Matching Logic
A New Program Verification Approach

Grigore Rosu
University of Illinois at Urbana-Champaign

(work started in 2009 with Wolfram Schulte at MSR, then continued with Chucky Ellison and Andrei Stefanescu)
Current Verification Efforts ...

- Relatively clear objectives of current V&V:
  - Better tools, more connected, more user friendly
  - Teach students verification early
  - Get the best from what we have

- But ... 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

Consider some programming language, L

• Formal semantics of L
  – Typically skipped: considered expensive and useless

• Model checkers for L
  – Based on some adhoc encodings of L

• Program verifiers for L
  – Based on some other adhoc encodings of L

• Runtime verifiers for L
  – Based on yet another adhoc encodings of L

• ...


Semantic Gap

• Why would I trust any of these tools for L?
• How do they relate to L?
• What is L?

• Example: the C (very informal) manual implies that \((x=0) + (x=0)\) is undefined
  – Yet, all C verifiers we looked into “prove” it = 0
Ideal Scenario

- Have *one formal definition of L* which serves all the semantic and verification purposes

- Execution of L programs (use for extensive testing)

- Model checking of L programs

- Proving L programs correct

...
How It Started

• NASA project runtime verification effort
  – Use runtime verification guarantees to ease the task for program verification

• Thus, looked for “off-the-shelf” verifiers
  – Very disappointing experience ...

• We started with a simple program, the list reverse example, and wanted to see how different verifiers can prove it correct ...
Our Little Benchmark

Reversing of a C list: if $x$ points to a list at the beginning, then $p$ points to its reverse at the end

```
p = 0;
while(x != 0) {
    y = *(x + 1);
    *(x + 1) = p;
    p = x;
    p = y;
    x = y;
}
```

We were willing to even annotate the program
Current Program Verifiers

• Current program verifiers are based on Hoare logic (and WP), separation logic, dynamic logic

• Hoare-logic-based
  – Caduceus/Why, VCC, HAVOC, ESC/Java, Spec#
  – Hard to reason about heaps, frame inference difficult; either (very) interactive, or very slow, or unsound

• Separation-logic-based
  – Smallfoot, Bigfoot, Holfoot* (could prove it! 1.5s), jStar
  – Very limited (only memory safety) and focused on the heap; Holfoot, the most general, is very slow
... therefore, we asked for professional help: Wolfram Schulte (Spec# and other tools)
Do we Have a Problem in what regards Program Verification?

• Blame is often on tools, such as SAT/SMT solvers, abstractions, debuggers, static analyzers, slow computers, etc.,

• ... but not on the theory itself, Hoare/Floyd logic – and its various extensions (such as separation logic)
Overview

• Hoare/Floyd logic
• Matching Logic
• Short Demo
• Conclusion and Future Work
Hoare/Floyd Logic

• Assignment rules
  – Hoare (backwards, but no quantifiers introduced)
    \[
    \{ \varphi[e/x] \} \ x := e \ {\varphi}\n    \]
  – Floyd (forwards, but introduces quantifiers)
    \[
    \{ \varphi \} \ x := e \ \exists v. \ (x = e[v/x]) \land \varphi[v/x]\n    \]
Hoare/Floyd Logic

- Loop invariants

\[
\{\varphi \land (e \neq 0)\} \triangleright \{\varphi\} \\
\{\varphi\} \text{ while } (e) \triangleright \{\varphi \land (e = 0)\}
\]

- Minor problem: does not work when \( e \) has side effects; those must be first isolated out
Hoare/Floyd Logic

Important observation

Hoare/Floyd logic, as well as many other logics for program verification, deliberately stay away from “low-level” operational details, such as program configurations

... missed opportunity
What We Want

• Forwards
  – more intuitive as it closely relates to how the program is executed; easier to debug; easier to combine with other approaches (model checking)

• No quantifiers introduced

• Conventional logics for specifications, say FOL

• To deal at least with existing languages and language extensions
  – E.g., Hoare logic has difficulty with the heap; separation logic only deals with heap extensions
Overview

• Hoare/Floyd logic
• Matching Logic
• Short Demo
• Conclusion and Future Work
Ideal Scenario

