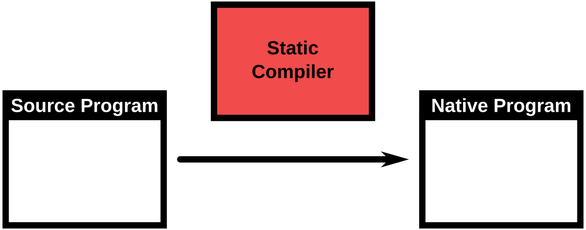
Verified native code generation in a JIT compiler

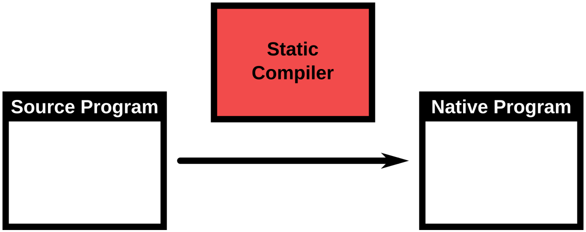
Cambium Seminar

Aurèle Barrière Sandrine Blazy David Pichardie

May 30th, 2022

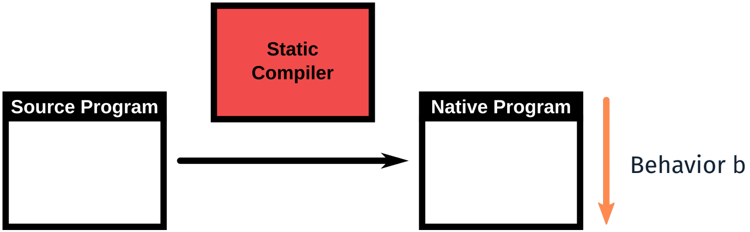
Formally verified **static** compilation





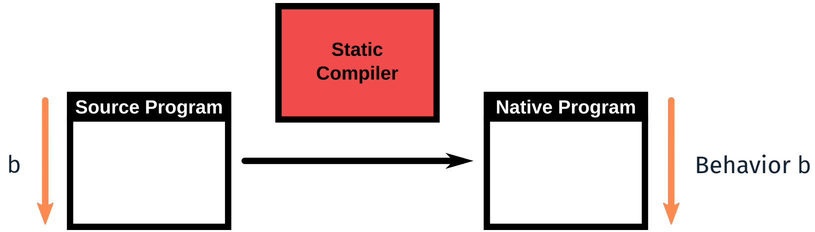
State of the Art: Verified static compilers

CompCert [Leroy 2006], CakeML [Kumar et al. 2014], VeLLVM [Zhao et al. 2012].
Compilation happens **statically**: the code is produced before its execution.



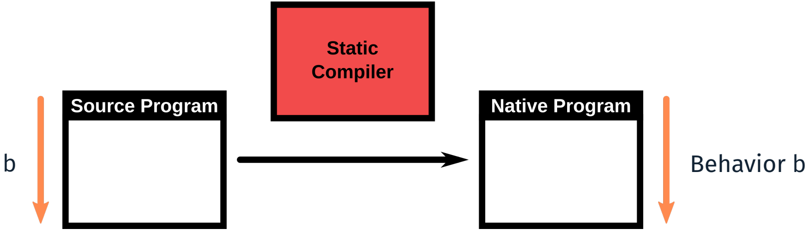
State of the Art: Verified static compilers

CompCert [Leroy 2006], CakeML [Kumar et al. 2014], VeLLVM [Zhao et al. 2012].
Compilation happens **statically**: the code is produced before its execution.



State of the Art: Verified static compilers

CompCert [Leroy 2006], CakeML [Kumar et al. 2014], VeLLVM [Zhao et al. 2012].
Compilation happens **statically**: the code is produced before its execution.



State of the Art: Verified static compilers

CompCert [Leroy 2006], CakeML [Kumar et al. 2014], VeLLVM [Zhao et al. 2012].
Compilation happens **statically**: the code is produced before its execution.

What about JIT compilation verification?

JIT compilation: Interleave execution and optimization of the program.

Execution Stack

Interpreter: f

Program

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

Function g():
 g1
 g2
```

Execution Stack

Interpreter: `f`

Interpreter: `g`

Program

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

Function g():
 g1
 g2
```

Executing a program with a JIT with speculative optimizations

Execution Stack

Interpreter: `f`

Optimizing Compiler

Program

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

Function g():
 g1
 g2

Function g_x86():
 g1
 Speculation (x=7)
 g2'
```

Execution Stack

Interpreter: `f`

Native: `g_x86`

Program

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

Function g():
 g1
 g2

Function g_x86():
 g1
 Speculation (x=7)
 g2'
```

Execution Stack

Interpreter: `f`

Native: `g_x86`

Speculation fails

Program

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

Function g():
 g1
 g2

Function g_x86():
 g1
 Speculation (x=7)
 g2'
```

Executing a program with a JIT with speculative optimizations

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=7)
 g2'
```

Executing a program with a JIT with speculative optimizations

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=7)
 g2'
```

Deoptimization requires the JIT to

- Synthesize interpreter stackframes in the middle of a function.
- Possibly synthesize many stackframes at once.

With speculation, JITs need precise execution stack manipulation.

Our Goals

A **verified** and **executable** JIT in Coq.

Modern and efficient JIT compilers features:

- Dynamic Optimizations.
- With native code generation and execution.
- With speculation and on-stack replacement.

Proof modularity and reusability:

- Using CompCert as a backend compiler (translating *RTL* to *x86*).
- Reusing CompCert's backend proof.
- Reusing CompCert's proof methodology (simulation framework).

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 →
  (∀ beh, program_behaves (Csem.semantics p) beh → not_wrong beh) →
  (∀ beh, program_behaves (Asm.semantics tp) beh → program_behaves (Csem.semantics p) beh).
```

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 →
  (∀ beh, program_behaves (Csem.semantics p) beh → not_wrong beh) →
  (∀ beh, program_behaves (Asm.semantics tp) beh → program_behaves (Csem.semantics p) beh).
```

JIT Theorem

If the semantics (`CoreIR_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_same_safe_behavior:
  ∀ (p:program),
    (∀ beh, program_behaves (CoreIR_sem p) beh → not_wrong beh) →
    (∀ beh, program_behaves (jit_sem p) beh →
                              program_behaves (CoreIR_sem p) beh).
```

How do we define `jit_sem` ?

