A Verified Information-Flow Architecture

Arthur Azevedo de Amorim, Nathan Collins, André DeHon, Delphine Demange, Cătălin Hriţcu, David Pichardie, Benjamin Pierce, Randy Pollack, Andrew Tolmach

January 2014

What if we could redesign computers for security?

BAE SYSTEMS

Penn
University of Pennsylvania

VE RI TAS
HARVARD

Northeastern

Clean-slate redesign of entire system stack

http://www.crash-safe.org

The SAFE Project

- Support for critical security primitives from hardware to application levels
  - Memory safety
  - Strong dynamic typing
  - Dynamic information flow and access control
- Design of key components informed by verification

SAFE Architecture

- Integrate well-known mechanisms for security (e.g. fat-pointers, type tags, . . .)
- Focus of this paper: flexible mechanism for supporting information-flow control (IFC) using a hardware cache and tags

Our Goal

Formalize and verify the core IFC mechanism proposed by SAFE using the Coq proof assistant

Information-Flow Control (IFC)

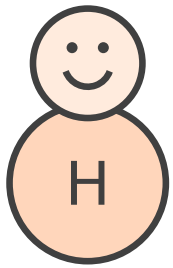
Track and limit information dependency in computations

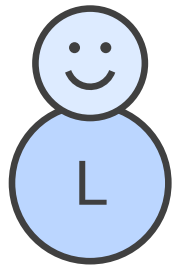
Noninterference (NI): Varying secret inputs does not affect public observations

