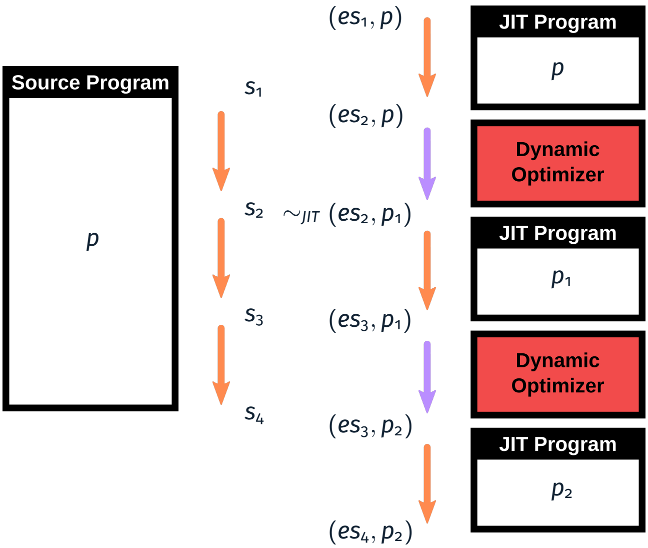
JIT Execution:
Both the program and the execution state (*e.g.* interpreter state) are evolving.



Source Execution:
Fixed program p.
Semantic states of p: s_1, s_2, s_3, s_4.

JIT Execution:
Both the program and the execution state (*e.g.* interpreter state) are evolving.

JIT Semantic states: (es, jp)
where es is an execution state,
jp is the current JIT program.



Source Execution:
Fixed program p.
Semantic states of p: s_1, s_2, s_3, s_4.

JIT Execution:
Both the program and the execution state (*e.g.* interpreter state) are evolving.

JIT Semantic states: (es, jp)
where es is an execution state,
jp is the current JIT program.

To Prove a Simulation:
Find and preserve an invariant relation $\sim_{JIT}$ between source states s and JIT states (es, jp).

Invariant Intuition:
At any point in the execution,

- the current execution state corresponds to a source semantic state,
- the current JIT program is equivalent to the source program p.

Invariant Intuition:
At any point in the execution,

- the current execution state corresponds to a source semantic state,
- the current JIT program is equivalent to the source program *p*.

This equivalence can be expressed with a backward simulation.

Invariant Intuition:
At any point in the execution,

- the current execution state corresponds to a source semantic state,
- the current JIT program is equivalent to the source program p.

This equivalence can be expressed with a backward simulation.

Defining the invariant

Nested Simulations: Our JIT simulation invariant contains another simulation (called *internal*).
$s \sim_{JIT} (es, jp)$ when

- there *exists* a simulation between p and jp, using an invariant $\sim_{in}$
- $s \sim_{in} es$.

The internal simulation changes as the JIT performs optimizations.

Invariant Intuition:
At any point in the execution,

- the current execution state corresponds to a source semantic state,
- the current JIT program is equivalent to the source program p.

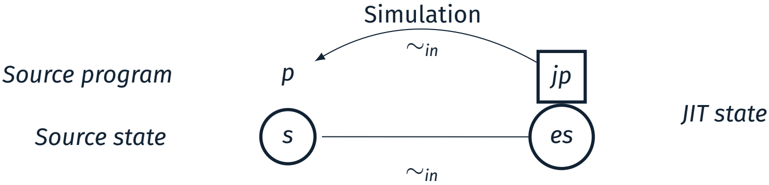
This equivalence can be expressed with a backward simulation.

Defining the invariant

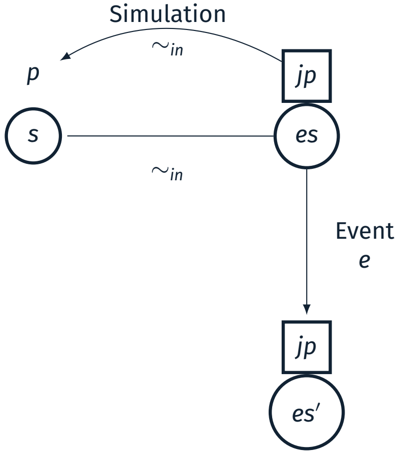
Nested Simulations: Our JIT simulation invariant contains another simulation (called *internal*).
$s \sim_{JIT} (es, jp)$ when

- there *exists* a simulation between p and jp, using an invariant $\sim_{in}$
- $s \sim_{in} es$.

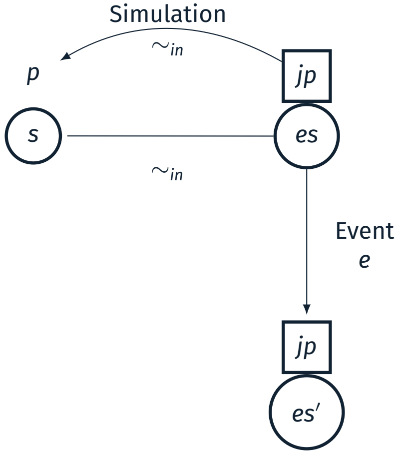
The internal simulation changes as the JIT performs optimizations.



The JIT did an **execution** step (*e.g.* interpreter), and updated its execution state *es*.

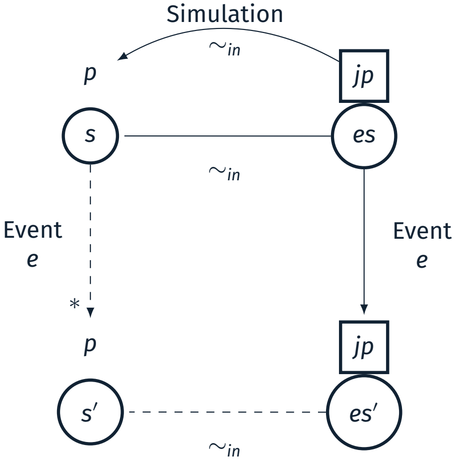


The JIT did an **execution** step (*e.g.* interpreter), and updated its execution state *es*.



We need to prove that there exists matching steps in the source semantics preserving the invariant and the observed behavior *e*.

The JIT did an **execution** step (*e.g.* interpreter), and updated its execution state *es*.

