K and Matching Logic

Grigore Rosu
University of Illinois at Urbana-Champaign

Joint work with the FSL group at UIUC (USA) and the FMSE group at UAIC (Romania)
… could it be that, after 40 years of program verification, we still lack the right semantically grounded program verification foundation?

Hoare logic

\[ \{ \pi_{pre} \} \text{ code } \{ \pi_{post} \} \]
Current State-of-the-Art in Program Analysis and Verification

Consider some programming language, L

- Formal semantics of L
  - Typically skipped: considered expensive and useless
- Model checkers for L
  - Based on some adhoc encodings/models of L
- Program verifiers for L
  - Based on some other adhoc encodings/models of L
- Runtime verifiers for L
  - Based on yet another adhoc encodings/models of L
- ...
Example of C Program

• What should the following program evaluate to?

```c
int main(void) {
    int x = 0;
    return (x = 1) + (x = 2);
}
```

• According to the C “standard”, it is **undefined**

• GCC4, MSVC: it returns **4**
  GCC3, ICC, Clang: it returns **3**

By April 2011, both Frama-C (with its Jessie verification plugin) and Havoc "prove" it returns **4**
A Formal Semantics Manifesto

• Programming languages must have formal semantics! (period)
  – And analysis/verification tools should build on them

• Informal manuals are not sufficient
  – Manuals typically have a formal syntax of the language (in an appendix)
  – Why not a formal semantics appendix as well?
Motivation and Goal

• We are facing a semantic chaos
  – Operational, denotational, axiomatic, etc.
  – Problematic when dealing with large languages

• Why so many semantic styles?
  – Since none of them is ideal, they have limitations

• We want a powerful, unified foundation for programming language semantics and verification
  – One semantics to serve all the purposes!
Minimal Requirements for an Ideal Language Semantic Framework

• Should be **expressive**
  – Substitution or environment-based definitions, abrupt control changes (callcc), concurrency, etc.

• Should be **executable**
  – So we can test it and use it in tools (symb. exec.)

• Should be **modular/compositional** (thus scale)
  – So each feature is defined once and for all

• Should serve as a **program logic**
  – So we can also prove programs correct with it
Current Semantic Approaches
-Structural Operational Semantics-

- Executable ... in principle
  - Norish’s C semantics is not executable
- Not very modular/compositional (except MSOS)
- Not appropriate for verification
- Only interleaving semantics
Current Semantic Approaches
-Evaluation Contexts-

• Executable
• Modular, deals better with control
• Very syntactic, rigid
  – Enforces substitution-based definitions
  – Unsuitable for environment-based definitions
• Not appropriate for verification
• Only interleaving semantics
Current Semantic Approaches
-Denotational Semantics-

Reasonable trade-offs

• Mathematical model, somehow compositional
• Not very executable
  – factorial(5) crashes Papaspyrou’s C semantics
• Not very good for verification
• Poor for concurrency
• Requires expert knowledge
Current Semantic Approaches
-The Chemical Abstract Machine Machine-

CHAM

- Intuitive computational model
- True concurrency (like in nature)
- No machine support
  - It would be very hard, because of its airlock
- Not appropriate for verification
Current Semantic Approaches -Floyd-Hoare Logic-

- Good for program verification
- 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
Towards a Better Semantic Approach
Starting Point: Rewriting Logic

Meseguer (late 80s, early 90s)

• Expressive
  – Any logic can be represented in RL (it is reflective)

• Executable
  – Quite efficiently; Maude often outperforms SML

• Modular
  – Allows rules to only “match” what they need

• Can potentially serve as a program logic
  – Admits initial model semantics, so it is amenable for inductive or fixed-point proofs
Rewriting Logic Semantics Project

• Project started jointly with Meseguer in 2003-4
• Idea: Define the semantics of a programming language as a rewrite theory (set of rules)
• Showed that most executable semantics approaches can be framed as rewrite logic semantics (Modular/SmallStep/BigStep SOS, evaluation contexts, continuation-based, etc.)
  – But they still had their inherent limitations
• Appropriate techniques/methodologies needed
The K Framework

**k-framework.org**

- A tool-supported rewrite-based framework for defining programming language semantics
- Inspired from rewriting logic
- Used regularly in teaching
- Main ideas:
  - Represent program configurations as a potentially nested structure of cells (like in the CHAM)
  - Flatten syntax into special computational structures (like in refocusing for evaluation contexts)
  - Define the semantics of each language construct by semantic rules (a small number, typically 1 or 2)
Since then it has been expanded and used for expressing and verifying concurrency features and anomalies for both sequentially...
Complete K Definition of KernelC