JIT-specific verification problems

- Speculative optimizations.
- Dynamic Optimizations interleaved with execution.
- Impure and non-terminating components.
- Integrate the correctness proof of a static compiler backend.

JIT-specific verification problems

- Speculative optimizations.
- Dynamic Optimizations interleaved with execution.
- Impure and non-terminating components.
- Integrate the correctness proof of a static compiler backend.

Previous Work: Formally verified speculation and deoptimization in a JIT compiler, POPL21

Aurèle Barrière, Sandrine Blazy, Olivier Flückiger, David Pichardie, Jan Vitek.
`https://github.com/Aurele-Barriere/CoreJIT`

- CoreIR, inspired by RTL and speculative instructions ([Flückiger et al. 2018]).
- Correctness theorem of CoreJIT with interpretation, dynamic optimizations, and speculations.

JIT-specific verification problems

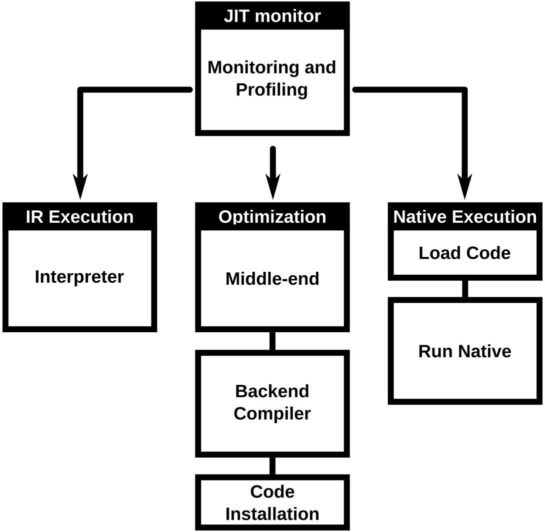
- Speculative optimizations.
- Dynamic Optimizations interleaved with execution.
- Impure and non-terminating components.
- Integrate the correctness proof of a static compiler backend.

Previous Work: Formally verified speculation and deoptimization in a JIT compiler, POPL21

Aurèle Barrière, Sandrine Blazy, Olivier Flückiger, David Pichardie, Jan Vitek.
`https://github.com/Aurele-Barriere/CoreJIT`

- CoreIR, inspired by RTL and speculative instructions ([Flückiger et al. 2018]).
- Correctness theorem of CoreJIT with interpretation, dynamic optimizations, and speculations.

A theorem about IR to IR transformation. No native code generation in the formal model.

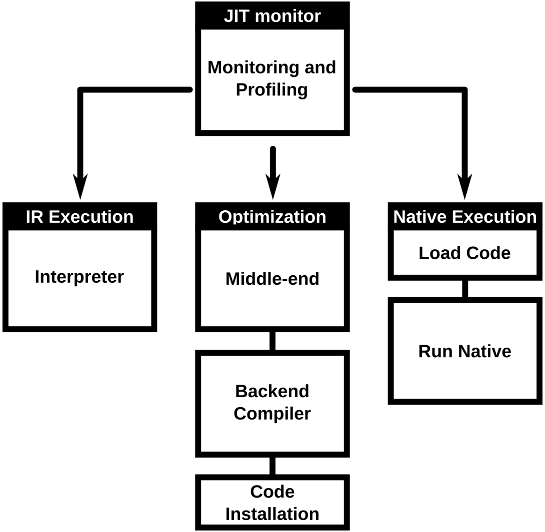


JIT architecture

Extends the architecture from [Barrière et al. 2021] with native code generation and execution.

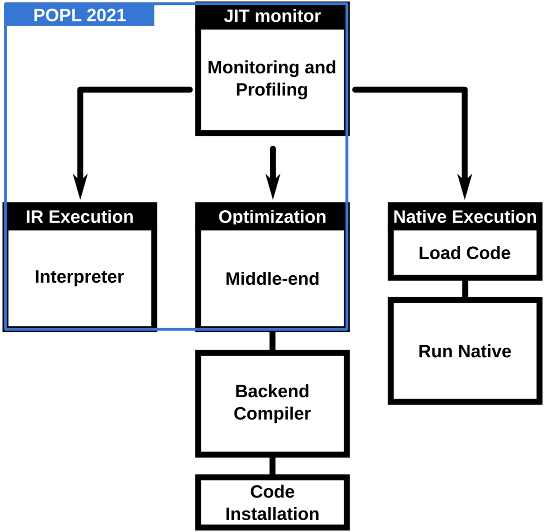
JIT loop

The **monitor** chooses the next step: execution or optimization.
Profiling: records information about the execution and suggests speculations.



Interpreter

Interpret the IR code that has not been compiled to native.



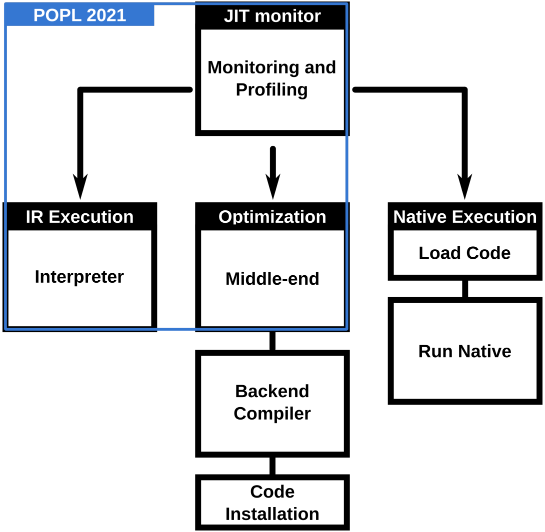
Middle-end Optimizer

From the IR to the IR.
Inserts speculation.

POPL21

The correctness theorem of our previous work is about these components.
A Coq proof that any behavior of this JIT prototype is a behavior of the input program.