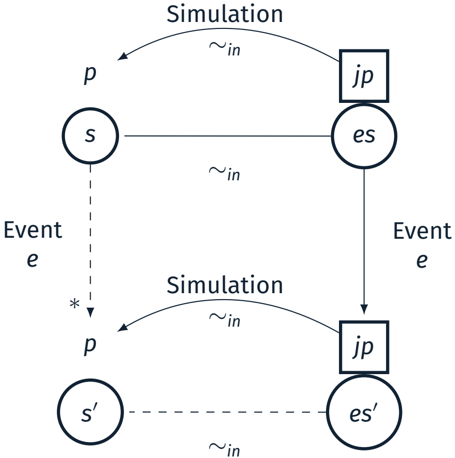


We use the internal simulation to exhibit source steps.

We need to prove that there exists matching steps in the source semantics preserving the invariant and the observed behavior *e*.

The JIT did an **execution** step (*e.g.* interpreter), and updated its execution state *es*.

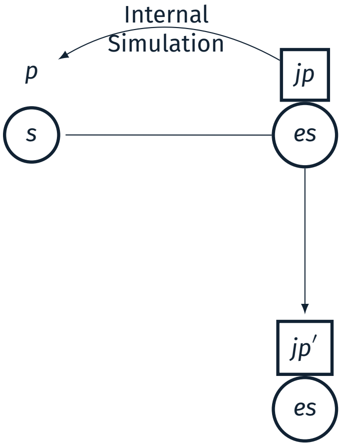
We use the internal simulation to exhibit source steps.



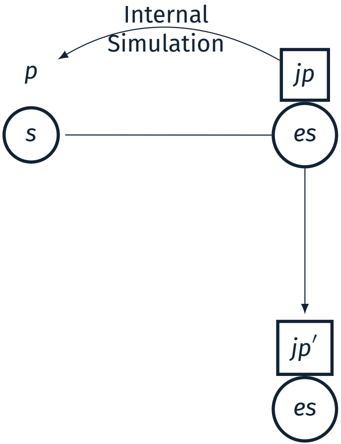
We need to prove that there exists matching steps in the source semantics preserving the invariant and the observed behavior *e*.

The JIT program *jp* did not change, and is still simulated with *p*.

The JIT did an **optimization** step and updated its program *jp*.

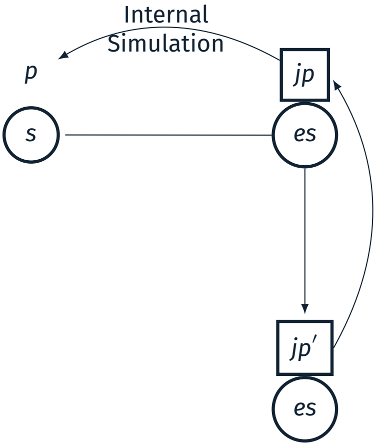


The JIT did an **optimization** step and updated its program *jp*.



We need to prove that jp' is still simulated with *p*.

The JIT did an **optimization** step and updated its program *jp*.

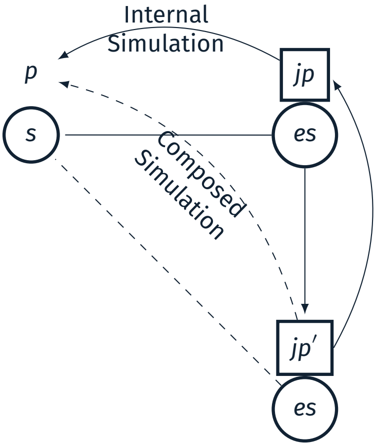


We need to prove that jp' is still simulated with p.

We prove the dynamic optimizer correct with a backward simulation.

```
Theorem optimizer_correct:
∀ jp jp’,
optimizer jp = jp’ →
backward_simulation jp jp’.
```

The JIT did an **optimization** step and updated its program *jp*.



We need to prove that *jp*′ is still simulated with *p*.

We prove the dynamic optimizer correct with a backward simulation.

```
Theorem optimizer_correct:
∀ jp jp',
optimizer jp = jp' →
backward_simulation jp jp'.
```

We compose this new simulation with the existing internal one.

To prove correct a JIT with dynamic optimizations, it suffices to prove its optimizer correct with a backward simulation.

To prove correct a JIT with dynamic optimizations, it suffices to prove its optimizer correct with a backward simulation.

Like a static compiler.

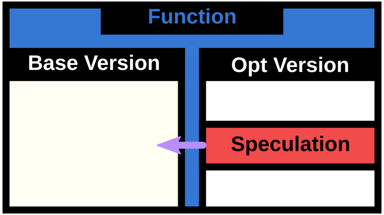
Formally Verified Speculation

Speculative instructions should have precisely defined semantics.

Speculative instructions should have precisely defined semantics.

CoreIR, an Intermediate Representation for Speculation

Two speculative instructions to represent two sides of speculation.
Inspired by Sourir [Flückiger et al. 2018] and CompCert's RTL.
Each function has a *base* version and an optional *optimized* one.



```
Function F(x,y):
Version Base:
l1: a ← 1


    b ← y + x


    Return (a + b)
```

```
Function F(x,y):
Version Base:
l1: a ← 1

    b ← y + x

    Return (a + b)
```

Profiler: we might want to speculate here later

```
Function F(x,y):
Version Base:
l1: a ← 1

    b ← y + x

    Return (a + b)
```

```
Version Opt:
l1: a ← 1
    Anchor F.l1 [x ← x, y ← y, a ← 1]

    b ← y + x

    Return (a + b)
```

Profiler: we might want to speculate here later

Timeline:

- Create an `Opt` version, with synchronization points `Anchor`

```
Function F(x,y):
Version Base:
l1: a ← 1

    b ← y + x

    Return (a + b)
```

```
Version Opt:
l1: a ← 1
    Anchor F.l1 [x ← x, y ← y, a ← 1]

    b ← y + x

    Return (a + b)
```

Timeline:

- Create an `Opt` version, with synchronization points `Anchor`

Profiler: speculate that x is 9

```
Function F(x,y):
Version Base:
l1: a ← 1



    b ← y + x



    Return (a + b)
```

```
Version Opt:
l1: a ← 1
    Anchor F.l1 [x ← x, y ← y, a ← 1]
    Assume (x=9) F.l1 [x ← x, y ← y, a ← 1]

    b ← y + x


    Return (a + b)
```

Timeline:

- Create an `Opt` version, with synchronization points `Anchor`
- Insert speculation `Assume`

Profiler: speculate that `x` is `9`

```
Function F(x,y):
Version Base:
l1: a ← 1

    b ← y + x

    Return (a + b)
```

```
Version Opt:
l1: a ← 1
    Anchor F.l1 [x ← x, y ← y, a ← 1]
    Assume (x=9) F.l1 [x ← x, y ← y, a ← 1]

    b ← y + 9

    Return (1 + b)
```

Timeline:

