Generating Java Compiler Optimizers Using Bidirectional CTL

Ling Fang and Masataka Sassa

Dept. of Mathematical and Computing Sciences
Tokyo Institute of Technology

Presentation outline

- Introduction
  - Previous related works and a brief view of our study
- Theories
  - Temporal logic, CTL, Kripke structure, program modeling
- Our research
  - Model checker, specification language, free variable
- Experiments and discussion
- Future work and conclusion

Introduction

Optimizer by CTL

- The approach of implementing optimizers by CTL (computational tree logic) has attracted interest in recent years.
- Optimization can be concisely expressed as following:

  $I \rightarrow I' \text{ if } \phi$

  The example of dead code elimination is:

  $v := e \rightarrow skip$

  if $\neg EX\,(E\neg\, def(v)\ U\ use\ (v))$

The approach has two main advantages

- The transformations are easier to write and prototype by writing several lines of specification language.
- The transformations can be formally analyzed and proved because they are more simply expressed.

Past work

- Work of Lacey et al. (2002–2004)
  - Introduces CTL–FV.
  - Describes the proposal and proves the correctness by hand.
  - The formulas that have been proven in the thesis are only a part of the real–world optimization.
  - No experimental data are given.

  Accordingly, It is very a valuable work but not practical.

- Work of Yamaoka et al. (2003)
  - An optimizer with CTL using the existing model checker SMV.
  - Only future temporal operators are allowed because SMV can not deal with bidirectional CTL.
  - It can rewrite only one instruction corresponding to one condition.

Accordingly, it can only deal with dead code elimination

- Work of Ban et al. (2004)
  - Treat PCTL (convert past tense to future tense).
  - Consumes time to remove the past temporal operators.
  - The formulas become very long after conversion.
  - It can rewrite only one instruction corresponding to one condition.

  Accordingly, the model checking time is considerable and only the dead code elimination and copy propagation can be done.

- Work of Lerner et al. (2005)
  - A domain specific language named Rhodium
  - It can be semi-automatically proved sound
  - It is not based on temporal logic and is different from ours.

Problem that remains open until 2005

- Is it possible to optimize a program in the same quality as usual algorithm by a comprehensible mathematical technique based on logic just like CTL?

The contribution of our research

Our research was done to solve the problem.

- We developed a Java optimizer by CTL
  - The optimization specification language is very expressive. It is very easy to describe a complex rewrite rule.
  - The compile time is practical.
  - The optimizer has a modest effectiveness.

  It is now close to the optimizers using usual algorithms.
- It is the first time that experimental data using the benchmarks and the test programs has been measured.
- Some problems of optimization by CTL were clarified.

Theory

Temporal logic and its categories

- Temporal logic is a kind of theory that the value of formula will be satisfied at different times/states.
- Categories
  - Classic kinds: LTL, CTL, CTL*, PCTL, NCTL, etc.
  - Enhanced kind: CTL-FV (introduced by Lacey et al.)

CTL-FV

- Formula
  - state formula:

    A(futureAll), E(futureExist), $\overleftarrow{A}$(pastAll), $\overleftarrow{E}$(pastExist)
  - path formula:

    U(Until), X(neXt), G(Global), F(Future)
- Syntax

  ϕ ::= true | false | prop | $\phi \wedge \phi$ | $\phi \vee \phi$ | $\neg \phi$

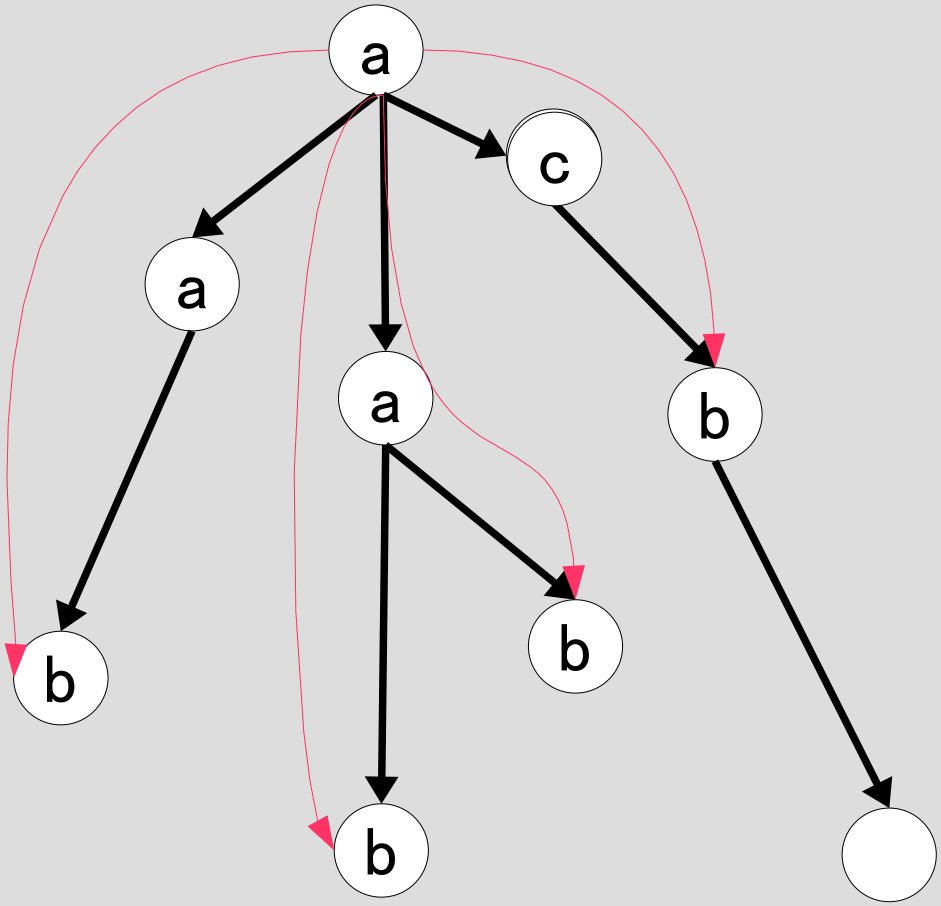
  | Eψ | Aψ | $\overleftarrow{E}\psi$ | $\overleftarrow{A}\psi$

  ψ ::= Xϕ | ϕ U ϕ |

  prop: predicate with free variable

  So, CTL formula is always like AϕUϕ, EϕUϕ, AϕXϕ, etc.

