Towards Complete Specifications with an Error Calculus

Quang Loc Le
Joint work with A. Sharma, F. Craciun and W.N. Chin

National University of Singapore

NFM 2013, CA US
May 12, 2013

Specification and Verification

Specification provides formal documentation and can guide *code verification*.

```
data node { int val;  node next;}
```

```
1  int get_data(node ptr){
2    return ptr.val;
3  }
```

Method Specification:

```
requires  ptr↦node⟨d, p⟩
ensures   res=d;
```

Safety Verification

Code can be modular but complex.

```
 4  int ll_sum(node ll){
 5    if (ll ≠ null){
 6      int d = get_data(ll);
 7      return (d + ll_sum(ll.next));
 8    }
 9    else return 0;
10  }
```

Example above requires acyclic list with summation.

What specification to provide?

Safety Specification

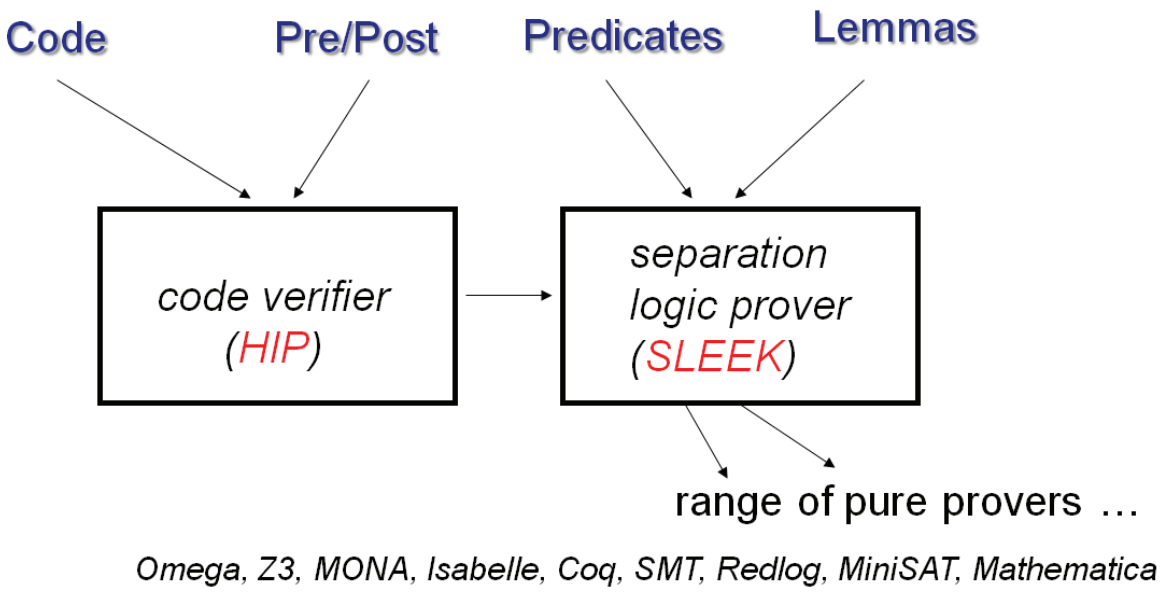
- Predicate for Acyclic List with Sum

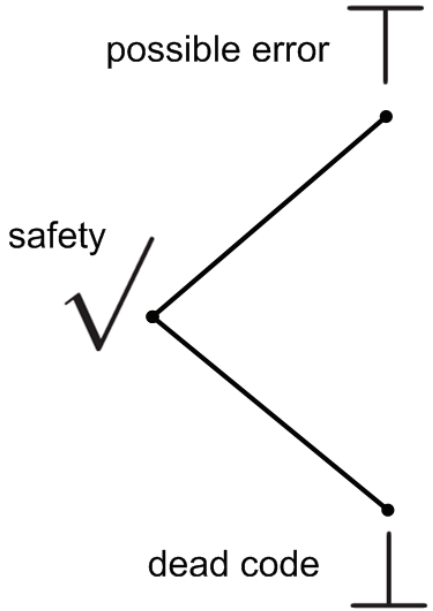
$$\begin{array}{l}\texttt{pred}\ \texttt{llist}\langle \texttt{root}, \texttt{S}\rangle \equiv \\ \quad (\texttt{root}{=}\texttt{null} \wedge \texttt{S}{=}\texttt{0}) \\ \vee\ (\exists\, \texttt{d}, \texttt{q}, \texttt{S}_1 \cdot \texttt{root}{\mapsto}\texttt{node}\langle \texttt{d}, \texttt{q}\rangle * \texttt{llist}\langle \texttt{q}, \texttt{S}_1\rangle \wedge \texttt{S}{=}\texttt{S}_1{+}\texttt{d})\end{array}$$