- Create an `Opt` version, with synchronization points `Anchor`
- Insert speculation `Assume`
- Constant propagation

```
Function F(x,y):
Version Base:
l1: a ← 1


    b ← y + x


    Return (a + b)
```

```
Version Opt:
l1: a ← 1
    Anchor F.l1 [x ← x, y ← y, a ← 1]
    Assume (x=9) F.l1 [x ← x, y ← y, a ← 1]
    Assume (y=7) F.l1 [x ← x, y ← y, a ← 1]
    b ← y + 9


    Return (1 + b)
```

Timeline:

- Create an `Opt` version, with synchronization points `Anchor`
- Insert speculation `Assume`
- Constant propagation
- Insert speculation

Profiler: speculate that `y` is `7`

```
Function F(x,y):
Version Base:
l1: a ← 1



    b ← y + x



    Return (a + b)
```

```
Version Opt:
l1: a ← 1
    Anchor F.l1 [x ← x, y ← y, a ← 1]
    Assume (x=9) F.l1 [x ← x, y ← y, a ← 1]
    Assume (y=7) F.l1 [x ← x, y ← y, a ← 1]
    b ← 16


    Return (17)
```

Timeline:

- Create an Opt version, with synchronization points Anchor
- Insert speculation Assume
- Constant propagation
- Insert speculation
- Constant propagation

```
Function F(x,y):
Version Base:
l1: a ← 1


    b ← y + x


    Return (a + b)
```

```
Version Opt:

    Anchor F.l1 [x ← x, y ← y, a ← 1]
    Assume (x=9) F.l1 [x ← x, y ← y, a ← 1]
    Assume (y=7) F.l1 [x ← x, y ← y, a ← 1]


    Return (17)
```

Timeline:

- Create an `Opt` version, with synchronization points `Anchor`
- Insert speculation `Assume`
- Constant propagation
- Insert speculation
- Constant propagation
- Dead code elimination

The Anchor instruction : How to Deoptimize

Expresses a synchronization between the original and optimized versions.

The Anchor instruction : How to Deoptimize

Expresses a synchronization between the original and optimized versions.

`Anchor F.l` [x $\leftarrow$ y+1] means that you **can** deoptimize to function `F`, label `l`, putting y+1 in register x.

The Anchor instruction : How to Deoptimize

Expresses a synchronization between the original and optimized versions.

`Anchor F.l` [x ← y+1]   means that you **can** deoptimize to function `F`, label `l`, putting y+1 in register x.

The Assume instruction : Making Speculation Explicit

Insert speculation at any time next to an `Anchor` using an `Assume`.

The Anchor instruction : How to Deoptimize

Expresses a synchronization between the original and optimized versions.

`Anchor F.l` [x ← y+1]   means that you **can** deoptimize to function `F`, label `l`, putting y+1 in register x.

The Assume instruction : Making Speculation Explicit

Insert speculation at any time next to an `Anchor` using an `Assume`.

`Assume` (y=3) `F.l` [x ← y+1]   deoptimizes if y is not 3, skips to the next instruction otherwise.

The Anchor instruction : How to Deoptimize

Expresses a synchronization between the original and optimized versions.

`Anchor F.l` [x ← y+1] means that you **can** deoptimize to function `F`, label `l`, putting y+1 in register x.

The Assume instruction : Making Speculation Explicit

Insert speculation at any time next to an `Anchor` using an `Assume`.

`Assume` (y=3) `F.l` [x ← y+1] deoptimizes if y is not 3, skips to the next instruction otherwise.

Non-deterministic Semantics

The semantics of an `Anchor` are non-deterministic: either deoptimize to the original version or continue to the next instruction in the optimized version.
`Assume` is deterministic: deoptimizes when the speculation fails.

To prove each transformation, prove a backward simulation.

```
Theorem assume_insertion_correct:
  ∀ p f guard lbl newp,
    insert_assume f guard lbl p = OK newp →
    backward_simulation p newp.
```

To prove each transformation, prove a backward simulation.

```
Theorem assume_insertion_correct:
  ∀ p f guard lbl newp,
    insert_assume f guard lbl p = OK newp →
    backward_simulation p newp.
```

First, design an **invariant** relating semantic states of CoreIR.

Semantic states of CoreIR:

$(S,\ C,\ pc,\ R,\ M)$

stack

code

label

registers

memory

```
Function G():
   Return F(4)

Function F(x):
Version Base:
l1:Return (x+16)

Version Opt (Cbefore):
la:Anchor F.l1 [x ←x]

l1:Return (x+16)
```

Assume Insertion

```
Function G():
   Return F(4)

Function F(x):
Version Base:
l1:Return (x+16)

Version Opt (Cbefore):
la:Anchor F.l1 [x ←x]

l1:Return (x+16)
```

```
Function G():
   Return F(4)

Function F(x):
Version Base:
l1:Return (x+16)

Version Opt (Cafter):
la:Anchor F.l1 [x ←x]
l  :Assume (x=4) F.l1 [x ←x]
l1:Return (x+16)
```

Assume Insertion

```
Function G():
   Return F(4)

Function F(x):
Version Base:
l1:Return (x+16)

Version Opt (Cbefore):
la:Anchor F.l1 [x ←x]

l1:Return (x+16)
```

```
Function G():
   Return F(4)

Function F(x):
Version Base:
l1:Return (x+16)

Version Opt (Cafter):
la:Anchor F.l1 [x ←x]
l  :Assume (x=4) F.l1 [x ←x]
l1:Return (x+16)
```

Simulation Invariant: $\sim$

Assume Insertion

```
Function G():
   Return F(4)

Function F(x):
Version Base:
l1:Return (x+16)

Version Opt (Cbefore):
la:Anchor F.l1 [x ←x]

l1:Return (x+16)
```

$\sim$

$\sim$

```
Function G():
   Return F(4)

Function F(x):
Version Base:
l1:Return (x+16)

Version Opt (Cafter):
la:Anchor F.l1 [x ←x]
l  :Assume (x=4) F.l1 [x ←x]
l1:Return (x+16)
```

Simulation Invariant: $\sim$

Outside the modified version:

$$\frac{C \neq C_{before} \qquad S_1 \approx S_2}{(S_1, C, pc, R, M) \sim (S_2, C, pc, R, M)}$$

$S_1 \approx S_2$ when stackframes of C_{before} have been replaced with stackframes of C_{after}.