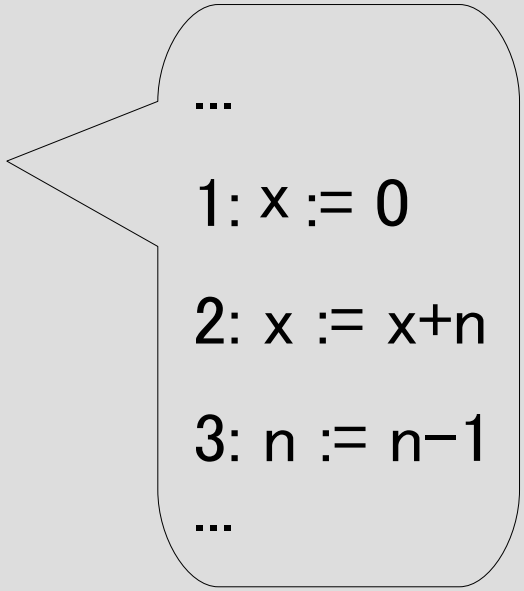
A a U b = false



A very simple example of CTL formula

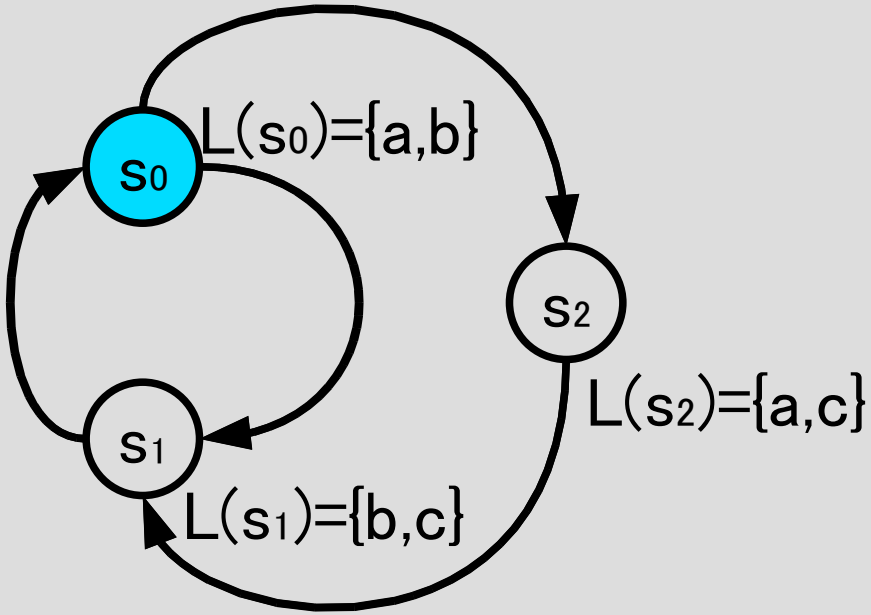
- Features
  - Conciseness, power of expression, and efficiency are excellent.
  - Easy to be used to analyze program.
- An example:

  ¬EX (E¬ def(x) U use (x))

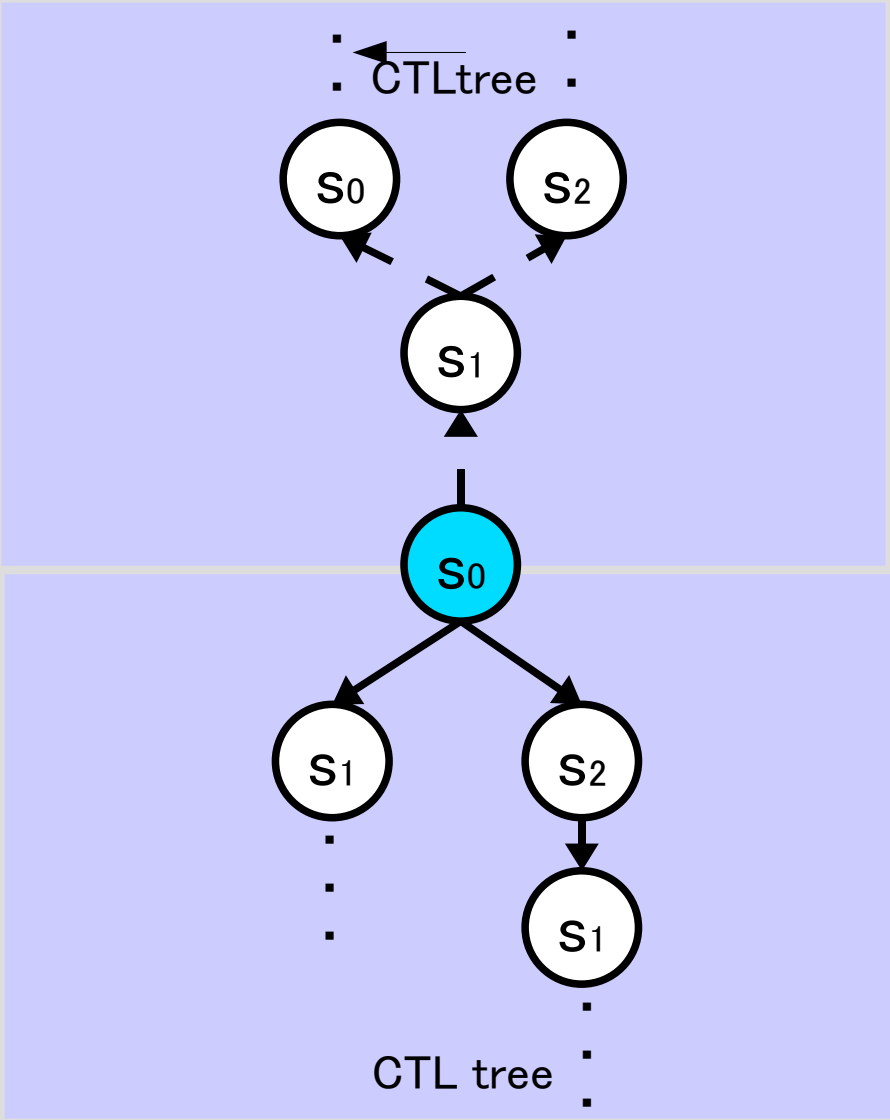


※ CTL−FV is denoted as CTL when it is not ambiguous.

Kripke structure and bidirectional CTL



Kripke structure :
- S: states
- $R \subseteq S \times S$: transition relations
- L: $S \rightarrow 2^{Prop}$



Bidirectional CTL tree

note : it is different with CTL syntax tree

Program and its Kripke model

0: read n

1: x := 0

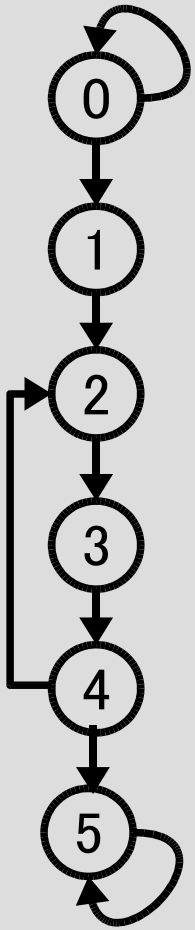
2: x := x+n

3: n := n−1

4: if n > 0 goto 2

5: write x

program



S: instructions of program
{0: read n, 1: x := 0,...}

R: transition relations
{0→1, 1→2,..., 4→2,...}

L(n): propositions
Lπ(0): {def(n),...}
Lπ(1): {def(x),...}
Lπ(2): {def(x), use(x), use(n),...}
Lπ(3): {def(n), use(n),...}
Lπ(4): {use(n),...}
Lπ(5): {use(x),...}

