Lecture 26

CIS 341: COMPILERS

Announcements

- HW 7: Optimization & Experiments
  - Due: April 29th
- Final Exam:
  - Thursday, May 7th
  - 9:00AM
  - Moore 216
- Visitor: Yaron Minsky of Jane St. Capital
  - Monday, April 27th
  - Lunch: noon – 1:15 (Raisler Lounge) sign-up sheet on Piazza
  - Talk: 2:00 – 3:00 (Raisler Lounge)
    From Theory into Practice: the story of Incremental

COMPILER VERIFICATION

Compiler Correctness?

- We have to relate the source and target language semantics across the compilation function $\mathfrak{C}$[-] : source → target.

$$\text{cmd} / \text{st} \;\; {}_{S}\!\longmapsto^{*} \;\; \texttt{SKIP} / \text{st}'$$

iff

$$\mathfrak{C}[\text{cmd}] / \mathfrak{C}[\text{st}] \;\; {}_{T}\!\longmapsto^{*} \;\; \mathfrak{C}[\text{st}']$$

- Is this enough?
- What if cmd goes into an infinite loop?

Comparing Behaviors

- Consider two programs P1 and P2 possibly in different languages.
  - e.g. P1 is an Oat program, P2 is its compilation to LL

- The semantics of the languages associate to each program a set of observable behaviors:

$$\mathfrak{B}(P) \text{ and } \mathfrak{B}(P')$$

- Note: $|\mathfrak{B}(P)| = 1$ if P is deterministic, > 1 otherwise

What is Observable?

- For C-like languages:

  observable behavior ::=
  | terminates(st) (i.e. observe the final state)
  | diverges
  | goeswrong

- For pure functional languages:

  observable behavior ::=
  | terminates(v) (i.e. observe the final value)
  | diverges
  | goeswrong

What about I/O?

- Add a *trace* of input-output events performed:

```
          t     ::=  []   |   e :: t          (finite traces)
coind.    T     ::=  []   |   e :: T          (finite and infinite traces)
```

```
observable behavior ::=
          | terminates(t, st)    (end in state st after trace t)
          | diverges(T)          (loop, producing trace T)
          | goeswrong(t)
```

Examples

- P1:
  `print(1);` / st ⇒ terminates(out(1)::[],st)

- P2:
  `print(1); print(2);` / st
  ⇒ terminates(out(1)::out(2)::[],st)

- P3:
  `WHILE true DO print(1) END` / st
  ⇒ diverges(out(1)::out(1)::…)

- So $\mathfrak{B}$(P1) ≠ $\mathfrak{B}$(P2) ≠ $\mathfrak{B}$(P3)

Bisimulation

- Two programs P1 and P2 are bisimilar whenever:

$$\mathfrak{B}(P1) = \mathfrak{B}(P2)$$

- The two programs are completely indistinguishable.

- But… this is often too strong in practice.

Compilation Reduces Nondeterminism

- Some languages (like C) have underspecified behaviors:
  - Example: order of evaluation of expressions f() + g()

- Concurrent programs often permit nondeterminism
  - Classic optimizations can reduce this nondeterminism
  - Example:

    a := x + 1; b := x + 1 || x := x+1

    vs.

    a := x + 1; b := a || x := x+1

- LLVM explicitly allows nondeterminism:
  - undef values (not part of LLVM lite)
  - see the discussion later

Backward Simulation

- Program P2 can exhibit fewer behaviors than P1:

$$\mathfrak{B}(P1) \supseteq \mathfrak{B}(P2)$$

- All of the behaviors of P2 are permitted by P1, though some of them may have been eliminated.
- Also called *refinement*.

What about goeswrong?

- Compilers often translate away bad behaviors.

x := 1/y ; x := 42	vs.	x := 42
(divide by 0 error)		(always terminates)

- Justifications:
  - Compiled program does not "go wrong" because the program type checks or is otherwise formally verified
  - Or just "garbage in/garbage out"

Safe Backwards Simulation

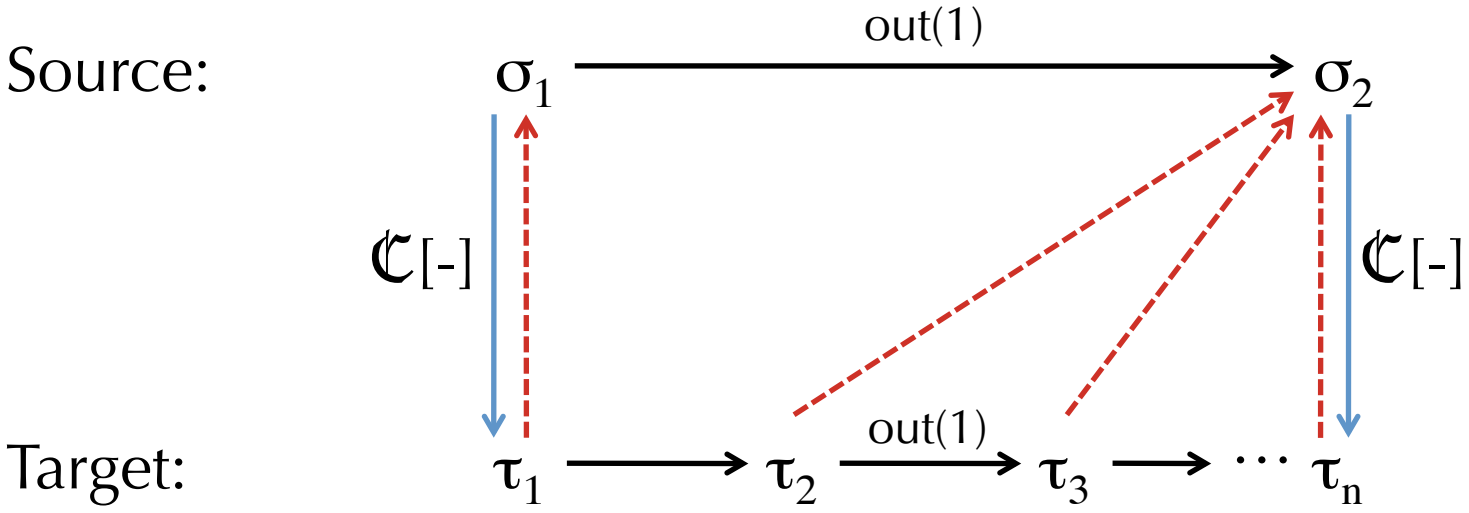
- Only require the compiled program's behaviors to agree if the source program could not go wrong:

$$\text{goeswrong}(t) \notin \mathfrak{B}(P1) \quad \Rightarrow \quad \mathfrak{B}(P1) \supseteq \mathfrak{B}(P2)$$

- Idea: let S be the *functional specification* of the program:
  A set of behaviors not containing goeswrong(t).
  - A program P satisfies the spec if $\mathfrak{B}(P) \subseteq S$

- Lemma: If P2 is a safe backwards simulation of P1 and P1 satisfies the spec, then P2 does too.

