Sendmail crackaddr - Static Analysis strikes back

Bogdan Mihaila

Technical University of Munich, Germany
December 6, 2014

Name Lastname < *name@mail.org* > ()()()()()()()()()() . . . ()()()

Abstract Interpretation Primer

Static program analysis using abstract interpretation

- use abstract domains to over-approximate concrete states
- abstract transformers simulate the concrete program semantics on the abstract state
- perform a fixpoint computation to infer invariants for each program point
- merge over all paths over-approximates all possible program executions (soundness)
- precision depends on the abstraction (completeness)
- for termination widening is necessary (introduces imprecision)

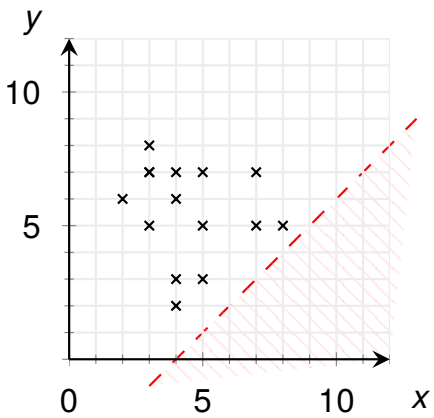
Abstraction Examples

Some examples of concrete values and their abstractions ...

Sets of Concrete Values and their Abstractions

Concrete Points

$\pm x = c$

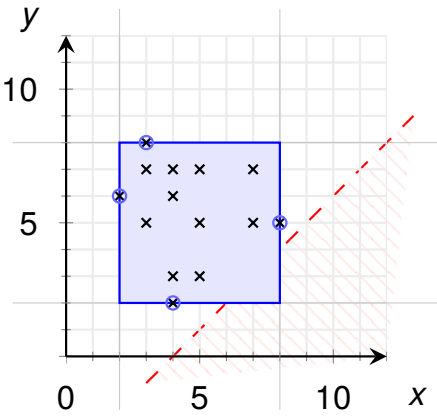


Constraints:

$$\begin{array}{l} x = 2 \wedge y = 6 \\ \vee\, x = 3 \wedge y = 5 \\ \vee\, x = 3 \wedge y = 7 \\ \vee\, x = 3 \wedge y = 8 \\ \vee\ \ldots \end{array}$$

Intervals

$\pm x \leq c$

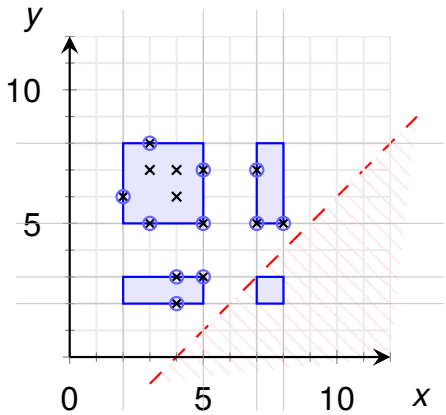


Constraints:

$$\begin{array}{l} 2 \leq x \wedge x \leq 8 \\ \wedge\, 2 \leq y \wedge y \leq 8 \end{array}$$

Interval Sets

$\bigvee_i (l_i \leq x \wedge x \leq u_i)$

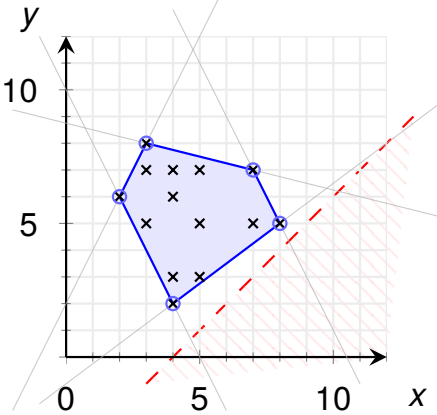


Constraints:

$$\begin{array}{l} 2 \leq x \wedge x \leq 5 \\ \vee\, 7 \leq x \wedge x \leq 8 \\ \vee\, 2 \leq y \wedge y \leq 3 \\ \vee\, 5 \leq y \wedge y \leq 8 \end{array}$$

Polyhedra

$\sum_i a_i x_i \leq c$



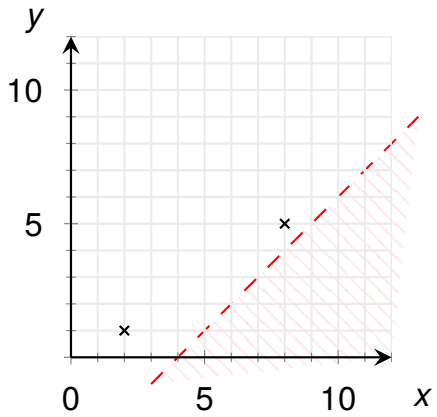
Constraints:

$$\begin{array}{l} 2x - y \leq -2 \\ \wedge\ -2x - y \leq -10 \\ \wedge\, 2x + y \leq -21 \\ \wedge\, 3x + 4y \leq 4 \\ \wedge\, x + 4y \leq 35 \end{array}$$

Sets of Concrete Values and their Abstractions

Concrete Points

$\pm x = c$

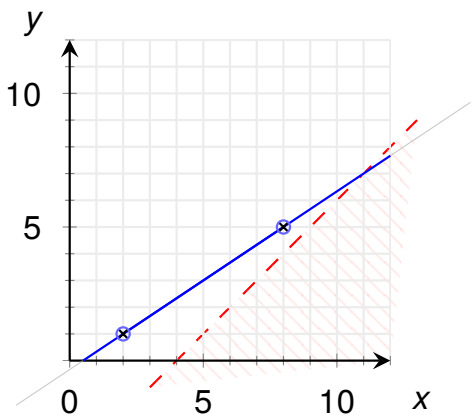


Constraints:

$x = 2 \wedge y = 1$
$\vee\, x = 8 \wedge y = 5$

Affine Equalities

$\sum_i a_i x_i = c$

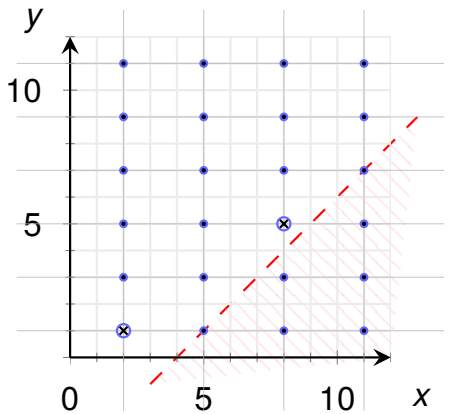


Constraints:

$2x - 3y = 3$

Congruences

$x \equiv b \pmod{a}$



Constraints:

$x \equiv 2 \pmod 3$
$\wedge\, y \equiv 1 \pmod 2$

Operations on Abstractions

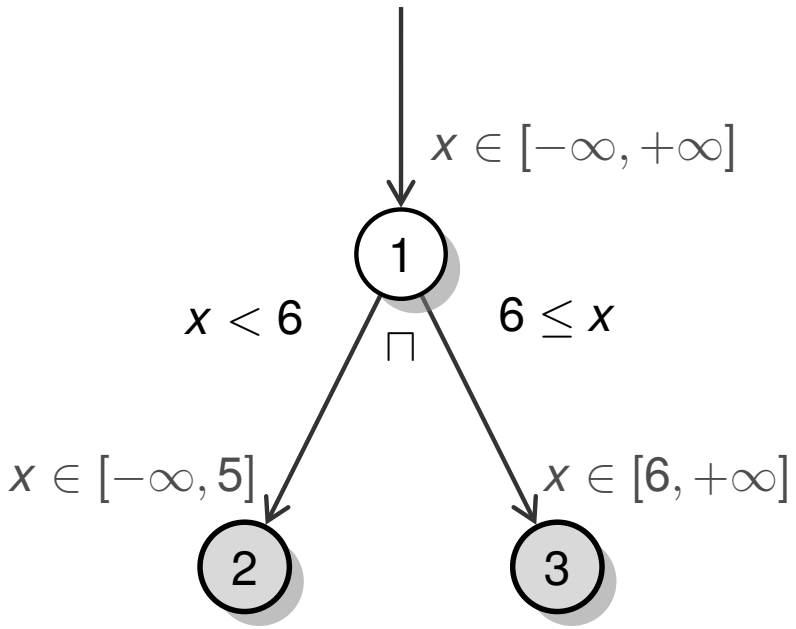
Some examples of operations on abstractions ...

