Refinement-Based CFG Reconstruction from Unstructured Programs

Sébastien Bardin, Philippe Herrmann, Franck Védrine

CEA LIST
(Paris, France)

Overview

Automatic analysis of executable files

- recent research field [Codesurfer/x86, SAGE, Jakstab, Osmose, etc.]
- many promising applications (COTS, mobile code, malware, etc.)

A key issue : Control-Flow Graph (CFG) reconstruction

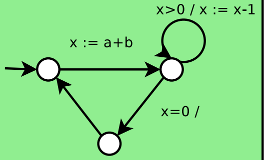
- prior to any other static analysis (SA)
- must be safe : otherwise, other SA unsafe
- must be precise : otherwise, other SA imprecise

This talk is about CFG reconstruction (from executable files)

- safe and precise technique
- based on abstraction-refinement

Binary code analysis

Model



Source code

```
int foo(int x, int y) {
 int k= x;
 int c=y;
 while (c>0) do {
   k++;
   c--;}
 return k;
}
```

Assembly

```
_start:
  load  A 100
  add B A
  cmp B 0
  jle label

label:
  move @100 B
```

Executable

```
ABFFF780BD70696CA101001BDE45
145634789234ABFFE678ABDCF456
5A2B4C6D009F5F5D1E0835715697
145FEDBCADACBDAD459700346901
3456KAHA305G67H345BFFADECAD3
00113456735FFD451E13AB080DAD
344252FFAADBDA457345FD780001
FFF22546ADDAE989776600000000
```

Binary code analysis is useful !

Always available

- commercial off-the-shelf software
- mobile code (including malware)
- third-party certification

Faithful

- optimising compilers and security
- optimising compilers and safety
- What You See Is Not What You eXecute [Reps 04,05]

Very precise

- worst case execution time, memory consumption, etc.

BUT binary code analysis is difficult ...

... i.e. more difficult than usual source-code analysis

Low-level semantic of data

- machine arithmetic, bitvector operations
- systematic usage of untyped memory (stack)

Low-level semantic of control

- no clear distinction between data and control
- no clean encapsulation of procedure calls
- dynamic jumps (`goto R0`)

  No easy (syntactic) recovery of the Control Flow Graph (CFG)

Diversity of architectures and instruction sets

- each ISA contains dozen of instructions
- lots of engineering work

BUT binary code analysis is difficult ...

... i.e. more difficult than usual source-code analysis

Low-level semantic of data

- machine arithmetic, bitvector operations
- systematic usage of untyped memory (stack)

Low-level semantic of control

- no clear distinction between data and control
- no clean encapsulation of procedure calls
- dynamic jumps (`goto R0`)

No easy (syntactic) recovery of the Control Flow Graph (CFG)

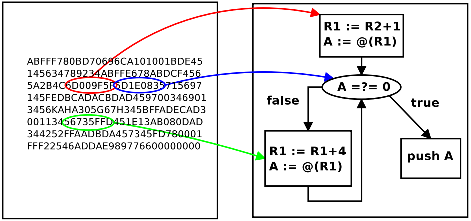
Diversity of architectures and instruction sets

- each ISA contains dozen of instructions
- lots of engineering work

CFG reconstruction

Input

- an executable file, i.e. an array of bytes
- the address of the initial instruction
- a basic decoder : exec f. $\times$ address $\mapsto$ instruction $\times$ size



Output : CFG of the program

CFG reconstruction (2)

Successor addresses are often syntactically known

- $\langle$ `addr : move a b` $\rangle \longrightarrow$
- $\langle$ `addr : goto 100` $\rangle \longrightarrow$
- $\langle$ `addr : ble 100` $\rangle \longrightarrow$

CFG reconstruction (2)

Successor addresses are often syntactically known

- ⟨ `addr : move a b` ⟩ → successor at `addr+size`
- ⟨ `addr : goto 100` ⟩ → successor at `100`
- ⟨ `addr : ble 100` ⟩ → successors at `100` and `addr+size`

CFG reconstruction (2)

Successor addresses are often syntactically known

- $\langle$ `addr : move a b` $\rangle \longrightarrow$ successor at `addr+size`
- $\langle$ `addr : goto 100` $\rangle \longrightarrow$ successor at `100`
- $\langle$ `addr : ble 100` $\rangle \longrightarrow$ successors at `100` and `addr+size`

But not always : successors of $\langle$`addr : goto a` $\rangle$?