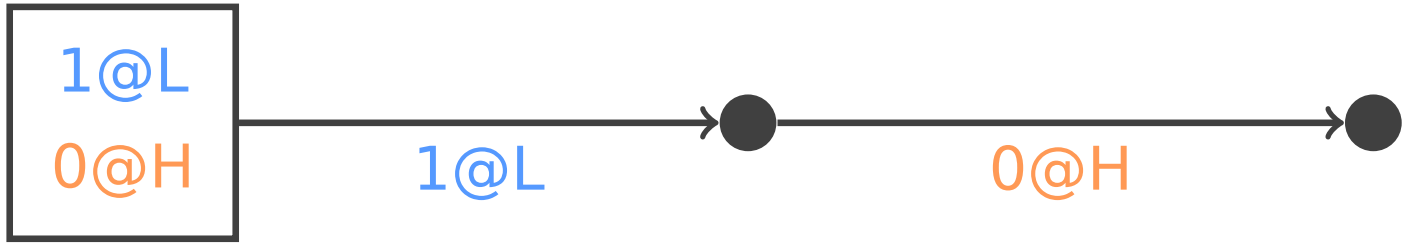

1@L
0@H

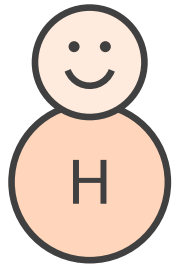
1@L

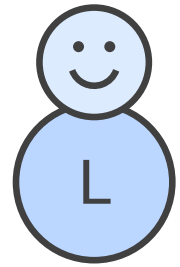
0@H

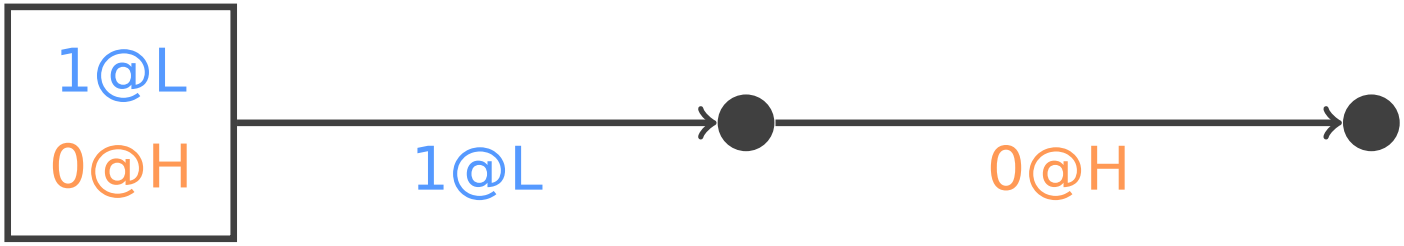


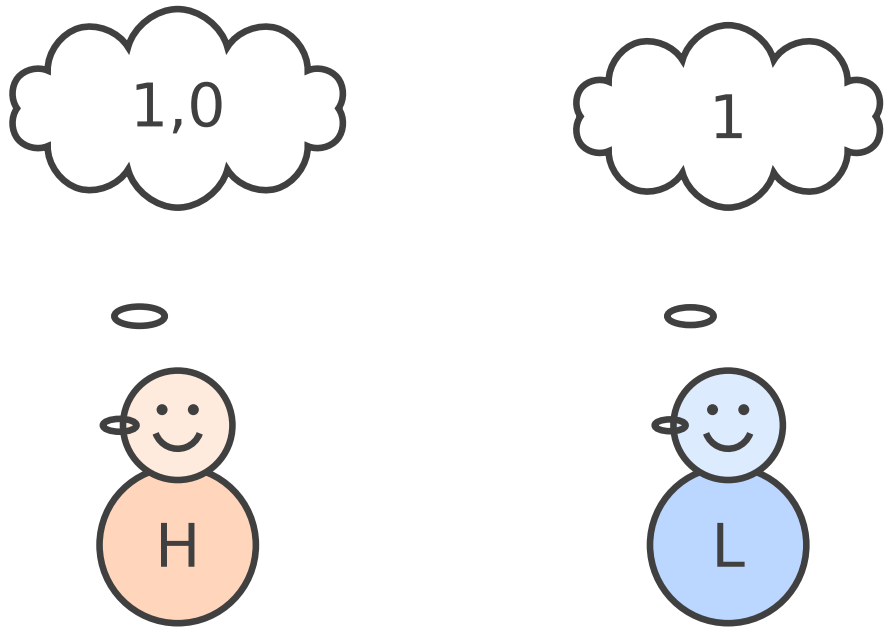


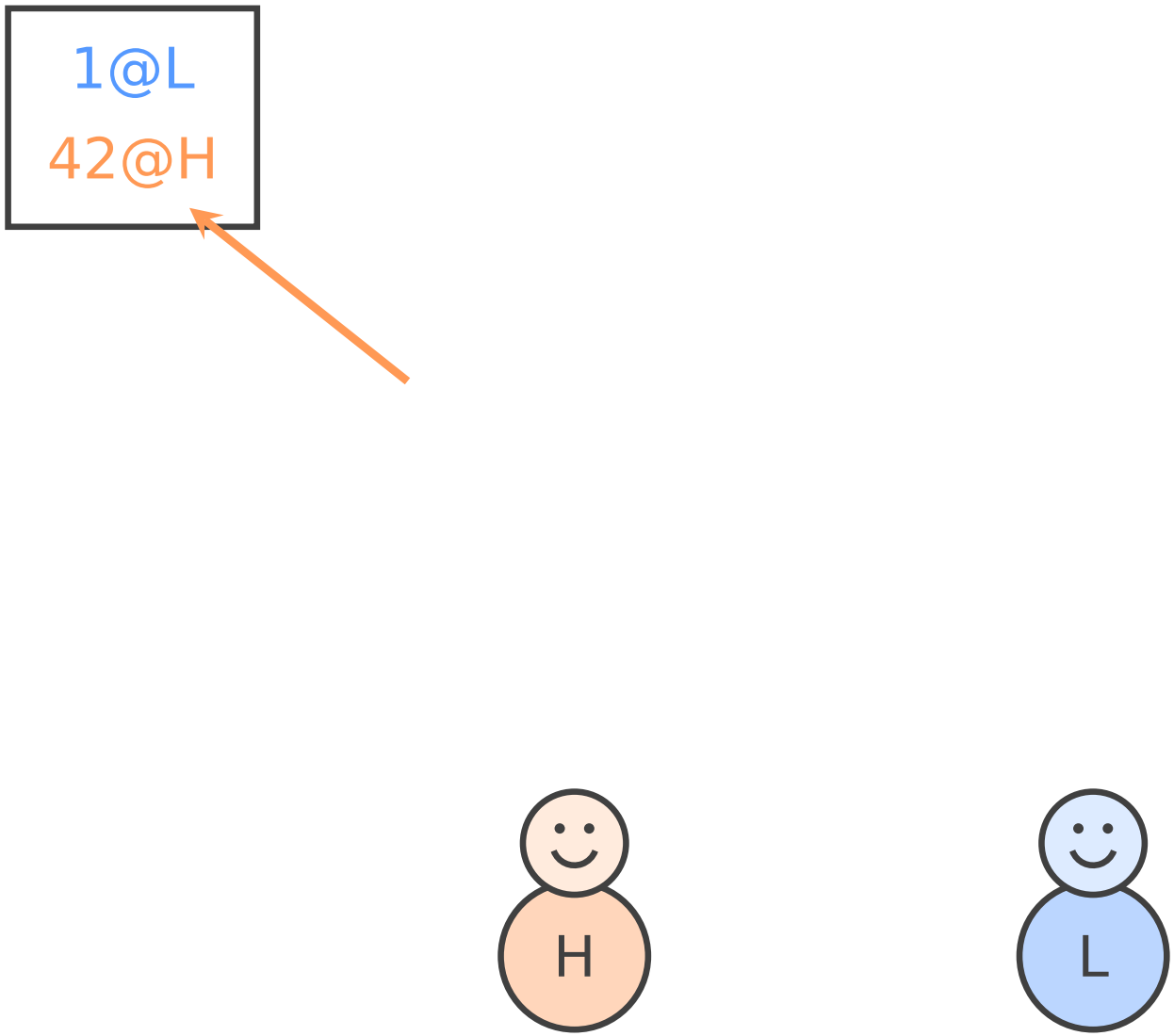

1@L
0@H
1@L
0@H

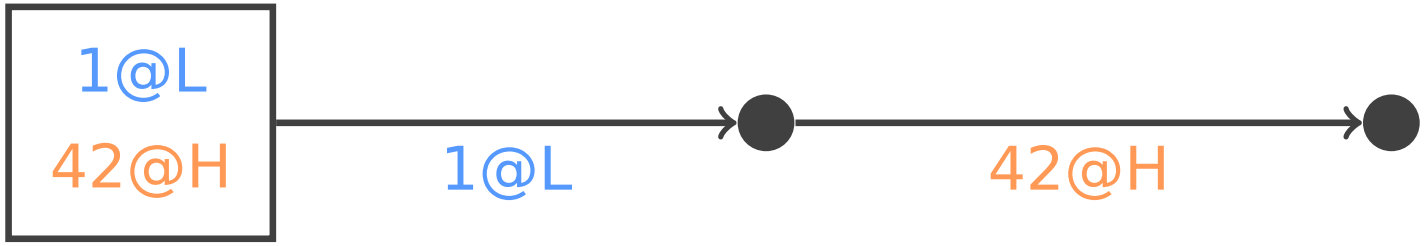

H

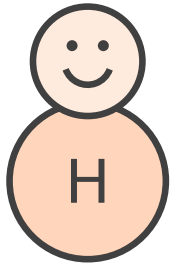

L

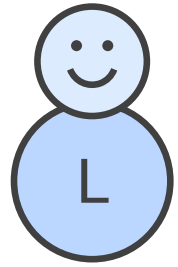

1@L
0@H
1@L
0@H

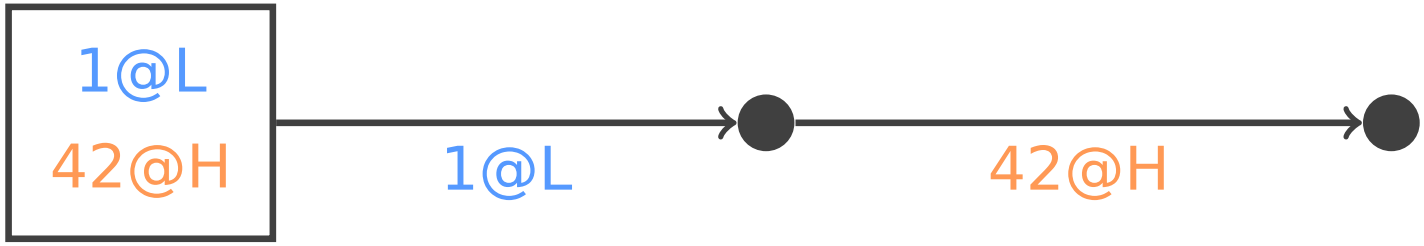

1,0
1
H
L

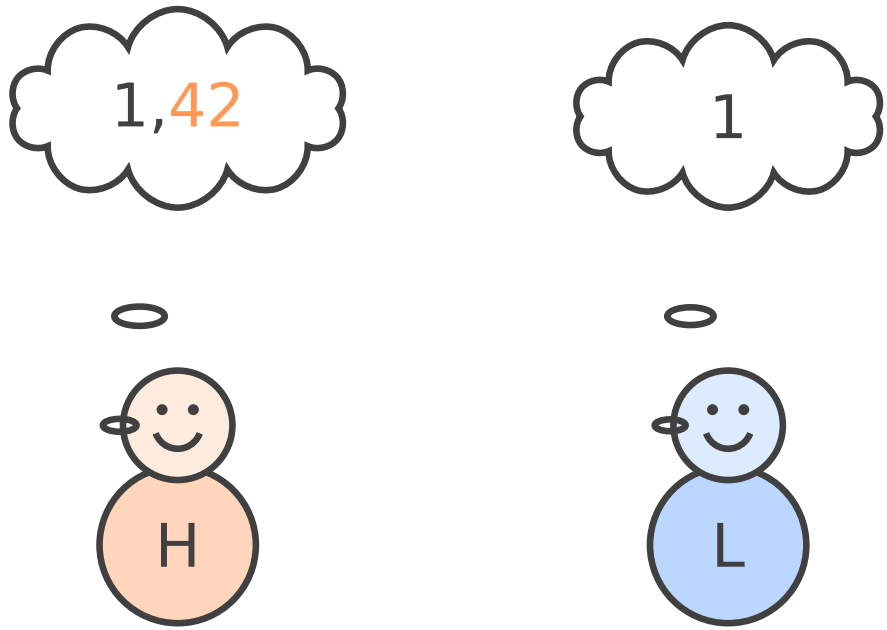

1@L
42@H
H
L

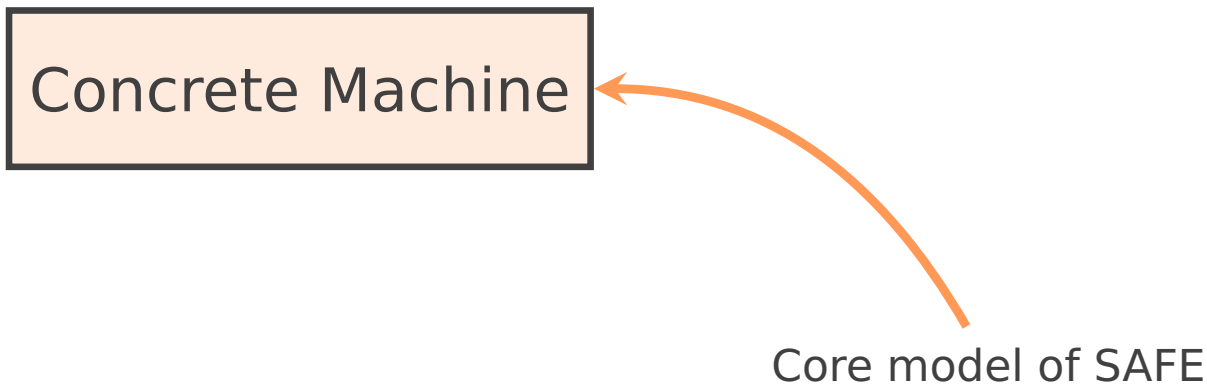

1@L
42@H
1@L
42@H

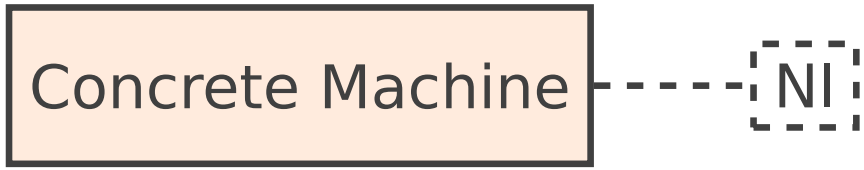

H

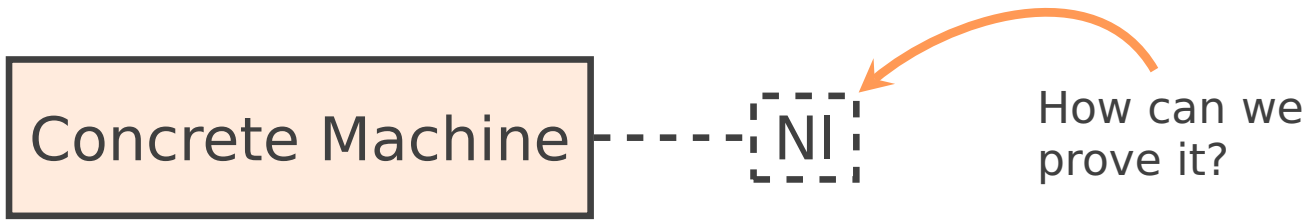

L

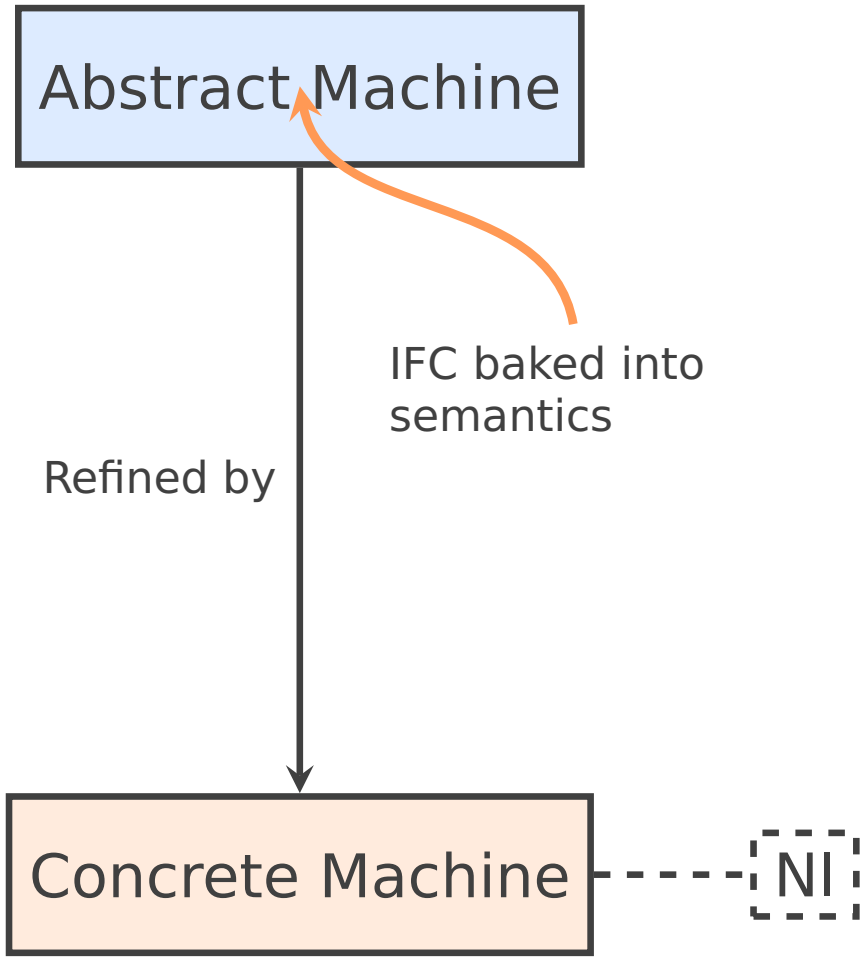

1@L
42@H
1@L
42@H

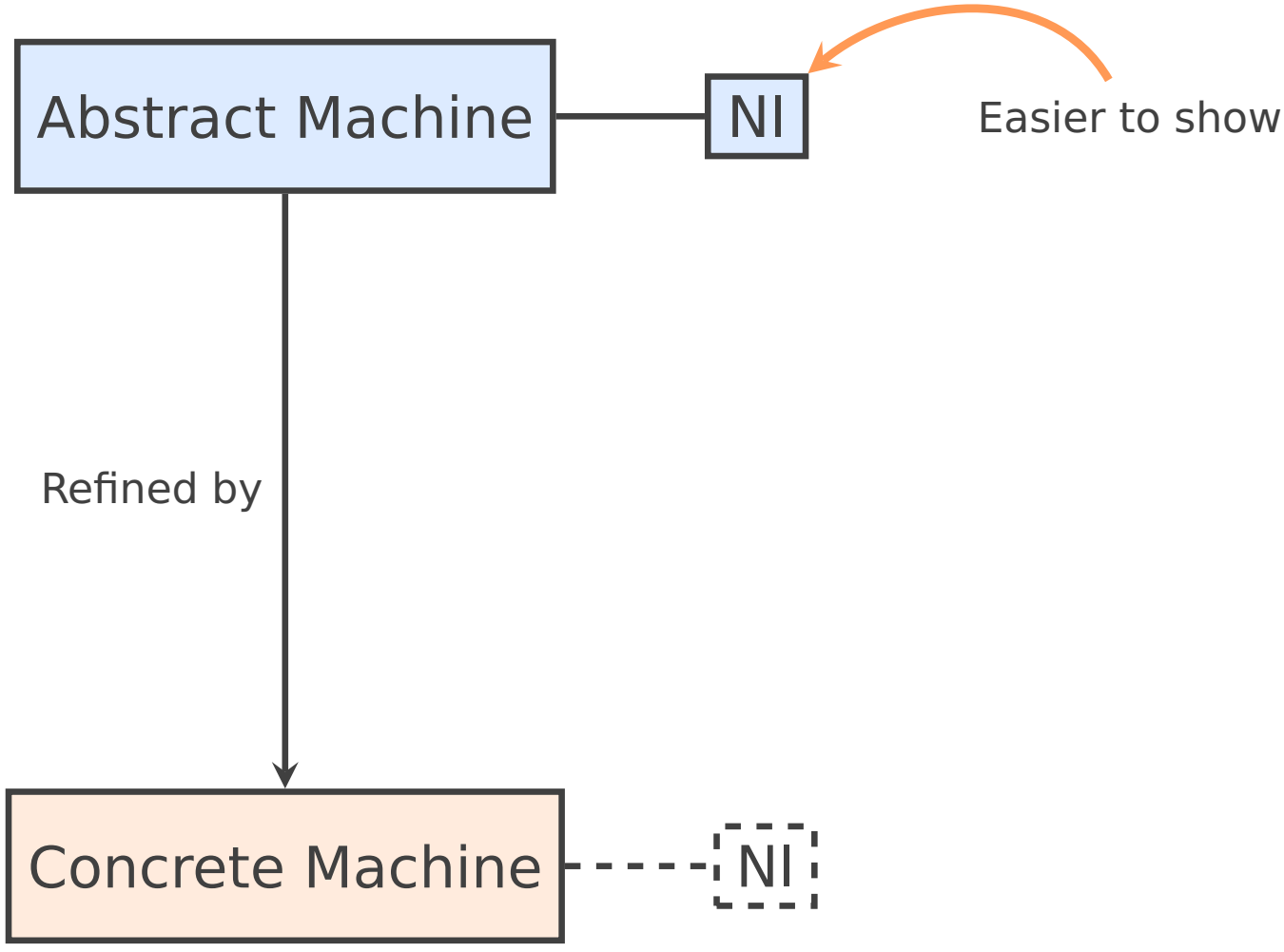

1,42
1
H
L

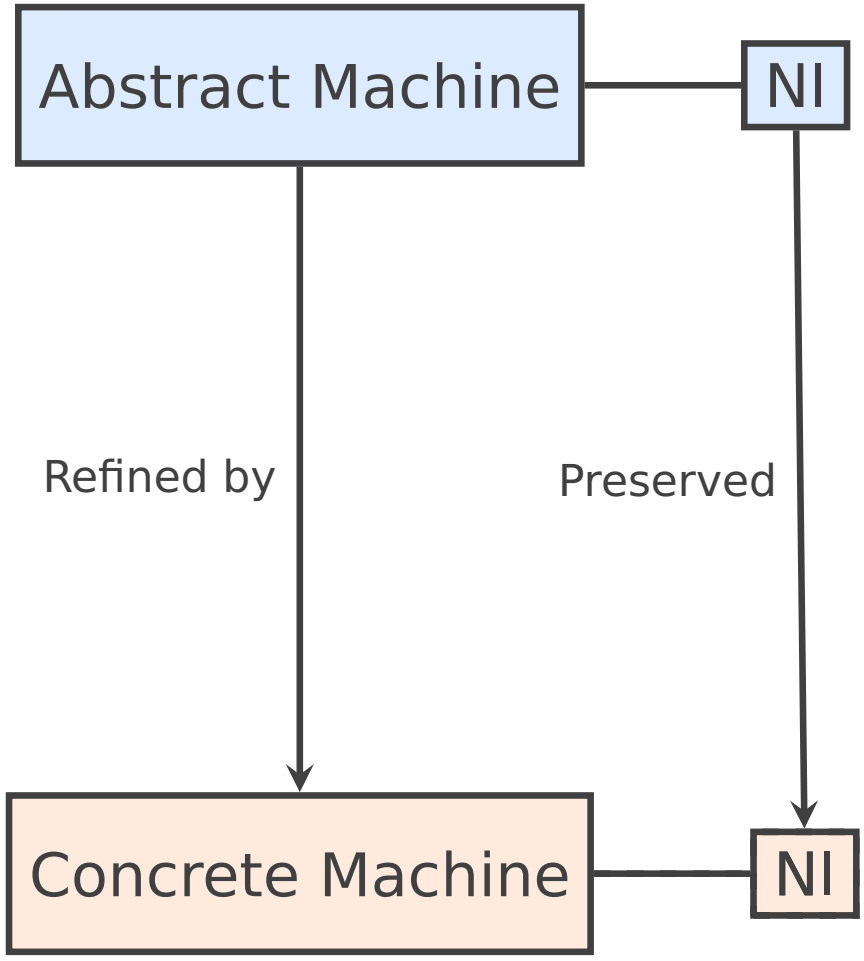

Concrete Machine
Core model of SAFE

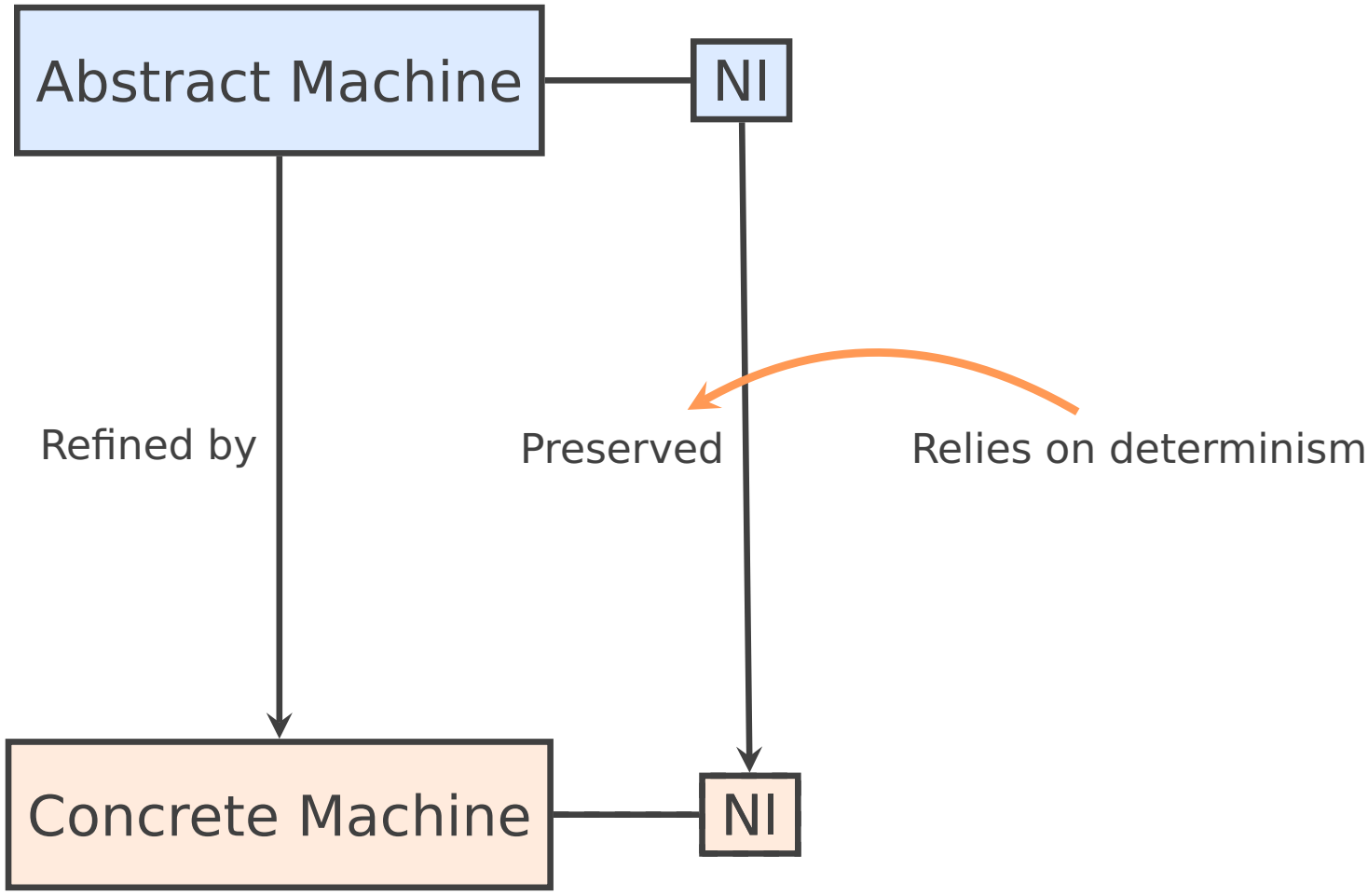

Concrete Machine
NI


Concrete Machine
NI
How can we
prove it?


Abstract Machine
IFC baked into
semantics
Refined by
Concrete Machine
NI


Abstract Machine
NI
Easier to show
Refined by
Concrete Machine
NI


Abstract Machine
NI
Refined by
Preserved
Concrete Machine
NI

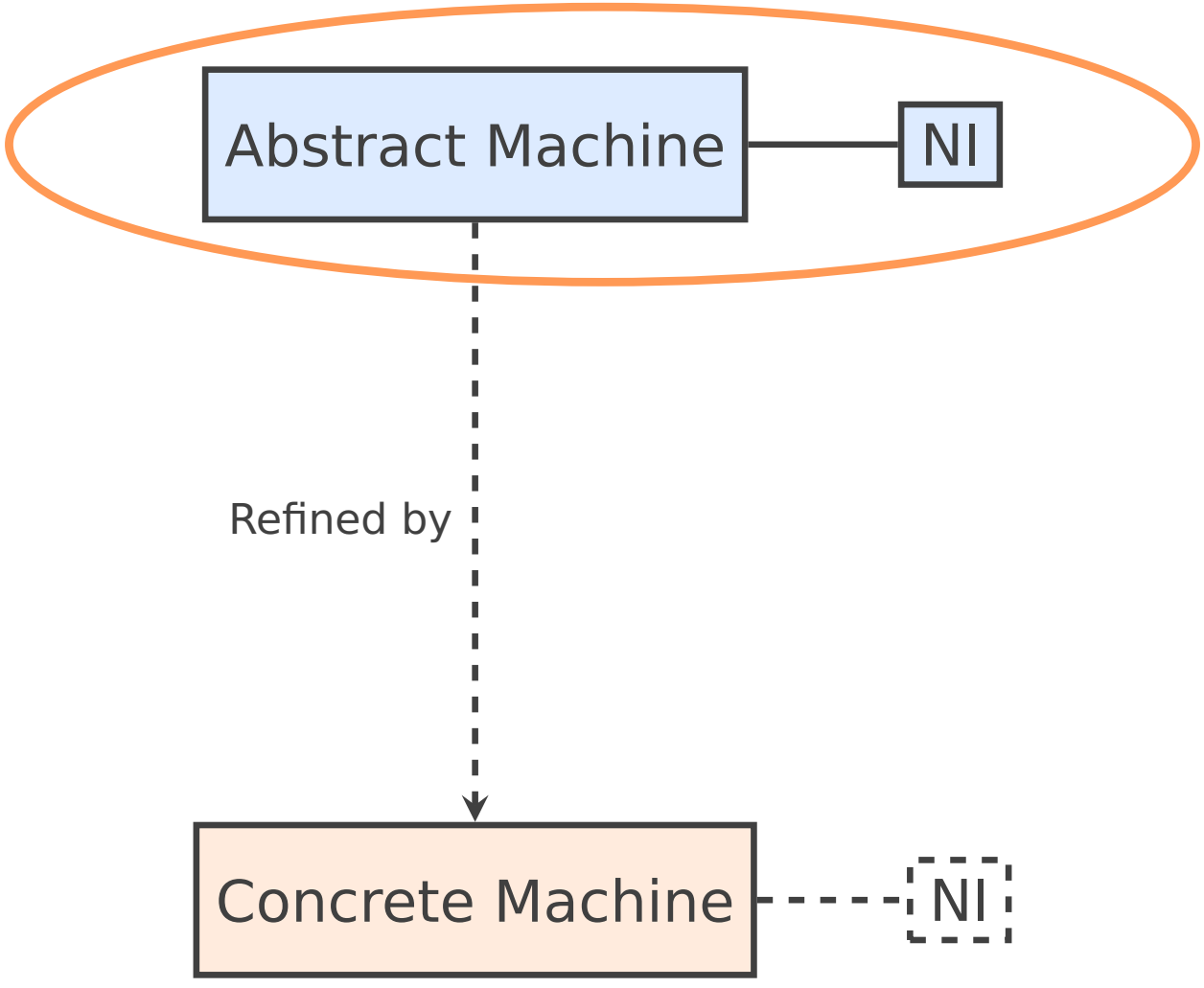

Abstract Machine
NI
Refined by
Preserved
Relies on determinism
Concrete Machine
NI

Our Contributions

Using Coq, we

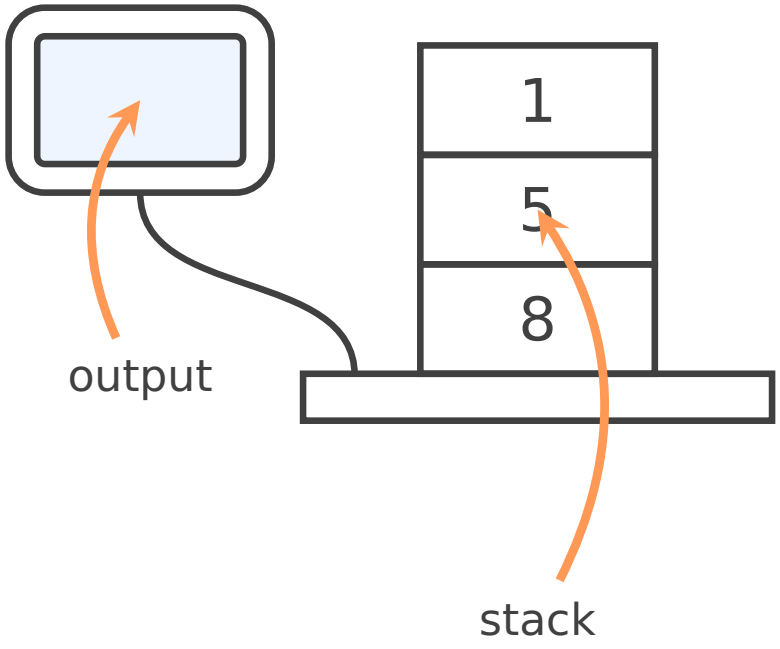
- Model the core mechanism for supporting IFC in SAFE (our so-called “concrete machine”)