Some Operations on Intervals

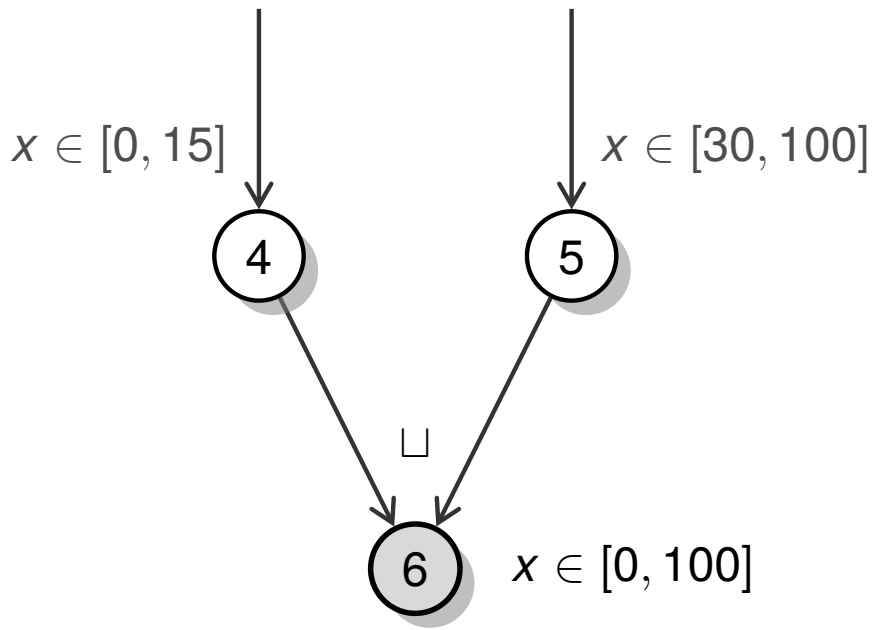
Arithmetics:

$$[0, 100] + [1, 2] = [1, 102]$$
$$[0, 100] - [1, 2] = [-2, 99]$$

Tests or Assumptions, Meet $\sqcap$



Merge of paths, Join $\sqcup$



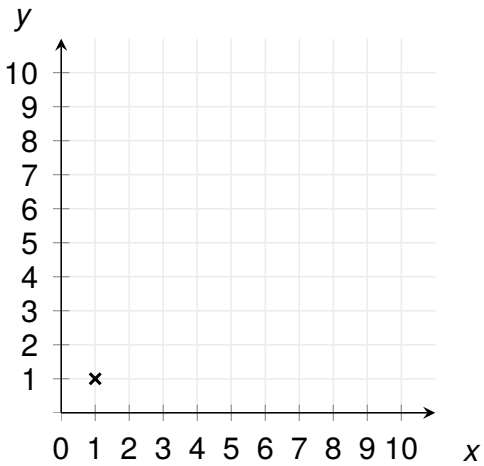
Operations on Abstractions

Widening and Narrowing

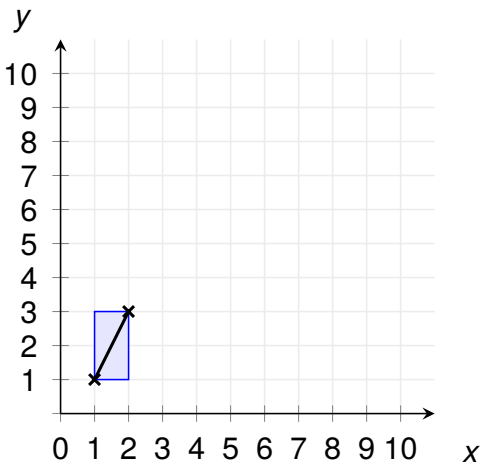
To analyze loops in less steps than the real iterations count ...
and especially always analyze loops in finite steps.
Analysis Termination!

Widening and Narrowing on Intervals

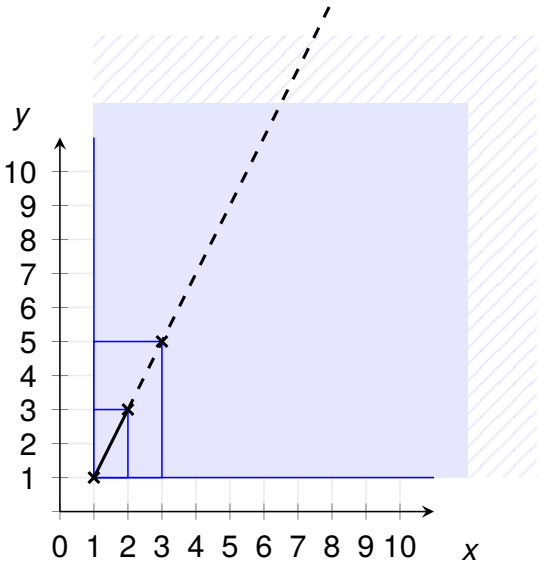
```
int x = 1;
int y = 1;
// shown x, y values
// are at loop head
while (x <= 6) {
  x = x + 1;
  y = y + 2;
}
```



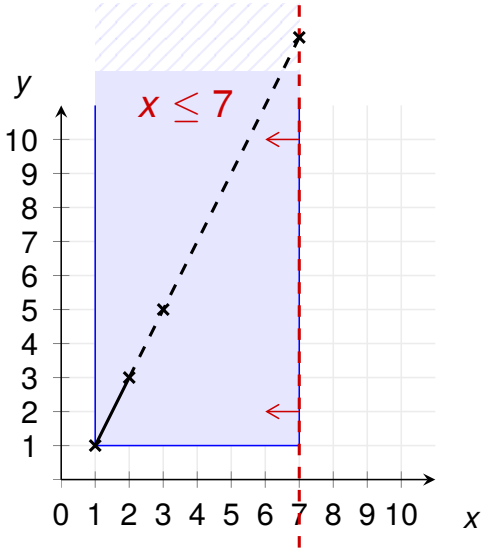
1^{st} Iteration



2^{nd} Iteration: $\sqcup$ join



3^{rd} Iteration: ∇ widening



4^{th} Iteration: Δ narrowing

Abstract Interpretation

Good intro and overview material:

- P. Cousot and R. Cousot, **A gentle introduction to formal verification of computer systems by abstract interpretation**, 2010
- P. Cousot, **Abstract Interpretation Based Formal Methods and Future Challenges**, 2001
- P. Cousot and R. Cousot, **Abstract Interpretation: Past, Present and Future**, 2014

Static Binary Analyzer

Now to the Analyzer ...

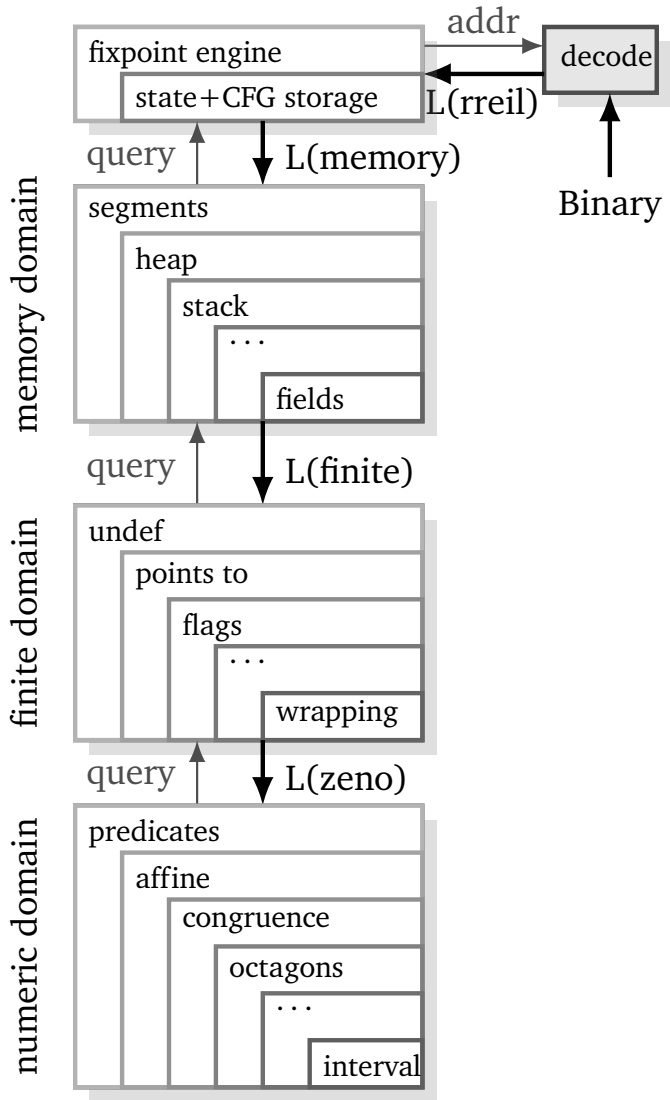
Analyzer Features

Analysis of binaries using abstract interpretation

- analyze machine code from disassembled executables
- translate machine code to intermediate language (RREIL)
- abstract transformers for instruction semantics of RREIL
- perform a reachability analysis to infer jump targets and
- use abstract domains to infer memory bounds and flag out-of-bounds accesses

Project page: **https://bitbucket.org/mihaila/bindead**

Analyzer Overview



- disassembler frontend produces RREIL for the analysis
- RREIL gets transformed to simpler languages for the abstract domains
- fixpoint and disassembly process are intertwined
- modular construction using co-fibered abstract domains
- domains stack is a partially reduced product of domains
- for interprocedural analysis we use either call-string or a summarization approach

Sendmail Bug

And finally to Sendmail ...