- Pre/Post Safety Specification

$$\begin{array}{l}\texttt{int ll\_sum(node ll)} \\ \quad \texttt{requires}\ \ \texttt{llist}\langle \texttt{ll}, \texttt{s}\rangle \\ \quad \texttt{ensures}\ \ \ \texttt{llist}\langle \texttt{ll}, \texttt{s}\rangle \wedge \texttt{res}{=}\texttt{s};\end{array}$$

HIP/SLEEK for Safety Verification





- symbolic execution based on separation logic
- safety guarantee : no memory errors, safe pre-conditions met, safe post-condition ensured.
- modularity: each function call captured via pre/post

Key Idea 1: Complete Specifications

Apart from safety verification, we are also interested in error scenarios to facilitate bug-finding.

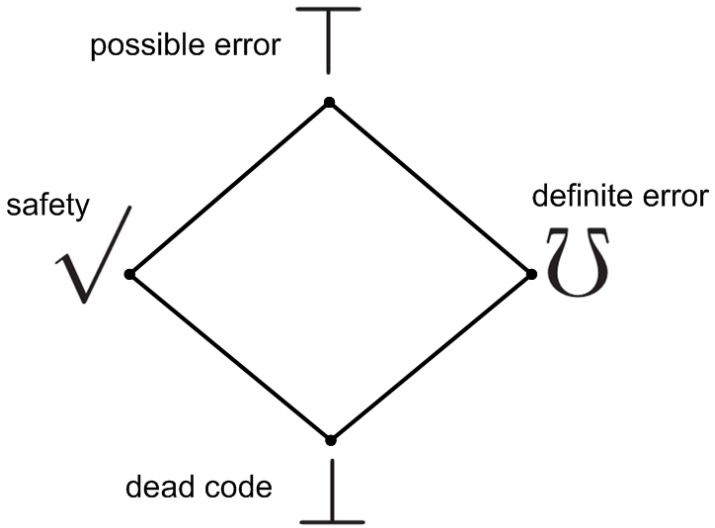
```
1  int get_data(node ptr){
2    return ptr.val;
3  }
```

```
int get_data(node ptr)
 requires ptr↦node⟨d, _⟩
 ensures res=d;
```

ptr = null?

Key Idea 1: Complete Specifications

Apart from safety verification, we are also interested in error scenarios to facilitate bug-finding.



We introduce new status $\mho$ to enable

- error scenarios be explicitly expressed
- definite errors be detected by verification.

Key Idea 1: Complete Specifications

```
1  int get_data(node ptr){
2    return ptr.val;
3  }
```