- Develop a proof methodology (by refinement) for proving this mechanism correct

Information-Flow Model


Abstract Machine
NI
Refined by
Concrete Machine
NI

Simple Abstract Machine

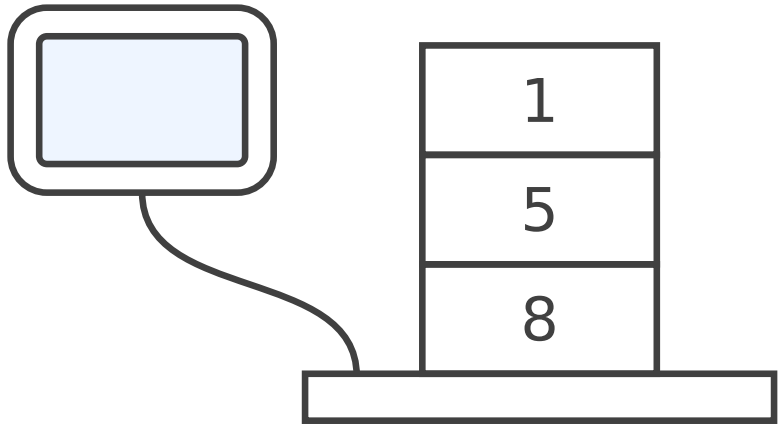
Stack machine with output channel, operating on integers



Simple Abstract Machine

Input/output model:

- The input of a program is its initial stack
- The result of executing a program is the sequence of its outputs

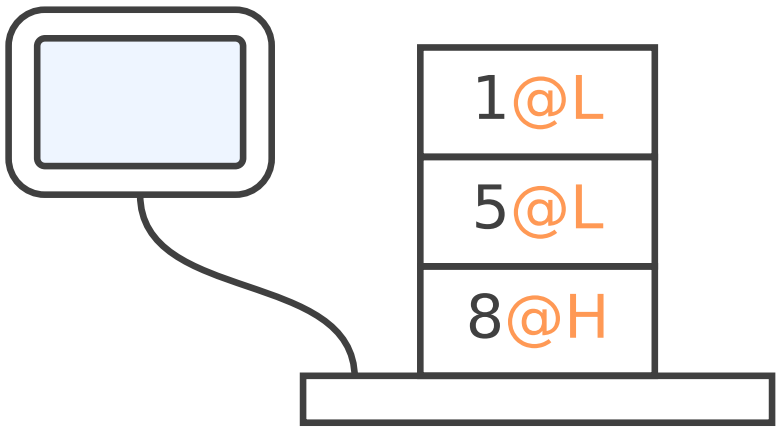


IFC Abstract Machine

Label data with security levels

H = high security
L = low security

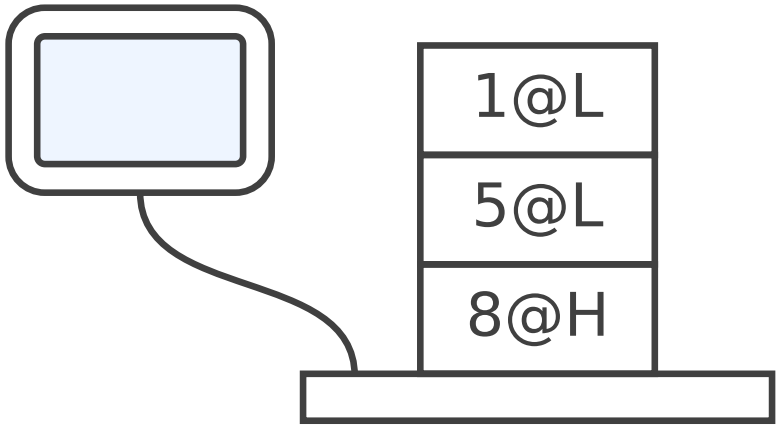
(or, more generally, any IFC lattice)



IFC Abstract Machine

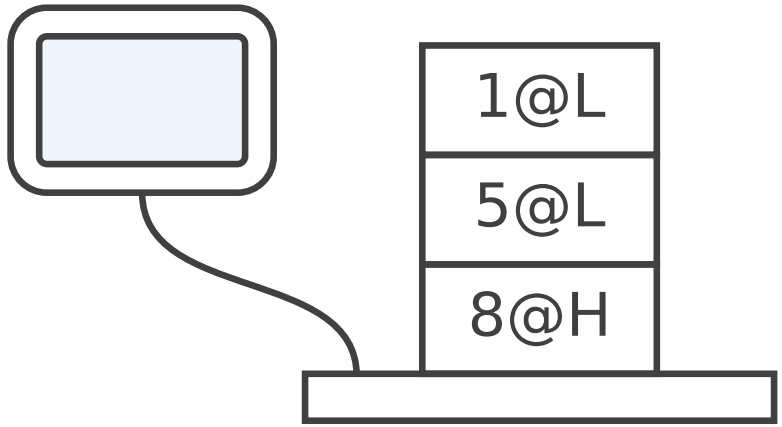
Machine has semantics
with standard dynamic IFC
baked-in

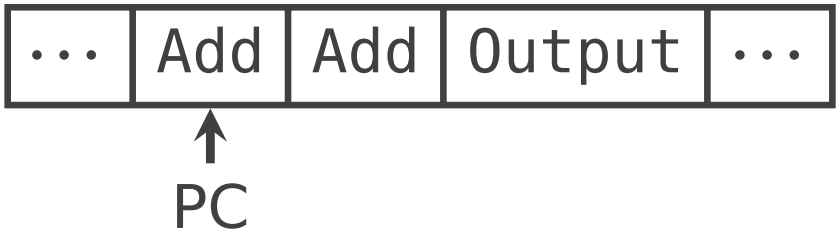
Mechanizing proof of NI is
relatively straightforward



IFC Abstract Machine

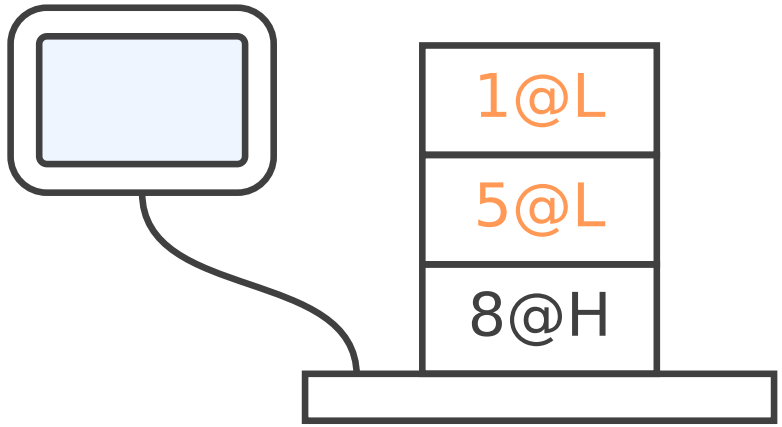
When low-security operands are combined, the result is low-security

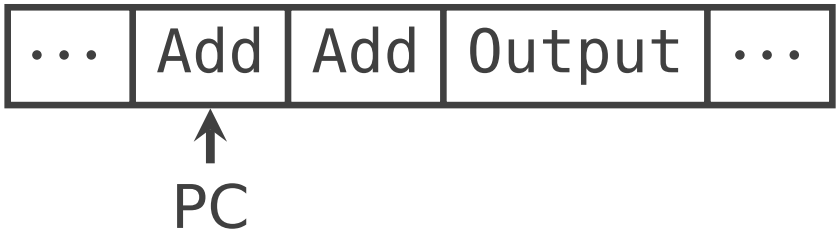




IFC Abstract Machine

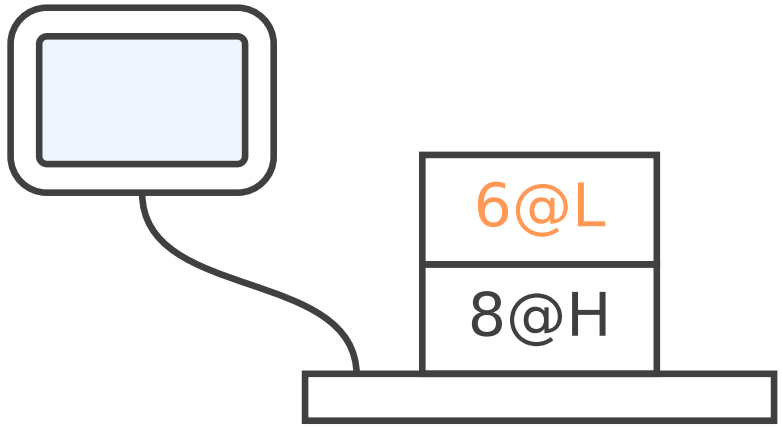
When low-security operands are combined, the result is low-security

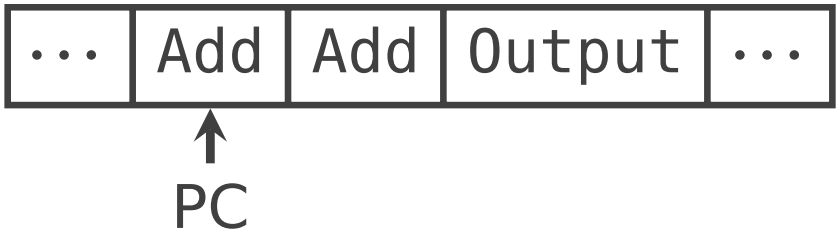




IFC Abstract Machine

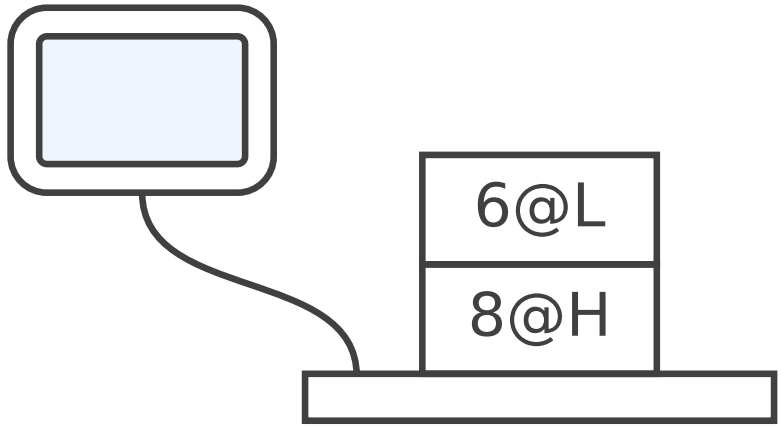
When low-security operands are combined, the result is low-security

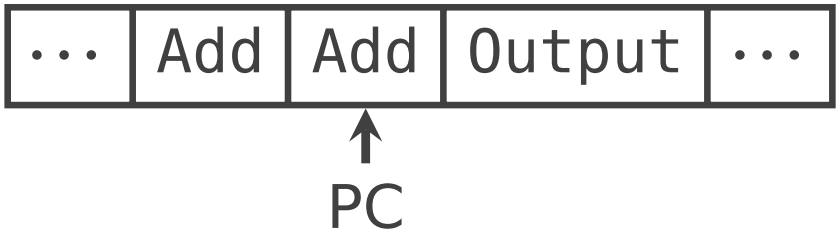




IFC Abstract Machine

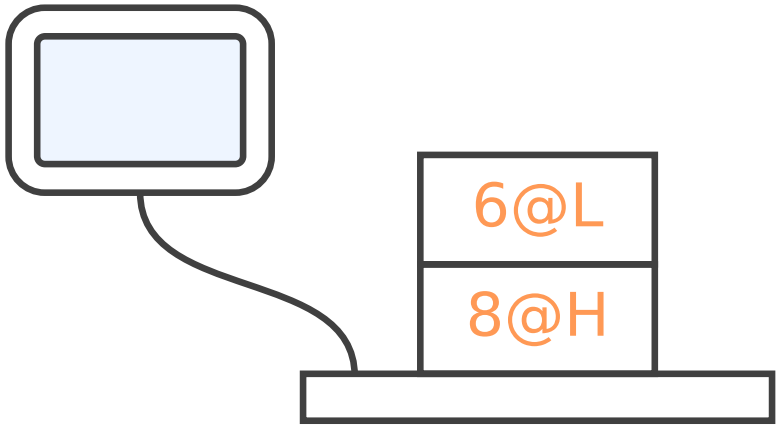
When one operand is high-security, result is high-security

