technology
from seed

Automatic Synthesis of Weakest Preconditions for Compiler Optimizations

Nuno Lopes
Advisor: José Monteiro

TÉCNICO LISBOA

Expectations for Compilers

- Improve performance
- Reduce code size
- Reduce energy consumption

Compilers Do Deliver

- LLVM 3.2 introduced a Loop Vectorizer
- Performance improvement of 10-300% in benchmarks

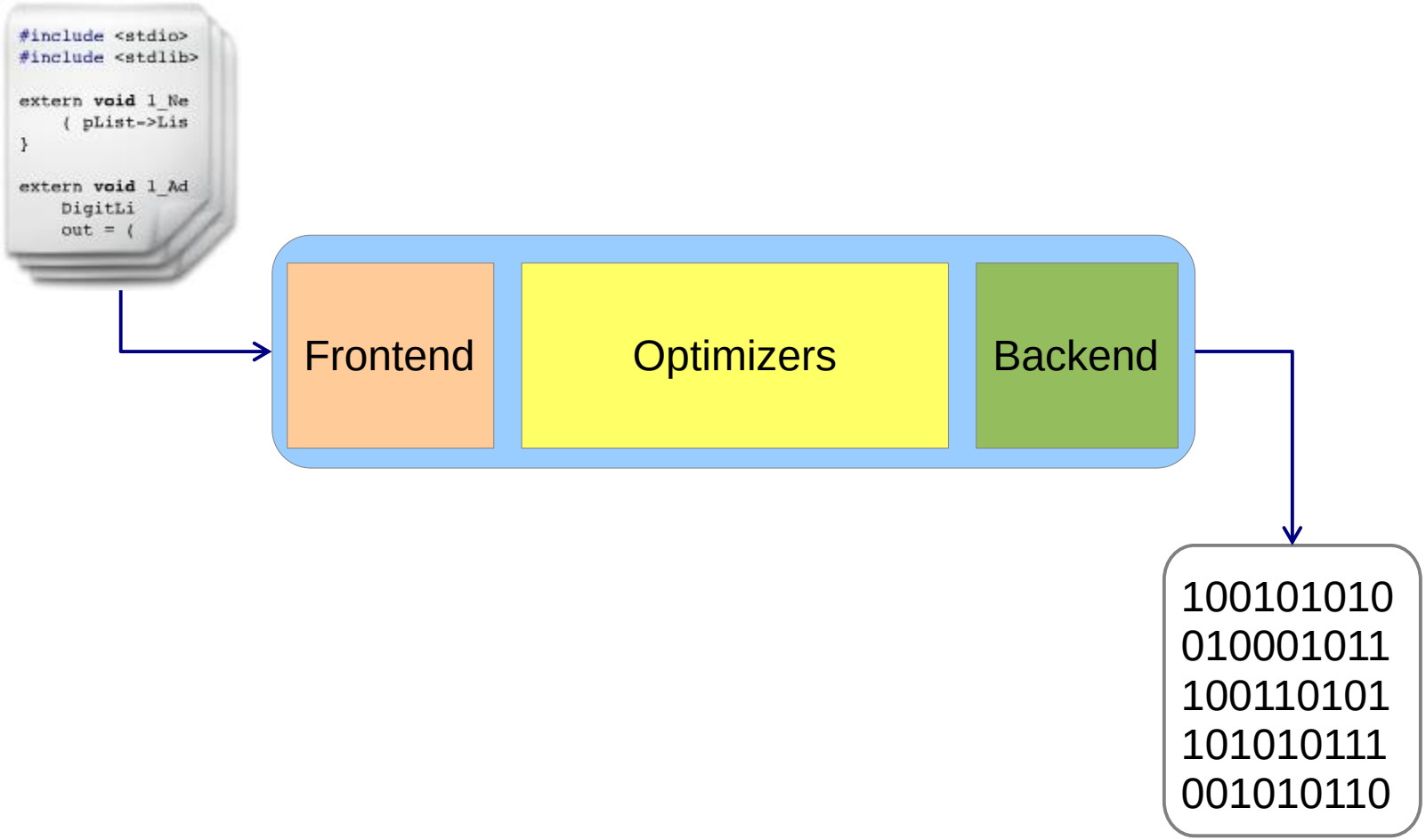
But Compilers are Full of Bugs

- Yang, Chen, Eide, Regehr [PLDI'12]:
  - 79 bugs in GCC (25 P1)
  - 202 bugs in LLVM
  - 2 wrong-code bugs in CompCert
- Le, Afshari, Su [PLDI'14]:
  - 40 wrong-code bugs in GCC
  - 42 wrong-code bugs in LLVM
- Last week:
  - 395 open wrong-code bug reports in GCC
  - 14 open wrong-code bug reports in LLVM