Assume Insertion

```
Function G():                              Function G():
   Return F(4)          ~                     Return F(4)

Function F(x):                             Function F(x):
Version Base:                              Version Base:
l1:Return (x+16)        ~                  l1:Return (x+16)

Version Opt (Cbefore):                     Version Opt (Cafter):
la:Anchor F.l1 [x←x]    ~                  la:Anchor F.l1 [x←x]
                                           l :Assume (x=4) F.l1 [x←x]
l1:Return (x+16)        ~                  l1:Return (x+16)
```

Simulation Invariant: $\sim$

Outside the modified version:

$$\frac{C \neq C_{before} \qquad S_1 \approx S_2}{(S_1, C, pc, R, M) \sim (S_2, C, pc, R, M)}$$

Inside the modified version:

$$\frac{pc \neq l \qquad S_1 \approx S_2}{(S_1, C_{before}, pc, R, M) \sim (S_2, C_{after}, pc, R, M)}$$

$S_1 \approx S_2$ when stackframes of C_{before} have been replaced with stackframes of C_{after}.

Assume Insertion

`Function G():`		`Function G():`
`    Return F(4)`	$\sim$	`    Return F(4)`
`Function F(x):`		`Function F(x):`
`Version Base:`		`Version Base:`
`l1:Return (x+16)`	$\sim$	`l1:Return (x+16)`
`Version Opt (Cbefore):`		`Version Opt (Cafter):`
`la:Anchor F.l1 [x ←x]`	$\sim$	`la:Anchor F.l1 [x ←x]`
	$\sim$	`l :Assume (x=4) F.l1 [x ←x]`
`l1:Return (x+16)`	$\sim$	`l1:Return (x+16)`

Simulation Invariant: $\sim$

Outside the modified version:

$$\frac{C \neq C_{before} \qquad S_1 \approx S_2}{(S_1, C, pc, R, M) \sim (S_2, C, pc, R, M)}$$

Inside the modified version:

$$\frac{pc \neq l \qquad S_1 \approx S_2}{(S_1, C_{before}, pc, R, M) \sim (S_2, C_{after}, pc, R, M)}$$

At the new `Assume`:

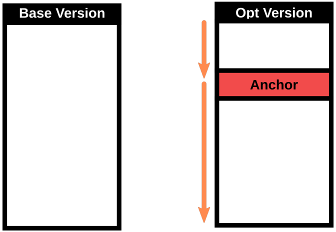
$$\frac{S_1 \approx S_2}{(S_1, C_{before}, l_a, R, M) \sim (S_2, C_{after}, l, R, M)}$$

$S_1 \approx S_2$ when stackframes of C_{before} have been replaced with stackframes of C_{after}.

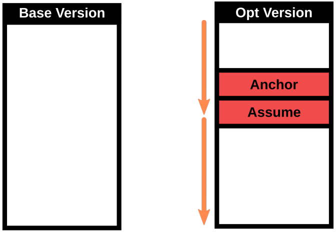
At the `Assume`, we reason by case analysis on the speculation:

- if it holds, we match this step to no deoptimization of the `Anchor`

Before `Assume` Insertion:

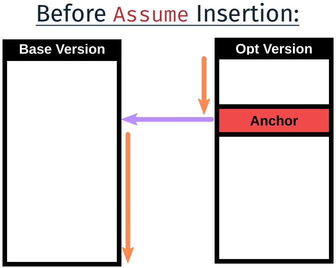


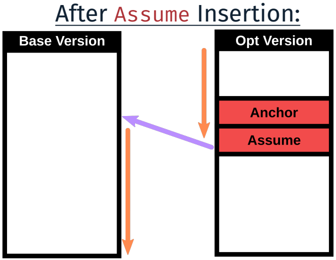
After `Assume` Insertion:



At the `Assume`, we reason by case analysis on the speculation:

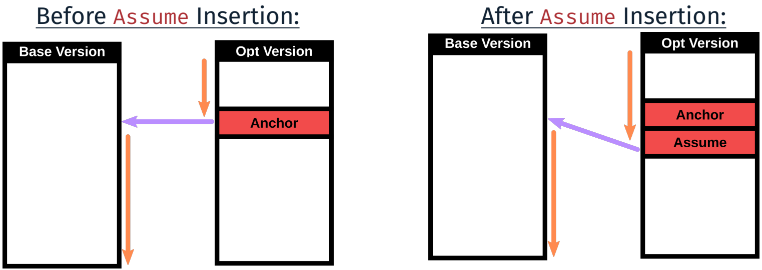
- if it holds, we match this step to no deoptimization of the `Anchor`
- if it fails, we match this step to deoptimization of the `Anchor`.





At the `Assume`, we reason by case analysis on the speculation:

- if it holds, we match this step to no deoptimization of the `Anchor`
- if it fails, we match this step to deoptimization of the `Anchor`.



Using the non-determinism of the `Anchor`, it is correct to insert a new `Assume`.

Non-deterministic semantics are adequate to represent the possibility of a deoptimization.

Backward Simulations: Handling Speculation

- **Inserting Anchor** instructions.
- **Inserting Assume** instructions using Anchors.
- **Lowering** : Removes Anchor instructions.

A Backward Simulation for Inlining

- **Inlining** : When deoptimizing from an inlined function, we need to synthesize an additional stackframe to restore the environment.

Proving Non-speculative Optimizations

- **Constant Propagation** : reusing the *forward to backward* methodology of CompCert.

Middle-end Correctness

Correct for any speculation and optimizations done by the middle-end.

```
Theorem middle_end_correct:
  ∀ jp jp' optim_list,
  middle_end jp optim_list = jp' →
  backward_simulation jp jp'.
```

Middle-end Correctness

Correct for any speculation and optimizations done by the middle-end.

```
Theorem middle_end_correct:
  ∀ jp jp' optim_list,
  middle_end jp optim_list = jp' →
  backward_simulation jp jp'.
```

Backend Correctness

- Transform a CoreIR function into RTL programs, using custom calling conventions to interact with the JIT.
- Then, use the CompCert backend to generate x86 code.

```
Theorem backend_correct:
  ∀ jp' jp'',
  backend jp' = jp'' → (* integrates CompCert backend *)
  backward_simulation jp' jp''.
```

Middle-end Correctness

Correct for any speculation and optimizations done by the middle-end.

```
Theorem middle_end_correct:
  ∀ jp jp' optim_list,
  middle_end jp optim_list = jp' →
  backward_simulation jp jp'.
```

Backend Correctness

- Transform a CoreIR function into RTL programs, using custom calling conventions to interact with the JIT.
- Then, use the CompCert backend to generate x86 code.

```
Theorem backend_correct:
  ∀ jp' jp'',
  backend jp' = jp'' → (* integrates CompCert backend *)
  backward_simulation jp' jp''.
```

Using **nested** simulations:

```
Theorem jit_simulation:
  ∀ p, backward_simulation p (jit_sem p).
```

Which implies the JIT behavior preservation theorem.

