Soundness of a Dataflow Analysis for Memory Monitoring

Dara Ly [1,3] Nikolai Kosmatov[1]
Frédéric Loulergue [2] Julien Signoles[1]

[1]CEA LIST, Software Reliability and Security Laboratory

[2]Northern Arizona Univ., School of Informatics Computing and Cyber Systems

[3]Univ. Orléans, Laboratoire d'Informatique Fondamentale

Table of Contents

Table of Contents

Runtime Assertion Checking

Static verification

Examples:

- abstract interpretation: mainly deals with undefined behaviours
- deductive verification: requires significant annotation work

Reasoning about all possible executions is hard!

Runtime Assertion Checking

Static verification

Examples:

- abstract interpretation: mainly deals with undefined behaviours
- deductive verification: requires significant annotation work

Reasoning about all possible executions is hard!

Dynamic verification

Analyze a single execution trace.

- Runtime Assertion Checking (RAC): check properties while running the program

E-ACSL: the RAC tool of Frama-C

Translate annotations written in
a dedicated specification language into
executable C code

```
int main(void) {
  int x = 0;
  /*@ assert x+1 == 0; */

  return 0;
}
```

Software Analyzers

E-ACSL: the RAC tool of Frama-C

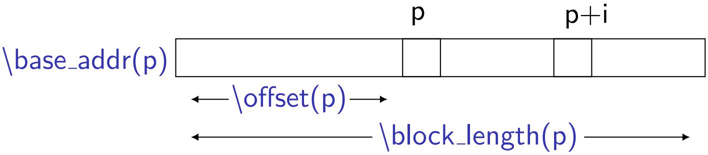
Translate annotations written in
a dedicated specification language into
executable C code

```
int main(void) {
  int x = 0;
  /*@ assert x+1 == 0; */
  e_acsl_assert(x + 1 == 0);
  return 0;
}
```

frama C
Software Analyzers

Expressive Memory-related builtins

Properties	Informal semantics
`\valid(a)`	dereferencing a is safe ?
`\init(a)`	contents of a has been initialized ?
`\base_address(a)`	base address of the block containing a
`\block_length(a)`	size (in bytes) of the block containing a
`\offset(a)`	offset (in bytes) of a in its block



Evaluating these properties requires storing metadata at runtime.

Code Generation

```
void f(void) {
  int x, y, z, *p;



  p = &x;
  x = 0;
  y = 1;
  z = 2;
  /*@ assert \valid(p); */




  *p = 3;



  return;
}
```

Code Generation

```
void f(void) {
  int x, y, z, *p;
  // local variable allocation
  store_block((void *)(&p), 4U); store_block((void *)(&z), 4U);
  store_block((void *)(&y), 4U); store_block((void *)(&x), 4U);
  full_init((void *)(&p)); p = &x; // initialization of p
  full_init((void *)(&x)); x = 0;  // initialization of x
  full_init((void *)(&y)); y = 1;  // initialization of y
  full_init((void *)(&z)); z = 2;  // initialization of z
  /*@ assert \valid(p); */
  // validity check
  { int e_acsl_valid;
    e_acsl_valid = valid((void *)p, sizeof(int));
    e_acsl_assert(e_acsl_valid); }
  *p = 3;
  // memory deallocation
  delete_block((void *)(&p)); delete_block((void *)(&z));
  delete_block((void *)(&y)); delete_block((void *)(&x));
  return;
}
```

Code Generation

```
void f(void) {
  int x, y, z, *p;
  // local variable allocation
  store_block((void *)(&p), 4U); store_block((void *)(&z), 4U);
  store_block((void *)(&y), 4U); store_block((void *)(&x), 4U);
  full_init((void *)(&p)); p = &x; // initialization of p
  full_init((void *)(&x)); x = 0;  // initialization of x
  full_init((void *)(&y)); y = 1;  // initialization of y
  full_init((void *)(&z)); z = 2;  // initialization of z
  /*@ assert \valid(p); */
  // validity check
  { int e_acsl_valid;
    e_acsl_valid = valid((void *)p, sizeof(int));
    e_acsl_assert(e_acsl_valid); }
  *p = 3;
  // memory deallocation
  delete_block((void *)(&p)); delete_block((void *)(&z));
  delete_block((void *)(&y)); delete_block((void *)(&x));
  return;
}
```

significant performance gain [Vorobyov, ISMM'17]

Table of Contents

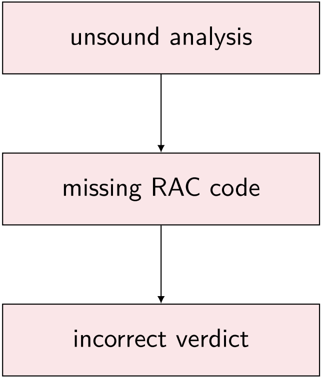
Motivation

Question

At a given program point, what are the memory blocks that could be used later to evaluate a predicate ?