Building Backward Simulations



Idea: The event trace along a (target) sequence of steps originating from a compiled program must correspond to some source sequence.

Tricky parts:

- Must consider all possible target steps
- If the compiler uses many target steps for once source step, we have invent some way of relating the intermediate states to the source.
- the compilation function goes the wrong way to help!

Safe Forwards Simulation

- Source program's behaviors are a subset of the target's:

$$\text{goeswrong}(t) \notin \mathfrak{B}(P1) \quad \Rightarrow \quad \mathfrak{B}(P1) \subseteq \mathfrak{B}(P2)$$

- P2 captures all the good behaviors of P1, but could exhibit more (possibly bad) behaviors.

- But: Forward simulation is significantly easier to prove:
  - Only need to show the existence of a compatible target trace.

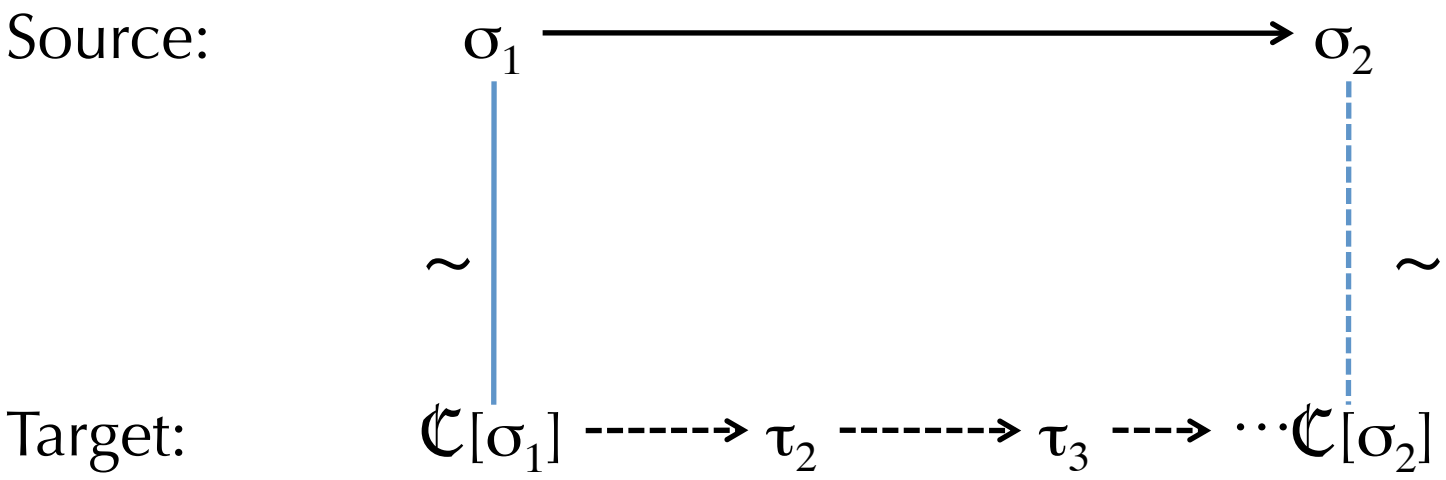
Determinism!

- Lemma: If P2 is deterministic then forward simulation implies backward simulation.

- Proof: $\varnothing \subset \mathfrak{B}(P1) \subseteq \mathfrak{B}(P2) = \{b\}$ so $\mathfrak{B}(P1) = \{b\}$.

- Corollary: safe forward simulation implies safe backward simulation if P2 is deterministic.

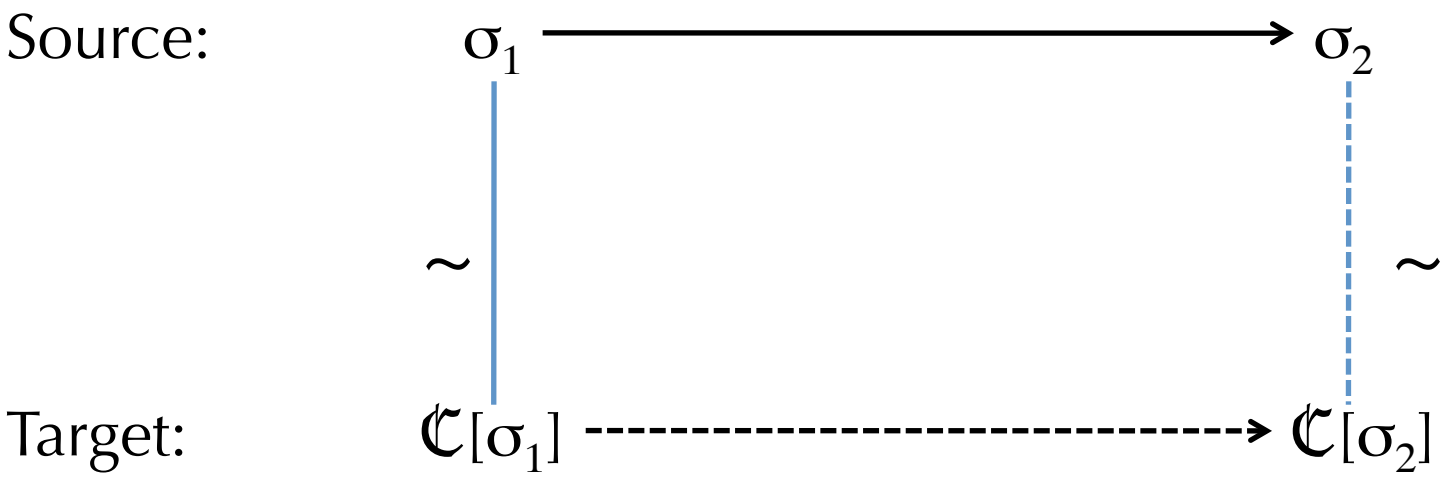
Forward Simulations



Idea: Show that every transition in the source program:

