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
- …

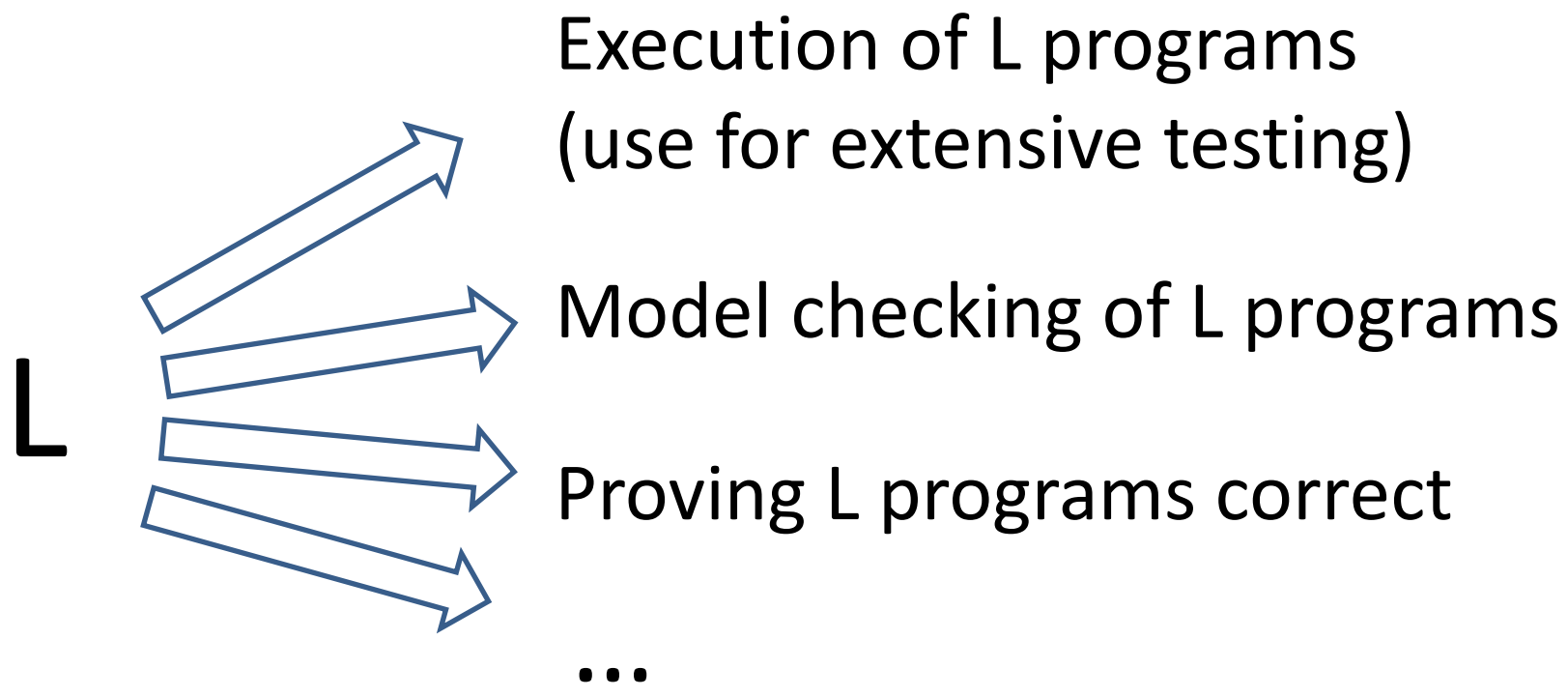
Semantic Gap

- 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



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 lists 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;
      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)

$$\frac{\cdot}{\{\varphi[\mathtt{e}/\mathtt{x}]\}\ \mathtt{x} := \mathtt{e}\ \{\varphi\}}$$

  - Floyd (forwards, but introduces quantifiers)

$$\frac{\cdot}{\{\varphi\}\ \mathtt{x} := \mathtt{e}\ \{\exists v.\ (\mathtt{x} = \mathtt{e}[v/\mathtt{x}]) \wedge \varphi[v\ /\ \mathtt{x}]\}}$$