A JIT Architecture

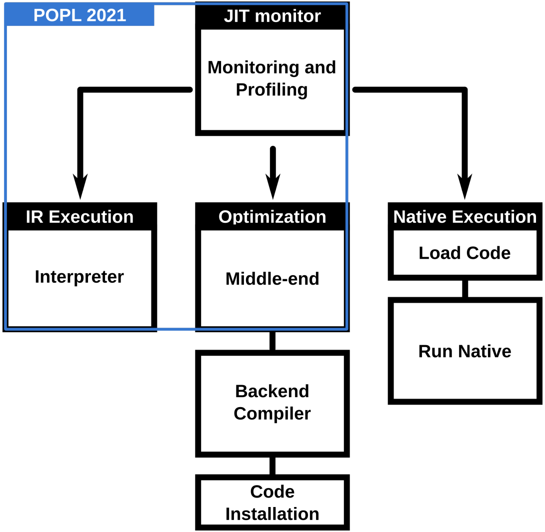


Backend Compilation

Generates native code, as in a static compiler backend.
Use the CompCert backend from RTL to x86.

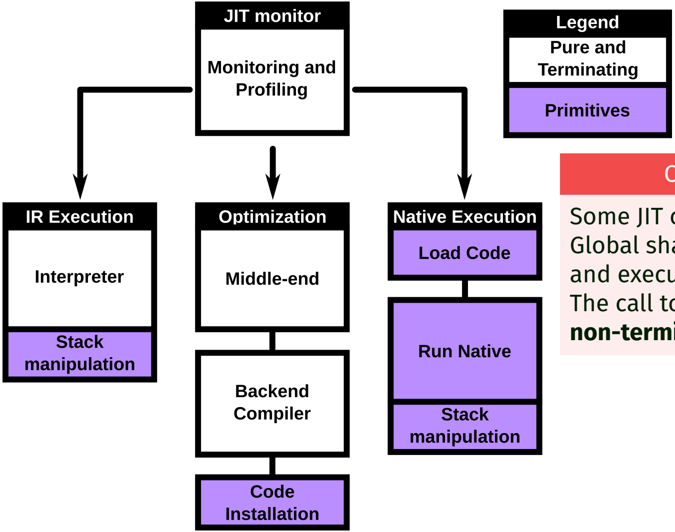
Code Installation

Install the dynamically generated code in memory.
Make it executable.



Setting up native execution
Get a function pointer for the installed code.

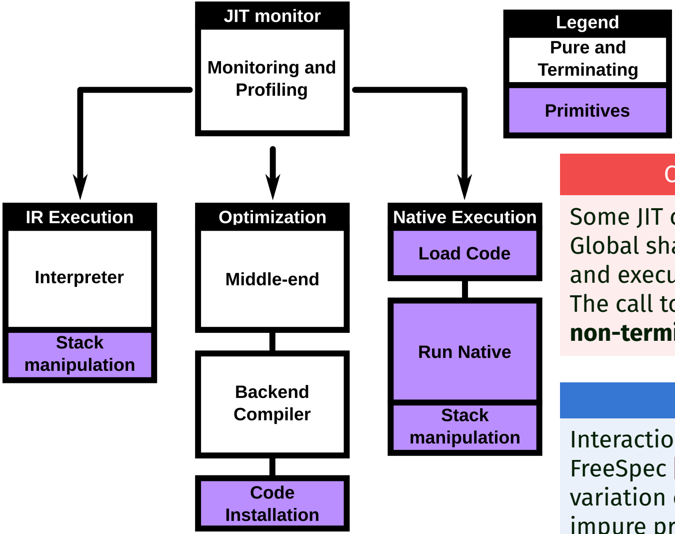
Native Code Execution
Run the generated code.



Can we really write a JIT in Coq?

Some JIT components are **impure**.
Global shared data-structures: execution stack and executable memory.
The call to native code may even be **non-terminating**.

A JIT Architecture



Can we really write a JIT in Coq?

Some JIT components are **impure**.
Global shared data-structures: execution stack and executable memory.
The call to native code may even be **non-terminating**.

Free Monads

Interaction Trees [Xia et al. 2020] and FreeSpec [Letan and Régis-Gianas 2020] use a variation of the **free monad** to reason about impure programs in Coq.

State monads are perfect to specify functions with an effect on 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.
```

State monads are perfect to specify functions with an effect on 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.


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.
```

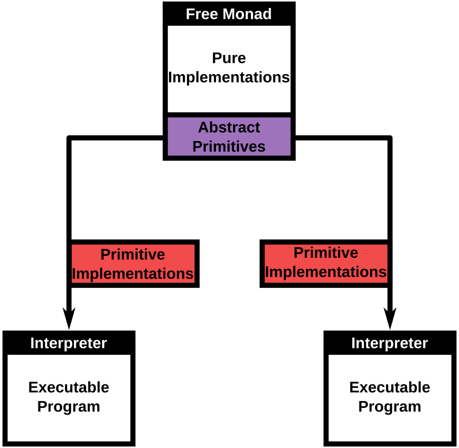
State monads are perfect to specify functions with an effect on a global state. Either the function fails, or it succeeds and returns the next global state. Found in CompCert.

```
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.


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 JIT

This is fine to specify the primitives, but the actual JIT should execute actual impure primitives.

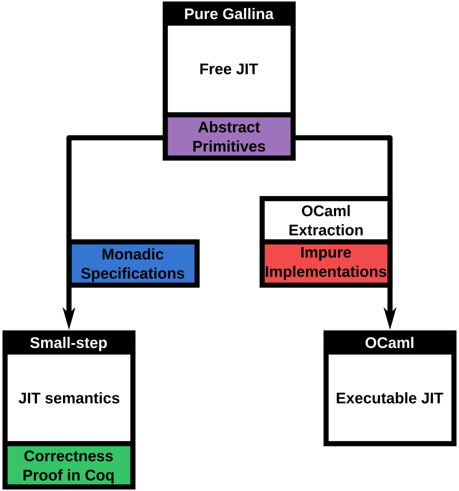
A Free Monad



Some parts of the JIT can be written in Coq, some can't. Let's find a way to write in Coq exactly the parts we want to extract to OCaml.
Free Monad: Representing programs where some impure primitives have yet to be implemented.

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

With different primitive implementations, the program can be executed differently.

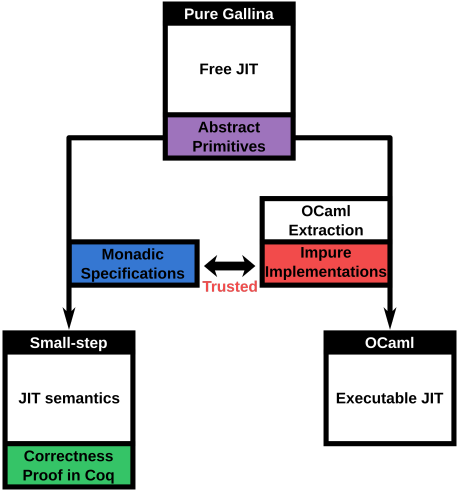


The Free JIT

A Free JIT without primitive implementations.
Given specifications, define small-step semantics.
Extract to OCaml with impure implementations.

Inspired by Free Monads, but adapted to fit the simulation framework of CompCert.

Our strategy for a verified executable impure JIT



The Free JIT

A Free JIT without primitive implementations.
Given specifications, define small-step semantics.
Extract to OCaml with impure implementations.

Inspired by Free Monads, but adapted to fit the simulation framework of CompCert.

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.
```