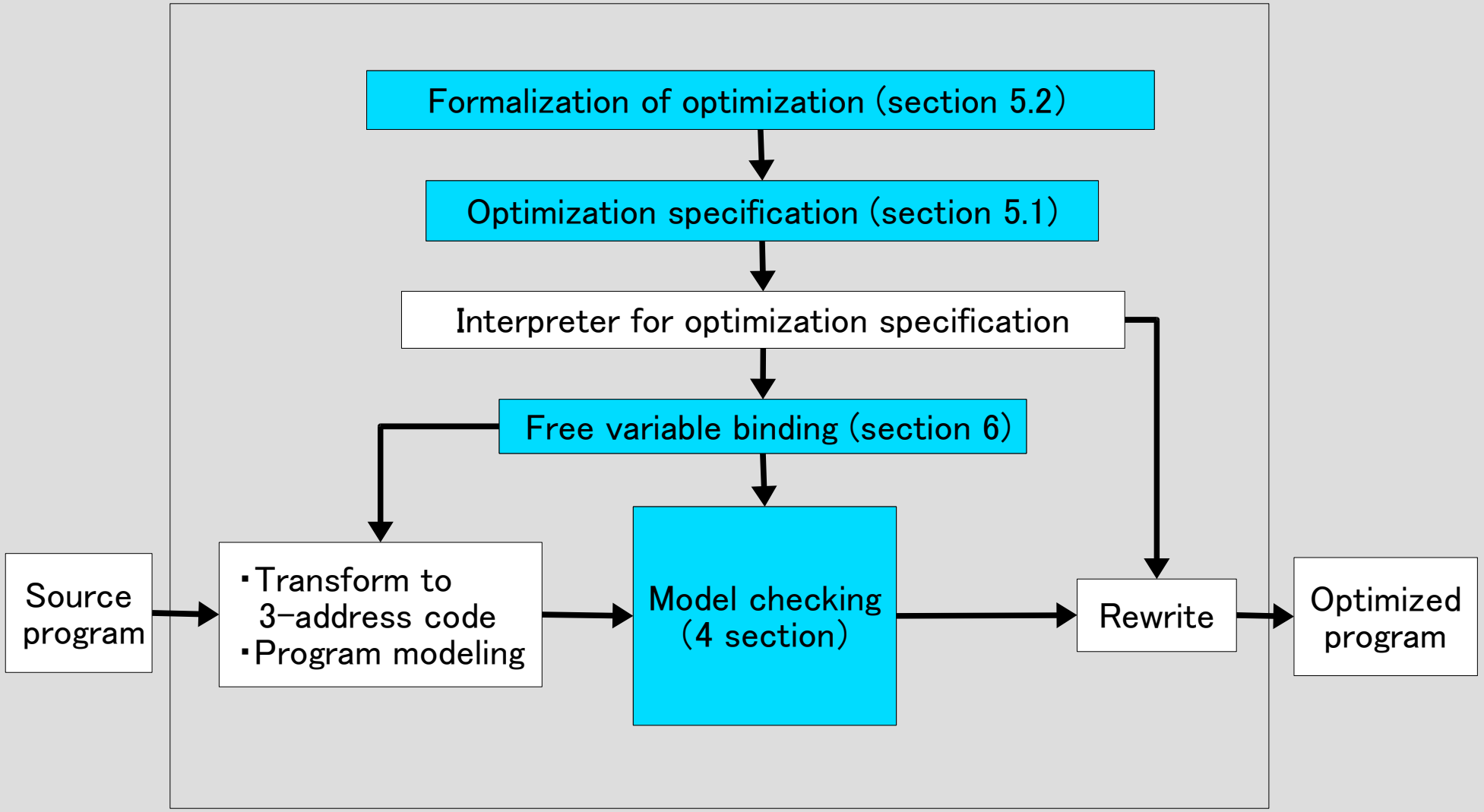
Values of the Kripke structure

Kripke model of the program

※ this program is the example for the following slides

Our research

Overview of the system



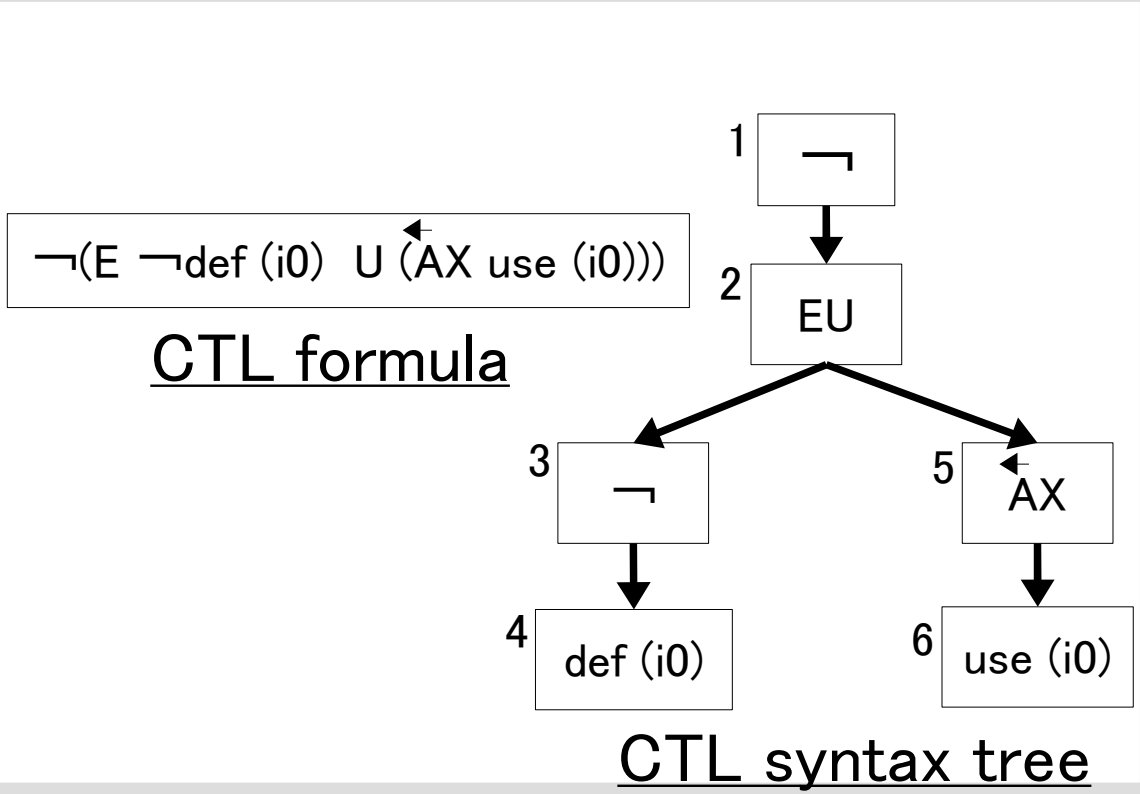
※ I will explain the important part colored with blue only.
※ The section number corresponds to the manuscript.

Model checker

- CTL formula Analysis
  - It can be shown with a syntax tree. The leaves are propositions and the nodes will be given number with depth priority.
  - Future temporal operators are checked with CTL tree and past temporal operators are checked using CTL tree.

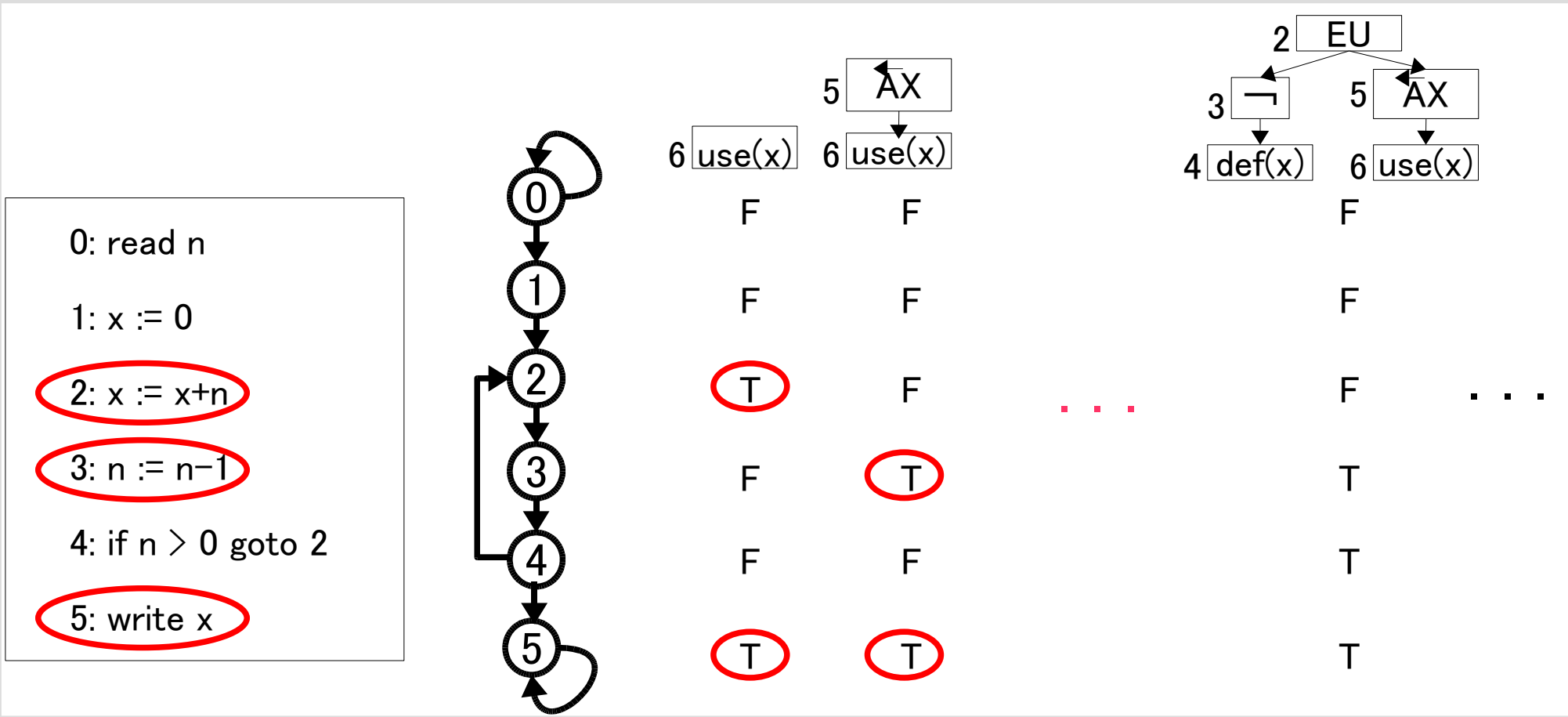
※ CTL syntax tree is different with CTL tree.