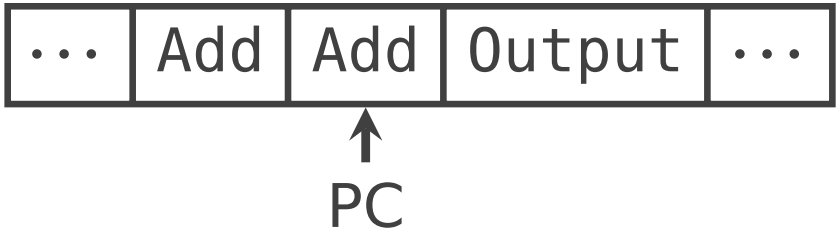




IFC Abstract Machine

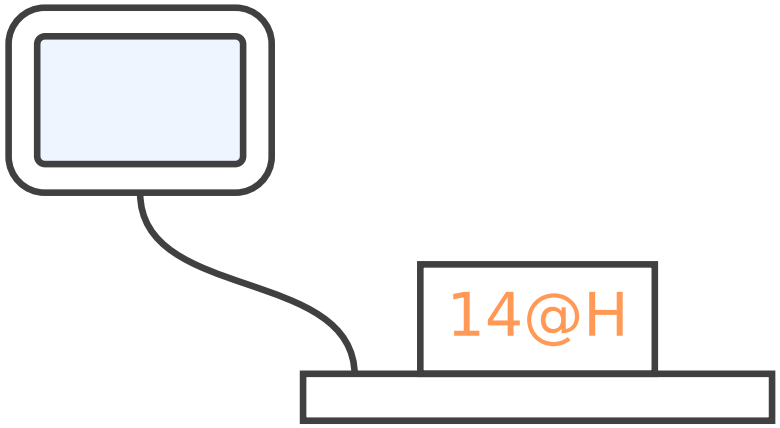
When one operand is high-security, result is high-security

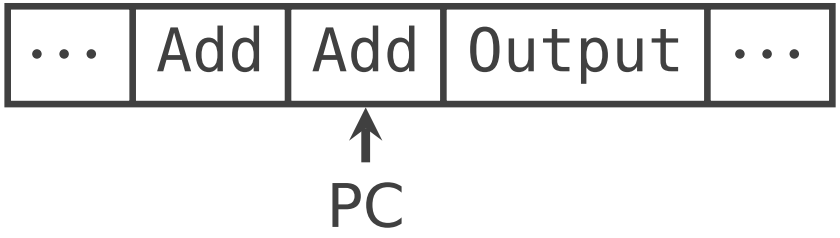




IFC Abstract Machine

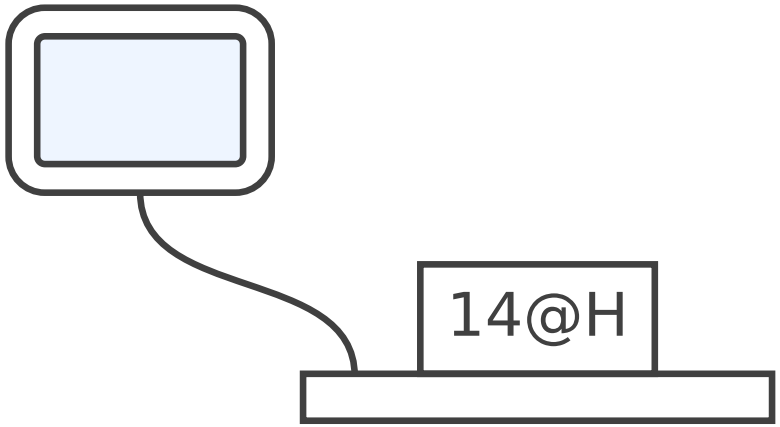
When one operand is high-security, result is high-security

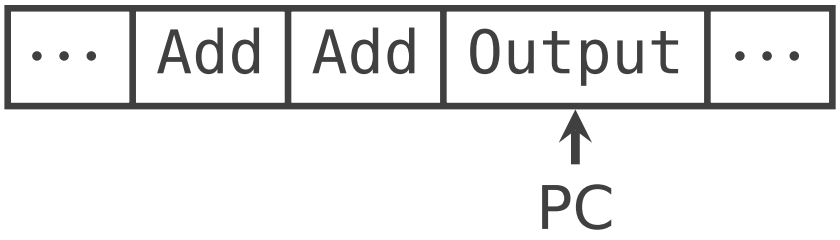




IFC Abstract Machine

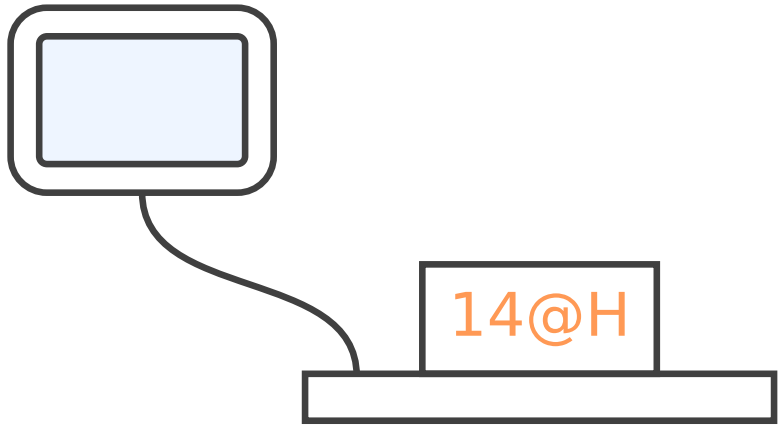
Labels on outputs mark
who is able to see them

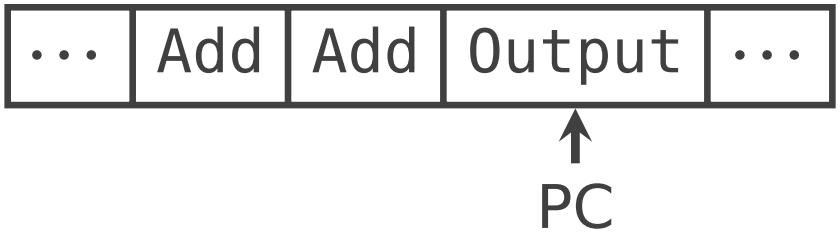




IFC Abstract Machine

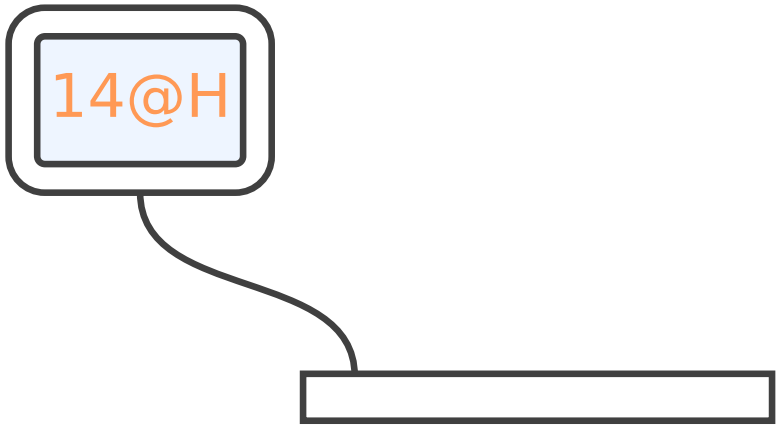
Labels on outputs mark
who is able to see them

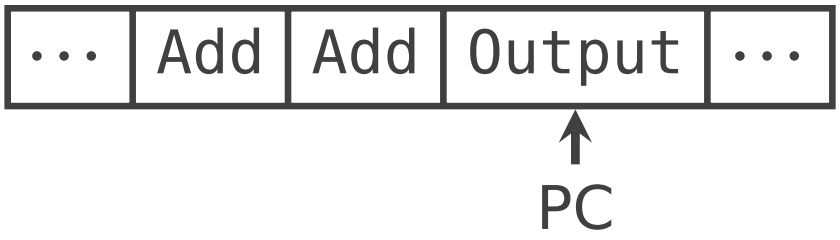




IFC Abstract Machine

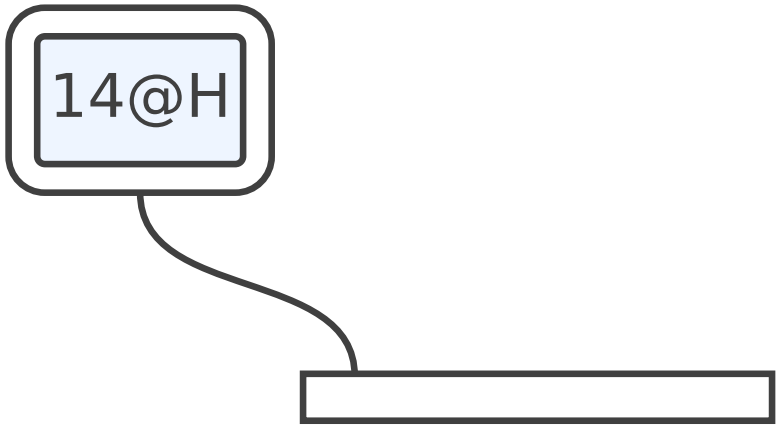
Labels on outputs mark
who is able to see them

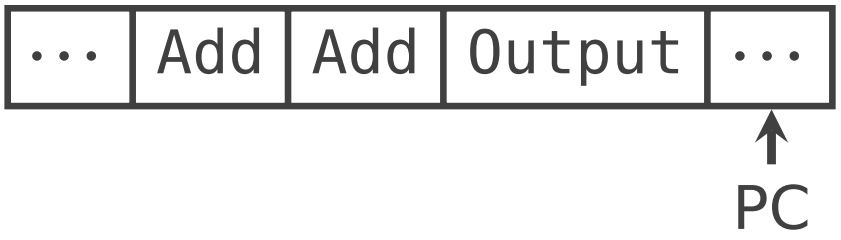




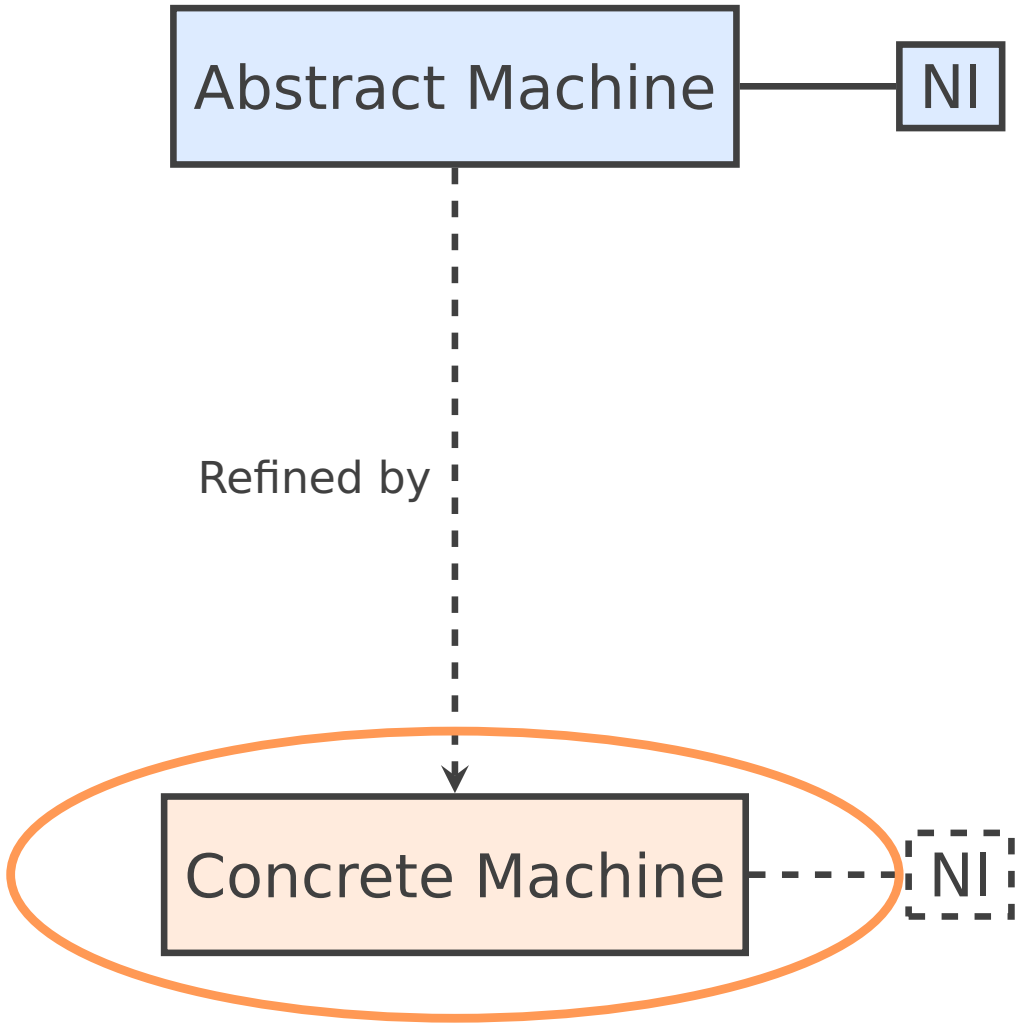
IFC Abstract Machine

Labels on outputs mark who is able to see them



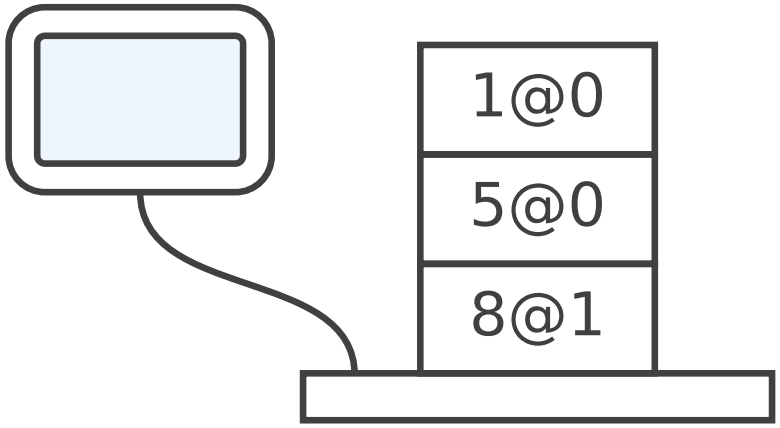


IFC in Hardware


Abstract Machine
NI
Refined by
Concrete Machine
NI

Concrete Machine

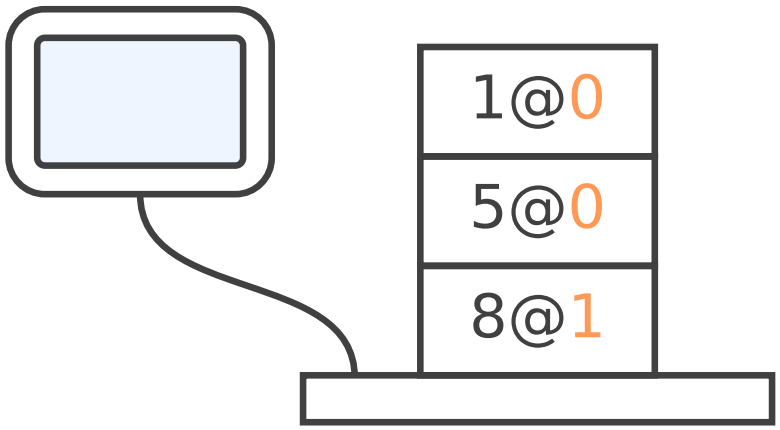
Similar to previous one,
but with hardware
mechanisms for supporting
IFC



Concrete Machine

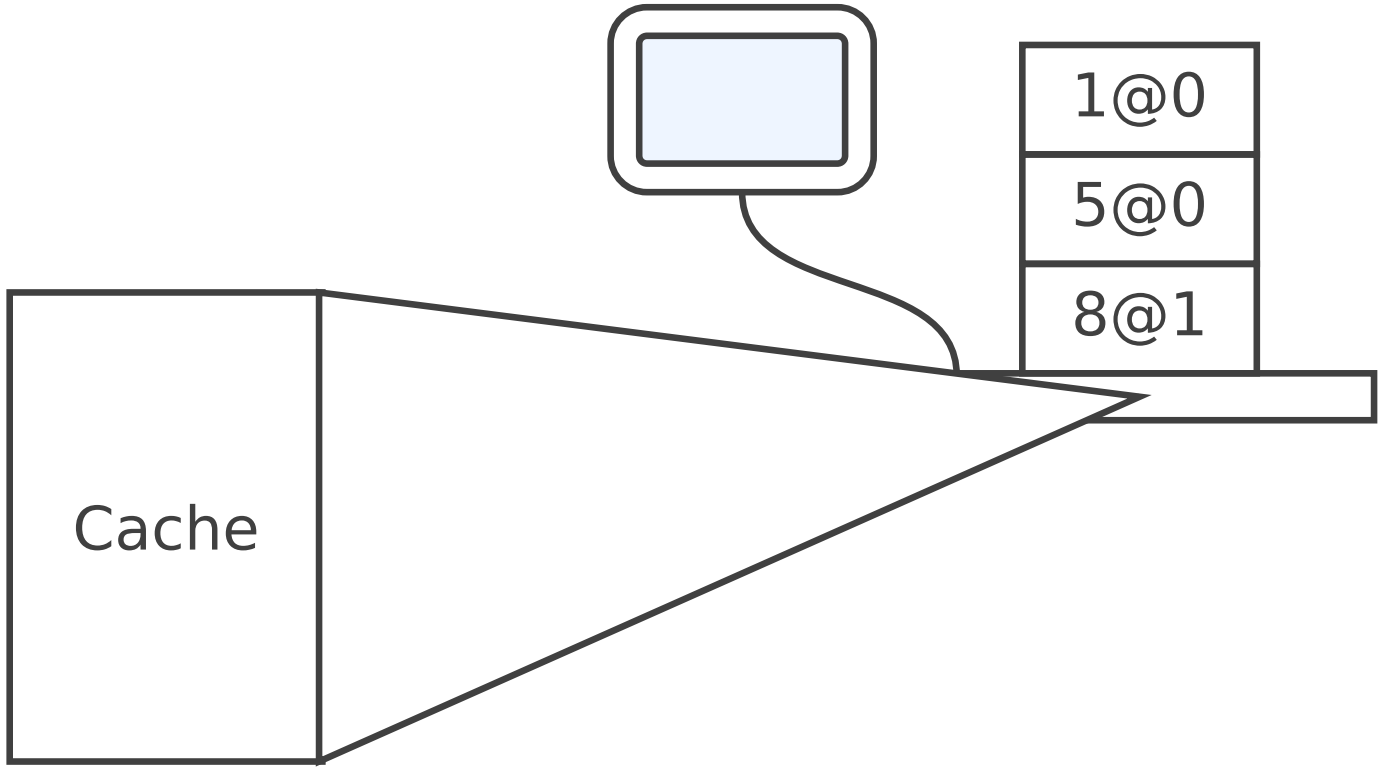
Plain integer tags instead of high-level IFC labels

Uninterpreted in hardware



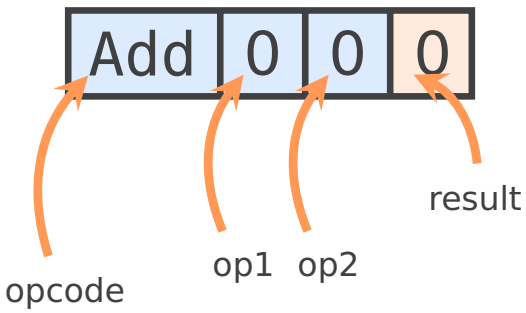
Concrete Machine

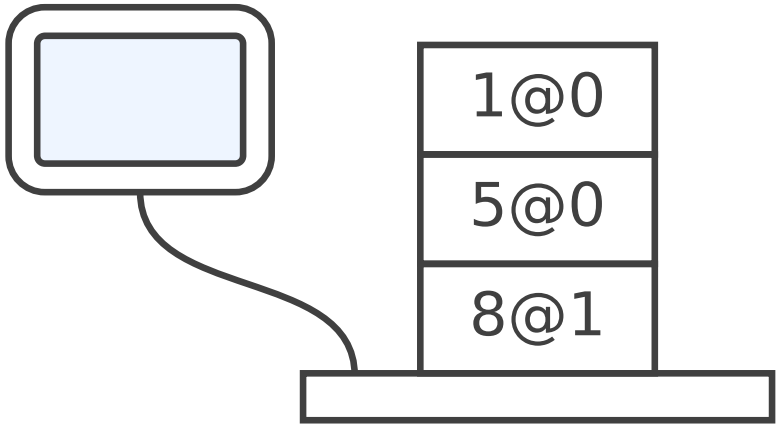
Hardware cache governs
tag propagation



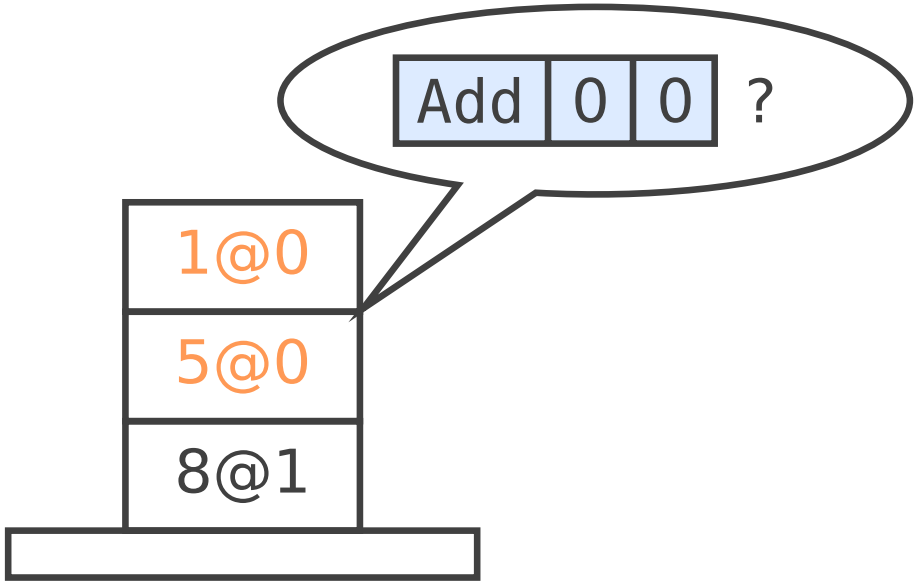
Concrete Machine

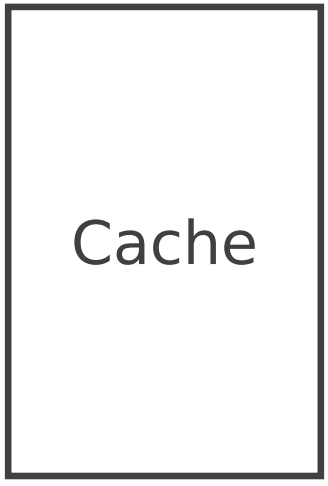
Cache line relates combination of instruction and operand tags to result tag