Configuration given as a nested cell structure. Leaves can be sets, multisets, lists, maps, or syntax
Complete K Definition of KernelC

\[
\begin{align*}
\text{X} &= \text{V} \\
\text{\langle k \rangle} \quad \text{X} &= \text{V} \Rightarrow \text{V} \\
\text{\langle env \rangle} \quad \text{X} |\rightarrow (\_ \Rightarrow \text{V}) \\
\end{align*}
\]
K Semantics are Useful

• Executable, help language designers
• Make teaching PL concepts hands-on and fun
• Currently compiled into
  – Maude, for execution, debugging, model checking
  – Latex, for human inspection and understanding
• Plans to be compiled to
  – OCAML, for fast execution
  – COQ, for meta-property verification
Medium-Size K Definition

• See the semantics of SIMPLE on the “K and Matching Logic” page
K Scales

Besides smaller and paradigmatic teaching languages, several larger languages were defined

- Scheme : by Pat Meredith
- Java 1.4 : by Feng Chen
- Verilog : by Pat Meredith and Mike Katelman
- C : by Chucky Ellison

etc.
The K Configuration of C

Heap

75 Cells!
Statistics for the C definition

- Total number of rules: ~1200

- Has been tested on thousands of C programs (several benchmarks, including the gcc torture test, code from the obfuscated C competition, etc.)
  - Passed 99.2% so far!
  - GCC 4.1.2 passes 99%, ICC 99.4%, Clang 98.3 (no opt.)

- The most complete formal C semantics

- Took more than 18 months to define ...
  - Wouldn’t it be uneconomical to redefine it in each tool?
Matching Logic Verification

= 

Rewriting (language semantics)

+ 

FOL (configuration reasoning)

+ 

Proof Rules (behavior reasoning)
Matching Logic

• A logic for reasoning about configurations

• Formulae
  – FOL over configurations, called patterns
  – Configurations are allowed to contain variables

• Models
  – Ground configurations

• Satisfaction
  – Matching for configurations, plus FOL for the rest
Examples of Patterns

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

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

---

```
RULE run(L) ↭ run(L, •) ↭ main()
```

---

```
RULE • threads S out T ↭ S result
```

---

```
SYNTAX Bag ::= run(KLabel) | run(KLabel, List{K})
```

---

```
SYNTAX MapItem ::= list(Id, List)
```

---

```
SYNTAX Id ::= trusted | untrusted | x
```

---

```
RULE x ↭ a
```

---

```
RULE env list(a, A) mem rev(A) ∧ |A| > 1
```

---

```
RULE untrusted() k trusted() fstack
```

---

```
MODULE KERNELC IMPORTS K, SHARED IMPORTS KERNELC, CONSISTENT, THREADS + KERNELC, PROGRAMS + KERNELC, SIMPLE, MALLOC
```

---

```
Putting everything together
Running a program identified by its name. maybe with an input list
```

---

```
Bag ::= run(KLabel) | run(KLabel, List{K})
```

---

```
RULE run(L) ↭ run(L, •) ↭ main()
```

---

```
RULE • threads S out T ↭ S result
```

---

```
SYNTAX Bag ::= run(KLabel) | run(KLabel, List{K})
```

---

```
SYNTAX MapItem ::= list(Id, List)
```

---

```
SYNTAX Id ::= trusted | untrusted | x
```

---

```
RULE x ↭ a
```

---

```
RULE untrusted() k trusted() fstack
```

---

```
MODULE KERNELC IMPORTS K, SHARED IMPORTS KERNELC, CONSISTENT, THREADS + KERNELC, PROGRAMS + KERNELC, SIMPLE, MALLOC
```
More Formally: Configurations

• For concreteness, assume configurations having the following syntax:

\[
\langle\langle\ldots\rangle_k \langle\ldots\rangle_{env} \langle\ldots\rangle_{heap} \langle\ldots\rangle_{in} \langle\ldots\rangle_{out} \ldots\rangle_{cfg}
\]

(matching logic works with any configurations)

• Examples of concrete (ground) configurations:

\[
\langle\langle x=^*y; y=x; \ldots \rangle_k \langle x \mapsto 7, y \mapsto 3, \ldots \rangle_{env} \langle 3 \mapsto 5 \rangle_{heap} \ldots \rangle_{cfg}
\]
\[
\langle\langle x \mapsto 3 \rangle_{env} \langle 3 \mapsto 5, 2 \mapsto 7 \rangle_{heap} \langle 1, 2, 3, \ldots \rangle_{in} \langle\ldots, 7, 8, 9 \rangle_{out} \ldots \rangle_{cfg}
\]
More Formally: Patterns

