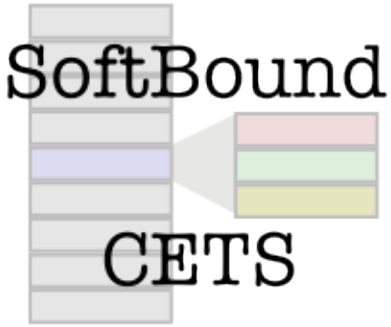
Jianzhou Zhao
Steve Zdancewic
Milo Martin
University of Pennsylvania

Santosh Nagarakatte
University of Pennsylvania/
Rutgers University

Genesis

Using LLVM since 2007

Pointer-based checking for memory Safety in the LLVM IR



[Nagarakatte, et al. *PLDI '09*]

Can we trust the implementations of the instrumentation?

Can we trust the implementation of the compiler?

Compilers are not Always Correct

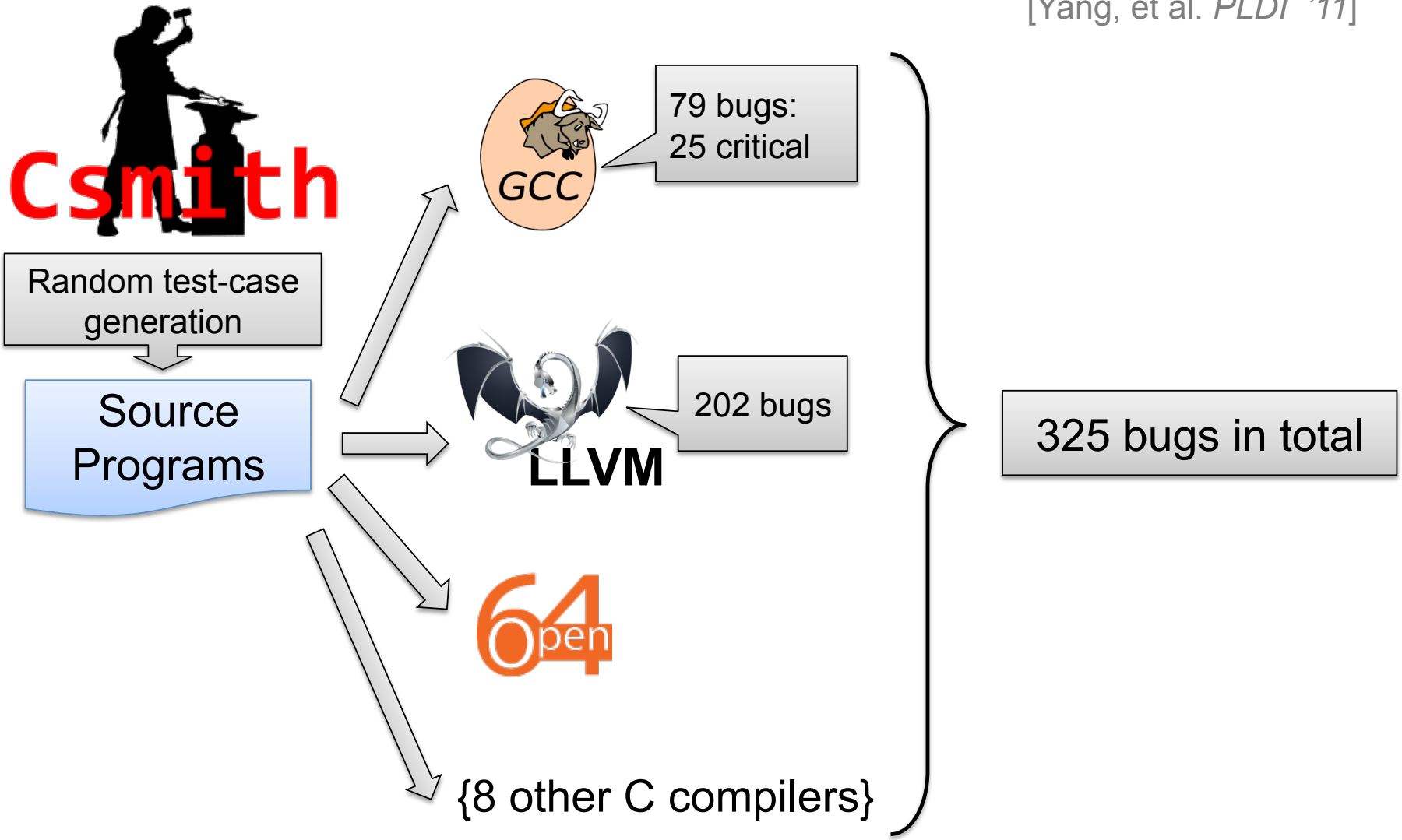
[Yang, et al. *PLDI '11*]

Random test-case generation

Source Programs → Compiler → Transformed Programs

Compilers are not Always Correct

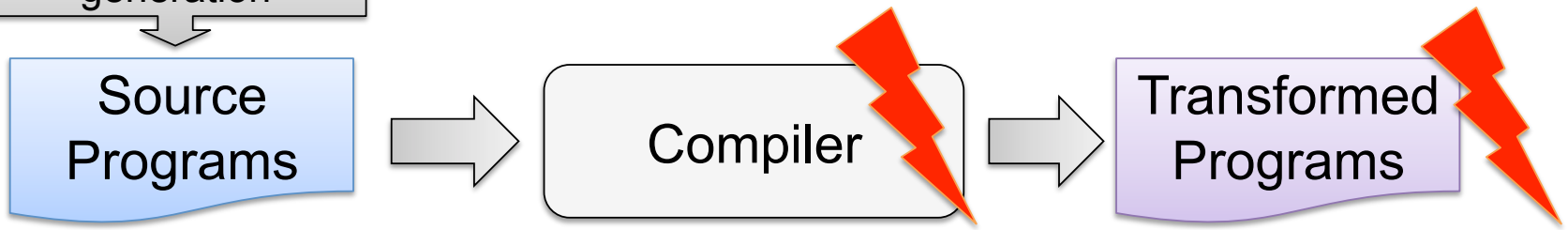
[Yang, et al. *PLDI '11*]



Compilers are not Always Correct

[Yang, et al. *PLDI '11*]

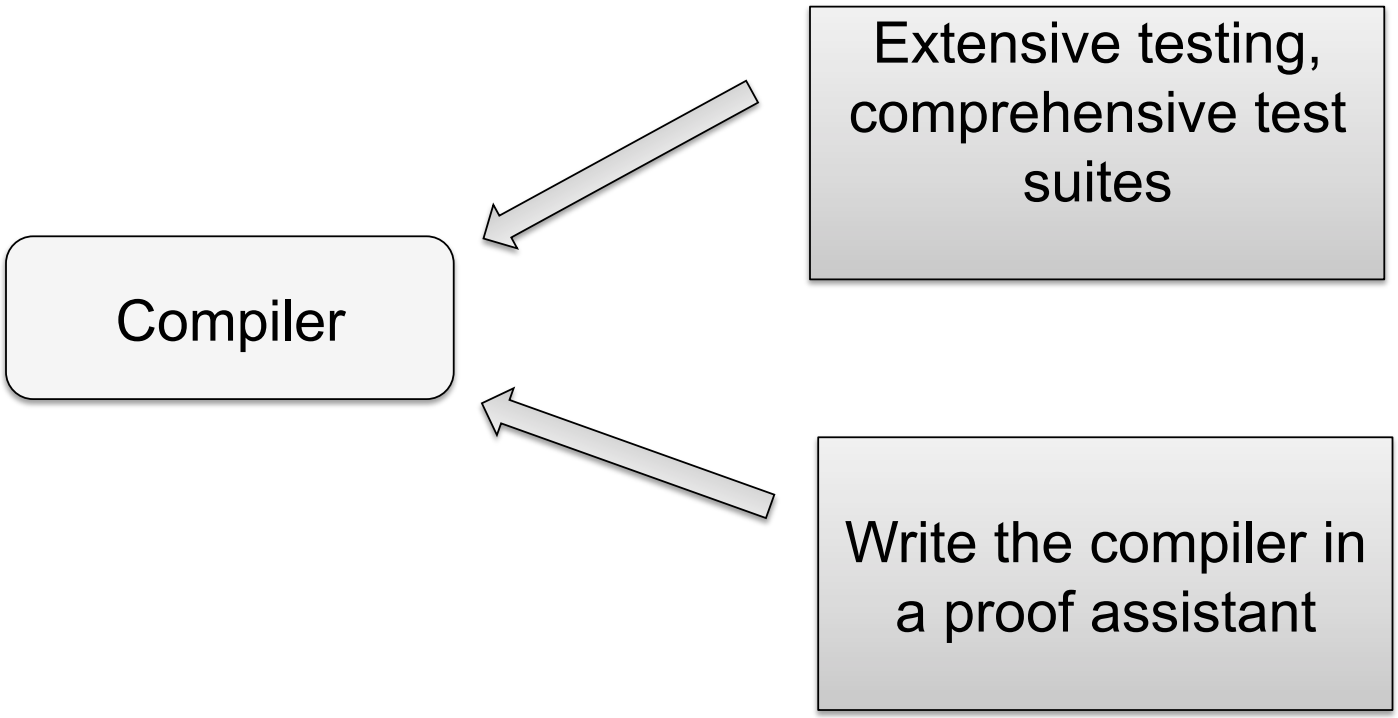
Random test-case generation



Bug kinds:
- Compile-time crash
- Output wrong-code

How can ensure that the compiler implementations are free of errors?

Eliminating Compiler Bugs

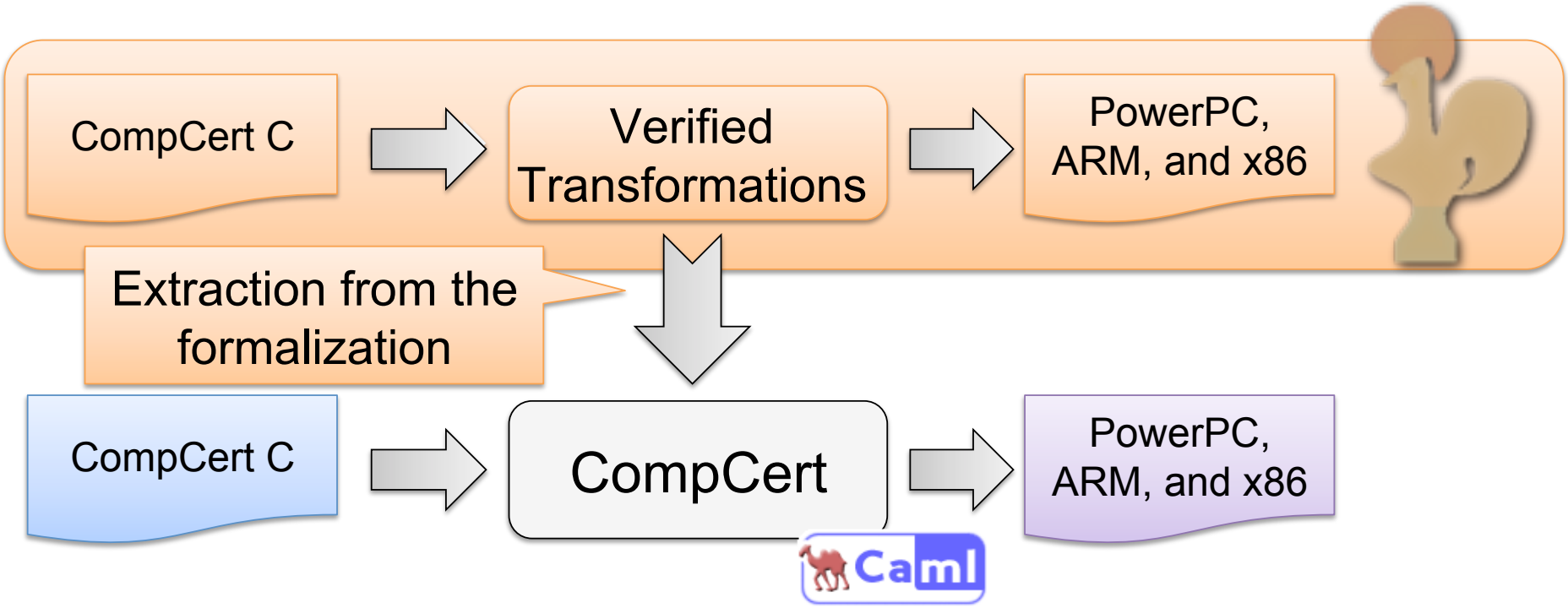


The CompCert Project

http://compcert.inria.fr

[Leroy, et al.]

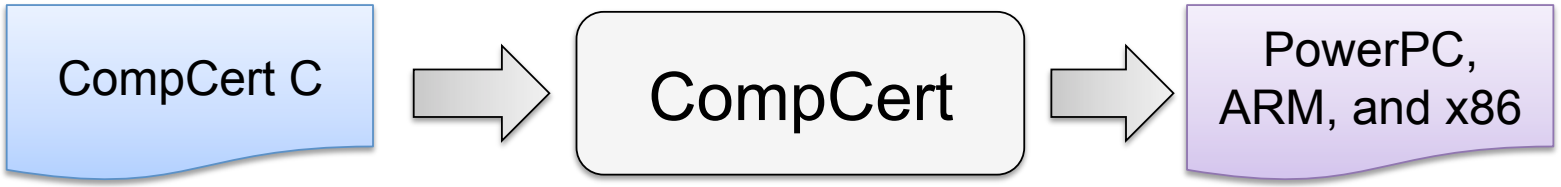
- A realistic C compiler
- Proved semantics-preservation in Coq



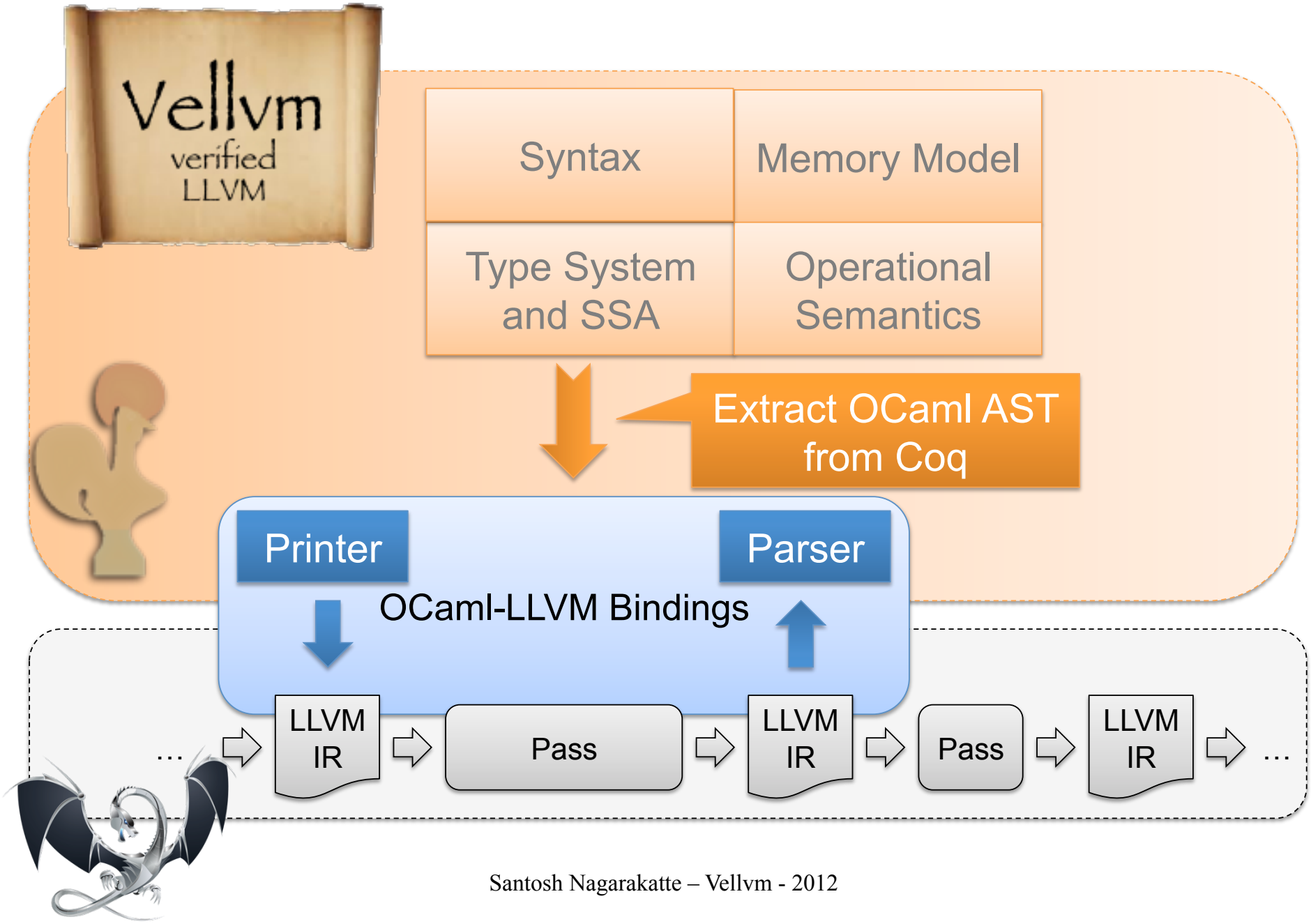
Verified Compilers are Robust

"The apparent ***unbreakability of CompCert*** supports a strong argument that ***developing compiler optimizations within a proof framework***, where safety checks are explicit and machine-checked, ***has tangible benefits*** for compiler users."