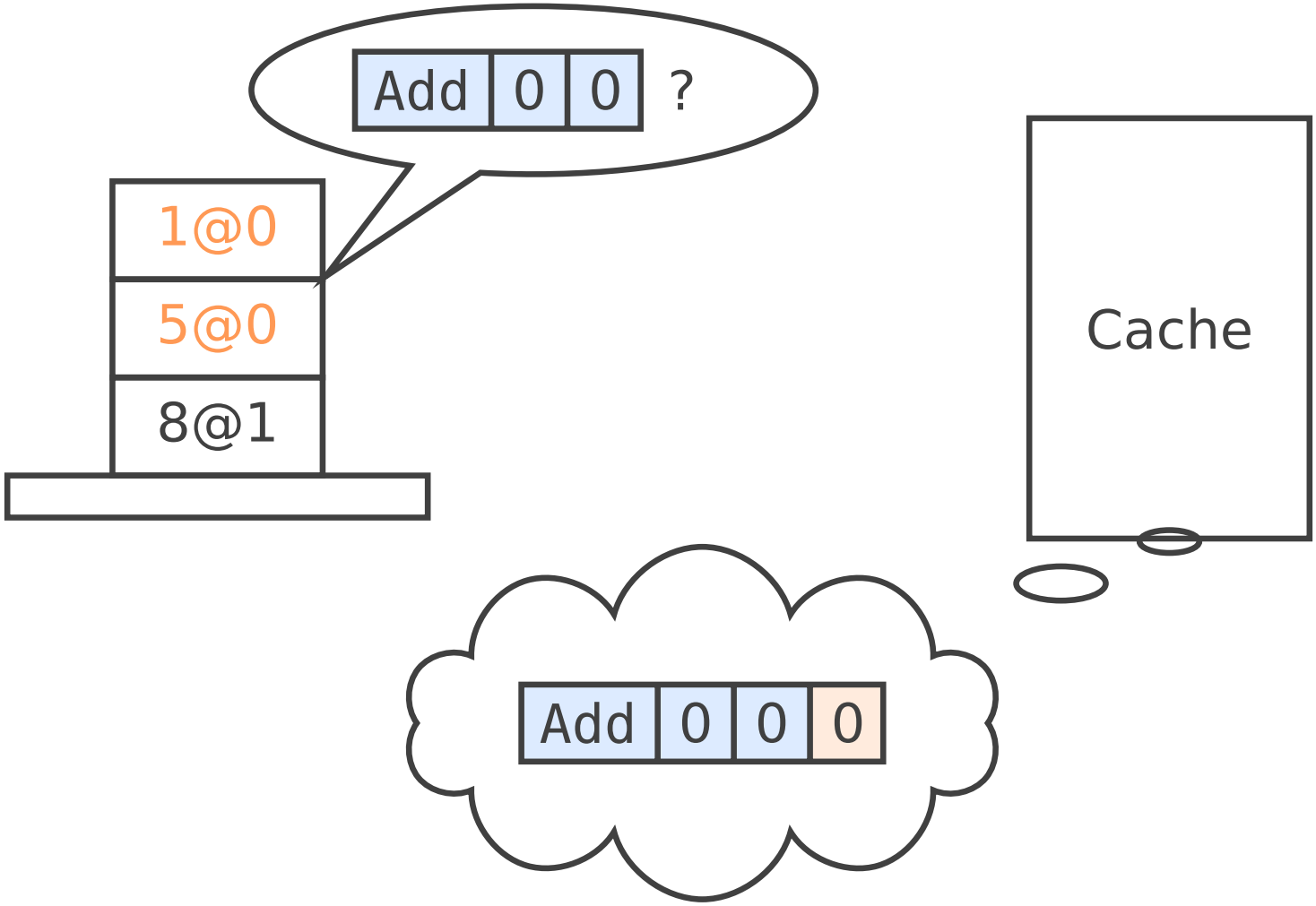


Cache Operation

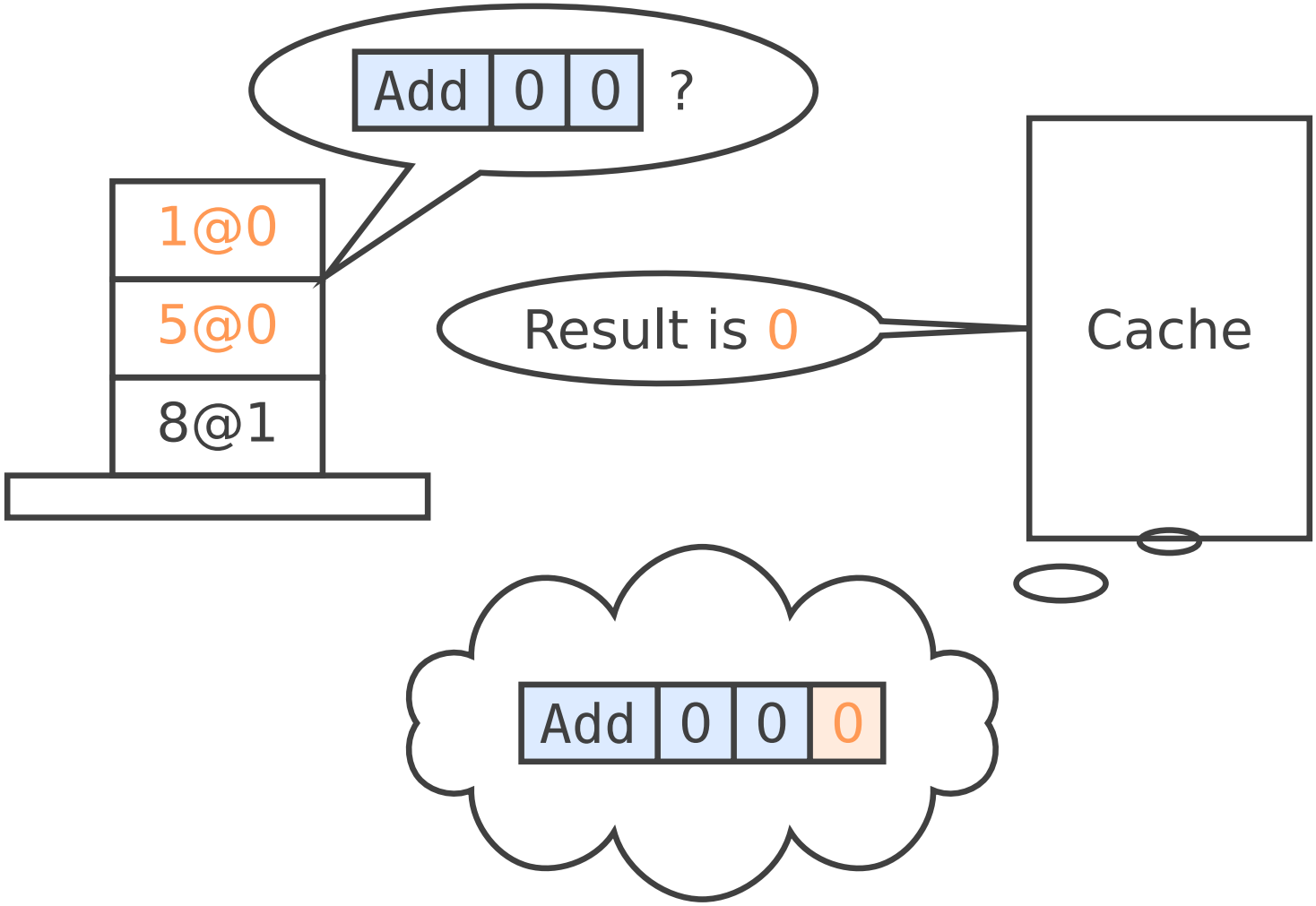




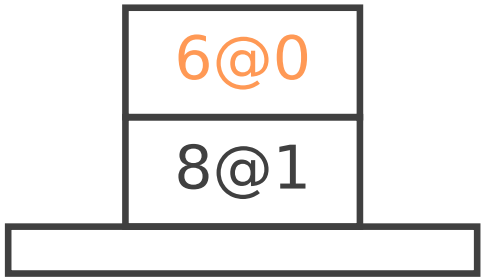
Cache Operation

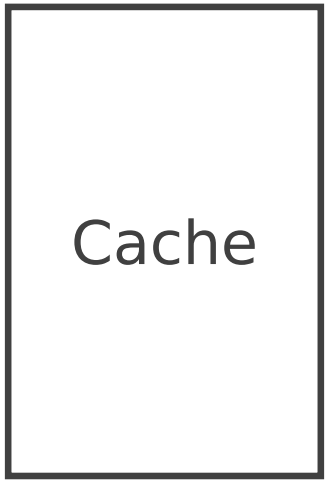


Cache Operation

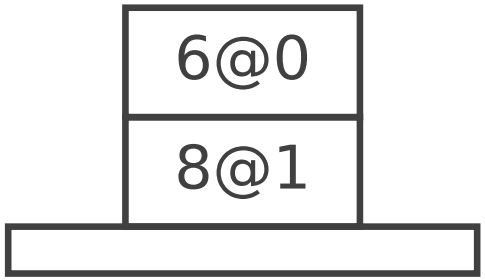


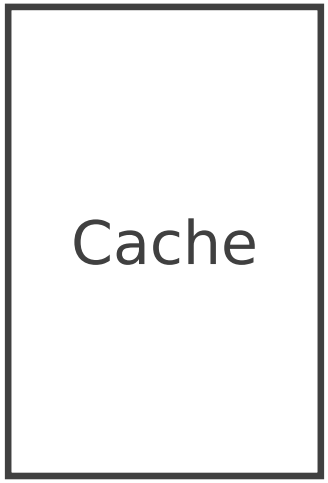
Cache Operation



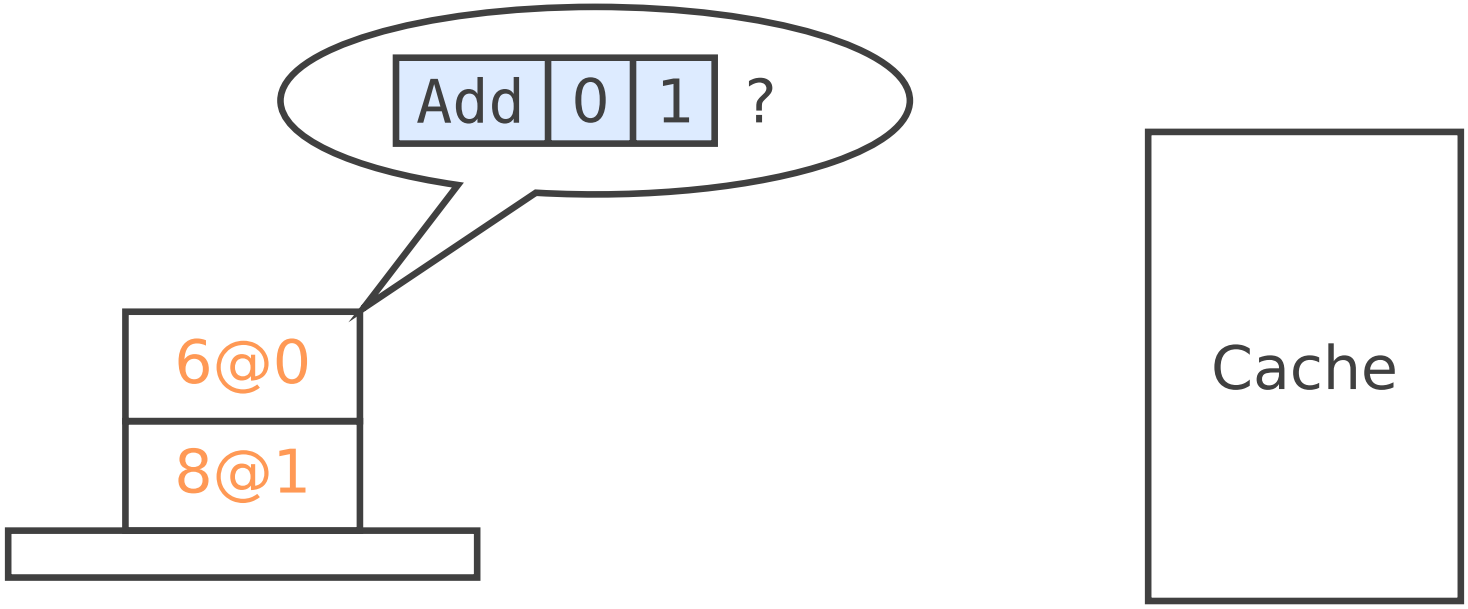


Cache Operation

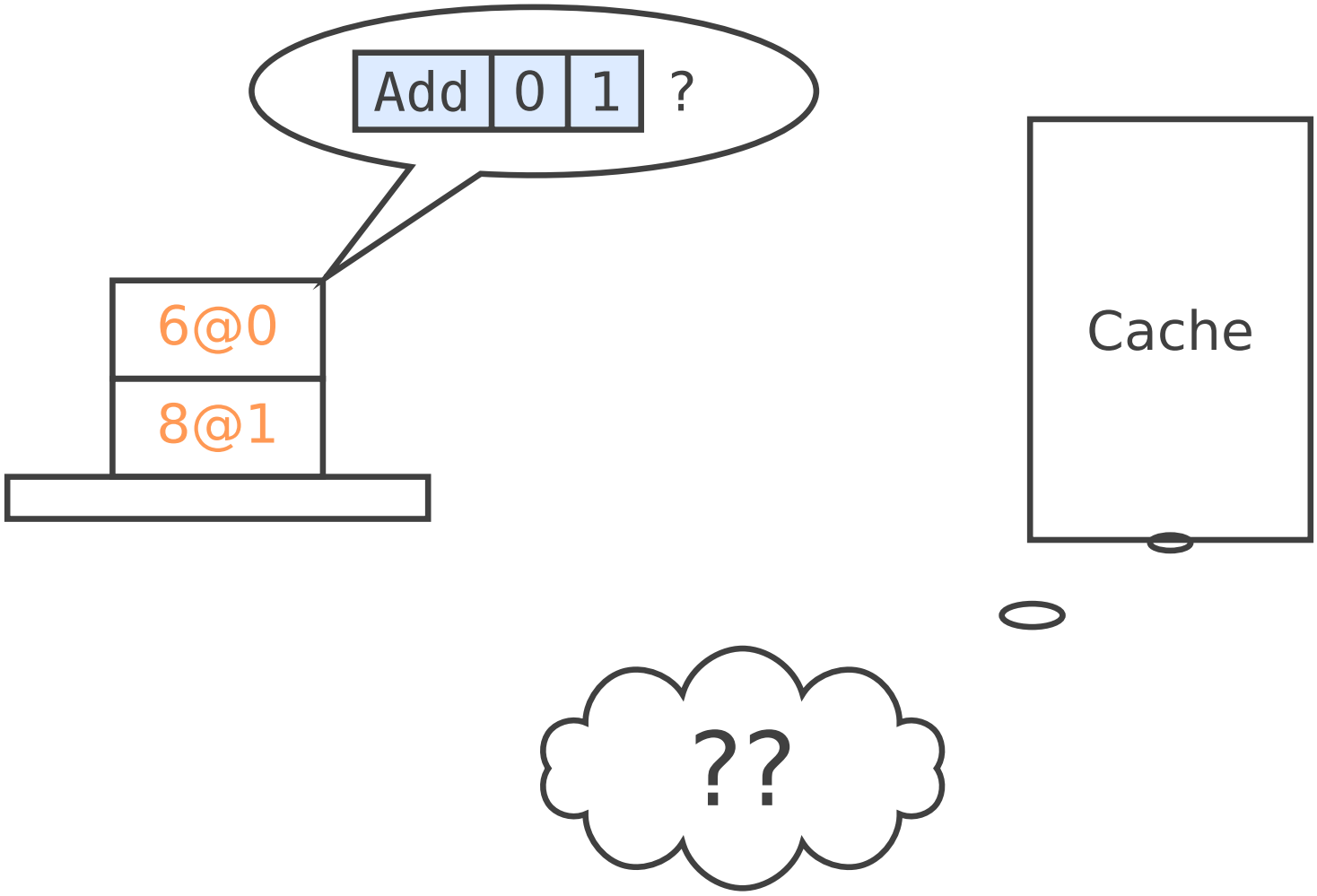




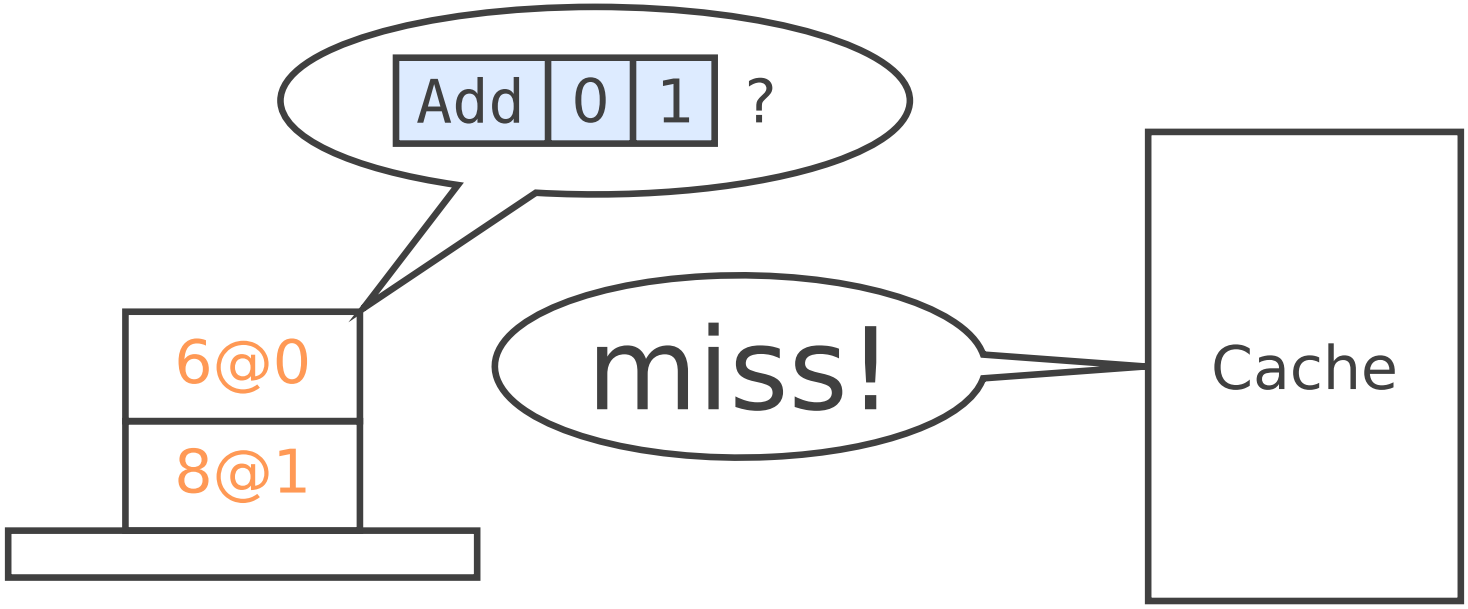
Cache Operation



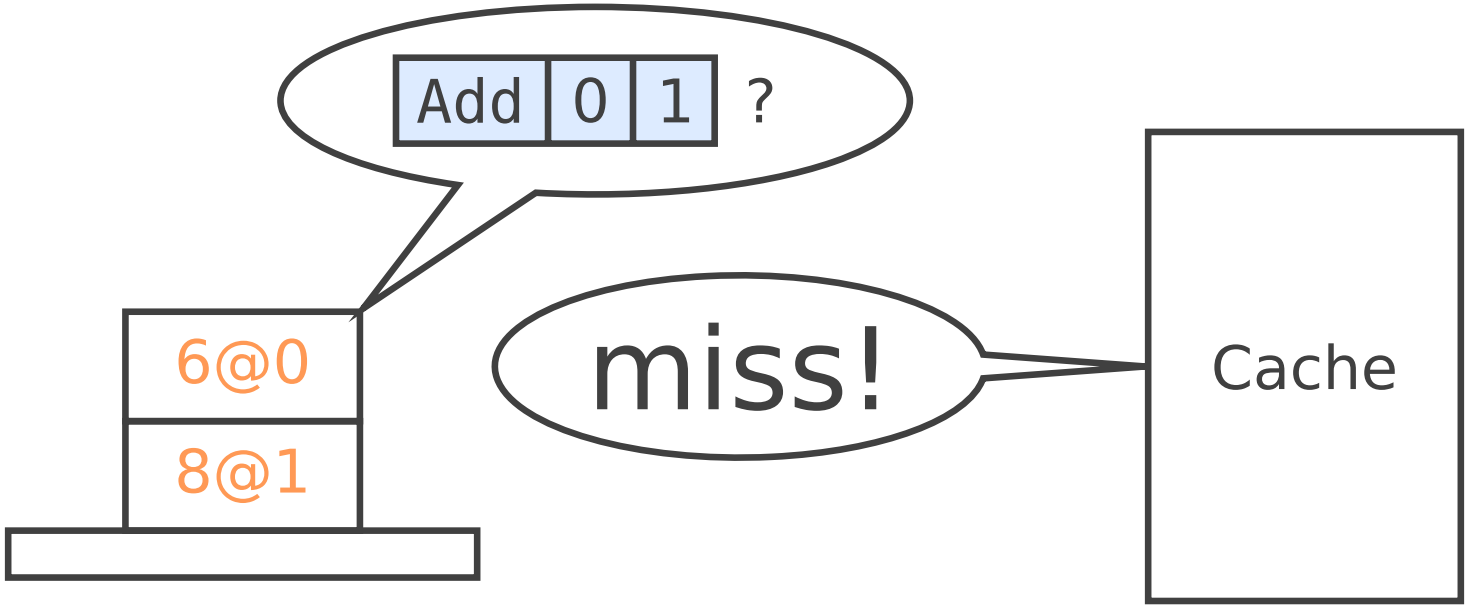
Cache Operation



Cache Operation



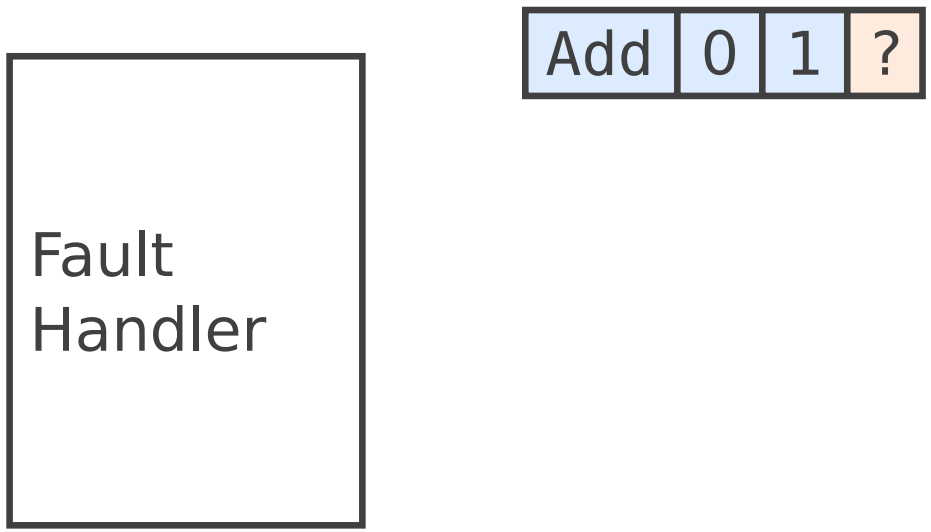
Cache Operation



Control is transferred to fault handler

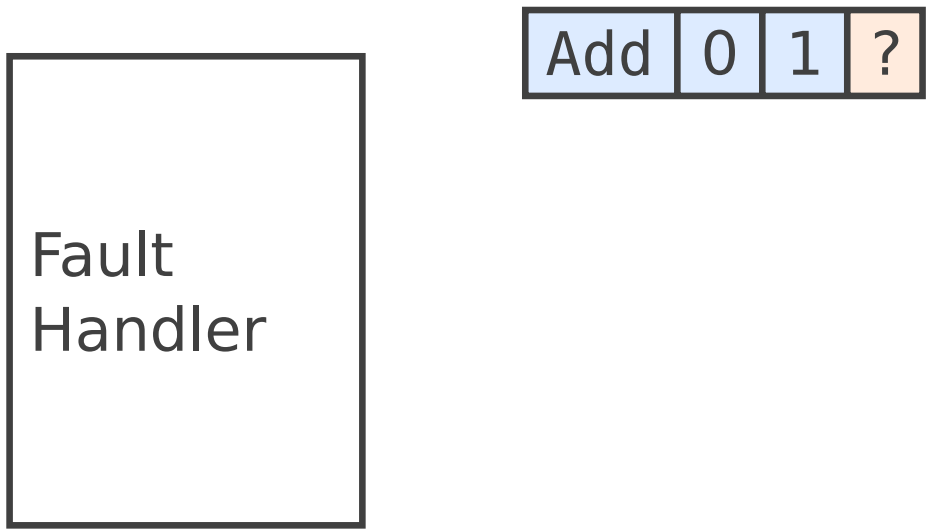
Fault-Handler Operation

Handler is a piece of machine code running in privileged mode



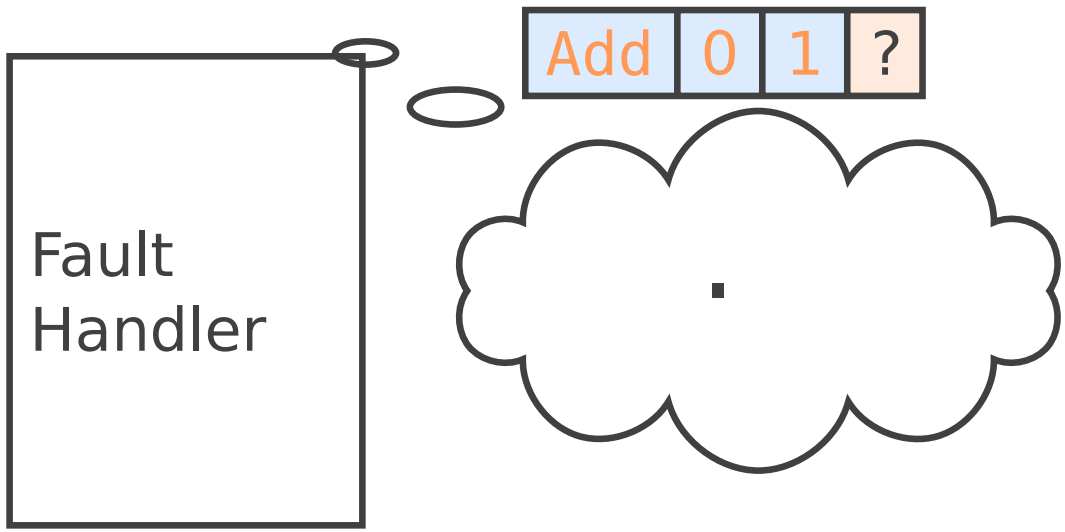
Fault-Handler Operation

It can modify the cache, as well as bypass it



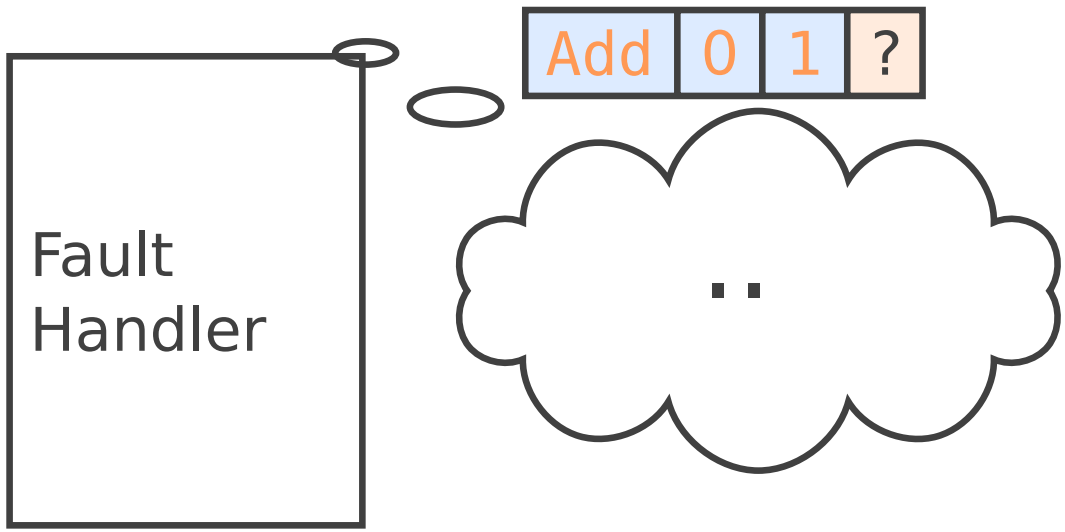
Fault-Handler Operation

It analyzes the fault context



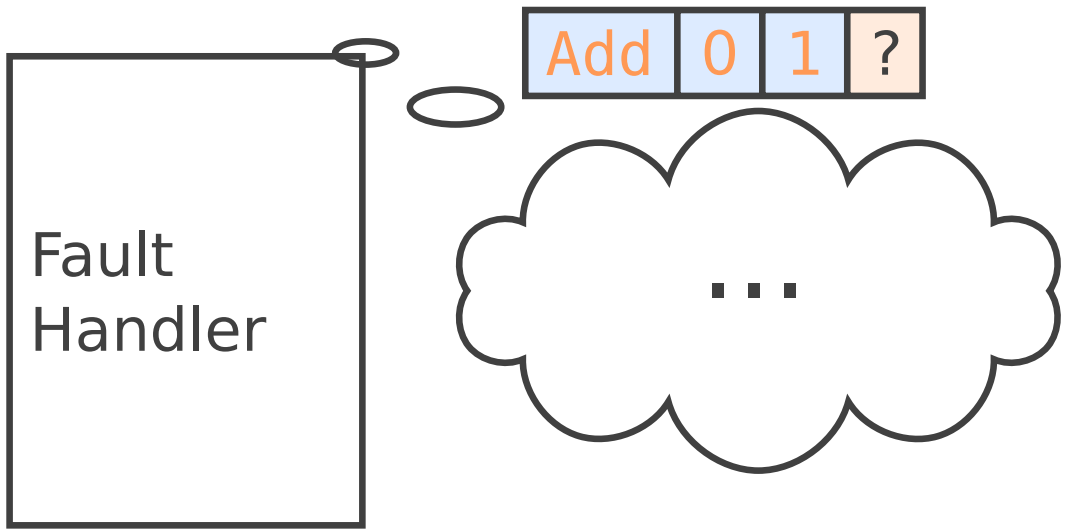
Fault-Handler Operation

It analyzes the fault context



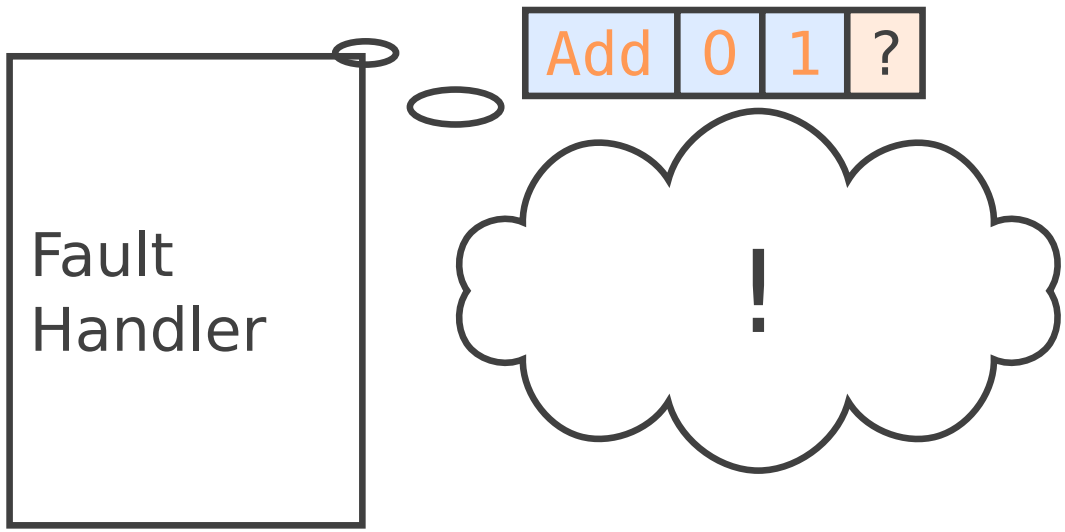
Fault-Handler Operation

It analyzes the fault context



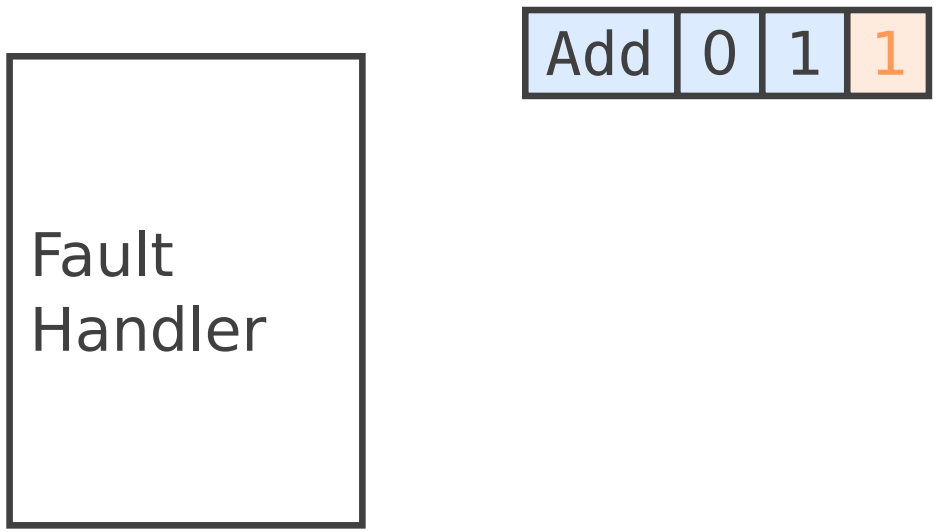
Fault-Handler Operation

It analyzes the fault context



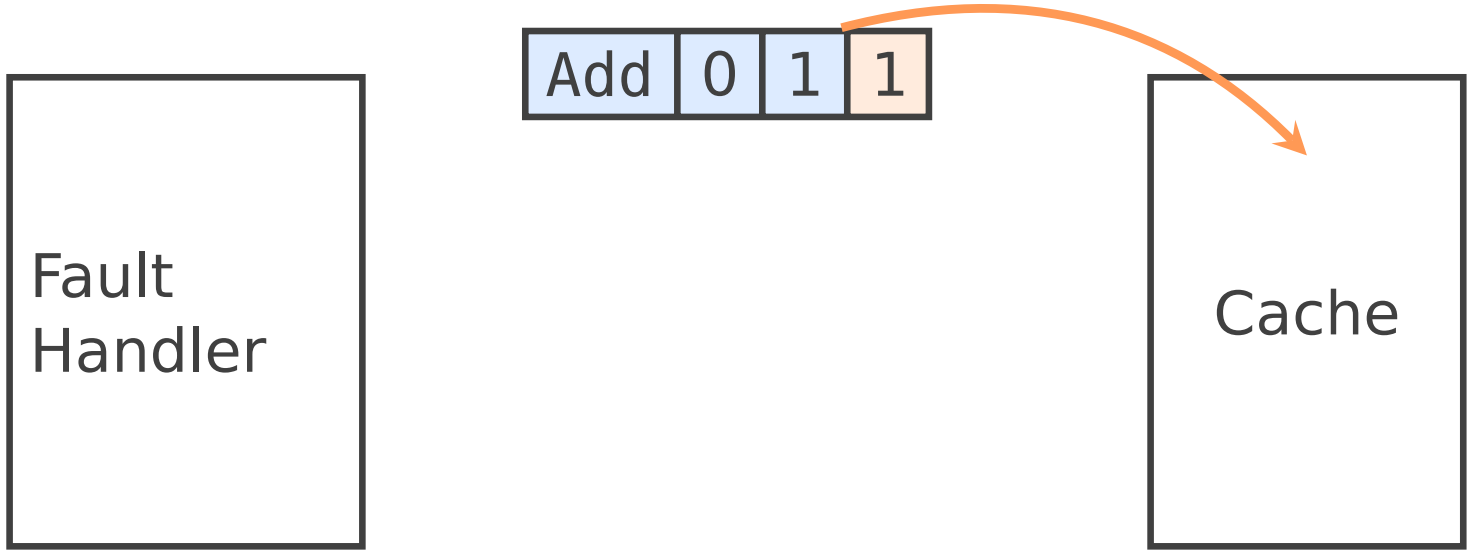
Fault-Handler Operation

. . . and computes corresponding result tag



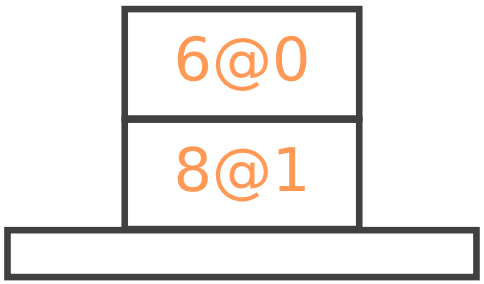
Fault-Handler Operation

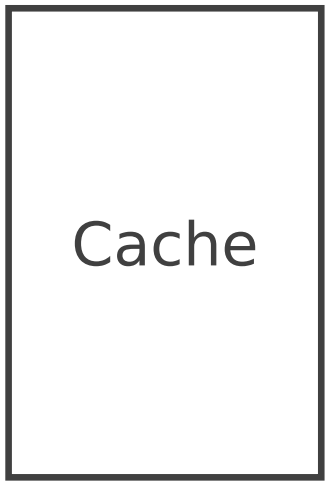
The handler then installs that line in the cache, returning to faulting instruction



Fault-Handler Operation

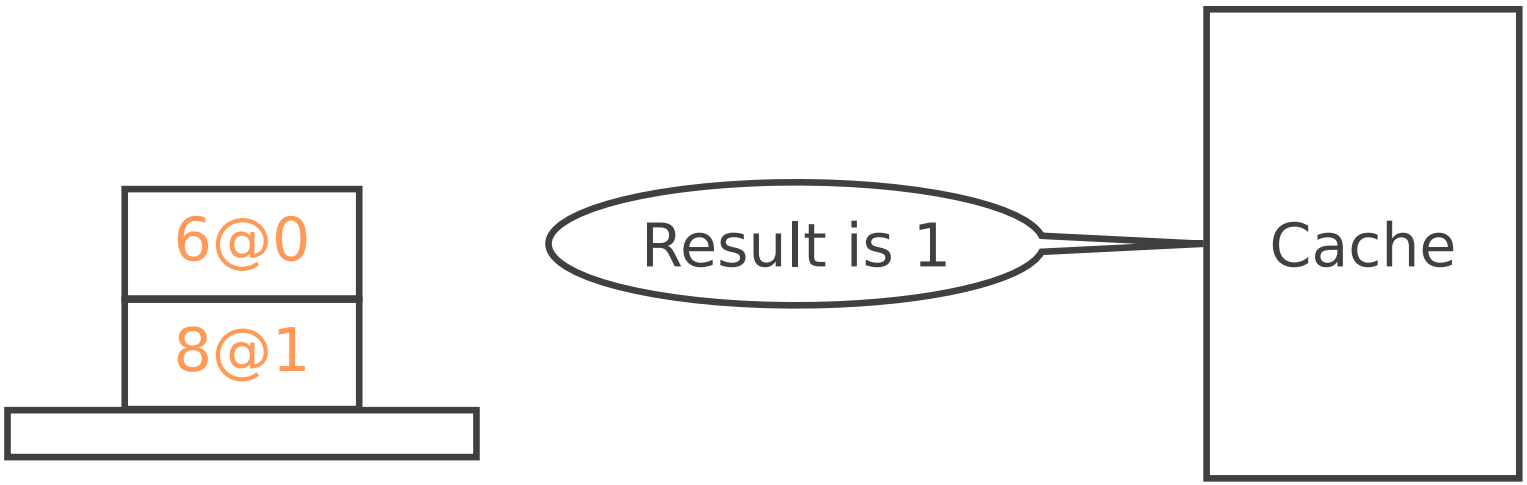
Cache look-up will then succeed,
allowing code to continue