A small set of *impure* primitives used in a JIT:

- store and load from the heap.
- push and pop from the stack.
- install, load and execute native code.

For primitives, write both C implementations and Coq specifications.

An encoding for programs using both Coq functions and Impure primitives:

```
Definition optimizer (f:function) : free unit :=
  do f2 ← ret (middle_end f);  (* written in Coq *)
  do f_x86 ← ret (backend f2); (* written in Coq, reusing CompCert *)
  Prim_Install_Code f_x86.  (* impure primitive *)
```

Solving the first two JIT challenges

POPL21: *Formally Verified Speculation and Deoptimization in a JIT compiler.* [Barrière et al. 2021]

- Dynamic Optimizations
- Speculations.

Solving the remaining two JIT challenges

POPL23: *Formally Verified Native Code Generation in an Effectful JIT*
or: Turning the CompCert Backend into a Formally Verified JIT Compiler. [Barrière et al. 2023]

- Impure Components
- Native code generation (reusing CompCert) and execution.

Solving the first two JIT challenges

POPL21: *Formally Verified Speculation and Deoptimization in a JIT compiler.* [Barrière et al. 2021]

- Dynamic Optimizations
- Speculations.

Solving the remaining two JIT challenges

POPL23: *Formally Verified Native Code Generation in an Effectful JIT*
or: Turning the CompCert Backend into a Formally Verified JIT Compiler. [Barrière et al. 2023]

- Impure Components
- Native code generation (reusing CompCert) and execution.

Executable Coq develoment: execute our JIT on CoreIR programs.
Speedups when speculating or generating native code.
`https://github.com/Aurele-Barriere/JIThm`

Designed a generic JIT architecture.

Identified JIT-specific verification challenges and presented composable correctness arguments.

A methodology to develop Formally Verified JITs, with a mechanized verified and executable Coq prototype.

Addressing our limitations

- **Impure primitive verification**: check the C implementations against their Coq specifications.
- **Generating efficient code**: the interfacing between native code and the JIT can be improved.

Addressing our limitations

- **Impure primitive verification**: check the C implementations against their Coq specifications.
- **Generating efficient code**: the interfacing between native code and the JIT can be improved.

A new playground to formalize/verify JIT features

- **Recompilation**: Recompiling from scratch a function that deoptimizes too often.
- **Contextual Dispatch**: Different versions of a function specialized for different call contexts. [Flückiger et al. 2020].
- **On-Stack-Replacement as en entry**: Jumping into native code from the interpreter.

Addressing our limitations

- **Impure primitive verification**: check the C implementations against their Coq specifications.
- **Generating efficient code**: the interfacing between native code and the JIT can be improved.

A new playground to formalize/verify JIT features

- **Recompilation**: Recompiling from scratch a function that deoptimizes too often.
- **Contextual Dispatch**: Different versions of a function specialized for different call contexts. [Flückiger et al. 2020].
- **On-Stack-Replacement as en entry**: Jumping into native code from the interpreter.

Towards Verified JITs for Realistic Languages

WebAssembly, a language for high-performance applications on web pages.
On-going work on formally verified Coq interpreter and compiler for WebAssembly [Watt et al. 2021].
We could then reuse our techniques to develop a formally verified WebAssembly JIT.

Recap - JIT-specific Verification Challenges

Dynamic Optimizations: JITs interleave optimizations with the program execution.
Our approach: adapt the simulation proof methodology of CompCert.

Speculative Optimizations: JITs insert and manipulate speculations in their programs.
Our approach: define semantics for speculative instructions, verify code transformations manipulating them.

Impure Components: Some JITs components are difficult to write in Coq (*e.g.* executing native code).
Our approach: free monadic encoding of the JIT with a pure specification of some impure primitives.

Static Compiler Reuse: Verified JITs should reuse formally verified compilers.
Our approach: integrate the CompCert backend and its proof to generate native code.

- Barrière, Aurèle, Sandrine Blazy, and David Pichardie (2023). "Formally Verified Native Code Generation in an Effectful JIT or: Turning the CompCert Backend into a Formally Verified JIT Compiler". In: *Proc. ACM Program. Lang.* POPL.
- Barrière, Aurèle et al. (2021). "Formally Verified Speculation and Deoptimization in a JIT Compiler". In: *Proc. ACM Program. Lang.* POPL.
- Brown, Fraser et al. (2020). "Towards a Verified Range Analysis for JavaScript JITs". In: *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020*. ACM, pp. 135–150.
- Flückiger, Olivier et al. (2018). "Correctness of Speculative Optimizations with Dynamic Deoptimization". In: POPL.
- Flückiger, Olivier et al. (2020). "Contextual Dispatch for Function Specialization". In: *Proc. ACM Program. Lang.* 4.OOPSLA, 220:1–220:24.
- Guo, Shu-yu and Jens Palsberg (2011). "The Essence of Compiling with Traces". In: *Proceedings of the Symposium on Principles of Programming Languages, POPL.*
- Kumar, Ramana et al. (2014). "CakeML: a Verified Implementation of ML". In: *Proceedings of POPL.*
- Leroy, Xavier (2006). "Formal Certification of a Compiler Back-end or: Programming a Compiler with a Proof Assistant". In: *Proceedings of POPL.*

- Myreen, Magnus O. (2010). “Verified Just-in-Time Compiler on x86”. In: *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010*. Ed. by Manuel V. Hermenegildo and Jens Palsberg. ACM, pp. 107–118.
- Watt, Conrad et al. (2021). “Two Mechanisations of WebAssembly 1.0”. In: *Formal Methods - 24th International Symposium, FM 2021*. Ed. by Marieke Huisman, Corina S. Pasareanu, and Naijun Zhan. Vol. 13047. Lecture Notes in Computer Science. Springer, pp. 61–79.
- WebKit (2020). *Speculation in JavaScriptCore*. `https://webkit.org/blog/10308/speculation-in-javascriptcore/`.
- Yang, Xuejun et al. (2011). “Finding and Understanding Bugs in C Compilers”. In: *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011*. Ed. by Mary W. Hall and David A. Padua. ACM, pp. 283–294.
- Zhao, Jianzhou et al. (2012). “Formalizing the LLVM Intermediate Representation for Verified Program Transformations”. In: *Proceedings of the Symposium on Principles of Programming Languages, POPL*.