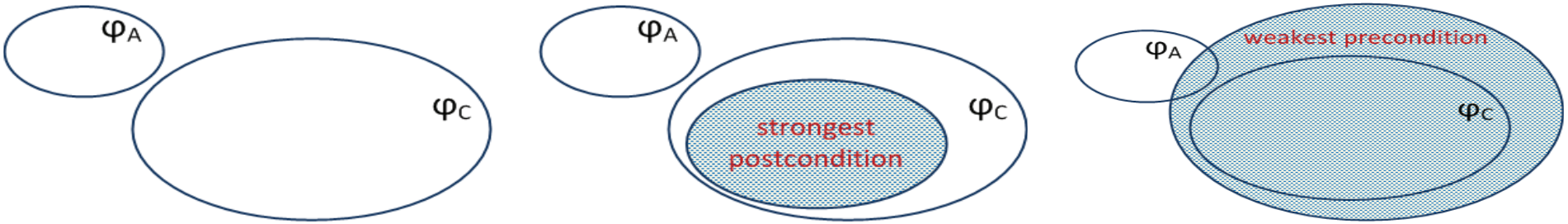
```
int get_data(node ptr)
 requires ptr↦node⟨d, _⟩
 ensures res=d;
```

```
int get_data(node ptr)
 requires ptr = null
 ensures (true) 𝔘;
```

Complete Specification

```
int get_data(node ptr)
 case {
   ptr ≠ null → requires ptr↦node⟨d, _⟩
                ensures (res=d) √;
   ptr = null → ensures (true) 𝔘;
}
```

Key Idea 2: Two Entailment Procedures for The Lattice



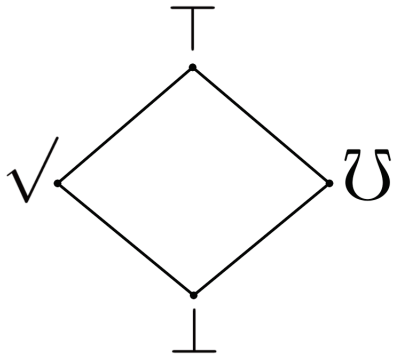
- Entailment Procedure for Checking: $\varphi_A \vdash \varphi_C$
- φ_C are postconditions or preconditions and may NOT be the best

Observation: in the scenarios that φ_A contradicts with φ_C

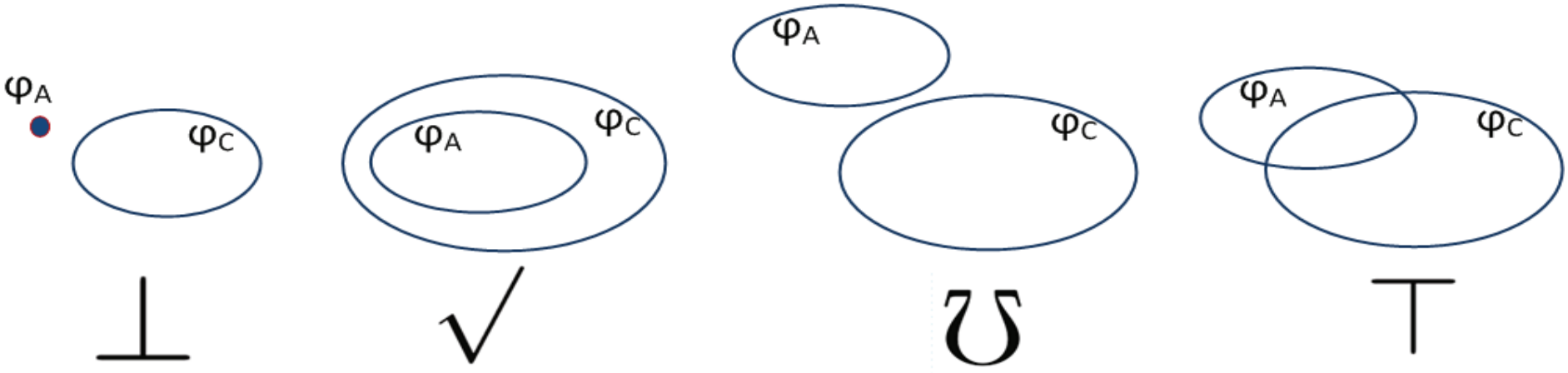
- when φ_C is a postcondition, it is sound to return an $\mho$ status
- when φ_C is precondition, it is unsound to return an $\mho$ status

We propose two entailment procedures: one for postconditions checking ($\vdash_{post}$) and another for precondition checking ($\vdash_{pre}$).

