Towards Formally Verified Just-in-Time compilation

Aurèle Barrière, Sandrine Blazy, David Pichardie

IRISA, Celtique

CoqPL, January 25th, 2020

Verified static compilers

CompCert, CakeML, VeLLVM…
Compilation happens *statically*.
No self-modification of code during execution.

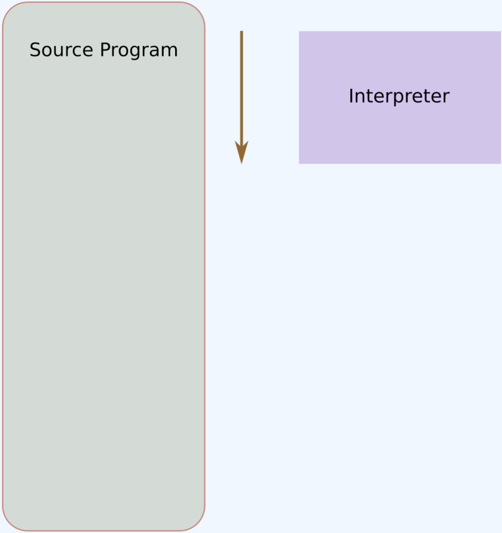
Verified static compilers

CompCert, CakeML, VeLLVM…
Compilation happens *statically*.
No self-modification of code during execution.

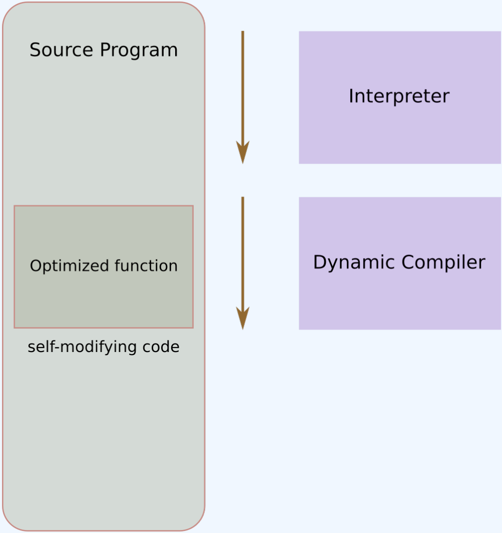
Verified static compilers

CompCert, CakeML, VeLLVM…
Compilation happens *statically*.
No self-modification of code during execution.

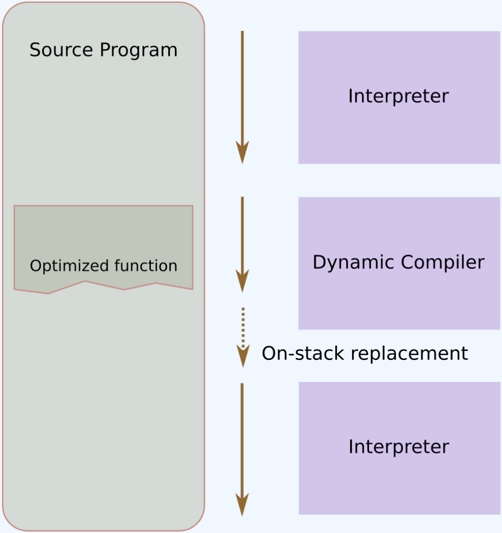
What about Just-in-Time compilation ?

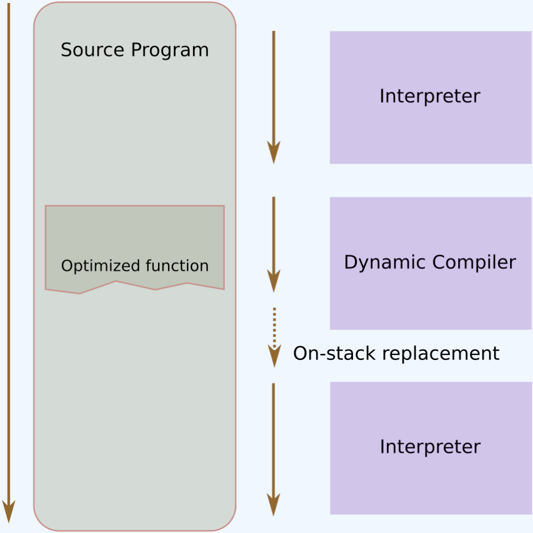


What about Just-in-Time compilation ?