- is simulated by some sequence of transitions in the target
- while preserving a relation ~ between the states

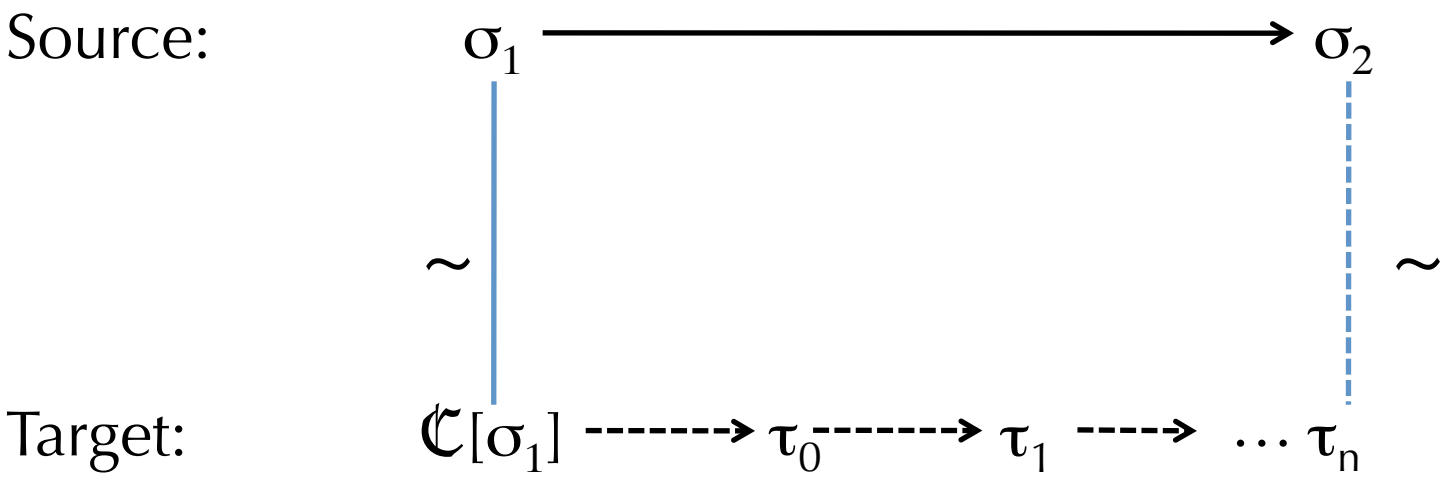
Lock-step Forward Simulation



A single source-program step is simulated by a single target step.

(Solid = assumptions, Dashed = must be shown)

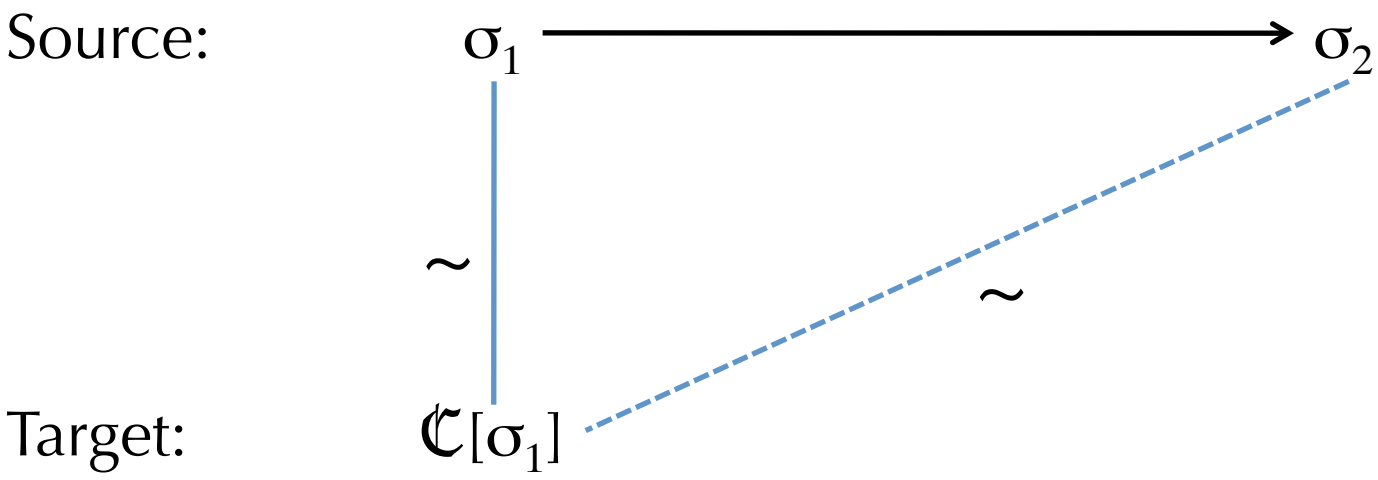
"Plus"-step Forward Simulation



A single source-program step is simulated by *one or more* target steps. (But only finitely many!)

