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?
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 at best, or should formally relate to them at worst
    • Otherwise they are adhoc and likely wrong

• 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
  – Axiomatic, denotational, operational, etc.

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

• We want a powerful, unified foundation for language design, semantics and verification
  – One semantic approach to serve all the purposes!
  – To work with realistic languages (C, Java, etc.)
**Minimal Requirements for an Ideal Language Semantic Framework**

- Should be **expressive**
  - Substitution or environment-based definitions, abrupt control changes (callcc), concurrency, etc.
- Should be (efficiently) **executable**
  - So we can test it and use it in tools (symb. exec.)
- Should be **modular** (thus scale)
  - So each feature is defined once and for all
- Should serve as a basis for **program reasoning**
  - So we can also prove programs correct with it
- Conventional Semantic Approaches -
- Advantages and Limitations –

Chronologically
- 1969: Floyd-Hoare Logic -

• Basis for program verification
• Not easily executable, and thus, hard to test
  – Semantic errors found by proving wrong properties
  – Soundness rarely or never proved in practice
• Not very expressive
  – Often requires heavy program transformations (e.g., to eliminate side effects, pointers, exceptions, etc.), to reduce languages to cores which can be given an axiomatic semantics
  – Structural program properties (e.g., about heap, stacks, input/output, etc.) hard to state; need special logic support
  – Structural framing (e.g., heap framing) hard to deal with
• Implementations of Floyd-Hoare verifiers for real languages still an art, who few master
1971: Denotational Semantics -

Reasonable trade-offs. “Compiles” programs into mathematical objects, so it can be in principle:

• **Expressive**, provided enough/appropriate mathematical domains available

• **Executable**: we can execute/approximate fixed-points
  – Although factorial(5) crashes Papaspyrou’s C semantics

• **Modular**, provided one uses advanced features
  – Monads, continuations, resumptions

• **Basis for program verification**
  – Program = least fixed point, so we can use induction

• **Hard to use and understand; requires expert knowledge; no overwhelming evidence it is practical for verification**
Quite intuitive, easy to understand and define. Requires minimal training and it scales.

- **Executable**, by its very nature
  - Although Norish’s C semantics not executable, evaluation contexts are inefficient, CHAM has no machine support, etc.

- **Expressive** ... in principle
  - Although hard to use both evaluation contexts (for call/CC, longjumps,...) and environment-store (for pointers, threads,...)

- **Modular**, when one uses
  - MSOS ideas for dealing with configuration changes, evaluation contexts ideas for control, CHAM ideas for concurrency, etc.

- Considered “too low level”, inappropriate for verification...
What We Want

• First, we want a semantic framework which, at the same time and uniformly well, is
  – Expressive, Executable, Modular, and
  – Suitable for program reasoning

• Second, we want to develop supporting tools for
  – Defining formal language semantics, and
  – Using the semantics for program verification
    • Put an end to having both operational and axiomatic or other semantics to languages. No more semantics equivalence proofs to be done and maintained!
Our quest

Next I will tell a story about our quest for such a unified and practical semantic framework

Take our approach as a possibility ... ... not as ultimate answer

Message to take home: This is not a dream!
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 serve as a basis for program reasoning**
  - Admits initial model semantics, so it is amenable for inductive or fixed-point proofs
The K Framework

\[ \textit{k-framework.org} \]

• A tool-supported rewrite-based framework for defining programming language semantics

• Main ideas

  – Represent program configurations as potentially nested structures 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)
Complete K Definition of KernelC

Syntax declared using annotated BNF

SYNTAX  \( Exp ::= \) 
\[ Exp = Exp [\text{strict(2)}] \]
Complete K Definition of KernelC

Configuration given as a nested cell structure.
Leaves can be sets, multisets, lists, maps, or syntax.
Since then it has been expanded and used for expressing and verifying concurrency features and anomalies for both sequentially.

PhD Thesis
University of Illinois
December 121

Listing 1: Exp

```
Exp ::= List{Exp}
Exp ::= Pgm
Pgm ::= StmtList

StmtList ::= S1 + S2

S1 ::= T1 + T2
T1 ::= Xl + Xl

Xl ::= l= E
Xl ::= Xl [E]

E ::= Int
E ::= indef
E ::= strict
E ::= l= E
E ::= Xl [E]
E ::= Xl [E]

```

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

<env>... X |→ (_ => V) ...</env>

<k> X = V => V ...</k>
Don’t Like Bubbles?

• The KernelC definition above was generated by the K tool using the bubble mode (bb)

\[
\begin{align*}
\text{bb mode} & \quad X = V \\
& \quad \downarrow \\
& \quad V
\end{align*}
\]

• K tool also provides a mathematical mode (mm), which may be preferred in formal writing

\[
\begin{align*}
\text{mm mode} & \quad \langle X = V \ldots \rangle_k \quad \langle \ldots \quad X \mapsto - \quad \ldots \rangle_{env} \\
& \quad \downarrow \quad \downarrow \\
& \quad V \quad V
\end{align*}
\]
What is K, after all ... ?

Except for its true concurrency semantics, based on graph rewriting, K is a technique and notation to define languages as rewrite systems with rules

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

(b is a side condition)

... and so are reduction semantics with evaluation contexts, (chemical) abstract machines, etc.
Translating K into rules \( l \Rightarrow r \ if \ b \)

**SYNTAX** \( Exp ::= Exp = Exp \ [\text{strict}(2)] \)

**RULE** \( E_1 = E_2 \ \Rightarrow \ E_2 \leadsto E_1 = \square \) if \( E_2 \notin \text{Val} \)

**RULE** \( E_2 \leadsto E_1 = \square \ \Rightarrow \ E_1 = E_2 \) if \( E_2 \in \text{Val} \)

**RULE** \( k(X = V \leadsto K) \ env(\rho_1, X \mapsto V', \rho_2) \)
\( \Rightarrow k(Y \leadsto K) \ env(\rho_1, X \mapsto V, \rho_2) \)
What is the K Semantics of a Program?