What about Just-in-Time compilation ?





Verification Challenge

How can we relate this execution (with interpretation, execution of compiled code, on-stack replacement) to the semantics of the original source program ?

Definition

Compile parts of the program (source code or bytecode) during its execution. Interleaves **interpreting** the unoptimized code, **compiling** it, and **executing** the optimized code.

Exploiting Dynamic information

As the optimization is done during the execution, one can use dynamic information to speculate on the future behavior of the program.

Speculative Optimizations

Exploiting dynamic information recorded by a **profiler** allows you to create specialized versions of the program.

Example

Dynamically-typed language: each + and * polymorphic operator must check the types of its arguments each time.

```
Function f () {
  int i;
  for (i=0; i<N; i++) {
    g(a,b,array,i); }}
```

```
Function g (a,b,array,i) {
  sum[i]     = a + array[i];
  product[i] = a * (array[i] + b);}
```

```
Function f () {
  int i;
  for (i=0; i<N; i++) {
    g(a,b,array,i); }}
```

```
Function g (a,array,i) {
  sum[i]     = a + array[i];
  product[i] = a * (array[i] + b);}
```

Speculate on the type of the arguments

We can generate dynamically the following code for g:

```
Speculation : a is int /\ array[i] is int /\ b = 0
ai          = array[i];
sum[i]      = int_add(a, ai);
product[i] = int_mult(a, ai);
i = i+1;
```

Deoptimization

We must provide a way to return to the original version if the speculation does not hold.

Executing a program with a JIT

Execution

g   profiling

Program

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

Function g():
  ...
```

- Interleaves execution of optimized and non-optimized functions.
- Keep several versions of each function.
- Instructions to deoptimize and restore environment.

Execution

g	profiling
g	profiling

Program

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

Function g():
  ...
```

- Interleaves execution of optimized and non-optimized functions.
- Keep several versions of each function.
- Instructions to deoptimize and restore environment.

Execution		Program
g	profiling	
g	profiling optimizing g	

Program

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

Function g():
  ...

Function g_opt():
  ...
 Speculation
  ...
```

- Interleaves execution of optimized and non-optimized functions.
- Keep several versions of each function.
- Instructions to deoptimize and restore environment.

EXECUTING A PROGRAM WITH A JIT

Execution

g	profiling
g	profiling optimizing g
g_opt	

Program

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

Function g():
  ...

Function g_opt():
  ...
 Speculation
  ...
```

- Interleaves execution of optimized and non-optimized functions.
- Keep several versions of each function.
- Instructions to deoptimize and restore environment.

Executing a program with a JIT

Execution		Program
g	profiling	
g	profiling optimizing g	
g_opt		
g_opt	speculation fails	

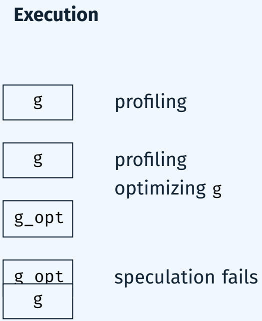
Program

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

Function g():
  ...

Function g_opt():
  ...
 Speculation
  ...
```

- Interleaves execution of optimized and non-optimized functions.
- Keep several versions of each function.
- Instructions to deoptimize and restore environment.



Program

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

Function g():
  ...

Function g_opt():
  ...
 Speculation
  ...
```

- Interleaves execution of optimized and non-optimized functions.
- Keep several versions of each function.
- Instructions to deoptimize and restore environment.