※ It is a simple bidirectional bidirectional CTL and mean nothing.



- Model checking

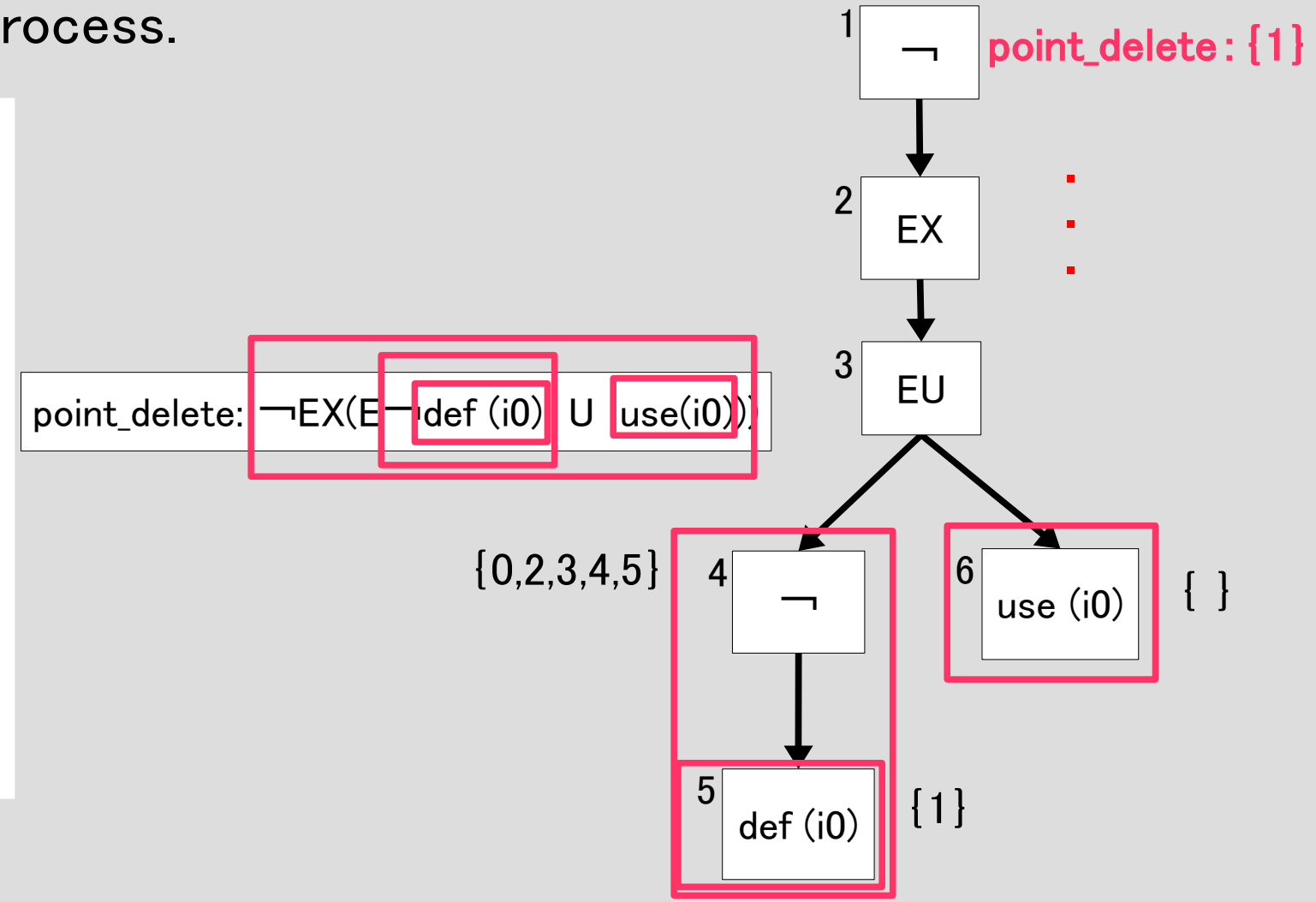
  A bottom-up calculation is done in the reverse order of the node numbers of the syntax tree given at the analysis stage.



- An example of model checking for dead code elimination

  The results of the model checking must be stored for the follow-ing rewriting process.

```
int sum()
{
0 int i0, i1, i2, i3;
1 i0 = 1;
2 i1 = 2;
3 i2 = 3;
4 i3 = i1 + i2;
5 return i3;
}
```



- Computational Complexity of Model Checking

  $O(\phi \times (|S|+|R|))$

  ϕ : the number of nodes of CTL syntax tree

  S: the number of nodes of source program model.

  R: the number of transition relations of program model.

  Generally, it is proportional to the size of CTL formula and the number of instruction of program.

return

Optimization specification

- The specification of optimization consists of three parts:
  - MATCH: specifies the pattern of target instructions.
  - CONDITION: states the conditions that must hold when an optimizing is processed
  - PROCESS: states how to process the instructions or edges that satisfy the conditional formulas in the CONDITION part.

- Format of the specification

  MATCH

  variable := expression

  CONDITION

  point_string : CTL formula (partial formula or condition formula)

  edge_string : point_string → point_string

  PROCESS

  point_string : Command statement

  point_string : Replace expression → expression

  edge_string : EdgeSplit statement

MATCH

v := b (variable := binary expression)

CONDITION

point_comp: use(x+y) ∧ trans(x+y)

point_spavin: . . .

point_spantout: . . .

point_insert: point_comp ∧ ¬point_spavin ∧ point_spantout

point_replace: . . .

edge_split: point_spavin → point_spantout

PROCESS

point_insert: Insert Before temp := x+y

point_replace: Replace x+y → temp

edge_split: Edge Split temp := x+y

If the code is:

…

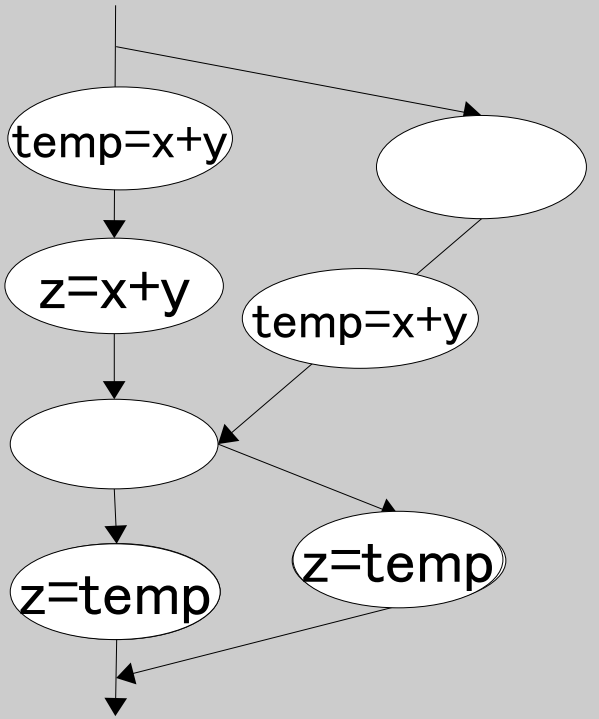
1: x := 0

2: z := x+y

Process for edge:

Process for instruction:

Insert “temp:=b” before the instructions that satisfy the formula named “point_insert” in the CONDITION.