- Concrete configurations are already patterns, but very simple ones, ground
- Example of more complex pattern

$$\exists c : \text{Cells}, \ e : \text{Env}, \ p : \text{Nat}, \ i : \text{Int}, \ \sigma : \text{Heap}$$
$$\langle \langle x \mapsto p, \ e \rangle_{\text{env}} \langle p \mapsto i, \ \sigma \rangle_{\text{heap}} \rangle_{\text{cfg}} \land i > 0 \land p \not= i$$

- Thus, patterns generalize both terms and FOL
- Models: concrete configurations + valuations
- Satisfaction: matching for patterns, FOL for rest
More Formally: Reasoning

- We can now prove (using FOL reasoning) properties about configurations, such as

\[
\forall c: \text{Cell}, \; e: \text{Env}, \; p: \text{Nat} \\
\langle \langle x \mapsto p, \; e \rangle_{\text{env}} \; \langle p \mapsto 9 \rangle_{\text{heap}} \; c \rangle_{\text{cfg}} \land p > 10 \\
\rightarrow \exists i: \text{Int}, \; \sigma: \text{Heap} \\
\langle \langle x \mapsto p, \; e \rangle_{\text{env}} \; \langle p \mapsto i, \; \sigma \rangle_{\text{heap}} \; c \rangle_{\text{cfg}} \land i > 0 \land p \neq i
\]
Matching Logic vs. Separation Logic

• Matching logic achieves separation through matching at the structural (term) level, not through special logical connectives (*)

• Matching logic realizes separation at all levels of the configuration, not only in the heap
  – the heap was only 1 out of the 75 cells in C’s def.

• Matching logic can stay within FOL, while separation logic needs to extend FOL
  – Thus, we can use the existing SMT provers, etc.
Matching Logic as a Program Logic

• Hoare style - not recommended
  
  \{ \pi_{pre} \} \text{code} \{ \pi_{post} \}

  – One has to redefine the PL semantics – impractical

• Rewriting (or K) style – recommended
  
  \text{left}[\text{code}] \rightarrow \text{right}

  – One can reuse existing K semantics – very good
Example – Swapping Values

- What is the K semantics of the swap function?
- Let $ be its body

```c
void swap(int *x, int *y)
{
    int t;
    t=*x;
    *x=*y;
    *y=t;
}
```

**Rule**

- If $ x = y $

**Heap**

- $ x \mapsto a $, $ y \mapsto b $,
  
- $ x \mapsto b $, $ y \mapsto a $,

**Rule**

- If $ x = y $
Example – Reversing a list

```c
struct listNode* reverse(struct listNode *x)
{
    struct listNode *p;
    struct listNode *y;
    p = 0 ;
    while(x) {
        y = x->next;
        x->next = p;
        p = x;
        x = y;
    }
    return p;
}
```

- What is the K semantics of the reverse function?
- Let $ be its body

$\begin{array}{l}
\text{rule } \langle k \rangle \; \langle $ \rangle \; \Rightarrow \; \text{return } \; \langle p \rangle ; \\
\langle \text{heap} \rangle \ldots \; \text{list}(x,A) \; \Rightarrow \; \text{list}(p,\text{rev}(A)) \; \\
\end{array}$
Partial Correctness

• We have two rewrite relations on configurations
  \[ \rightarrow \] given by the language K semantics; \textbf{safe}
  \[ \rightarrow \] given by specifications; \textbf{unsafe}, has to be proved

• Idea (simplified for deterministic languages):
  – Pick \textbf{left} \[ \rightarrow \] \textbf{right}. Show that always \textbf{left} \[ \rightarrow \] \((\rightarrow \cup \rightarrow)^* \textbf{right}
    modulo matching logic reasoning (between rewrite steps)

• Theorem (soundness):
  – If \textbf{left} \[ \rightarrow \] \textbf{right} and “\textit{config matches left}” such that \textit{config}
    has a normal form for \[ \rightarrow \], then “\textit{nf(config) matches right}”
More Formally:
Matching Logic Rewriting

• Matching logic rewrite rules are rewrite rules over matching logic formulae: \( \varphi \Rightarrow \varphi' \)

• Since patterns generalize terms, matching logic rewriting captures term rewriting