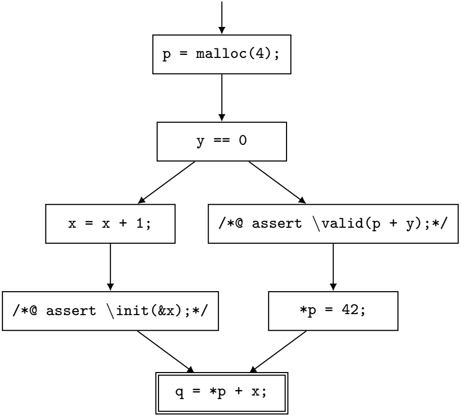
Sound Analysis Required

sound = all needed blocks are monitored



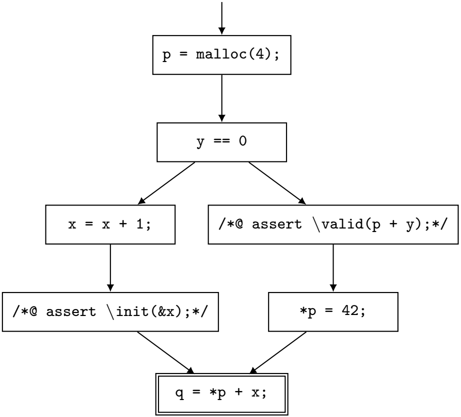
Backward Over-approximating Dataflow Analysis

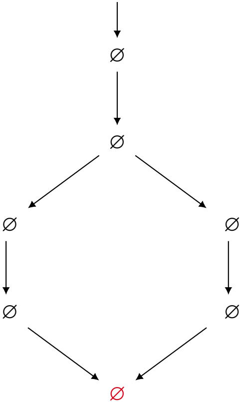
to optimize (minimize !) code generation



Backward Over-approximating Dataflow Analysis

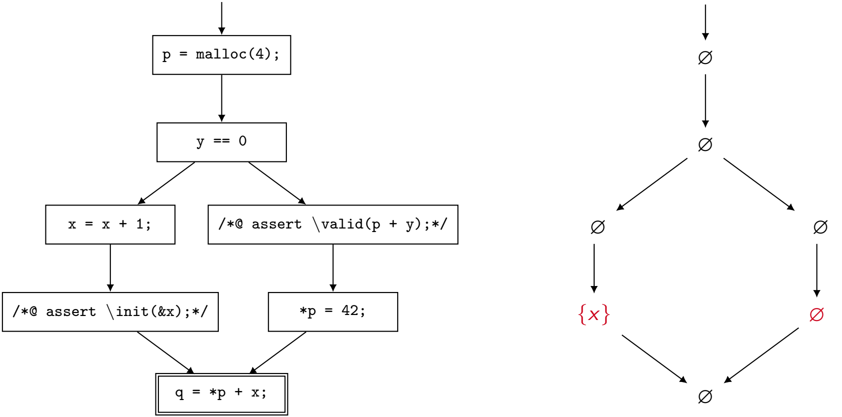
to optimize (minimize !) code generation





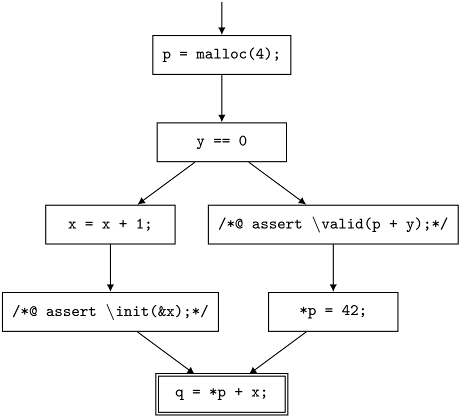
Backward Over-approximating Dataflow Analysis

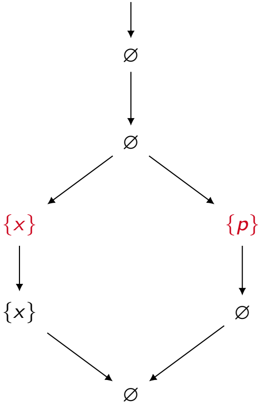
to optimize (minimize !) code generation