Random test-case generation



Can we build LLVM within a Proof Assistant?


Vellvm
verified
LLVM
Syntax
Memory Model
Type System
and SSA
Operational
Semantics
Extract OCaml AST
from Coq
Printer
Parser
OCaml-LLVM Bindings
…
LLVM
IR
Pass
LLVM
IR
Pass
LLVM
IR
…

Vellvm's Contributions

- Formal semantics of LLVM IR (in a Proof Assistant)
  - **Reverse-engineering**

Vellvm's Contributions

- Formal semantics of LLVM IR (in a Proof Assistant)
  - **Reverse-engineering**

Our Contributions

- Formal semantics of LLVM IR (in a Proof Assistant)
  - Reverse-engineering
  - **Typed, SSA**
  - **Non-deterministic, with high-level memory model**
  - **Formalized with proofs in mind**

Our Contributions

- Formal semantics of LLVM IR (in a Proof Assistant)
  - Reverse-engineering
  - Typed, SSA
  - Non-deterministic, with high-level memory model
  - Formalized with proofs in mind
- **Tools for interacting with LLVM**

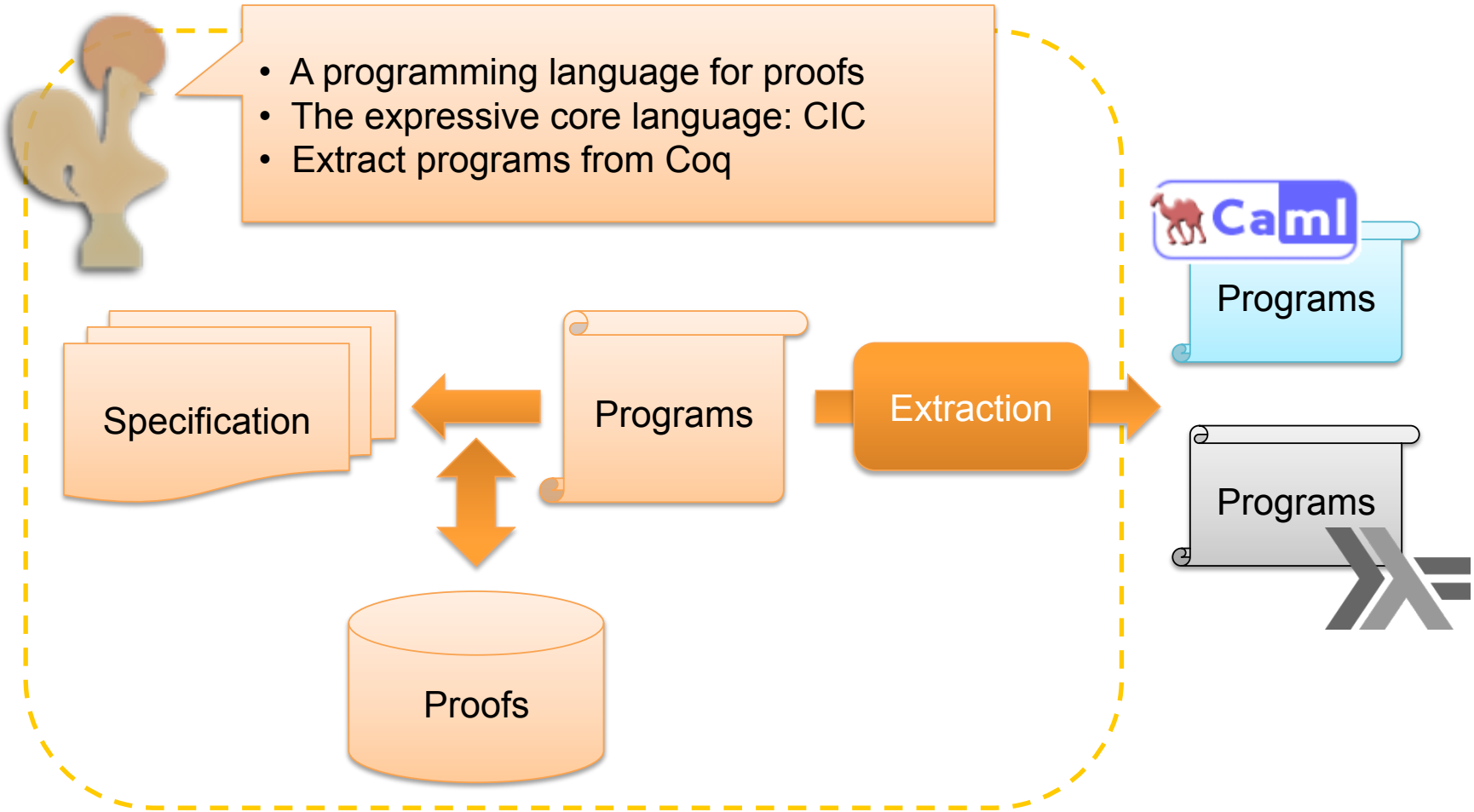
Our Contributions

- Formal semantics of LLVM IR (in a Proof Assistant)
  - Reverse-engineering
  - Typed, SSA
  - Non-deterministic, with high-level memory model
  - Formalized with proofs in mind
- Tools for interacting with LLVM
- Application of the semantics
  - **Verified SoftBound Transformation**
  - **Verified Mem2reg Transformation**

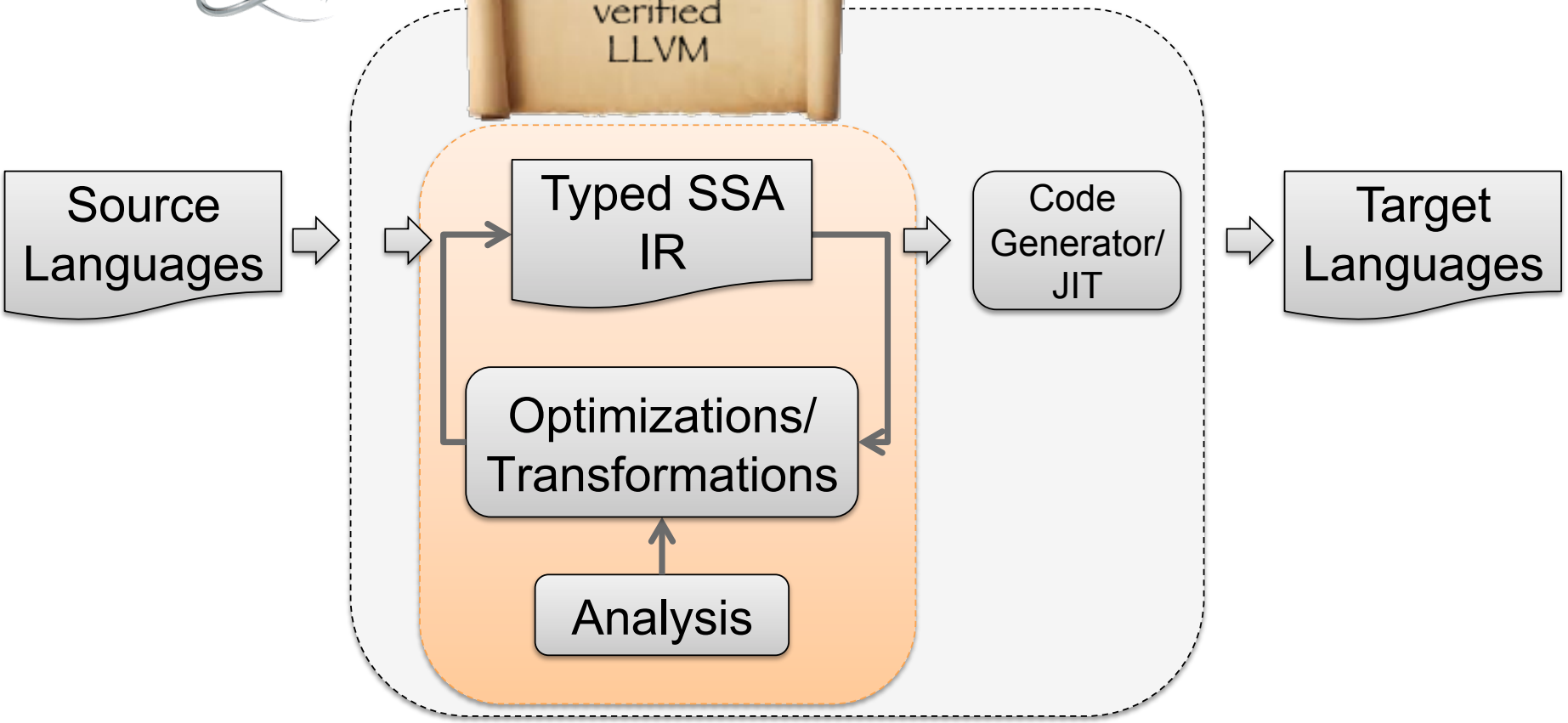
Outline

- Background
- Semantics of the LLVM IR
- Applications of the semantics
  - Tool chain
  - Verified SoftBound transformation
  - Verified Mem2reg optimization
- Lesson learned & Conclusion

The Coq Proof Assistant



- Formal semantics
- Verified transformations



Vellvm

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

define %ST @foo(%ST %st0, i8* %ptr){
entry:
  %p = malloc %ST, i32 1
  store %ST %st0, %p
  %r = gep %ST* %p, i32 0, i32 0
  store i10 648, %r
  %s = gep %ST* %p, i32 0, i32 1, i32 0
  store i8* %ptr, %s
  br undef %loop %succ

loop:
  %x = phi i32 [%z, %loop], [0, %entry]
  %z = phi i32 [%x, %loop], [1, %entry]
  %b = icmp leq %x %z
  br %b %loop %succ