CoreIR Syntax

Operands:

$$
\begin{array}{lll}
op & ::= r & \text{Register} \\
 & \mid v & \text{Value}
\end{array}
$$

Expressions:

$$
\begin{array}{lll}
e & ::= op + op \mid op - op \mid op * op & \text{Arithmetic} \\
 & \mid op < op \mid op = op & \text{Relational} \\
 & \mid op & \text{Register or value}
\end{array}
$$

Instructions:

$$
\begin{array}{lll}
i & ::= \mathbf{Nop}\ l & \text{Noop} \\
 & \mid (r \leftarrow e)^*\ l & \text{Operations} \\
 & \mid \mathbf{Cond}\ e\ l_t\ l_f & \text{Branch} \\
 & \mid r \leftarrow \mathbf{Call}\ f\ e^*\ l & \text{Call} \\
 & \mid \mathbf{Return}\ e & \text{Return} \\
 & \mid r \leftarrow \mathbf{Load}\ e\ l & \text{Memory load} \\
 & \mid e \leftarrow \mathbf{Store}\ e\ l & \text{Memory store} \\
 & \mid \mathbf{Print}\ e\ l & \text{Output value} \\
 & \mid \mathbf{Anchor}\ deop\ l & \text{Deoptimization anchor} \\
 & \mid \mathbf{Assume}\ e^*\ deop\ l & \text{Speculation}
\end{array}
$$

Metadata:

$$
\begin{array}{lll}
vm & ::= (r \leftarrow e)^* & \text{Varmap} \\
syn & ::= f.l\ r\ vm & \text{Stack frame} \\
deop & ::= f.l\ vm\ syn^* & \text{Deopt metadata}
\end{array}
$$

Programs:

$$
\begin{array}{lll}
V & ::= l \mapsto i & \text{Code} \\
F & ::= \{r^*, l, V, option\ V\} & \text{Function} \\
P & ::= f \mapsto F & \text{Program}
\end{array}
$$

CoreIR Speculative Instructions Semantics

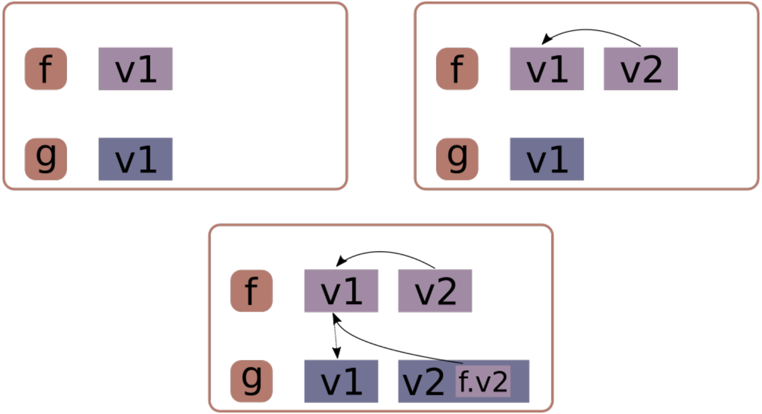
$$\textsc{Ignore}\ \frac{\begin{array}{c} V[l_{pc}] \;=\; \texttt{Anchor}\ f.l\ vm\ st^*\ l_{next} \\ \texttt{deopt\_regmap}\ vm\ R \;=\; R' \qquad \texttt{synthesize\_frame}\ R\ st^* \;=\; S' \end{array}}{S\ V\ l_{pc}\ R\ M \;\longrightarrow\; S\ V\ l_{next}\ R\ M}$$

$$\textsc{Deopt}\ \frac{\begin{array}{c} V[l_{pc}] \;=\; \texttt{Anchor}\ f.l\ vm\ st^*\ l_{next} \\ \texttt{deopt\_regmap}\ vm\ R \;=\; R' \qquad \texttt{synthesize\_frame}\ R\ st^* \;=\; S' \end{array}}{S\ V\ l_{pc}\ R\ M \;\longrightarrow\; (S' +\!\!+\, S)\ (\texttt{base\_version}_P\ f)\ l\ R'\ M}$$

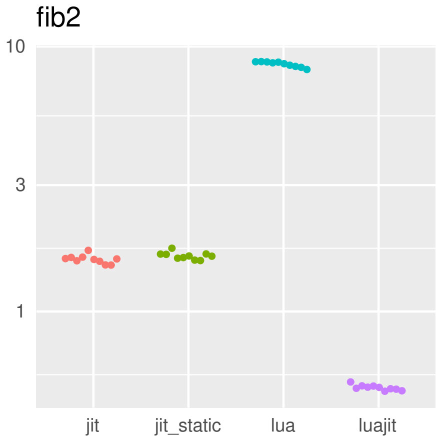
$$\textsc{AssumePass}\ \frac{V[l_{pc}] \;=\; \texttt{Assume}\ e^*\ f.l\ vm\ st^*\ l_{next} \qquad (e^*, R) \Downarrow\ true}{S\ V\ l_{pc}\ R\ M \;\longrightarrow\; S\ V\ l_{next}\ R\ M}$$

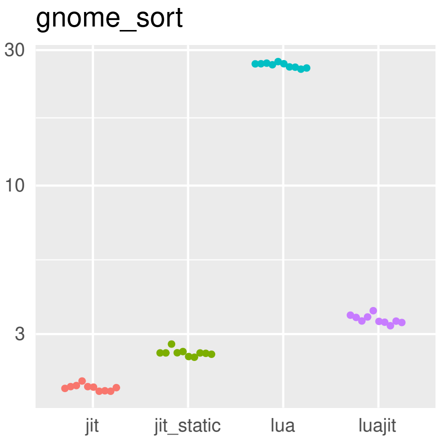
$$\textsc{AssumeFail}\ \frac{\begin{array}{c} V[l_{pc}] \;=\; \texttt{Assume}\ e^*\ f.l\ vm\ st^*\ l_{next} \qquad (e^*, R) \Downarrow\ false \\ \texttt{deopt\_regmap}\ vm\ R \;=\; R' \qquad \texttt{synthesize\_frame}\ R\ st^* \;=\; S' \end{array}}{S\ V\ l_{pc}\ R\ M \;\longrightarrow\; (S' +\!\!+\, S)\ (\texttt{base\_version}_P\ f)\ l\ R'\ M}$$