CFG reconstruction (2)

Successor addresses are often syntactically known

- ⟨ `addr : move a b` ⟩ → successor at `addr+size`
- ⟨ `addr : goto 100` ⟩ → successor at `100`
- ⟨ `addr : ble 100` ⟩ → successors at `100` and `addr+size`

But not always : successors of ⟨`addr : goto a` ⟩ ?

Dynamic jump is the enemy !

Dynamic jumps are pervasive : introduced by compilers

- `switch`, function pointers, virtual methods, etc.

Unsafe approaches to CFG recovery

... current industrial practise ...

Linear sweep decoding [brute force]

- decode instructions at each code address

- miss every "dynamic" edge of the CFG
- may still miss instructions [too optimistic hypothesises]

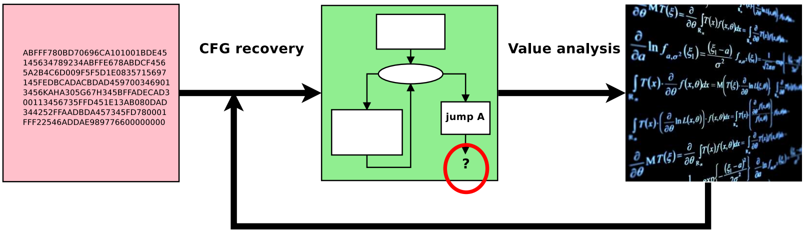
Recursive traversal

- decode recursively from entry point, stop on dynamic jump

- miss large parts of CFG

Safe CFG recovery

VA and CFG reconstruction must be interleaved



Very difficult to get precise : imprecision on jumps → extra propagation on false targets → more imprecision on value analysis → possibly more imprecision on jumps → ...

Existing safe approaches

CodeSurfer/x86 [Balakrishnan-Reps 04,05,07,...]

- abstract domain : strided intervals (+ affine relationships)

- imprecise : abstract domain not suited to sets of jump targets (arbitrary values from compiler)
- in practicse many false targets

Jakstab [Kinder-Veith 08,09,10]

- abstract domain : sets of bounded cardinality (k-sets)
- precise when the bound k is well-tuned

- not robust to the parameter k : possibly inefficient if k too large, very imprecise if k not large enough

Contribution

Key observations

- k-sets are the only domain well-suited to precise CFG reconstruction
- for most programs, only a few facts need to be tracked precisely to resolve dynamic jumps
- good candidate for abstraction-refinement

Contribution [VMCAI 2011]

- A refinement-based approach to safe CFG reconstruction
- An implementation and a few experiments
- The technique is safe, precise, robust and reasonably efficient

The problem

Unstructured Programs : $P = (L, V, A, T, l_0)$ where

- $L \subseteq \mathbb{N}$ finite set of code addresses
- V finite set of program variables, A finite set of arrays
- T maps code addresses to instructions
- l_0 initial code address
- instructions : assignments v `:=`e and $a[e_1]$ `:=`e_2, static jumps `goto` l, branching instructions `ite`$(cond, l_1, l_2)$, dynamic jumps `cgoto`(v)

Problem : compute an invariant of P such that no dynamic target evaluates to $\top$, or fail

- do not fail "too often"
- do not add "too many" false targets

Sketch of the procedure

abstract domain = k-sets

k-set cardinality bounds are local to each location

- gain efficiency through loss of precision
- still a global bound *Kmax* over local bounds

procedure : propagate forward until a dynamic target expression evaluates to $\top$, then try to refine the domain to avoid this $\top$ value

- domain refinement = increase some k-set cardinality bounds
- if no domain update then fail, else restart propagation with new domains

Refinement

For each target evaluating to $\top$

- follows backward data dependencies
- only interested in $\top$-values (other locations are safe until now)
- only interested in correcting initial causes of precision loss