succ:
  %st1 = load (%ST*) %p
  free (%ST*) %p
  ret %ST %st1
}
```

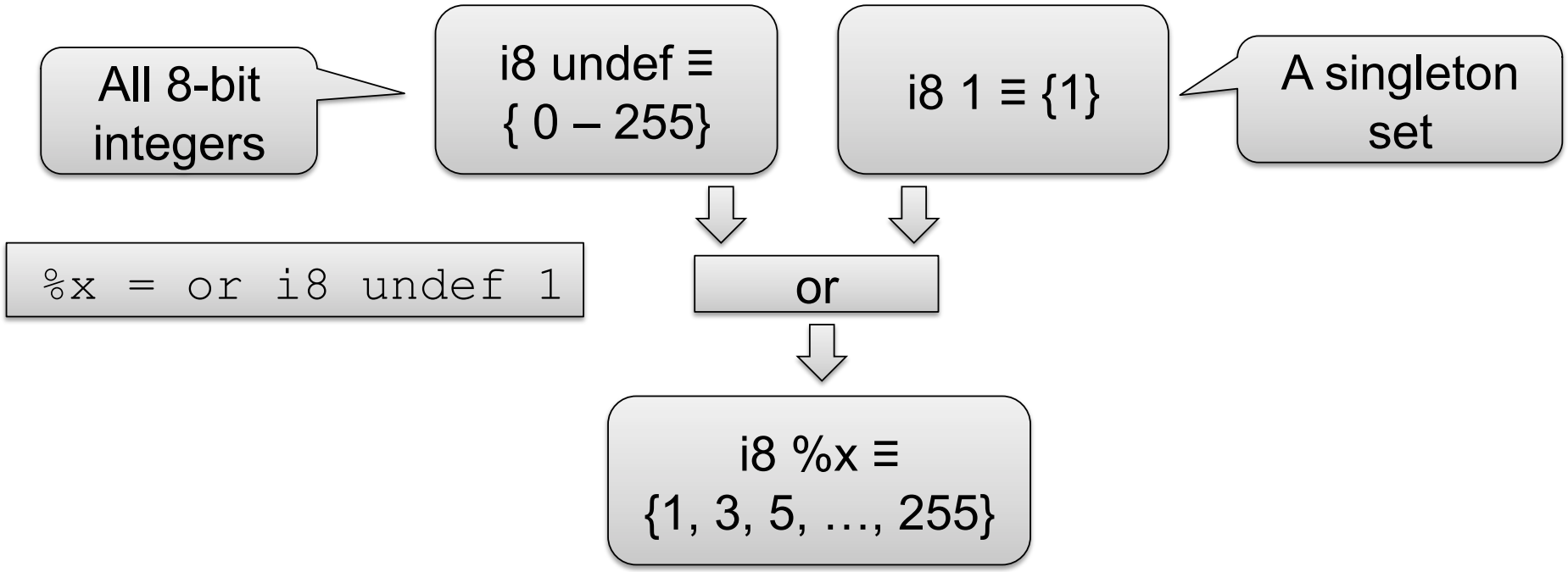
- LLVM assembly language
- Typed
- SSA
- First-class high-level data type
  - Store/Load
  - GEP (accessing sub-fields)

Vellvm formalizes domination analysis to reason about SSA

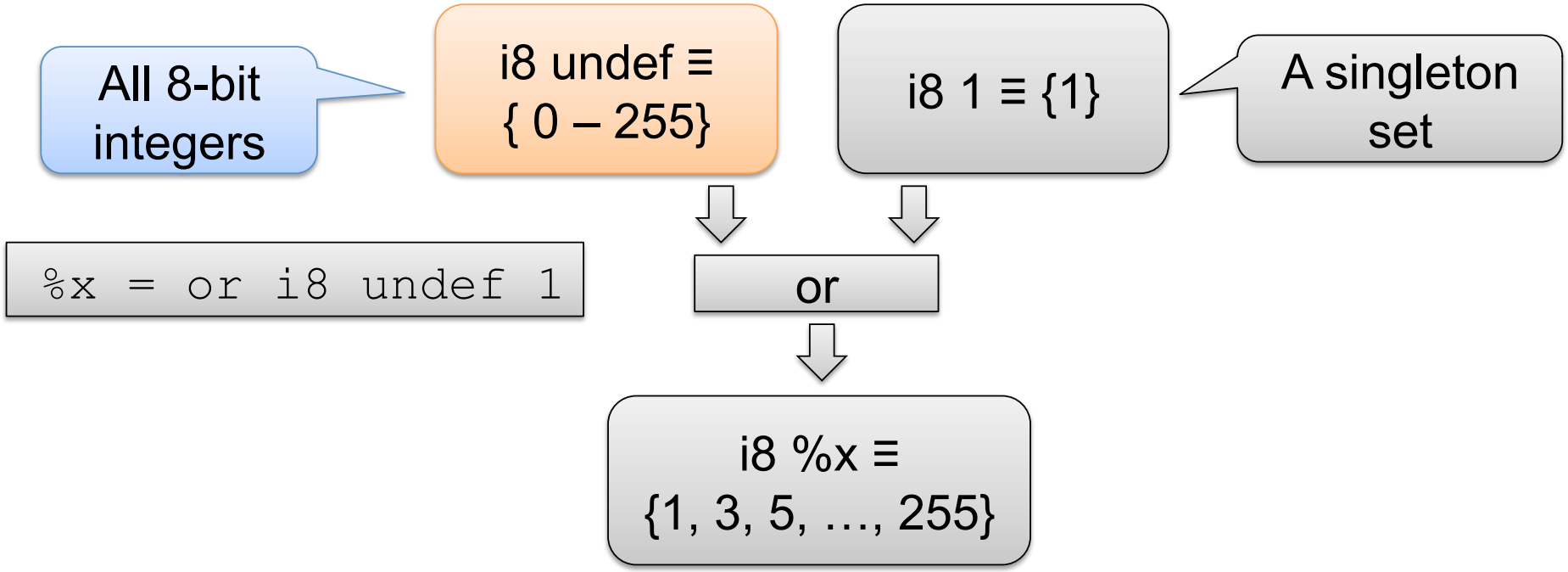
LLVM operational semantics is non-deterministic.

Non-determinism with Undef values