- Features of the specification
  - Even complex optimization formulas that can deal with real-world optimization can be written very naturally and easily.
  - Efficiency can be improved because free variables for instruction numbers are not referred to.

return

Formalization of optimization with our specification language

- Specification based on the optimization condition

  The example for dead code elimination will be:

  $\neg EX\,(E\neg def(v)\ U\ use(v))$

  Its meaning is:

  $\neg EX\,(E\neg def(v)\ U\ use(v))$

From the starting point. it not **E**xist a ne**X**t state that **E**xist a path where **N**ot **def**ine (v) **U**ntil **use** (v)

- Specification based on the dataflow equation

$DEAD(B) = \{x \mid x \notin \bigcup_{S \in succ(B)} LIVE(S)\}$ $LIVE(B) = USE(B) \cup [\{\bigcup_{S \in succ(B)} LIVE(S)\} - KILL(B)]$ $USE(B) = \ldots$ $KILL(B) = \ldots$	point_use: use(x) point_kill: ... point_live: ... point_dead: ...

Dataflow equation (left) and CTL formulas (right) of dead code elimination

- Dataflow equation for PRE of Paleri et al.

$$
\begin{aligned}
TRANSP_i &= trans(e) \\
COMP_i &= use(e) \cdot TRANSP_i(e) \\
ANTLOC_i &= use(e) \cdot TRANSP_i(e) \\
AVIN_i &= \begin{cases} false & \textbf{if } entry \\ \prod_{j \in preds(i)} AVOUT_j & \textbf{if } otherwise \end{cases} \\
AVOUT_i &= COMP_i + AVIN_i \cdot TRANSP_i \\
ANTOUT_i &= \begin{cases} false & \textbf{if } exit \\ \prod_{j \in preds(i)} ANTIN_j & \textbf{if } otherwise \end{cases} \\
ANTIN_i &= ANTLOC_i + ANTOUT_i \cdot TRANSP_i \\
SAFEIN_i &= AVIN_i + ANTIN_i \\
SAFEOUT_i &= AVOUT_i + ANTOUT_i \\
SPAVIN_i &= \begin{cases} false & \textbf{if } i = entry \text{ or } \neg\, SAFEIN_i \\ \sum_{j \in preds(i)} SPAVOUT_j & \textbf{if } otherwise \end{cases} \\
SPAVOUT_i &= \begin{cases} false & \textbf{if } i = entry \text{ or } \neg\, SAFEOUT_i \\ COMP_i + SPAVIN_i \cdot TRANSP_i & \textbf{if } otherwise \end{cases} \\
SPANTOUT_i &= \begin{cases} false & \textbf{if } i = entry \text{ or } \neg\, SAFEOUT_i \\ \sum_{j \in preds(i)} SPANTIN_j & \textbf{if } otherwise \end{cases} \\
SPANTIN_i &= \begin{cases} false & \textbf{if } i = entry \text{ or } \neg\, SAFEIN_i \\ ANTLOC_i + SPANTOUT_i \cdot TRANSP_i & \textbf{if } otherwise \end{cases} \\
INSERT_i &= COMP_i \cdot \neg\, SPAVIN_i \cdot SPANTOUT_i \\
INSERT_{i,\,j} &= \neg\, SPAVOUT_i \cdot SPAVIN_j \cdot SPANTIN_j \\
REPLACE_i &= ANTLOC_i \cdot SPAVIN_i + COMP_i \cdot SPANTOUT_i
\end{aligned}
$$

- Our specification based on this dataflow equation

MATCH

v := e

CONDITION

point_comp: use(e) ∧ trans(e) ∧ ¬entry

point_avin: $\overleftarrow{A}X$ ($\overleftarrow{A}$ trans(e) U use(e))

point_avout: point_comp ∨ (point_avin ∧ trans(e)) ∧ ¬entry

point_antout: AX(A trans(e) U use(e))

...

point_edge2: ¬point_spavout

edge_split: point_edge1 → point_edge2

PROCESS

point_insert: InsertBefore temp := e

edge_split: EdgeSplit temp := e

point_replace: replace e → temp

In general, recursive relation can be written by dataflow equetions and solve from the point that has a fix value.
But CTL can not deal with recursive relations. it can only process path..

Ref : Lacey et al.'s specification

return

Free variable

- A free variable appears in a conditional formula and they must be bound to actual object (variables, expressions, and etc.) in the program.
- Binding is a exhaustive algorithm

  If the CTL formula is:

  ¬EX (E¬def (v)) U use(e)) (v: variables e: expression)

  and variables and expressions in program are:

  { x, n,...}, {x+n, n−1,...}

  so, v and e will be bound as follows:

  { v→x, e→x+n}

  { v→x, e→n−1}

  { v→n, e→x+n}

  ...

  Model checking is necessary to be done on every combination.

- Computational complexity of free variable binding

  $O(m^n)$

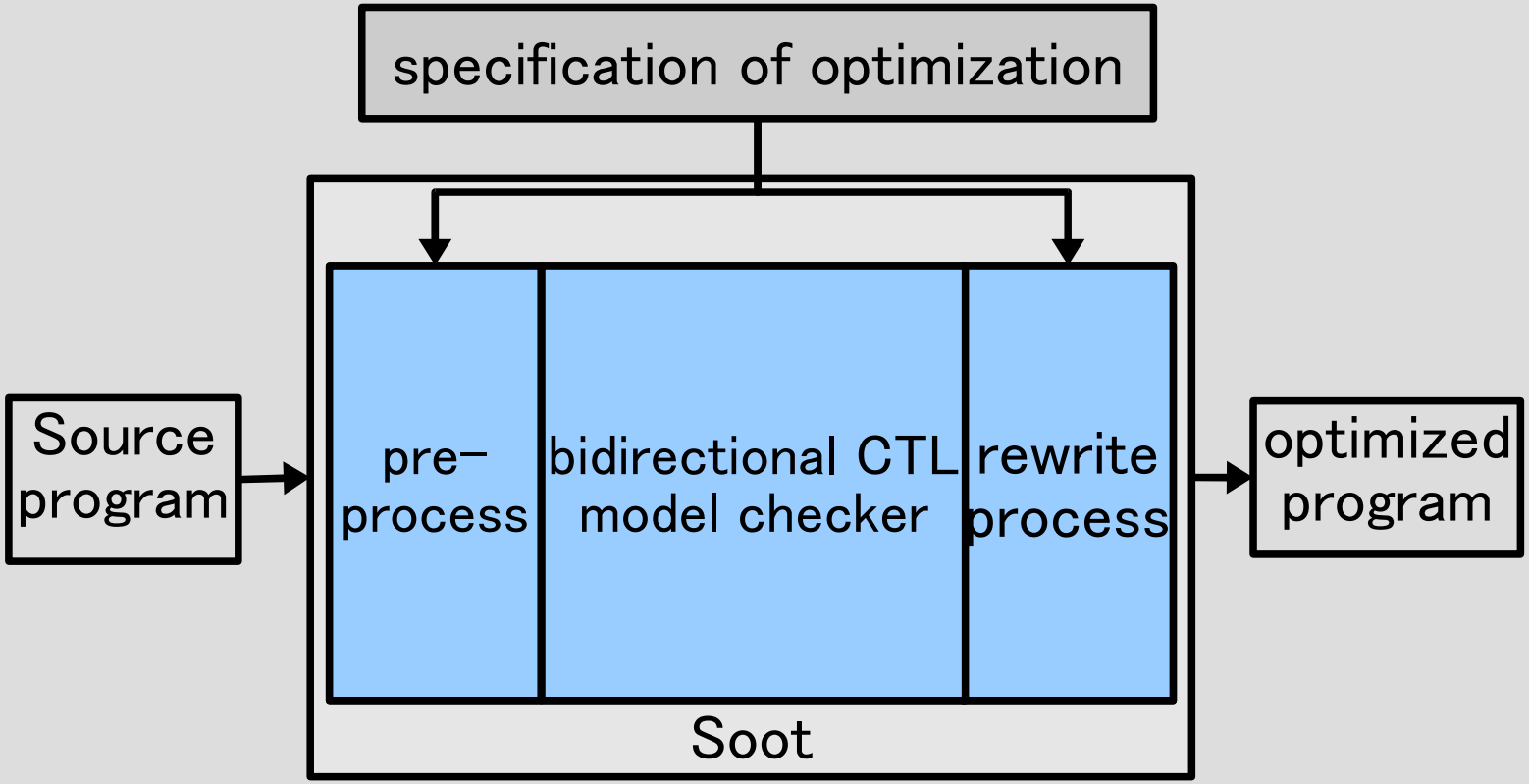
  n: number of free variables in conditional formulas

  m: number of objects that can be the target of binding, such as variables or expressions in the program.