Inlining Speculation - Extra Stackframe on Deoptimization

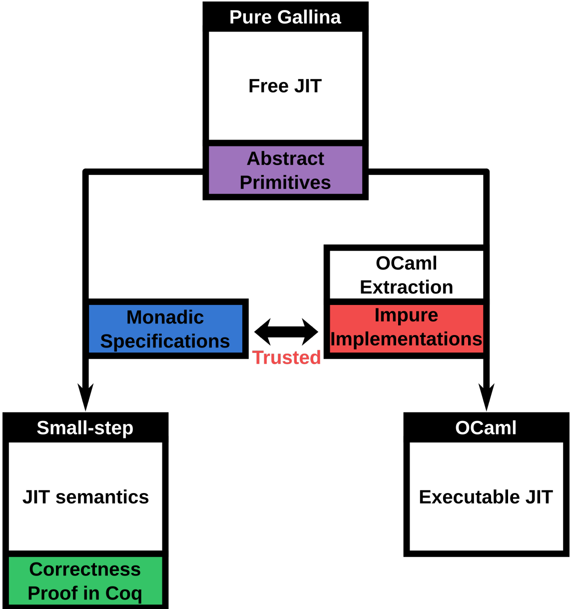


CoreIR Experiments





Free Monadic Encoding



State and Error Monad

A **pure** encoding of functions modifying and accessing a global state.
Either the function fails, or it succeeds and returns the next global state. Found in CompCert.

```
Inductive sres (state:Type) (A:Type): Type :=
| SError : errmsg → sres state A
| SOK: A → state → sres state A.
Definition state_mon {state:Type} (A:Type) : Type := state → sres state A.
```

Monadic constructors:

```
Definition state_ret {state:Type} {A:Type} (x:A) : state_mon A :=
fun (s:state) ⇒ SOK x s.

Definition state_bind {state:Type} {A B:Type} (f: state_mon A) (g:A → state_mon B) : state_mon B :=
  fun (s:state) ⇒
    match (f s) with
    | SError msg ⇒ SError msg
    | SOK a s' ⇒ g a s'
    end.
```

Executable Impure JIT

This is fine to specify the primitives, but the executable JIT should execute **impure** code.

Free Monad Definitions - An Example

In that example, we want to write programs that can access a single global variable of type nat.

A list of primitives our programs can use:

```
Inductive primitive: Type → Type :=
| Get : primitive nat
| Put (x:nat) : primitive unit.
```

We can then define the free monad:

```
Inductive free (T :Type) : Type :=
  | pure (x : T) :  free T
  | impure {R}
    (prim : primitive R) (next : R → free T).
```

Binding free monadic computations:

```
Fixpoint free_bind {X Y} (f: free X) (g: X →
free Y) : free Y :=
  match f with
  | pure x ⇒ g x
  | impure R prim next ⇒
    impure prim (fun x ⇒ free_bind (next x) g)
    end.
```

We can now encode incomplete programs.
Next, let’s see two ways to complete them.

Giving Semantics to our Free Monad - An Example

Given primitive implementations, we want to turn a free monadic function into an executable state monadic one. A **monadic specification** consists in one state monad for each primitive:

```
Record monad_spec: Type :=
  mk_mon_spec {
    prim_get: state_mon nat;
    prim_put: nat → state_mon unit; }.
```

```
Definition prim_spec {R:Type} (p:primitive R)
                  (i:monad_spec) : state_mon R :=
  match p with
  | Get ⇒ prim_get i
  | Put x ⇒ prim_put i x
  end.
```

We can now define **semantics** for our free monadic encoding, replacing each primitive with its specification:

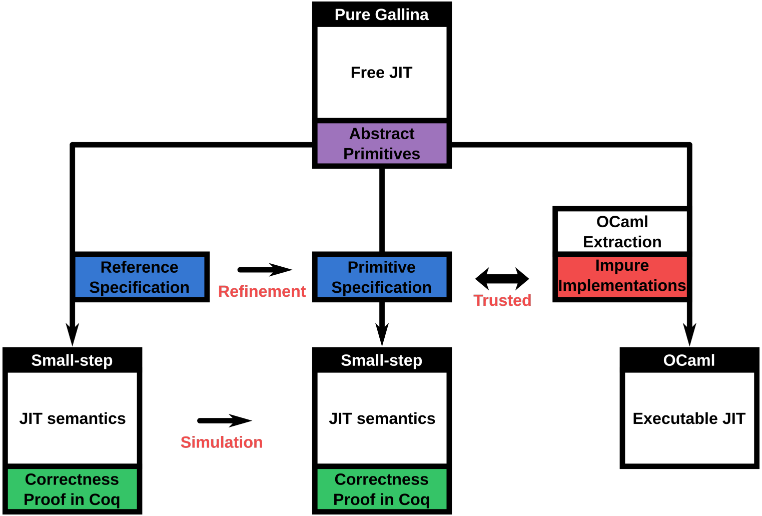
```
Fixpoint free_sem {A:Type} (f:free A) (i:monad_spec): state_mon A :=
  match f with
  | pure a ⇒ state_ret a
  | impure R prim cont ⇒
    state_bind (prim_spec prim i) (fun r:R ⇒ free_sem (cont r) i)
  end.
```

Executing the Free Monad - An Example

Finally, we extract the free monadic JIT to OCaml.
We can write a new way to **execute** a free monadic function, calling impure primitives when needed.

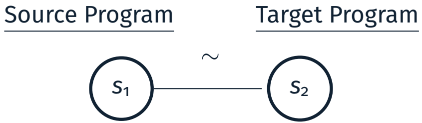
```
(* impure primitive implementations, using a global reference *)
let prim_impl (p:'x primitive) : 'x =
match p with
  | Get -> !global
  | Put (n) -> global := n
(* from OCaml, we can also call C implementations *)

(* executing free monads *)
let rec exec (f:'A free) : 'A =
  match f with
  | Coq_pure (a) -> a
  | Coq_ferror (e) -> print_error e; failwith "JIT crashed"
  | Coq_impure (prim, cont) ->
     let x = prim_impl prim in
     exec (cont x)
```

Free Monadic Refinement



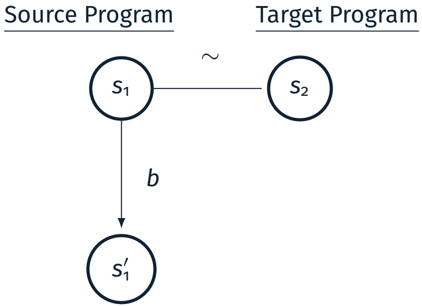
Forward Simulation in CompCert:

- Design an invariant $\sim$, a relation between semantic states.
- Show that each semantic step of the source program is matched with steps of the target program preserving the invariant.