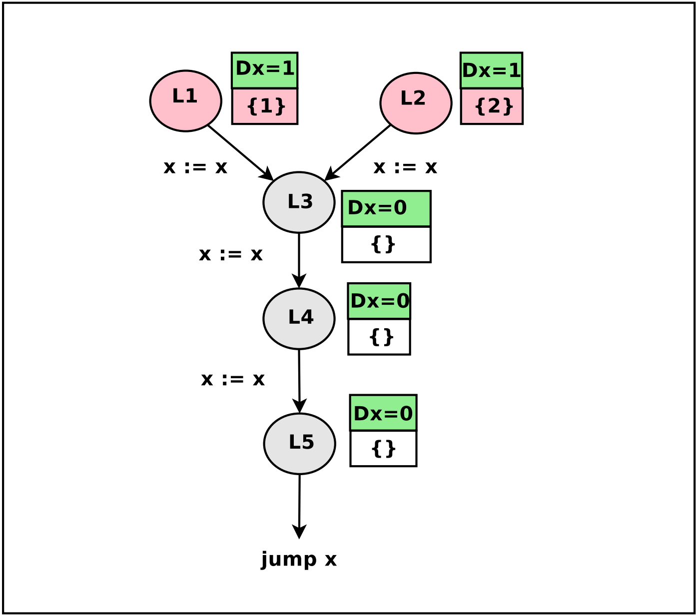
Finding the initial causes of precision loss

- add tags to $\top$-values, recording origin : $\top, \top_{init}, \top_{\langle c_1,\ldots,c_n \rangle}$
- initial causes of precision loss are of the form $\top_{init}, \top_{\langle c_1,\ldots,c_n \rangle}$

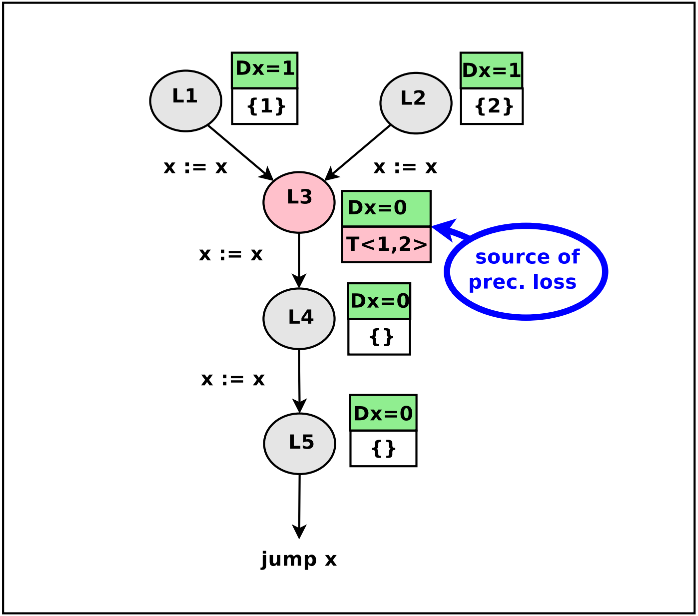
How to correct

- $\top_{init}$ cannot be avoided
- $\top_{\langle c_1,\ldots,c_n \rangle}$ may be avoided if $n \leq Kmax$ (set local bound to n)