  But the Complexity can be lowered at MATCH stage actually.

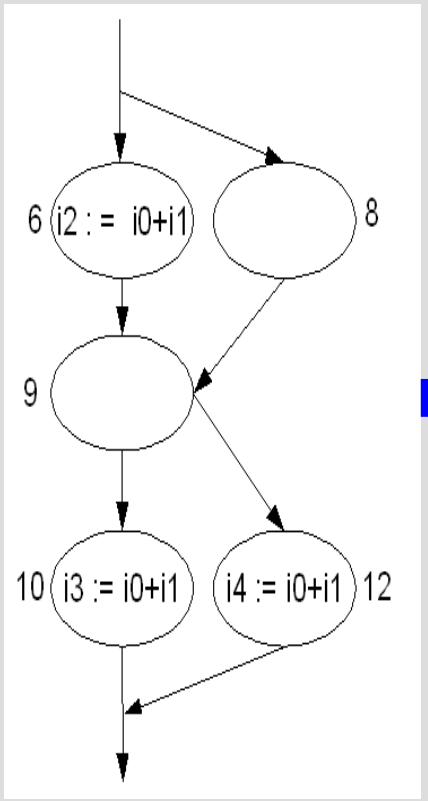
return

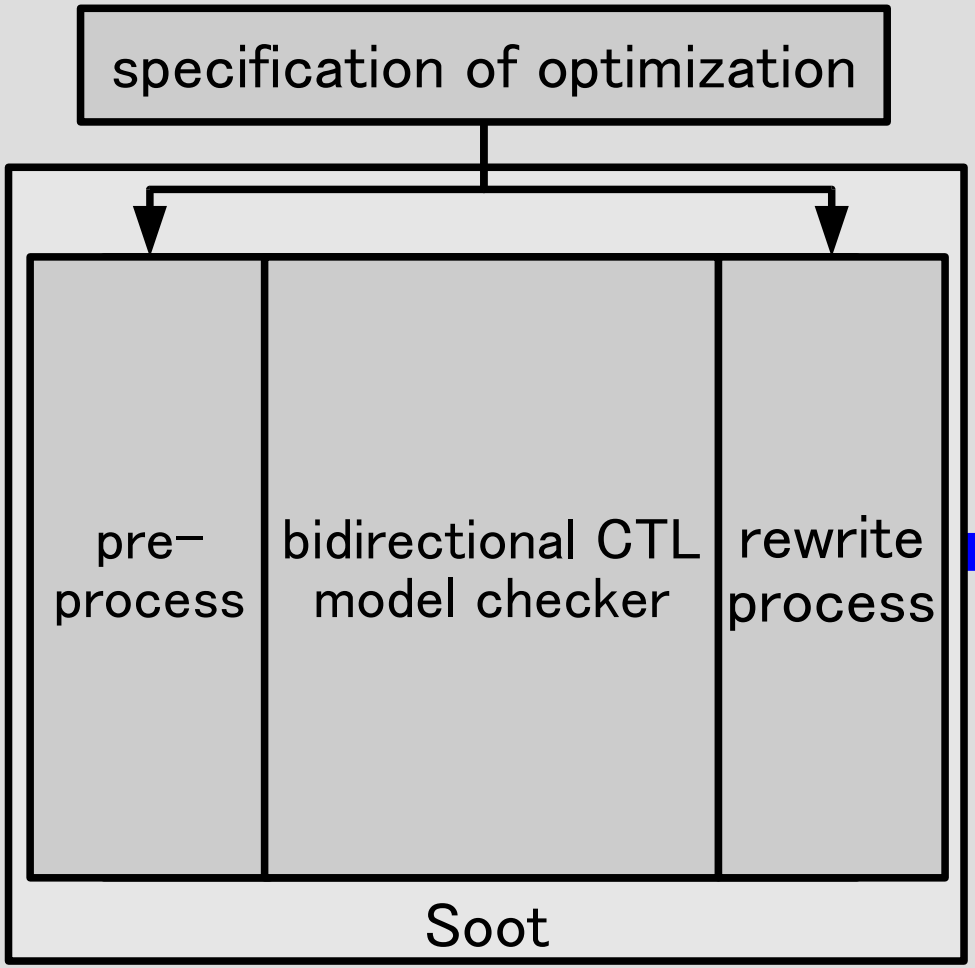
Summarize the system

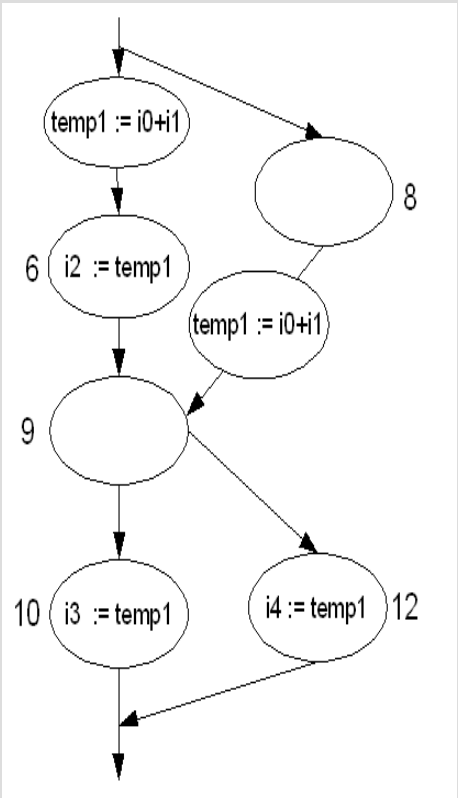


- Pre-process:
  - Making the Kripke model
  - Binding free variables

- An example of optimization for PRE







- Computational complexity of the optimizer

$$\mathbf{O}(\phi \times (|S|+|R|) \times |S|^{n})$$

ϕ : the number of nodes of the CTL syntax tree

S: the number of nodes of source program model.

R: the number of transition relations of program model.

n: the number of free variables in the CTL formula

Experiments and discussion

Experiments

- Our experimental data were acquired by using the 7 benchmarks of SPECjvm98 and Okumura's Java programs.
- Experimental Environment
  - CPU: Celeron 2GHz
  - Memory: 512MB
  - Soot: version 2.2.0
- JVM options: -Xint -Xms128m -Xmx128m (to exclude the influence of JIT and the memory)

- We have made some progress after the acceptance, so the result of the experiments is a bit different from that of the manuscript.
  - Modified the algorithm of AU
  - Avoid repeat calculation

    This cause a great improvement because Soot does inline expansion everytime when the procedure is called.
  - Exception is very convenient when escape from recursive procedure, but it consumes time. Therefore, we rewrite this kind of procedure.

- Optimization applied by our system:
  - Partial redundancy elimination (includes common subexpression elimination and loop invariant code motion), copy propagation (includes constant propagation),dead code elimination.
- Optimization applied by Soot (for comparison):
  - common subexpression elimination, partial redundancy elimination, copy propagation, constant propagation and folding, conditional branch folding, dead assignment elimination, unreachable code elimination, unconditional branch folding, and unused local elimination.