Churn in Compiler's code

- +0.5M LoC added to LLVM last year
- 20k commits
- Over 4M LoC in LLVM

Compilers by Dragon's Lenses



Loop Unswitching


```
while I < N do
    if B then
        S1
    else
        S2
    I := I + 1
```

→

```
if B then
    while I < N do
        S1
        I := I + 1
else
    while I < N do
        S2
        I := I + 1
```

S_1, **S_2** are template statements
B is a template Boolean expression

Specifying Compiler Optimizations

- Transformation function
- Precondition
- Profitability heuristic

Contributions

- Automatic weakest precondition synthesis for compiler optimizations
- Automatic partial equivalence checking, applied to compiler optimization verification

Why WP Synthesis for Compiler Optimizations?

- Deriving preconditions by hand is hard; WPs are often non-trivial
- WPs derived by hand are often wrong!
- Weaker preconditions expose more optimization opportunities

Verification to the Rescue: LLVM PR17827

lib/Transforms/InstCombine/InstCombineCompares.cpp

```
// For a logical right shift, we can fold if the comparison is not
// signed. We can also fold a signed comparison if the shifted mask
// value and the shifted comparison value are not negative.
// These constraints are not obvious, but we can prove that they are
// correct using an SMT solver such as "Z3" :
// http://rise4fun.com/Z3/Tslfh
```

```
if (ShiftOpcode == Instruction::AShr) {
    // There may be some constraints that make this possible,
    // but nothing simple has been discovered yet.
    CanFold = false;
}
```

Loop Unswitching


```
while I < N do
    if B then
        S1
    else
        S2
    I := I + 1
```

→

```
if B then
    while I < N do
        S1
        I := I + 1
else
    while I < N do
        S2
        I := I + 1
```

Loop Unswitching: Example Instantiation


```
…
while I < N do
    if N > 5 then
        A := A + N
    else
        A := A + 1
    I := I + 1
…
```

→

```
if N > 5 then
    while I < N do
        A := A + N
        I := I + 1
else
    while I < N do
        A := A + 1
        I := I + 1
```

Instantiation:

$B \longmapsto N > 5$

$S_1 \longmapsto A := A + N$

$S_2 \longmapsto A := A + 1$

Loop Unswitching: Weakest Precondition


```
while I < N do
    if B then
        S1
    else
        S2
    I := I + 1
```

→

```
if B then
    while I < N do
        S1
        I := I + 1
else
    while I < N do
        S2
        I := I + 1
```

Precondition:

$I \notin R(B) \wedge$
$W(S_1) \cap R(B) = \emptyset \wedge$
$W(S_2) \cap R(B) = \emptyset$

Language of Preconditions

- Read and Write sets for each template statement/expression
- Arbitrary quantifier-free constraints over read/write sets
- In practice constraints are only over R/W and W/W intersection
  - $v \notin R(B)$
  - $W(S_1) \cap R(B) = \emptyset$
  - $W(S_1) \cap W(S_2) = \emptyset$

Language of Preconditions: Suitability

- Books and developers already informally speak about read and write sets
- Similar to PEC's
- Can be efficiently discharged using current compiler technology:
  - Memory dependence analysis
  - Alias/pointer analysis
  - Loop analysis
  - Range analysis
  - …

Synthesizing WP for Loop Unswitching


```
while I < N do
    if B then
        S1
    else
        S2
    I := I + 1
```

→

```
if B then
    while I < N do
        S1
        I := I + 1
else
    while I < N do
        S2
        I := I + 1
```

1) Find counterexample

technology
from seed

inesc id lisboa

```
while I < N do
    if B then
        S1
    else
        S2
    I := I + 1
```

→

```
if B then
    while I < N do
        S1
        I := I + 1
else
    while I < N do
        S2
        I := I + 1
```