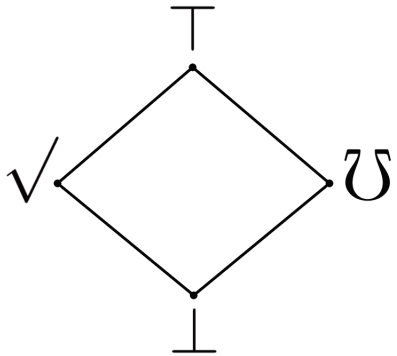
Key Idea 2: Two Entailment Procedures for The Lattice



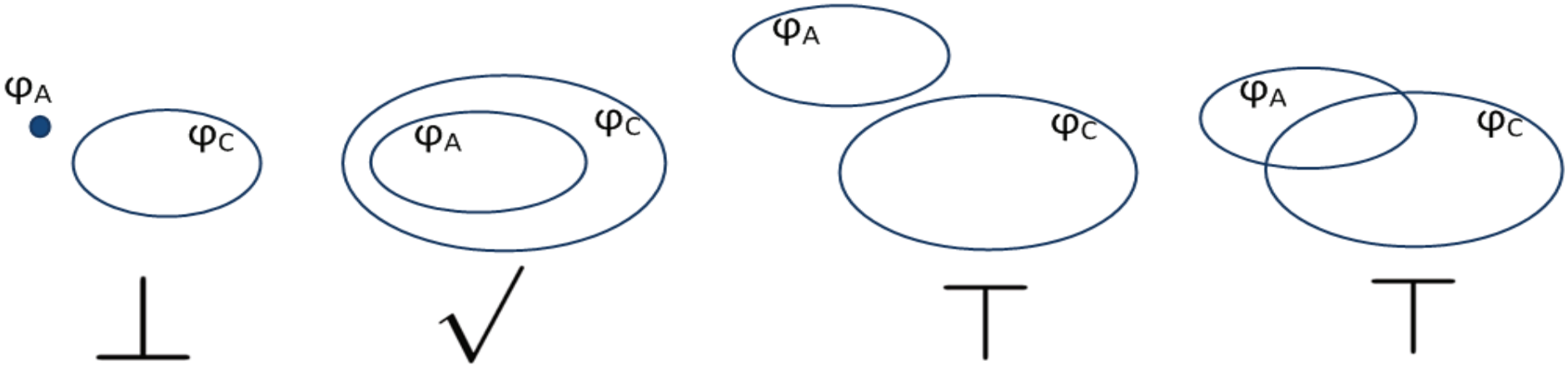
Entailment Procedure for Postcondition Checking: $\varphi_A \vdash_{post} \varphi_C$



Key Idea 2: Two Entailment Procedures for The Lattice

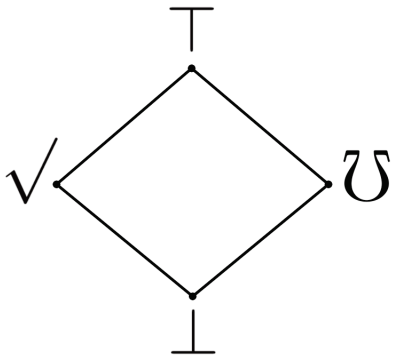


Entailment Procedure for Precondition Checking: $\varphi_A \vdash_{pre} \varphi_C$



How to get ʊ for preconditions proving at function calls?

Key Idea 2: Two Entailment Procedures for The Lattice




```
1. node get_data(node ptr){
2.  return ptr.val;
3.}
4.  int ll_sum(node x){
  ......
6.      int d = get_data(x);
  ......
10.}
```

```
int get_data(node ptr)
 case {
   ptr ≠ null → requires ptr↦node(d,_)
                ensures (res=d) √;
   ptr = null → ensures (true) ʊ;
}
```

Safety Proving and Bugs Finding for Function Calls

- two statuses
  1. come from postconditions
  2. produced by entailment procedure for precondition checking
- need operators to combine two statuses