Sendmail Bug

Discovered 2003 by Mark Dowd

Buffer overflow in an email address parsing function of Sendmail. Consists of a parsing loop using a state machine. ~500 LOC.

Sendmail Bug

Discovered 2003 by Mark Dowd

Buffer overflow in an email address parsing function of Sendmail. Consists of a parsing loop using a state machine. ~500 LOC.

Bounty for Static Analyzers since 2011

Thomas Dullien extracted a smaller version of the bug as an example of a hard problem for static analyzers. ~50 LOC.

Sendmail Bug

Discovered 2003 by Mark Dowd

Buffer overflow in an email address parsing function of Sendmail. Consists of a parsing loop using a state machine. ∼500 LOC.

Bounty for Static Analyzers since 2011

Thomas Dullien extracted a smaller version of the bug as an example of a hard problem for static analyzers. ∼50 LOC.

Since then . . .

Various talks at security conferences about the static analysis of the problem and a paper using Havoc/Boogie. Unfortunately, their solution required manual specification of the loop invariant.

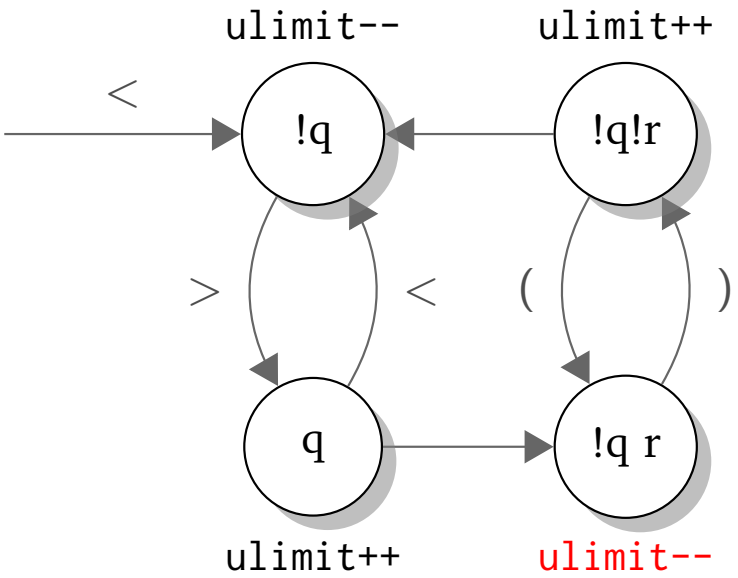
Sendmail Bug Code

```
1  #define BUFFERSIZE 200
2  #define TRUE 1
3  #define FALSE 0
4  int copy_it (char *input, unsigned int length) {
5      char c, localbuf[BUFFERSIZE];
6      unsigned int upperlimit = BUFFERSIZE - 10;
7      unsigned int quotation = roundquote = FALSE;
8      unsigned int inputIndex = outputIndex = 0;
9      while (inputIndex < length) {
10         c = input[inputIndex++];
11         if ((c == '<') && (!quotation)) {
12             quotation = TRUE; upperlimit--;
13         }
14         if ((c == '>') && (quotation)) {
15             quotation = FALSE; upperlimit++;
16         }
17         if ((c == '(') && (!quotation) && !roundquote) {
18             roundquote = TRUE; upperlimit--; // decrementation was missing in bug
19         }
20         if ((c == ')') && (!quotation) && roundquote) {
21             roundquote = FALSE; upperlimit++;
22         }
23         // If there is sufficient space in the buffer, write the character.
24         if (outputIndex < upperlimit) {
25             localbuf[outputIndex] = c;
26             outputIndex++;
27         }
28     }
29     if (roundquote) {
30         localbuf[outputIndex] = ')'; outputIndex++; }
31     if (quotation) {
32         localbuf[outputIndex] = '>'; outputIndex++; }
33 }
```

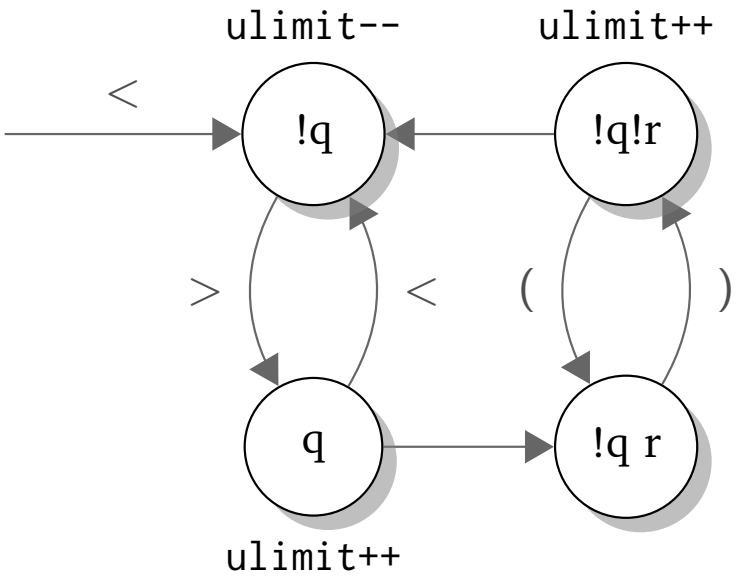
State Machine of Parser

We need to verify that `outputIndex < upperlimit < BUFFERSIZE` always holds in the good version.

Good:



Bad:



In the bad version `upperlimit` can be steadily incremented and a write outside of the stack allocated buffer can be triggered.

Sendmail Bug Analysis

Why hard?

Symbolic Execution

Runs into path explosion due to the loop and automaton.

Sendmail Bug Analysis

Why hard?

Symbolic Execution

Runs into path explosion due to the loop and automaton.

Model Checking

Not able to infer loop invariant. Requires manual specification of the automaton invariant.

Sendmail Bug Analysis

Why hard?

Symbolic Execution

Runs into path explosion due to the loop and automaton.

Model Checking

Not able to infer loop invariant. Requires manual specification of the automaton invariant.

Abstract Interpretation

The over-approximation, the join and widening of states introduce too much imprecision. Not able to separate the states of the parsing automaton inside the loop.

Sendmail Bug Analysis

Why hard?

Symbolic Execution

Runs into path explosion due to the loop and automaton.

Model Checking

Not able to infer loop invariant. Requires manual specification of the automaton invariant.

Abstract Interpretation

The over-approximation, the join and widening of states introduce too much imprecision. Not able to separate the states of the parsing automaton inside the loop.

Sendmail Bug Analysis

Why hard?

Symbolic Execution

Runs into path explosion due to the loop and automaton.

Model Checking

Not able to infer loop invariant. Requires manual specification of the automaton invariant.

Abstract Interpretation

The over-approximation, the join and widening of states introduce too much imprecision. Not able to separate the states of the parsing automaton inside the loop.