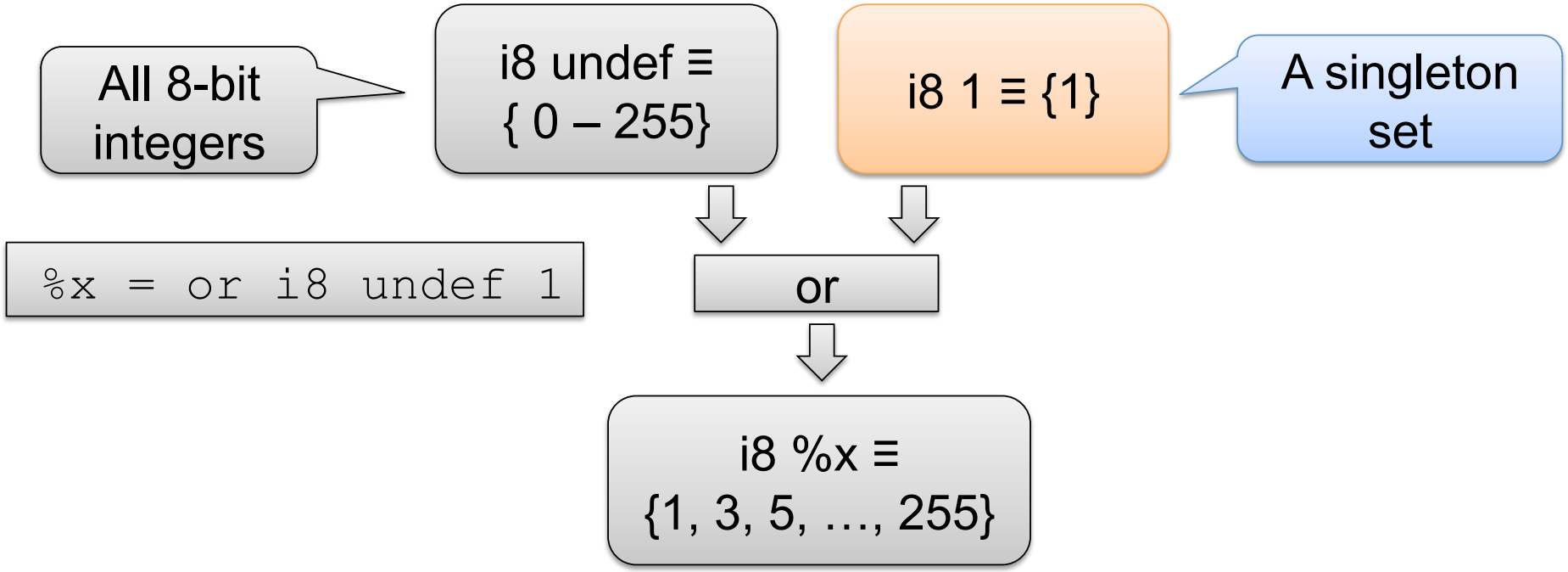
An LLVM value is a set of values.



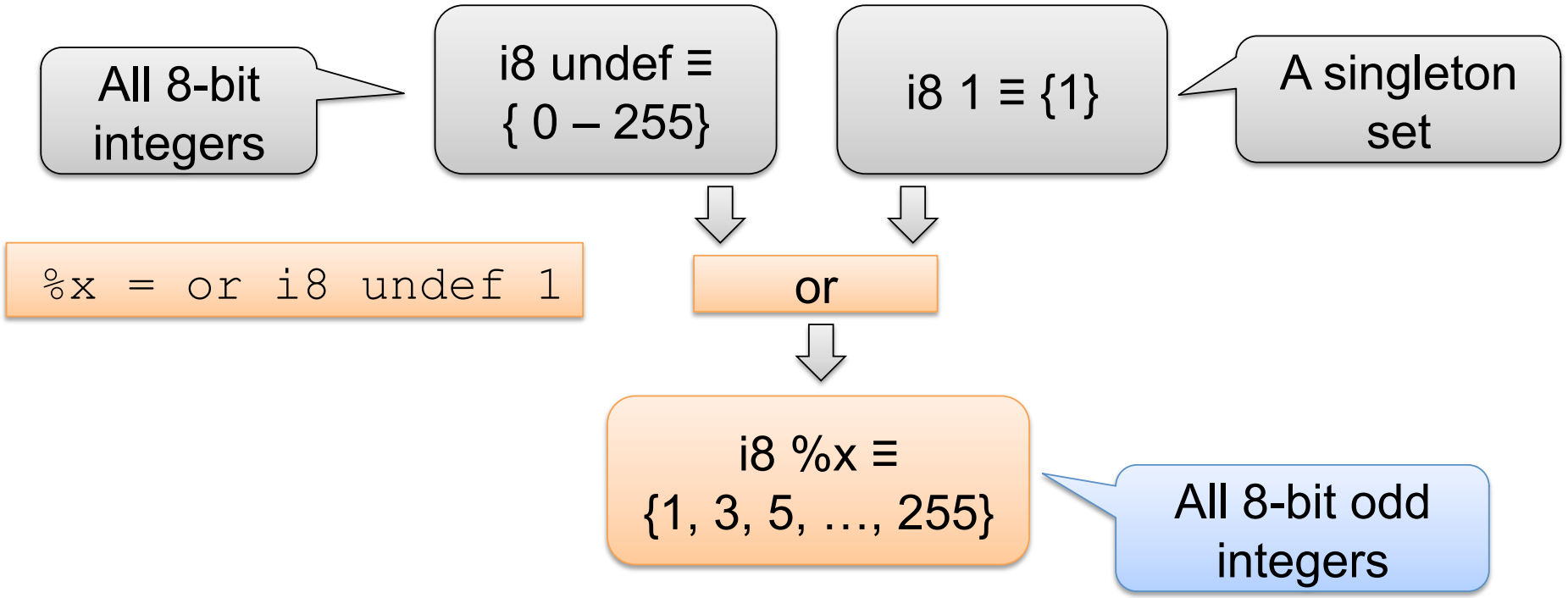
An LLVM value is a set of values.



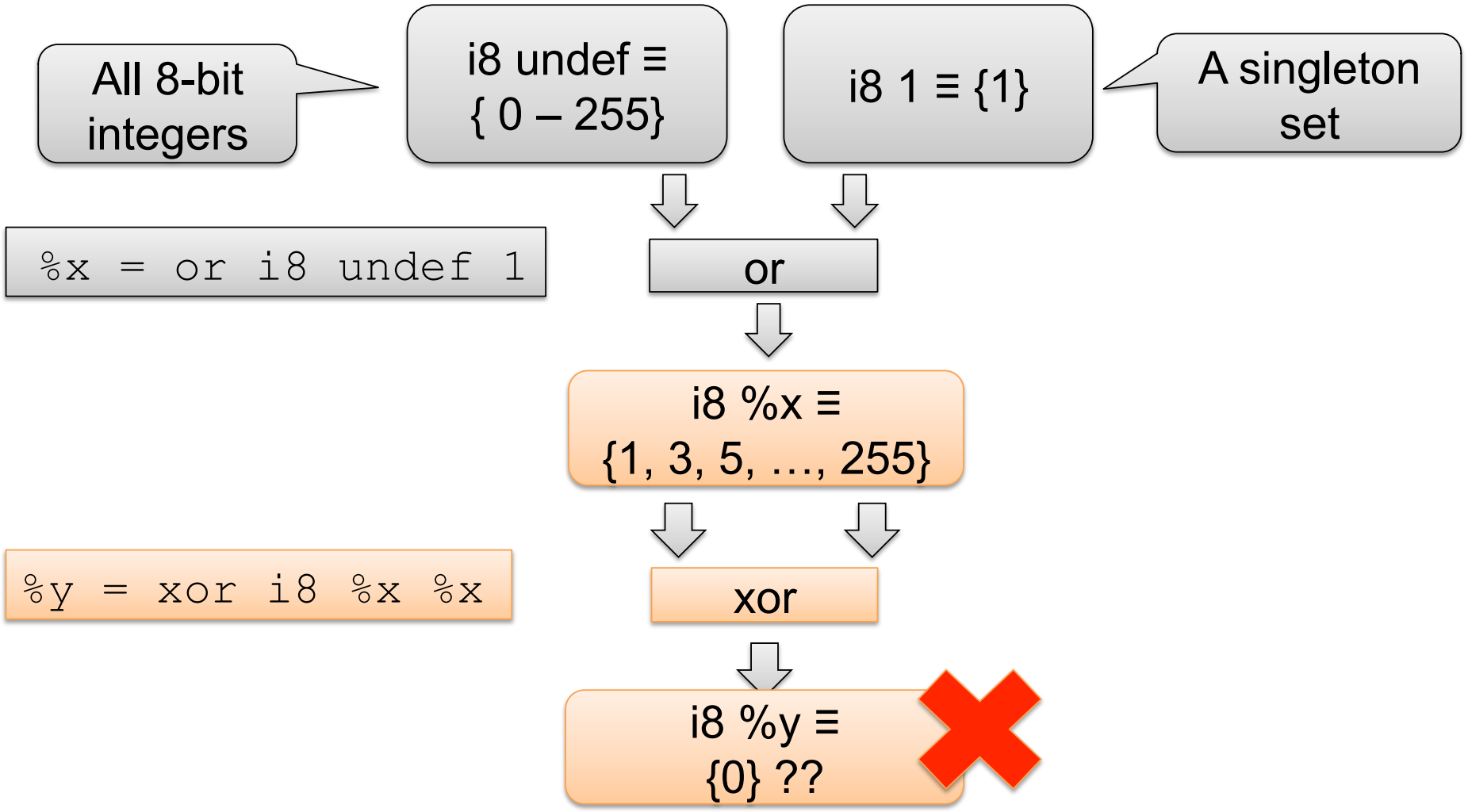
An LLVM value is a set of values.



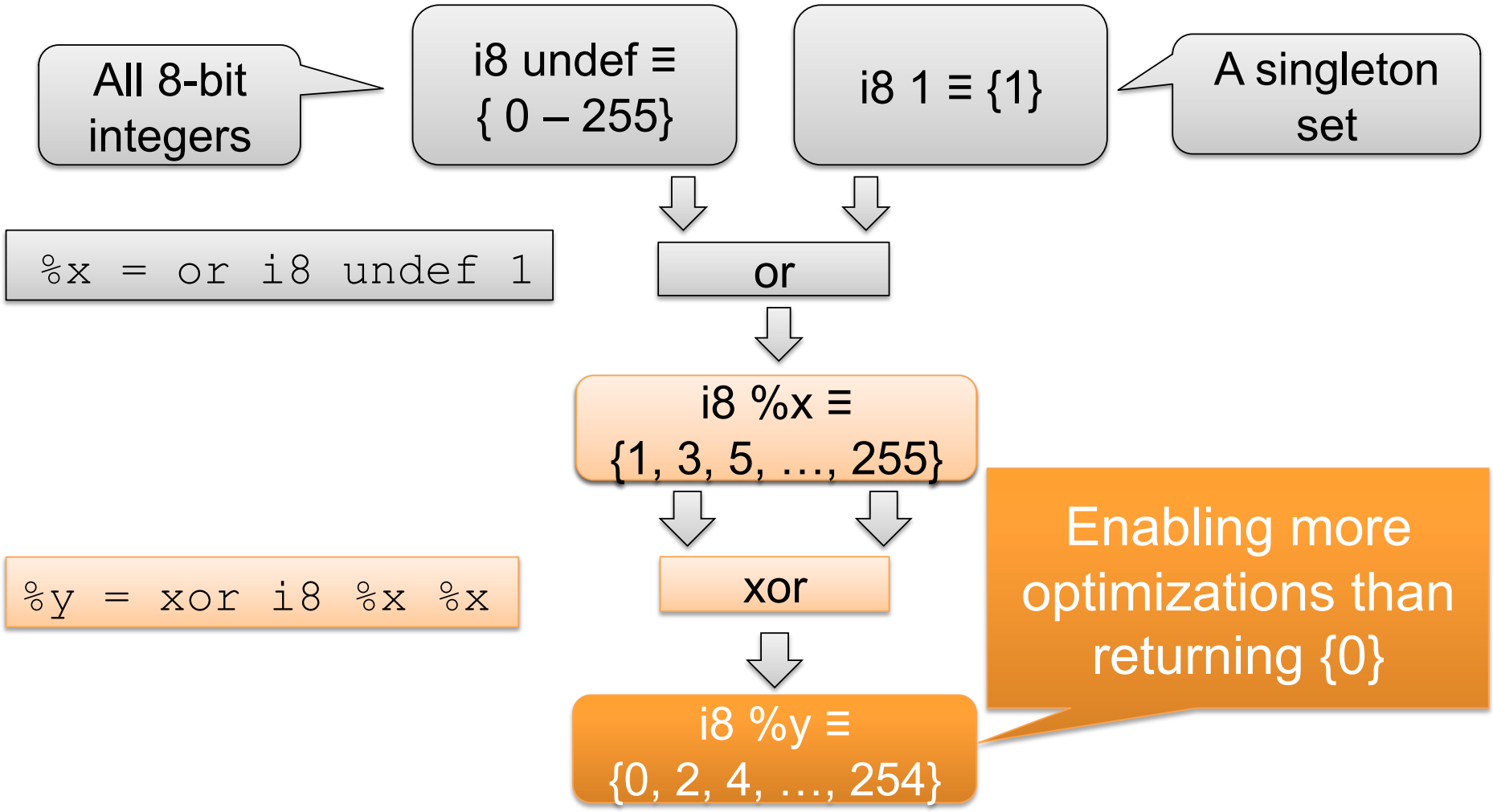
An LLVM value is a set of values.



One more example…



One more example…

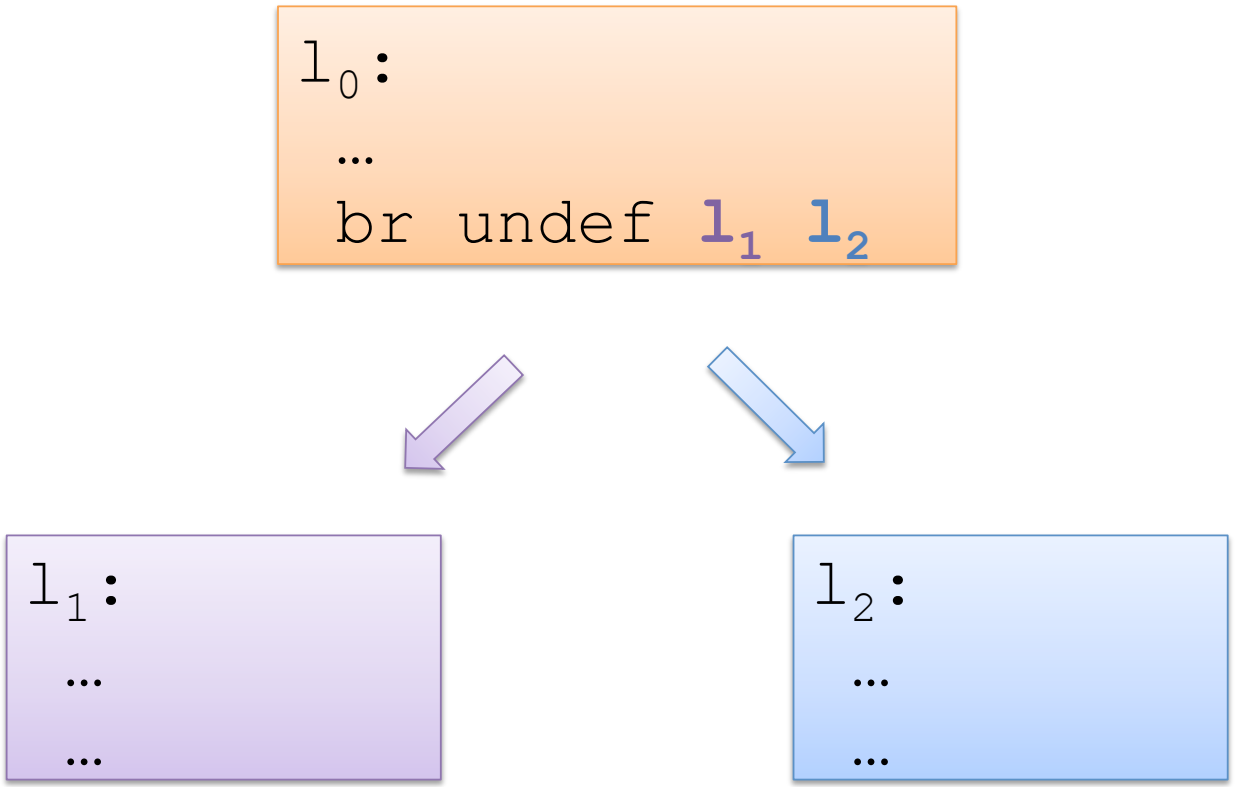


Non-deterministic Branching