ʊ status can be obtained from the combination

Key Idea 3: Operators for Error Calculus

Separation Logic Entailment:

$$\Delta_a \vdash \Delta_c$$

Δ_a and Δ_c are of conjunction form
However:

1. structural programs with conditional statements (e.g. if)
2. verification with case specifications
3. proof search with lemmas

produce more complex entailment:

$$\Delta_1 \vee \Delta_2 \vdash \Delta_3 \vee \Delta_4$$

need operators to combine error statuses

Key Idea 3: Operators for Error Calculus

SEARCH - $\oplus$

Search for success:

$$\frac{[\textbf{SE}\text{–}[\oplus\ \textbf{SEARCH}]]}{\begin{array}{c}\pi \vdash \pi_1 \rightsquigarrow \tau_1 \\ \pi \vdash \pi_2 \rightsquigarrow \tau_2\end{array}}{\pi \vdash \pi_1 \vee \pi_2 \rightsquigarrow \tau_1 \oplus \tau_2}$$

$\oplus$	$\top$	$\mho$	$\surd$	$\bot$
$\top$	$\top$	$\top$	$\surd$	$\bot$
$\mho$	$\top$	$\mho$	$\surd$	$\bot$
$\surd$	$\surd$	$\surd$	$\surd$	$\bot$
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$

Key idea 3: Operators for Error Calculus

SEARCH - $\oplus$

Search for success. Example:

$$x{=}\texttt{null} \vdash_{post} (x{=}\texttt{null}) \vee (\exists q \cdot \texttt{x}{\mapsto}\texttt{node}\langle \_, \texttt{q}\rangle * \texttt{llist}\langle \texttt{q}, \_\rangle)$$

- $x{=}\texttt{null} \vdash_{post} x{=}\texttt{null} \leadsto \surd$
- $x{=}\texttt{null} \vdash_{post} \exists q \cdot \texttt{x}{\mapsto}\texttt{node}\langle \_, \texttt{q}\rangle * \texttt{llist}\langle \texttt{q}, \_\rangle \leadsto \mho$

$$x{=}\texttt{null} \vdash_{post} (x{=}\texttt{null}) \vee (\exists q \cdot \texttt{x}{\mapsto}\texttt{node}\langle \_, \texttt{q}\rangle * \texttt{llist}\langle \texttt{q}, \_\rangle) \leadsto \surd \oplus \mho{=}\surd$$

Key Idea 3: Operators for Error Calculus

JOIN - $\sqcup$
Least upper bound

$$\frac{\pi_1 \vdash \pi \rightsquigarrow \tau_1 \quad \pi_2 \vdash \pi \rightsquigarrow \tau_2}{\pi_1 \vee \pi_2 \vdash \pi \rightsquigarrow \tau_1 \sqcup \tau_2}\ [\textbf{SE}\text{–}[\sqcup\ \textbf{JOIN}]]$$

$\sqcup$	$\top$	$\mho$	$\surd$	$\bot$
$\top$	$\top$	$\top$	$\top$	$\top$
$\mho$	$\top$	$\mho$	$\top$	$\mho$
$\surd$	$\top$	$\top$	$\surd$	$\surd$
$\bot$	$\top$	$\mho$	$\surd$	$\bot$

Key idea 3: Operators for Error Calculus

JOIN - $\sqcup$
Least upper bound

For example, join statuses for path traces:

$$(x{>}0 \wedge \texttt{res}{=}x) \vee (\neg(x{>}0) \wedge \texttt{res}{=}x) \vdash_{post} \texttt{res}{>}0$$

```
int foo (int x)
  requires true;
  ensures res>0;
{
   if (x>0){
     ...../not change x
   }
   else {
     ...../not change x
  }
  return x;
```

- $x{>}0 \wedge \texttt{res}{=}x \vdash_{post} \texttt{res}{>}0 \rightsquigarrow \surd$
- $\neg(x{>}0) \wedge \texttt{res}{=}x \vdash_{post} \texttt{res}{>}0 \rightsquigarrow \mho$

$$(x{>}0 \wedge \texttt{res}{=}x) \vee (\neg(x{>}0) \wedge \texttt{res}{=}x) \vdash_{post} \texttt{res}{>}0 \rightsquigarrow \surd \sqcup \mho = \top$$