Verified Just-In-Time Compiler on x86

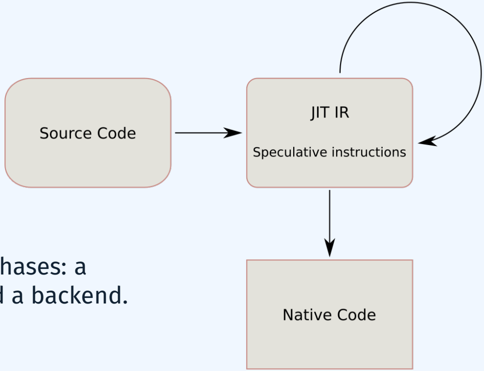
[Myreen 2010] From a stack-based bytecode to x86. Verified with HOL4.
No optimization. No speculation.

Jitk: A Trustworthy In-Kernel Interpreter Infrastructure

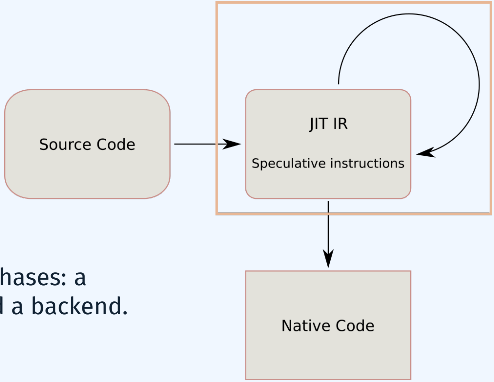
[Wang et al. 2014] Implements in-kernel interpreters, interfaced with CompCert.
No speculative optimization. No self-modifying code.

Correctness of Speculative Optimizations with Dynamic Deoptimization

[Flückiger et al. 2018] An intermediate representation, **Sourir**, designed for speculative optimization.
Paper proofs of some speculative optimizations. No mechanized proofs.



2 compilation phases: a middle-end and a backend.



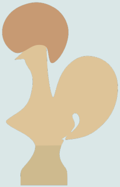
2 compilation phases: a middle-end and a backend.

Our prototype

We focus on the manipulation of a JIT IR with speculation, including middle-end compiling, interpretation, profiling...

A formally verified JIT middle-end prototype

- Realistic architecture.
  Optimizations, interpretation and speculation.
- Modular correctness proofs.
- Can be extracted and executed.
- JIT correctness theorem.



Component	Implementation	Proof
Parser	OCaml	
JIT step	Coq	✓
Interpreter	Coq	✓
Constant Propagation	Coq	✓
Adding speculation	Coq	✓
Inlining	Coq	In progress
Profiler	Ocaml	Not needed

Static Compiler correctness

If compilation succeeds, and the original program has a behavior (safe), then any behavior of the compiled program matches a behavior of the source program.

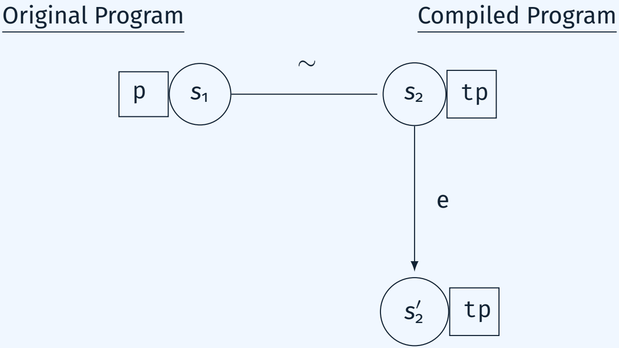
```
Theorem transf_c_program_correct:
  ∀ p tp,
    transf_c_program p = OK tp →
    backward_simulation (Csem.semantics p) (Asm.semantics tp).
```