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 Free Monads:

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

```
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.
```

Given primitive implementations, we want to turn a free monad into an executable state monad. An **implementation** is one state monad for each primitive:

```
Record monad_impl: Type :=
  mk_mon_imp {
      prim_get: state_mon nat;
      prim_put: nat → state_mon unit; }.
```

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

Given primitive implementations, we want to turn a free monad into an executable state monad. An **implementation** is one state monad for each primitive:

```
Record monad_impl: Type :=
  mk_mon_imp {
     prim_get: state_mon nat;
     prim_put: nat → state_mon unit; }.
```

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

We can now give semantics to our Free Monads:

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

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

```
(* impure primitives *)
let nm_exec_prim (p:'x primitive) : 'x =
match p with
  | Get -> !global
  | Put (n) -> global := n

(* executing free monads *)
let rec nm_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 = nm_exec_prim prim in
     nm_exec (cont x)
```

Every JIT component can be written as a Free Monad:

```
Definition optimizer (f:function) : free unit :=
    do f_rtl ← ret (IRtoRTL f);
    do f_x86 ← ret (backend f_rtl); (* using CompCert backend *)
    Prim_Install_Code f_x86.
```

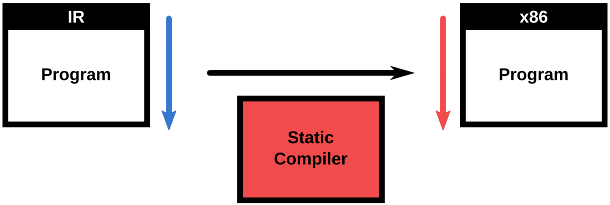
Every JIT component can be written as a Free Monad:

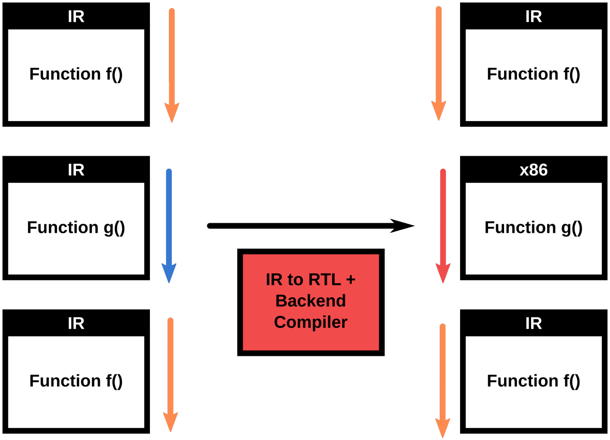
```
Definition optimizer (f:function) : free unit :=
    do f_rtl ← ret (IRtoRTL f);
    do f_x86 ← ret (backend f_rtl); (* using CompCert backend *)
    Prim_Install_Code f_x86.
```

C implementation

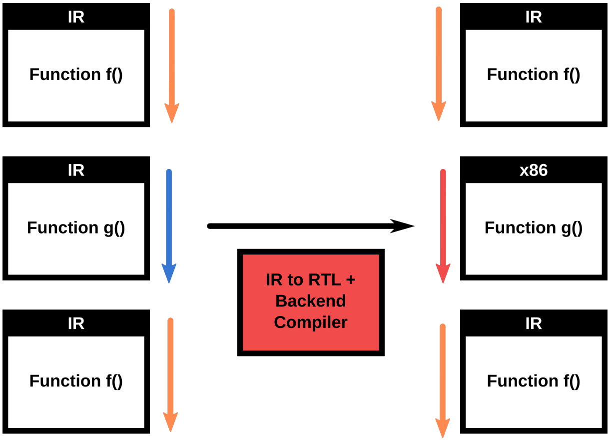
- Calls an assembler to produce binary code.
- Allocates writable memory with `mmap`.
- Writes the binary code in that memory.
- Makes the memory executable with `mprotect`.

CompCert preserves the observable behavior of the program.





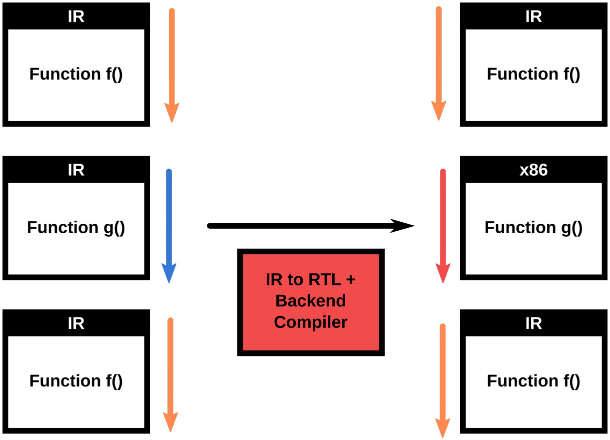
CompCert preserves the observable behavior of the program.



CompCert preserves the observable behavior of the program.

CompCert as a JIT backend

Compiles whole programs (no arguments).
Effects on the stack and heap should be preserved too.

