Matching Logic

Grigore Rosu
University of Illinois at Urbana-Champaign

Joint work with Andrei Stefanescu and Chucky Ellison. Started with Wolfram Schulte at Microsoft Research in 2009
Question

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

Hoare logic

\[
\{
\pi_{pre}\}
\xrightarrow{\text{code}}
\{
\pi_{post}\}
\]
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
- 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
  – Norish’s C semantics is not 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-

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

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

Inspired from rewriting logic

Used regularly in teaching undergraduate courses

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
Complete K Definition of KernelC

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
• Soon to be compiled to
  – OCAML, for fast execution
  – COQ, for meta-property verification
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 = K + FOL

• 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

\[
\begin{align*}
\text{env} & \quad x \mapsto a \\
\text{mem} & \quad \text{list}(a, A) \\
\text{out} & \quad \text{rev}(A) \land |A| > 1
\end{align*}
\]

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

\[
\begin{align*}
\text{k} & \quad \text{untrusted}() \\
\text{fstack} & \quad \text{trusted}()
\end{align*}
\]
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

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

– 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 <k> $ => return; <_/k>
  <heap_>
  x|->(a=>b),
  y|->(b=>a)
  <_/heap>
```

```
rule <k> $ => return; <_/k>
  <heap_>
  x|-> a
  y|-> a
  <_/heap>
```

```
if x = y
```

```
rule <k> $ => return; <_/k>
  <heap_> x|-> a 
  if x = y
```

Example – Reversing a list

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”
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.
## Assertion Checking

```c
#include <stdio.h>
int sum(int n) {
    int s = 0;
    while (n > 0) {
        s += n;
        n -= 1;
    }
    return s;
}

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

    //@ assert <out> [55] </out>
    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$ ```
Full Verification

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

int sum(int n)
{ //@ rule <k> $ => return (n * (n + 1)) / 2; </k> if n >= 0
   int s;
   s = 0;
   //@ inv s = (old(s) + n) 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> [!
   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
Beyond Separation Logic Tools
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