Pre = true

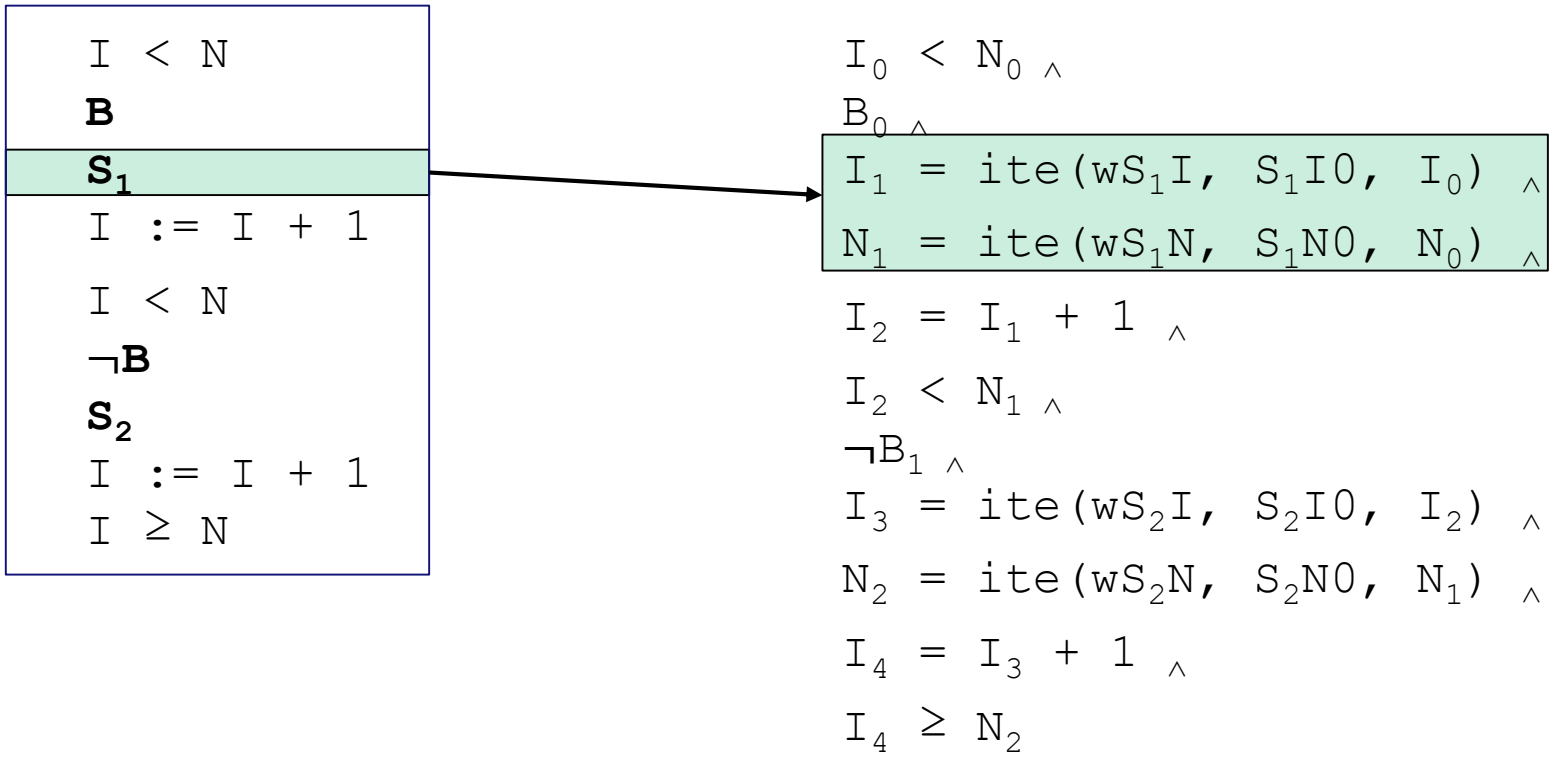
```
I < N
B
S1
I := I + 1
```

```
I < N
¬B
S2
I := I + 1
I ≥ N
```

```
B
I < N
S1
I := I + 1
```

```
I < N
S1
I := I + 1
I ≥ N
```