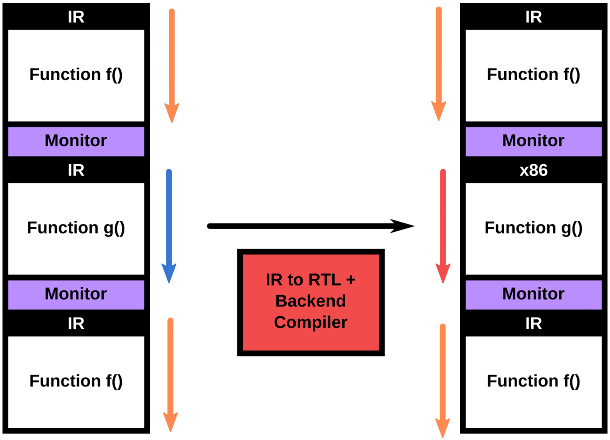


CompCert preserves the observable behavior of the program.

CompCert as a JIT backend

Compiles whole programs (no arguments). Effects on the stack and heap should be preserved too.

Make the generated code call the primitives. The stack and heap are external, not part of the CompCert memory model.



CompCert preserves the observable behavior of the program.

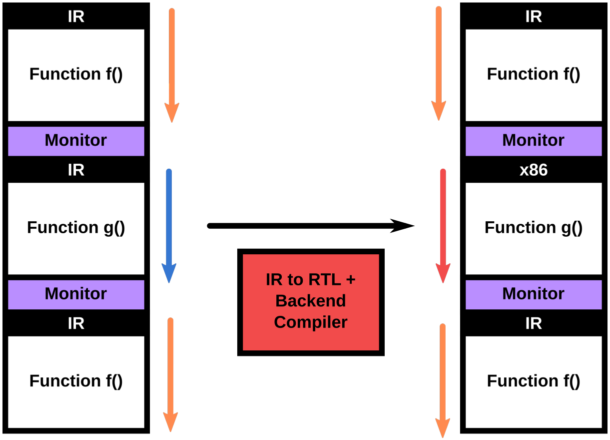
CompCert as a JIT backend

Compiles whole programs (no arguments). Effects on the stack and heap should be preserved too.

Make the generated code call the primitives. The stack and heap are external, not part of the CompCert memory model.

Compiling Function Calls

We have to go through the monitor.



CompCert preserves the observable behavior of the program.

CompCert as a JIT backend

Compiles whole programs (no arguments). Effects on the stack and heap should be preserved too.

Make the generated code call the primitives. The stack and heap are external, not part of the CompCert memory model.

Compiling Function Calls

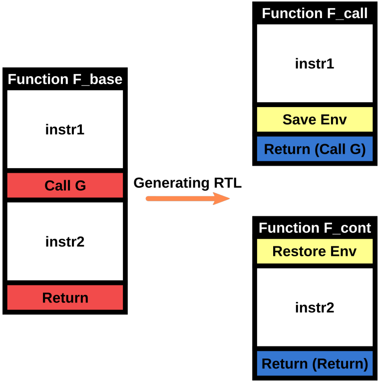
We have to go through the monitor.

Split the functions at calls.

Generating Several RTL Programs

Generating RTL code that uses custom calling conventions with our primitives.

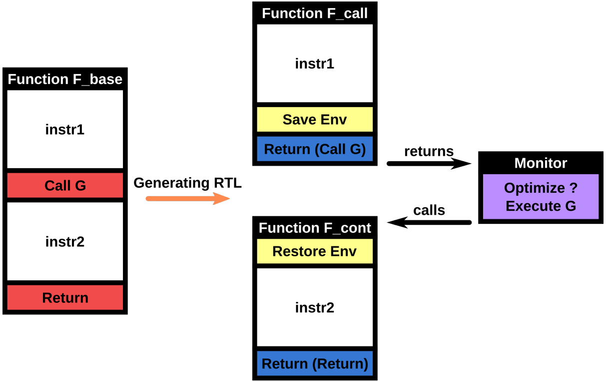
- Primitives are *external calls*.
- Each RTL function returns to the monitor.
- One Continuation per Call instruction.



Generating Several RTL Programs

Generating RTL code that uses custom calling conventions with our primitives.

- Primitives are *external calls*.
- Each RTL function returns to the monitor.
- One Continuation per Call instruction.

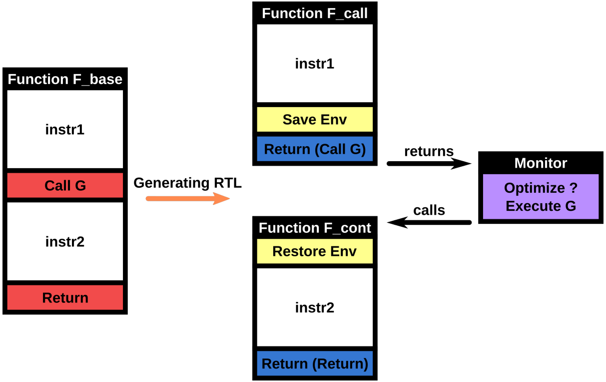


Generating Several RTL Programs

Generating RTL code that uses custom calling conventions with our primitives.

- Primitives are *external calls.*
- Each RTL function returns to the monitor.
- One Continuation per Call instruction.

CompCert does not handle the heap and stack. It interacts with it through primitive calls.



CoreIR Function

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

CoreIR Function

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

RTL Functions

```
$1() {
  x8 = "Pop"()
  x9 = x8 + 4 (int)
  x1 = "Push" (x8)
  x1 = "Close"(1, 2, 10)
  x1 = "Push"(x9)
  x1 = "Push"(1)
  x1 = "Push"(7)
  x7 = RETCALL
  return x7 }

$1() {
  x10 = "Pop"()
  x8 = "Pop"()
  x10 = x8 + x10
  x1 = "Push"(x10)
  x7 = RETRET
  return x7 }
```