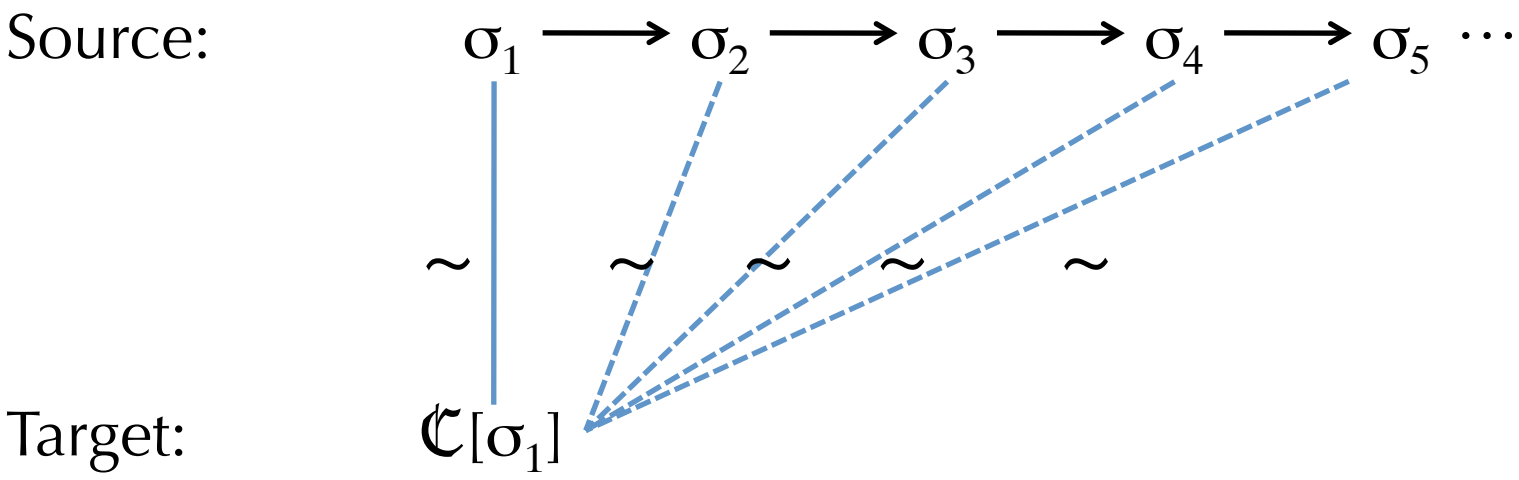
(Solid = assumptions, Dashed = must be shown)

Optional Forward Simulation



A single source-program step is simulated by zero steps in the target.

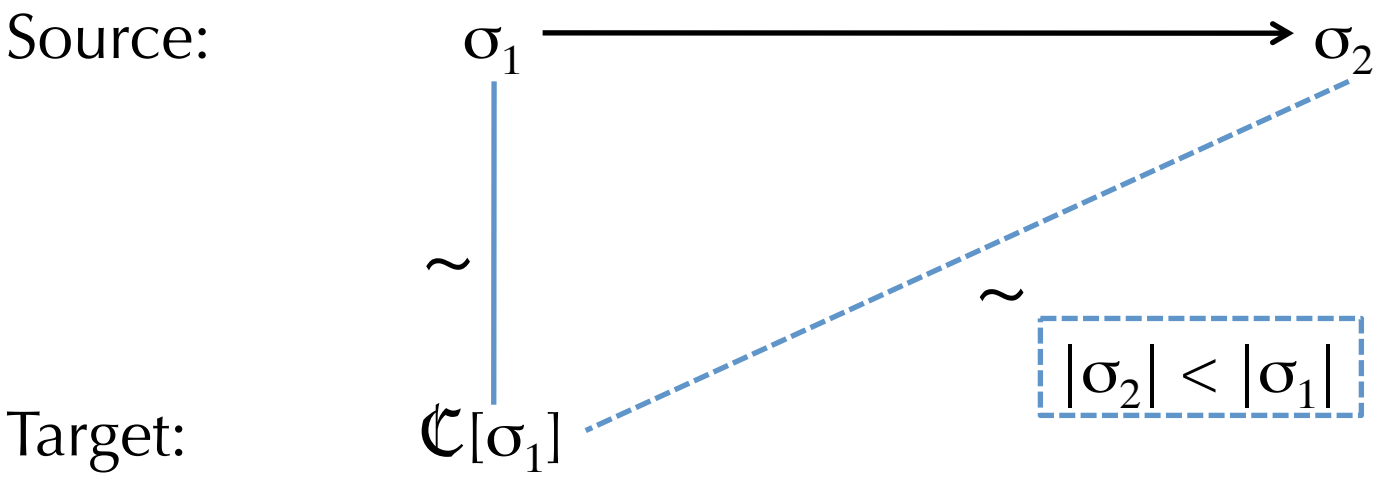
Problem with "Infinite Stuttering"



An infinite sequence of source transitions can be "simulated" by 0 transitions in the target!

(This simulation doesn't preserve nontermination.)

Solution: Disallow such "trivial" simulations



Equip the source language with a measure $|\sigma|$ and require that $|\sigma_2| < |\sigma_1|$.

The measure can't decrease indefinitely, so the target program must either take a step or the source must terminate.

The target diverges if the source program does.

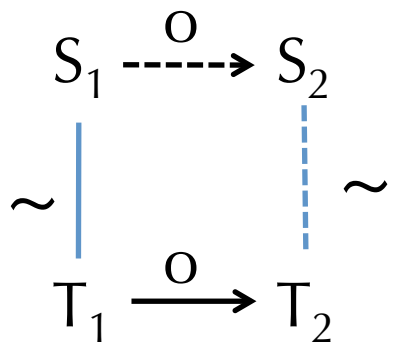
Is Backward Simulation Hopeless?

- Suppose the source & target languages are the same.
  - So they share the same definition of program state.
- Further suppose that the steps are very "small".
  - Abstract machine (i.e. no "complex" instructions).
- Further suppose that "compilation" is only a very minor change.
  - add or remove a single instruction
  - substitute a value for a variable

- Then: backward simulation is more achievable
  - it's easier to invent the "decompilation" function because the "compilation" function is close to trivial

- Happily: This is the situation for many LLVM optimizations

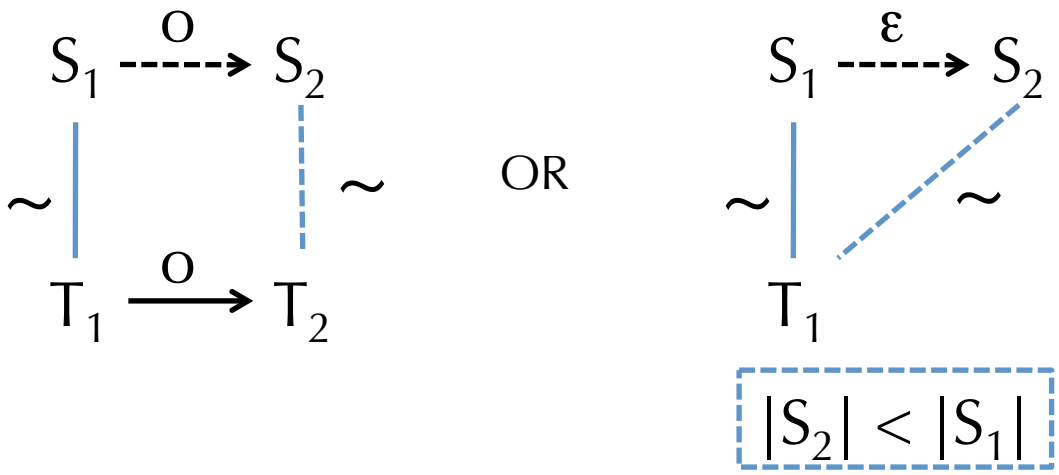
Lock-Step Backward Simulation



o is either an "observable event" or a "silent event"
o ::= e | ε

Example use: proving variable substitution correct.