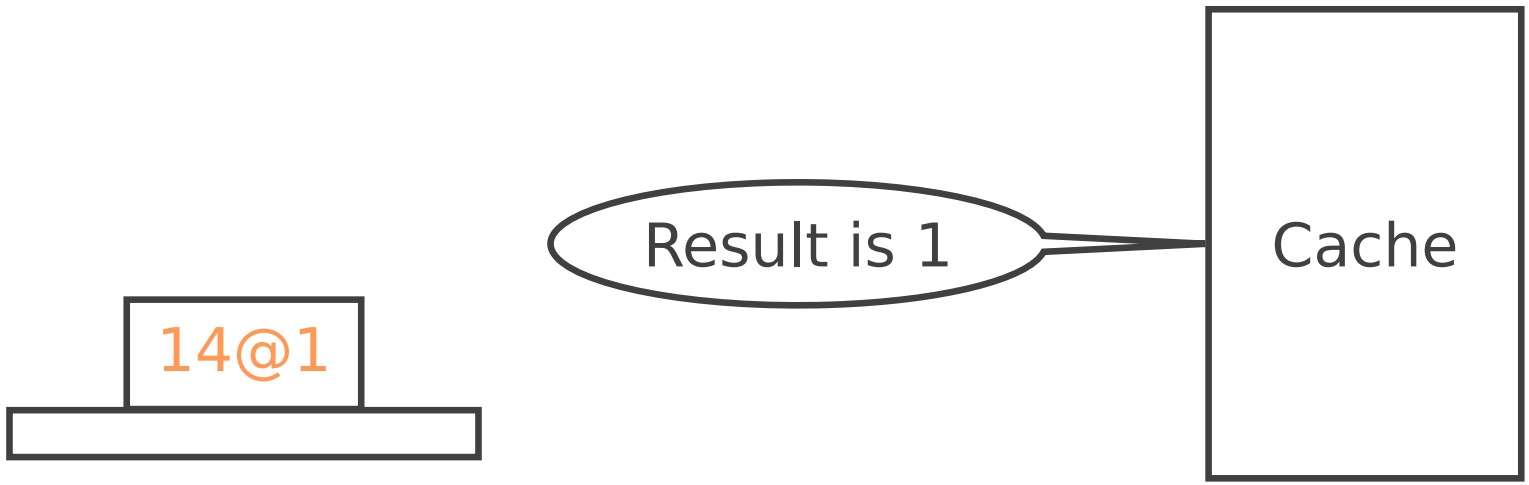
Fault-Handler Operation

Cache look-up will then succeed,
allowing code to continue

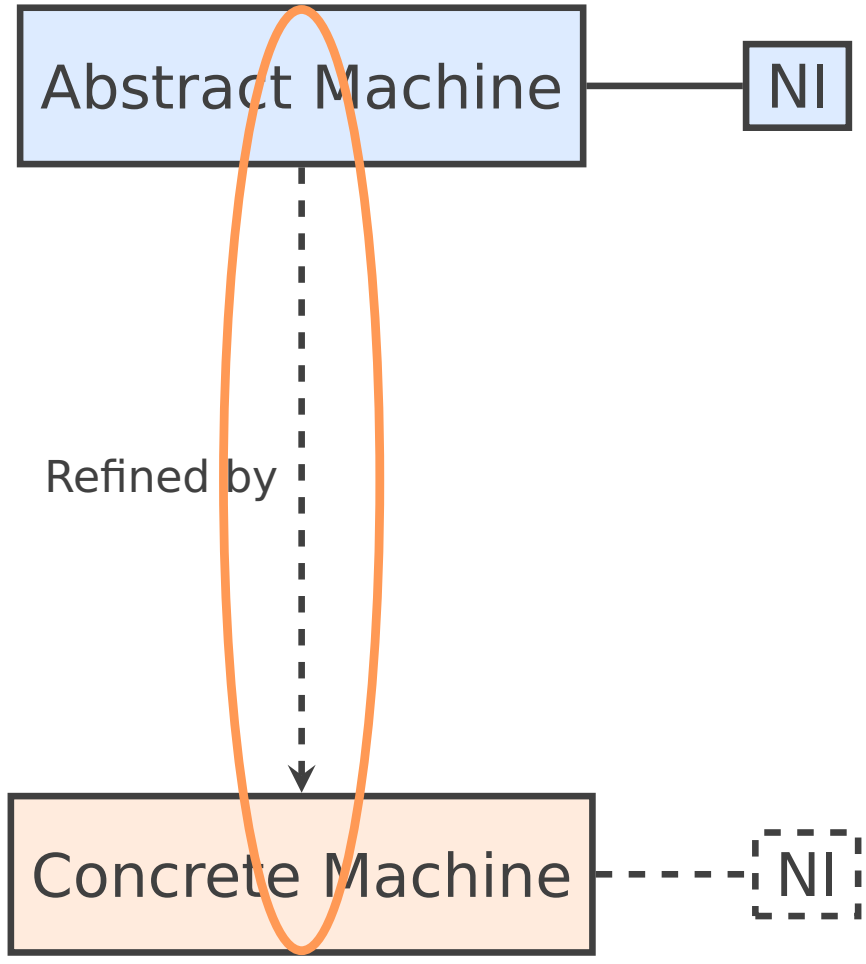


Fault-Handler Operation

Cache look-up will then succeed,
allowing code to continue

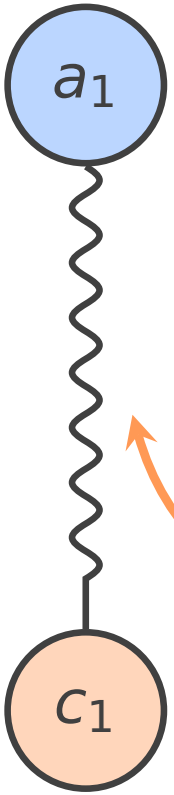


Proving the Refinement


Abstract Machine
NI
Refined by
Concrete Machine
NI

Refinement Structure

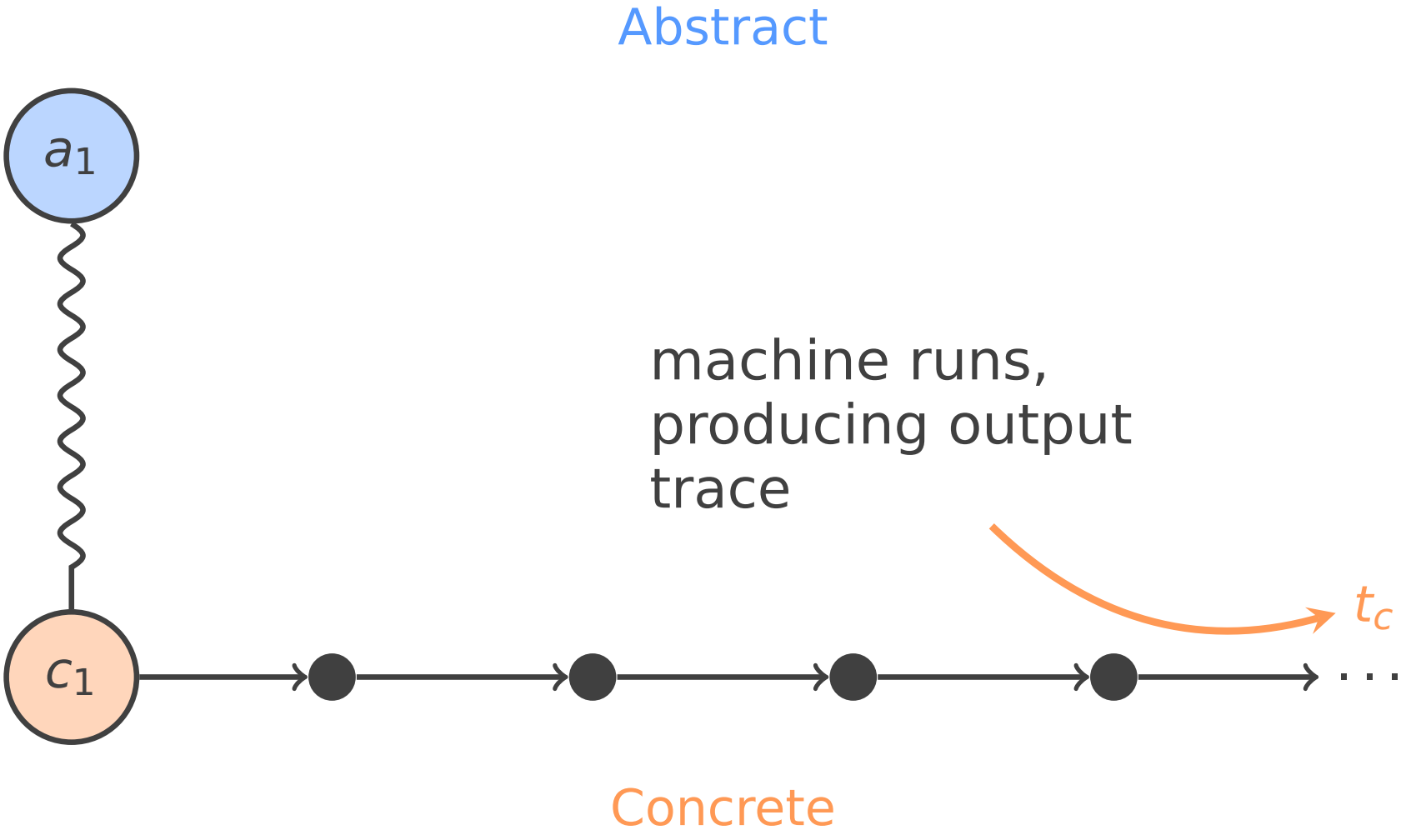
Abstract



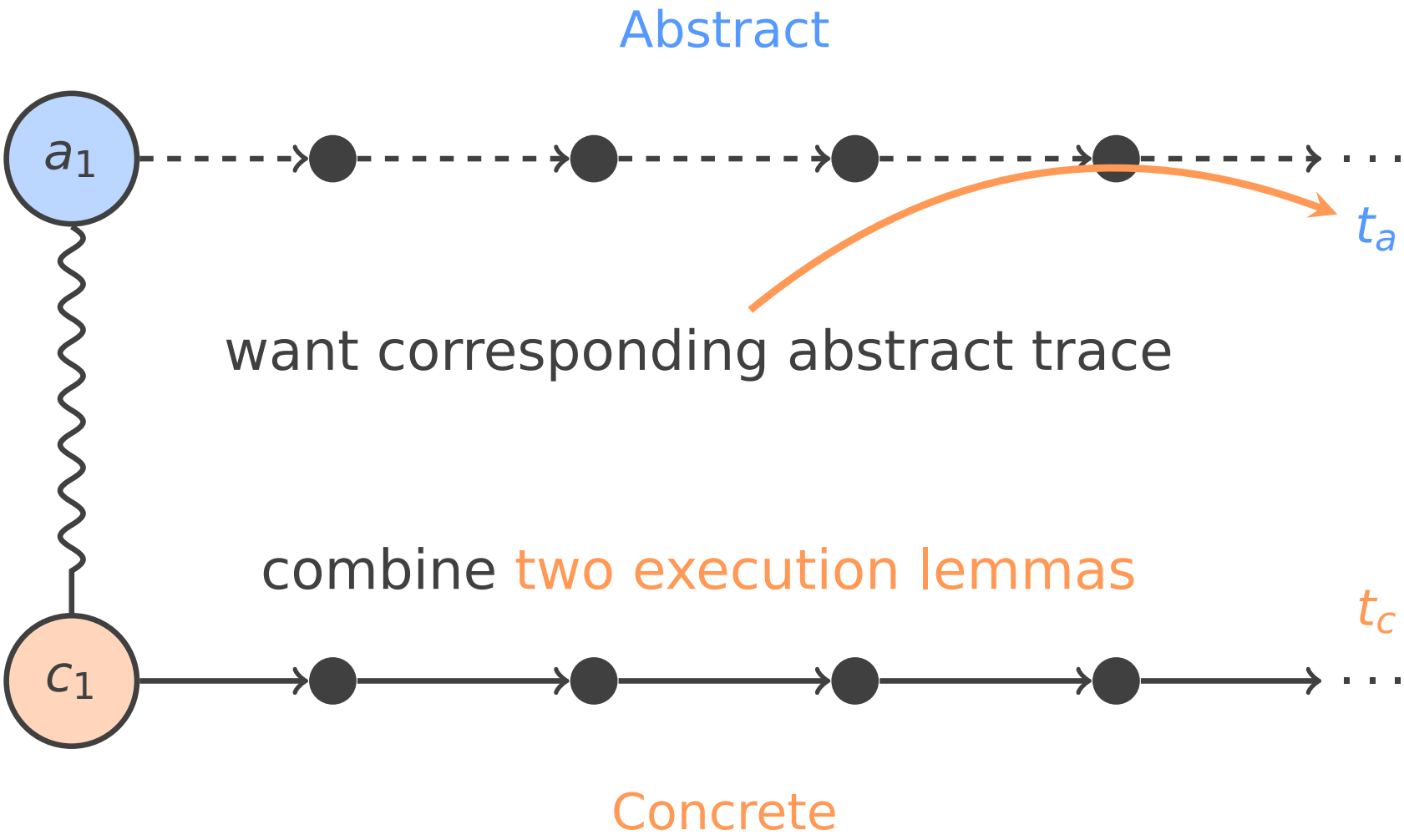
labels correctly represented, cache compatible with IFC rules, machine in user mode