2) Synthesize WP for counterexample: VC Gen



2) Synthesize WP for counterexample: Conditional Ackermannization

$I_0 < N_0 \wedge$
$B_0 \wedge$
$I_1 = ite(wS_1I, S_1I0, I_0) \wedge$
$N_1 = ite(wS_1N, S_1N0, N_0) \wedge$
$I_2 = I_1 + 1 \wedge$
$I_2 < N_1 \wedge$
$\neg B_1 \wedge$
$I_3 = ite(wS_2I, S_2I0, I_2) \wedge$
$N_2 = ite(wS_2N, S_2N0, N_1) \wedge$
$I_4 = I_3 + 1 \wedge$
$I_4 \geq N_2$

B_0 and B_1 are equal if the values of the variables in R(B) are equal

$$\begin{pmatrix}(I \in R(B) \rightarrow I_0 = I_2) \wedge \\ (N \in R(B) \rightarrow N_0 = N_1)\end{pmatrix} \rightarrow B_0 = B_1$$

2) Synthesize WP for counterexample: Final constraint

inesc id lisboa

```
I < N                    B
B                        I < N
S1                       S1
I := I + 1               I := I + 1
-------------------------------------------
I < N                    I < N
¬B                       S1
S2                       I := I + 1
I := I + 1               I ≥ N
I ≥ N
```

$$\exists S \; \forall V \; Path \wedge Ackermann \wedge MustWrite \wedge \ldots \rightarrow PathIsCorrect$$

S = Read/Write sets

V = Vars from VCGen, Must-write vars

A possible model:

$W(S_1) = \emptyset$

$R(S_1) = \emptyset$

$R(B) = \emptyset$

2) Synthesize WP for counterexample: Disjunction of all models

`I < N`	**B**
B	`I < N`
S_1	**S_1**
`I := I + 1`	`I := I + 1`
`I < N`	`I < N`
¬B	**S_1**
S_2	`I := I + 1`
`I ≥ N`	`I ≥ N`

Precondition:

$I \notin R(B) \wedge$

$W(S_1) \cap R(B) = \emptyset$

3) Iterate until no more counterexamples can be found


```
while I < N do
    if B then
        S1
    else
        S2
    I := I + 1
```

→

```
if B then
    while I < N do
        S1
        I := I + 1
else
    while I < N do
        S2
        I := I + 1
```

Precondition:

$I \notin R(B) \wedge$
$W(S_1) \cap R(B) = \emptyset \wedge$
$W(S_2) \cap R(B) = \emptyset$

Algorithm

1) Find counterexample
2) Generate WP that rules out the counterexample
3) Iterate until no more counterexamples can be found

Optimizations

- Model generalization
- Exploit UNSAT cores
- Bias towards R/W and W/W intersections

PSyCO: Results

Optimization	# Counterexamples	# Models	WP Time	Total Time
Code hoisting	1	1	0.07s	0.23s
Constant propagation	1	1	0.04s	0.16s
Copy propagation	0	0	0s	0.11s
If-conversion	0	0	0s	0.11s
Partial redundancy elimin.	1	1	0.10s	0.30s
Loop fission	6	36	1.28s	2.18s
Loop flattening	1	1	0.07s	3.31s
Loop fusion	6	36	1.26s	2.19s
Loop interchange	11	25	1.42s	23.8s
Loop invariant code motion	3	3	0.22s	0.55s
Loop peeling	0	0	0s	0.27s
Loop reversal	4	7	0.25s	0.54s
Loop skewing	1	1	0.06s	163s
Loop strength reduction	1	2	1.14s	1.41s
Loop tiling	1	1	0.07s	4.60s
Loop unrolling	2	4	0.13s	0.50s
Loop unswitching	2	2	0.15s	0.77s
Software pipelining	1	2	0.13s	0.58s

Example of Synthesized WP: Software Pipelining

inesc id lisboa

$$\textbf{while } V_1 < V_2 \textbf{ do} \quad \Rightarrow \quad \textbf{if } V_1 < V_2 \textbf{ then}$$
$$\quad S_1 \qquad\qquad\qquad\quad S_1$$
$$\quad S_2 \qquad\qquad\qquad\quad \textbf{while } V_1 < (V_2 - 1) \textbf{ do}$$
$$\quad V_1 := V_1 + 1 \qquad\qquad\quad S_2$$
$$\qquad\qquad\qquad\qquad\quad V_1 := V_1 + 1$$
$$\qquad\qquad\qquad\qquad\quad S_1$$
$$\qquad\qquad\qquad\qquad S_2$$
$$\qquad\qquad\qquad\qquad V_1 := V_1 + 1$$