Key idea 3: Operators for Error Calculus

COMPOSE - $\otimes$

Group relevant sub-formulas:

$$\frac{\pi \vdash \pi_1 \rightsquigarrow \tau_1 \quad \pi \vdash \pi_2 \rightsquigarrow \tau_2}{\pi \vdash \pi_1 \wedge \pi_2 \rightsquigarrow \tau_1 \otimes \tau_2}\ [\textbf{SE}\text{–}[\otimes\ \textbf{COMPOSE}]]$$

$\otimes$	$\top$	$\mho$	$\surd$	$\bot$
$\top$	$\top$	$\mho$	$\top$	$\bot$
$\mho$	$\mho$	$\mho$	$\mho$	$\bot$
$\surd$	$\top$	$\mho$	$\surd$	$\bot$
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$

Key idea 3: Operators for Error Calculus

COMPOSE - $\otimes$

Group relevant sub-formulas:
Example:

$$x{>}3 \wedge y{>}7 \ \vdash_{post} \ x{<}0 \wedge y{>}3$$

- $x{>}3 \wedge y{>}7 \ \vdash_{post} \ x{<}0 \rightsquigarrow \mho$
  - do slice: $x{>}3 \ \vdash_{post} \ x{<}0 \rightsquigarrow \mho$
- $x{>}3 \wedge y{>}7 \ \vdash_{post} \ y{>}3 \rightsquigarrow \surd$

$$x{>}3 \wedge y{>}7 \ \vdash_{post} \ x{<}0 \wedge y{>}3 \rightsquigarrow \mho \otimes \surd = \mho$$

Other Issues

- Soundness proof of structural entailment procedure
- Extend the calculus
  - with error messages
  - with error localization
  - for separation logic

Experiment

Calculus Performance for Heap-Based (bug-free) Programs:
overhead around 1.62 seconds (8%)

Programs (specified props)	Size		Proc	Time(sec.)		Invo.(#)	
	LOC	LOS	#	wo	w	wo	w
Linked list (size,interval)	327	50	26	0.44	0.46	2738	3202
Linked list (size,sets)	157	27	13	0.58	0.6	1520	1724
Sorted llist (size,sness,sets)	98	11	6	0.46	0.49	955	1060
Doubly llist (size,interval)	186	23	13	0.34	0.34	1864	2083
Doubly llist (size,sets)	91	13	5	0.5	0.5	1309	1429
CompleteT (size,minheight)	106	12	5	0.87	0.94	2149	2533
Heap trees (size,maxelem)	179	13	5	1.9	1.91	4540	4954
AVL (height, size)	313	27	12	3.44	3.59	7863	8585
AVL2 (height,size,bal)	152	37	7	2.83	3	6959	7876
BST (size,height)	177	18	9	0.35	0.37	1883	2192
BST (size,height,interval)	153	12	6	0.3	0.31	1581	1836
RBT (size,blackheight)	508	48	19	3.32	3.38	13069	16687
Bubble sort (size)	75	9	4	0.21	0.21	1092	1254
Quick sort (size, sets)	82	10	4	0.27	0.28	778	832
Merge sort (size,sets)	109	11	6	0.47	0.5	1035	1074
Quick sort - queue (size)	127	4	2	4.25	5.27	13218	21139
Total	2840	325	142	20.53	22.15	62553	78460

Experiment: Calculus Usability

1. taken from Siemens/SIR benchmark
2. $\top^*$ results from JOIN ($\sqcup$) of one safety and one must error: $\top^* = \mho \sqcup \surd$