• Have *one formal definition of L* which serves all the semantic and verification purposes

- Execution of L programs
  - (use for extensive testing)

- Model checking of L programs

- Proving L programs correct

...
Our Approach

• Define languages using the K framework
  – A rewrite based framework which generalizes both evaluation contexts and the CHAM

• A programming language is a K system
  – Algebraic signature (syntax + configuration)
  – K rewrite rules (make read/write parts explicit)

• “Compile” K to different back-ends
  – To OCAML for efficient interpreters (experimental)
  – To Maude for execution, debugging, verification
K: Program Configurations (no heap)

• Simple configuration using a computation and an environment

\[
\langle \langle \cdots \rangle \rangle_k \langle \cdots \rangle_{env}
\]

• Example

\[
\langle \langle x := 1; y := 2 \rangle \rangle_k \langle x \mapsto 3, y \mapsto 3, z \mapsto 5 \rangle_{env}
\]
K: Program Configurations (add heap)

• Add a heap to the configuration structure:

\[ ⟨\langle\cdots\rangle_k \langle\cdots\rangle_{env} \langle\cdots\rangle_{mem}⟩ \]

• Example

\[ ⟨\langle x \leftarrow 2, y \leftarrow 2 \rangle_{env} \langle 2 \leftarrow 7 \rangle_{mem}⟩ \]
Complex Program Configuration
The CHALLENGE Language (J.LAP 2010)
Matching Logic

• Builds upon operational semantics
  – We use K, but in principle can work with any op semantics: a formal notion of configuration is necessary
  – With K, we do not modify anything in the original sem!
• Specifications: special FOL\(=\) formulae, \textit{patterns}
• Configurations \textit{match} patterns
• Patterns can be used to
  1. Give an axiomatic semantics to a language, so that we can reason about programs
  2. Define and reason about patterns of interest in program configurations
Patterns

• Configuration terms with constrained variables

\[
\langle \langle x := 1 ; y := 2 \rangle_k \langle x \mapsto \#a, y \mapsto \#a, \#\rho \rangle_{env} \langle \#a \geq 0 \rangle_{form} \rangle
\]

configuration term with variables constraints

\[
\langle \langle [x] := 5 ; z := [y] \rangle_k \langle x \mapsto \#a, y \mapsto \#a, \#\rho \rangle_{env} \langle \#a \mapsto \#v, \#\sigma \rangle_{mem} \langle \#a \geq 0 \rangle_{form} \rangle
\]

configuration term with variables constraints
Pattern Matching

• Configurations *match* \( \models \) patterns iff they match the structure and satisfy the constraints

\[
\langle \langle x := 1; y := 2 \rangle_k \langle x \mapsto 3, y \mapsto 3, z \mapsto 5 \rangle_{env} \rangle \models \langle \langle x := 1; y := 2 \rangle_k \langle x \mapsto ?a, y \mapsto ?a, ?\rho \rangle_{env} \langle ?a \geq 0 \rangle_{form} \rangle
\]

\[
\langle \langle [x] := 5; z := [y] \rangle_k \langle x \mapsto 2, y \mapsto 2 \rangle_{env} \langle 2 \mapsto 7 \rangle_{mem} \rangle \models \langle \langle [x] := 5; z := [y] \rangle_k \langle x \mapsto ?a, y \mapsto ?a, ?\rho \rangle_{env} \langle ?a \mapsto ?v, ?\sigma \rangle_{mem} \langle ?a \geq 0 \rangle_{form} \rangle
\]
What Can We Do With Patterns?