Example

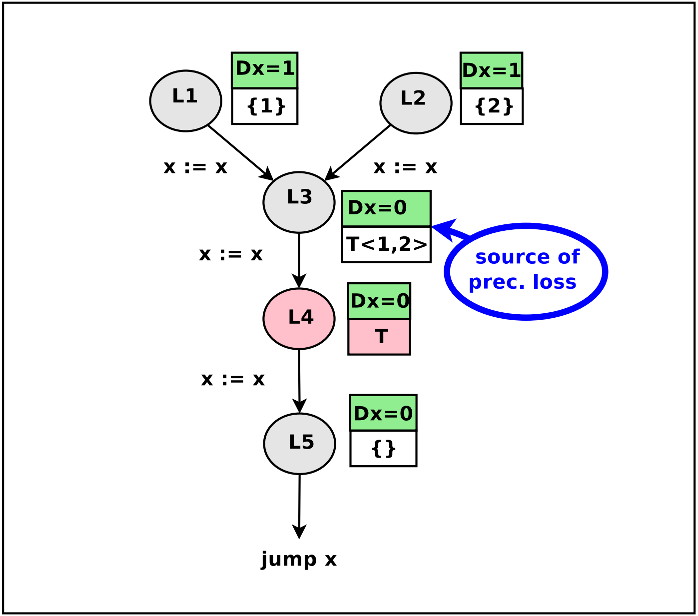


Example

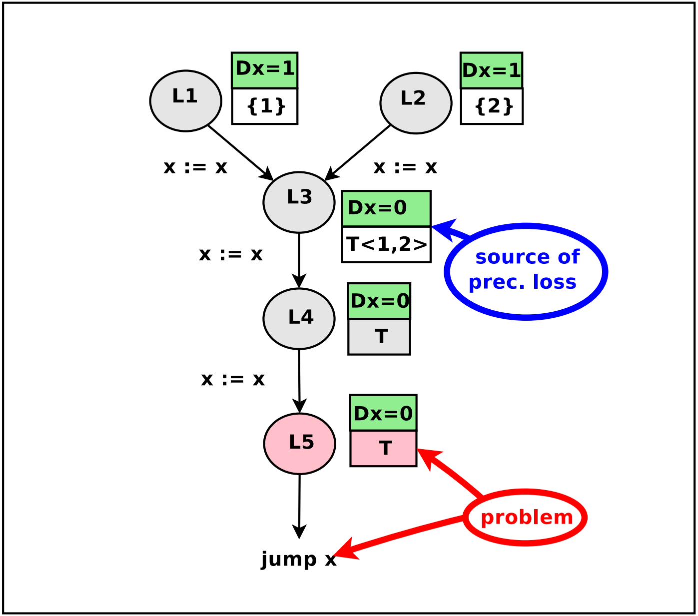


L1
Dx=1
{1}
L2
Dx=1
{2}
x := x
x := x
L3
Dx=0
T<1,2>
source of
prec. loss
x := x
L4
Dx=0
{}
x := x
L5
Dx=0
{}
jump x

Example

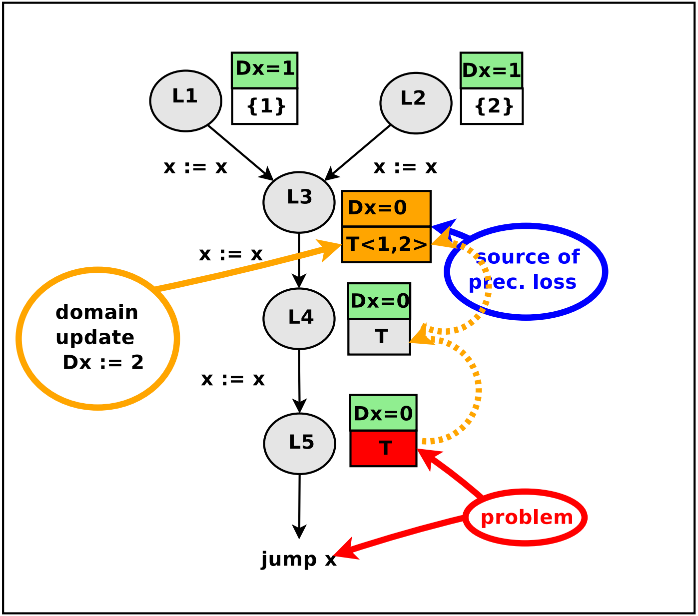

L1
Dx=1
{1}
L2
Dx=1
{2}
x := x
x := x
L3
Dx=0
T<1,2>
source of
prec. loss
x := x
L4
Dx=0
T
x := x
L5
Dx=0
{}
jump x

Example

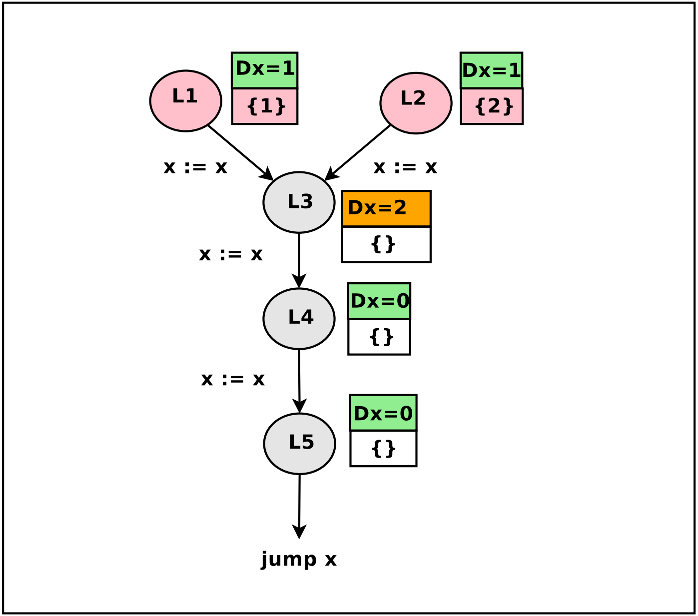


L1
Dx=1
{1}
L2
Dx=1
{2}
x := x
x := x
L3
Dx=0
T<1,2>
source of
prec. loss
x := x
L4
Dx=0
T
x := x
L5
Dx=0
T
problem
jump x