CoreIR Function

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

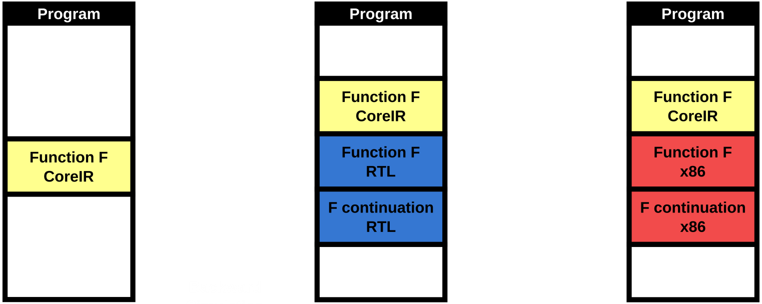
RTL Functions

```
$1() {
  x8 = "Pop"()
  x9 = x8 + 4 (int)
  x1 = "Push" (x8)
  x1 = "Close"(1, 2, 10)
  x1 = "Push"(x9)
  x1 = "Push"(1)
  x1 = "Push"(7)
  x7 = RETCALL
  return x7 }

$1() {
  x10 = "Pop"()
  x8 = "Pop"()
  x10 = x8 + x10
  x1 = "Push"(x10)
  x7 = RETRET
  return x7 }
```

Assembler Continuation Function

```
# File generated by CompCert 3.8
$1:
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
```

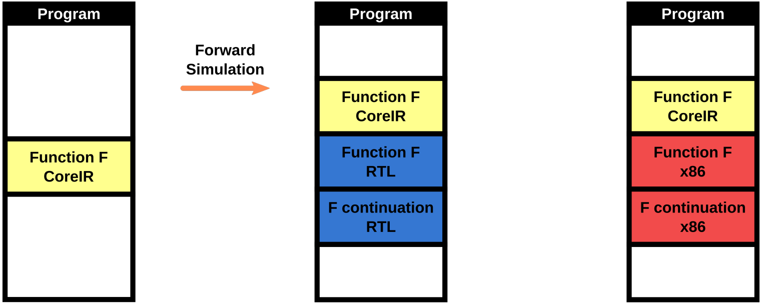
To get behavior equivalence, we need to prove *backward simulations* (from CompCert).



We keep the original version of F in case of deoptimizations.

From CoreIR to RTL: generate new calling conventions.

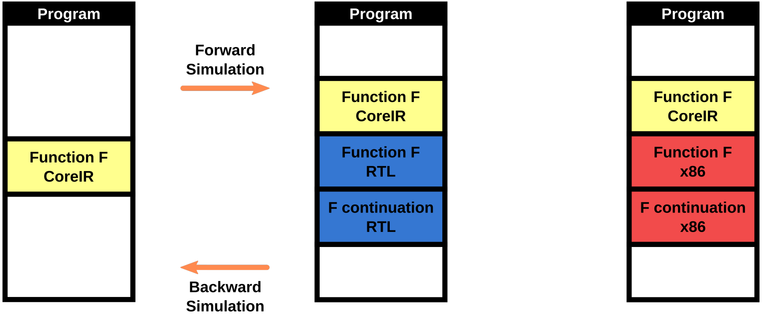
To get behavior equivalence, we need to prove *backward simulations* (from CompCert).



We keep the original version of F in case of deoptimizations.

From CoreIR to RTL: generate new calling conventions.
A forward simulation is easier to prove.

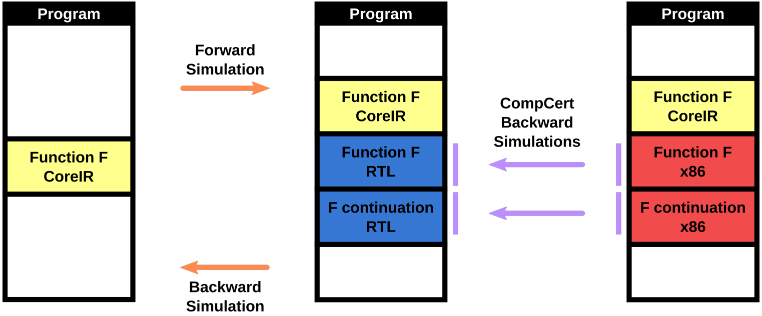
To get behavior equivalence, we need to prove *backward simulations* (from CompCert).



We keep the original version of F in case of deoptimizations.

From CoreIR to RTL: generate new calling conventions.
A forward simulation is easier to prove.
And can be used to prove a backward one.

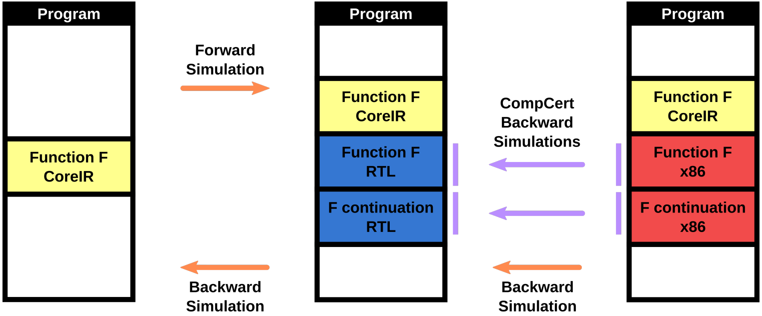
To get behavior equivalence, we need to prove *backward simulations* (from CompCert).



We keep the original version of F in case of deoptimizations.

From RTL to x86: use CompCert for the function and its continuations.

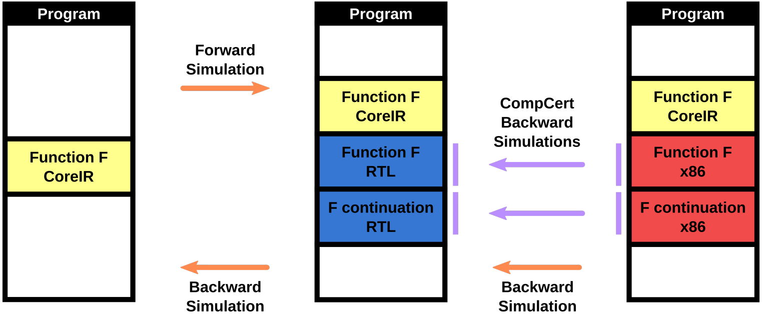
To get behavior equivalence, we need to prove *backward simulations* (from CompCert).



We keep the original version of F in case of deoptimizations.

From RTL to x86: use CompCert for the function and its continuations.
Use the CompCert simulations to prove a simulation for the entire program.