Right-Option Backward Simulation

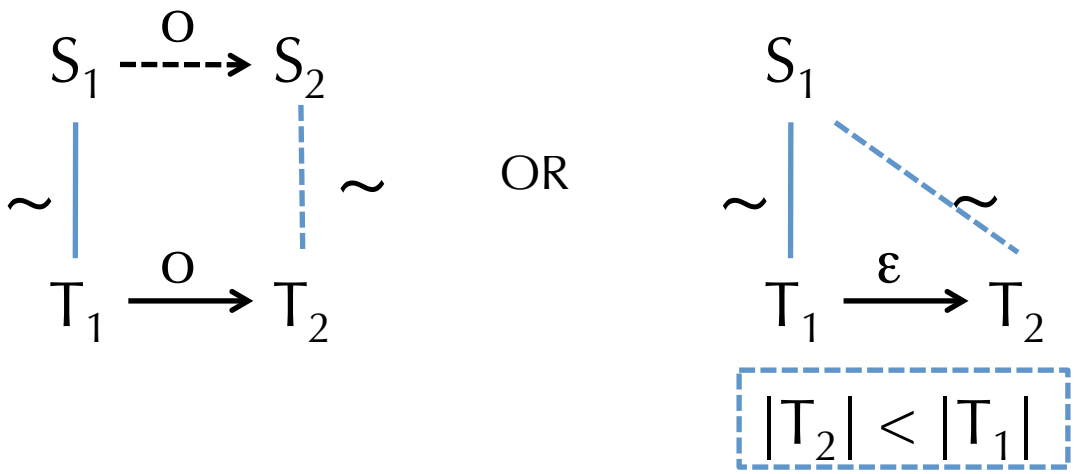


- Either:
  - the source and target are in lock-step simulation.

  Or
  - the source takes a silent transition to a smaller state

  Example use: removing an instruction in the target.

Left-Option Backward Simulation



- Either:
  - the source and target are in lock-step simulation.

  Or
  - the target takes a silent transition to a smaller state

  Example use: adding an instruction to the target.

Verifying optimizations at the LLVM level of abstraction.

EXAMPLE: VELLVM

Step 1: Define LLVM IR Semantics

- Essentially: define an interpreter for LLVM IR code

- But: more complex than the LLVMlite we use in class
  - Aggregate / Structured data
  - Undefined behaviors
  - Nondeterminism

- So: can't be just an interpreter
  - Semantics is given by a relation

Other Parts of the LLVM IR

```
op      ::= %uid | constant | undef          Operands
bop     ::= add | sub | mul | shl | …        Operations
cmpop   ::= eq | ne | slt | sle | …          Comparison

insn ::=
  |  %uid = alloca ty                        Stack Allocation
  |  %uid = load ty op1                      Load
  |  store ty op1, op2                       Store
  |  %uid = getelementptr ty op1 …           Address Calculation
  |  %uid = call rt fun(…args…)              Function Calls
  |  …

phi ::=
  |  Φ[op1;lbl1]...[opn;lbln]

terminator ::=
  |  ret %ty op
  |  br op label %lbl1, label %lbl2
  |  br label %lbl
```

Sources of Undefined Behavior

Target-dependent Results

- Uninitialized variables:

```
%v = add i32 %x, undef
```

- Uninitialized memory:

```
%ptr = alloca i32
%v = load (i32*) %ptr
```

- Ill-typed memory usage

Nondeterminism

Fatal Errors

- Out-of-bounds accesses
- Access dangling pointers
- Free invalid pointers
- Invalid indirect calls

Stuck States

Sources of Undefined Behavior

Target-dependent Results

- Uninitialized variables:

```
%v = add i32 %x, undef
```

- Uninitialized memory:

```
%ptr = alloca i32
%v = load (i32*) %ptr
```

- Ill-typed memory usage

Nondeterminism

Defined by a predicate on the program configuration.

A program configuration is *stuck* if there is no transition it can make.