```
l0:
 …
 br undef l1 l2
```

??

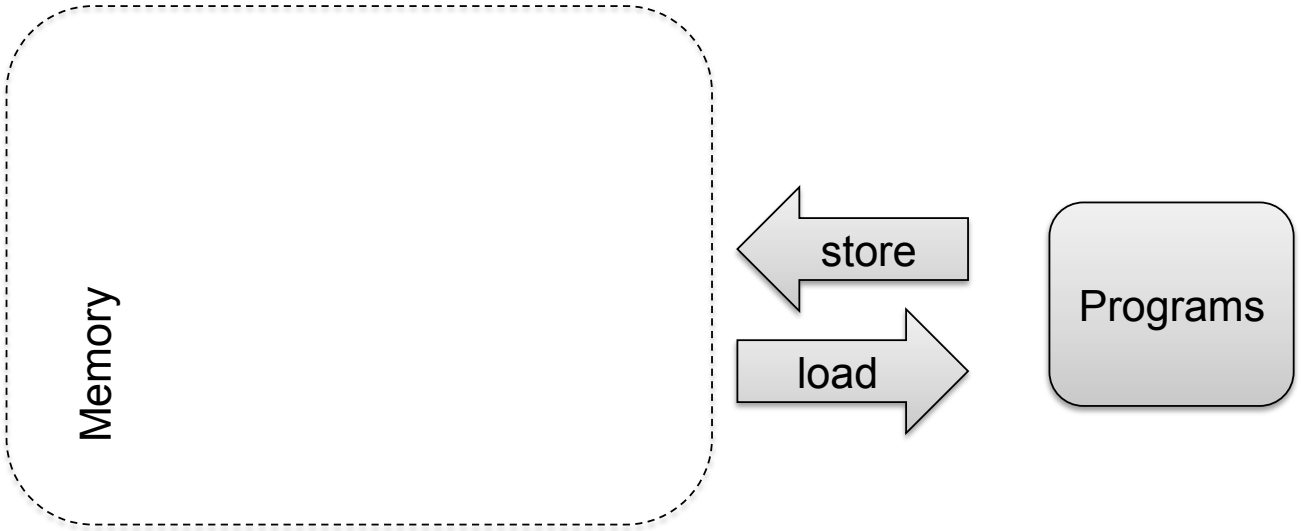
Non-deterministic Branching



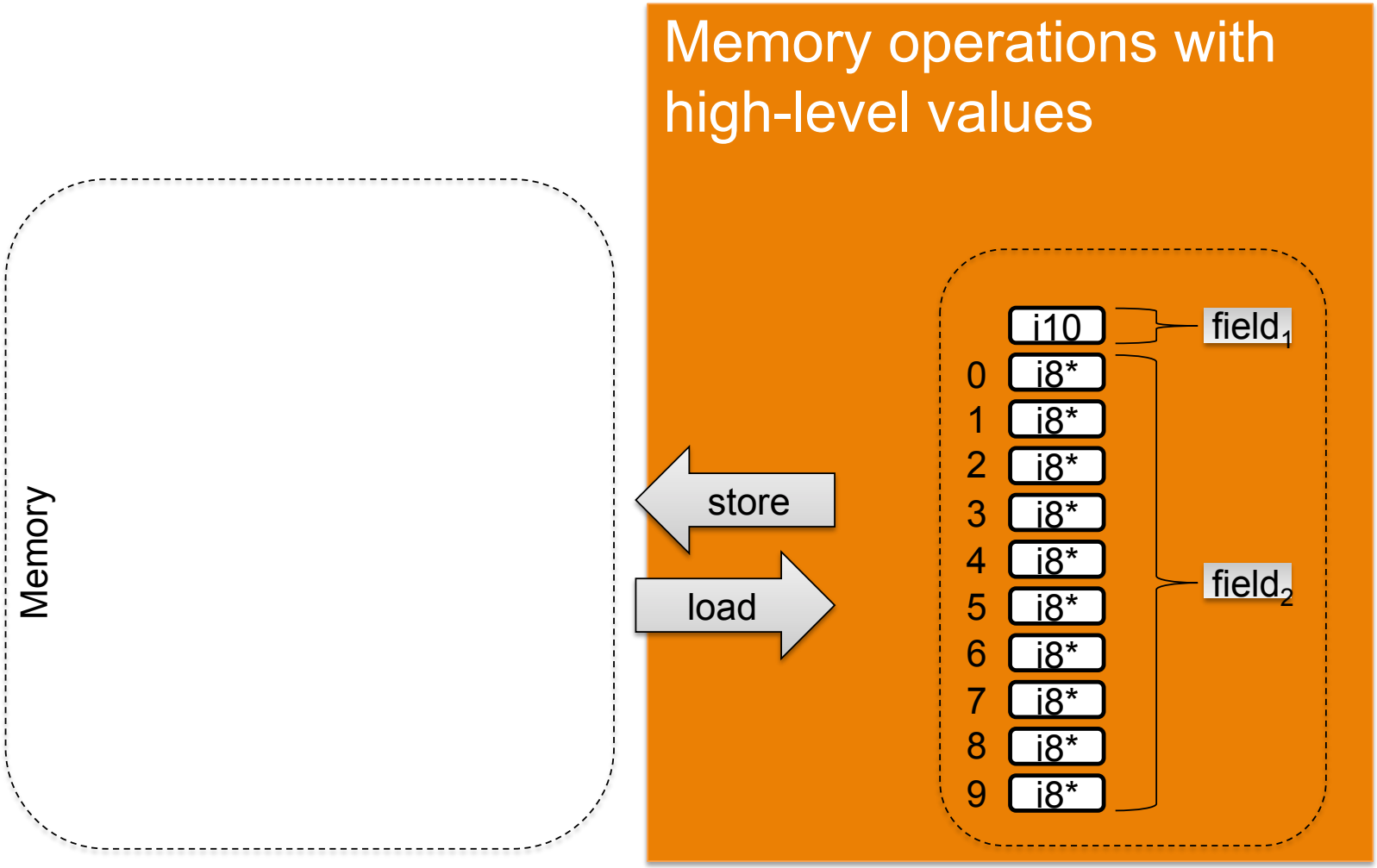
Memory Model

- Reason about memory operations
  - Without being specific to the platform/memory manager

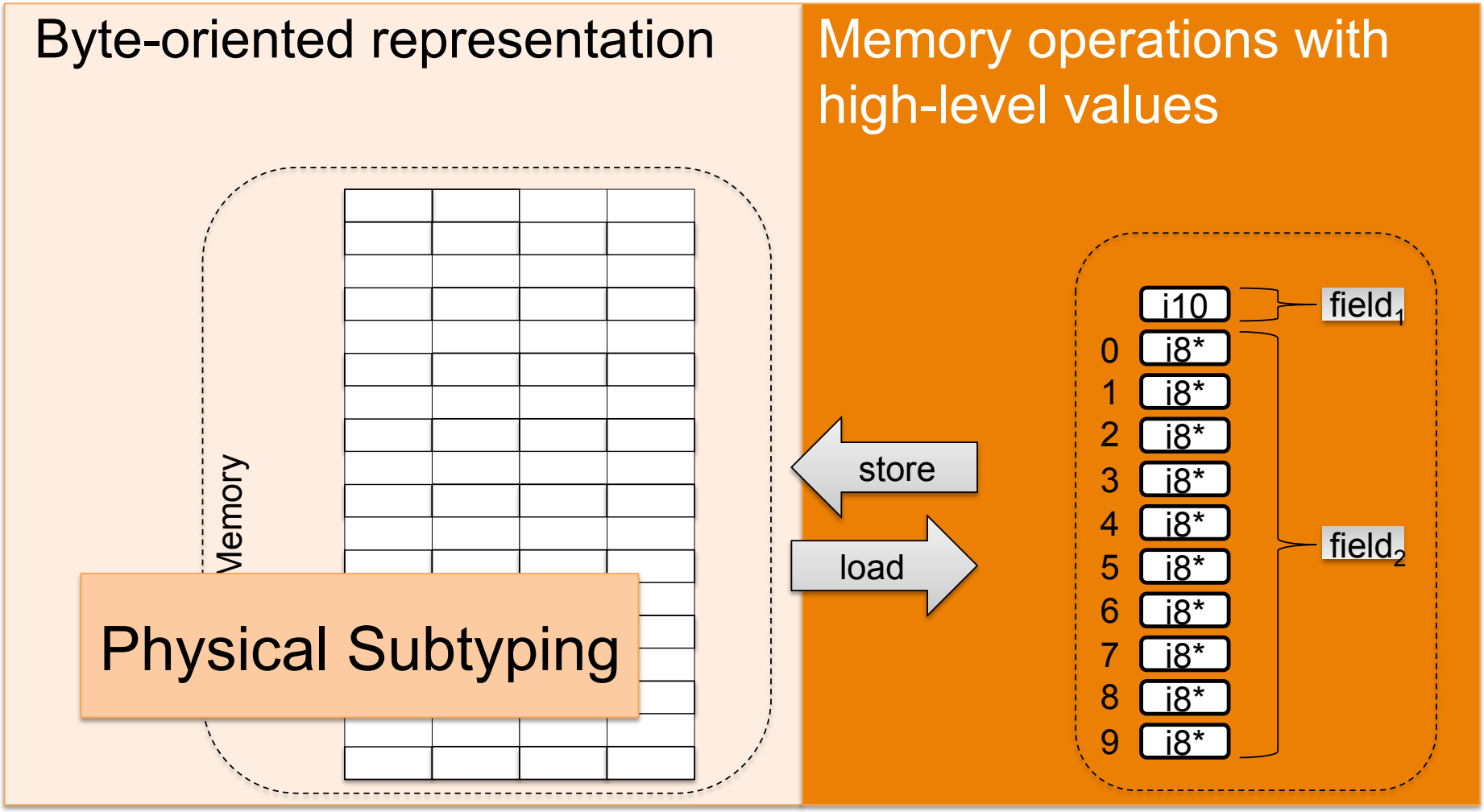
Being pragmatic, we reuse and extend CompCert's memory model



The LLVM Memory Model

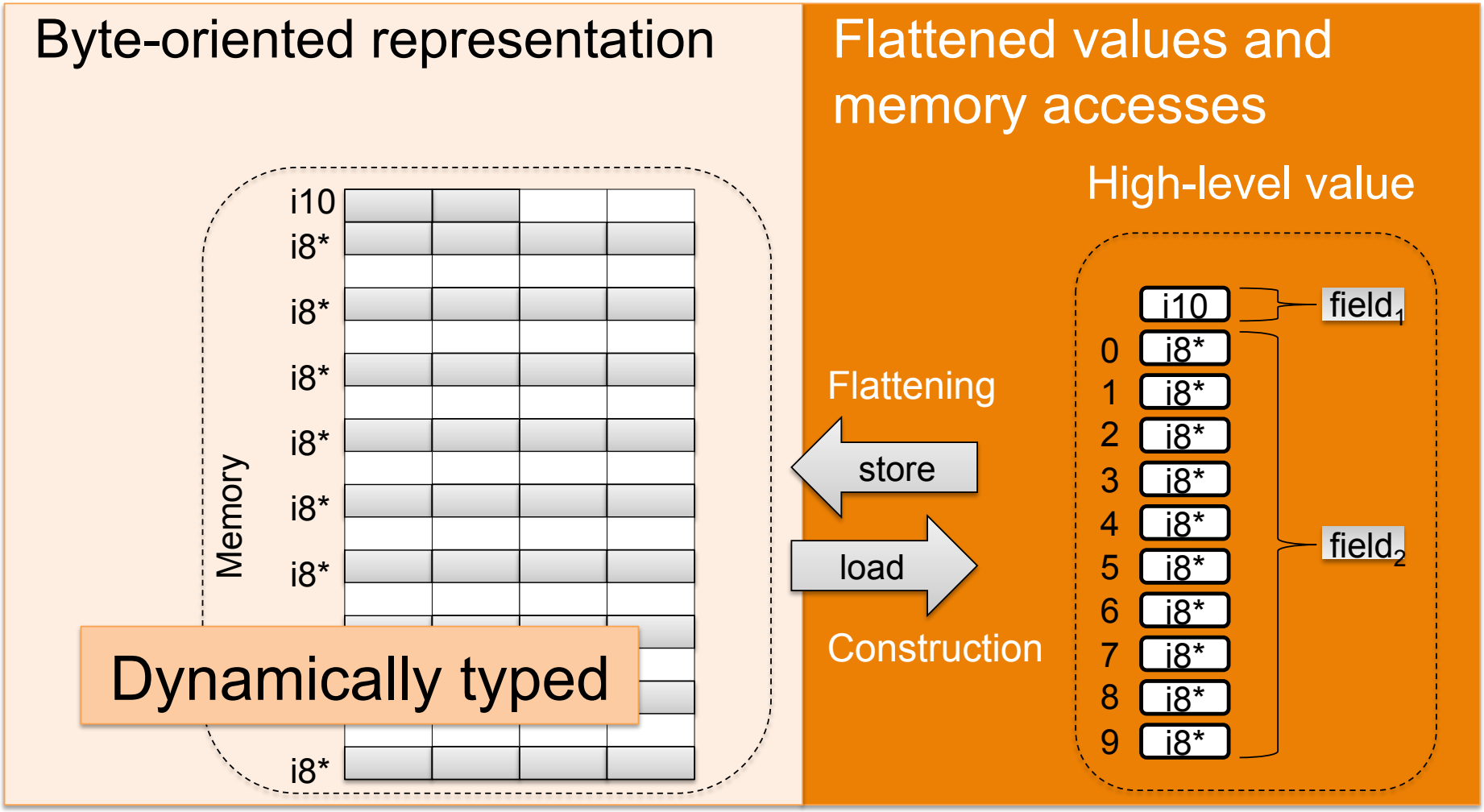


The LLVM Memory Model



The LLVM Memory Model in Vellvm

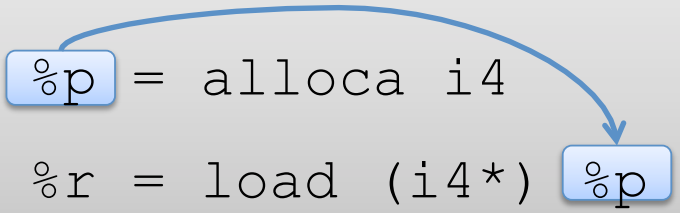
Accessing memory with incompatible types returns undefs



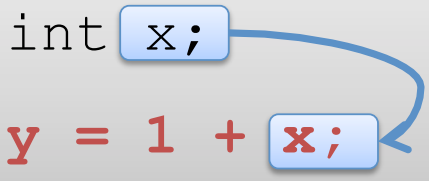
Modeling Undefined Behaviors

Sources of Undefined Behaviors

Loading from uninitialized memory locations



Uninitialized variables



Loading with mismatched types

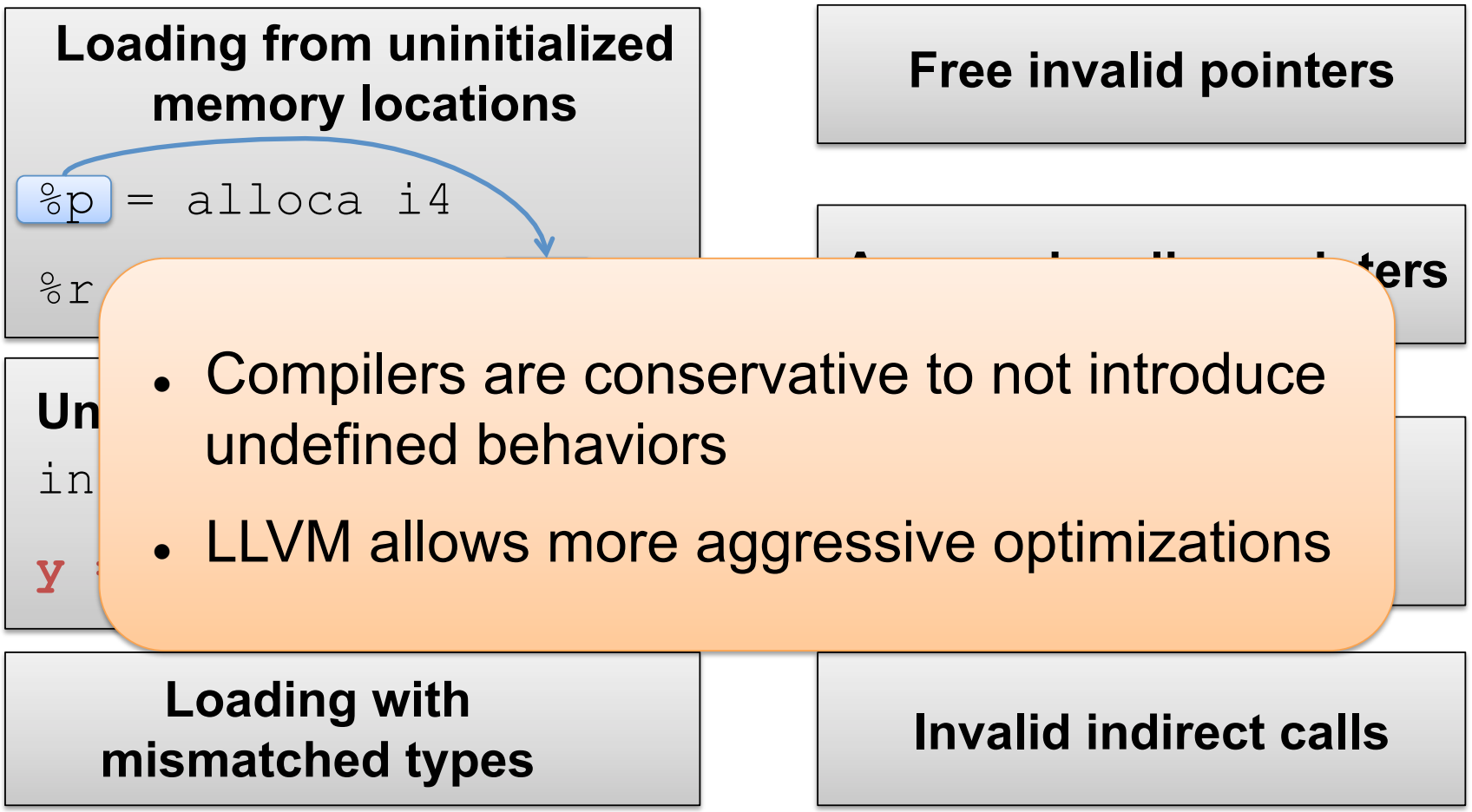
Free invalid pointers

Access dangling pointers

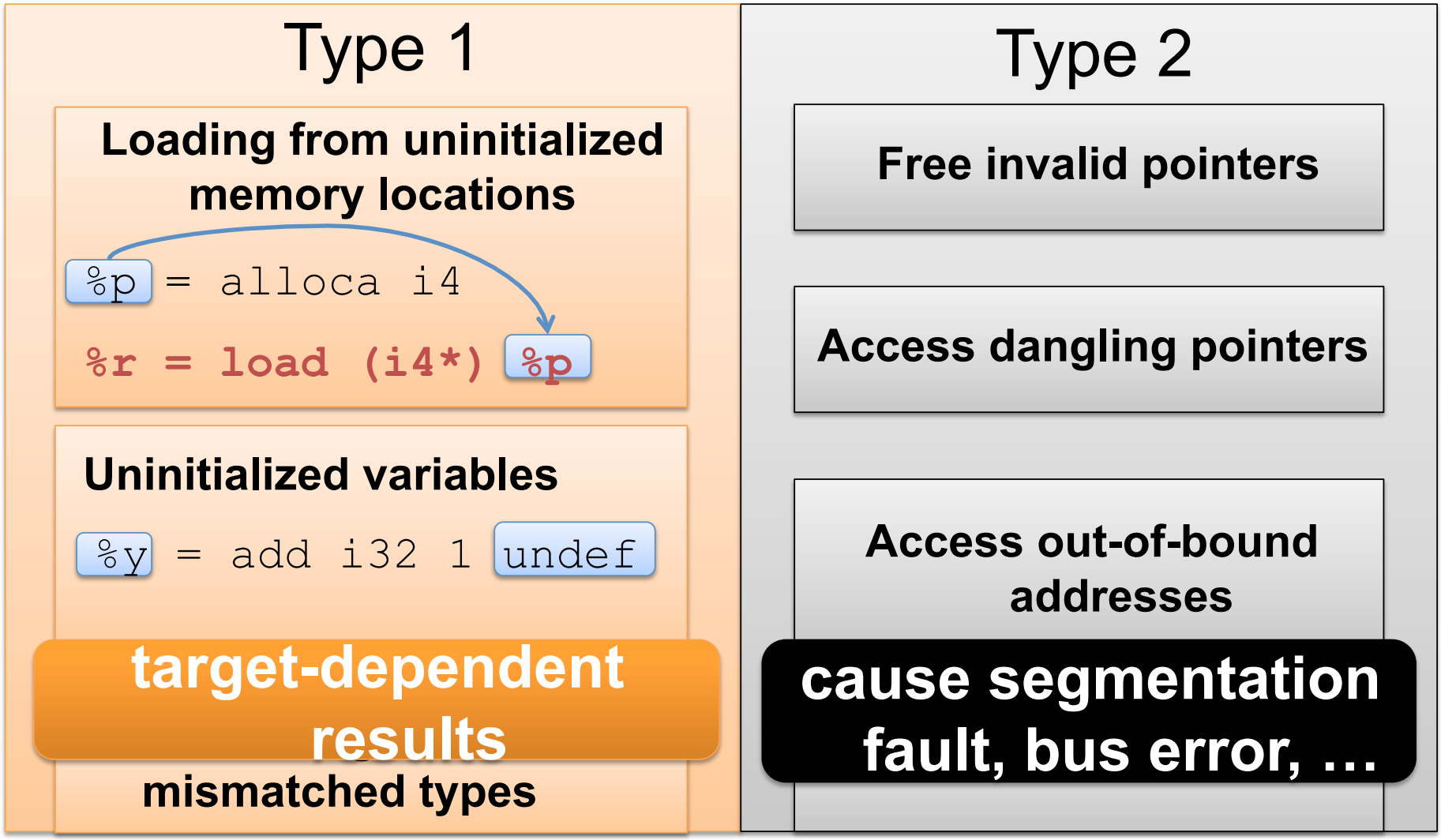
Access out-of-bound addresses

Invalid indirect calls

Sources of Undefined Behaviors



Undefs with LLVM



Undefs in LLVM

Undefined Values

Loading from uninitialized memory locations