$\rightsquigarrow$ the non-vulnerable version is flagged as vulnerable, too, by a static analyzer

Sendmail Code Revisited (Problems and Ideas)

```
#define BUFFERSIZE 200
#define TRUE 1
#define FALSE 0     prove memory correctnes for all possible concrete inputs!
int copy_it (char *input, unsigned int length) { *input[i] ∈ [−∞, +∞], length ∈ [0, +∞]
    char c, localbuf[BUFFERSIZE];
    unsigned int upperlimit = BUFFERSIZE - 10;
    unsigned int quotation = roundquote = FALSE;
    unsigned int inputIndex = outputIndex = 0;
    while (inputIndex < length) {
        c = input[inputIndex++];
        if ((c == '<') && (!quotation)) {
            quotation = TRUE; upperlimit--;
        }
        if ((c == '>') && (quotation)) {
            quotation = FALSE; upperlimit++;
        }
        if ((c == '(') && (!quotation) && !roundquote) {
            roundquote = TRUE; upperlimit--; // decrementation was missing in bug
        }
        if ((c == ')') && (!quotation) && roundquote) {
            roundquote = FALSE; upperlimit++;
        }
        // If there is sufficient space in the buffer, write the character.
        if (outputIndex < upperlimit) {
            localbuf[outputIndex] = c; prove that outputIndex < BUFFERSIZE holds
            outputIndex++;
        }
    }
    if (roundquote) { prove that invariant outputIndex < BUFFERSIZE holds
        localbuf[outputIndex] = ')'; outputIndex++; }
    if (quotation) { prove that invariant outputIndex < BUFFERSIZE holds
        localbuf[outputIndex] = '>'; outputIndex++; }
}
```

Sendmail Code Revisited (Problems and Ideas)

```
#define BUFFERSIZE 200
#define TRUE 1
#define FALSE 0
int copy_it (char *input, unsigned int length) {
    char c, localbuf[BUFFERSIZE];
    unsigned int upperlimit = BUFFERSIZE - 10;
    unsigned int quotation = roundquote = FALSE;
    unsigned int inputIndex = outputIndex = 0;   inputIndex ∈ [0, 0], outputIndex ∈ [0, 0]
    while (inputIndex < length) {
        c = input[inputIndex++];    inputIndex ∈ [1, 1]
        if ((c == '<') && (!quotation)) {
            quotation = TRUE; upperlimit--;
        }
        if ((c == '>') && (quotation)) {
            quotation = FALSE; upperlimit++;
        }
        if ((c == '(') && (!quotation) && !roundquote) {
            roundquote = TRUE; upperlimit--; // decrementation was missing in bug
        }
        if ((c == ')') && (!quotation) && roundquote) {
            roundquote = FALSE; upperlimit++;
        }
        // If there is sufficient space in the buffer, write the character.
        if (outputIndex < upperlimit) {
            localbuf[outputIndex] = c; prove that outputIndex < BUFFERSIZE holds
            outputIndex++;   outputIndex ∈ [1, 1]
        }    ⊔ : outputIndex ∈ [0, 1]
    }
    if (roundquote) { prove that invariant outputIndex < BUFFERSIZE holds
        localbuf[outputIndex] = ')'; outputIndex++; }
    if (quotation) { prove that invariant outputIndex < BUFFERSIZE holds
        localbuf[outputIndex] = '>'; outputIndex++; }
}
```

Sendmail Code Revisited (Problems and Ideas)

```
#define BUFFERSIZE 200
#define TRUE 1
#define FALSE 0
int copy_it (char *input, unsigned int length) {
    char c, localbuf[BUFFERSIZE];
    unsigned int upperlimit = BUFFERSIZE - 10;
    unsigned int quotation = roundquote = FALSE;
    unsigned int inputIndex = outputIndex = 0;   inputIndex ∈ [0, 0], outputIndex ∈ [0, 0]
    while (inputIndex < length) { widening ∇: inputIndex ∈ [0, +∞], outputIndex ∈ [0, +∞]
        c = input[inputIndex++];    inputIndex ∈ [1, 1]
        if ((c == '<') && (!quotation)) {
            quotation = TRUE; upperlimit--;
        }
        if ((c == '>') && (quotation)) {
            quotation = FALSE; upperlimit++;
        }
        if ((c == '(') && (!quotation) && !roundquote) {
            roundquote = TRUE; upperlimit--; // decrementation was missing in bug
        }
        if ((c == ')') && (!quotation) && roundquote) {
            roundquote = FALSE; upperlimit++;
        }
        // If there is sufficient space in the buffer, write the character.
        if (outputIndex < upperlimit) {
            localbuf[outputIndex] = c; prove that outputIndex < BUFFERSIZE holds
            outputIndex++;   outputIndex ∈ [1, 1]
        }    ⊔ : outputIndex ∈ [0, 1]
    }
    if (roundquote) { prove that invariant outputIndex < BUFFERSIZE holds
        localbuf[outputIndex] = ')'; outputIndex++; }
    if (quotation) { prove that invariant outputIndex < BUFFERSIZE holds
        localbuf[outputIndex] = '>'; outputIndex++; }
}
```

Sendmail Code Revisited (Problems and Ideas)

```
#define BUFFERSIZE 200
#define TRUE 1
#define FALSE 0
int copy_it (char *input, unsigned int length) {
    char c, localbuf[BUFFERSIZE];
    unsigned int upperlimit = BUFFERSIZE - 10; upperlimit ∈ [190, 190]
    unsigned int quotation = roundquote = FALSE;
    unsigned int inputIndex = outputIndex = 0;  inputIndex ∈ [0, 0], outputIndex ∈ [0, 0]
    while (inputIndex < length) { widening ∇: inputIndex ∈ [0, +∞], outputIndex ∈ [0, +∞]
        c = input[inputIndex++];   inputIndex ∈ [1, 1]
        if ((c == '<') && (!quotation)) {
            quotation = TRUE; upperlimit--;
        }
        if ((c == '>') && (quotation)) {
            quotation = FALSE; upperlimit++;
        }
        if ((c == '(') && (!quotation) && !roundquote) {
            roundquote = TRUE; upperlimit--; // decrementation was missing in bug
        }
        if ((c == ')') && (!quotation) && roundquote) {
            roundquote = FALSE; upperlimit++;
        }
        // If there is sufficient space in the buffer, write the character.
        if (outputIndex < upperlimit) { use threshold outputIndex < upperlimit for widening!
            localbuf[outputIndex] = c; prove that outputIndex < BUFFERSIZE holds
            outputIndex++;  outputIndex ∈ [1, 1]
        }   ⊔ : outputIndex ∈ [0, 1]
    }
    if (roundquote) { prove that invariant outputIndex < BUFFERSIZE holds
        localbuf[outputIndex] = ')'; outputIndex++; }
    if (quotation) { prove that invariant outputIndex < BUFFERSIZE holds
        localbuf[outputIndex] = '>'; outputIndex++; }
}
```

Sendmail Code Revisited (Problems and Ideas)

```
#define BUFFERSIZE 200
#define TRUE 1
#define FALSE 0
int copy_it (char *input, unsigned int length) {
    char c, localbuf[BUFFERSIZE];
    unsigned int upperlimit = BUFFERSIZE - 10; upperlimit ∈ [190, 190]
    unsigned int quotation = roundquote = FALSE;
    unsigned int inputIndex = outputIndex = 0;
    while (inputIndex < length) {
        c = input[inputIndex++];
        if ((c == '<') && (!quotation)) {
            quotation = TRUE; upperlimit--;
        } ⊔ : upperlimit ∈ [189, 190]
        if ((c == '>') && (quotation)) {
            quotation = FALSE; upperlimit++;
        } ⊔ : upperlimit ∈ [189, 191]
        if ((c == '(') && (!quotation) && !roundquote) {
            roundquote = TRUE; upperlimit--; // decrementation was missing in bug
        } ⊔ : upperlimit ∈ [188, 191]
        if ((c == ')') && (!quotation) && roundquote) {
            roundquote = FALSE; upperlimit++;
        } ⊔ : upperlimit ∈ [188, 192]
        // If there is sufficient space in the buffer, write the character.
        if (outputIndex < upperlimit) {
            localbuf[outputIndex] = c;
            outputIndex++;
        }
    }
    if (roundquote) {
        localbuf[outputIndex] = ')'; outputIndex++; }
    if (quotation) {
        localbuf[outputIndex] = '>'; outputIndex++; }
}
```

Sendmail Code Revisited (Problems and Ideas)

```
#define BUFFERSIZE 200
#define TRUE 1
#define FALSE 0
int copy_it (char *input, unsigned int length) {
    char c, localbuf[BUFFERSIZE];
    unsigned int upperlimit = BUFFERSIZE - 10; upperlimit ∈ [190, 190]
    unsigned int quotation = roundquote = FALSE;
    unsigned int inputIndex = outputIndex = 0;
    while (inputIndex < length) { widening ∇ removes bounds: upperlimit ∈ [−∞, +∞]
        c = input[inputIndex++];
        if ((c == '<') && (!quotation)) {
            quotation = TRUE; upperlimit--;
        } ⊔ : upperlimit ∈ [189, 190]
        if ((c == '>') && (quotation)) {
            quotation = FALSE; upperlimit++;
        } ⊔ : upperlimit ∈ [189, 191]
        if ((c == '(') && (!quotation) && !roundquote) {
            roundquote = TRUE; upperlimit--; // decrementation was missing in bug
        } ⊔ : upperlimit ∈ [188, 191]
        if ((c == ')') && (!quotation) && roundquote) {
            roundquote = FALSE; upperlimit++;
        } ⊔ : upperlimit ∈ [188, 192]
        // If there is sufficient space in the buffer, write the character.
        if (outputIndex < upperlimit) {
            localbuf[outputIndex] = c;
            outputIndex++;
        }
    }
    if (roundquote) {
        localbuf[outputIndex] = ')'; outputIndex++; }
    if (quotation) {
        localbuf[outputIndex] = '>'; outputIndex++; }
}
```