Backward Over-approximating Dataflow Analysis

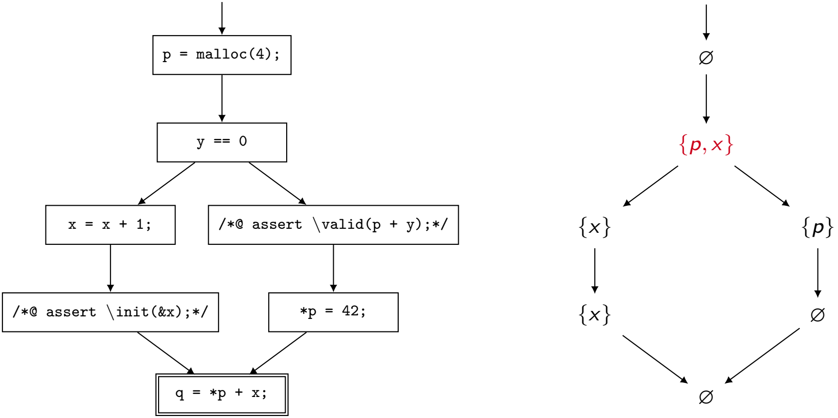
to optimize (minimize !) code generation





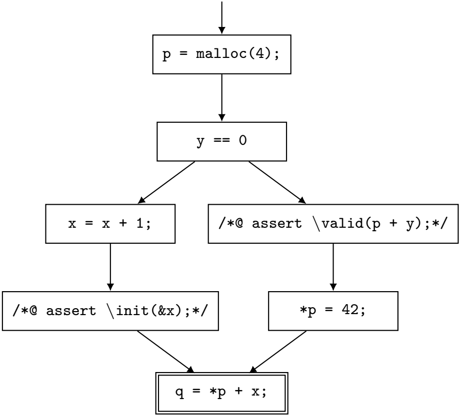
Backward Over-approximating Dataflow Analysis

to optimize (minimize !) code generation



Backward Over-approximating Dataflow Analysis

to optimize (minimize !) code generation



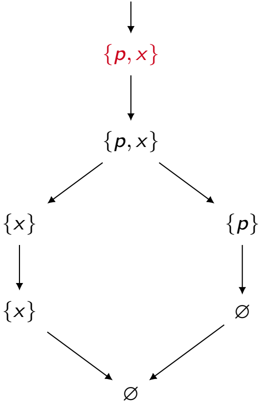


Table of Contents

Elements of the Formalization

- (abstract) syntax
- memory model
- semantics

Syntax

Expressions ::= pointer arithmetics (addresses, offsets), ...

Statements s ::= `skip;`
| lv = e;
| lv = `malloc(`e`);`
| `free(`lv`);`
| `/*@ assert` p; `*/`
| `if (`e`) then` s `else` s
| `while (`e`)` s
| s s

Predicates ::= propositional logic + built-in memory properties

Proposed Semantics

Standard small-steps operational semantics:

$$\langle x = 3;\ , \varnothing \rangle \rightarrow \langle \texttt{skip;}\ \ , [x \mapsto 3] \rangle$$

Extended semantics for memory monitoring:

- the usual execution memory M
- a new observation memory $\overline{M}$

Proposed Semantics

Standard small-steps operational semantics:

$$\langle x = 3;\ , \varnothing\rangle \rightarrow \langle \texttt{skip;}\ \ , [x \mapsto 3]\rangle$$

Extended semantics for memory monitoring:

- the usual execution memory M
- a new observation memory $\overline{M}$
  - represents the "real" memory, used by the E-ACSL monitor to evaluate predicates

$$\langle s_1, M_1, \overline{M_1}\rangle \rightarrow \langle s_2, M_2, \overline{M_2}\rangle$$

Two Flavors of Memory

Execution Memory

- adapted from the CompCert Memory Model [Leroy08]
- an abstract type mem + 4 operations

$$
\begin{aligned}
\texttt{alloc} &: \texttt{mem} \times \mathbb{Z} \times \mathbb{Z} \rightarrow \texttt{block} \times \texttt{mem} \\
\texttt{dealloc} &: \texttt{mem} \times \texttt{block} \rightarrow \texttt{option mem} \\
\texttt{load} &: \texttt{mem} \times \texttt{block} \times \mathbb{Z} \rightarrow \texttt{option val} \\
\texttt{store} &: \texttt{mem} \times \texttt{block} \times \mathbb{Z} \times \texttt{val} \rightarrow \texttt{option mem}
\end{aligned}
$$

- algebraic definitions

Observation Memory

similarly defined

Allocation Rule

Semantics defined by a **set of inference rules**

EVAL-MALLOC

$$\frac{}{\langle lv = \texttt{malloc}(e);\ , M_1, \overline{M_1}\rangle \rightarrow (M_2, \overline{M_2})}$$