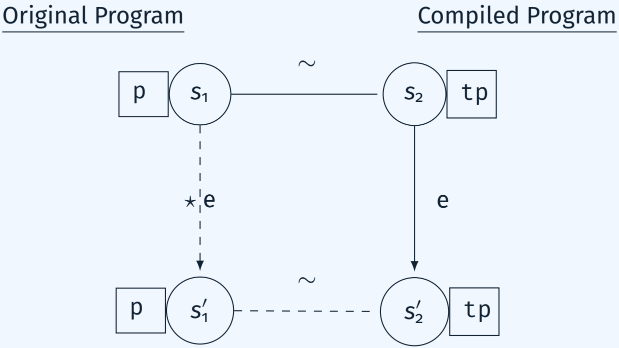
JIT correctness

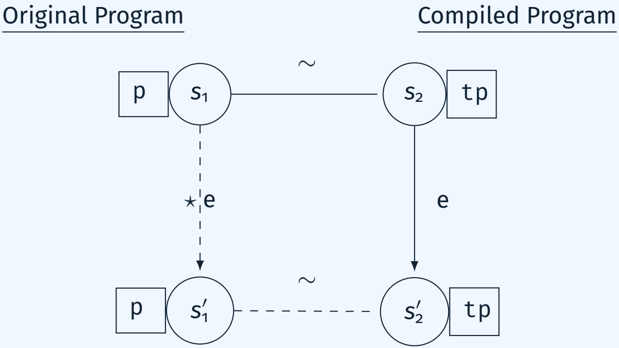
We need an interpreter correctness theorem.
If the original program is safe, then the JIT makes some progress and any of its possible executions matches a behavior of the source program semantics.

Original Program

Compiled Program


Original Program
Compiled Program
p
s_1
$\sim$
s_2
tp
e
s_2'
tp


Original Program
Compiled Program
∼
p
s1
s2
tp
⋆ e
e
∼
p
s1′
s2′
tp

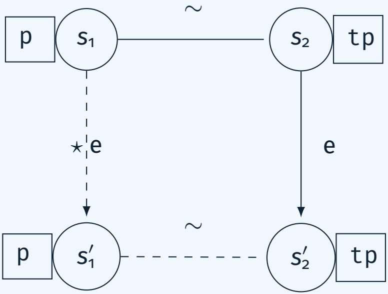


Behavior refinement

Every compiled behavior is matched by a source behavior.

Original Program

Compiled Program

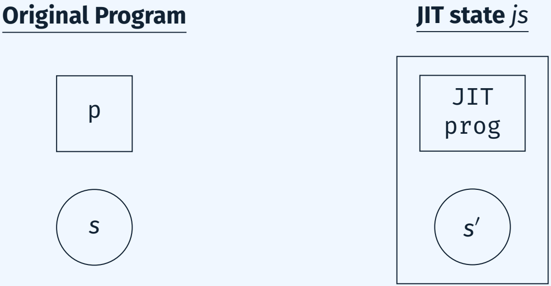


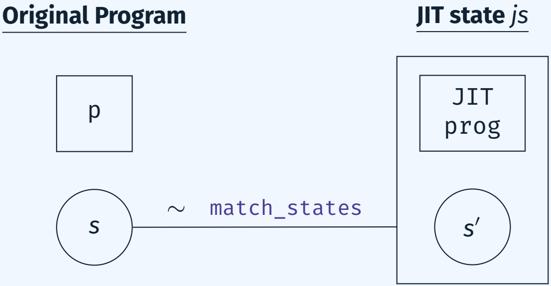
Same Program

In a static compiler, only the semantic state changes, not the program.

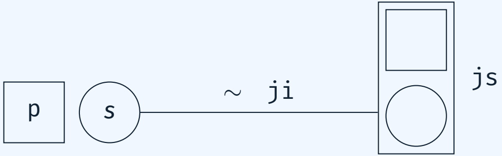
Behavior refinement

Every compiled behavior is matched by a source behavior.