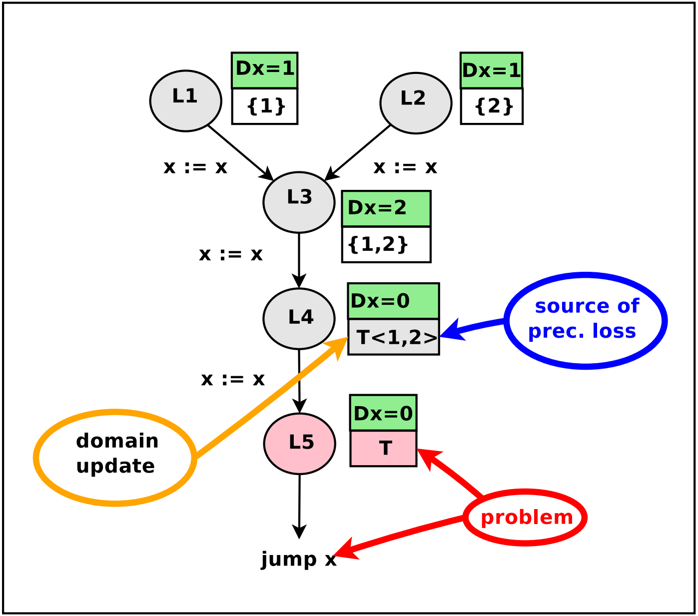
Example


L1
Dx=1
{1}
L2
Dx=1
{2}
x := x
x := x
L3
Dx=0
T<1,2>
source of
prec. loss
x := x
domain
update
Dx := 2
L4
Dx=0
T
x := x
L5
Dx=0
T
problem
jump x

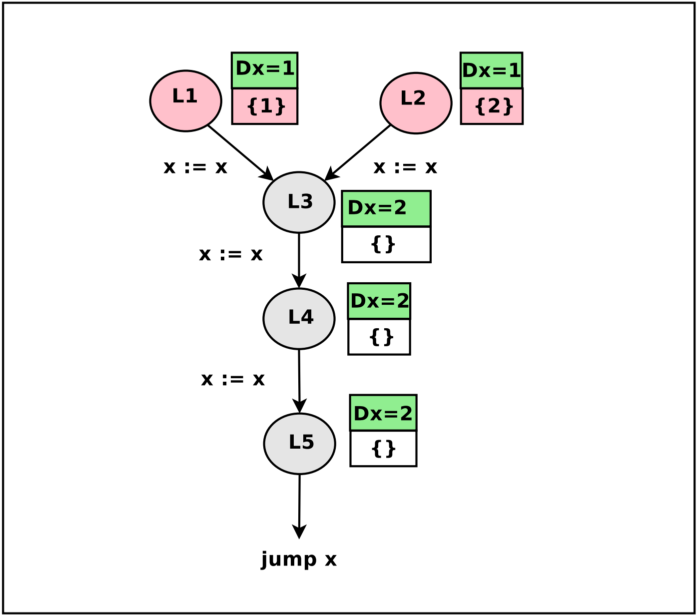
Example



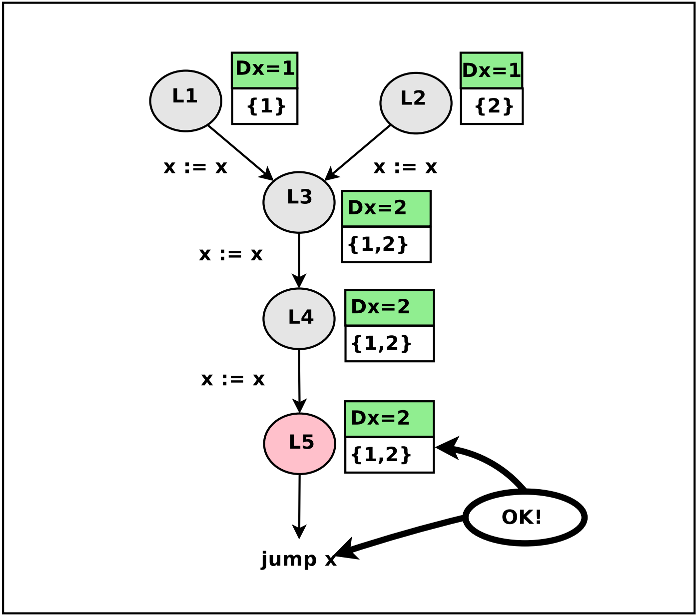
Example


L1
Dx=1
{1}
L2
Dx=1
{2}
x := x
x := x
L3
Dx=2
{1,2}
x := x
L4
Dx=0
T<1,2>
source of prec. loss
x := x
domain update
L5
Dx=0
T
problem
jump x