Precondition:

$$V_2 \notin \mathsf{W}(S_2) \wedge$$
$$\big((\mathsf{R}(S_1) \cap \mathsf{W}(S_2) = \emptyset \wedge$$
$$\mathsf{R}(S_1) \cap \mathsf{W}(S_1) = \emptyset \wedge$$
$$\mathsf{R}(S_2) \cap \mathsf{W}(S_2) = \emptyset) \vee$$
$$V_1 \notin \mathsf{W}(S_2)\big)$$

(Weaker than PEC's [PLDI'09])

Verifying Optimizations with CORK

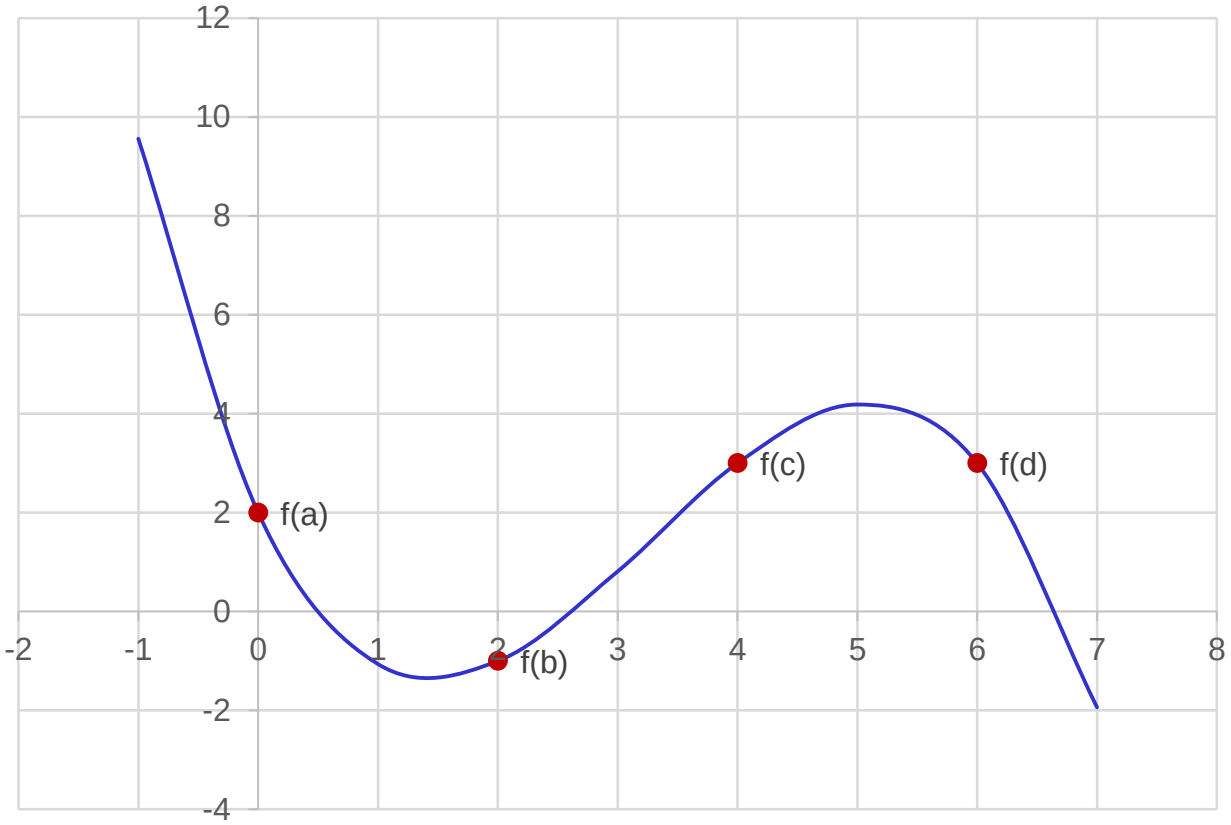
- Template statements/expressions become UFs over the read and write sets
  - $\mathtt{S}_1$ -> $S_1(x, y, z)$ w/ $R(S_1) = \{x, y, z\}$