Programs	LOC	Pr.#	Ver.#	F#	$\mho$	$\top^*$	$\top$	LOE	time(s)
tcas	173	9	41	48	31	14	3	3.48	3.06
schedule2	374	16	10	10	5	0	3	3	8.25
schedule1a	412	18	10	16	15	0	1	4.38	18.13
schedule1b	413	18	9	8	7	0	1	4.25	32.29
replace	564	21	24	24	18	0	6	4.21	17.89
print_tokens2	570	19	10	10	7	0	1	4.88	20.42
print_tokens	726	18	7	9	8	0	1	3.67	6.73
Total/(Average)	3232	119	111	125	91	14	16	(3.98)	(15.25)

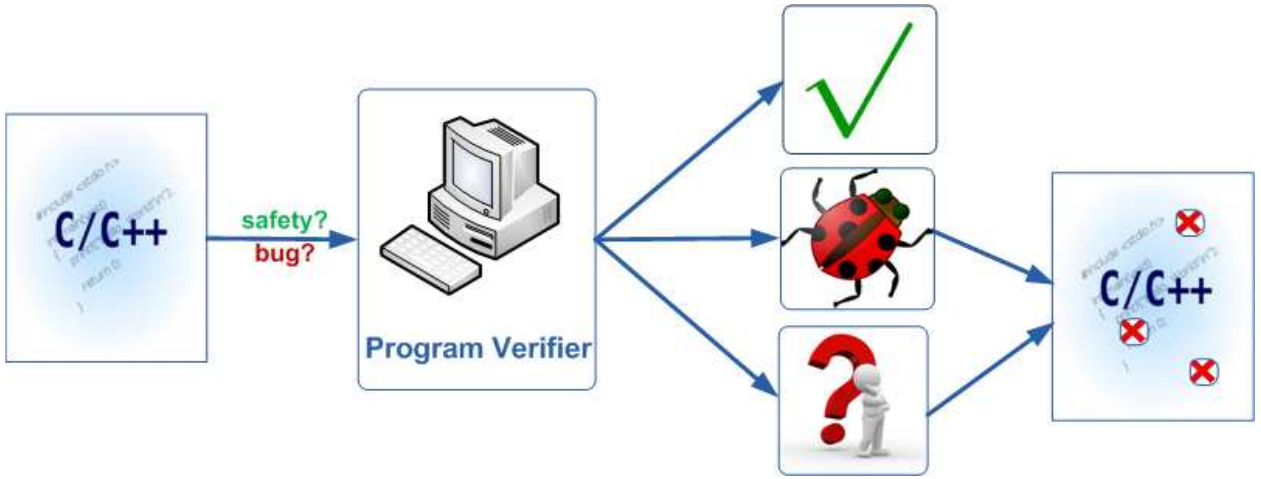
HIPEE detected 97% of real bugs: 73%, 11% and 13% of $\mho$, $\top^*$ and $\top$ errors, respectively.

Conclusions

We have presented

- a lattice domain with four distinct statuses on possible program states.
- a mechanism for complete specifications
- a calculus (for the 4-point lattice domain)

Conclusions



We have implemented an automatic *modular* verification system

1. for safety proving and bugs finding
2. and further assist users through errors localization.
3. on heap-manipulating programs through separation logic.

Further Study

Stronger specifications inference and Counterexample generation

- abduction (Calcagno et. al. POPL09, Aiken et. al. PLDI12).
- the weakest precondition (Snugglebug Chandra et. al. PLDI09) and negation on separation logic.
- CEGAR based (Berdine et. al. CAV12)

Questions?

Thank you!

Specs with Sound and Complete Requirements

```
1.  bool ischedule(int prio){
2.    if (prio>3) return true ;/*run it */
3.    else if (prio<0) abort();
4.    else{
5.      printf("Allow this task to run? y or n");
6.      char c=getc();
7.      if (c =′ y′) return true ;/*run it */
8.      if (c =′ n′) abort();
9.      printf("input must be y or n! Please try again");
10.     return ischedule(prio);
11.   }
12. }
```

```
requires prio>3
ensures (res=true);
```

$prio = 0$

```
case{
  prio>3 → ensures(res=true)√;
  prio<0 → ensures (true)Ʊ;
  0≤prio≤3 → ensures (true)⊤;
}
```