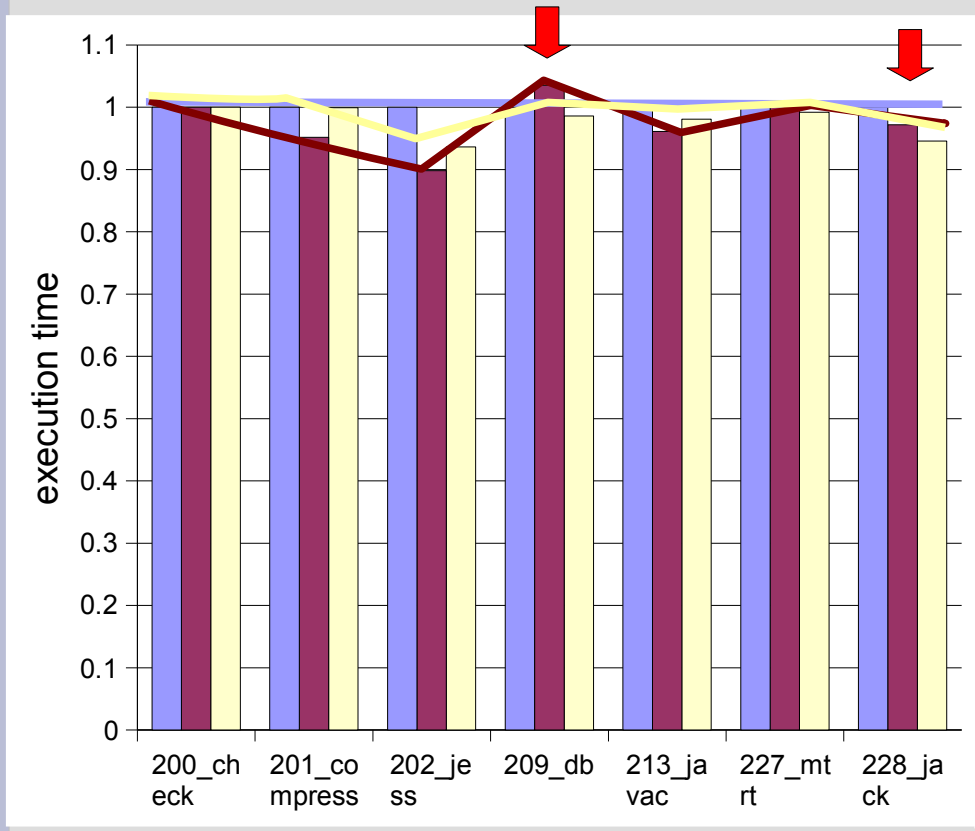
※ We use Soot as a reference for comparison

- The optimization time for the SPECjvm98 benchmarks and Oku-mura's Java programs. (unit: millisecond)

test code/size of code		Binding	Model checking	Rewrite	Other	Total
Bench mark	200_check/5145	140	2138	48	939	3265
	201_compress/5050	328	4326	32	1236	5922
	202_jess/26674	296	13027	125	3624	17072
	209_db/5316	158	2687	31	1095	3971
	213_javac/57937	529	24797	46	6592	31964
	227_mtrt/9713	264	18080	93	1810	20247
	228_jack/21895	499	236069	202	3363	240133
test code	PiByMachin/108	125	344	16	46	531
	CubeRoot/224	94	408	15	170	687
	Cardano/242	110	1062	31	109	1312
	CountingSort/175	125	468	15	79	687
	Nqueens/247	110	656	16	93	875
	Jacobi/1061	171	3690	32	482	4375
	LogE/1496	109	2362	48	263	2782
	Fibonacci/206	109	344	16	187	656
	Exp/1045	141	1439	32	327	1939

- The results indicate:
  - The optimization time is from 4 seconds to 4 minutes in 7 of the benchmarks.
  - It is slow compared to common compilers that take from milliseconds to several seconds but it is quite fast compared with results of previous work (some of which cannot perform such optimizations) .
  - Although optimizer by CTL was assumed to be impractical, we have implemented it within 4 minutes.

- Comparison of execution times of object code before and after optimization by our system and Soot (execution time without optimization is normalized to 1)



SPECjvm98

Okumura's test program

No optimization

After optimization by Soot

After optimization by ours

- The results indicate:
  - Optimization by our system has a modest effect, although our optimization implements only a part of the optimization applied in Soot.
  - There are a few programs where our technique beat Soot.

- Comparison with related work (about optimization)
  - Comparison with Lacey

    Lacey didn't give any data and the specification is complex to imitate in our system, we gave up to compare with Lacey's work
  - Comparison with Lerner

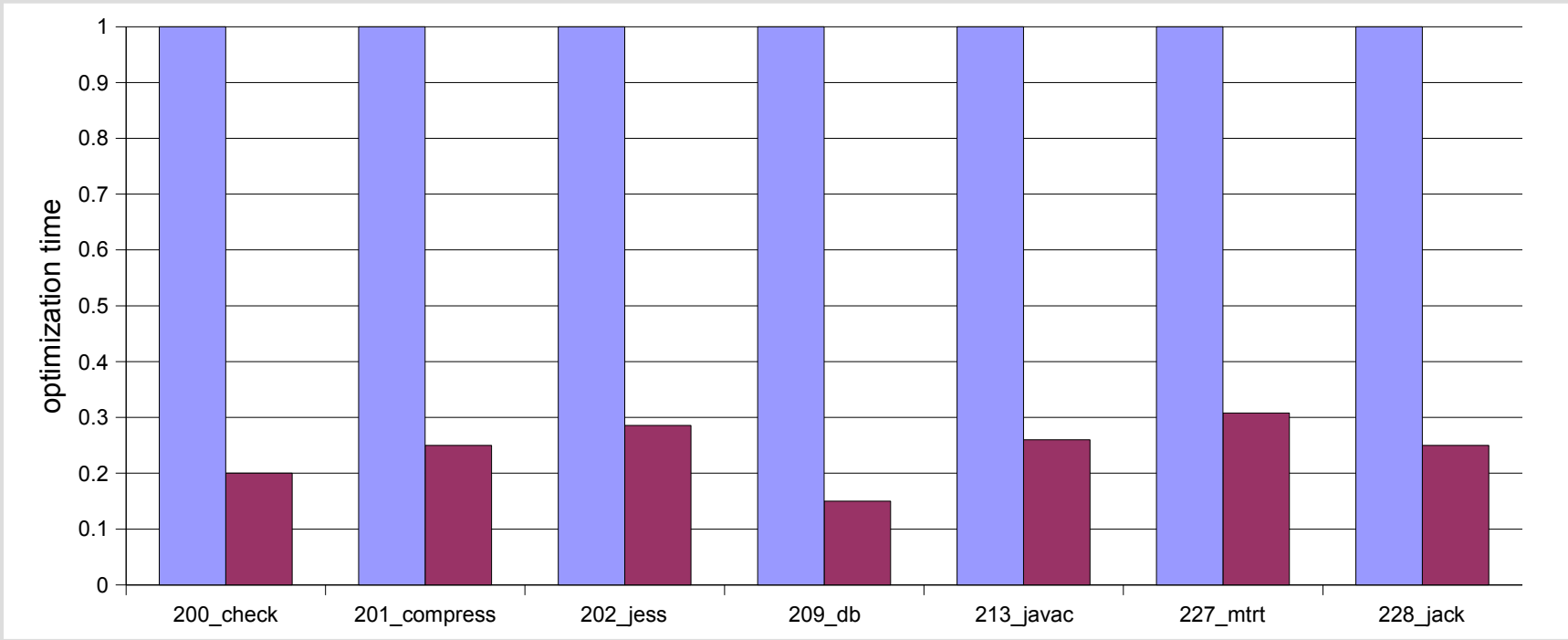
    It is a domain specific optimizer called Rhodium which can be semi-automatically verified. But it is different from ours. So we didn't do the comparison.

