Algebraic Program Analysis

Martin Rinard
MIT

Input Parsing Approaches

Parser generator

Parser combinators

Filtered iterators

Hand coded (current most common practice)

Parser Generator Pros and Cons

Pros

- Correct by construction
- Clean identification of input language

Cons

- New tool in your tool chain
- New code you can manipulate only indirectly
- Input language changes may take language outside scope of the parser generator
- Only partially capture input language semantics
- Only get a parse tree

Parser Combinator Pros and Cons

Pros

- Clear direction about how to build parser
- Promotes correct coding styles
- Provides

Cons

- Must use stylized coding patterns
- Dependence on parser combinator library
- Directed towards producing parse tree

What is a filtered iterator?

for each input unit

parse and process input unit

- Input divided into units
- Iterate over input units

- Parsing and processing execute atomically
- Abort if error or assertion violation
- Result: code executes as if input unit never existed

Filtered Iterator Pros and Cons

Pros

- Very flexible coding style
- Can safely code actions close to corresponding part of input
- Protection against unanticipated faults
- Supports application-specific properties

Cons

- Dependence on atomic execution mechanism
- Potential dependences on discarded input units

Another Option: Program Analysis

Write program according to a specified style

Program analysis system targets/exploits style

Three phases

- Parse: read input into data structures
- Check: traverse data structures to verify valid input
- Execute: execute command specified by input

Key phase safety properties checked by program analyses

Verifying Parse Phase Produces Parse Tree

Identify code in parse phase (specify top level procedure)

Verify that code creates a parse tree

- All input data either 1) written into newly allocated data structures or 2) discarded
- Allocated data structures comprise a tree
- Code never dereferences uninitialized/null references

Allocated data structures big enough to hold input data

Potential error cases checked for and handled

More Parse Phase Analyses

No externally visible side effects

- No writes to externally visible data
- No invocations of effectful operations

Correspondence between

- Control flow graph
- Deterministic context-free grammar

No memory leaks (errors handled correctly)

Verifying Check Phase

Identify code in check phase (specify top level procedure)

Verify application specific data structure properties checked

No externally visible side effects

Values of data structure fields are within bounds

Required elements/correlations between fields are present

More application specific properties...

Result: know that the data structure holding the input satisfies (potentially quite sophisticated) consistency properties

Example Execute Analyses

No input operations performed

Verify how data flows from

- Data structures holding the input
- To the procedures that process pieces of input

Verify that appropriate checks have been applied to pieces of input before pieces are processed

Program Analysis Pros and Cons

Pros

- Flexible coding style
- No explicit dependence on any external component or system
- Can deploy analysis as necessary/desirable
- Can verify application-specific properties
- Can verify how input affects execute phase

Cons

- Need an effective program analysis system/implementation
- Constraints on coding style