Sendmail Code Revisited (Problems and Ideas)

```
#define BUFFERSIZE 200
#define TRUE 1
#define FALSE 0
int copy_it (char *input, unsigned int length) {
    char c, localbuf[BUFFERSIZE];
    unsigned int upperlimit = BUFFERSIZE - 10; upperlimit ∈ [190, 190]
    unsigned int quotation = roundquote = FALSE;
    unsigned int inputIndex = outputIndex = 0;
    while (inputIndex < length) { widening ∇ removes bounds: upperlimit ∈ [−∞, +∞]
        c = input[inputIndex++];   use relation with flag variables quotation and roundquote
        if ((c == '<') && (!quotation)) { to keep upperlimit bounded!
            quotation = TRUE; upperlimit--;
        } ⊔ : upperlimit ∈ [189, 190]
        if ((c == '>') && (quotation)) {
            quotation = FALSE; upperlimit++;
        } ⊔ : upperlimit ∈ [189, 191]
        if ((c == '(') && (!quotation) && !roundquote) {
            roundquote = TRUE; upperlimit--; // decrementation was missing in bug
        } ⊔ : upperlimit ∈ [188, 191]
        if ((c == ')') && (!quotation) && roundquote) {
            roundquote = FALSE; upperlimit++;
        } ⊔ : upperlimit ∈ [188, 192]
        // If there is sufficient space in the buffer, write the character.
        if (outputIndex < upperlimit) {
            localbuf[outputIndex] = c;
            outputIndex++;
        }
    }
    if (roundquote) {
        localbuf[outputIndex] = ')'; outputIndex++; }
    if (quotation) {
        localbuf[outputIndex] = '>'; outputIndex++; }
}
```

Sendmail Code Revisited (Problems and Ideas)

```
#define BUFFERSIZE 200
#define TRUE 1
#define FALSE 0
int copy_it (char *input, unsigned int length) {
    char c, localbuf[BUFFERSIZE];
    unsigned int upperlimit = BUFFERSIZE - 10;
    unsigned int quotation = roundquote = FALSE; quotation ∈ [0, 0], roundquote ∈ [0, 0]
    unsigned int inputIndex = outputIndex = 0;
    while (inputIndex < length) {
        c = input[inputIndex++];
        if ((c == '<') && (!quotation)) {
            quotation = TRUE; upperlimit--;
        } ⊔ : quotation ∈ [0, 1]
        if ((c == '>') && (quotation)) {
            quotation = FALSE; upperlimit++;
        } ⊔ : quotation ∈ [0, 1]
        if ((c == '(') && (!quotation) && !roundquote) {
            roundquote = TRUE; upperlimit--; // decrementation was missing in bug
        } ⊔ : quotation ∈ [0, 1], roundquote ∈ [0, 1]
        if ((c == ')') && (!quotation) && roundquote) {
            roundquote = FALSE; upperlimit++;
        } ⊔ : quotation ∈ [0, 1], roundquote ∈ [0, 1]
        // If there is sufficient space in the buffer, write the character.
        if (outputIndex < upperlimit) {
            localbuf[outputIndex] = c;
            outputIndex++;
        }
    }
    if (roundquote) {
        localbuf[outputIndex] = ')'; outputIndex++; }
    if (quotation) {
        localbuf[outputIndex] = '>'; outputIndex++; }
}
```

Sendmail Code Revisited (Problems and Ideas)

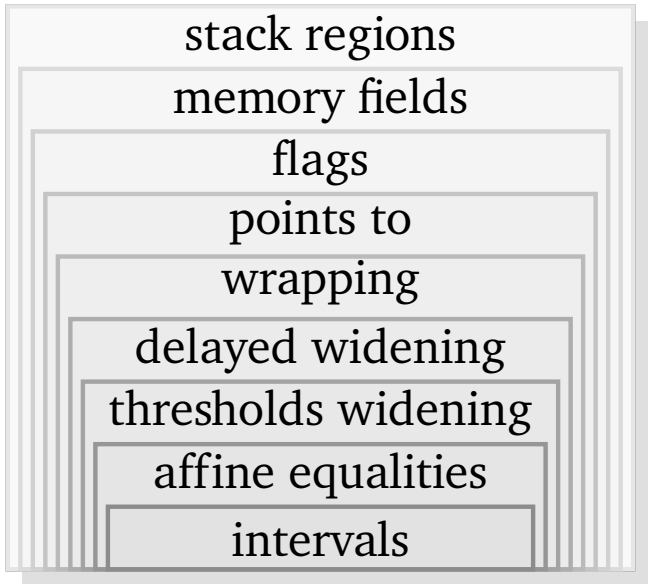
```
#define BUFFERSIZE 200
#define TRUE 1
#define FALSE 0
int copy_it (char *input, unsigned int length) {
    char c, localbuf[BUFFERSIZE];
    unsigned int upperlimit = BUFFERSIZE - 10;
    unsigned int quotation = roundquote = FALSE; quotation ∈ [0, 0], roundquote ∈ [0, 0]
    unsigned int inputIndex = outputIndex = 0;
    while (inputIndex < length) { ▽ removes bounds: quotation ∈ [0, +∞], roundquote ∈ [0, +∞]
        c = input[inputIndex++];
        if ((c == '<') && (!quotation)) {
            quotation = TRUE; upperlimit--;
        } ⊔ : quotation ∈ [0, 1]
        if ((c == '>') && (quotation)) {
            quotation = FALSE; upperlimit++;
        } ⊔ : quotation ∈ [0, 1]
        if ((c == '(') && (!quotation) && !roundquote) {
            roundquote = TRUE; upperlimit--; // decrementation was missing in bug
        } ⊔ : quotation ∈ [0, 1], roundquote ∈ [0, 1]
        if ((c == ')') && (!quotation) && roundquote) {
            roundquote = FALSE; upperlimit++;
        } ⊔ : quotation ∈ [0, 1], roundquote ∈ [0, 1]
        // If there is sufficient space in the buffer, write the character.
        if (outputIndex < upperlimit) {
            localbuf[outputIndex] = c;
            outputIndex++;
        }
    }
    if (roundquote) {
        localbuf[outputIndex] = ')'; outputIndex++; }
    if (quotation) {
        localbuf[outputIndex] = '>'; outputIndex++; }
}
```

Sendmail Code Revisited (Problems and Ideas)

```
#define BUFFERSIZE 200
#define TRUE 1
#define FALSE 0
int copy_it (char *input, unsigned int length) {
    char c, localbuf[BUFFERSIZE];
    unsigned int upperlimit = BUFFERSIZE - 10;
    unsigned int quotation = roundquote = FALSE; quotation ∈ [0, 0], roundquote ∈ [0, 0]
    unsigned int inputIndex = outputIndex = 0;
    while (inputIndex < length) { ▽ removes bounds: quotation ∈ [0, +∞], roundquote ∈ [0, +∞]
        c = input[inputIndex++];   delay widening until flags and relations stable!
        if ((c == '<') && (!quotation)) {
            quotation = TRUE; upperlimit--;
        } ⊔ : quotation ∈ [0, 1]
        if ((c == '>') && (quotation)) {
            quotation = FALSE; upperlimit++;
        } ⊔ : quotation ∈ [0, 1]
        if ((c == '(') && (!quotation) && !roundquote) {
            roundquote = TRUE; upperlimit--; // decrementation was missing in bug
        } ⊔ : quotation ∈ [0, 1], roundquote ∈ [0, 1]
        if ((c == ')') && (!quotation) && roundquote) {
            roundquote = FALSE; upperlimit++;
        } ⊔ : quotation ∈ [0, 1], roundquote ∈ [0, 1]
        // If there is sufficient space in the buffer, write the character.
        if (outputIndex < upperlimit) {
            localbuf[outputIndex] = c;
            outputIndex++;
        }
    }
    if (roundquote) {
        localbuf[outputIndex] = ')'; outputIndex++; }
    if (quotation) {
        localbuf[outputIndex] = '>'; outputIndex++; }
}
```

Stack of required domains

To verify the code (disassembled from the binary) we used these abstract domains:



Adding more domains (e.g. predicates, congruences, octagons, polyhedra, interval-sets) improves the precision of the inferred bounds after widening but is not necessary to verify the code.