Allocation Rule

Semantics defined by a **set of inference rules**

EVAL-MALLOC

$$\frac{M_1 \vDash_{\mathrm{e}} e \Rightarrow \mathtt{Int}(n) \qquad M_1 \vDash_{\mathrm{lv}} lv \Leftarrow (b, \delta)}{\langle lv = \mathtt{malloc}(e);\ , M_1, \overline{M_1} \rangle \to (M_2, \overline{M_2})}$$

Allocation Rule

Semantics defined by a **set of inference rules**

EVAL-MALLOC

$$\frac{\begin{array}{l} hi - lo = n \\ M_1 \vDash_{\mathrm{e}} e \Rightarrow \texttt{Int}(n) \quad \texttt{alloc}(M_1, lo, hi) = (b', M_3) \\ M_1 \vDash_{\mathrm{lv}} lv \Leftarrow (b, \delta) \end{array}}{\langle lv = \texttt{malloc}(e);\ , M_1, \overline{M_1} \rangle \rightarrow (M_2, \overline{M_2})}$$

Allocation Rule

Semantics defined by a **set of inference rules**

EVAL-MALLOC

$$\frac{\begin{array}{l} hi - lo = n \\ M_1 \vDash_{\mathrm{e}} e \Rightarrow \mathtt{Int}(n) \quad \mathtt{alloc}(M_1, lo, hi) = (b', M_3) \\ M_1 \vDash_{\mathrm{lv}} lv \Leftarrow (b, \delta) \quad \mathtt{store}(M_3, b, \delta, \mathtt{Ptr}(b', 0)) = \mathtt{Some}(M_2) \end{array}}{\langle lv = \mathtt{malloc}(e);\ , M_1, \overline{M_1} \rangle \to (M_2, \overline{M_2})}$$

Allocation Rule

Semantics defined by a **set of inference rules**

EVAL-MALLOC

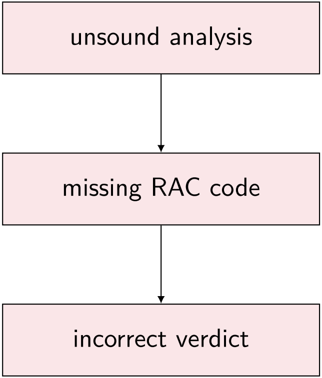
$$\dfrac{\begin{array}{ll} & \texttt{store\_block}(\overline{M_1}, b', lo, hi) = \overline{M_3} \\ hi - lo = n & \texttt{initialize}(\overline{M_3}, b, \delta) = \overline{M_2} \\ M_1 \vDash_{\mathrm{e}} e \Rightarrow \texttt{Int}(n) & \texttt{alloc}(M_1, lo, hi) = (b', M_3) \\ M_1 \vDash_{\mathrm{lv}} lv \Leftarrow (b, \delta) & \texttt{store}(M_3, b, \delta, \texttt{Ptr}(b', 0)) = \texttt{Some}(M_2) \end{array}}{\langle lv = \texttt{malloc}(e);\ , M_1, \overline{M_1} \rangle \rightarrow (M_2, \overline{M_2})}$$

Remember our Initial Motivation. . .

Question

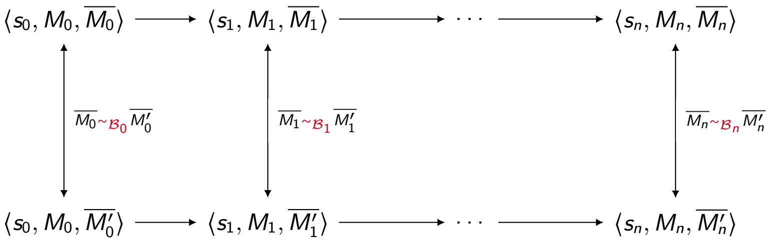
At a given program point, what are the memory blocks that could be used later to evaluate a predicate ?