Key Aspect

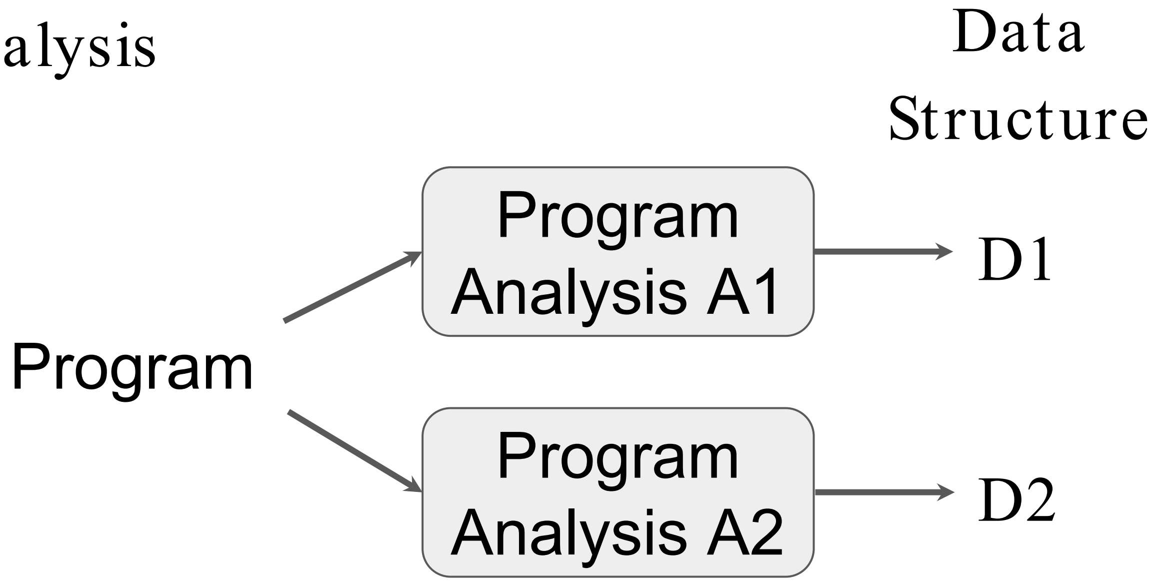
Multiple analyses deployed together

- Interprocedural control flow analyses
- Linear pointer analysis
- Value flow analyses
- Data structure consistency properties

The analyses we need to solve this (and other) problems are composite - they involve combining multiple analyses

Algebraic Program Analysis Framework

Current state of the art in program analysis



What is the relationship between D1 and D2?

Can be hard to tell...

How do I combine D1 and D2? Unclear...

Program traces as universal program analysis comparison mechanism

Program + D1 ⇒ Trace Predicate P1 ⇒ Set of Traces T1

Program + D2 ⇒ Trace Predicate P2 ⇒ Set of Traces T2

Key Questions:

$T1 \subseteq T2$? $T2 \subseteq T1$? $T1 \cap T2$? $T1 \cup T2$?

Answer: Traces induce lattice over program analysis

Order: $\supseteq$ Least upper bound: $\cap$ Top: $\emptyset$

Programs and Traces

Program P has a set of statement labels $l \in L$ and states $s \in S$

- l is label of next statement to execute, e.g. 5
- s is values of variables - e.g. [x=5, y=4, z=10]

An execution of P is a trace $t \in (L \times S)^*$

A trace t is therefore a sequence of <l,s> pairs

- l is the label of the next statement to execute
- s is the program state in which the statement at l executes
- Execution produces the next <l, s> in the trace

Traces and Program Analysis

traces(P) is the set of all valid traces of the program P

Program analyses A discover facts about possible executions

- Facts constrain possible executions
- Will capture meaning of these constraints as sets of traces

Each program analysis A defines a set of traces $A(P) \subseteq (L \times S)^*$

Will use traces A(P) as universal way to

- Characterize the information the analysis A extracts
- Compare analyses A1, A2
- Combine analyses A1, A2

Key concept

$$A'(P) \subset A(P)$$

A'(P) is more precise than A(P)

Another key concept

$$\text{traces}(P) \subseteq A'(P)$$

$$\text{traces}(P) \subseteq A(P)$$

So both A'(P) and A(P) are conservative

Many analyses are not conservative

Definitions based on traces for analysis A(P)

- If traces(P) $\subseteq$ A(P) then **A is conservative for P**
- If A is conservative for all programs P, it is conservative.
- If A1(P) $\subseteq$ A2(P) then **A1 is at least as precise as A2 for P**
- If A1(P) $\subset$ A2(P) then **A1 is more precise than A2 for P**

Definitions based on traces for analysis A(P)

- If **A1, A2** both conservative for P
  Then **A1 ∩ A2** is conservative for P

- If **A1, A2** both conservative for P
  Then **A1 ∪ A2** is conservative for P

- We now have a lattice of program analyses

Sets of Traces Concepts

Why are sound analyses (typically) conservative?

- Value abstraction imprecision
- Control-flow imprecision
- Intersect sets of traces to reduce imprecision

Sets of traces can be unsound

- Do not include some traces that program can exhibit
- (model checking, fuzz testing, concolic testing, …)

Value Analysis Example

Program P

$1: x = 6;$

$2: \text{if}\,(x < 10)$

$\quad 3: y = 7;$

else

$\quad 4: y = 8;$

$5: \text{halt}$

Value Analysis VA(P)

Data Structure (represents facts)

$1: []$

$2: [x \in \{6\}]$

$3: [x \in \{6\}]$

$4: [x \in \{6\}]$

$5: [x \in \{6\}, y \in \{7,8\}]$

Illustrative Example Traces from VA(P)

1: []

2: $[x \in \{6\}]$

3: $[x \in \{6\}]$

4: $[x \in \{6\}]$

5: $[x \in \{6\}, y \in \{7,8\}]$

```
1: x = 6;
2: if (x < 10)
        3: y = 7;
else
    4: y = 8;
5:halt
```

