Analysing Memory Resource Bounds for Low-Level Programs

Andreas Abel

Seminar Static Program Analysis
03.07.2010
• on March 12, 1995 the railway switch tower (Stellwerk) at Hamburg-Altona station was replaced by a fully computerized system for more than 30 Million €

• on March 13, 1995 the central computer of the new system failed
  ➢ the whole station had to be closed for more than two days
  ➢ about 30,000 passengers each day had to start from locations up to 25km away
  ➢ every fifth train in Germany was delayed
the reason:

– the stack size was 3.5kB
– already the normal office-hour traffic caused a stack overflow

the solution:

– the stack size was increased to 4kB
Problem

- embedded software systems often operate on platforms with limited memory
- failing, because of insufficient memory, can have costly consequences
- over-estimating the required memory leads to greater hardware cost without giving any guarantee on adequacy
Solution

• develop a formal system to predict memory requirements that is
  – provably sound
  – uses only safe approximations
Previous Work

• mostly based on analyzing functional programs
• other works rely on user-supplied annotations
Overview

• useful definitions and techniques
• overview of the inference system
• details of the inference system
• experiments
Memory Usage and Bounds

• Memory Usage: net usage at the end of the computation
• Memory Bound: high watermark of memory usage at all points over its computation
Example

```c
void f(c x, c y, d z) {
    x = new c();
    dispose(z);
    y = new c();
    dispose(x);
    dispose(y);
}
```

<table>
<thead>
<tr>
<th>Usage</th>
<th>Bound</th>
</tr>
</thead>
<tbody>
<tr>
<td>(c, d)</td>
<td>(c, d)</td>
</tr>
<tr>
<td>(0, 0)</td>
<td>(0, 0)</td>
</tr>
<tr>
<td>(1, 0)</td>
<td>(1, 0)</td>
</tr>
<tr>
<td>(1, -1)</td>
<td>(1, 0)</td>
</tr>
<tr>
<td>(2, -1)</td>
<td>(2, 0)</td>
</tr>
<tr>
<td>(1, -1)</td>
<td>(2, 0)</td>
</tr>
<tr>
<td>(0, -1)</td>
<td>(2, 0)</td>
</tr>
</tbody>
</table>
void f1(int n, c x, c y) {
    int v = n+1; //\{(c,0)\}
    if (v<5) {
        x = new c(); //\{(c,1)\}
        y = new c(); //\{(c,2)\}
        dispose(x); //\{(c,1)\}
        dispose(y); //\{(c,0)\}
    } else {
        x = new c(); //\{(c,1)\}
        dispose(x); //\{(c,0)\}
        x = new c(); //\{(c,1)\}
    }
}

Guarded expression
for heap usage: \( n < 4 \rightarrow \{(c, 0)\}, n \geq 4 \rightarrow \{(c, 1)\} \)
for heap bound: \( n < 4 \rightarrow \{(c, 2)\}, n \geq 4 \rightarrow \{(c, 1)\} \)
guarded expression
for heap usage: \( \{ n < 4 \rightarrow \{(c, 0)\}, n \geq 4 \rightarrow \{(c, 1)\}\} \)
for heap bound: \( \{ n < 4 \rightarrow \{(c, 2)\}, n \geq 4 \rightarrow \{(c, 1)\}\} \)

in general, a guarded expression is of the form

\[
\big\{ g_i \rightarrow B_i \big\}_{i=1}^n
\]
we use a low-level language because

• it can represent the intermediate form for a variety of higher-level languages

• resource usage may be affected by optimizing compilers
Syntax

\[ P ::= M_1, \ldots, M_n \]

\[ M ::= t \ m(t_1, \ldots, t_n) \ l \ \{ E \} \]

\[ E ::= \text{Cmd} \mid E_1; E_2 \mid \text{if } E_1 \ E_2 \mid \text{while } E \]

\[ \text{Cmd} ::= \text{load}(t) \ i \mid \text{store}(t) \ i \mid \text{invoke } m \mid \text{const}(t) \ k \]

\[ \mid \text{new } c \mid \text{dispose } c \]

\[ t ::= \text{bool} \mid \text{int} \mid \text{float} \mid \text{ref} \mid \text{void} \mid \cdots \]

\[ c \in \text{ObjType} \quad \text{(Set of Object Types)} \]
Goal:
given a method
\[ tm(t_1, \ldots, t_n)l\{\ldots\} \]
infer the following extended declaration
\[ tm(t_1, \ldots, t_n)l; \phi_{pr}; F; \phi_{po}; S; H_{po}; M_{po}\{\ldots\} \]