Analyzed Code

Lets analyze the code!

```
int copy_it (char *input, unsigned int length) {
    char c, localbuf[200];
    unsigned int upperlimit = 190;
    unsigned int q = rq = 0;
    unsigned int inputIndex = outputIndex = 0;
    while (inputIndex < length) {
        c = input[inputIndex++];
        if (... && (!q)) {
            q = 1; upperlimit--;
        }
        if (... && (q)) {
            q = 0; upperlimit++;
        }
        if (... && (!q) && !rq) {
            rq = TRUE; upperlimit--;
        }
        if (... && (!q) && rq) {
            rq = 0; upperlimit++;
        }
        if (outputIndex < upperlimit) {
            localbuf[outputIndex] = c;
            outputIndex++;
        }
    }
    if (rq) {
        localbuf[outputIndex] = ')'; outputIndex++; }
    if (q) {
        localbuf[outputIndex] = '>'; outputIndex++; }
}
```

Analyzed Code

1st iteration: infers the affine equality between the variables: $upperlimit + q + rq = 190$

```
int copy_it (char *input, unsigned int length) {
    char c, localbuf[200];
    unsigned int upperlimit = 190;
    unsigned int q = rq = 0;
    unsigned int inputIndex = outputIndex = 0;
    while (inputIndex < length) { inputIndex = 0, outputIndex = 0, length ∈ [−∞, +∞]
        c = input[inputIndex++];
        if (... && (!q)) { upperlimit = 190, q = 0, rq = 0
            q = 1; upperlimit--;
        }   ⊔ : upperlimit + q = 190, upperlimit ∈ [189, 190], q ∈ [0, 1]
        if (... && (q)) {
            q = 0; upperlimit++;
        }
        if (... && (!q) && !rq) {
            rq = TRUE; upperlimit--;
        }
        if (... && (!q) && rq) {
            rq = 0; upperlimit++;
        }
        if (outputIndex < upperlimit) {
            localbuf[outputIndex] = c;
            outputIndex++;
        }
    }
    if (rq) {
        localbuf[outputIndex] = ')'; outputIndex++; }
    if (q) {
        localbuf[outputIndex] = '>'; outputIndex++; }
}
```

Analyzed Code

1st iteration: infers the affine equality between the variables: $upperlimit + q + rq = 190$

```
int copy_it (char *input, unsigned int length) {
    char c, localbuf[200];
    unsigned int upperlimit = 190;
    unsigned int q = rq = 0;
    unsigned int inputIndex = outputIndex = 0;
    while (inputIndex < length) {
        c = input[inputIndex++];
        if (... && (!q)) {
            q = 1; upperlimit--;
        }
        if (... && (q)) { upperlimit = 189, q = 1, rq = 0
            q = 0; upperlimit++;
        }   ⊔ : upperlimit + q = 190, upperlimit ∈ [189, 190], q ∈ [0, 1]
        if (... && (!q) && !rq) {
            rq = TRUE; upperlimit--;
        }
        if (... && (!q) && rq) {
            rq = 0; upperlimit++;
        }
        if (outputIndex < upperlimit) {
            localbuf[outputIndex] = c;
            outputIndex++;
        }
    }
    if (rq) {
        localbuf[outputIndex] = ')'; outputIndex++; }
    if (q) {
        localbuf[outputIndex] = '>'; outputIndex++; }
}
```

Analyzed Code

1st iteration: infers the affine equality between the variables: $upperlimit + q + rq = 190$

```
int copy_it (char *input, unsigned int length) {
    char c, localbuf[200];
    unsigned int upperlimit = 190;
    unsigned int q = rq = 0;
    unsigned int inputIndex = outputIndex = 0;
    while (inputIndex < length) {
        c = input[inputIndex++];
        if (... && (!q)) {
            q = 1; upperlimit--;
        }
        if (... && (q)) {
            q = 0; upperlimit++;
        }
        if (... && (!q) && !rq) { upperlimit = 190, q = 0, rq = 0
            rq = TRUE; upperlimit--;
        }   ⊔ : upperlimit + q + rq = 190, upperlimit ∈ [189, 190], q ∈ [0, 1], rq ∈ [0, 1]
        if (... && (!q) && rq) {
            rq = 0; upperlimit++;
        }
        if (outputIndex < upperlimit) {
            localbuf[outputIndex] = c;
            outputIndex++;
        }
    }
    if (rq) {
        localbuf[outputIndex] = ')'; outputIndex++; }
    if (q) {
        localbuf[outputIndex] = '>'; outputIndex++; }
}
```

Analyzed Code

1st iteration: infers the affine equality between the variables: $upperlimit + q + rq = 190$

```
int copy_it (char *input, unsigned int length) {
    char c, localbuf[200];
    unsigned int upperlimit = 190;
    unsigned int q = rq = 0;
    unsigned int inputIndex = outputIndex = 0;
    while (inputIndex < length) {
        c = input[inputIndex++];
        if (... && (!q)) {
            q = 1; upperlimit--;
        }
        if (... && (q)) {
            q = 0; upperlimit++;
        }
        if (... && (!q) && !rq) {
            rq = TRUE; upperlimit--;
        }
        if (... && (!q) && rq) {  upperlimit = 189, q = 0, rq = 1
            rq = 0; upperlimit++;
        }    ⊔ : upperlimit + q + rq = 190, upperlimit ∈ [189, 190], q ∈ [0, 1], rq ∈ [0, 1]
        if (outputIndex < upperlimit) {
            localbuf[outputIndex] = c;
            outputIndex++;
        }
    }
    if (rq) {
        localbuf[outputIndex] = ')'; outputIndex++; }
    if (q) {
        localbuf[outputIndex] = '>'; outputIndex++; }
}
```

Analyzed Code

1st iteration: infers the affine equality between the variables: $upperlimit + q + rq = 190$

```
int copy_it (char *input, unsigned int length) {
    char c, localbuf[200];
    unsigned int upperlimit = 190;
    unsigned int q = rq = 0;
    unsigned int inputIndex = outputIndex = 0;
    while (inputIndex < length) {
        c = input[inputIndex++];
        if (... && (!q)) {
            q = 1; upperlimit--;
        }
        if (... && (q)) {
            q = 0; upperlimit++;
        }
        if (... && (!q) && !rq) {
            rq = TRUE; upperlimit--;
        }
        if (... && (!q) && rq) {
            rq = 0; upperlimit++;
        }
        if (outputIndex < upperlimit) {  upperlimit ∈ [189, 190], outputIndex = 0
            localbuf[outputIndex] = c;
            outputIndex++;
        }    ⊔ : upperlimit ∈ [189, 190], outputIndex ∈ [0, 1]
    }
    if (rq) {
        localbuf[outputIndex] = ')'; outputIndex++; }
    if (q) {
        localbuf[outputIndex] = '>'; outputIndex++; }
}
```

Analyzed Code

2nd iteration: widening ∇ suppressed by the "delayed widening" domain because of the flag assignments. Join $\sqcup$ performed instead. We analyze the loop again with still valid equality: $upperlimit + q + rq = 190$

```
int copy_it (char *input, unsigned int length) {
    char c, localbuf[200];
    unsigned int upperlimit = 190;
    unsigned int q = rq = 0;
    unsigned int inputIndex = outputIndex = 0;
    while (inputIndex < length) { ⊔ : inputIndex ∈ [0, 1], outputIndex ∈ [0, 1]
        c = input[inputIndex++];
        if (... && (!q)) { upperlimit + rq = 190, upperlimit ∈ [189, 190], q = 0, rq ∈ [0, 1]
            q = 1; upperlimit--;
        }   ⊔ : upperlimit + q + rq = 190, upperlimit ∈ [188, 190], q ∈ [0, 1], rq ∈ [0, 1]
        if (... && (q)) {
            q = 0; upperlimit++;
        }
        if (... && (!q) && !rq) {
            rq = TRUE; upperlimit--;
        }
        if (... && (!q) && rq) {
            rq = 0; upperlimit++;
        }
        if (outputIndex < upperlimit) {
            localbuf[outputIndex] = c;
            outputIndex++;
        }
    }
    if (rq) {
        localbuf[outputIndex] = ')'; outputIndex++; }
    if (q) {
        localbuf[outputIndex] = '>'; outputIndex++; }
}
```

Analyzed Code

2nd iteration: widening ∇ suppressed by the "delayed widening" domain because of the flag assignments. Join $\sqcup$ performed instead. We analyze the loop again with still valid equality: $upperlimit + q + rq = 190$

```
int copy_it (char *input, unsigned int length) {
    char c, localbuf[200];
    unsigned int upperlimit = 190;
    unsigned int q = rq = 0;
    unsigned int inputIndex = outputIndex = 0;
    while (inputIndex < length) {
        c = input[inputIndex++];
        if (... && (!q)) {
            q = 1; upperlimit--;
        }
        if (... && (q)) { upperlimit + rq = 189, upperlimit ∈ [188, 189], q = 1, rq ∈ [0, 1]
            q = 0; upperlimit++;
        }   ⊔ : upperlimit + q + rq = 190, upperlimit ∈ [188, 190], q ∈ [0, 1], rq ∈ [0, 1]
        if (... && (!q) && !rq) {
            rq = TRUE; upperlimit--;
        }
        if (... && (!q) && rq) {
            rq = 0; upperlimit++;
        }
        if (outputIndex < upperlimit) {
            localbuf[outputIndex] = c;
            outputIndex++;
        }
    }
    if (rq) {
        localbuf[outputIndex] = ')'; outputIndex++; }
    if (q) {
        localbuf[outputIndex] = '>'; outputIndex++; }
}
```