• Moreover, deals naturally with side conditions: rewrite rules of the form

\[ l \Rightarrow r \text{ if } b \]

are captured as matching logic rules of the form

\[ l \land b \Rightarrow r \]
More Formally: Proof System I

- Rules of operational nature

**Reflexivity:**

\[ \overrightarrow{\mathcal{A}} \vdash \varphi \Rightarrow \varphi \]

\[ \theta : \text{Var} \rightarrow T_{\Sigma}(\text{Var}) \]

\[ \varphi \Rightarrow \varphi' \text{ if } \varphi_1 \Rightarrow \varphi'_1, \ldots, \varphi_n \Rightarrow \varphi'_n \in \mathcal{A} \]

**Replacement:**

\[ \mathcal{A} \vdash \{ \theta(\varphi_1) \Rightarrow \theta(\varphi'_1), \ldots, \theta(\varphi_n) \Rightarrow \theta(\varphi'_n) \} \]

\[ \mathcal{A} \vdash \theta(\varphi) \Rightarrow \theta(\varphi') \]

**Transitivity:**

\[ \mathcal{A} \vdash \{ \varphi_1 \Rightarrow \varphi_2, \varphi_2 \Rightarrow \varphi_3 \} \]

\[ \mathcal{A} \vdash \varphi_1 \Rightarrow \varphi_3 \]
More Formally: Proof System II

- Rules of deductive nature

**Consequence:**

\[
\begin{align*}
\models \varphi_1 \rightarrow \varphi'_1 \\
\models \varphi'_2 \rightarrow \varphi_2 \\
\mathcal{A} \vdash \varphi'_1 \Rightarrow \varphi'_2
\end{align*}
\]

\[
\mathcal{A} \vdash \varphi_1 \Rightarrow \varphi_2
\]

**Logic frame:**

\[
\begin{align*}
\psi \text{ is a FOL}_- \text{ formula} \\
\mathcal{A} \vdash \varphi \Rightarrow \varphi' \\
\mathcal{A} \vdash \varphi \land \psi \Rightarrow \varphi' \land \psi
\end{align*}
\]
More Formally: Proof System IV

- Main proof rule of matching logic rewriting

\[
\text{Circularity:}
\]

\[
C \text{ is the set } \{ \varphi_1 \Rightarrow \varphi'_1, \ldots, \varphi_n \Rightarrow \varphi'_n \}
\]

\[
\mathcal{A} \vdash \{ \varphi_1 \Rightarrow^+ \varphi''_1, \ldots, \varphi_n \Rightarrow^+ \varphi''_n \}
\]

\[
\mathcal{A} \cup C \vdash \{ \varphi''_1 \Rightarrow \varphi'_1, \ldots, \varphi''_n \Rightarrow \varphi'_n \}
\]

\[
\mathcal{A} \vdash C
\]
Fact

• Matching logic generalizes both operational semantics and axiomatic semantics
  – Operational semantics by means of capturing term rewriting as discussed above
  – Axiomatic semantics by noticing that Hoare triples are particular pattern rewrites:

\[
\text{HL2ML} (\{\psi\} s \{\psi'\}) = \\
\langle s, \sigma_Z \rangle \land \sigma_Z (\psi) \Rightarrow \exists Z (\langle \text{skip}, \sigma_Z \rangle \land \sigma_Z (\psi'))
\]
Theorem

• Any operational behavior can also be derived using matching logic reasoning

• For any Hoare triple \( \{\psi\} s \{\psi'\} \) derived with axiomatic semantics, the corresponding matching logic rule \( \text{HL2ML}(\{\psi\} s \{\psi'\}) \) can be derived with the matching logic proof system

• Partial correctness
MatchC Tool DEMO

try it online first at
http://fsl.cs.uiuc.edu/index.php/Special:MLOnline
Semantic Execution