Stuck(f, σ) = BadFree(f, σ)
˅ BadLoad(f, σ)
˅ BadStore(f, σ)
˅ …
˅ …

Stuck States

LLVM's memory model

```
%ST = type {i10,[10 x i8*]}
```

High-level Representation

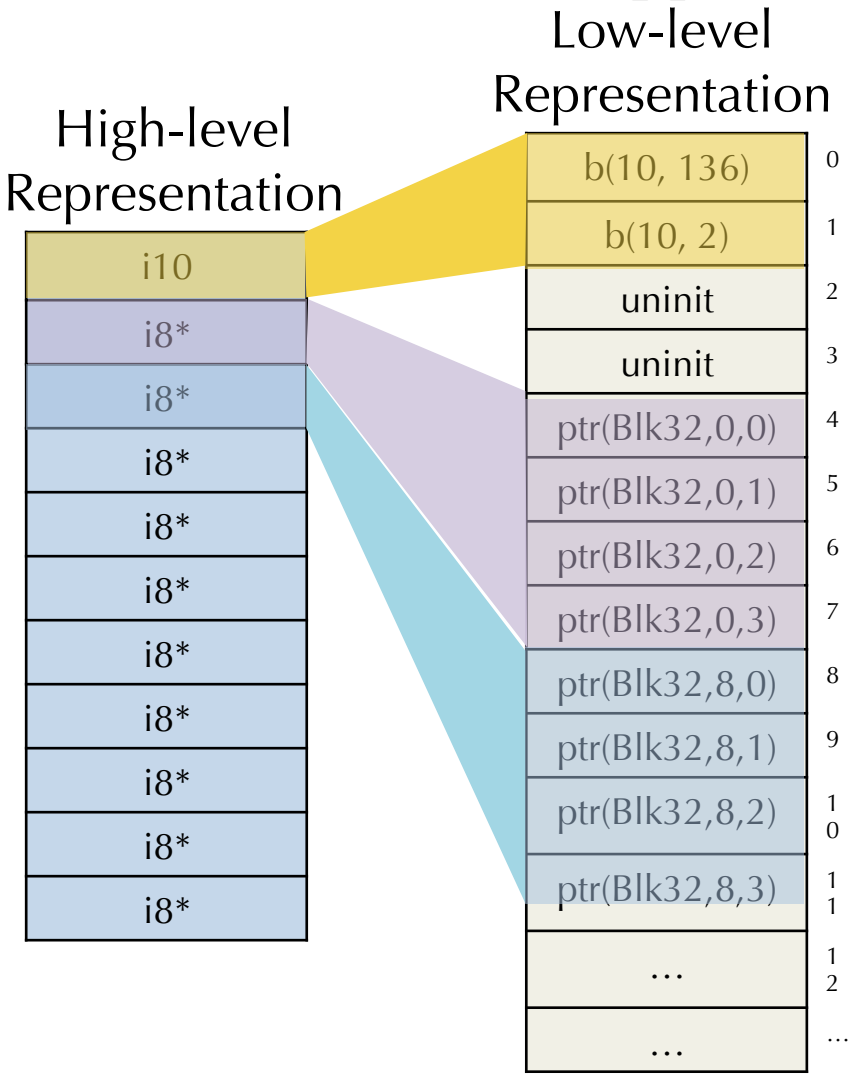
i10
i8*
i8*
i8*
i8*
i8*
i8*
i8*
i8*
i8*
i8*
i8*

- Manipulate structured types.

```
%val = load %ST* %ptr
…
store %ST* %ptr, %new
```

LLVM's memory model

```
%ST = type {i10,[10 x i8*]}
```

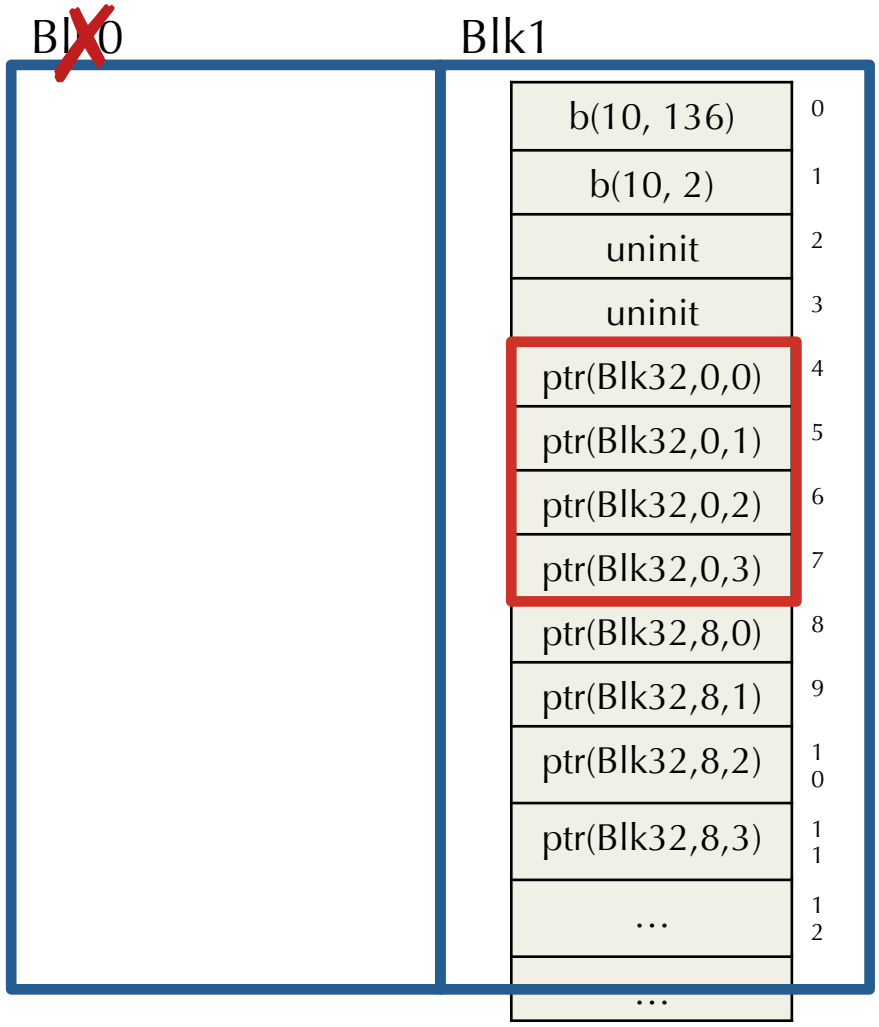


- Manipulate structured types.

```
%val = load %ST* %ptr
…
store %ST* %ptr, %new
```

- Semantics is given in terms of byte-oriented low-level memory.
  - padding & alignment
  - physical subtyping