- Originates 2 UF+IA programs

CORK: Partial Equivalence Checking of UF+IA Programs

1. UFs abstracted by polynomials
   - $S_1(x, y, z)$ -> $ax + by + cz + d$ (w/ $u(S_1) \leq 2$)
2. Loops summarized using recurrences
3. Sequential composition
   - Reduces to safety checking of loop-free + integer arithmetic program

CORK: Polynomial Interpolation



CORK: Results

Optimization	PEC	Queries	Recurrences	Time
Code hoisting	✓	2	0	0.32s
Constant propagation	✓	0	0	0.33s
Copy propagation	✓	0	0	0.33s
If-conversion	✓	2	0	0.34s
Partial redundancy elim.	✓	2	0	0.34s
Loop inv. code motion	✓	7	5	3.48s
Loop peeling	✓	9	5	3.26s
Loop unrolling	✓	13	8	12.17s
Loop unswitching	✓	14	14	8.19s
Software pipelining	✓	9	5	8.02s
Loop fission	$\checkmark_p$	10	12	23.45s
Loop fusion	$\checkmark_p$	10	12	23.34s
Loop interchange	$\checkmark_p$	15	24	29.30s
Loop reversal	$\checkmark_p$	7	5	8.41s
Loop skewing	$\checkmark_p$	16	24	8.50s
Loop flattening	×	—	—	T/O
Loop strength reduction	×	6	4	5.63s
Loop tiling	×	7	9	10.94s

Future Work

- Apply to production compilers
- Synthesize implementation of optimizations (pattern matching, VC Gen, code transformation)
- Explain reasons for optimization failure
- Preserve debug info automatically
- Preserve analysis data across optimizations

Conclusion

- There is significant on-going effort to improve compilers, which compromises correctness
- Presented the first algorithm for the automatic synthesis of WPs for compiler optimizations
- Presented the first algorithm for automatic partial equivalence checking of UF+IA programs
  - Applied to verification of compiler optimizations

inesc id
lisboa

CORK: UFs -> Polynomials

- $f(x_1, \ldots, x_n) = \sum_{\alpha \cdot 1 \leq d} C_\alpha X^\alpha$

- $u(f) \leq \binom{n+d}{n}$

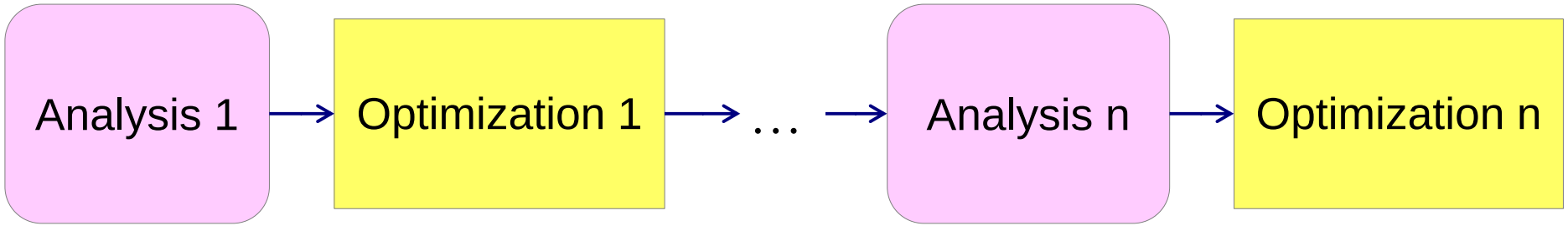
2) Synthesize WP for counterexample: Must-write vs may-write

$$
\begin{aligned}
& I_0 < N_0 \wedge \\
& B_0 \wedge \\
& I_1 = \text{ite}(wS_1I, S_1I0, I_0) \wedge \\
& N_1 = \text{ite}(wS_1N, S_1N0, N_0) \wedge \\
& I_2 = I_1 + 1 \wedge \\
& I_2 < N_1 \wedge \\
& \neg B_1 \wedge \\
& I_3 = \text{ite}(wS_2I, S_2I0, I_2) \wedge \\
& N_2 = \text{ite}(wS_2N, S_2N0, N_1) \wedge \\
& I_4 = I_3 + 1 \wedge \\
& I_4 \geq N_2
\end{aligned}
$$