Concrete

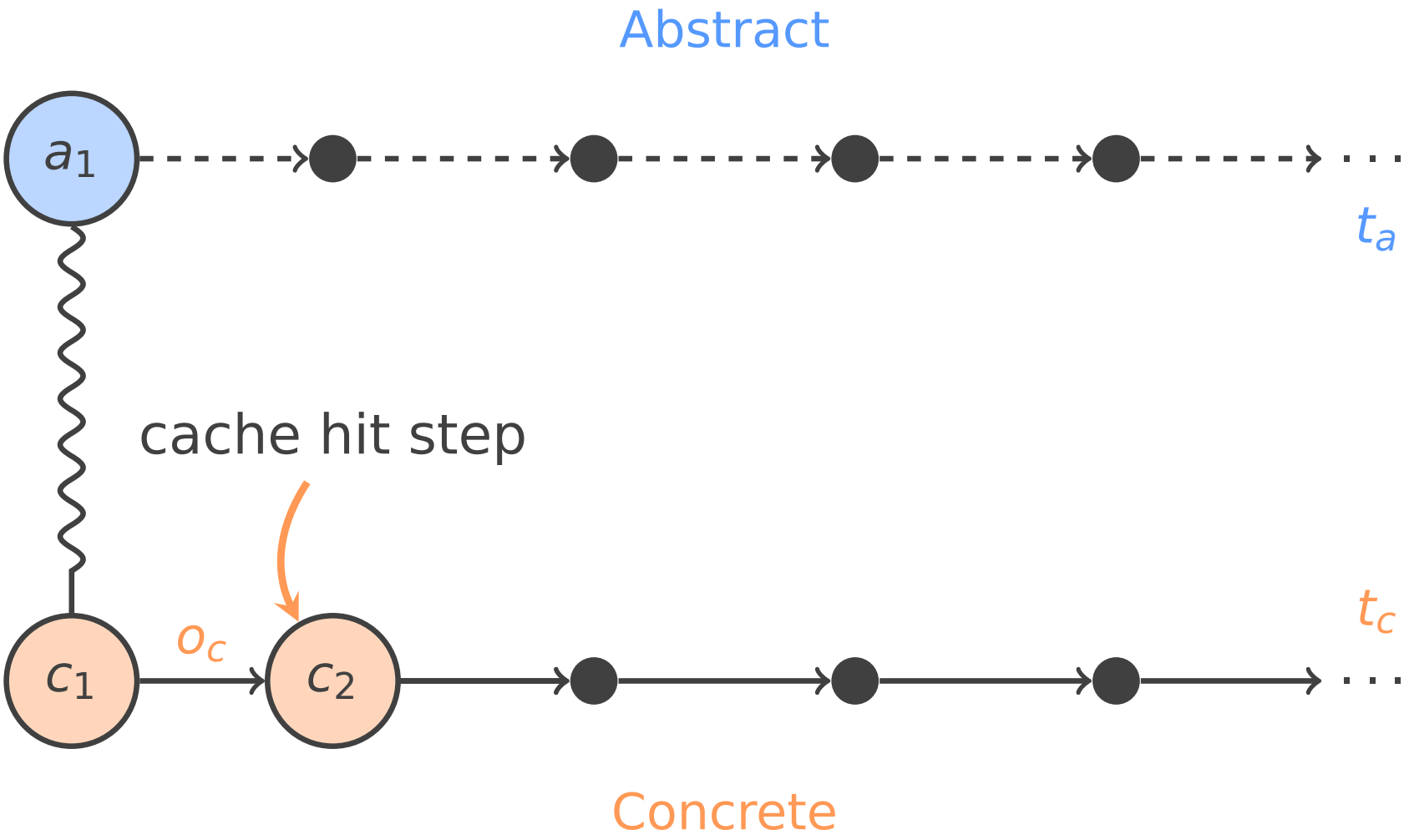
Refinement Structure



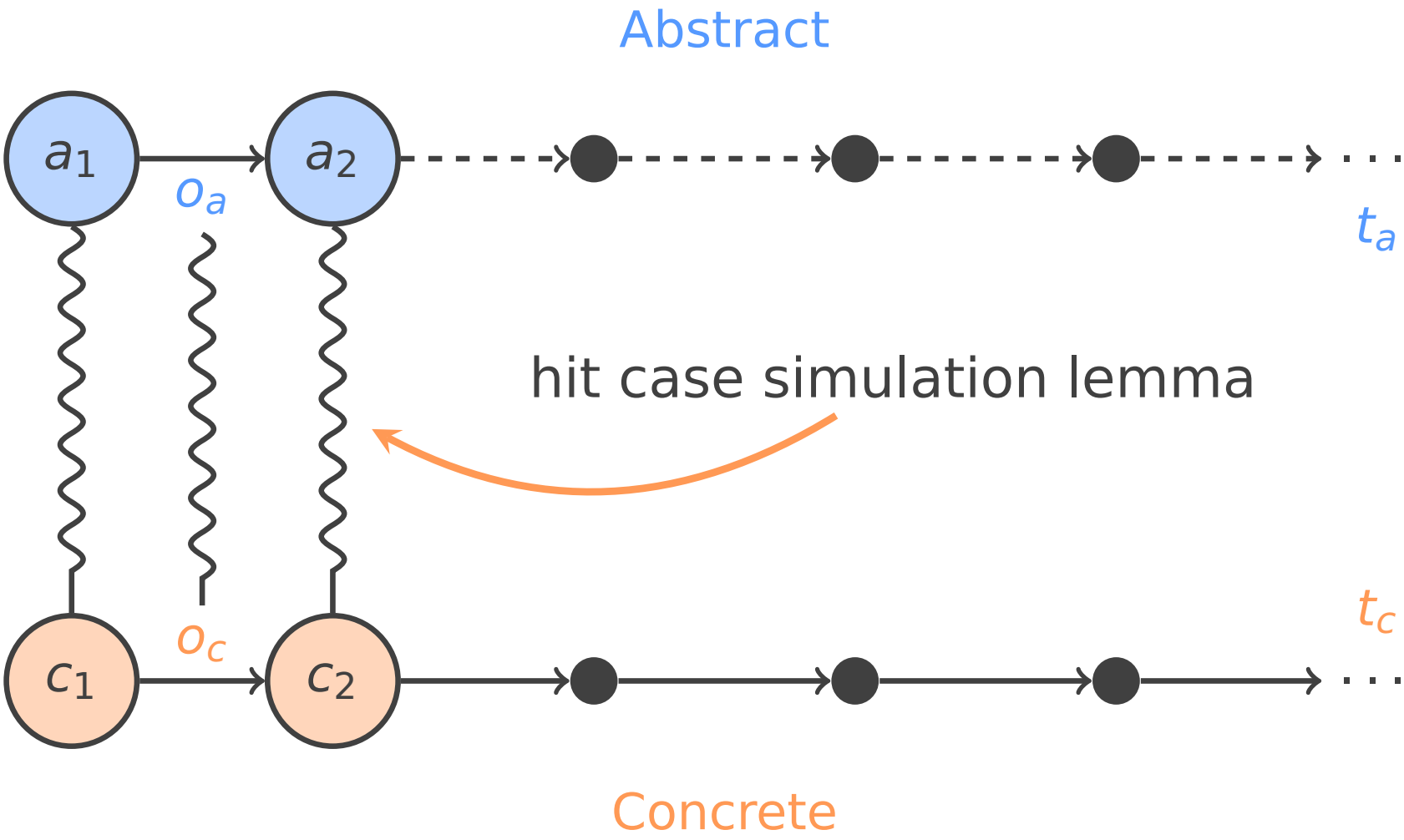
Refinement Structure



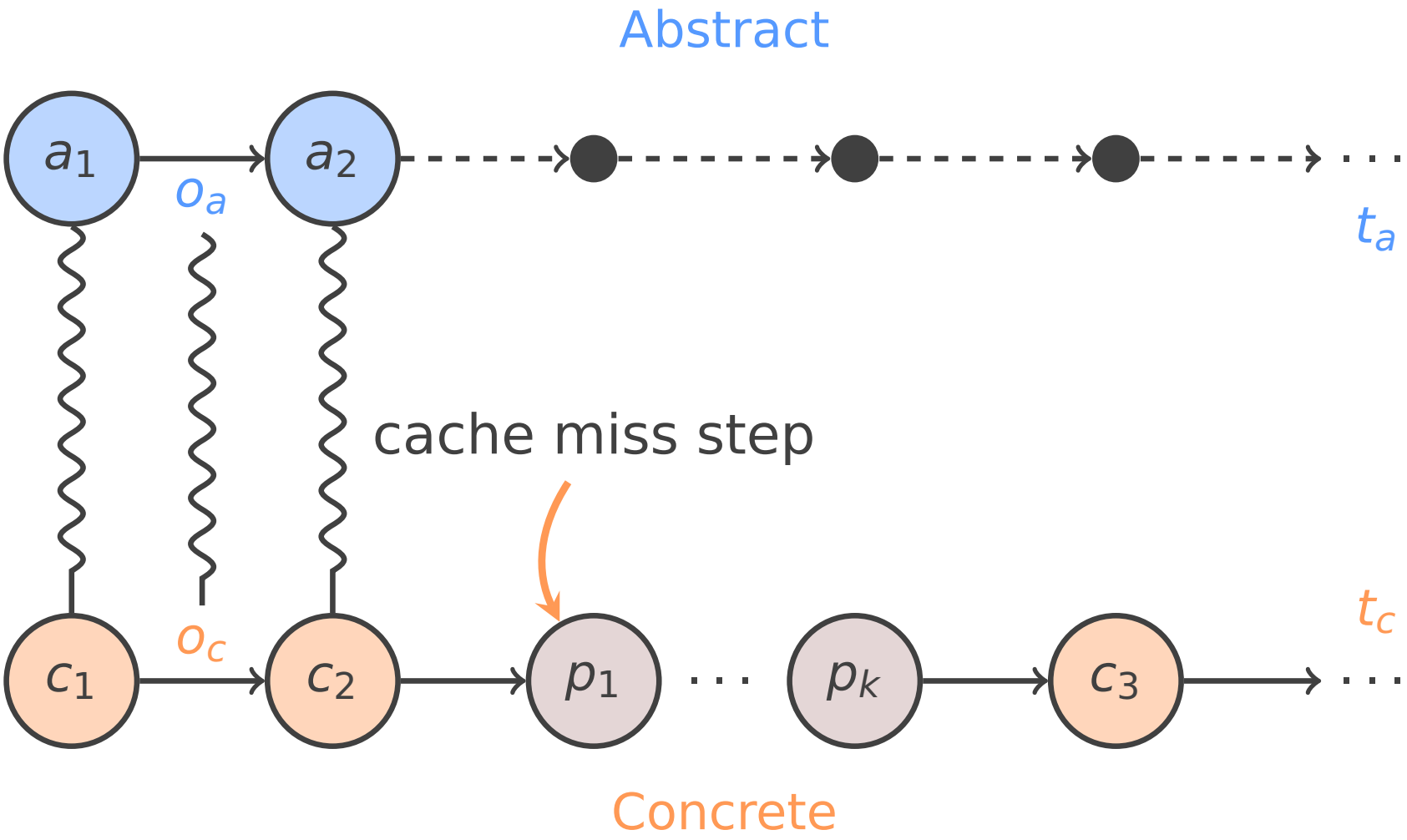
Refinement Structure



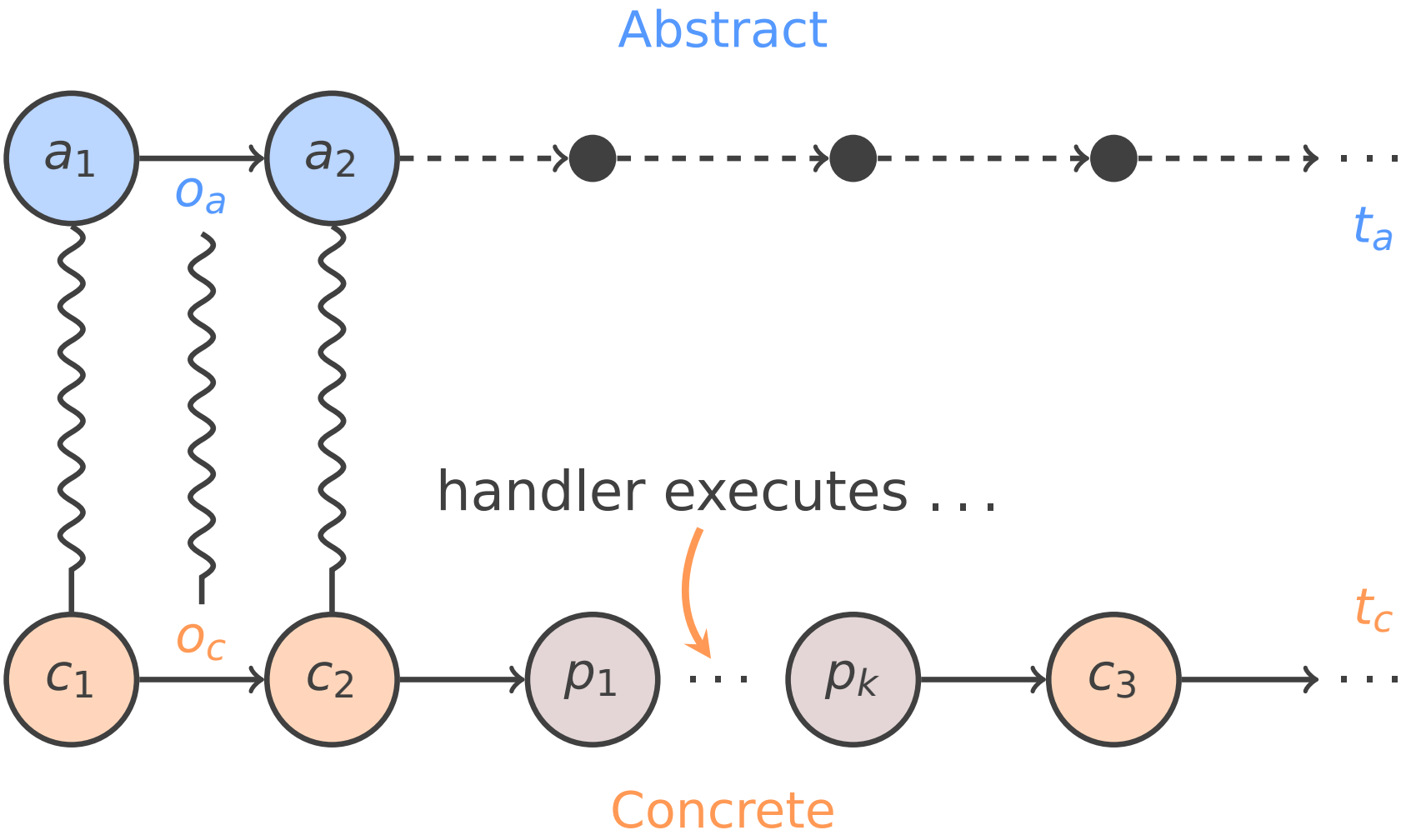
Refinement Structure



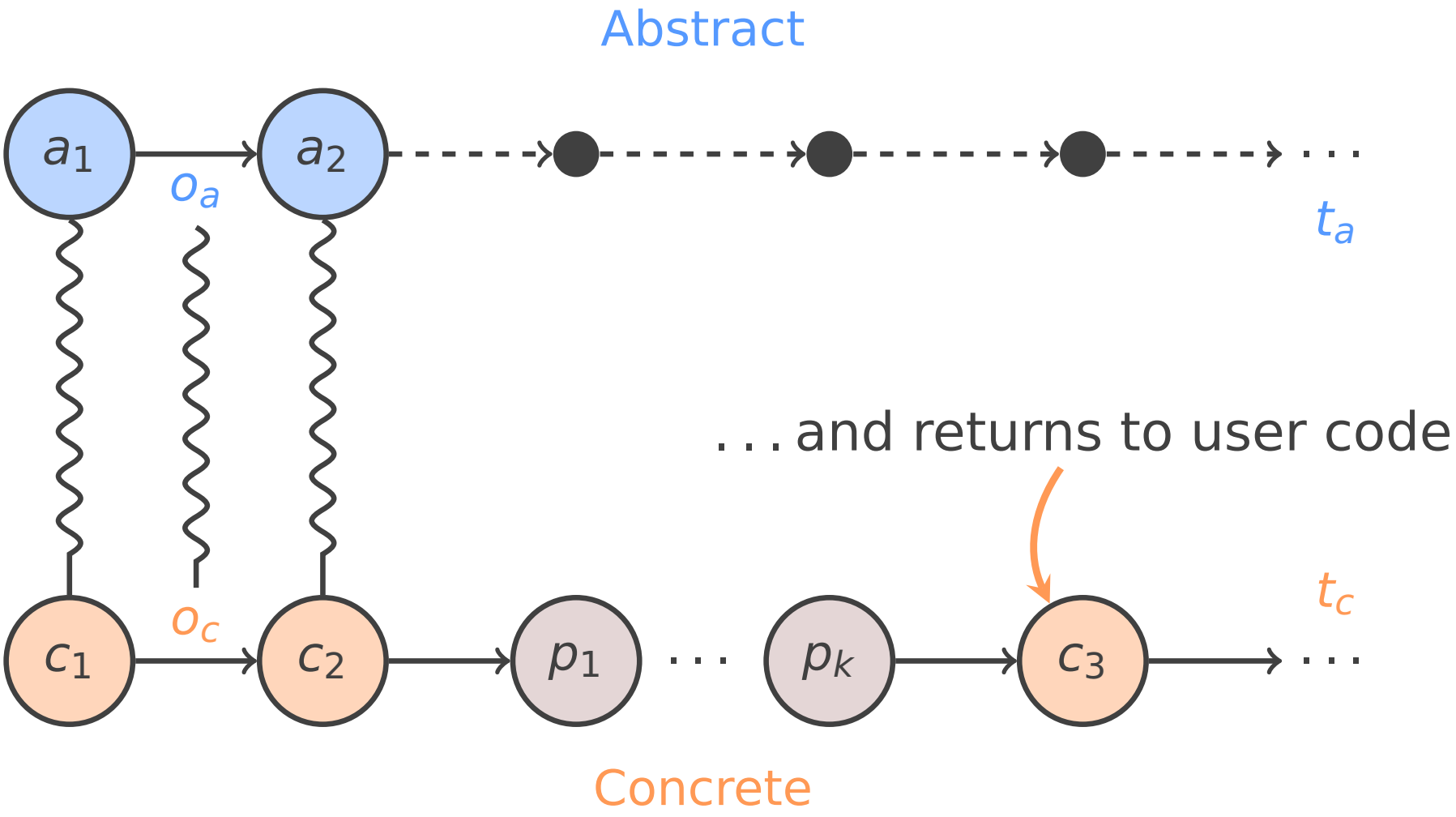
Refinement Structure



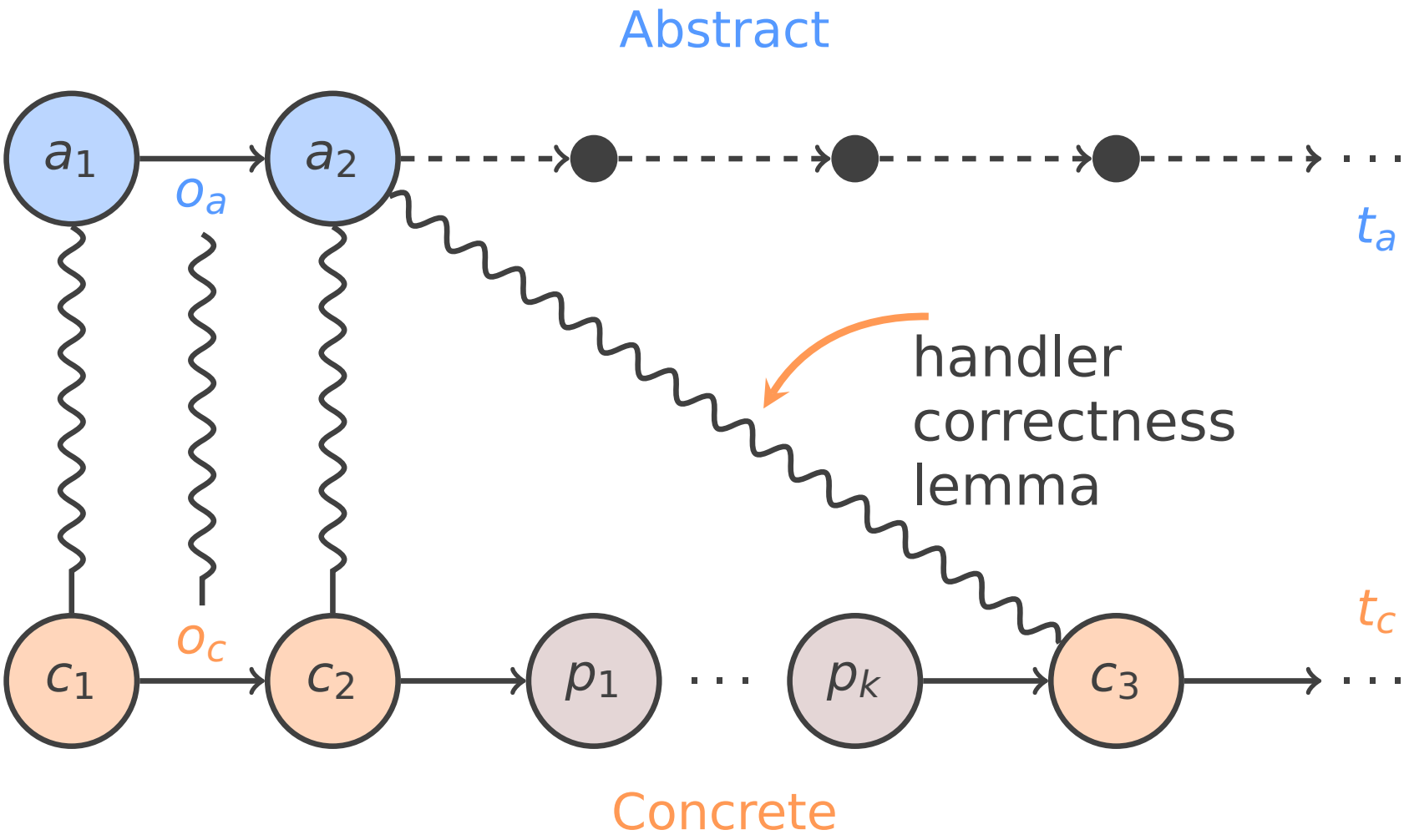
Refinement Structure



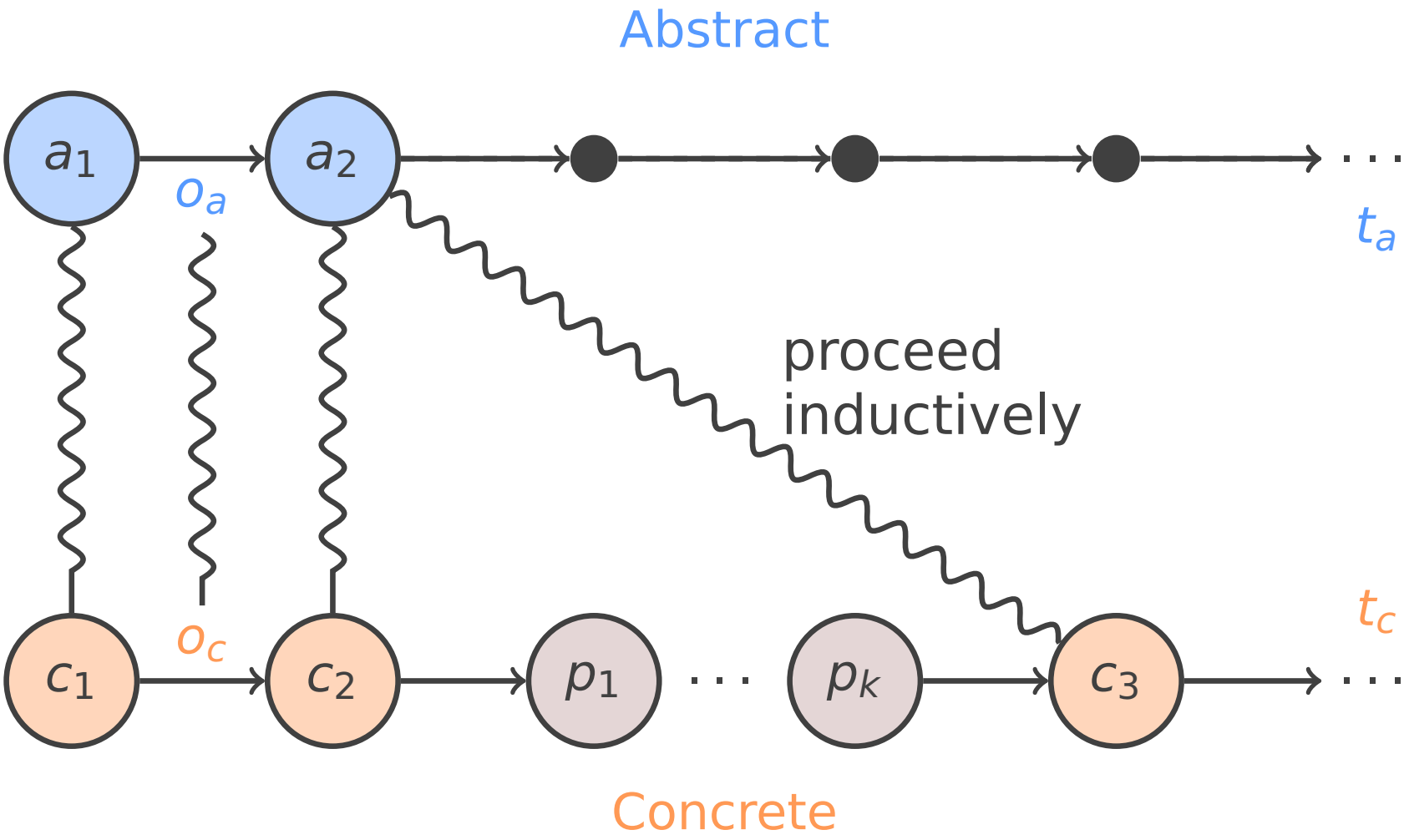
Refinement Structure



Refinement Structure



Refinement Structure

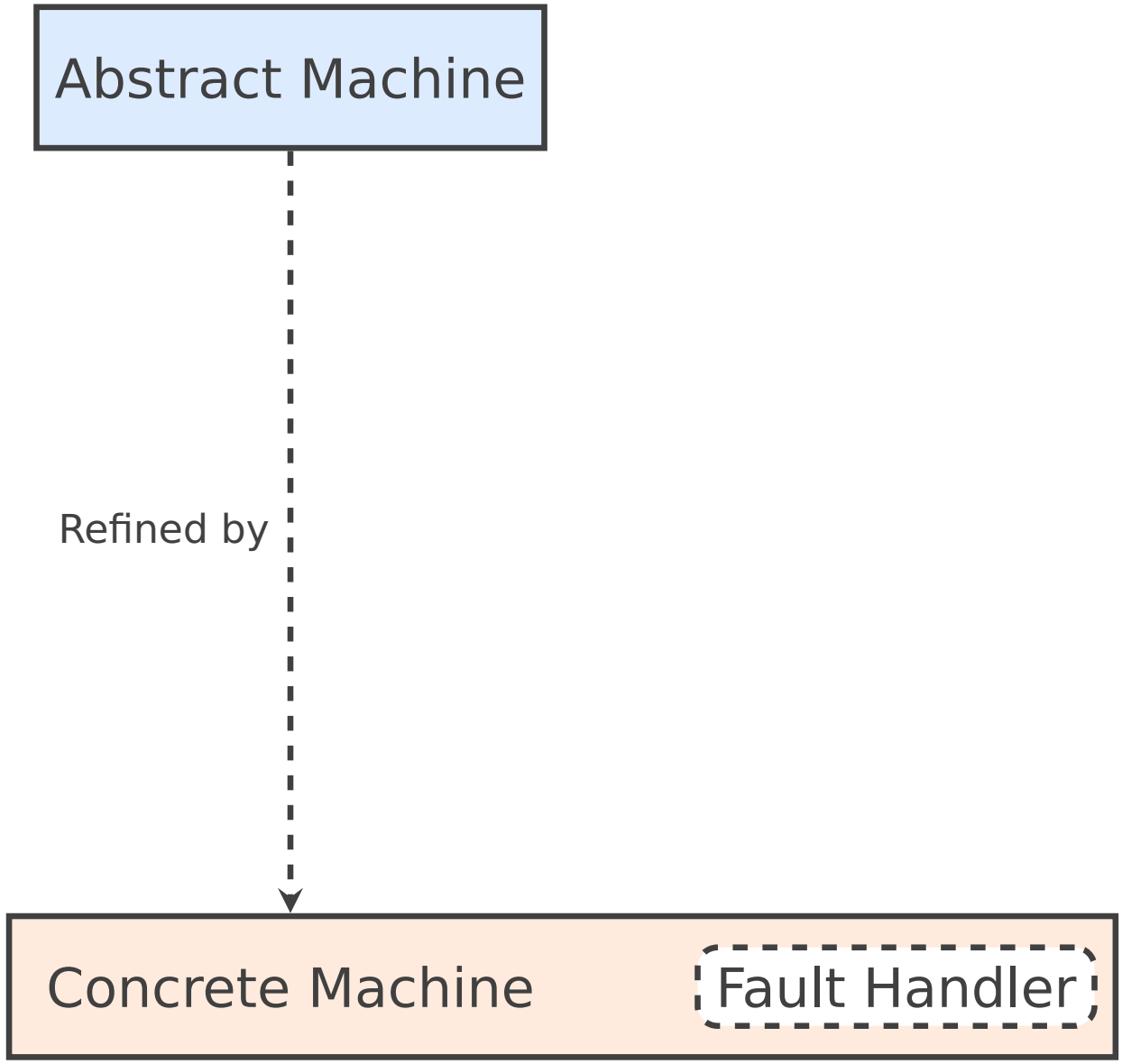


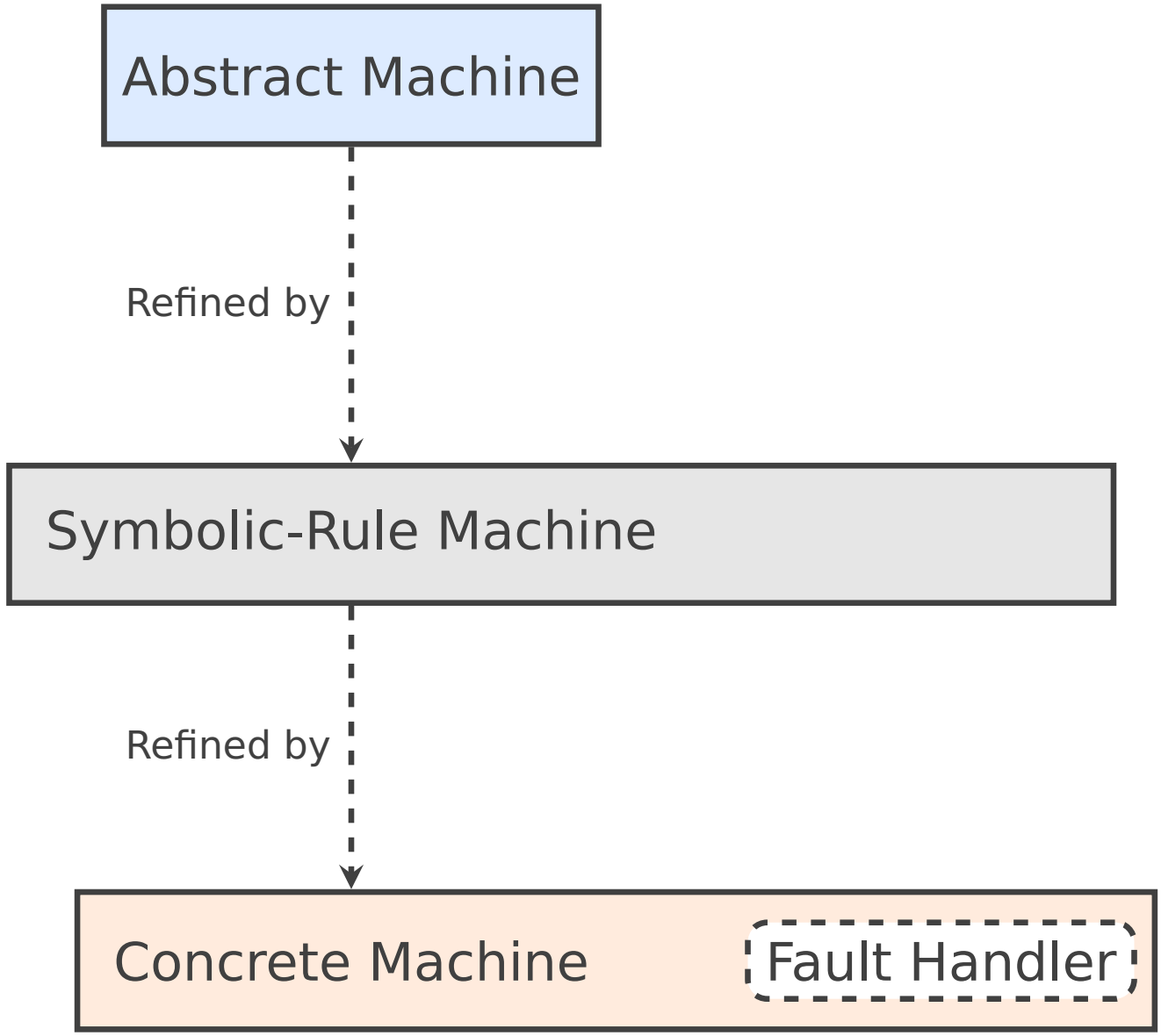
Challenges

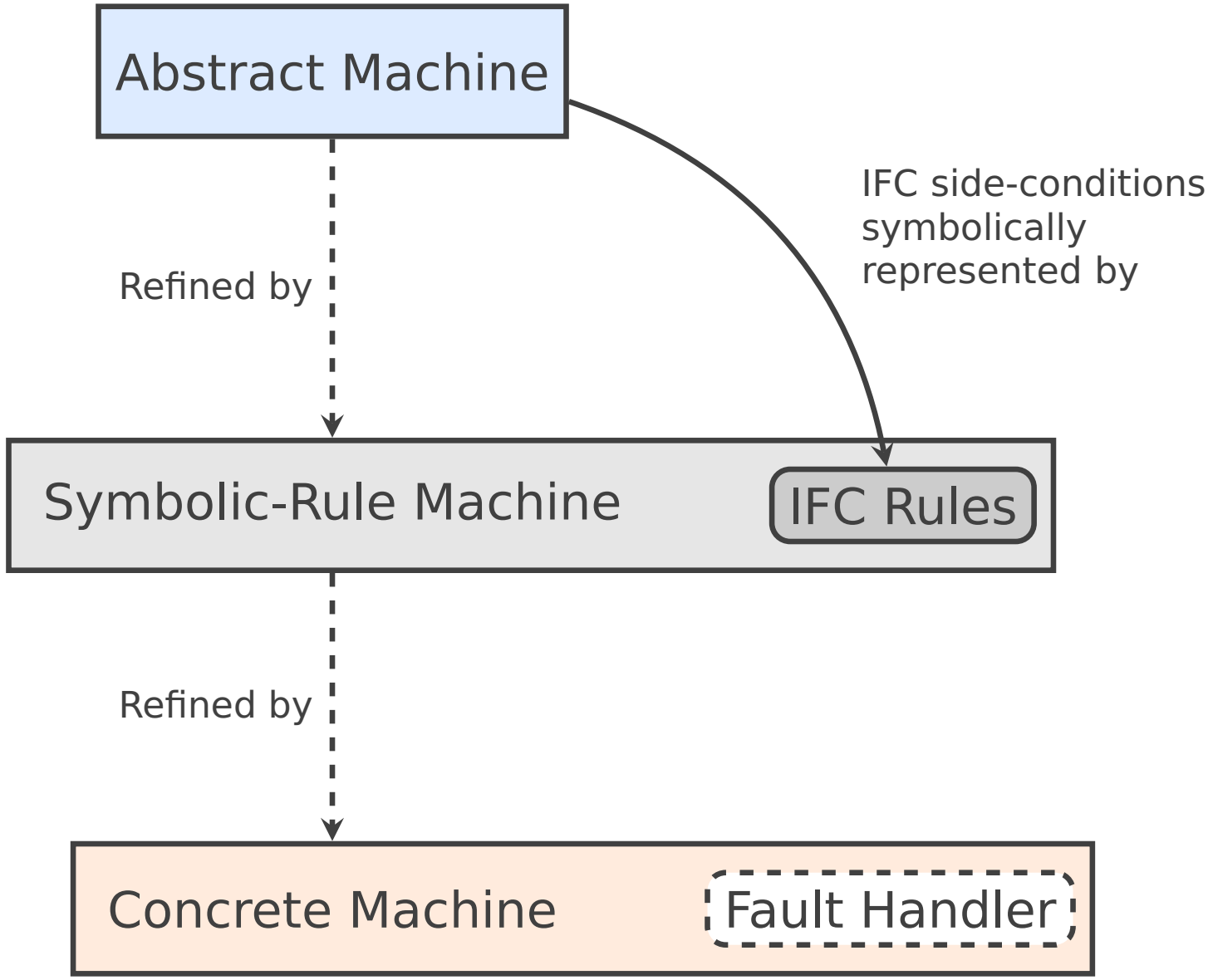
Interesting issues involved in these proofs

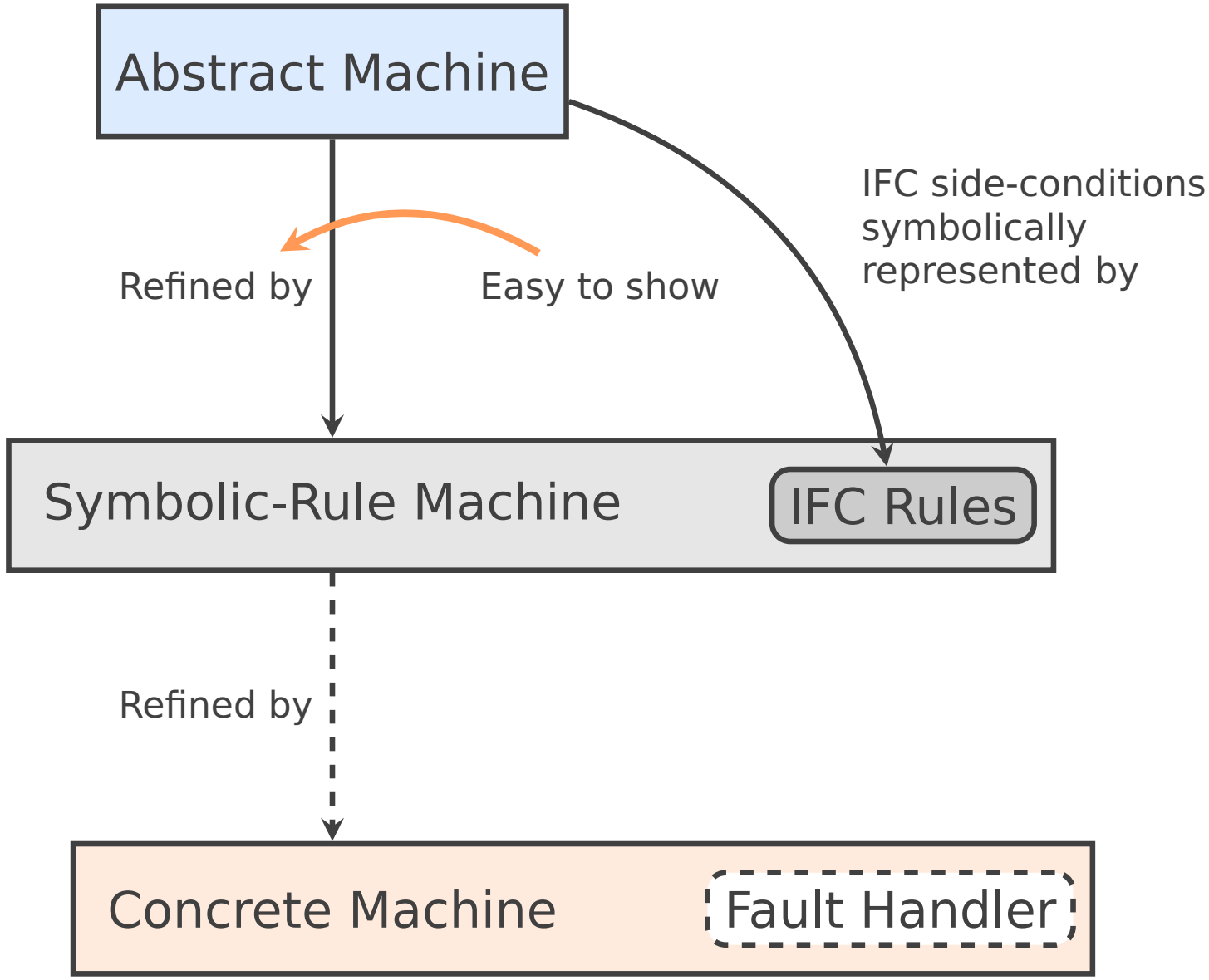
- Verification of machine code is difficult
- Need to formalize notion of compatibility between cache and IFC rules
- How to make it work for any IFC lattice?