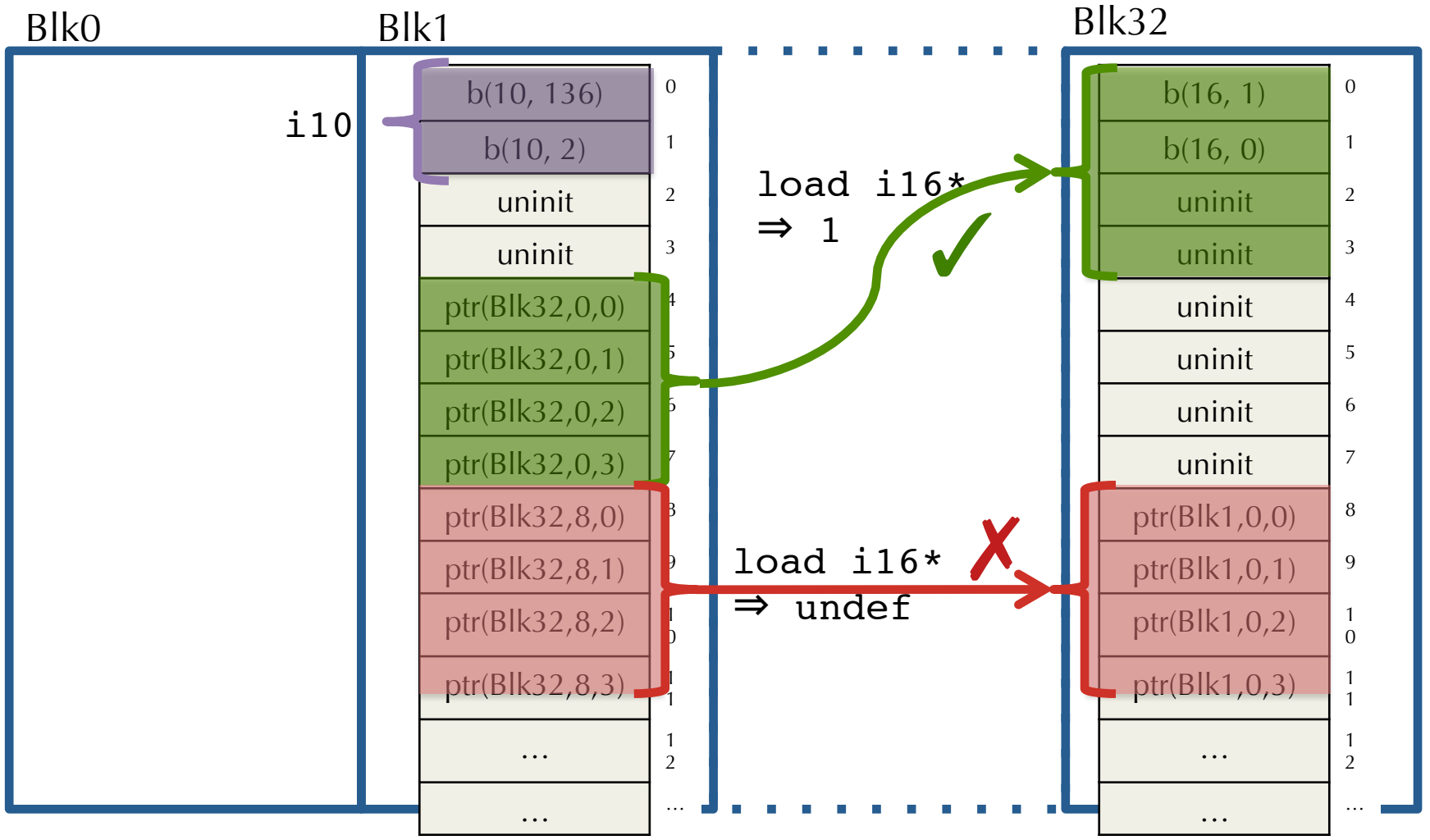
Adapting CompCert's Memory Model



- Data lives in blocks
- Represent pointers abstractly
  - block + offset
- Deallocate by invalidating blocks
- Allocate by creating new blocks
  - infinite memory available

Dynamic Physical Subtyping

[Nita, et al. *POPL '08*]



undef

- What is the value of `%y` after running the following?

```
%x = or i8 undef, 1
%y = xor i8 %x %x
```

- One plausible answer: 0
- Not LLVM's semantics!

  (LLVM is more liberal to permit more aggressive optimizations)

undef

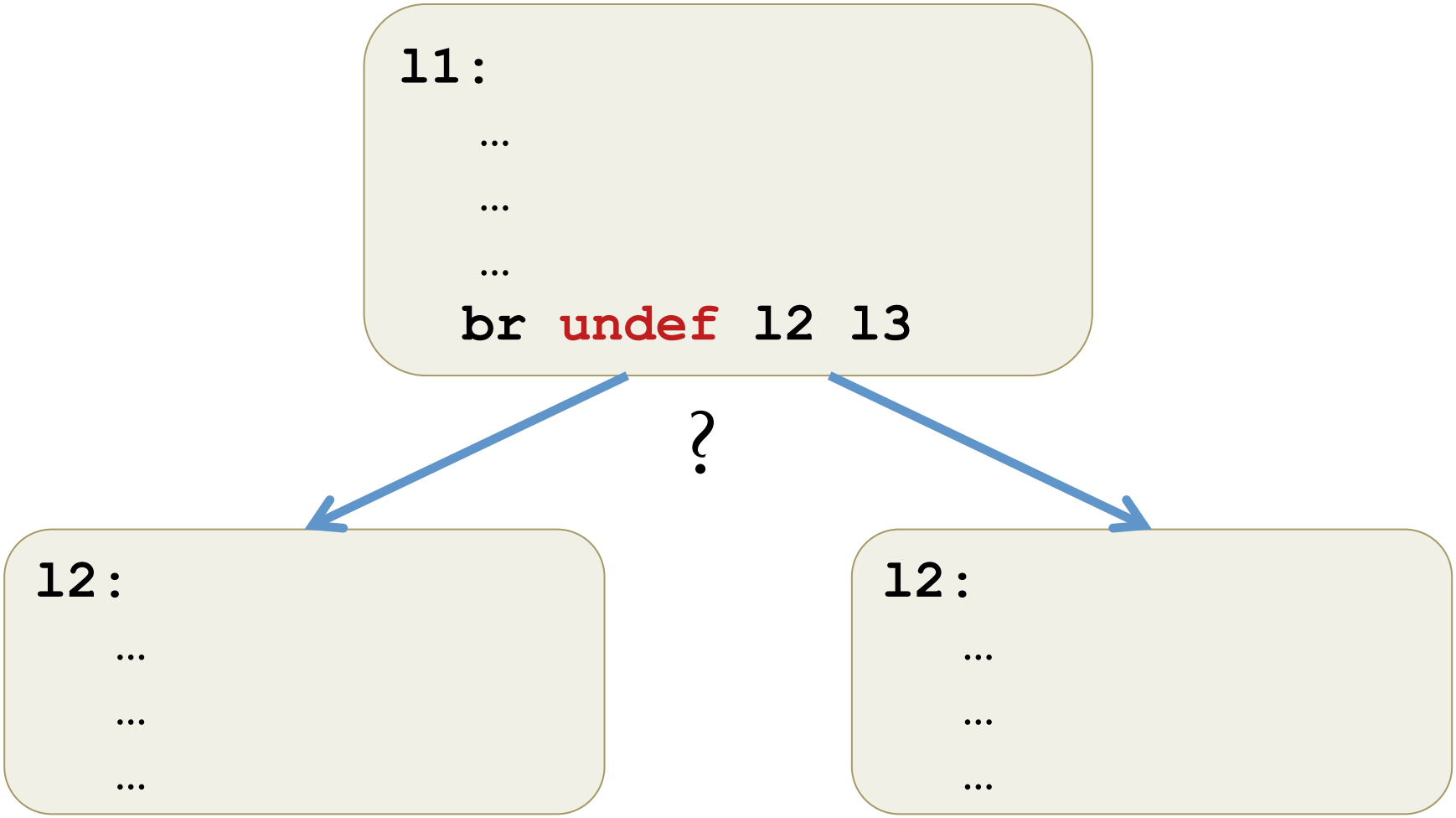
- Partially defined values are interpreted *nondeterministically* as sets of possible values:

```
%x = or i8 undef, 1
%y = xor i8 %x %x
```