To get behavior equivalence, we need to prove *backward simulations* (from CompCert).



We keep the original version of F in case of deoptimizations.

```
Theorem optimizer_correct:
  ∀ p p', exec (optimizer p) = SOK p' →
    backward_simulation p p'.
```

Output, Stack and Heap Primitives

- `Print`
- `Pop` and `Push`
- `MemSet` and `MemGet`
- Push and pop entire interpreter stackframes

Code Segment Primitives

- Install a native function in the executable memory.
- Load a function (or one of its continuations).
- Check if a function has been compiled.

Running Native Code

We define a special primitive to run native code.
Its specification is a monad describing the small-step semantics of x86 code.

Output, Stack and Heap Primitives

- `Print`
- `Pop` and `Push`
- `MemSet` and `MemGet`
- Push and pop entire interpreter stackframes

Can be called from the native code.

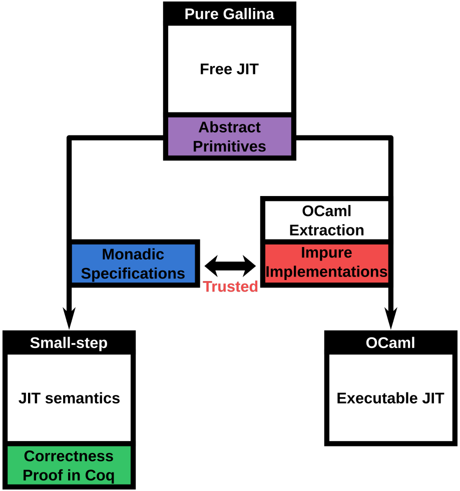
Code Segment Primitives

- Install a native function in the executable memory.
- Load a function (or one of its continuations).
- Check if a function has been compiled.

Running Native Code

We define a special primitive to run native code.
Its specification is a monad describing the small-step semantics of x86 code.

Bridging the gap between specification and implementation



What if there is a significant distance between the monadic specification and the impure implementation?

Monadic Specification (Coq)

A list of stackframe: its structure helps us write simulation invariants.

```
Record ASM_stackframe: Type := mk_sf {
  caller: int;
  next_pc: int;
  retreg: int;
  live_regs: list int }.
```

```
(* List of complete stackframes and
   the incomplete one at the top *)
Definition stack: Type :=
  list ASM_stackframe * list int.
```

Monadic Specification (Coq)

A list of stackframe: its structure helps us write simulation invariants.

```
Record ASM_stackframe: Type := mk_sf {
  caller: int;
  next_pc: int;
  retreg: int;
  live_regs: list int }.
```

```
(* List of complete stackframes and
   the incomplete one at the top *)
Definition stack: Type :=
  list ASM_stackframe * list int.
```

Impure implementation (C)

Unstructured array that the native code can access.

```
int stack[STACK_SIZE];
int sp = 0;
```

Monadic Specification (Coq)

A list of stackframe: its structure helps us write simulation invariants.

```
Record ASM_stackframe: Type := mk_sf {
  caller: int;
  next_pc: int;
  retreg: int;
  live_regs: list int }.
```

```
(* List of complete stackframes and
   the incomplete one at the top *)
Definition stack: Type :=
  list ASM_stackframe * list int.
```

An intermediate Monadic Specification (Coq)

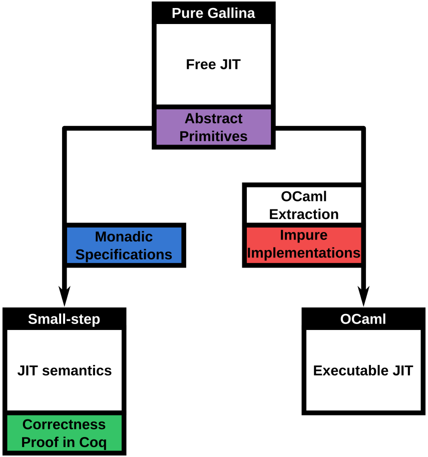
Unstructured specification, closer to the C implementation.

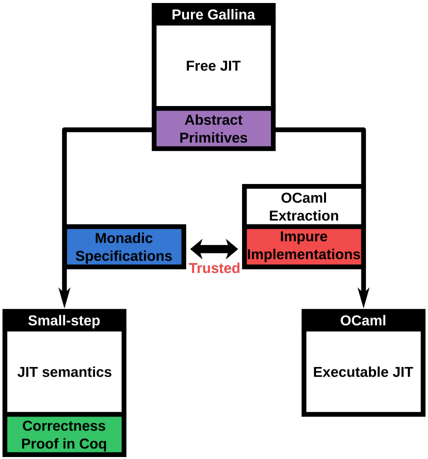
```
Definition stack: Type := list int.
```

Impure implementation (C)

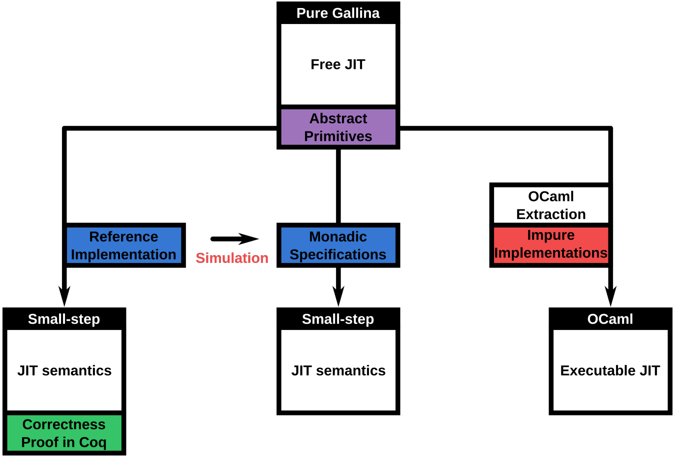
Unstructured array that the native code can access.

```
int stack[STACK_SIZE];
int sp = 0;
```