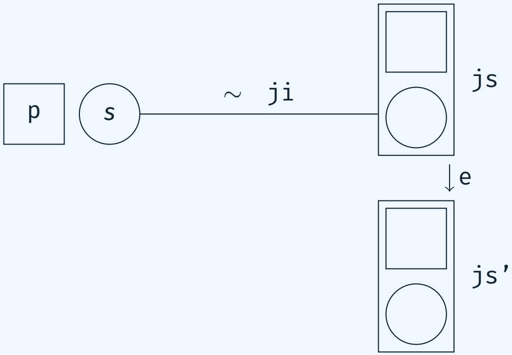

Original Program
p
s
JIT state *js*
JIT prog
s′


Original Program
JIT state *js*
p
JIT prog
s
∼ match_states
s′

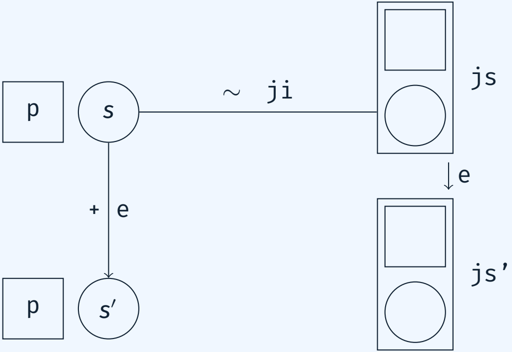

```
Theorem jit_correctness:
  ∀ (p:program) (s:state) (js:jit_state) (ji:jit_index),
  • input_prog p →
    match_states p s js ji →
    safe p s →
    ∃ js', ∃ e,
        jit.jit_step js = OK(js',e) ∧
        ((∃ s', ∃ ji', plus p s (traceof e) s' ∧ match_states p s' js' ji') ∨
        (∃ ji', match_states p s js' ji' ∧ jit_order ji' ji ∧ silent e)).
```



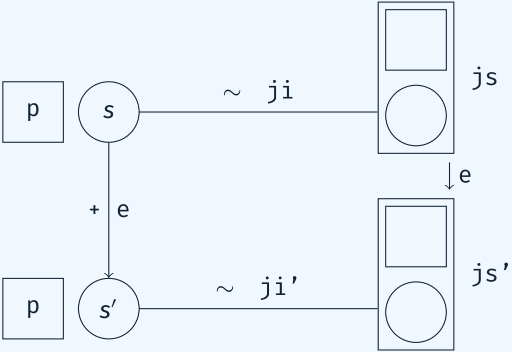

```
Theorem jit_correctness:
  ∀ (p:program) (s:state) (js:jit_state) (ji:jit_index),
    input_prog p →
  • match_states p s js ji →
    safe p s →
    ∃ js', ∃ e,
      jit.jit_step js = OK(js',e) ∧
      ((∃ s', ∃ ji', plus p s (traceof e) s' ∧ match_states p s' js' ji') ∨
      (∃ ji', match_states p s js' ji' ∧ jit_order ji' ji ∧ silent e)).
```