Hoare/Floyd Logic

- Loop invariants

$$\frac{\{\varphi \wedge (\mathtt{e} \neq 0)\}\ \mathtt{s}\ \{\varphi\}}{\{\varphi\}\ \mathtt{while}\,(\mathtt{e})\ \mathtt{s}\ \{\varphi \wedge (\mathtt{e} = 0)\}}$$

- 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

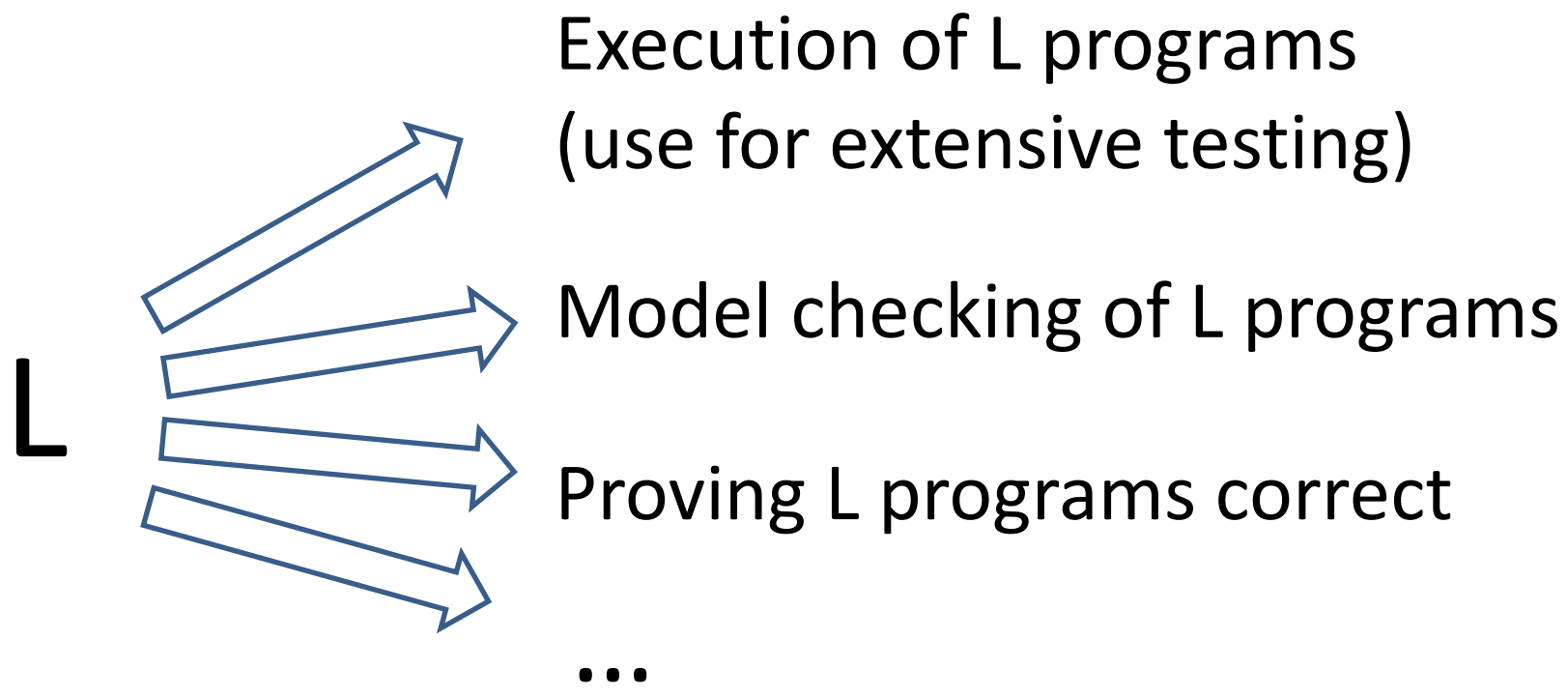
- Forwards
  - more intuitive as it closely relates to how the program is executed; easier to debug; easier to combine with other approaches (model checking)
- No quantifiers introduced
- Conventional logics for specifications, say FOL
- To deal at least with existing languages and language extensions
  - 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