John Regehr and his team included the K semantics of C as part of his CSMITH tool chain, to make sure that the generated C programs are defined.
```c
#include <stdlib.h>
#include <stdio.h>

int sum(int n) {
    int s;
    s = 0;
    while (n > 0) {
        s += n;
        n -= 1;
    }
    return s;
}

int main() {
    int s;
    s = sum(10);
    printf("The sum for the first 10 natural numbers: \d\n", s);
    return 0;
}
```

```
bash-3.2$
bash-3.2$ time gcc sum2.c ; a.out

real    0m0.042s
user    0m0.027s
sys     0m0.015s
The sum for the first 10 natural numbers: 55
bash-3.2$
bash-3.2$
bash-3.2$
bash-3.2$ matchC sum2.c
Compiling program ... DONE! [0.258s]
Loading Maude ........ DONE! [0.576s]
Verifying program ... DONE! [0.013s]
Verification succeeded! [2965 rewrites, 1 feasible and 0 infeasible paths]
Output: 55
bash-3.2$
bash-3.2$
```

```
-uu::-Fl sum2.c (C Abbrev)--L1--All---------------
```
Full Verification

```c
#include <stdlib.h>
#include <stdio.h>

int sum(int n) {
    int s;
    s = 0;
    // @ rule <k> $ => return (n * (n + 1)) / 2; </k> if n >= 0
    while (n > 0) {
        s += n;
        n -= 1;
    }
    return s;
}

int main() {
    int s;
    s = sum(10);
    printf("The sum for the first 10 natural numbers: %d\n", s);
    // @ assert <out> [55]
    return 0;
}
```

```
bash-3.2$
bash-3.2$ time gcc sum3.c ; a.out
real 0m0.042s
user 0m0.023s
sys 0m0.018s
The sum for the first 10 natural numbers: 55
bash-3.2$
bash-3.2$
bash-3.2$
bash-3.2$ matchC sum3.c
```

```
Compiling program ... DONE! [0.260s]
Loading Maude ...... DONE! [0.584s]
Verifying program ... DONE! [0.046s]
Verification succeeded! [33083 rewrites, 3 feasible and 0 infeasible paths]
Output: 55
bash-3.2$
bash-3.2$
```
List Examples – Borrowed from SL tools

```c
struct listNode;
/* rule <k> */
    <heap>
{
    struct listNode

    p = 0;
    //@ inv <heap>
    while(x) {
        struct listNode
        y = x->next;
        x->next = p;
        p = x;
        x = y;
    }
    return p;
}
```

```bash
bash-3.2$
bash-3.2$ time gcc list3.c ; a.out
real 0m0.045s
user 0m0.025s
sys 0m0.020s
x: 1 2 3 4 5
reverse(x): 5 4 3 2 1
x: 1 2 3
y: 1 2 3
append(x, y): 1 2 3 1 2 3
bash-3.2$
bash-3.2$
bash-3.2$ matchC list3.c
Compiling program ... DONE! [0.523s]
Loading Maude ........ DONE! [0.678s]
Verifying program ... DONE! [0.881s]
VerIFICATION succeeded! [418155 rewrites, 12 feasible and 0 infeasible paths]
Output: 1 2 3 4 5 5 4 3 2 1 1 2 3 1 2 3 1 2 3
bash-3.2$
bash-3.2$ 
```
Beyond Separation Logic Tools

```c
struct listNode *toListIterative(struct treeNode *t)
/*@ rule <k> $ => return ?l; </k>
    <heap_> tree(t)(T) => list(?l)(tree2list(T)) </heap> */
{
    struct listNode *l;
    struct stackNode *s;

    if (t == 0)
        return 0;

    l = 0;
    s = (struct stackNode *) malloc(sizeof(struct stackNode));
    s->val = t;
    s->next = 0;
   /*@ inv <heap_> treeList(s)(?TS), list(l)(?A) </heap>
     \ tree2list(T) = treeList2list(rev(?TS)) @ ?A */
    while (s != 0) {
        struct treeNode *tn;
        struct listNode *ln;
        struct stackNode *sn;
```
Beyond Separation Logic – 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);
```

```
```
Beyond Separation Logic – Stack Inspection

```c
void trusted(int n)
/*@ rule <k> $ => return; </k> <stack> S </stack> <out_> epsilon => A </out>
   if n >= 10 \ in(hd(ids(S)), {main, trusted}) */
{
    printf("%d ", n);
    untrusted(n);
    any(n);
    if (n)
        trusted(n - 1);
}

void untrusted(int n)
/*@ rule <k> $ => return; </k> <stack> S </stack> <out_> epsilon => A </out>
   if in(trusted, ids(S)) */
{
    printf("%d ", -n);
    if (n)
        any(n - 1);
}

void any(int n)
{
    // untrusted(n);
    if(n > 10)
        // possible security violated if n < 10
        trusted(n - 1);
}
```

Conclusions

• K (semantics) and Matching Logic (verification)
• Formal semantics is useful and practical!
• One can use an executable semantics of a language as is also for program verification
  – As opposed to redefining it as a Hoare logic
• Giving a formal semantics is not necessarily painful, it can be fun if one uses the right tools