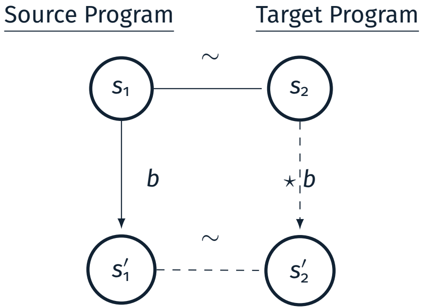
Forward Simulation in CompCert:

- Design an invariant $\sim$, a relation between semantic states.
- Show that each semantic step of the source program is matched with steps of the target program preserving the invariant.



Forward Simulation in CompCert:

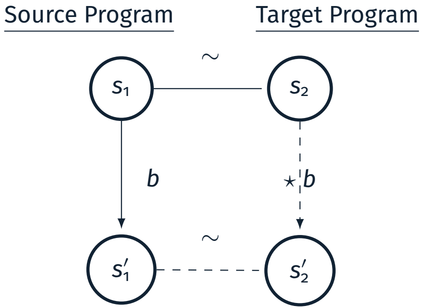
- Design an invariant $\sim$, a relation between semantic states.
- Show that each semantic step of the source program is matched with steps of the target program preserving the invariant.



Simulations and Refinement

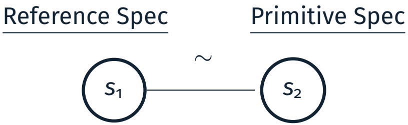
Forward Simulation in CompCert:

- Design an invariant $\sim$, a relation between semantic states.
- Show that each semantic step of the source program is matched with steps of the target program preserving the invariant.



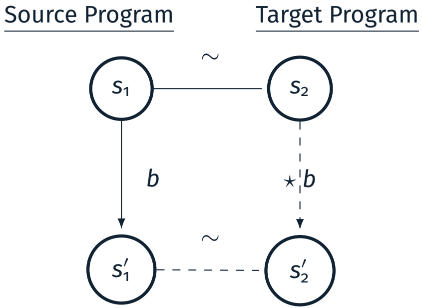
Refinement:

- Design an invariant $\sim$, a relation between monadic states.
- Show that each primitive execution of the first monadic specification is matched with a primitive execution of the second monadic specification preserving the invariant.



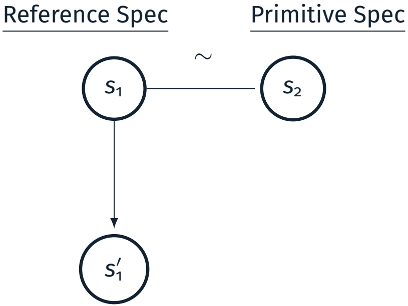
Forward Simulation in CompCert:

- Design an invariant $\sim$, a relation between semantic states.
- Show that each semantic step of the source program is matched with steps of the target program preserving the invariant.



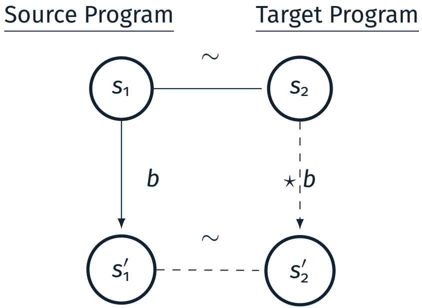
Refinement:

- Design an invariant $\sim$, a relation between monadic states.
- Show that each primitive execution of the first monadic specification is matched with a primitive execution of the second monadic specification preserving the invariant.



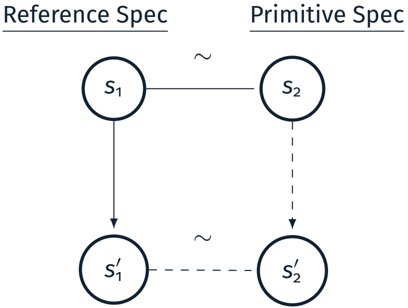
Forward Simulation in CompCert:

- Design an invariant $\sim$, a relation between semantic states.
- Show that each semantic step of the source program is matched with steps of the target program preserving the invariant.

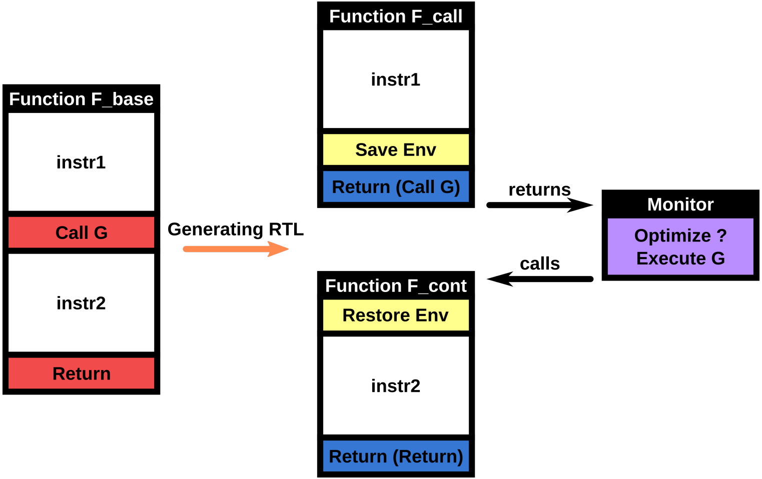


Refinement:

- Design an invariant $\sim$, a relation between monadic states.
- Show that each primitive execution of the first monadic specification is matched with a primitive execution of the second monadic specification preserving the invariant.



Custom Calling Conventions



Generating Natice Code using Primitives - An Example

CoreIR Function

```
Function Fun1 (reg1):
  reg2 ← 4 + reg1
  reg3 ← Call Fun7 (reg2)
  reg3 ← reg1 + reg3
  Return reg3
```

RTL Functions

```
$1() {
  x8 = "Pop"()
  x9 = x8 + 4 (int)
  // Save environment
  x1 = "Push" (x8)
  x1 = "Push"(Fun1, $2)
  x1 = "Push"(x9)
  x1 = "Push"(Fun7)
  return RETCALL }

$2() {
  x10 = "Pop"()
  // Restore environment
  x8 = "Pop"()
  x10 = x8 + x10
  x1 = "Push"(x10)
  return RETRET }
```

Assembler Continuation Function

```
# File generated by CompCert 3.8
$2:
leaq    32(%rsp), %rax
movq    %rax, 0(%rsp)
movq    %rbx, 8(%rsp)
call    _Pop
movq    %rax, %rbx
call    _Pop
leal    0(%eax,%ebx,1), %edi
call    _Push
movl    $RETRET, %eax
movq    8(%rsp), %rbx
addq    $24, %rsp
ret
```