Actual program execution is in VA(P):

$$<1,[]><2,[x=6]><3,[x=6]><5,[x=6,y=7]>$$

Control flow imprecision in VA(P):

$$<1,[]><2,[x=6]><4,[x=6]><5,[x=6,y=8]>$$

Completely unrealizable paths in VA(P):

$$<5,[x=6,y=7]><3,[x=6]> \in VA(P)$$

Incomplete traces with extra information:

$$<3,[x=6, y=9]> \in VA(P)$$

Set of traces VA(P) defined by trace predicate over data structure

Program P	Data Structure	Trace predicate over traces t:
		$t \in VA(P)$ if
1: $x = 6$;	1: []	$\forall \langle l, s \rangle$ in t:
2: if $(x < 10)$	2: $[x \in \{6\}]$	if $(l = 2)$ $s(x)=6$
3: $y = 7$;	3: $[x \in \{6\}]$	if $(l = 3)$ $s(x)=6$
else	4: $[x \in \{6\}]$	if $(l = 4)$ $s(x)=6$
4: $y = 8$;	5: $[x \in \{6\}, y \in \{7,8\}]$	if $(l = 5)$ $s(x)=6 \wedge$
5: halt		$(s(y)=7 \vee$
		$s(y)=8)$

More Precise Value Analysis VA’(P)

Program P

$1: x = 6;$

$2: \text{if}\,(x < 10)$

$\quad\quad 3: y = 7;$

else

$\quad 4: y = 8;$

$5: \text{halt}$

Value Analysis VA’(P)

More Precise Data Structure

$1: []$

$2: [x \in \{6\}]$

$3: [x \in \{6\}]$

$4: [x \in \{6\}]$

$\cancel{5: [x \in \{6\}, y \in \{7, 8\}]}$

$5: [x \in \{6\}, y \in \{7\}]$

More precise trace predicate for VA'(P)

Data Structure

1: []

2: [x ∈ {6}]

3: [x ∈ {6}]

4: [x ∈ {6}]

5: [x ∈ {6}, y ∈ {7}]

More precise trace predicate over traces t:

t ∈ A1(P) if

∀ <l, s> in t:

if (l = 2) s(x)=6

if (l = 3) s(x)=6

if (l = 4) s(x)=6

if (l = 5) s(x)=6 ∧ s(y) = 7

~~(s(y)=7 ∨ s(y)=8)~~

Illustrative Example Traces from VA'(P)

1 : []
2 : [x ∈ {6}]
3 : [x ∈ {6}]
4 : [x ∈ {6}]
5 : [x ∈ {6}, y ∈ {7}]

```
1: x = 6;
2: if (x < 10)
        3: y = 7;
else
    4: y = 8;
5:halt
```

Actual program execution is in VA'(P):

<1,[]><2,[x=6]><3,[x=6]><5,[x=6,y=7]>

~~Control flow imprecision in VA'(P):~~

~~<1,[]><2,[x=6]><4,[x=6]><5,[x=6,y=8]>~~

Completely unrealizable paths in VA'(P):

<5,[x=6,y=7]><3,[x=6]> ∈ VA'(P)

Incomplete traces with extra information:

<3,[x=6, y=9]> ∈ VA'(P)

Summary from Value Analysis

Analyses produce a data structure

Data structure captures information that analysis produces

Use data structure to define trace predicate

Trace predicate identifies which traces are in the analysis

One analysis is more precise than another if

Traces of first analysis are subset of traces of second

If all traces of program are traces of analysis

Then analysis is conservative

Example Trace Predicates

Constant Analysis

“x is always 5 before statement S”

Reaching Definitions

“the value of x read in statement S always comes from one

of definitions in the set D”

Control Flow Resolution

“if statement S always takes the true branch”

Very Busy Expressions

“expression e is evaluated on all control flow paths before

Trace Predicates and Safety Properties

Safety property: “nothing bad happens”

- Value analysis safety property:
  - No execution with different value
- Reaching definitions safety property:
  - No use from a definition that does not reach use

Liveness property: “something good eventually happens”

- Program executes statement S infinitely often

Trace predicates express **only** safety properties

Sets of traces can express liveness properties

Different Kinds of Trace Predicates

Late predicates - true->false only at terminated traces

Every lock acquire followed by a lock release

Early predicates - true->false only at nonterminated traces

Variable x has value 5 before S executes

Mixed predicates - intersections of early/late predicates

Very busy expressions

Reaching Definitions

Concepts

Definition - assignment to a variable, e.g. $x = y + z$ defines x

A definition $x = \ldots$ reaches a label l if:

- $x = \ldots$ occurs before l in some execution and
- No assignment to x in between definition and l

Analysis may be conservative

- Analysis must find all reaching definitions
- May include additional reaching definitions

Reaching Definition Example (Forward)

Program P

$1: x = 6;$

$2: y = 2;$

$3: \text{if}\,(x < 10)$

$\quad 4: y = 3;$

$5: \text{halt}$

Reaching Definition RD(P)

Data Structure

$1: [x \mapsto \{\}, y \mapsto \{\}]$

$2: [x \mapsto \{1\}, y \mapsto \{\}]$

$3: [x \mapsto \{1\}, y \mapsto \{2\}]$

$4: [x \mapsto \{1\}, y \mapsto \{2\}]$

$5: [x \mapsto \{1\}, y \mapsto \{2, 4\}]$

To each label, associate set of labels, instead of set of variable values

Illustrative Example Traces from RD(P)

$1: [x \mapsto \{\}, y \mapsto \{\}]$

$2: [x \mapsto \{1\}, y \mapsto \{\}]$

$3: [x \mapsto \{1\}, y \mapsto \{2\}]$

$4: [x \mapsto \{1\}, y \mapsto \{2\}]$

$5: [x \mapsto \{1\}, y \mapsto \{2,4\}]$

```
1: x = 6;
2: y= 2;
3: if (x < 10)
        4: y = 3;
5:halt
```

Check that each active definition is in the result of the analysis.

Values don't matter for RD(P):

<1,[]><2,[x=*]><3,[x=*,y=*]><5,[x=*,y=*]>

Only labels and definitions matter for RD(P):

<1,*><2,*><4,*><5,*>

<2,*>

<5,*><2,*>

Set of traces RD(P) defined by trace predicate over data structure

Program P

1: x = 6;
2: y= 2;
3: if (x < 10)
    4: y = 3;
5: halt

Data Structure DS_{RD}

1: $[x \mapsto \{\}, y \mapsto \{\}]$
2: $[x \mapsto \{1\}, y \mapsto \{\}]$
3: $[x \mapsto \{1\}, y \mapsto \{2\}]$
4: $[x \mapsto \{1\}, y \mapsto \{2\}]$
5: $[x \mapsto \{1\}, y \mapsto \{2,4\}]$

Trace predicate over traces t:

$t=<l_1, s_1><l_2,s_2>...<l_n,s_n> \in RP(P)$ if:

$\forall\ v \in \{x,y\},\ 1 \le i \le j \le n,$
$write(l_i,v) \wedge (\forall k \in i<k<j,\ \neg\, write(l_k,v)) \Rightarrow l_i \in DS_{RD}(l_j)(v)$

For this program:
$write(l,v) = (v=x \wedge l=1) \vee (v=y \wedge ((l=2) \vee (l=4)))$

Set of traces RD(P) defined by trace predicate over data structure

Program P

1: x = 6;
2: y= 2;
3: if (x < 10)
    4: y = 3;
5:halt

Data Structure

1: $[x \mapsto \{\}, y \mapsto \{\}]$
2: $[x \mapsto \{1\}, y \mapsto \{\}]$
3: $[x \mapsto \{1\}, y \mapsto \{2\}]$
4: $[x \mapsto \{1\}, y \mapsto \{2\}]$
5: $[x \mapsto \{1\}, y \mapsto \{2,4\}]$

Trace predicate over traces t:

$t=<l_1, s_1><l_2,s_2>\ldots<l_n,s_n> \in RP(P)$ if:

$\forall\, v \in \{x,y\},\ 1 \le i \le j \le n,$

$write(l_i,v) \wedge \forall i<k<j,\ \neg write(l_k,v) \Rightarrow$

$l_j = 1 \Rightarrow True \wedge$

$l_j = 2 \Rightarrow [(v=x) \Rightarrow l_i \in \{1\}] \wedge$

$l_j = 3 \Rightarrow [(v=x) \Rightarrow l_i \in \{1\} \wedge (v=y) \Rightarrow l_i \in \{2\}] \wedge$

$l_j = 4 \Rightarrow [(v=x) \Rightarrow l_i \in \{1\} \wedge (v=y) \Rightarrow l_i \in \{2\}] \wedge$

$l_j = 5 \Rightarrow [(v=x) \Rightarrow l_i \in \{1\} \wedge (v=y) \Rightarrow l_i \in \{2,4\}]$