```
%p = alloca i4
%r = load (i4*) %p
```

undefined value

Uninitialized variables

```
%y = add i32 1 undef
```

undefined value

Loading with mismatched types

Undefined Behaviors

Free invalid pointers

- Undefined values
  - a set of values
  - aggressive optimizations

Invalid indirect calls

Nondeterministic Operational Semantics

An LLVM runtime value is a **set** of values

small-step

$LLVM_{ND}$: *config* ⊢ $S \rightarrow S'$

Relational:
non-deterministic relations between program states

Otherwise, the semantics is straightforward and tractable.

Partiality

Free invalid pointers

```
free (i8*) NULL
free (i8*) %p
free (i8*) %p
```

Access dangling pointers

```
free (i8*) %p
%r = load (i8*) %p
```

Stuck (config, S) = BadFree (config, S))
∨ BadLoad (config, S))
∨ BadStore (config, S))
∨ …
∨ …

```
%p = malloc i8
%q = gep %p, i32 255
store i8 0, %q
```

Access out-of-bound addresses

… … …

Preservation and Progress

S is well-typed and in the SSA form:

If *config* ⊢ S

Preservation:

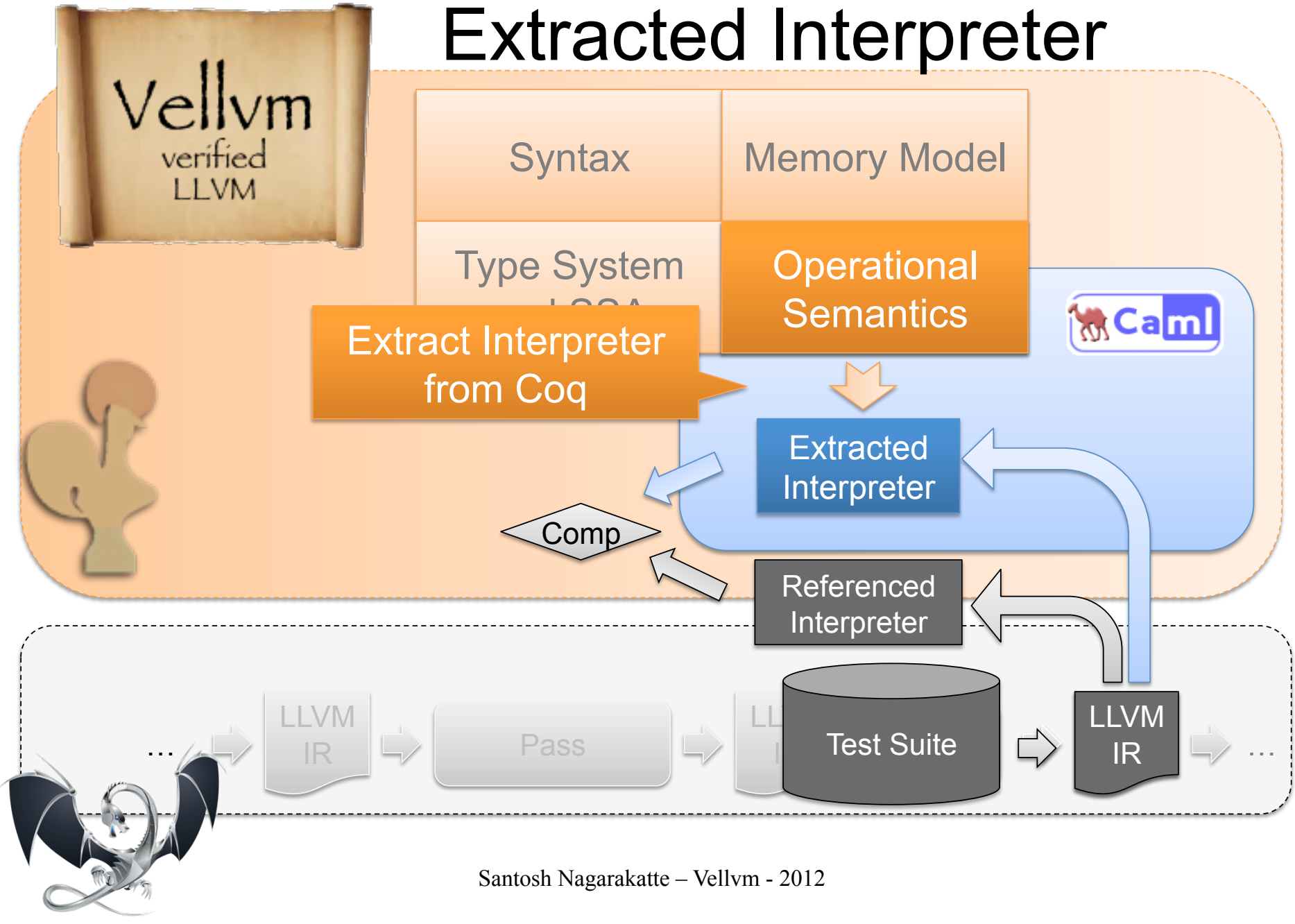
If *config* ⊢ S, and *config* ⊢ *S* → *S'*, then *config* ⊢ *S'*.

Progress:

If *config* ⊢ *S*, then *S* terminates or *Stuck(config, S)* or exists *S'*, *config* ⊢ *S* → *S'*.

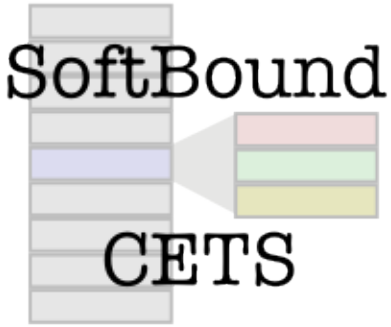
Tools & Applications

Extracted Interpreter



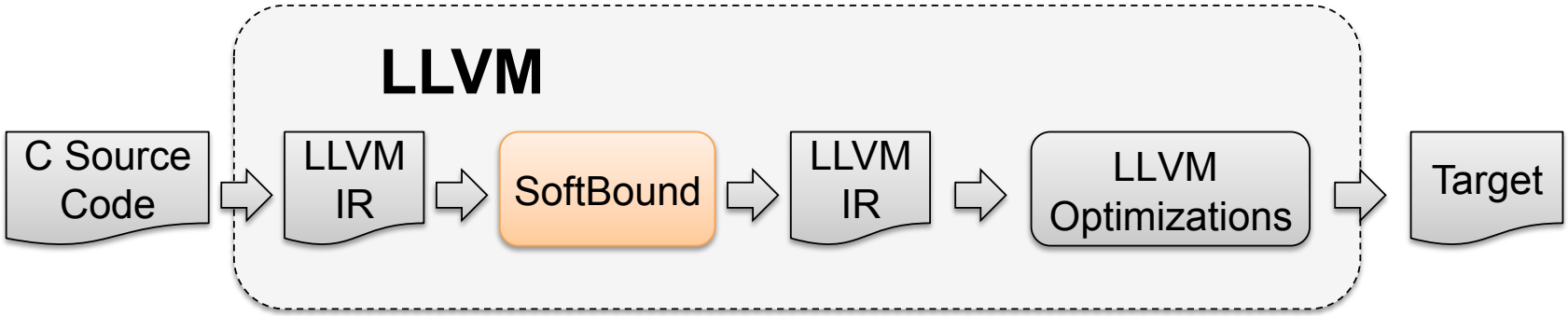
Verified SoftBound

Santosh Nagarakatte – Vellvm - 2012

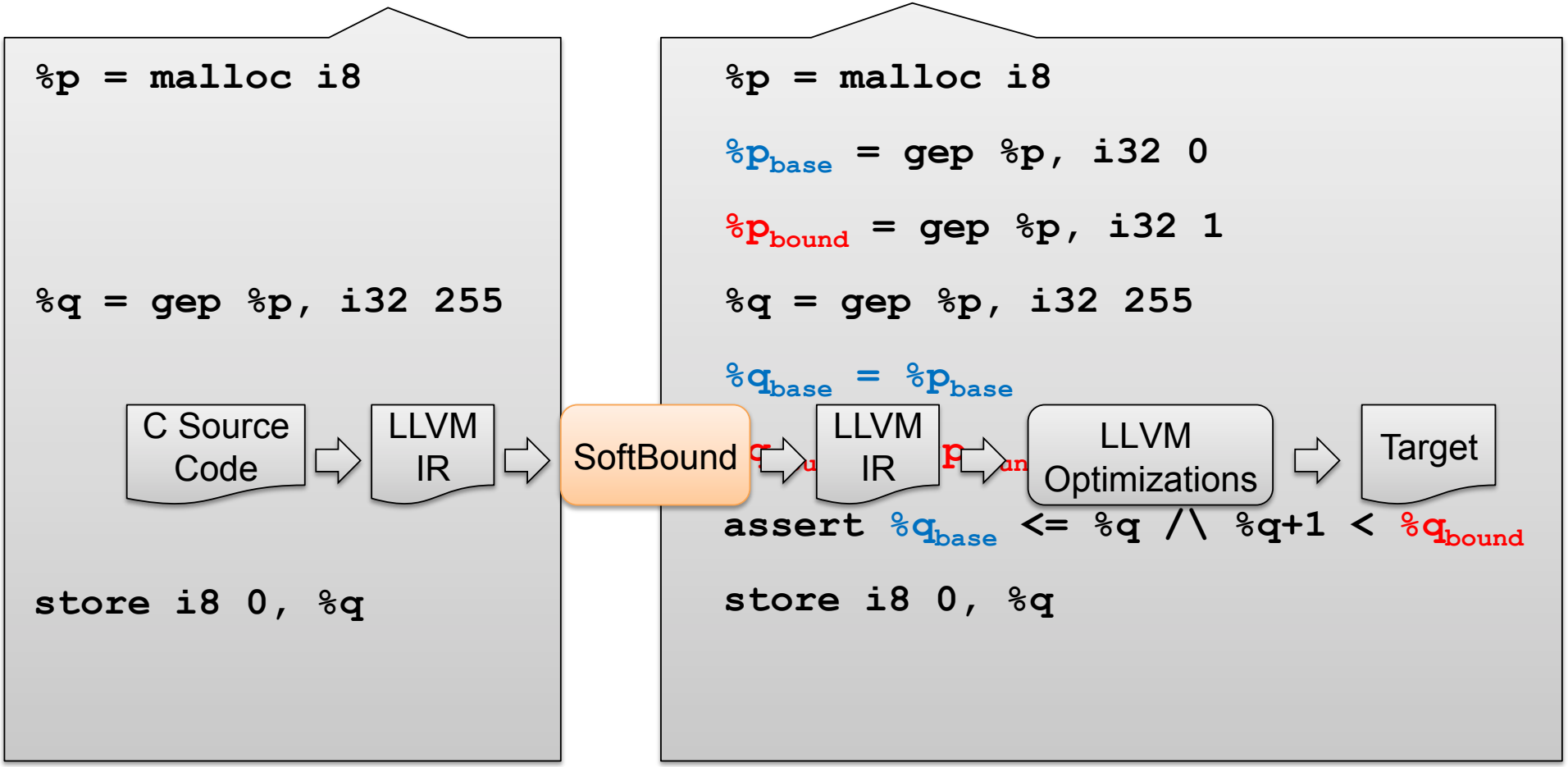


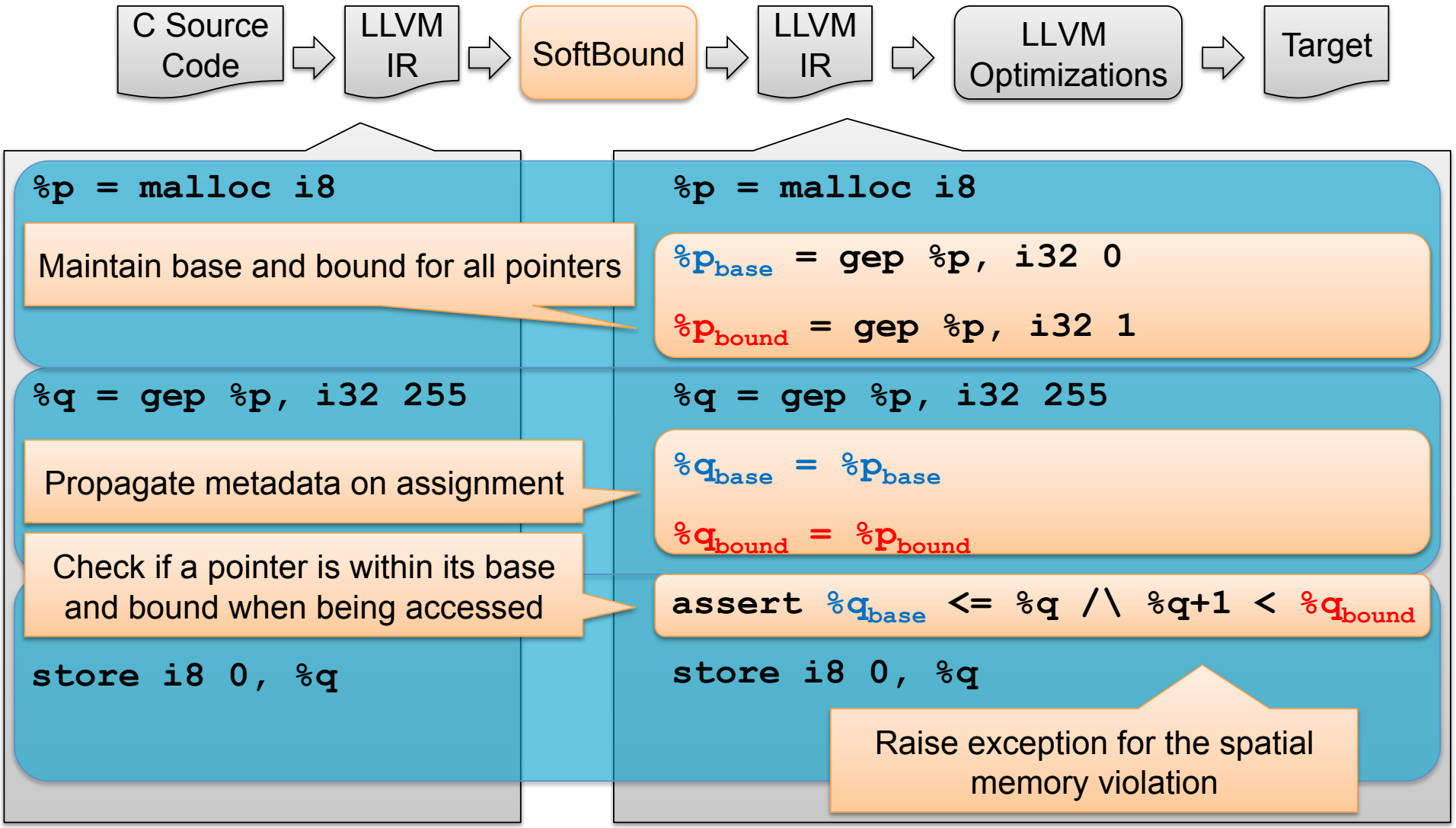
[Nagarakatte, et al. *PLDI '09*]

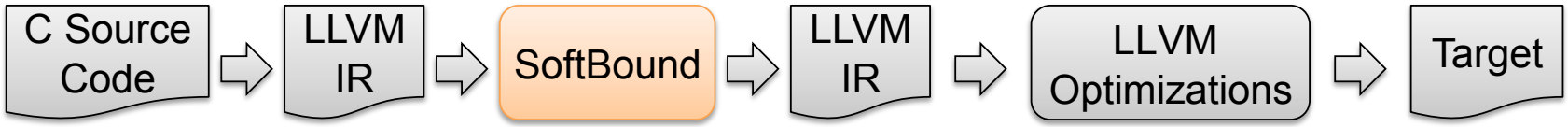
- An instrumentation pass in LLVM
- Detect spatial memory safety violation (buffer overflow, …)

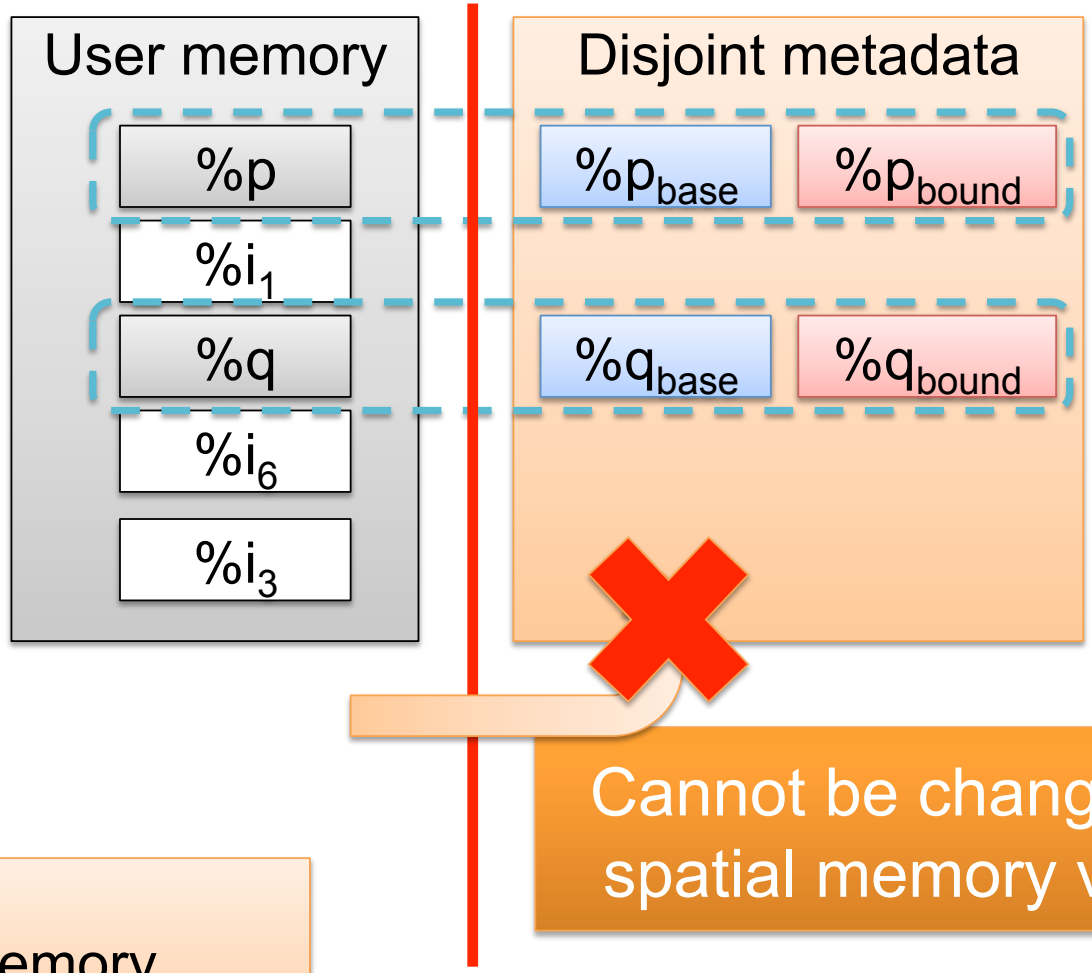


http://www.cis.upenn.edu/acg/softbound/


%p = malloc i8
%q = gep %p, i32 255
store i8 0, %q
C Source Code
LLVM IR
SoftBound
%p = malloc i8
%p_base = gep %p, i32 0
