Matching Logic
- A New Program Verification Approach -

Grigore Rosu and Andrei Stefanescu
University of Illinois at Urbana-Champaign
... could it be that, after 40 years of program verification, we still lack the right semantically grounded program verification foundation?

Floyd-Hoare logic

\[ \{ \pi_{\text{pre}} \} \text{ code} \{ \pi_{\text{post}} \} \]
Question

... could it be that, after 40 years of program verification, we still lack the right semantically grounded program verification foundation?

Floyd-Hoare logic

\[ \{ \pi_{\text{pre}} \} \text{ code} \{ \pi_{\text{post}} \} \]
Limitations of Floyd-Hoare Logic

• Requires encodings of structural program configuration properties as predicates
  – Heap, stacks, input/output, etc.
  – Framing is hard to deal with

• Not based on a formal executable semantics
  – Thus, hard to test
  – Semantic errors found by proving wrong properties
  – Soundness rarely or never proved in practice

• Implementations of Floyd-Hoare verifiers for real languages still an art, who few master
Ideal Program Logic

• Based on a formal *executable* semantics
  – So we can test it by executing 1000’s of programs
  – Sound “by construction”

• Allows us to state any structural properties about configurations
  – Heap, stacks, input/output, etc.
  – Framing would be straightforward; nothing special

• Leads to immediate implementations of program verifiers, based on the executable semantics
Matching Logic

• A logic for reasoning about configurations
• Builds upon executable/operational semantics
  – Provides ground configurations and transitions
• Matching Logic
  – Formulae / Specifications
    • FOL over configurations with variables, called patterns
  – Models
    • Ground configurations
  – Satisfaction
    • Matching for configurations, plus FOL for the rest
Formal Executable Semantics of KernelC
Formal Executable Semantics of KernelC

Syntax declared using annotated BNF

\[ \text{Exp} ::= \begin{cases} \text{Exp} = \text{Exp} \quad [\text{strict}(2)] \\ \text{Exp} \end{cases} \]
Configuration given as a nested cell structure. Leaves can be sets, multisets, lists, maps, or syntax.
Formal Executable Semantics of KernelC

Semantic rules given contextually
Formal Executable Semantics of KernelC

K Framework

http://k-framework.org

Highlights:
• Modular, scales well (C, Scheme, Verilog, …)
• Easy to use: user by undergrads
• Multiple uses: interpreters, model checkers, …
Examples of Patterns

• x points to sequence A with $|A| > 1$, and the reversed sequence $\text{rev}(A)$ has been output

• \textit{untrusted()} can only be called from \textit{trusted()}

\[ \text{env} \quad \text{x} \rightarrow a \quad \text{mem} \quad \text{list}(a,A) \quad \text{out} \quad \text{rev}(A) \quad \land \quad |A| > 1 \]
Does it Really Work?

- We implemented a proof-of-concept matching logic verifier for a fragment of C
- Could verify all properties currently verifiable with existing separation logic based verifiers (for C-like languages)
- Could also verify properties that cannot be expressed in separation logic
- See Matching Logic poster tomorrow
  - Demo possible
void readWriteBuffer(int n)
{
    int i;
    struct ListNode *x;

    i = 0;
    x = 0;

    while (i < n) {
        struct ListNode *y;

        y = x;
        x = (struct ListNode*) malloc(sizeof(struct ListNode));
        scanf("%d", &(x->val));
        x->next = y;
        i += 1;
    }

    while (x) {
        struct ListNode *y;

        y = x->next;
        printf("%d ", x->val);
        free(x);
        x = y;
    }
}
MatchC at Work – Heap and I/O

MatchC will simply run the semantics.

```c
void readWriteBuffer(int n)
{
    int i;
    struct ListNode *x;
    i = 0;
    x = 0;

    while (i < n) {
        struct ListNode *y;

        y = x;
        x = (struct ListNode*) malloc(sizeof(struct ListNode));
        scanf("%d", &x->val);
        x->next = y;
        i += 1;
    }

    while (x) {
        struct ListNode *y;

        y = x->next;
        printf("%d ",x->val);
        free(x);
        x = y;
    }
}
```
MatchC at Work – Heap and I/O

```c
void readWriteBuffer(int n)
{
    int i;
    struct ListNode *x;
    i = 0;
    x = 0;

    while (i < n) {
        struct ListNode *y;

        y = x;
        x = (struct ListNode*) malloc(sizeof(struct ListNode));
        scanf("%d", &(x->val));
        x->next = y;
        i += 1;
    }

    while (x) {
        struct ListNode *y;

        y = x->next;
        printf("%d ", x->val);
        free(x);
        x = y;
    }
}
```