Analyzed Code

2nd iteration: widening ∇ suppressed by the "delayed widening" domain because of the flag assignments. Join $\sqcup$ performed instead. We analyze the loop again with still valid equality: $upperlimit + q + rq = 190$

```
int copy_it (char *input, unsigned int length) {
    char c, localbuf[200];
    unsigned int upperlimit = 190;
    unsigned int q = rq = 0;
    unsigned int inputIndex = outputIndex = 0;
    while (inputIndex < length) {
        c = input[inputIndex++];
        if (... && (!q)) {
            q = 1; upperlimit--;
        }
        if (... && (q)) {
            q = 0; upperlimit++;
        }
        if (... && (!q) && !rq) { upperlimit = 190, q = 0, rq = 0
            rq = TRUE; upperlimit--;
        }   ⊔ : upperlimit + q + rq = 190, upperlimit ∈ [188, 190], q ∈ [0, 1], rq ∈ [0, 1]
        if (... && (!q) && rq) {
            rq = 0; upperlimit++;
        }
        if (outputIndex < upperlimit) {
            localbuf[outputIndex] = c;
            outputIndex++;
        }
    }
    if (rq) {
        localbuf[outputIndex] = ')'; outputIndex++; }
    if (q) {
        localbuf[outputIndex] = '>'; outputIndex++; }
}
```

Analyzed Code

2nd iteration: widening ∇ suppressed by the "delayed widening" domain because of the flag assignments. Join $\sqcup$ performed instead. We analyze the loop again with still valid equality: $upperlimit + q + rq = 190$

```
int copy_it (char *input, unsigned int length) {
    char c, localbuf[200];
    unsigned int upperlimit = 190;
    unsigned int q = rq = 0;
    unsigned int inputIndex = outputIndex = 0;
    while (inputIndex < length) {
        c = input[inputIndex++];
        if (... && (!q)) {
            q = 1; upperlimit--;
        }
        if (... && (q)) {
            q = 0; upperlimit++;
        }
        if (... && (!q) && !rq) {
            rq = TRUE; upperlimit--;
        }
        if (... && (!q) && rq) {  upperlimit = 189, q = 0, rq = 1
            rq = 0; upperlimit++;
        }   ⊔ : upperlimit + q + rq = 190, upperlimit ∈ [188, 190], q ∈ [0, 1], rq ∈ [0, 1]
        if (outputIndex < upperlimit) {
            localbuf[outputIndex] = c;
            outputIndex++;
        }
    }
    if (rq) {
        localbuf[outputIndex] = ')'; outputIndex++; }
    if (q) {
        localbuf[outputIndex] = '>'; outputIndex++; }
}
```

Analyzed Code

2nd iteration: widening ∇ suppressed by the "delayed widening" domain because of the flag assignments. Join $\sqcup$ performed instead. We analyze the loop again with still valid equality: $upperlimit + q + rq = 190$

```
int copy_it (char *input, unsigned int length) {
    char c, localbuf[200];
    unsigned int upperlimit = 190;
    unsigned int q = rq = 0;
    unsigned int inputIndex = outputIndex = 0;
    while (inputIndex < length) {
        c = input[inputIndex++];
        if (... && (!q)) {
            q = 1; upperlimit--;
        }
        if (... && (q)) {
            q = 0; upperlimit++;
        }
        if (... && (!q) && !rq) {
            rq = TRUE; upperlimit--;
        }
        if (... && (!q) && rq) {
            rq = 0; upperlimit++;
        }
        if (outputIndex < upperlimit) {  upperlimit ∈ [188, 190], outputIndex ∈ [0, 1]
            localbuf[outputIndex] = c;
            outputIndex++;
        }    ⊔ : upperlimit ∈ [188, 190], outputIndex ∈ [0, 2]
    }
    if (rq) {
        localbuf[outputIndex] = ')'; outputIndex++; }
    if (q) {
        localbuf[outputIndex] = '>'; outputIndex++; }
}
```

Analyzed Code

3rd iteration: now widening ∇ is applied using the widening threshold: $outputIndex - 1 < upperlimit$.
Widening changes the lower bound of *upperlimit* but reduction with the equality $upperlimit + q + rq = 190$ restores it

```
int copy_it (char *input, unsigned int length) {
    char c, localbuf[200];
    unsigned int upperlimit = 190;
    unsigned int q = rq = 0;
    unsigned int inputIndex = outputIndex = 0;
    while (inputIndex < length) { ∇ : inputIndex ∈ [0, +∞],outputIndex ∈ [0, 190],upperlimit ∈ [0, 190]
        c = input[inputIndex++];
        if (... && (!q)) { upperlimit + rq = 190, upperlimit ∈ [189, 190], q = 0, rq ∈ [0, 1]
            q = 1; upperlimit--;
        }   ⊔ : upperlimit + q + rq = 190, upperlimit ∈ [188, 190], q ∈ [0, 1], rq ∈ [0, 1]
        if (... && (q)) {
            q = 0; upperlimit++;
        }
        if (... && (!q) && !rq) {
            rq = TRUE; upperlimit--;
        }
        if (... && (!q) && rq) {
            rq = 0; upperlimit++;
        }
        if (outputIndex < upperlimit) {
            localbuf[outputIndex] = c;
            outputIndex++;
        }
    }
    if (rq) {
        localbuf[outputIndex] = ')'; outputIndex++; }
    if (q) {
        localbuf[outputIndex] = '>'; outputIndex++; }
}
```

Analyzed Code

3rd iteration: now widening ∇ is applied using the widening threshold: $outputIndex - 1 < upperlimit$.
Widening changes the lower bound of *upperlimit* but reduction with the equality $upperlimit + q + rq = 190$ restores it

```
int copy_it (char *input, unsigned int length) {
    char c, localbuf[200];
    unsigned int upperlimit = 190;
    unsigned int q = rq = 0;
    unsigned int inputIndex = outputIndex = 0;
    while (inputIndex < length) {
        c = input[inputIndex++];
        if (... && (!q)) {
            q = 1; upperlimit--;
        }
        if (... && (q)) { upperlimit + rq = 189, upperlimit ∈ [188, 189], q = 1, rq ∈ [0, 1]
            q = 0; upperlimit++;
        }   ⊔ : upperlimit + q + rq = 190, upperlimit ∈ [188, 190], q ∈ [0, 1], rq ∈ [0, 1]
        if (... && (!q) && !rq) {
            rq = TRUE; upperlimit--;
        }
        if (... && (!q) && rq) {
            rq = 0; upperlimit++;
        }
        if (outputIndex < upperlimit) {
            localbuf[outputIndex] = c;
            outputIndex++;
        }
    }
    if (rq) {
        localbuf[outputIndex] = ')'; outputIndex++; }
    if (q) {
        localbuf[outputIndex] = '>'; outputIndex++; }
}
```

Analyzed Code

3rd iteration: now widening ∇ is applied using the widening threshold: $outputIndex - 1 < upperlimit$.
Widening changes the lower bound of *upperlimit* but reduction with the equality $upperlimit + q + rq = 190$ restores it

```
int copy_it (char *input, unsigned int length) {
    char c, localbuf[200];
    unsigned int upperlimit = 190;
    unsigned int q = rq = 0;
    unsigned int inputIndex = outputIndex = 0;
    while (inputIndex < length) {
        c = input[inputIndex++];
        if (... && (!q)) {
            q = 1; upperlimit--;
        }
        if (... && (q)) {
            q = 0; upperlimit++;
        }
        if (... && (!q) && !rq) { upperlimit = 190, q = 0, rq = 0
            rq = TRUE; upperlimit--;
        }   ⊔ : upperlimit + q + rq = 190, upperlimit ∈ [188, 190], q ∈ [0, 1], rq ∈ [0, 1]
        if (... && (!q) && rq) {
            rq = 0; upperlimit++;
        }
        if (outputIndex < upperlimit) {
            localbuf[outputIndex] = c;
            outputIndex++;
        }
    }
    if (rq) {
        localbuf[outputIndex] = ')'; outputIndex++; }
    if (q) {
        localbuf[outputIndex] = '>'; outputIndex++; }
}
```