%p_bound = gep %p, i32 1
%q = gep %p, i32 255
%q_base = %p_base
LLVM IR
LLVM Optimizations
Target
assert %q_base <= %q /\ %q+1 < %q_bound
store i8 0, %q


C Source Code
LLVM IR
SoftBound
LLVM IR
LLVM Optimizations
Target
%p = malloc i8
Maintain base and bound for all pointers
%p = malloc i8
%p_base = gep %p, i32 0
%p_bound = gep %p, i32 1
%q = gep %p, i32 255
Propagate metadata on assignment
%q = gep %p, i32 255
%q_base = %p_base
%q_bound = %p_bound
Check if a pointer is within its base and bound when being accessed
assert %q_base <= %q /\ %q+1 < %q_bound
store i8 0, %q
store i8 0, %q
Raise exception for the spatial memory violation
Pointers in temporaries





Cannot be changed by any spatial memory violation!!!

Pointers in memory

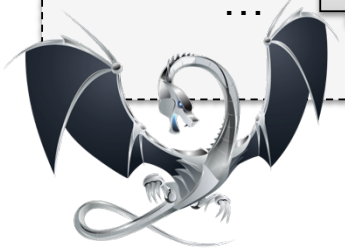
Verified SoftBound

Sy

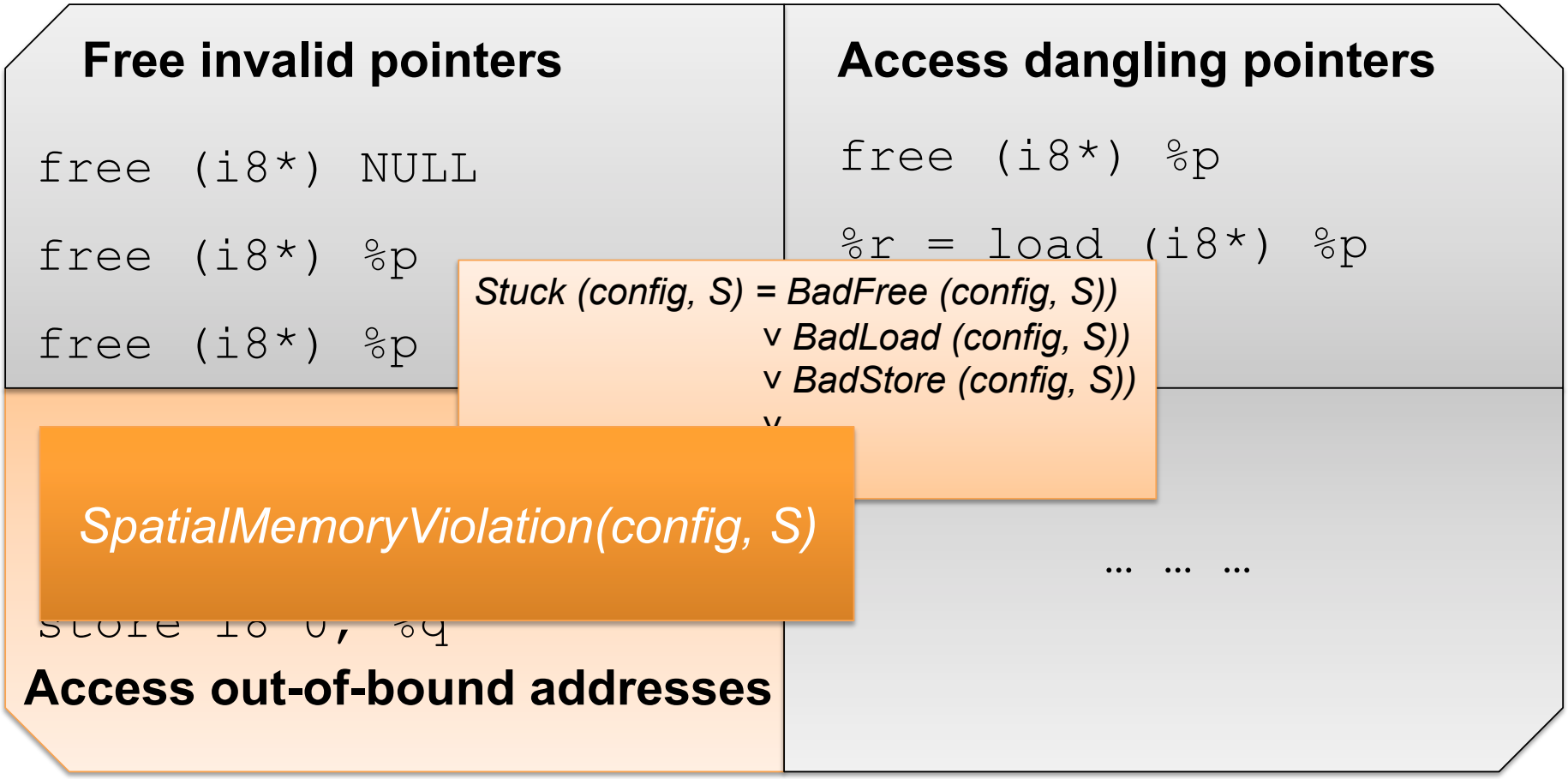
Type
and

Extracting an executable SoftBound pass from the formal development in Coq.

SoftBound



Partiality


Free invalid pointers
free (i8*) NULL
free (i8*) %p
free (i8*) %p
Access dangling pointers
free (i8*) %p
%r = load (i8*) %p
Stuck (config, S) = BadFree (config, S))
∨ BadLoad (config, S))
∨ BadStore (config, S))
SpatialMemoryViolation(config, S)
… … …
Access out-of-bound addresses

Correctness of SoftBound

If $config_1 \vdash S_1$,
$SoftBound(config_1, S_1) = (config_2, S_2)$, and
$config_2 \vdash S_2 \rightarrow^* S'_2$,
then ~ $SpatialMemoryViolation(config_2, S'_2)$

A transformed program
- has no spatial memory safety violation

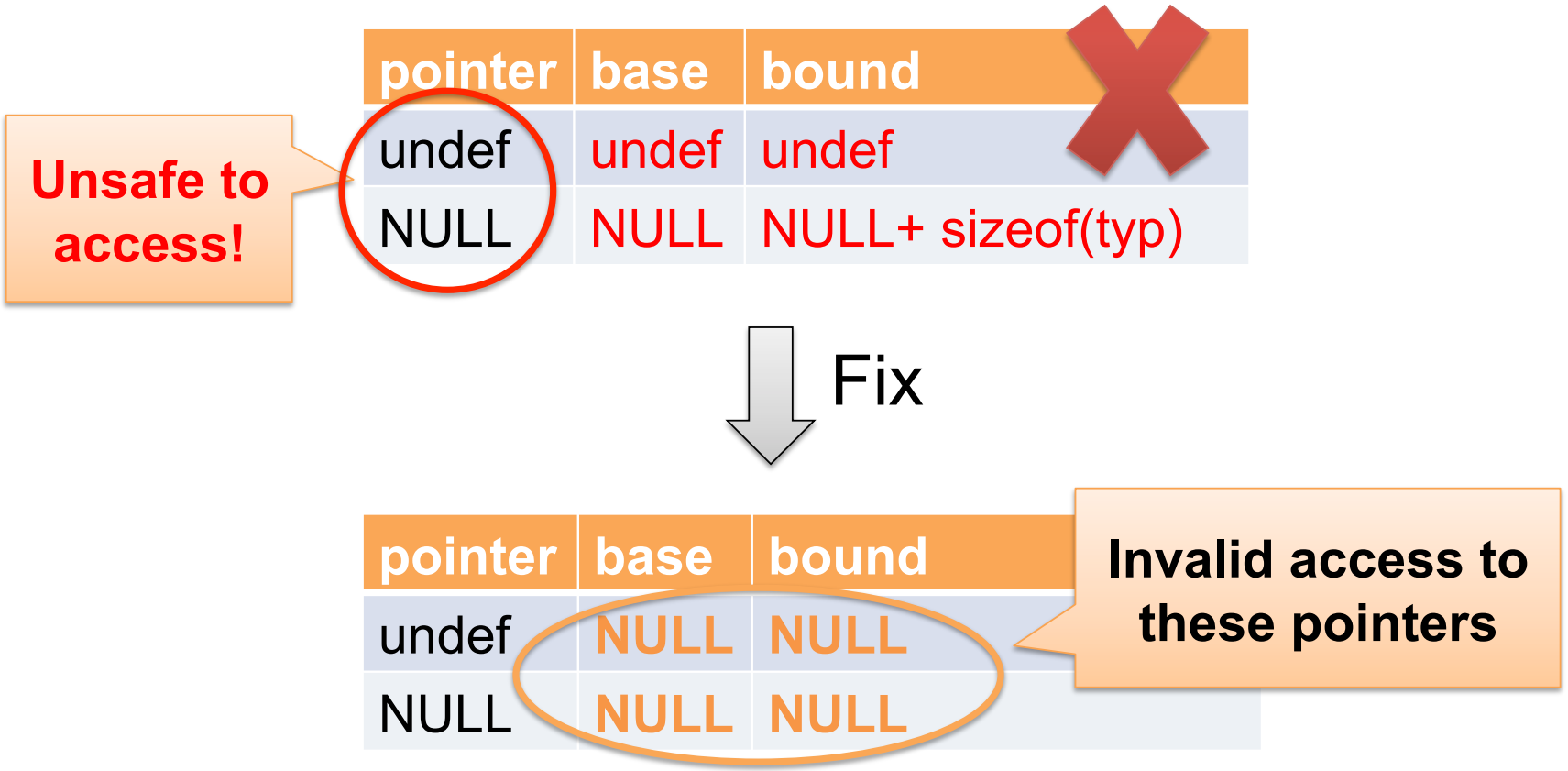
Correctness of SoftBound

If $config_1 \vdash S_1$,
$SoftBound(config_1, S_1) = (config_2, S_2)$, and
$config_2 \vdash S_2 \rightarrow^* S'_2$,
then ~ $SpatialMemoryViolation(config_2, S'_2) \wedge$
exists S'_1,
$config_1 \vdash S_1 \rightarrow^* S'_1 \wedge (config_1, S'_1) \approx (config_2, S'_2)$.

A transformed program
- has no spatial memory safety violation