-bash-3.2$ ..../matchC io3.c
Compiling program ... DONE! [0.261s]
Loading Maude ........ DONE! [0.291s]
Verifying program ... DONE! [0.035s]
Verification succeeded! [10184 rewrites, 1 feasible and 5 infeasible paths]
Output: 5 4 3 2 1
bash-3.2$
void readWriteBuffer(int n)
{
    int i;
    struct ListNode *x;

    i = 0;
    x = 0;

    while (i < n) {
        struct ListNode *y;

        y = x;
        x = (struct ListNode*) malloc(sizeof(struct ListNode));
        scanf("%d", &(x->val));
        x->next = y;
        i += 1;
    }

    while (x) {
        struct ListNode *y;

        y = x->next;
        printf("%d ", x->val);
        free(x);
        x = y;
    }
}
void readWriteBuffer(int n)
{ /*@ rule $\langle k \rangle$ $\Rightarrow$ return; $\langle / k \rangle$ $\langle \text{in} \rangle$ $A \Rightarrow \varepsilon$ $\langle / \text{in} \rangle$ $\langle \text{out} \rangle$ $\varepsilon \Rightarrow \text{rev}(A)$ $\langle / \text{out} \rangle$*/
    int i;
    struct ListNode *x;

    i = 0;
    x = 0;
    /*@ inv $\langle \text{in} \rangle$ $?B$ $\langle / \text{in} \rangle$ $\langle \text{heap} \rangle$ list(x)(?A) $\langle / \text{heap} \rangle$
        \( / \langle i <= n \rangle \land \text{len(?B)} = n - i \land A = \text{rev(?A)} \land ?B */
    while (i < n) {
        struct ListNode *y;

        y = x;
        x = (struct ListNode*) malloc(sizeof(struct ListNode));
        scanf("%d", &(x->val));
        x->next = y;
        i += 1;
    }
    /*@ inv $\langle \text{out} \rangle$ $?A$ $\langle / \text{out} \rangle$ $\langle \text{heap} \rangle$ list(x)(?B) $\langle / \text{heap} \rangle \land A = \text{rev(?A)} \land ?B */
    while (x) {
        struct ListNode *y;

        y = x->next;
        printf("%d ", x->val);
        free(x);
        x = y;
    }
}
MatchC at Work – Heap and I/O

```c
void readWriteBuffer(int n)
/**< rule << $ => return; </> << A => epsilon </> << out_> epsilon => rev(A) </> out_>
  if n = len(A) */
{
  int i;
  struct ListNode *x;

  i = 0;
  x = 0;
/**< inv << ?B << heap > list(x)(?A) </> heap>
  \( i <= n \) \( \text{len}(?B) = n - i \) \( A = \text{rev}(?A) @ ?B \) */
  while (i < n) {
    struct ListNode *y;

    y = x;
    x = (struct ListNode*) malloc(sizeof(struct ListNode));
    scanf("%d", &x->val);
    x->next = y;
    i += 1;
  }
/**< inv << out_> ?A </> out_> heap > list(x)(?B) </> heap \( A = \text{rev}(?A @ ?B) \)
  while (x) {
    struct ListNode *y;

    y = x->next;
    printf("%d ",x->val);
    free(x);
    x = y;
  }

  //uw...--Fl io3.c
  Wrote /home/grosu/grosu/matchC/io3.c
```
Conclusion

• Hoare Logic may not be the ultimate answer to the problem of program verification!
• In Matching Logic, we use an executable semantics of a language as is for verification
  – As opposed to redefining it as a Hoare logic
  – Executable semantics is testable and reusable
  – Giving an executable semantics is not necessarily painful, it can be fun if one uses the right tools