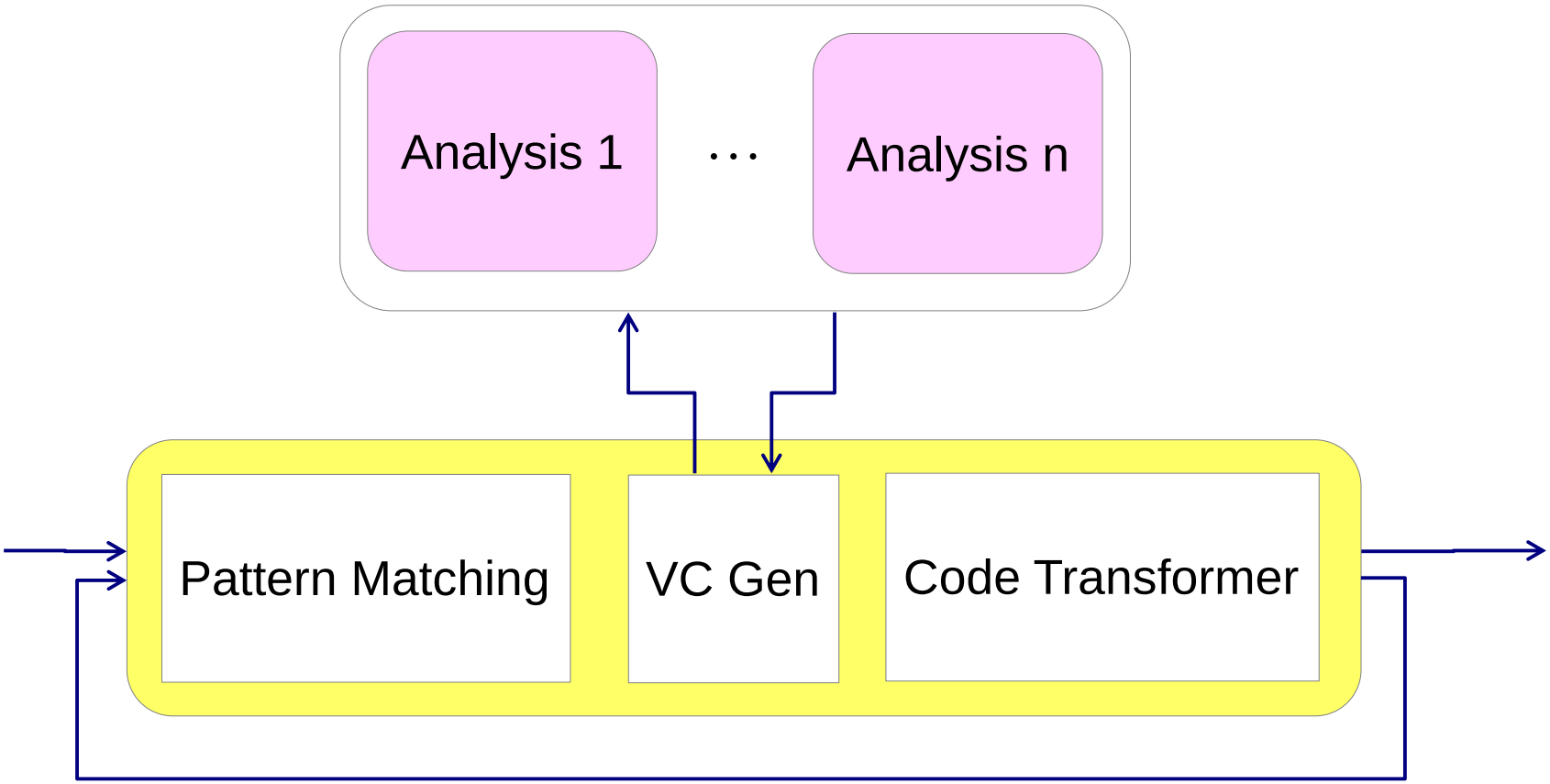
If a variable is in the write set of a statement, it may or may not be written.