For this program:
$write(l,v) = (v=x \wedge l=1) \vee (v=y \wedge ((l=2) \vee (l=4)))$

More Precise RD'(P)

Program P

$1: x = 6;$

$2: y = 2;$

$3: \text{if}\,(x < 10)$

$\qquad 4: y = 3;$

$5: \text{halt}$

Reaching Definition RD'(P)

More Precise Data Structure

$1: [x \mapsto \{\}, y \mapsto \{\}]$

$2: [x \mapsto \{1\}, y \mapsto \{\}]$

$3: [x \mapsto \{1\}, y \mapsto \{2\}]$

$4: [x \mapsto \{1\}, y \mapsto \{2\}]$

$\cancel{5: [x \mapsto \{1\}, y \mapsto \{2, 4\}]}$

$5: [x \mapsto \{1\}, y \mapsto \{2\}]$

More precise trace predicate for RD'(P)

Data Structure

$1: [x \mapsto \{\}, y \mapsto \{\}]$

$2: [x \mapsto \{1\}, y \mapsto \{\}]$

$3: [x \mapsto \{1\}, y \mapsto \{2\}]$

$4: [x \mapsto \{1\}, y \mapsto \{2\}]$

$5: [x \mapsto \{1\}, y \mapsto \{2\}]$

More precise trace predicate over traces t:

$t = \langle l_1, s_1 \rangle \langle l_2, s_2 \rangle \ldots \langle l_n, s_n \rangle \in RP(P)$ if:

$\forall\, v \in \{x,y\},\ 1 \le i \le j \le n,$

$\quad write(l_i, v) \wedge \forall k \in i<k<j,\ \neg\, write(l_k, v) \Rightarrow$

$\quad l_j = 1 \Rightarrow True \wedge$

$\quad l_j = 2 \Rightarrow [(v=x) \Rightarrow l_i \in \{1\}] \wedge$

$\quad l_j = 3 \Rightarrow [(v=x) \Rightarrow l_i \in \{1\} \wedge (v=y) \Rightarrow l_i \in \{2\}] \wedge$

$\quad l_j = 4 \Rightarrow [(v=x) \Rightarrow l_i \in \{1\} \wedge (v=y) \Rightarrow l_i \in \{2\}] \wedge$

$\quad l_j = 5 \Rightarrow [(v=x) \Rightarrow l_i \in \{1\} \wedge (v=y) \Rightarrow l_i \in \{2, \cancel{4}\}]$

Illustrative Example Traces from RD’(P)

$1: [x \mapsto \{\}, y \mapsto \{\}]$

$2: [x \mapsto \{1\}, y \mapsto \{\}]$

$3: [x \mapsto \{1\}, y \mapsto \{2\}]$

$4: [x \mapsto \{1\}, y \mapsto \{2\}]$

$5: [x \mapsto \{1\}, y \mapsto \{2\}]$

$1: x = 6;$

$2: y = 2;$

$3: \text{if } (x < 10)$

$\quad 4: y = 3;$

$5: \text{halt}$

Check that each active definition is in the result of the analysis.

Values don’t matter in RD’(P):

<1,[]><2,[x=*]><3,[x=*,y=*]><5,[x=*,y=*]>

Actually, only labels matter for RD’(P):

~~<1,*><2,*><4,*><5,*>~~

<2,*>

<5,*><2,*>

Very Busy Expressions

An expression x+y is very busy at a program point p if

x+y is evaluated along all control flow paths from p
before x or y written

Purpose of very busy expressions - evaluate expressions early

Don't introduce new exceptions
Don't introduce new work

Very Busy Expressions Trace Predicate

Intersection of an early and late trace predicate

(early) whenever x or y written, x+y has been evaluated

(late) when program halts, x+y has been evaluated

Note: if program does not halt, x+y need not be evaluated…

Ensuring Very Busy Expressions are Always Evaluated

Perform a termination analysis

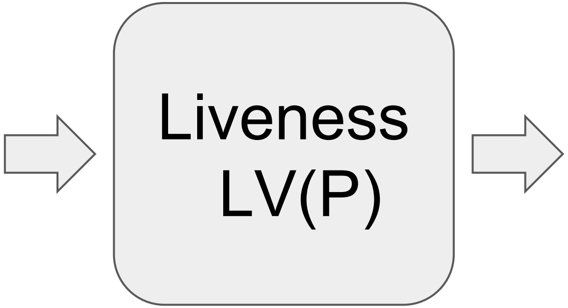
Intersect resulting termination trace predicate with very busy expressions trace predicate

