Extending L2CA for the verification of multi-threaded programs

Polyvios Pratikakis

Radu Iosif

Multi-threaded programming

- Multi-core CPUs are increasingly available
- Demand for parallel software increasing
- Threads and shared memory is the dominant programming model
- Complex to program
- Easy to make mistakes
- Difficult to debug
- Increased need for automatic verification

L2CA: verification of list-programs

- Analyze simple programs
- Precise analysis of the heap
- Summarize singly-linked lists with n elements: finite heap state-space
- Compile to counter-automata
- Check memory safety
- Sound, complete

L2CA in a nutshell

- Simple abstract language:
  - Variables have only integer and list types
  - Statements are assignments, loops, conditionals
- Forward data flow analysis
- Heap is list segments
- Computes a set of heaps per program point
- Automaton states are (heap, program-point) pairs
- Abstract counters used to summarize the length of lists in the heap

L2CA: an example

```
count(struct list *i) {
  int a;
  struct list *j;

  j = i;
  a = 0;
  while (j != NULL) {
    a++;
    j = j→next;
  }
}
```

Formalising sequential L2CA

$$
\begin{array}{llcl}
\text{Locations} & \rho & \in & \mathcal{R} \\
\text{Values} & v & ::= & n \mid \rho \mid \mathsf{true} \mid \mathsf{false} \\
\text{Expressions} & e & ::= & x \mid \mathsf{let}\ x = e\ \mathsf{in}\ e \mid \mathsf{if}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e \mid e := e \\
 & & \mid & \mathsf{while}\ e\ \mathsf{do}\ e \mid e; e \mid \mathsf{null} \mid \mathsf{new} \mid e.\mathsf{next} \\
 & & \mid & e = e \mid e \leq e \mid \neg e \mid e \wedge e \mid e \vee e \mid e \pm e \\
\text{Types} & \tau & ::= & \mathit{int} \mid \mathit{list} \mid \mathit{bool} \\
\text{Heaps} & H & ::= & \cdot \mid H, (\rho, \rho) \mid H, (\rho, \bot) \\
\text{Counters} & c & \in & \mathcal{C} \\
\text{Abstract Heaps} & \bar{H} & ::= & \cdot \mid \bar{H}, (\rho, c, \rho) \mid \bar{H}, (\rho, c, \bot) \\
\text{Execution state} & s & ::= & \langle H, e \rangle \\
\text{Automaton state} & \bar{s} & ::= & \langle \bar{H}, e \rangle
\end{array}
$$

Operational semantics

- Defined concrete semantics, e.g.:

$$\textsc{Assign-Null}\ \frac{\rho \in dom(H)}{\langle H, \rho := \mathsf{null} \rangle \longrightarrow \langle H[\rho \mapsto \bot], \mathsf{null} \rangle}$$

- State space is infinite
- Use abstract heaps to summarize:
  - Abstract heaps represent sets of concrete heaps
  - Abstract heap space is finite, for finite programs
  - Possible to explore exhaustively

Create the automaton

- For every concrete operational semantics transition, define corresponding abstract-state transition(s):

$$\frac{\bar{H}(\rho) = (c, \alpha)}{\langle \bar{H}, \rho := \mathsf{null} \rangle \xrightarrow{c=0} \langle \bar{H}[\rho \mapsto (c, \bot)], \mathsf{null} \rangle}$$

$$\frac{\bar{H}(\rho) = (c, \alpha)}{\langle \bar{H}, \rho := \mathsf{null} \rangle \xrightarrow{c>0} \langle \textbf{Memory leak} \rangle}$$

- Perform fixpoint analysis on program code to discover all states and transitions of the automaton

L2CA status

- Limitations
  - Works only for a simple, abstract, imperative language
  - Intra-procedural, no support for function calls
  - Sinlge-threaded, sequential code
- Goal: extensions
  - Extend to full C
  - Support function calls, intra-procedural analysis
  - Analyze multi-threaded programs with locks

Extending to C

- Rewrote in OCaml using CIL, accepts subset of C
- Can only handle simple C programs with lists: discovers recursive types with one recursive field, ignores the rest
- Currently rejects all type-unsafety in input C programs

Adding parallelism

- Constant number of parallel threads
- Common initial heap, global variables
- Lock types, lock/unlock statements
- Lock state in the heap: every heap cell can be *owned* by a thread
- Adjust operational semantics for non-deterministic thread interleavings:
  - Any thread can take a step, unless it is waiting for a lock
- Create an automaton state for every possible heap, at every reachable combination of thread program points

Parallelism, formally

Extensions

$$
\begin{array}{llll}
\text{Expressions} & e & ::= & \ldots \mid \mathsf{lock}\ e \mid \mathsf{unlock}\ e \\
\text{Heaps} & H & ::= & \cdot \mid H, (\rho, t_{id}, \rho) \mid H, (\rho, t_{id}, \bot) \\
\text{Abstract Heaps} & \overline{H} & ::= & \cdot \mid \overline{H}, (\rho, t_{id}, c, \rho) \mid \overline{H}, (\rho, t_{id}, c, \bot) \\
\text{Execution state} & s & ::= & \langle H, \vec{e} \rangle \\
\text{Automaton state} & \overline{s} & ::= & \langle \overline{H}, \vec{e} \rangle
\end{array}
$$

Operational semantics: each thread can take a step

$$
\frac{\langle H, e_i \rangle \longrightarrow^{i} \langle H', e'_i \rangle}{\langle H, \vec{e} \rangle \longrightarrow \langle H', \vec{e}[e'_i / e_i] \rangle}
$$

State Explosion!

Solution attempts

- Partial order reduction:
  - Not all interleavings give different behaviors
  - Some interleavings are equivalent to others (e.g. access to thread-local data)
  - Idea: Merge interleavings that are equivalent
  - Effect: less interleavings to worry about, less states
  - Threads execute bigger "chunks" of code between context-switches
- Context limit
  - Idea: limit the number of context switches allowed per thread
  - Pros: bigger pieces of serial code, less states
  - Cons: Might mask some errors that manifestate with more context-switches (sacrifices soundness)

Adding function support

- Simple algorithm:
  1. At every function call, partition the heap (frame rule)
     - Only keep what is relevant (reachable from globals and parameters)
  2. Analyze function body using the relevant part of the heap
  3. Memoize the caller heap, and exit states of the automaton of the called function
  4. At subsequent calls, reuse function automaton if heap matches, otherwise re-analyze using new "relevant" heap part
- Drawbacks:
  - Does not handle recursion
  - Worst-case time is equivalent to simply inlining

Function example

```
void append(struct list *l, struct list *l2) {
  while(l→next != NULL) l = l→next;
  l→next = l2;
}

main() {
  struct list * c1 = malloc();
  struct list * c2 = malloc();
  struct list * c3 = malloc();
  append(c1, c2);
  append(c1, c3);
}
```

Open problem: recursive functions

Current insights

- Tail recursion is easy, equivalent to simple loops
- Non-tail recursion is equivalent to stack-counter automata in the general case: We cannot easily answer reachability questions
- More general solution: summarize and overapproximate
  - Loses completeness, might produce false warnings

Summary

- L2CA analyzes programs that manipulate lists, using counter automata
- Adding parallelism causes state explosion
  - Reduced somewhat by partial order reduction
  - Context bound analysis gives further improvement, but loses soundness
- Adding function support
  - Enables inter-procedural analysis, bigger programs
  - Summary-based analysis of functions scales better
  - Does not handle recursion without loss of completeness
  - Is expensive, amounts to inlining at worst case