- precondition
- frame bound
- postcondition
- stack bound
- heap bound
- heap usage
Steps

- Frame bound inference
- Abstract State Inference
- Stack inference
- Heap Inference
the size of each stack frame is inferred using rules of the following form:

\[ l, \Gamma \vdash_F E \rightsquigarrow A, \Gamma_1, \mathcal{F} \]

intermediate code:

\[
A ::= (p, E_A) \\
E_A ::= Cmd \mid A; A \mid \text{if } A \mid \text{while } A
\]
Frame Bound Inference

- The `const<t> k` instruction places the value k of type t on the top of the stack.

\[
\text{[FS-CONST]}
\]

\[
k::t \quad \Gamma_1 = t: \Gamma
\]

\[
l, \Gamma \vdash_F \text{const}(<t> \ k) \leadsto (|\Gamma|, \text{const}(<t> \ k)), \Gamma_1, |\Gamma_1|
\]
the `load< t >` instruction copies the value of the local variable at position $i$ (which needs to have type $t$) to the top of the stack.
Frame Bound Inference

- The \texttt{store<t> i} instruction moves the value from the top of the stack to the local variable at position \texttt{i}.

\[
\frac{i \leq l \leq \lvert \Gamma \rvert}{l, t: \Gamma \vdash F \text{ store(t) } i \leadsto (r, \text{ store(t) } i), \Gamma_1, r}
\]
Frame Bound Inference

- \( \text{new } c \) command allocates a new object of type \( c \) on the heap and places a reference to it on the top of the stack.
Frame Bound Inference

- the `dispose c` command removes an object of type `c` from the heap whose reference is on top of the stack

\[
\begin{align*}
\text{[FS-DISPOSE]} \\
\begin{array}{c}
el \leq |\Gamma| \\
r = |\Gamma| + 1 \\
l, \text{ref: } \Gamma \vdash_F \text{dispose } c \rightsquigarrow (r, \text{dispose } c), \Gamma, r
\end{array}
\end{align*}
\]
Frame Bound Inference

- a boolean test is expected to be already on top of the stack
Frame Bound Inference

\[
\begin{align*}
\text{[FS-IF]} \\
\frac{l, \Gamma \vdash_F E_1 \leadsto A_1, \Gamma_1, \mathcal{F}_1 \quad l \leq |\Gamma| \quad |\Gamma_1| = |\Gamma_2|}{l, \Gamma \vdash_F E_2 \leadsto A_2, \Gamma_2, \mathcal{F}_2 \quad \mathcal{F}_3 = \max(\mathcal{F}_1, \mathcal{F}_2) \quad \Gamma_3 = \Gamma_1 \sqcup \Gamma_2}
\end{align*}
\]

\[
l, \text{bool}: \Gamma \vdash_F \text{if } E_1 \ E_2 \leadsto (|\Gamma| + 1, \text{if } A_1 \ A_2), \Gamma_3, \mathcal{F}_3
\]

the if $E_1 \ E_2$ instruction:

1. infer the frame bounds $\mathcal{F}_1$ and $\mathcal{F}_2$ for $E_1$ and $E_2$
2. the new frame bound is $\max(\mathcal{F}_1, \mathcal{F}_2)$
Frame Bound Inference

- The `invoke m` instruction calls a method `m` after its arguments have been placed on the stack; on return, the arguments are removed and the result is placed on the stack.
Frame Bound Inference

- for a sequence of two instructions, the frame bound is the maximum of the two frame bounds

\[
\text{[FS-SEQ]}
\]

\[
\begin{align*}
 l, \Gamma \vdash_F E_1 & \rightsquigarrow A_1, \Gamma_1, F_1 \\
 l, \Gamma_1 \vdash_F E_2 & \rightsquigarrow A_2, \Gamma_2, F_2 \\
 l, \Gamma \vdash_F E_1; E_2 & \rightsquigarrow (|\Gamma|, A_1; A_2), \Gamma_2, \max(F_1, F_2)
\end{align*}
\]
the +2 is because of the pointer to the previous frame and the return address
What we have achieved so far

Goal:
given a method
\[ tm(t_1, \ldots, t_n) l \{ \ldots \} \]
infer the following extended declaration
\[ tm(t_1, \ldots, t_n) l; \phi_{pr}; \mathcal{F}; \phi_{po}; S; \mathcal{H}_{po}; \mathcal{M}_{po} \{ \ldots \} \]
Steps