Example of building up more sophisticated analyses from smaller analyses

Liveness Example (Backward)

Program P

1 : x = 6;
2 : y= x +1 ;
3 : if (x<3)
4 : y = x+y;
5 :return(y)



Data Structure

1: {}
2: {x }
3: {x, y }
4: {x, y }
5: {y }

For each label, associate set of variables

Illustrative Example Traces from LV(P)

1: {}

2: {x }

3: {x, y }

4: {x, y }

5: {y }

Values don’t matter in LV(P):

<1,[]><2,[x=*]><3,[x=*,y=*]><5,[x=*,y=*]>

```
1: x = 6;
2: y= x +1;
3: if (x<3)
        4: y = x+y;
5:return(y)
```

Only labels matter for LV(P):

<1,*><2,*><4,*><5,*>

<2,*>

<5,*><2,*>

Set of traces LV(P) defined by trace predicate over data structure

Program P	Data Structure DS_{LV}
1: x = 6;	1: {}
2: y= x +1;	2: {x }
3: if (x<3)	3: {x, y }
4: y = x+y;	4: {x, y }
5:return(y)	5: {x, y }

Trace predicate over traces t:

$t=<l_1, s_1><l_2,s_2>...<l_n,s_n>\in LV(P)$ if:

$\forall\ v\in\{x,y\},\ 1\leq i\leq j\leq n,$

$write(l_i,v) \wedge read(l_j,v) \wedge$

$(\forall k \in i<k<j,\ \neg write(l_k,v)) \Rightarrow$

$\forall m \in i\leq m\leq j,\ v\in DS_{LV}(l_m)$

For this program:

$write(l,v) = (v=x \wedge l=1) \vee (v=y \wedge ((l=2) \vee (l=4)))$

$read(l,v) = (v=x \wedge ((l=2) \vee (l=4))) \vee (v=y \wedge ((l=4) \vee (l=5)))$

More Precise LV'(P)

Program P

1 : x = 6 ;

2 : y= x +1 ;

3 : if (x<3)

4 : y = x+y;

5 :return(y)

Liveness LV'(P)

More Precise Data Structure

1: {}

2: {x }

3: {}

4: {}

5: {y }

Illustrative Example Traces from LV’(P)

1: {}
2: {x }
3: {}
4: {}
5: {y }

```
1: x = 6;
2: y= x +1;
3: if (x<3)
            4: y = x+y;
5:return(y)
```

Values don’t matter in LV’(P):

~~<1,[]><2,[x=*]><3,[x=*,y=*]><5,[x=*,y=*]>~~

Actually, only labels matter for LV’(P):

~~<1,*><2,*><4,*><5,*>~~

<2,*>

<5,*><2,*>

Analyses that underapproximate traces

Fuzzing

- Defines set of traces (fuzzed executions)
- Set of traces grows monotonically as fuzz
- Conservative in the limit

Bounded model checking

- Reasons about sets of traces (finitely bounded)
- via SAT/SMT solvers

Unified framework points the way toward combining analyses

Space of Analyses

- Analyses form a complete lattice
  - Order is reverse subset inclusion ($\supseteq$)
  - Least upper bound ($\vee$) is $\cap$ (correspond to ANDing predicates)
  - Greatest lower bound ($\wedge$) is $\cup$ (correspond to ORing predicates)
  - Analyses become more precise as move up the lattice
  - Top ($\top$) is $\emptyset$, bottom ($\bot$) is all traces
- Analyses of note
  - The trivial analysis: all traces
  - The empty analysis: no traces
  - Interpretation analysis: all valid traces of a program

Where we are so far

Standard dataflow analyses

Produce data structures that store analysis results

Data structures induce trace predicates that define sets of traces

Analyses ordered by lattice over sets of traces

Examples:

- Value analysis hierarchy
- Reaching definition hierarchy
- Live variables hierarchy

The Big Picture

Traces of an analysis allow us to:

Characterize conservative analyses

Compare analyses even when they analyse different things

Talk about analysis that are not conservative

Treat unsound and sound analyses in same framework

Understand effects of combining analyses

<u>One conceptual framework, all analyses live in the same space</u>

Acknowledgements

Collaborators: Henny Sipma, Thomas Borgeaut

Funding: DARPA AMP

Inspiration: Conversations with Sergey Bratus