$$[\![\texttt{i8 undef}]\!] = \{0,\ldots,255\}$$
$$[\![\texttt{i8 1}]\!] = \{1\}$$

$$[\![\texttt{\%x}]\!] = \{a \texttt{ or } b \mid a \in [\![\texttt{i8 undef}]\!],\ b \in [\![\texttt{1}]\!]\}$$
$$= \{1,3,5,\ldots,255\}$$
$$[\![\texttt{\%y}]\!] = \{a \texttt{ xor } b \mid a \in [\![\texttt{\%x}]\!],\ b \in [\![\texttt{\%x}]\!]\}$$
$$= \{0,2,4,\ldots,254\}$$

Nondeterministic Branches



LLVM_{ND} Operational Semantics

- Define a transition relation:

$$f \vdash \sigma_1 \longmapsto \sigma_2$$

  - f is the program
  - σ is the program state: pc, locals(δ), stack, heap
- Nondeterministic
  - δ maps local `%uids` to sets.
  - Step relation is nondeterministic
- Mostly straightforward (given the heap model)
  - Another wrinkle: phi-nodes executed atomically

Need for Atomic Phi-node Updates

```
blk:
      %x = phi i32 [ %z, %blk ], [ 0, %pred ]
      %z = phi i32 [ %x, %blk ], [ 1, %pred ]
      %b = icmp leq %x %z
      br %b %blk %succ
```

Operational Semantics

	Small Step	Big Step
Nondeterministic	$LLVM_{ND}$	
Deterministic		

Deterministic Refinement

	Small Step	Big Step
Nondeterministic	$LLVM_{ND}$	
Deterministic	$LLVM_{D}$	

Instantiate ‘undef’ with default value (0 or null) ⇒ deterministic.

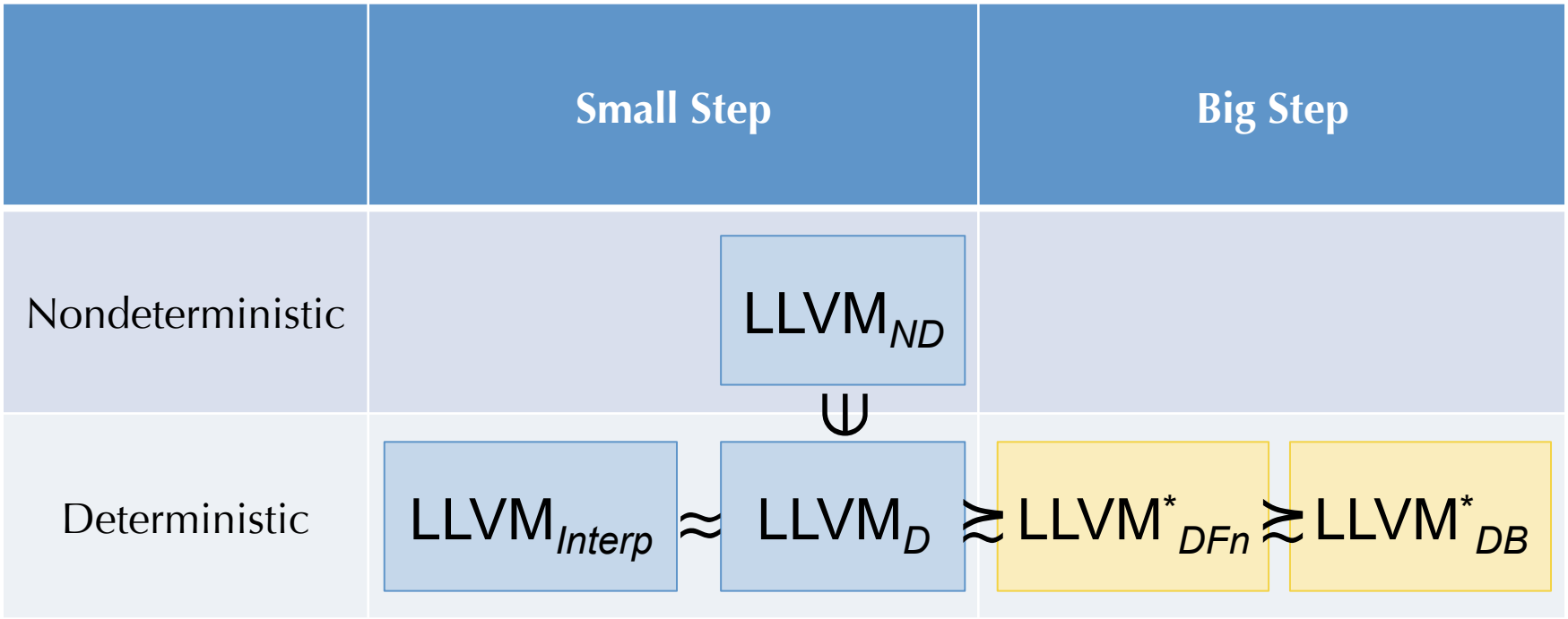
Big-step Deterministic Refinements

	Small Step	Big Step
Nondeterministic	LLVM_{ND}	
Deterministic	$\text{LLVM}_{Interp} \approx \text{LLVM}_{D}$	

Bisimulation up to "observable events":

- external function calls

Big-step Deterministic Refinements



Simulation up to "observable events":

- useful for encapsulating behavior of function calls
- large step evaluation of basic blocks

[Tristan, et al. *POPL '08*, Tristan, et al. *PLDI '09*]