• Frame bound inference
• Abstract State Inference
• Stack inference
• Heap Inference
Abstract State Inference

• this stage attempts to infer an abstract program state at every program point
• each abstract state $\Delta$ is expressed as a Presburger formula over values on the stack $[\pi_p, ..., \pi_1]$
• $\pi_i$ denotes the original value and $\pi_i'$ the latest value of the stack at location $i$
Abstract State Inference

the rules are of the following form:

$$\Delta \vdash_A A \leadsto B, \Delta_1$$

abstract state before the evaluation of A

abstract state after the evaluation of A

expression previously annotated with top frame pointers

output expression:

$$B ::= (p, \Delta, E_B)$$
$$E_B ::= \text{Cmd} \mid B; B \mid \text{if } B \mid \text{while } B \Delta_1$$
Abstract State Inference

\[
\begin{align*}
\text{[AS-LOAD]} \\
\Delta_1 &= \Delta \land \pi'_{p+1} = \pi'_i \\
\Delta \vdash_A (p, \text{load}(t) \ i) & \leadsto (p, \Delta, \text{load}(t) \ i), \Delta_1
\end{align*}
\]

- the value of the local variable at position \(i\) is copied to the top of the stack
- therefore the post-state is extended with an equation that relates the value on top of the stack to the value at position \(i\)
Abstract State Inference

\[
\begin{align*}
\text{[AS-STORE]} \\
\Delta_1 &= \Delta \circ \{\pi_i\} \quad \pi_i^\prime = \pi_p^\prime \\
\Delta \vdash_A (p, \text{store}(t) \ i) &\sim (p, \Delta, \text{store}(t) \ i), \exists \pi_p^\prime \cdot \Delta_1
\end{align*}
\]

- moves the value from the top of the stack to position \(i\)
- the composition \(\Delta \circ_{\pi_i} \pi_i^\prime = \pi_p^\prime\) removes all occurrences of \(\pi_i\) from \(\Delta\) and adds \(\pi_i^\prime = \pi_p^\prime\)
- \(\exists \pi_p^\prime \cdot \Delta_1\) removes the references to the value on top of the stack from \(\Delta_1\)
Abstract State Inference

\[\begin{align*}
\exists \pi'_p \cdot (\Delta \land \pi'_p = 1) & \vdash_A A_1 \leadsto B_1, \Delta_1 \\
\exists \pi'_p \cdot (\Delta \land \pi'_p = 0) & \vdash_A A_2 \leadsto B_2, \Delta_2
\end{align*}\]

\[\Delta \vdash_A (p, \text{if } A_1 A_2) \leadsto (p, \Delta, \text{if } B_1 B_2), \Delta_1 \lor \Delta_2\]

- add $\pi'_p = 1$ and $\pi'_p = 0$ to the abstract states before the evaluation of $A_1$ and $A_2$
- $\Delta_1 \lor \Delta_2$ combines the postconditions of both branches
Abstract State Inference

- for the invoke m instruction, first the precondition of m is checked
- the postcondition of m is added to the current abstract state
- the arguments are removed from the stack
- a constraint that relates the result of m to the new top of the stack is added
Abstract State Inference

- while loops are considered to be recursive functions
- the parameters are the values currently on the stack (they are assumed to be passed by reference)
- to approximate the effect of recursion, fixpoint analysis is applied for loops and method declarations
What we have achieved so far

Goal:
given a method
\[ tm(t_1, \ldots, t_n)l\{\ldots\} \]
infer the following extended declaration
\[ tm(t_1, \ldots, t_n)l; \phi_{pr}; F; \phi_{po}; S; H_{po}; M_{po}\{\ldots\} \]
Steps

- Frame bound inference
- Abstract State Inference
- Stack inference
- Heap Inference
Stack Inference

• to support interprocedural analysis, we must analyze how method invocations affect the global stack
• so far we have only considered the frames of single methods
• at the end of each method call there is always zero net stack usage, so we only need to infer the stack bound
the rules are of the following form:

\[
\text{arity of the current method} \quad \alpha \vdash S \quad B \sim S
\]

expression previously annotated with top frame pointers and abstract states

stack bound
Stack Inference

\[
\text{[SS-INVOKE]}
\begin{align*}
\tau m_1(t_1, \ldots, t_n) l; \phi_{pr}; \mathcal{F}; \phi_{po}; S \{ B \} & \in P \quad r = p - n + 2 \\
\rho = [\pi_i \mapsto \pi'_{p-n+i}]_{i=1}^n & \quad S_1 = \text{enrich}(a, \Delta, \rho S) + r \\
\hline
\end{align*}
\]

\[
a \vdash_S (p, \Delta, \text{invoke } m_1) \rightsquigarrow S_1
\]