Sound Analysis Required



Theorem

Intuition: the unoptimized program and the optimized one have the same behaviour



- $\mathcal{B}_i$: blocks computed by the analysis at program point i
- $\overline{M} \sim_{\mathcal{B}} \overline{M'}$: observation memories have the same values on $\mathcal{B}$

Conclusion

Contributions

- defined the semantics of a language with Runtime Assertion Checking for memory-related properties, using a notion of observation memory
- formally defined a compiler's optimization for such a language
- proved its soundness

Conclusion

Contributions

- defined the semantics of a language with Runtime Assertion Checking for memory-related properties, using a notion of observation memory
- formally defined a compiler's optimization for such a language
- proved its soundness

Future work

- additional constructs
- interprocedural analysis
- formalizing the E-ACSL translation

Expressions Evaluation

$$\textsc{LE-var}\quad \frac{E(x) = b}{M \vDash_{\mathrm{lv}} x \Leftarrow (b, 0)}$$

$$\textsc{LE-deref}\quad \frac{M \vDash_{\mathrm{e}} a \Rightarrow \mathtt{Ptr}(b, \delta)}{M \vDash_{\mathrm{lv}} \star a \Leftarrow (b, \delta)}$$

$$\textsc{EE-lval}\quad \frac{v \neq \mathtt{Undef} \qquad M \vDash_{\mathrm{lv}} l \Leftarrow (b, \delta) \qquad \mathtt{load}(M, b, \delta) = \mathtt{Some}(v)}{M \vDash_{\mathrm{e}} l \Rightarrow v}$$

$$\textsc{EE-shift}\quad \frac{M \vDash_{\mathrm{e}} a \Rightarrow \mathtt{Ptr}(b, \delta) \qquad M \vDash_{\mathrm{e}} e \Rightarrow \mathtt{Int}(n)}{M \vDash_{\mathrm{e}} a \mathbin{++} e \Rightarrow \mathtt{Ptr}(b, \delta + n)}$$

$$\textsc{EE-addr}\quad \frac{M \vDash_{\mathrm{lv}} l \Leftarrow (b, \delta)}{M \vDash_{\mathrm{e}} \& l \Rightarrow \mathtt{Ptr}(b, \delta)}$$

Statements Evaluation

ASSIGN

$$\frac{\begin{array}{ll} M_1 \vDash_{\mathrm{e}} e \Rightarrow v & \texttt{store}(M_1, b, \delta, v) = \texttt{Some}(M_2) \\ M_1 \vDash_{\mathrm{lv}} l \Leftarrow (b, \delta) & \texttt{initialize}(\overline{M_1}, b, \delta) = \overline{M_2} \end{array}}{\langle l = e;\ , M_1, \overline{M_1}\rangle \to (M_2, \overline{M_2})}$$

FREE

$$\frac{\begin{array}{c} M_1 \vDash_{\mathrm{e}} a \Rightarrow (b, 0) \\ \texttt{free}(M_1, b) = \texttt{Some}(M_2) \qquad \texttt{delete\_block}(\overline{M_1}, b) = \overline{M_2} \end{array}}{\langle \texttt{free}(a);\ , M_1, \overline{M_1}\rangle \to (M_2, \overline{M_2})}$$

Dataflow Analysis Definition

$$
\begin{aligned}
live_{out}(l) &\supseteq \begin{cases} \varnothing & \text{if } l \in \mathcal{F}(s) \\ \bigcup\{live_{in}(l') \mid (l, l') \in flow(s)\} & \text{otherwise} \end{cases} \\
live_{in}(l) &\supseteq live_{out}(l) \cup gen(l)
\end{aligned}
$$

$$
gen\left([= e;]^l\right) = \begin{cases} \{(e)\} & \text{if } lv \text{ is a pointer, and } \exists x \in live_{out}(l),\ \&x\mathcal{A} \\ \varnothing & \text{otherwise} \end{cases}
$$

$$
\begin{aligned}
gen\left([\texttt{skip;}]^l\right) &= \varnothing & gen\left([e]^l\right) &= \varnothing & gen\left([p]^l\right) &= \rho(p) \\
gen\left([lv = \texttt{alloc}(e);]^l\right) &= \varnothing & gen\left([\texttt{free}(l);]^l\right) &= \varnothing
\end{aligned}
$$

$$
\begin{aligned}
\rho(\texttt{\textbackslash valid}(a)) &= \{(a)\} & \rho(p_1 \odot p_2) &= \rho(p_1) \cup \rho(p_2) & \odot &\in \{\wedge, \vee\} \\
\rho(\texttt{\textbackslash init}(a)) &= \{(a)\} & \rho(t_1 \diamond t_2) &= \theta(t_1) \cup \theta(t_2) & \diamond &\in \{\equiv, \leq\} \\
\rho(\neg p) &= \rho(p)
\end{aligned}
$$

$$
\begin{aligned}
\theta(e) &= \varnothing & \theta(\texttt{\textbackslash base\_address}(a)) &= \{(a)\} \\
\theta(\texttt{\textbackslash offset}(a)) &= \{(a)\} & \theta(\texttt{\textbackslash block\_length}(a)) &= \{(a)\}
\end{aligned}
$$