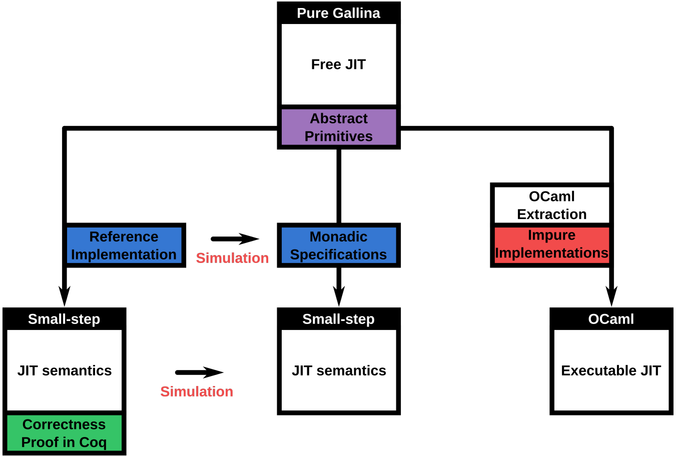

Pure Gallina
Free JIT
Abstract Primitives
Monadic Specifications
OCaml Extraction
Impure Implementations
Small-step
JIT semantics
Correctness Proof in Coq
OCaml
Executable JIT


Pure Gallina
Free JIT
Abstract
Primitives
OCaml
Extraction
Monadic
Specifications
Impure
Implementations
Trusted
Small-step
JIT semantics
Correctness
Proof in Coq
OCaml
Executable JIT

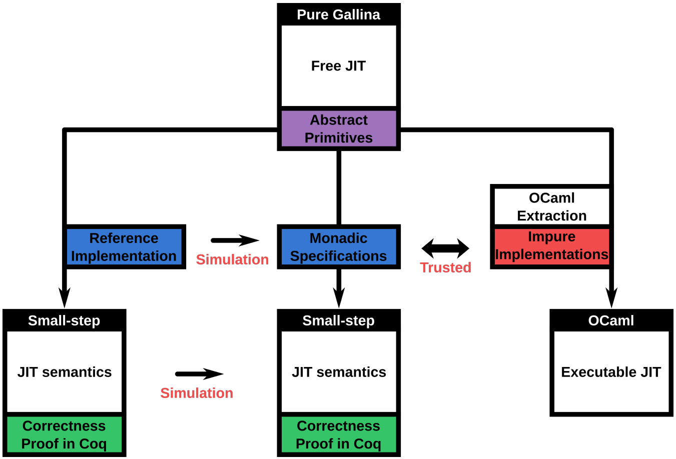
Refinement with Implementation Simulation



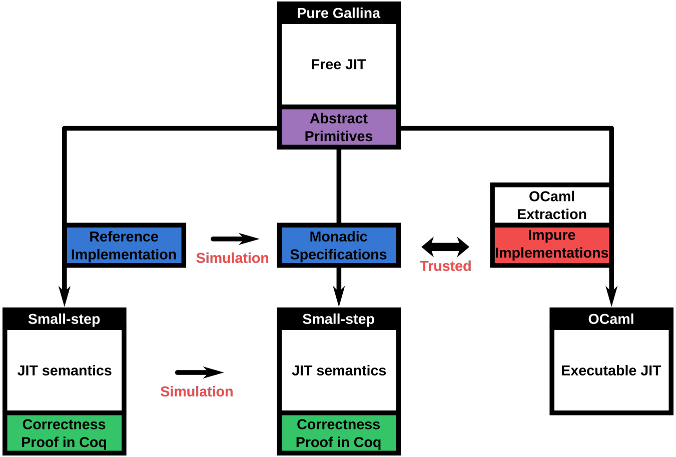
Refinement with Implementation Simulation



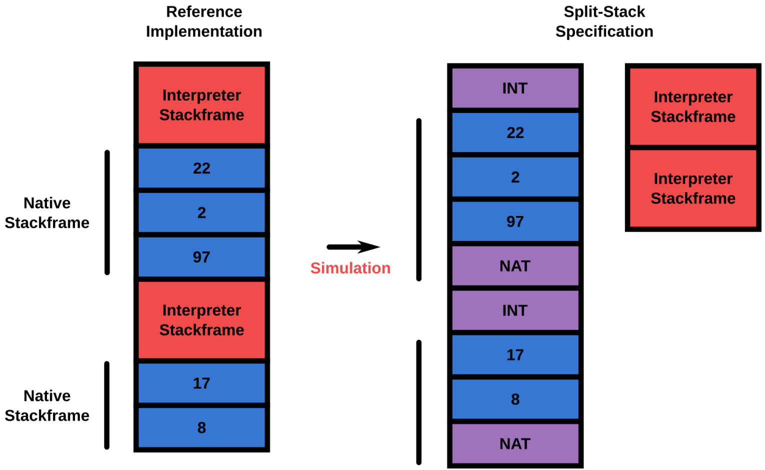
Refinement with Implementation Simulation



Refinement with Implementation Simulation




```
Theorem refines:
  ∀ prog i j
    (R: implementation_simulation i j),
    forward_simulation (monad_sem prog i) (monad_sem prog j).
```



Split Stack

Optimization proofs are easier to conduct on a single mixed stack. But stack primitives called from the native code should only interact with an array of integers.

A Free JIT

- We can derive both small-step semantics and an executable OCaml JIT **(ongoing)**.
- Native code generation and execution are part of the formal model.
- Each pure JIT component is properly specified and proved.
- Each impure component is specified with a state monad.
- A correctness proof of the JIT small-step semantics.
- We reuse the simulation methodology of CompCert.
- We reuse the simulation proof of CompCert's backend **(ongoing)**.

Trusted Code Base

- Coq extraction to OCaml.
- The primitive impure implementations correspond to their monadic specifications.
- The call to the generated native code has been specified with a free monad.

References

- Barrière, Aurèle et al. (2021). “Formally verified speculation and deoptimization in a JIT compiler”. In: *Proc. ACM Program. Lang.* POPL.
- Flückiger, Olivier et al. (2018). “Correctness of speculative optimizations with dynamic deoptimization”. In: 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*.
- Letan, Thomas and Yann Régis-Gianas (2020). “FreeSpec: specifying, verifying, and executing impure computations in Coq”. In: *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP*.
- Xia, Li-yao et al. (2020). “Interaction trees: representing recursive and impure programs in Coq”. In: *Proc. ACM Program. Lang.* POPL.
- 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*.