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

$$\langle \langle \cdots \rangle_k \ \langle \cdots \rangle_{env} \rangle$$

- Example

$$\langle \langle \mathtt{x := 1 ; y := 2} \rangle_k \ \langle \mathtt{x} \mapsto 3, \ \mathtt{y} \mapsto 3, \ \mathtt{z} \mapsto 5 \rangle_{env} \rangle$$

K: Program Configurations (add heap)

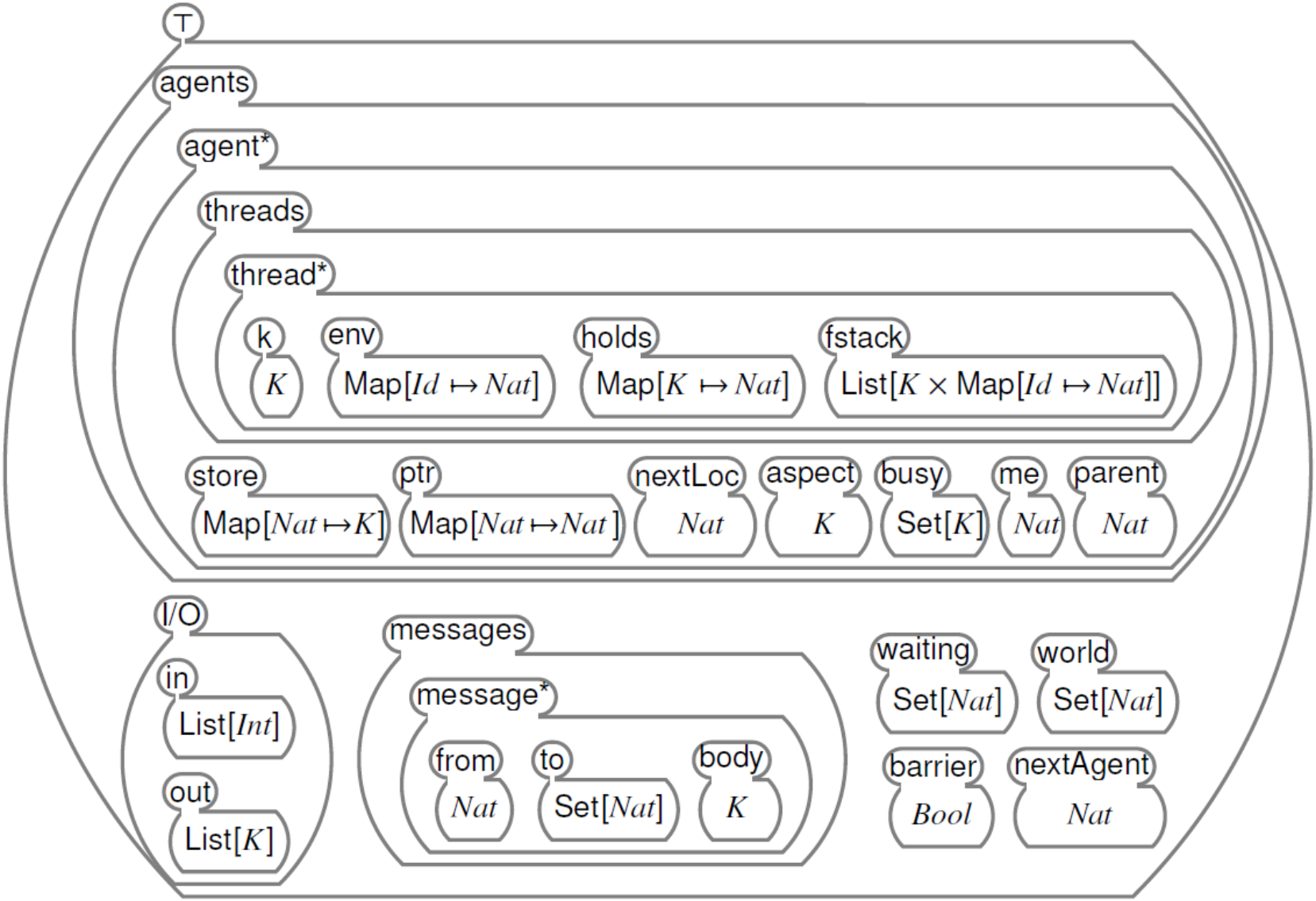
- Add a heap to the configuration structure:

