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

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

    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)
Complete K Definition of KernelC
Complete K Definition of KernelC

Syntax declared using annotated BNF

SYNTAX

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

Configuration given as a nested cell structure. Leaves can be sets, multisets, lists, maps, or syntax
Exploring relaxed memory models

Grigore Rosu, Wolfram Schulte, and Traian Florin Serban

University of Illinois at Urbana-Champaign

Import K-Syntax

Syntax Exp ::=
  | Id
  | Exp = Exp [assoc]
  | Exp += Exp [assoc]
  | Exp = Exp [strict]
  | Exp += Exp [strict]
  | Exp = Exp [leq]
  | Exp += Exp [leq]
  | Exp = Exp [geq]
  | Exp += Exp [geq]
  | Exp = Exp [lt]
  | Exp += Exp [lt]
  | Exp = Exp [gt]
  | Exp += Exp [gt]
  | Exp = Exp [==]
  | Exp += Exp [==]
  | Exp = Exp [!=]
  | Exp += Exp [!=]
  | Exp = Exp [<=]
  | Exp += Exp [<=]
  | Exp = Exp [>=]
  | Exp += Exp [>=]
  | Exp = Exp [<]
  | Exp += Exp [<]
  | Exp = Exp [>]
  | Exp += Exp [>]
  | Exp = Exp [not] [lt]
  | Exp += Exp [not] [lt]
  | Exp = Exp [not] [gt]
  | Exp += Exp [not] [gt]
  | Exp = Exp [not] [>=]
  | Exp += Exp [not] [>=]
  | Exp = Exp [not] [<=]
  | Exp += Exp [not] [<=]
  | Exp = Exp [not] [==]
  | Exp += Exp [not] [==]
  | Exp = Exp [not] [!=]
  | Exp += Exp [not] [!=]
  | Exp = Exp [not] [lt]
  | Exp += Exp [not] [lt]
  | Exp = Exp [not] [gt]
  | Exp += Exp [not] [gt]
  | Exp = Exp [not] [>=]
  | Exp += Exp [not] [>=]
  | Exp = Exp [not] [<=]
  | Exp += Exp [not] [<=]
  | Exp = Exp [not] [==]
  | Exp += Exp [not] [==]
  | Exp = Exp [not] [!=]
  | Exp += Exp [not] [!=]
  | Exp = Exp [true]
  | Exp += Exp [true]
  | Exp = Exp [false]
  | Exp += Exp [false]
  | Exp = Exp [null]
  | Exp += Exp [null]

Syntactic operations:

- Assoc: (Exp) = (Exp)  
- Stricts3: ((Exp)) = (Exp)

Semantic rules given contextually

<k> X = V => V ... </k>
<env>... X |-> (\_ => V) ... </env>
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

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

• For concreteness, assume configurations having the following syntax:

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

(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

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

• 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

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

\[
\begin{align*}
\text{rule} & \quad \text{k} \quad \$ \Rightarrow \text{return}; \quad \ldots \text{/k} \\
& \quad \text{<heap>...} \\
& \quad x \mapsto (a \Rightarrow b), \\
& \quad y \mapsto (b \Rightarrow a) \\
& \quad \ldots \text{/heap>}
\end{align*}
\]

\[
\begin{align*}
\text{rule} & \quad \text{k} \quad \$ \Rightarrow \text{return}; \quad \ldots \text{/k} \\
& \quad \text{<heap>...} \\
& \quad x \mapsto a \\
& \quad \text{if} \quad x = y \\
& \quad \text{<heap>...} \quad x \mapsto a \quad \ldots \text{/heap>}
\end{align*}
\]
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; \textit{safe}
  \[ \rightarrow \] given by specifications; \textit{unsafe}, has to be proved

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

• Theorem (soundness):
  – If \texttt{left} \rightarrow \texttt{right} and “\texttt{config matches left}” such that \texttt{config}
    has a normal form for \rightarrow, then “\texttt{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:
\[
\text{Reflexivity:} \quad \frac{}{\mathcal{A} \vdash \varphi \Rightarrow \varphi}
\]

Replacement:
\[
\text{Replacement:} \quad \frac{\theta : \text{Var} \rightarrow T_{\Sigma}(\text{Var})
\varphi \Rightarrow \varphi' \text{ if } \varphi_1 \Rightarrow \varphi'_1, \ldots, \varphi_n \Rightarrow \varphi'_n \in \mathcal{A}
\mathcal{A} \vdash \{ \theta(\varphi_1) \Rightarrow \theta(\varphi'_1), \ldots, \theta(\varphi_n) \Rightarrow \theta(\varphi'_n) \}}{\mathcal{A} \vdash \theta(\varphi) \Rightarrow \theta(\varphi')}
\]

Transitivity:
\[
\text{Transitivity:} \quad \frac{\mathcal{A} \vdash \{ \varphi_1 \Rightarrow \varphi_2, \varphi_2 \Rightarrow \varphi_3 \}}{\mathcal{A} \vdash \varphi_1 \Rightarrow \varphi_3}
\]
More Formally: Proof System II

• Rules of deductive nature

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

Consequence:

\[
\psi \text{ is a FOL}_- \text{ formula} \\
\mathcal{A} \vdash \varphi \Rightarrow \varphi' \\
\hline \\
\mathcal{A} \vdash \varphi \land \psi \Rightarrow \varphi' \land \psi
\]

Logic frame:
More Formally: Proof System IV

• Main proof rule of matching logic rewriting

**Circularity:**

C 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

• A Matching Logic Verifier for (a fragment of) C
• Uses the K semantics of the 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
• Andrei Stefanescu will give a demo next
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