Example



Example



Some details

Too much refinement = inefficiency

A journal of the forward propagation

- record observed feasible branches / alias / dynamic targets
- prune backward data dependencies accordingly

Two possible failure policies during refinement

- optimistic : fails only when no local domain is corrected
- pessimistic : fails as soon as one "cause of precision loss" cannot be corrected

Basic theoretical properties

Soundness : returns either FAIL or an invariant such that no jump target evaluates to $\top$

Complexity : polynomial number of refinements

Precision : perfect relative precision for a non trivial subclass of programs (see next)

What about precision ?

Relative completeness (RC) : PaR is relatively complete if $PaR(P, Kmax)$ returns successfully when the forward k-set propagation with parameter $Kmax$ does

Bad news : no RC in the general case

- mainly because of control dependencies

Good news : RC for a non trivial subclass of programs

- non deterministic branches [new : only feasible branches]
- guarded aliases
- restricted class of operators : $+, -, \times k$ ok, but not $\times$
- RC even for the procedure with "pessimistic failure"

Experiments

Implementation : CFG reconstruction from 32-bit PowerPC (PPC)

Bench : Safety critical program from Sagem

- 32 kloc, 51 dynamic jumps, up to 16 targets a jump

Results

- precision : resolve every jump, only 7% of false targets

  (standard program analysis cannot recover better than between 400% and 4000% of false targets)

- robustness : results independent of *Kmax* (if large enough)

- locality : tight value of max-k, low value of mean-k

About efficiency

Terminates in 18 min [$\leq$ 5 min now]

- ok for a preliminary implementation
- already sufficient for some industrial application
- however (as expected) procedure inlining is an issue

1x - 3x faster than adequate k-set propag
3x - 5x faster than iterated k-set propag

- we expected more gap
- lots of redundant work from one refinement step to the other
- can probably be improved

Conclusion

We investigate safe CFG reconstruction from executable files

Results

- an original refinement-based procedure
- safe, precise, robust and reasonably efficient
- both theoretical and empirical evidence

Future work

- improve efficiency [inlining, redundant work]
- experiments on non-critical programs [dynamic alloc]
- ultimate goal : executables coming from large C++ programs

Backup 1

Relative completeness : why it does not work (general case)

let us suppose $Kmax = 1$

1. x :=1, goto 2
2. if x==1 then goto 3 else goto 4
3. t :=100, goto 5
4. t :=200, goto 5 // dead code
5. jump t

Backup 1

Relative completeness : why it does not work (general case)

let us suppose $Kmax = 1$

1. x :=1, goto 2 // x=$\top$
2. if x==1 then goto 3 else goto 4 // x=$\{1\}$
3. t :=100, goto 5 // x=$\{1\}$
4. t :=200, goto 5 // x=$\bot$ // dead code
5. jump t // t=$\{100\}$

Forward propagation with $Kmax = 1$ succeeds.

Backup 1

Relative completeness : why it does not work (general case)

let us suppose $Kmax = 1$

1. x :=1, goto 2 // $x=\top$
2. if x==1 then goto 3 else goto 4 // $x=\top$
3. t :=100, goto 5 // $x=\top$
4. t :=200, goto 5 // $x=\top$ // dead code
5. jump t // $t=\top_{\langle 100,200 \rangle}$

Forward propagation with $Kmax = 1$ succeeds.

Our procedure fails :

- believes that $(5, t)$ can take at least values $\{100, 200\}$
- do not notice that `else` branch infeasible

Backup 2

Relative completeness : why it works (restricted class)

- KSET(k) is as precise as KSET(Kmax), as long as there is no $\top$-cast
- loss of relative precision happens only because of $\top$-cast

$\Rightarrow$ on the restricted subclass, as long as no alias / jump evaluates to $\top$, KSET(k) and KSET(Kmax) computes the same proper k-sets

$\Rightarrow$ same aliases and same dynamic targets (if proper k-sets)

Actually, more powerful than RC ...