1. Give axiomatic semantics to programming languages, to reason about programs
   - Like Hoare logic, but different

2. Give axioms over configurations, to help identify patterns of interest in them
   - Like lists, trees, graphs, etc.
1. Axiomatic Semantics
- follow the operational semantics -

- Partial correctness pairs:
  \[ \Gamma \Downarrow \Gamma' \]

- Assignment
  \[
  \frac{
    \langle \langle e \rangle_k C \rangle \Downarrow \langle \langle v \rangle_k C' \rangle
  }{
    \langle \langle x = e \rangle_k C \rangle \Downarrow \langle \langle \cdot \rangle_k C'[x \leftarrow v] \rangle
  }
  \] (ML-ASGN)

- While
  \[
  \frac{
    \langle \langle e \rangle_k C \rangle \Downarrow \langle \langle v \rangle_k C' \rangle
  }{
    \langle \langle s \rangle_k (C' \land (v \neq 0)) \rangle \Downarrow \langle \langle \cdot \rangle_k C \rangle
  }
  \]
  \[
  \frac{
    \langle \langle \text{while}(e)s \rangle_k C \rangle \Downarrow \langle \langle \cdot \rangle_k (C' \land (v = 0)) \rangle
  }{
    \langle \langle \text{while}(e)s \rangle_k C \rangle \Downarrow \langle \langle \cdot \rangle_k (C' \land (v = 0)) \rangle
  }
  \] (ML-WHILE)
2. Configuration Axioms

• For example, lists in the heap:

\[ \langle \langle \text{list}(p, \alpha), \sigma \rangle_{\text{mem}} \langle \varphi \rangle_{\text{form}} \ C \rangle \]
\[
\Leftrightarrow \langle \langle \sigma \rangle_{\text{mem}} \langle p = 0 \land \alpha = \epsilon \land \varphi \rangle_{\text{form}} \ C \rangle 
\]
\[
\lor \langle \langle p \mapsto [a, q], \text{list}(q, \beta), \sigma \rangle_{\text{mem}} \langle \alpha = a : \beta \land \varphi \rangle_{\text{form}} \ C \rangle 
\]

• Sample configuration properties:

\[ \langle 5 \mapsto 2, 6 \mapsto 0, 8 \mapsto 3, 9 \mapsto 5, \sigma \rangle_{\text{mem}} \ C \rangle \Rightarrow \langle \langle \text{list}(8, 3 : 2), \sigma \rangle_{\text{mem}} \ C \rangle, \text{ and} 
\]
\[ \langle \langle \text{list}(8, 3 : 2), \sigma \rangle_{\text{mem}} \ C \rangle \Rightarrow \langle 8 \mapsto 3, 9 \mapsto q, q \mapsto 2, q + 1 \mapsto 0, \sigma \rangle_{\text{mem}} \ C \rangle \]
Demo

• See the Matching Logic “Try it Online” web interface, or, alternatively, download the project from Google projects and then go to examples/matchC/demo.
Highlights

• Matching logic builds upon giants’ shoulders
  – Matching and rewriting “modulo” have been researched extensively; efficient algorithms (Maude) despite its complexity (NP complete w/o constraints)
  – Mathematical universe axiomatized using well understood and developed algebraic specification

\[
\begin{align*}
\text{rev}(\text{nil}) & = \text{nil} \\
\text{rev}([a]) & = [a] \\
\text{rev}(A_1 @ A_2) & = \text{rev}(A_1) @ \text{rev}(A_2)
\end{align*}
\]
Matching is Powerful

• The underlying rewrite machinery of K works by means of matching
  – So programming language semantics, which is most of the matching logic rules, is matching
• Pattern assertion reduces to matching
• Framing reduces to matching
• Separation reduces to matching

• Nothing special needs to be done for separation or for framing!
K and Matching Logic Scale

• We defined several real languages so far
  – Complete: C (C99), Scheme
  – Large subsets: Verilog, Java 1.4
  – In work: X10, Haskell, JavaScript
• And tens of toy or paradigmatic languages
• We next give an overview of the C definition
  – Defined by Chucky Ellison (PhD at UIUC)
Configuration of C

- 57 leaf cells
- 63 nested cells

Heap
Statistics for the C definition

- Syntactic constructs: 173
- Total number of rules: 812
- Total number of lines: 4688

- Has been tested on thousands of C programs (several benchmarks, including the gcc torture test – passed 95% so far)
Conclusion and Future Work

• Formal verification should start with a formal, executable semantics of the language
• Once a well-tested formal semantics is available, developing program verifiers should be an easy task, available to masses

• Matching logic aims at the above
• It makes formal semantics useful!
• It additionally encourages developing formal semantics to languages, which in K is easy and fun