Two types of K rules:

**Structural**: rearrange configuration, unobservable

**Computational**: count as computational steps

\[ t_1, \ldots, t_n = \text{structural variants} \]
What does the K Tool Offer?

Efficient and interactive execution (interpreters)
State-space exploration (search and model-checking)
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

- dynamic semantics
- type checker

of SIMPLE
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.

- untrusted() can only be called from trusted().
More Formally: Configurations

• For concreteness, assume configurations having the following syntax:

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

(matching logic works with any configurations)

• Examples of concrete (ground) configurations:

\[
\langle \langle x = ^* y ; y = x ; \ldots \rangle_k \langle x \mapsto 7, y \mapsto 3, \ldots \rangle_{\text{env}} \langle 3 \mapsto 5 \rangle_{\text{heap}} \ldots \rangle_{\text{cfg}}
\]

\[
\langle \langle x \mapsto 3 \rangle_{\text{env}} \langle 3 \mapsto 5, 2 \mapsto 7 \rangle_{\text{heap}} \langle 1, 2, 3, \ldots \rangle_{\text{in}} \langle \ldots, 7, 8, 9 \rangle_{\text{out}} \ldots \rangle_{\text{cfg}}
\]
More Formally: Patterns

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

\[ \exists c: \text{Cells}, \ e: \text{Env}, \ p: \text{Nat}, \ i: \text{Int}, \ \sigma: \text{Heap} \]
\[ \langle \langle x \mapsto p, \ e \rangle_{\text{env}} \langle p \mapsto i, \ \sigma \rangle_{\text{heap}} \rangle_{\text{cfg}} \land i > 0 \land p \neq i \]

• 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

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

• Hoare style - not recommended

\{ \pi_{\text{pre}} \} \text{ code} \ { \pi_{\text{post}} \}

– Need to redefine the PL semantics – impractical

• Rewriting (or K) style – recommended

\text{left} [\text{code}] \rightarrow \text{right}

– Can reuse existing operational semantics – 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:**

$ \Rightarrow return; \quad \ldots$  

**Heap:**

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

```
x \mapsto a

\mapsto b

\mapsto a
```

**Rule:**

$ \Rightarrow return; \quad \ldots$  

**Heap:**

```
if x = y
```

$ \Rightarrow return; \quad \ldots$  

**Heap:**

```
x \mapsto a
```

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

```
rule <k> $ => return p; </k>
<heap>... list(x,A) => list(p,rev(A)) ...</heap>
```
Partial Correctness

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

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

• Theorem (soundness):
  – If left \( \rightarrow \) right and “config matches left” such that config
    has a normal form for \( \rightarrow \), then “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**

$$A \vdash \varphi \Rightarrow \varphi$$

**Axiom**

$$\varphi \Rightarrow \varphi' \in A$$

$$A \vdash \varphi \Rightarrow \varphi'$$
More Formally: Proof System II

**Substitution**

\[
\mathcal{A} \vdash \varphi \Rightarrow \varphi' \quad \theta : \text{Var} \rightarrow T_\Sigma(\text{Var}) \\
\frac{}{\mathcal{A} \vdash \theta(\varphi) \Rightarrow \theta(\varphi')}
\]

**Transitivity**

\[
\mathcal{A} \vdash \varphi_1 \Rightarrow \varphi_2 \quad \mathcal{A} \vdash \varphi_2 \Rightarrow \varphi_3 \\
\frac{}{\mathcal{A} \vdash \varphi_1 \Rightarrow \varphi_3}
\]
More Formally: Proof System III

- Rules of deductive nature

**Case analysis**

\[
\begin{align*}
\mathcal{A} \vdash \varphi_1 \Rightarrow \varphi & \quad \mathcal{A} \vdash \varphi_2 \Rightarrow \varphi \\
\mathcal{A} \vdash \varphi_1 \lor \varphi_2 \Rightarrow \varphi
\end{align*}
\]

**Logic framing**

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

**Consequence**

\[ \models \varphi_1 \rightarrow \varphi'_1 \quad \mathcal{A} \vdash \varphi'_1 \Rightarrow \varphi'_2 \quad \models \varphi'_2 \rightarrow \varphi_2 \]

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

**Abstraction**

\[ \mathcal{A} \vdash \varphi \Rightarrow \varphi' \quad X \cap \text{FreeVars}(\varphi') = \emptyset \]

\[ \mathcal{A} \vdash \exists X \varphi \Rightarrow \varphi' \]
More Formally: Proof System V

- Main proof rule of matching logic rewriting

\[
\begin{align*}
\mathcal{A} \vdash \varphi \Rightarrow^+ \varphi'' & \quad \mathcal{A} \cup \{\varphi \Rightarrow \varphi'\} \vdash \varphi'' \Rightarrow \varphi' \\
\mathcal{A} \vdash \varphi \Rightarrow \varphi' & 
\end{align*}
\]
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:

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

• 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
  – Proof is constructive, not existential
• Partial correctness
  – Holds for ALL languages
MatchC

• A Matching Logic Verifier for (a fragment of) C
• Uses K semantics of C fragment *unchanged*
• Has verified a series of challenging programs
  – Undefiness, typical Hoare-like programs, heap programs (lists, trees, stacks, queues, graphs), sortings, AVL trees, Schorr-Waite graph marking
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