Matching Logic
- A New Program Verification Approach -

Grigore Rosu and Andrei Stefanescu
University of Illinois at Urbana-Champaign
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}} \}
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
Syntax declared using annotated BNF

\[
\text{Exp ::= } \begin{cases} 
\text{Exp} 
\end{cases}
\]

Formal Executable Semantics of KernelC
Configuration given as a nested cell structure. Leaves can be sets, multisets, lists, maps, or syntax.
Formal Executable Semantics of KernelC
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

• `untrusted()` can only be called from `trusted()`
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
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;
    }
}
```
MatchC at Work – Heap and I/O

Reads integers from standard input and stores them in a list in the heap; then it deallocates the list, printing its elements to the standard output.

Since nothing is requested, MatchC will simply run the semantics.
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! [0.201s]
Verifying program ... DONE! [0.035s]

Verification succeeded! [10184 rewrites, 1 feasible and 5 infeasible paths]
Output: 5 4 3 2 1
bash-3.2$
```
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;
    }
}
```

Wrote /home/grosu/grosu/matching-logic/matchC/demo/2-io/io3.c
MatchC at Work – Heap and I/O

```c
void readWriteBuffer(int n)
/*@ rule <k> $ => return; </k> <in> A => epsilon </in> <out> epsilon => rev(A) </out>
  if n = len(A) */
{
  int i;
  struct ListNode *x;
  i = 0;
  x = 0;
  /*@ inv <in> ?B </in> <heap> list(x)(?A) </heap>
   \i <= n \ /\ len(?B) = n - i \ /\ A = 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 = rev(?A @ ?B)
  while (x) {
    struct ListNode *y;
    y = x->next;
    printf("%d ", x->val);
    free(x);
    x = y;
  }
  */
```

Wrote /home/grosu/grosu/matching-logic/matchC/demo/2-io/io3.c
MatchC at Work – Heap and I/O

```c
void readWriteBuffer(int n)
{  /* rule <= $ => return; <= A => epsilon <= <in> <out> epsilon => rev(A) <= */
    int i;
    struct ListNode *x;
    i = 0;
    x = 0;
    /*@ inv <in> ?B <= <in> <heap> list(x)(?A) <= <heap> */
    /*@ inv <out> ?A <= <out> <heap> list(x)(?B) <= <heap> */
    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
bash-3.2$ ../../../matchC io3.c
Compiling program ... DONE! [0.312s]
Loading Maude ........ DONE! [0.210s]
Verifying program ... DONE! [0.095s]
Verification succeeded! [79897 rewrites, 4 feasible and 2 infeasible paths]
Output: 5 4 3 2 1
bash-3.2$
```

Wrote /home/grosu/grosu/matchC/io3.087
```
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