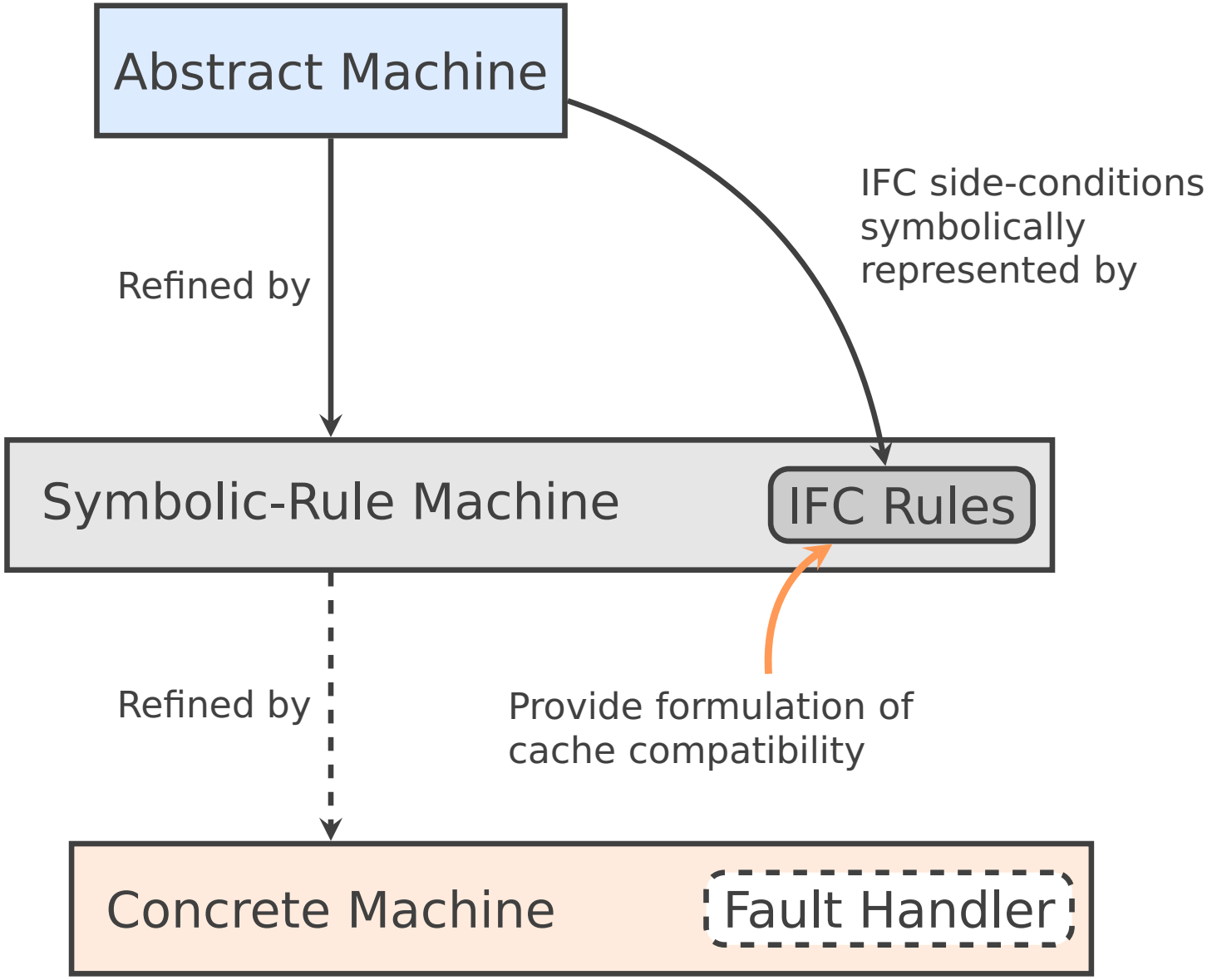
Idea: Structure proof and implementation with another refinement

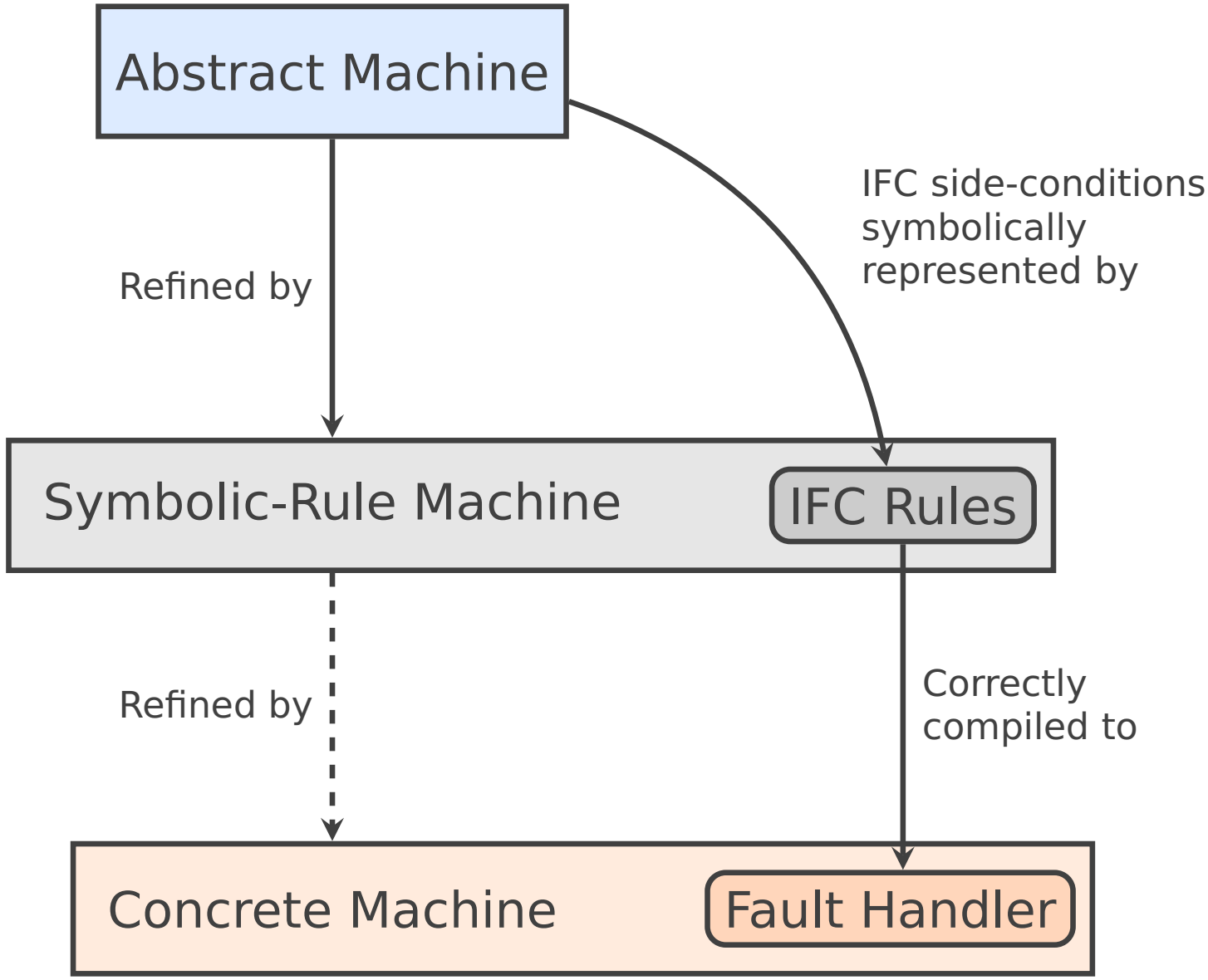

Abstract Machine
Refined by
Concrete Machine
Fault Handler

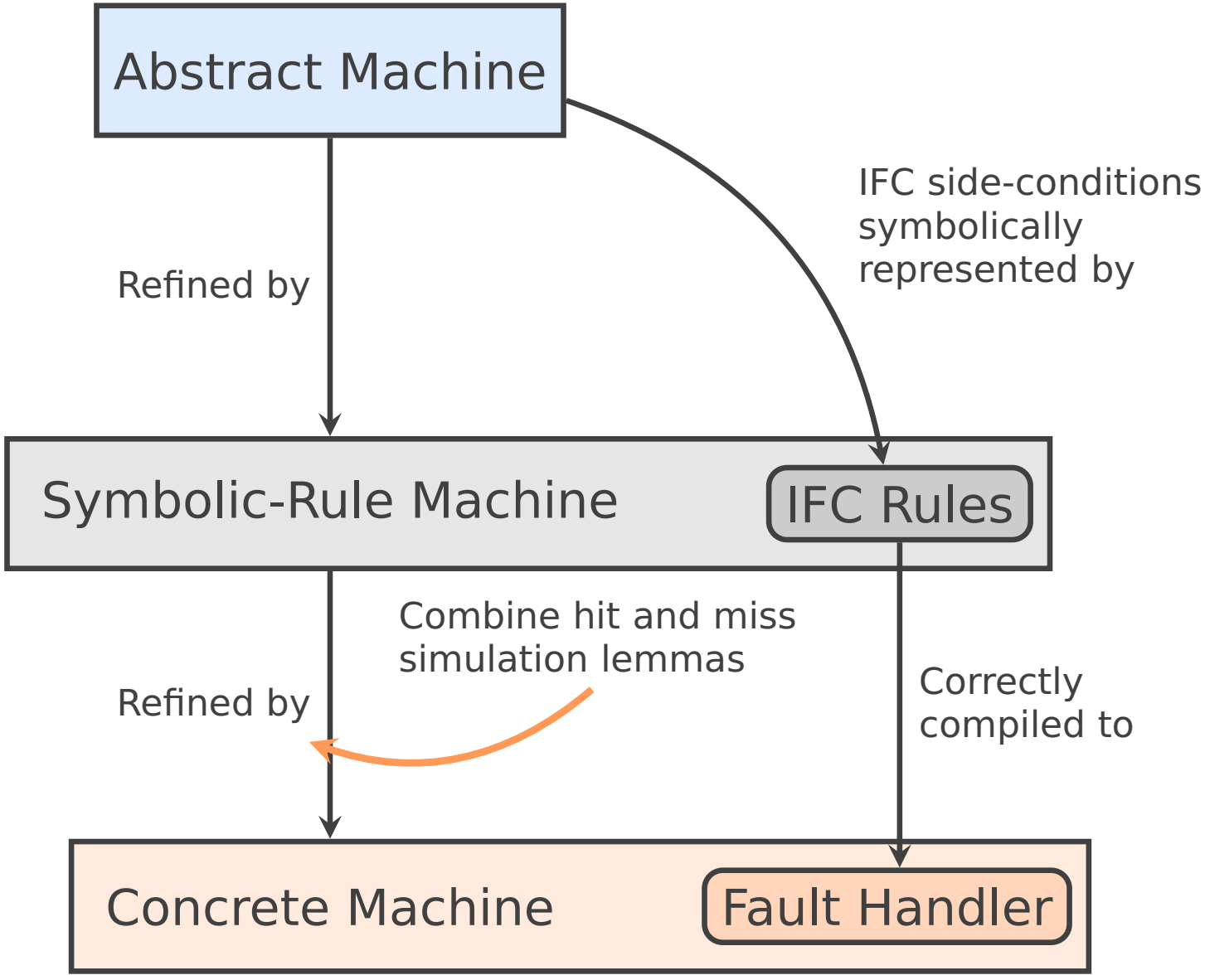

Abstract Machine
Refined by
Symbolic-Rule Machine
Refined by
Concrete Machine
Fault Handler


Abstract Machine
IFC side-conditions
symbolically
represented by
Refined by
Symbolic-Rule Machine
IFC Rules
Refined by
Concrete Machine
Fault Handler

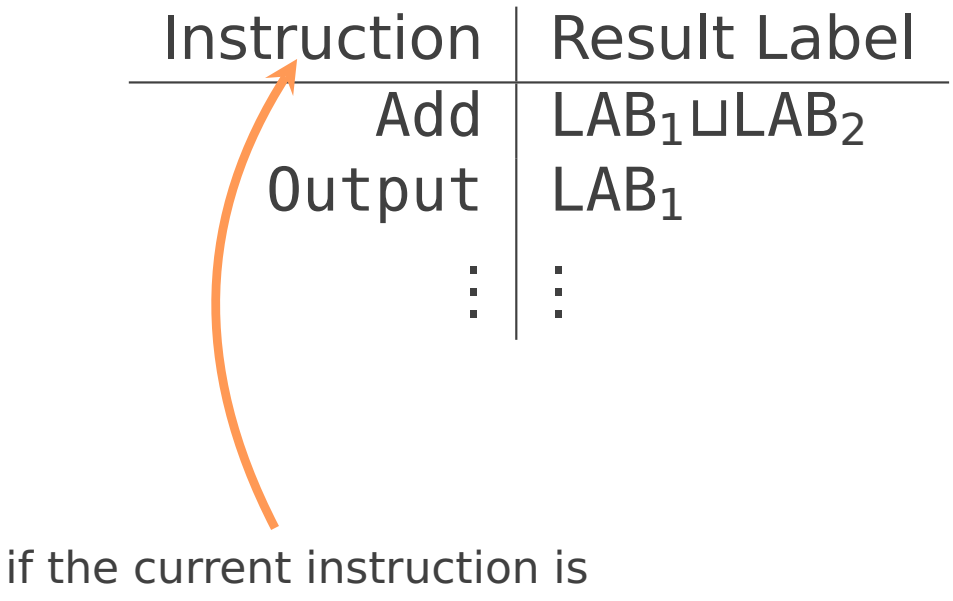

Abstract Machine
IFC side-conditions
symbolically
represented by
Refined by
Easy to show
Symbolic-Rule Machine
IFC Rules
Refined by
Concrete Machine
Fault Handler


Abstract Machine
IFC side-conditions
symbolically
represented by
Refined by
Symbolic-Rule Machine
IFC Rules
Refined by
Provide formulation of
cache compatibility
Concrete Machine
Fault Handler


Abstract Machine
Refined by
IFC side-conditions
symbolically
represented by
Symbolic-Rule Machine
IFC Rules
Refined by
Correctly
compiled to
Concrete Machine
Fault Handler

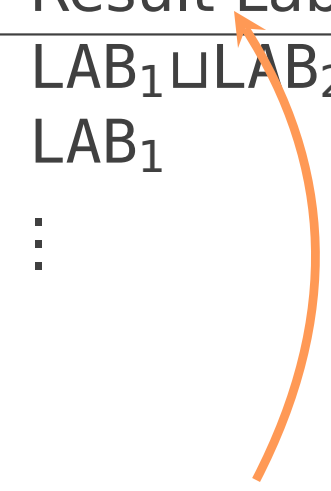

Abstract Machine
Refined by
IFC side-conditions
symbolically
represented by
Symbolic-Rule Machine
IFC Rules
Combine hit and miss
simulation lemmas
Refined by
Correctly
compiled to
Concrete Machine
Fault Handler

Symbolic Rule Table



Symbolic Rule Table

Instruction	Result Label
Add	$LAB_1 \sqcup LAB_2$
Output	LAB_1
⋮	⋮



label the result with

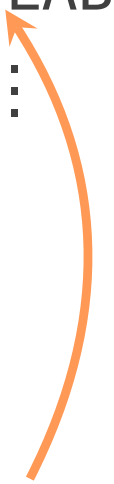
Symbolic Rule Table

Instruction	Result Label
Add	$LAB_1 \sqcup LAB_2$
Output	LAB_1
⋮	⋮

for Add, result is as secret as operands

Symbolic Rule Table

Instruction	Result Label
Add	$LAB_1 \sqcup LAB_2$
`Output`	LAB_1
⋮	⋮



for `Output`, use same label

Handler Implementation and Verification

Structured-Code Generators

Structured programming instead of assembly programming

- Define structured-code generators as Coq functions
- Generators provide a structured language for the machine (`if`, `case`, `and`, `or`, `while`, `...`)
- Prove Hoare-logic rules for each generator

Compiling IFC Rules

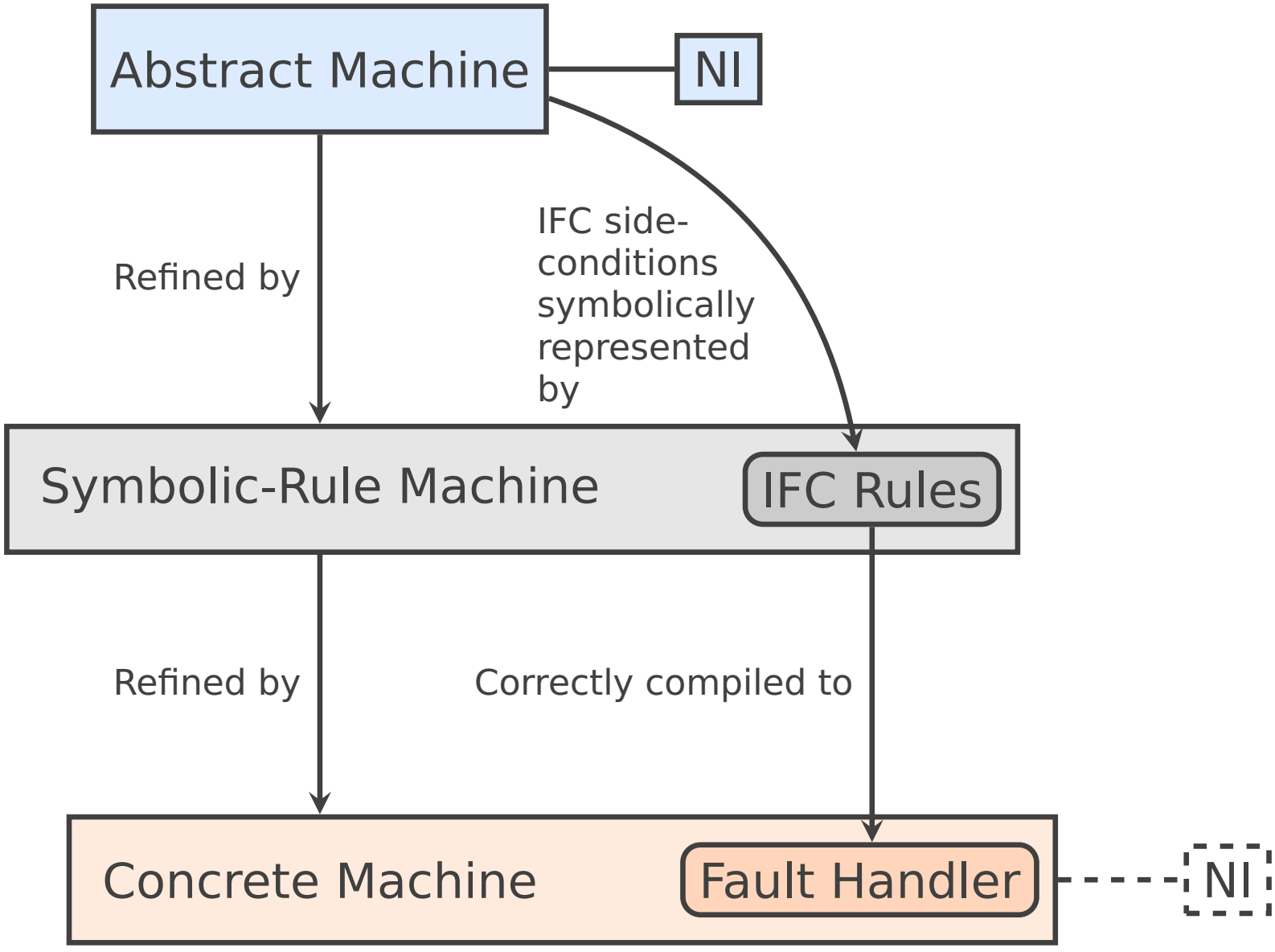
Write a rule table compiler in Coq

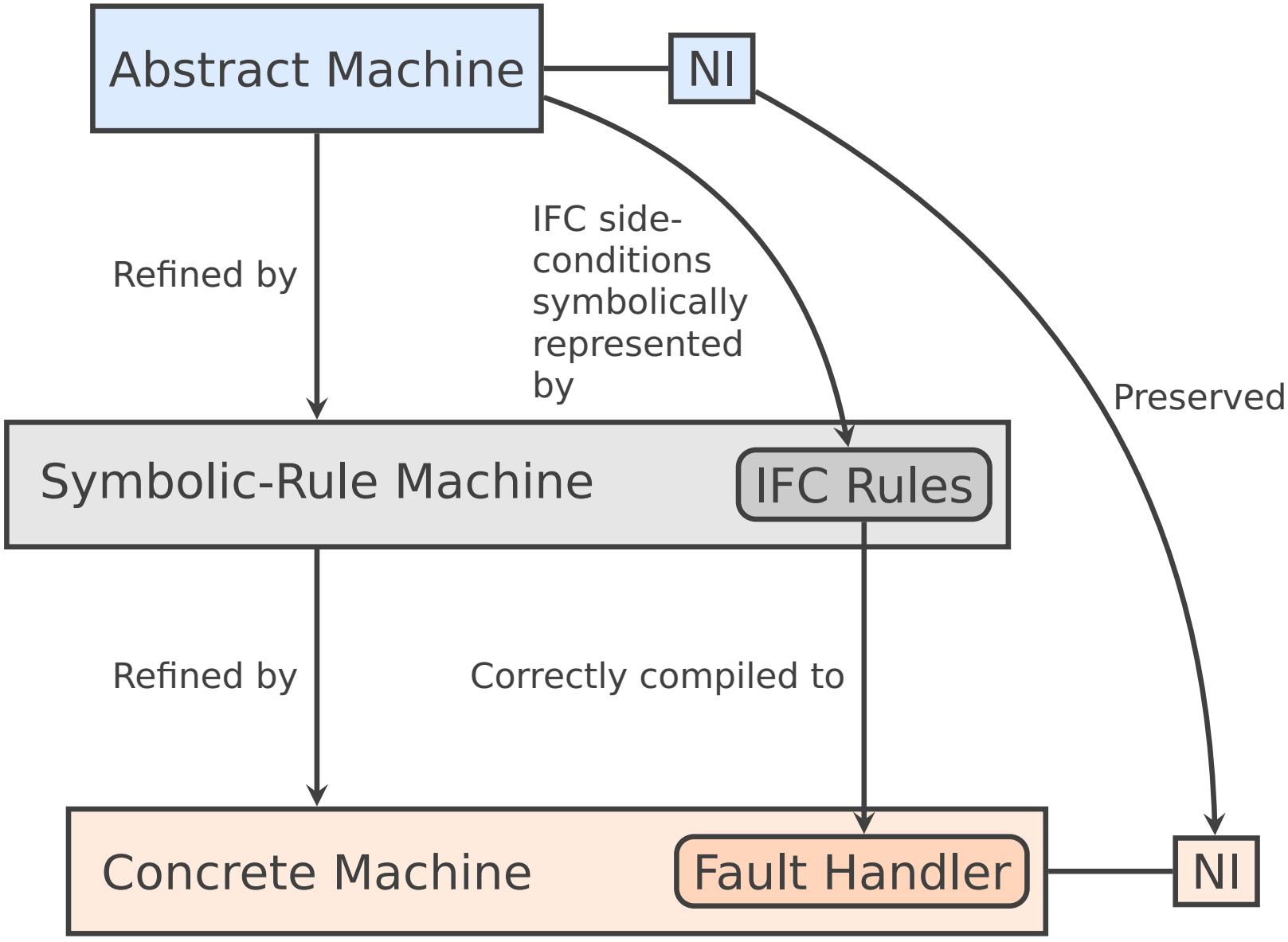
- Use generators as a backend
- Parameterized over correct implementation of lattice primitives
- Compose Hoare triples to show compiler correctness

Algorithm

- Fetch instruction and operand tags from faulting context
- Compute the result tag from this data using compiled rule table
- Install computed line into the cache

Proven correct by composing compiler lemma with triples for the glue code


Abstract Machine
NI
Refined by
IFC side-conditions symbolically represented by
Symbolic-Rule Machine
IFC Rules
Refined by
Correctly compiled to
Concrete Machine
Fault Handler
NI


Abstract Machine
NI
Refined by
IFC side-conditions symbolically represented by
Preserved
Symbolic-Rule Machine
IFC Rules
Refined by
Correctly compiled to
Concrete Machine
Fault Handler
NI

What Else?

More in the Paper

Complete model includes more features

- Control flow and user-level procedures
- Block-structured memory with dynamic allocation
- System calls for implementing new IFC primitives
- Richer IFC labels (sets of principals represented as pointers to memory arrays)

Addressed Challenges

- Track implicit flows

Addressed Challenges

- Track implicit flows
- Allocation and noninterference

Addressed Challenges

- Track implicit flows
- Allocation and noninterference
  - Pointer values could leak secrets

Addressed Challenges

- Track implicit flows
- Allocation and noninterference
  - Pointer values could leak secrets
- Label representation depending on machine state

Addressed Challenges

- Track implicit flows
- Allocation and noninterference
  - Pointer values could leak secrets
- Label representation depending on machine state
- Low-level code for array manipulation and corresponding proofs

Wrapping Up

Conclusions

- Described a hardware mechanism for dynamic tag-checking and propagation
- Proof architecture for connecting it to high-level property
  - Refinement provides structure to proof
- Everything formalized in Coq

Coq Development

Entire formalization available at `www.crash-safe.org`

Complete machine and corresponding proofs in approximately 15k LOC

Future Work

- More interesting IFC features (concurrency, declassification, dynamic principal generation, . . .)
- Make model more realistic
  - Incorporate more features of SAFE
  - Study tag cache in the context of conventional architectures
- Mechanism is not IFC-specific, investigate other applications.

Thank You!

Questions?