Analyzed Code

3rd iteration: now widening ∇ is applied using the widening threshold: $outputIndex - 1 < upperlimit$.
Widening changes the lower bound of $upperlimit$ but reduction with the equality $upperlimit + q + rq = 190$ restores it

```
int copy_it (char *input, unsigned int length) {
    char c, localbuf[200];
    unsigned int upperlimit = 190;
    unsigned int q = rq = 0;
    unsigned int inputIndex = outputIndex = 0;
    while (inputIndex < length) {
        c = input[inputIndex++];
        if (... && (!q)) {
            q = 1; upperlimit--;
        }
        if (... && (q)) {
            q = 0; upperlimit++;
        }
        if (... && (!q) && !rq) {
            rq = TRUE; upperlimit--;
        }
        if (... && (!q) && rq) { upperlimit = 189, q = 0, rq = 1
            rq = 0; upperlimit++;
        }   ⊔ : upperlimit + q + rq = 190, upperlimit ∈ [188, 190], q ∈ [0, 1], rq ∈ [0, 1]
        if (outputIndex < upperlimit) {
            localbuf[outputIndex] = c;
            outputIndex++;
        }
    }
    if (rq) {
        localbuf[outputIndex] = ')'; outputIndex++; }
    if (q) {
        localbuf[outputIndex] = '>'; outputIndex++; }
}
```

Analyzed Code

3rd iteration: now widening ∇ is applied using the widening threshold: $outputIndex - 1 < upperlimit$.
Widening changes the lower bound of *upperlimit* but reduction with the equality $upperlimit + q + rq = 190$ restores it

```
int copy_it (char *input, unsigned int length) {
    char c, localbuf[200];
    unsigned int upperlimit = 190;
    unsigned int q = rq = 0;
    unsigned int inputIndex = outputIndex = 0;
    while (inputIndex < length) {
        c = input[inputIndex++];
        if (... && (!q)) {
            q = 1; upperlimit--;
        }
        if (... && (q)) {
            q = 0; upperlimit++;
        }
        if (... && (!q) && !rq) {
            rq = TRUE; upperlimit--;
        }
        if (... && (!q) && rq) {
            rq = 0; upperlimit++;
        }
        if (outputIndex < upperlimit) { upperlimit ∈ [188, 190], outputIndex ∈ [0, 189]
            localbuf[outputIndex] = c;
            outputIndex++;
        }   ⊔ : upperlimit ∈ [188, 190], outputIndex ∈ [0, 190]
    }
    if (rq) {
        localbuf[outputIndex] = ')'; outputIndex++; }
    if (q) {
        localbuf[outputIndex] = '>'; outputIndex++; }
}
```

Analyzed Code

4th iteration: loop is stable; outside of the loop body the value of *outputIndex* is still bounded!

```
int copy_it (char *input, unsigned int length) {
    char c, localbuf[200];
    unsigned int upperlimit = 190;
    unsigned int q = rq = 0;
    unsigned int inputIndex = outputIndex = 0;
    while (inputIndex < length) { ⊑: inputIndex ∈ [0, +∞], outputIndex ∈ [0, 190]
        c = input[inputIndex++];
        if (... && (!q)) {
            q = 1; upperlimit--;
        }
        if (... && (q)) {
            q = 0; upperlimit++;
        }
        if (... && (!q) && !rq) {
            rq = TRUE; upperlimit--;
        }
        if (... && (!q) && rq) {
            rq = 0; upperlimit++;
        }
        if (outputIndex < upperlimit) {
            localbuf[outputIndex] = c;
            outputIndex++;
        }
    }
    if (rq) { outputIndex ∈ [0, 190]
        localbuf[outputIndex] = ')'; outputIndex++; }
    if (q) {  outputIndex ∈ [1, 191]
        localbuf[outputIndex] = '>'; outputIndex++; }
}
```

Key Points

- widening needs to be suppressed until the flag variables are stable to infer the equality relation with *upperlimit*
- the inferred equality $\mathit{upperlimit} + q + rq = 190$ and reduction between domains results in more precise values for *upperlimit*; recovers precision loss of widening
- narrowing does not help here; instead the threshold $\mathit{outputIndex} < \mathit{upperlimit}$ must be used for widening
  $\leadsto$ *outputIndex* is also restricted outside of the loop for the following two writes to the buffer
- in the vulnerable version because of the missing decrementation the equality relation does not hold
  $\leadsto$ *upperlimit* is unbounded after widening

Unfortunately

The original Sendmail code is more complex!

- the fix is not only one line but in more than just one place

Unfortunately

The original Sendmail code is more complex!

- the fix is not only one line but in more than just one place
- the code contains ∼10 loops (nesting depth is 4) and gotos

Unfortunately

The original Sendmail code is more complex!

- the fix is not only one line but in more than just one place
- the code contains ∼10 loops (nesting depth is 4) and gotos
- lots of pointer arithmetic inside loops

Unfortunately

The original Sendmail code is more complex!

- the fix is not only one line but in more than just one place
- the code contains ∼10 loops (nesting depth is 4) and gotos
- lots of pointer arithmetic inside loops
- uses string manipulating functions

Unfortunately

The original Sendmail code is more complex!

- the fix is not only one line but in more than just one place
- the code contains ∼10 loops (nesting depth is 4) and gotos
- lots of pointer arithmetic inside loops
- uses string manipulating functions

Unfortunately

The original Sendmail code is more complex!

- the fix is not only one line but in more than just one place
- the code contains ∼10 loops (nesting depth is 4) and gotos
- lots of pointer arithmetic inside loops
- uses string manipulating functions

$\rightsquigarrow$ we cannot yet prove the invariant on that code

Conclusion

- abstract domains can infer surprisingly nice results.
  $\leadsto$ in this case a simpler invariant than expected.

Conclusion

- abstract domains can infer surprisingly nice results.
  $\rightsquigarrow$ in this case a simpler invariant than expected.
- but the fixpoint computation and the reduction between the domains is quite complex
  $\rightsquigarrow$ hard to debug and reason about inferred invariants given the load of data (multiply with verbosity of machine code)

Conclusion

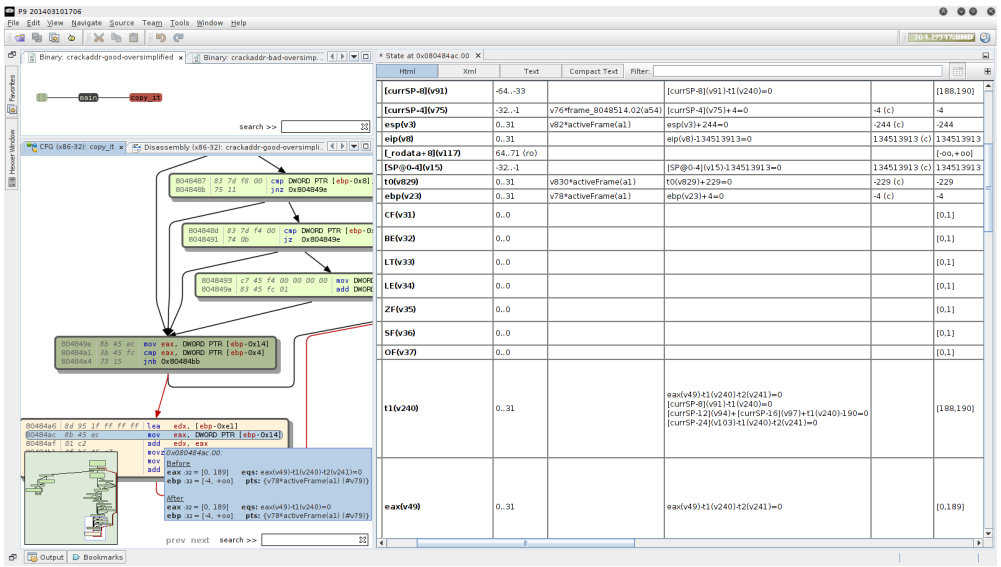
- abstract domains can infer surprisingly nice results.
  $\leadsto$ in this case a simpler invariant than expected.
- but the fixpoint computation and the reduction between the domains is quite complex
  $\leadsto$ hard to debug and reason about inferred invariants given the load of data (multiply with verbosity of machine code)
- if an expected invariant cannot be proved it is difficult to find out why and fix it

Conclusion

- abstract domains can infer surprisingly nice results.
  $\rightsquigarrow$ in this case a simpler invariant than expected.
- but the fixpoint computation and the reduction between the domains is quite complex
  $\rightsquigarrow$ hard to debug and reason about inferred invariants given the load of data (multiply with verbosity of machine code)
- if an expected invariant cannot be proved it is difficult to find out why and fix it
- being able to understand and debug an abstract analyzer is key to building useful analyses!
  $\rightsquigarrow$ general adoption of static analyzers is an uphill battle :(

Demo

Time to analyze it!



Project page: **https://bitbucket.org/mihaila/bindead**