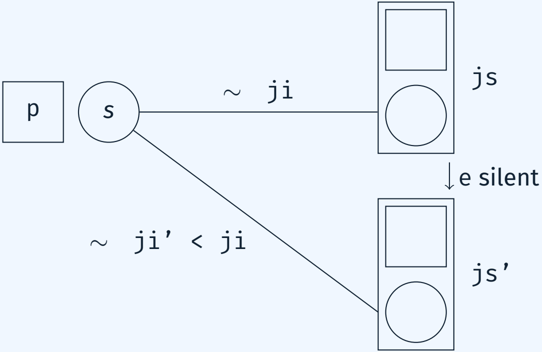

```
Theorem jit_correctness:
  ∀ (p:program) (s:state) (js:jit_state) (ji:jit_index),
    input_prog p →
    match_states p s js ji →
    safe p s →
    ∃ js', ∃ e,
      jit.jit_step js = OK(js',e) ∧
      ((∃ s', ∃ ji', plus p s (traceof e) s' ∧ match_states p s' js' ji') ∨
      (∃ ji', match_states p s js' ji' ∧ jit_order ji' ji ∧ silent e)).
```




```
Theorem jit_correctness:
 ∀ (p:program) (s:state) (js:jit_state) (ji:jit_index),
   input_prog p →
   match_states p s js ji →
   safe p s →
   ∃ js’, ∃ e,
     jit.jit_step js = OK(js’,e) ∧
     ((∃ s’, ∃ ji’, plus p s (traceof e) s’ ∧ match_states p s’ js’ ji’) ∨
     (∃ ji’, match_states p s js’ ji’ ∧ jit_order ji’ ji ∧ silent e)).
```




```
Theorem jit_correctness:
  ∀ (p:program) (s:state) (js:jit_state) (ji:jit_index),
    input_prog p →
    match_states p s js ji →
    safe p s →
    ∃ js', ∃ e,
        jit.jit_step js = OK(js',e) ∧
        ((∃ s', ∃ ji', plus p s (traceof e) s' ∧ match_states p s' js' ji') ∨
        (∃ ji', match_states p s js' ji' ∧ jit_order ji' ji ∧ silent e)).
```




```
Theorem jit_correctness:
 ∀ (p:program) (s:state) (js:jit_state) (ji:jit_index),
   input_prog p →
   match_states p s js ji →
   safe p s →
   ∃ js', ∃ e,
       jit.jit_step js = OK(js',e) ∧
       ((∃ s', ∃ ji', plus p s (traceof e) s' ∧ match_states p s' js' ji') ∨
       (∃ ji', match_states p s js' ji' ∧ jit_order ji' ji ∧ silent e)).
```

Summary

- Untyped, simple integer values, simple memory.
- Similar to CompCert RTL.
- An Assume instruction, the same as in Sourir ([Flückiger et al. 2018]).
- Function versions.

The only language of our JIT

- No backend compilation yet. Optimized code is also interpreted.
- The initial program should not have any speculation, and only one version per function.

Syntax

```
Assume (expr list) target (varmap) [synth frame list]
```

- `expr list`: the speculation
- `target`: deoptimization target
- `varmap`: restore the register environment
- `synth frame list`: restore extra stack frames

Example

```
Assume (x = 0, y = 3)  F.V1.lbl5 {(a,10)} []
```

- First, test if (x = 0) and (y = 3) hold.
- If not, deoptimize to function `F`, version `V1`, line `<lbl5>`.
- Put value 10 in register `a`.

Speculating on the values of function arguments.
The profiler records the values at each function call.

Example

```
Function F (r1, r2) :
  Version V1:
  <lbl1> Return (r1 + r2)
```

Speculating on the values of function arguments.
The profiler records the values at each function call.

Example

```
Function F (r1, r2) :
  Version V1:
  <lbl1> Return (r1 + r2)
```

The new Version

```
  Version V2:
  <lbl0> Assume (r2 = 10)  F.V1.lbl1 {(r1,r1) (r2,r2)} []
  <lbl1> Return (r1 + r2)
```

F.V1.lbl1: deoptimize to Function F, Version V1, line <lbl1>.

Optimizes the function based on the previously inserted speculation.

Example

```
Function F (r1, r2, r3) :
  Version 1:
  r1 = 4
  Assume (r2 = 0)  G.V2.lbl3 {(r1,r1) (r2,r2)} []
  Return r1 + r2 + r3
```

The optimized version

```
  Version 2:
  r1 = 4
  Assume (r2 = 0)  G.V2.lbl3 {(r1,4) (r2,r2)} []
  Return 4 + r3
```

Verification

Uses a fixpoint solver library from CompCert.

Replaces a function call by its code.
Name-mangling and synthesizing new stackframes in `Assume`.