- preserves the semantics of its original program

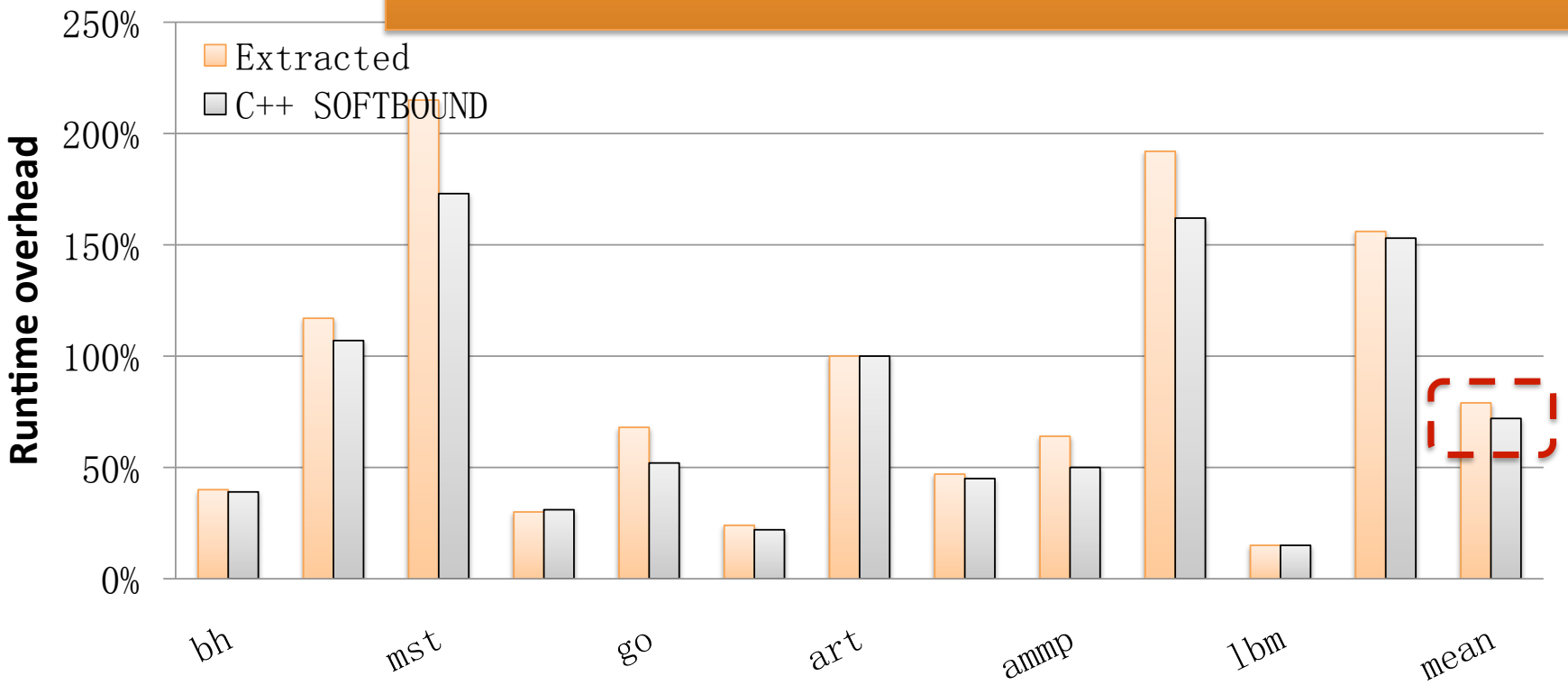
Bugs Found in the Original SoftBound

- Incorrect metadata initialization



Competitive Runtime Overhead

The performance of extracted SoftBound is competitive with the non-verified original.



Written with verification in mind

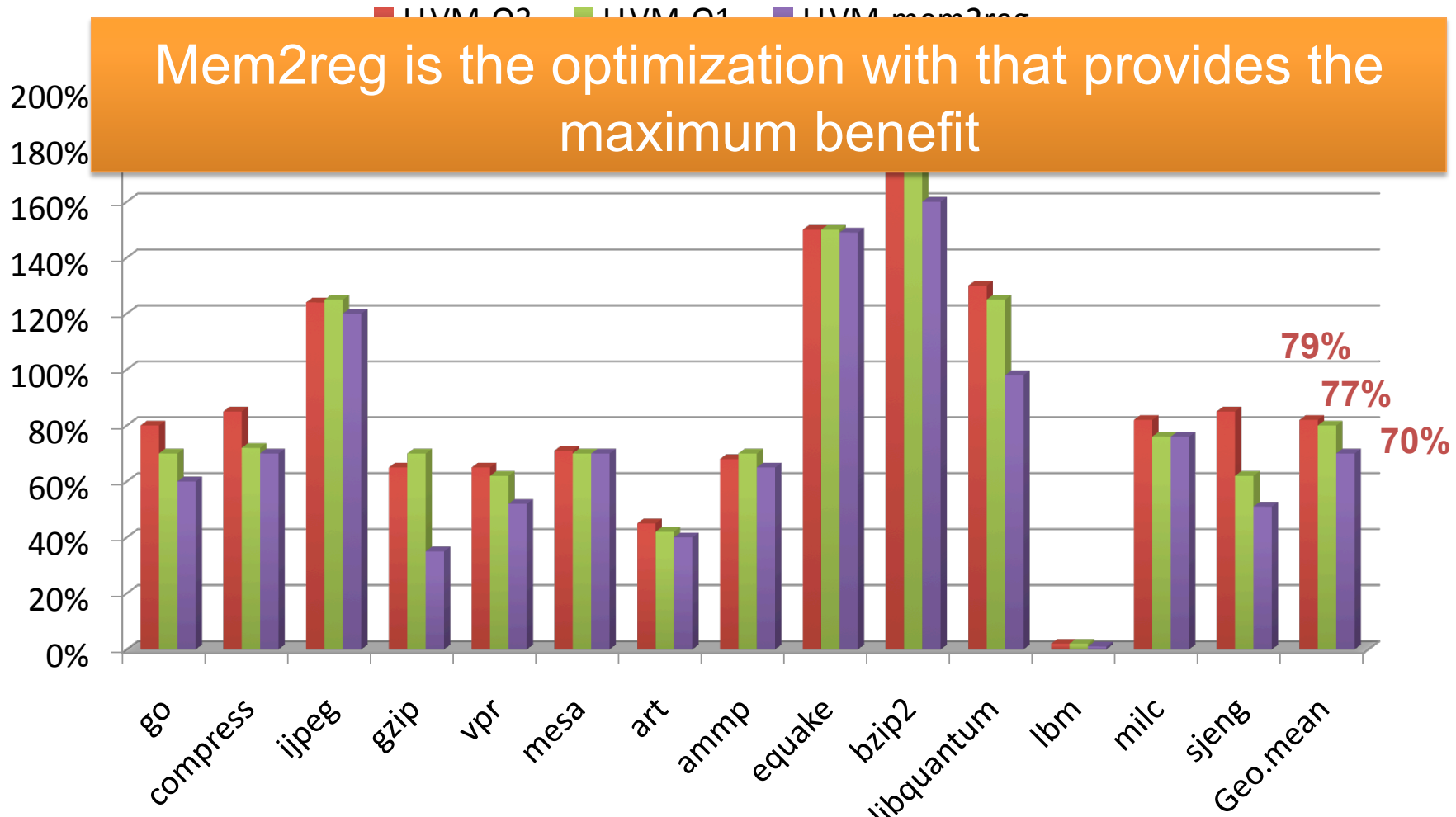
- Simplify both design and proofs
- Increase robustness

A verified and more robust SoftBound with competitive performance in practice.

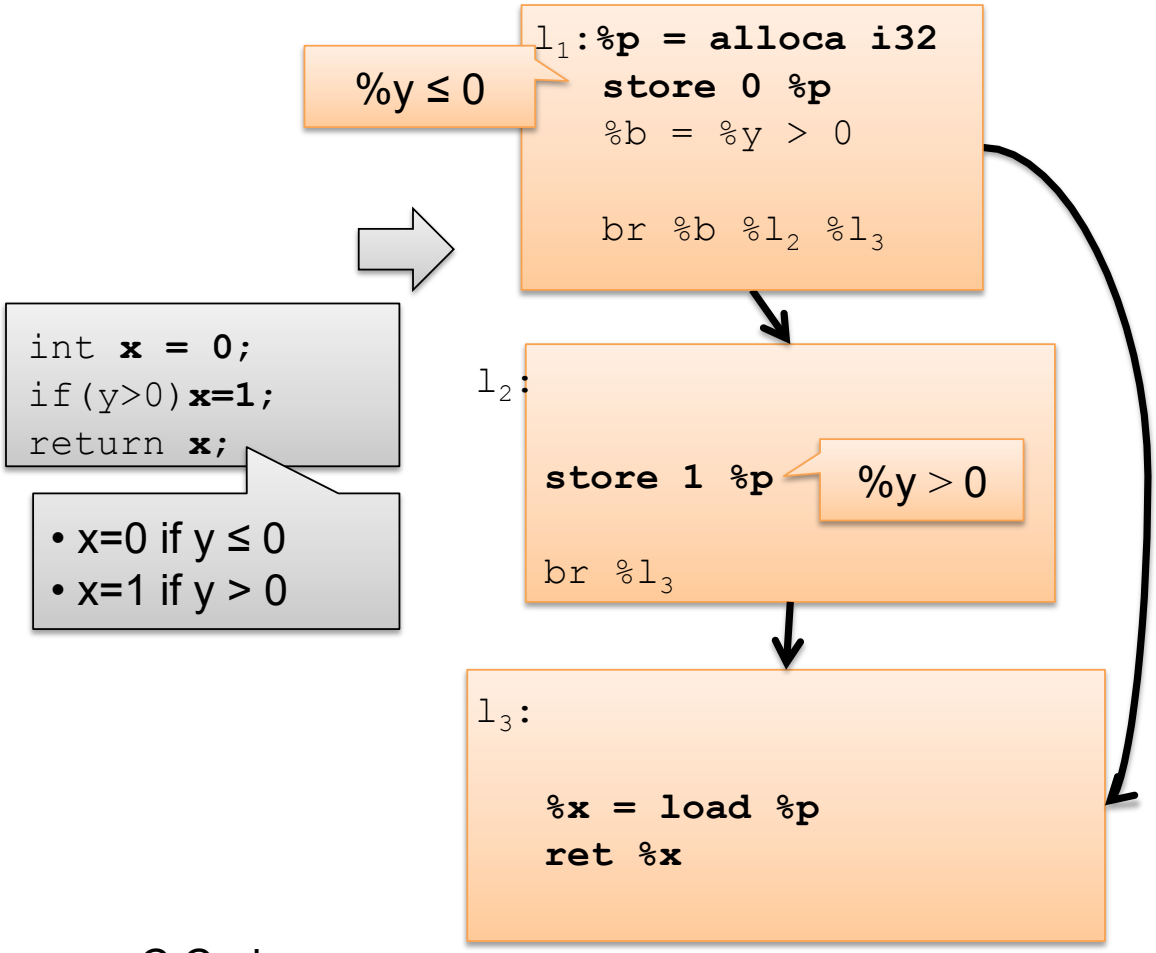
Optimization with maximum performance benefit

Santosh Nagarakatte – Vellvm - 2012

Performance-critical Optimization Pass in LLVM



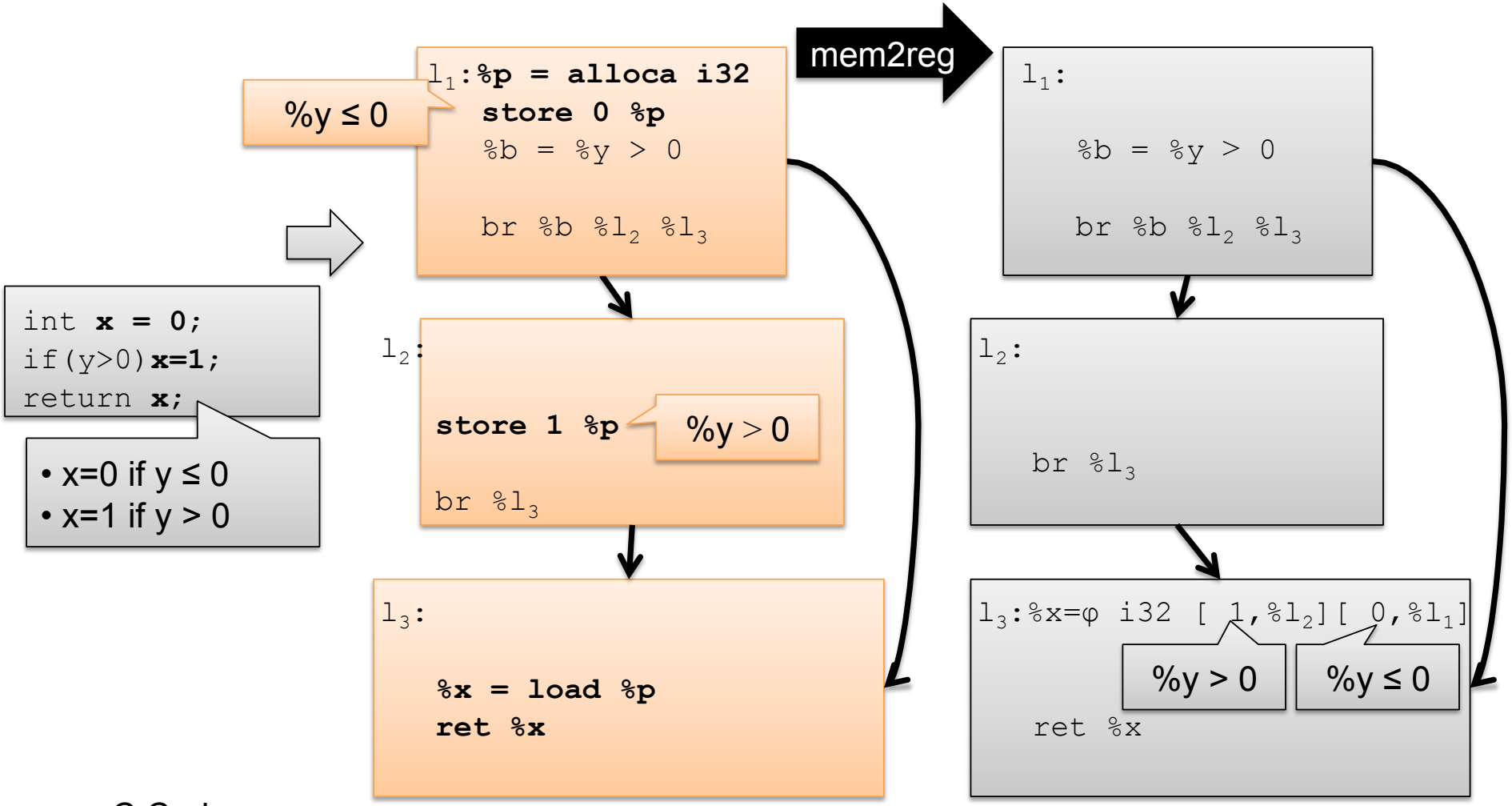
Mem2reg in LLVM



C Code

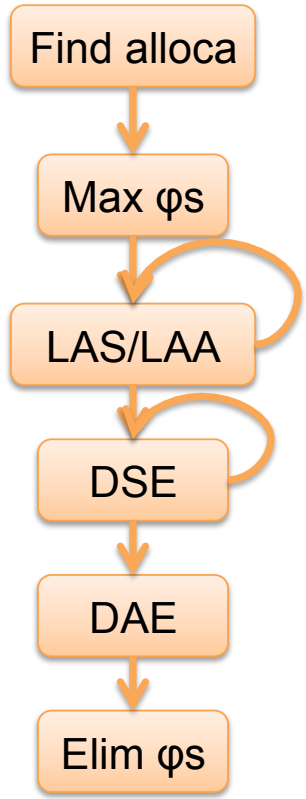
The LLVM IR in the trivial SSA form

Mem2reg in LLVM



Verified mem2reg

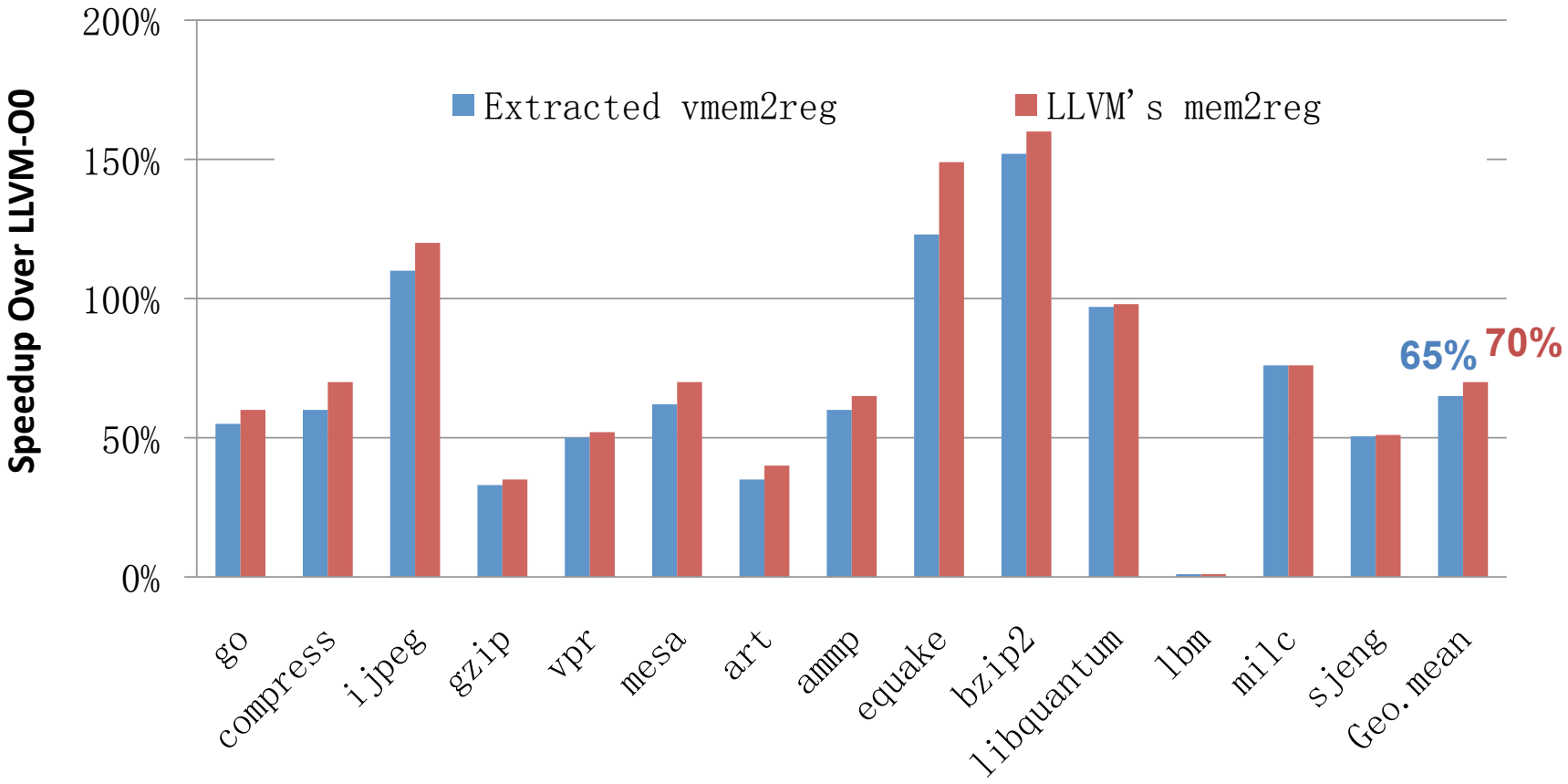
- Wegman et al's SSA construction is commonly used
- Intermediate stage breaks the SSA invariant
- Cannot be proved easily



We designed a new mem2reg that preserves SSA at each step

The runtime speedup of vmem2reg

Vmem2reg does not generate the pruned SSA form



Lessons & Conclusions

- Formal semantics of LLVM IR
  - Meta-theoretic results & tools
  - Verified SoftBound and verified Mem2reg
  - Foundations for a large number of tools/research

- Developer involvement is useful
  - Interesting to explore tradeoffs
    - Easier to prove vs performance vs compilation time

Looking Ahead

Syntax

Memory Model

Foundation for verifying the industry-strength modern compiler --- LLVM

Interpreter

SoftBound

Translation Validator

- Verified LLVM frontends
- Verified LLVM backends
- Reasoning about LLVM programs
- Verified optimizations
- Verified program analysis
- …

…

LLVM IR

Pa

Want to try?

http://www.cis.upenn.edu/~jianzhou/Vellvm

email us for the latest LLVM version