Ref : Lacey

Ref : Rhodium

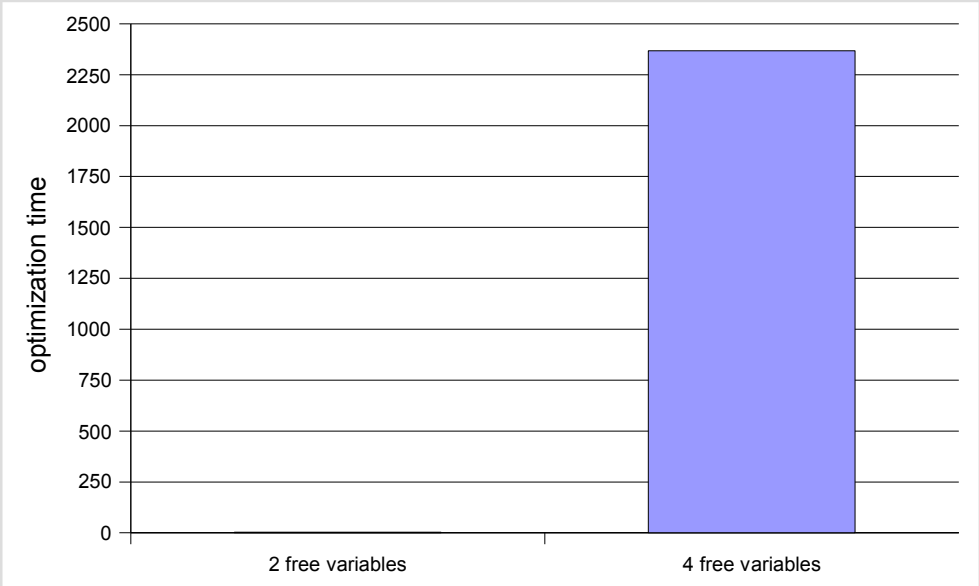
- Comparison with Ban's work

We used the formula of copy propagation involving past operators to compare with Ban's method. (Ban's optimization time is normalized to 1)



- Optimization time explosion

The optimization time exploded from 4 seconds to 39 minutes when free variables increased from 2 to 4 using different formulas for the same optimization.



Discussion

- Expressive power of CTL
  - Optimization can be specified easily and concisely in several lines by CTL.
  - The expressive power of CTL formula is inferior to common optimization algorithms in some cases.
    - An example is when the algorithm cannot be represented by CTL such as conditional constant propagation.
    - Time-optimal property of partial redundancy elimination.

- Efficiency of the compiler optimizer
  - Free variables greatly influence the efficiency of the system as mentioned before.
  - Model checking is an exhaustive algorithm. Therefore, its efficiency is inferior to that of common algorithm-based compiler optimizers.

★ The problem caused by free variables is the most important.

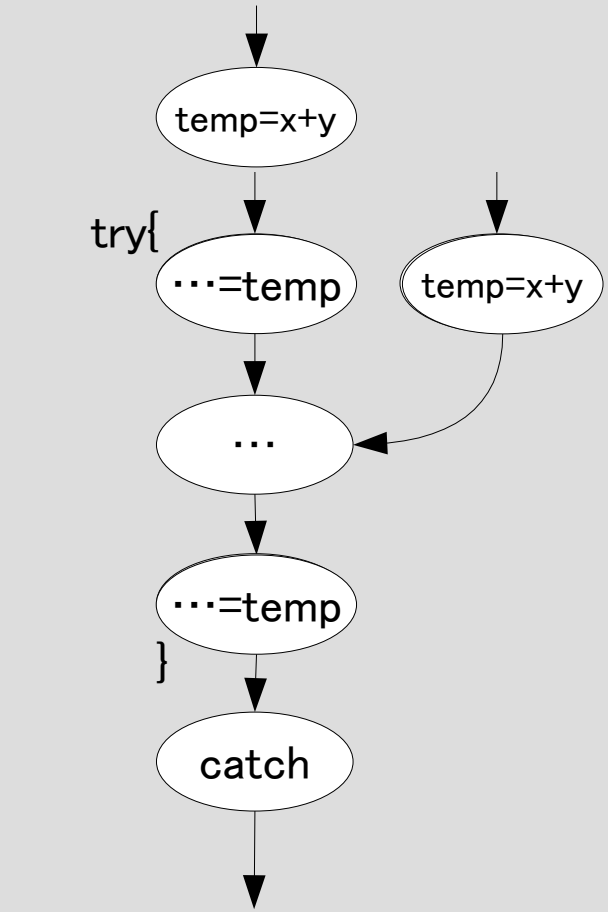
- Effectiveness of optimization

  It is limited by the following cases:

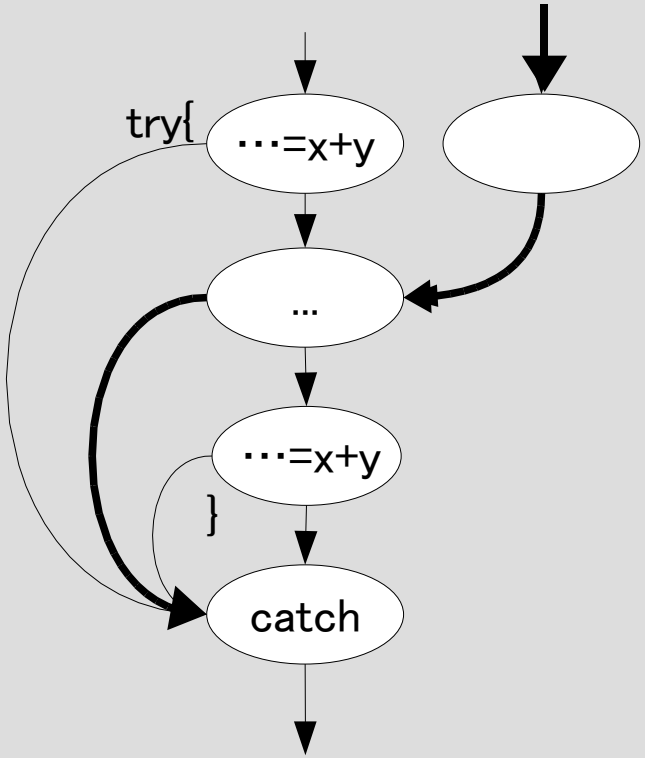
  - Move of instruction that across an exception point
  - Instructions that may cause run-time exceptions.
  - Optimization that cannot be prototyped.

★ The problem caused by exception point is the most important.

- Obstruction of optimization by exceptions



Optimizing on BriefGraph



Optimizing on CompleteGraph

Evaluation of our research

- Optimizer with CTL can be evaluated by specification (effective-ness), verification, efficiency. Because there is no data using the benchmarks besides ours. The follows is just our selfish judg-ment.

Item / work	Specification	Verification	Efficiency	Feature
Lacey	O	O	x	- Introduced CTL-FV - Describe the proposal and prove the correctness by hand - Only a part of real-world optimizers - No experimental data
Yamaoka	x	O	x	It can only deal with dead code elimination
Ban	x	O	x	The model checking time is considerable
Lerner	◎	O	-	It use a domain specification language to semi-auto verify
Our work	◎	O	O	- Practical time - Modest effectiveness - A concise specification language

Future work and conclusion

Future work

Three categories:

- Reducing optimization time
  - Making model checking faster

    Binary decision diagrams (BDD), partial evaluation or bit vector
  - Reducing or eliminating free variables
- Improving the effectiveness of the optimized program
  - Overcoming the obstruction caused by exceptions
  - Detailed analysis (loops, for and goto statements, etc.)
  - Specifying complex optimizations (conditional constant propagation, alias analysis etc.)
- verification
  - Proving the optimizations are correct

Conclusion

- We developed a Java optimizer by CTL
  - The optimization specification language is very expressive. It is very easy to describe a complex rewrite rule.
  - The compile time is practical.
  - The optimizer has a modest effectiveness.

  It is now close to optimizers using traditional algorithms.
- It is the first time that experimental data using the benchmarks and the test programs has been measured.
- Some problems of optimization by CTL were clarified.

- Our research achieved considerable progress in this field.
- Optimizer by CTL are expected to be a new way of optimizing programs or supplementing the current optimizers.

Thank you