Changing Assumptions in the inlined code

`Assume (r1 = 4) H.V2.lbl7 (r1,r1)` in the inlined code becomes
`Assume (R1 = 4) H.V2.lbl7 (r1,R1) [f.v.l ret]`
Where

- `R1` is the mangled name of `r1`.
- `f.v.l` is the location of the instruction after the call in the original caller function.
- `ret` is the variable of the caller function that receives the callee's return.

Reusing CompCert Forward Simulation Methodology

Show that each step of the program before the optimization matches some steps in the program after optimization.
Forward to backward theorem: a forward simulation implies a backward simulation.

Proving the JIT correct

We showed that, if each optimization pass is proved, the entire JIT is correct.
Every behavior of the JIT matches a behavior of the original program.

```
Theorem optimization_correctness:
∀ p ps newp,
  optimize ps p = OK (newp) →
  spec_wf p →
  ∃ order, ∃ (r:relation),
     bwd_sim p newp order r ∧ reflexive_wf p r.
```

A Coq JIT

- A Coq model of a realistic JIT architecture.
- An executable prototype.
- A backward simulation for JIT correctness.

Verification work

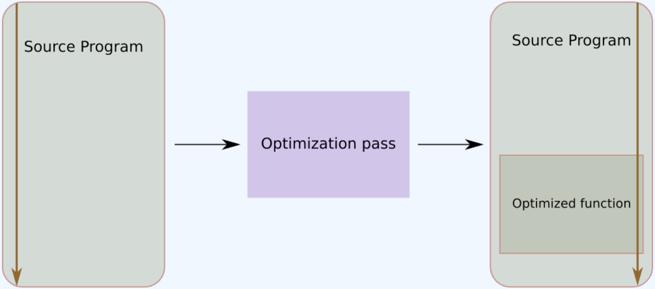
Adding an optimization pass in the JIT middle-end can be proved with the same forward simulation methodology as CompCert.

A Coq JIT

- A Coq model of a realistic JIT architecture.
- An executable prototype.
- A backward simulation for JIT correctness.

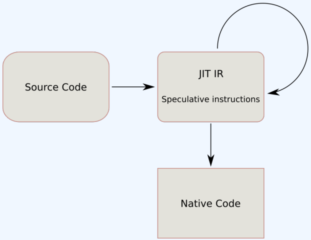
Verification work

Adding an optimization pass in the JIT middle-end can be proved with the same forward simulation methodology as CompCert.



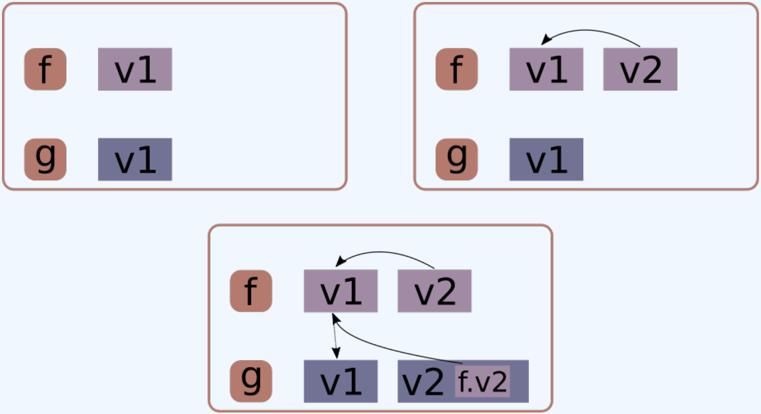
Sourir Transparency Invariant

From [Flückiger et al. 2018].
Prove that deoptimizing, even when the conditions hold, does not change the behavior of the program.
Useful in some speculation-specific optimizations.



Backend compilation

Using the translation of CompCert ? Its specification doesn't suit our needs.


f
v1
g
v1
f
v1
v2
g
v1
f
v1
v2
g
v1
v2
f.v2