- \( p-n+2 \): size of the current stack frame after the removal of the arguments
- \( \text{enrich}(a, \Delta, S) \) incorporates the path-sensitive guarded formula \( S \) into the current abstract state \( \Delta \)
Stack Inference

we can mark each tail call with a special \texttt{invoke\_Tail} instruction

- for each such invocation we can build the next stack frame by overwriting the current one

\[
\begin{align*}
&\left[\text{SS-Invoke-Tail}\right] \\
&m_1(t_1, \ldots, t_n) \phi_{pr}; F; \phi_{po}; S \{ \cdots \} \in P \\
&\rho = \left[\pi_i \mapsto \pi'_{p-n+i}\right]_{i=1}^{n} \quad S_1 = enrich(a, \Delta, \rho S) \\
&\vdash_S (p, \Delta, \text{invoke}_\text{Tail} m_1) \rightsquigarrow S_1
\end{align*}
\]
What we have achieved so far

Goal:
given a method
\[tm(t_1, \ldots, t_n)l\{\ldots\}\]
infer the following extended declaration
\[tm(t_1, \ldots, t_n)l; \phi_{pr}; F; \phi_{po}; S; H_{po}; M_{po}\{\ldots\}\]
Steps

• Frame bound inference
• Abstract State Inference
• Stack inference
• Heap Inference
Abstract State Inference

the rules are of the following form:

\[ a, \mathcal{H} \vdash_H B \sim \mathcal{H}_1, \mathcal{M} \]

- arity of the current method
- heap effect before execution of B
- heap effect after execution of B

expression previously annotated with top frame pointers

high watermark of heap usage during the execution of B
Heap Inference

\[
\begin{align*}
[\text{HS-SEQ}] \\
a, \mathcal{H} \vdash_H B_1 \leadsto \mathcal{H}_1, \mathcal{M}_1 \\
a, \mathcal{H}_1 \vdash_H B_2 \leadsto \mathcal{H}_2, \mathcal{M}_2
\end{align*}
\]

\[
a, \mathcal{H} \vdash_H (p, \Delta, B_1; B_2) \leadsto \mathcal{H}_2, \mathcal{M}_1 \cup \mathcal{M}_2
\]

- the heap usage of the sequence is the heap usage after the second instruction
- the heap bound is the upper bound of the two heap bounds
the heap usage is affected by new and dispose instructions:

\[
\begin{align*}
\text{[HS-NEW]} & \quad \mathcal{H}_1 = \mathcal{H} + \text{enrich}(a, \Delta, \{(c, 1)\}) \\
\text{a, } \mathcal{H} \vdash_H (p, \Delta, \text{new } c) & \rightsquigarrow \mathcal{H}_1, \mathcal{H}_1
\end{align*}
\]

\[
\begin{align*}
\text{[HS-DISPOSE]} & \quad \mathcal{H}_1 = \mathcal{H} + \text{enrich}(a, \Delta, \{(c, -1)\}) \\
\text{a, } \mathcal{H} \vdash_H (p, \Delta, \text{dispose } c) & \rightsquigarrow \mathcal{H}_1, \mathcal{H}
\end{align*}
\]

enrich\((a, \Delta, \mathcal{H})\) adds the current abstract state \(\Delta\) into the guards of a heap specification \(\mathcal{H}\).
What we have achieved

Goal:
given a method

$tm(t_1, ..., t_n)l\{...\}$

infer the following extended declaration

$tm(t_1, ..., t_n)l; \phi_{pr}; F; \phi_{po}; S; H_{po}; M_{po}\{...\}$
• a prototype has been tested with a set of small programs
• all stack bounds were successfully captured except for the Ackermann function
• the time for abstract state inference roughly correlated with the program size and with the complexity of the relations between program variables
• the additional time for stack inference was significant (the test programs used the stack intensively)
• time for heap inference was less substantial
Conclusions

a sound inference system was developed that

• can infer net usage and upper bound of stack and heap space
• uses a special guarded expression form
• can handle recursion and loops
Questions?