$$wS_1I \rightarrow I \in W(S_1)$$
$$wS_1N \rightarrow N \in W(S_1)$$

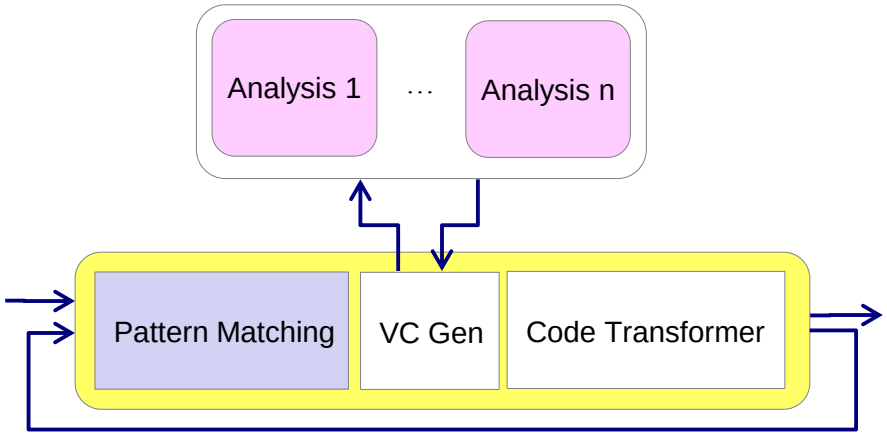
Optimizers by Dragon's Lenses



An Optimizer from the Future



An Optimizer from the Future: Pattern Matching



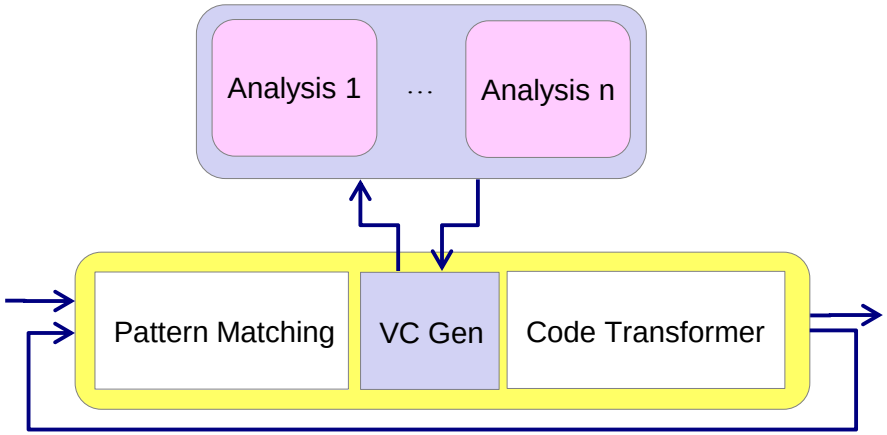

```
if B then
    S1
```

$B \longmapsto N > 5$

$S_1 \longmapsto A := A + N$

```
…
while I < N do
    if N > 5 then
        A := A + N
    else
        A := A + 1
    I := I + 1
…
```

An Optimizer from the Future: Verification



$$B \longmapsto N > 5$$
$$S_1 \longmapsto A := A + N$$

\+ Precondition = φ

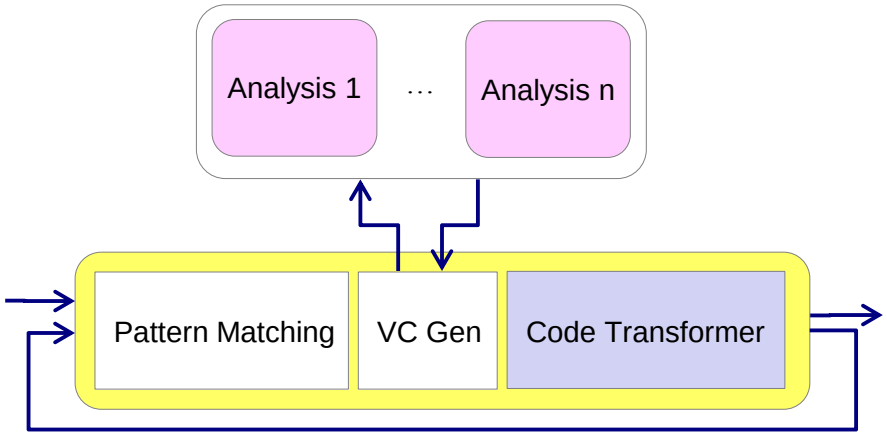
Range Analysis
Alias Analysis
Scalar Evolution
…

SLAM SLAYER

Duality
HSF
Terminator
…

Z3

An Optimizer from the Future: Code Transformation




```
if B then
    S1
```

→

```
A1 := A + 1
if N then
    skip
```

$$B \longmapsto N > 5$$
$$S_1 \longmapsto A := A + N$$