$$\langle \langle \cdots \rangle_k \langle \cdots \rangle_{env} \langle \cdots \rangle_{mem} \rangle$$

- Example

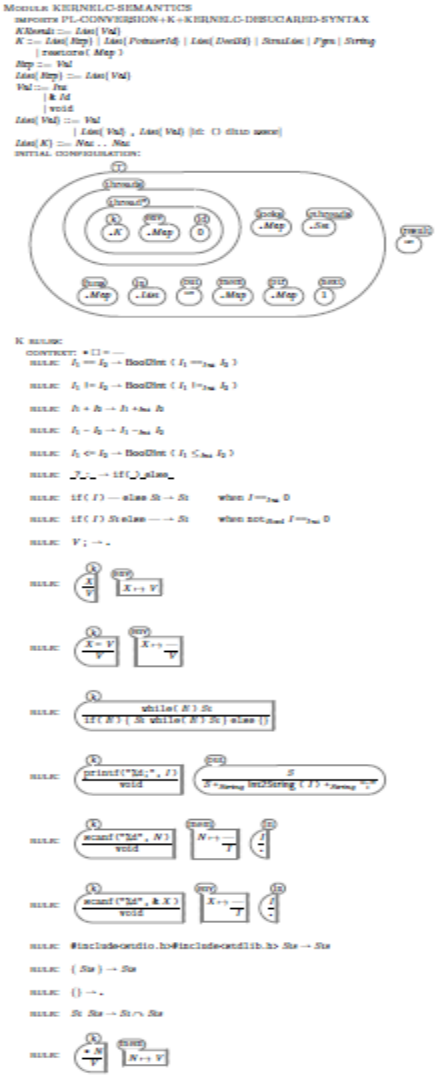
$$\langle \langle \texttt{[x] := 5 ; z := [y]} \rangle_k \langle \texttt{x} \mapsto 2, \texttt{y} \mapsto 2 \rangle_{env} \langle 2 \mapsto 7 \rangle_{mem} \rangle$$

Complex Program Configuration
The CHALLENGE Language (J.LAP 2010)



KernelC

```
MODULE KERNELC-SYNTAX
  IMPORTS PL-ID+PL-INT
  PointerId ::= Id
              | * PointerId [strict]
  Exp ::= Int | PointerId | DeclId
        | * Exp [ditto]
        | Exp + Exp [strict]
        | Exp - Exp [strict]
        | Exp == Exp [strict]
        | Exp != Exp [strict]
        | Exp <= Exp [strict]
        | ! Exp
        | Exp && Exp
        | Exp || Exp
        | Exp ? Exp : Exp
        | Exp = Exp [strict(2)]
        | printf("%d;", Exp ) [strict]
        | scanf("%d", Exp ) [strict]
        | & Id
        | Id ( List{Exp} ) [strict(2)]
        | Id ()
        | Exp ++
        | NULL
        | free( Exp ) [strict]
        | (int*)malloc( Exp *sizeof(int)) [strict]
        | Exp [ Exp ]
        | spawn Exp
        | acquire( Exp ) [strict]
        | release( Exp ) [strict]
        | join( Exp ) [strict]
  StmtList ::= Stmt
             | StmtList StmtList
  List{Bottom} ::= Bottom
                 | ()
                 | List{Bottom} , List{Bottom} [id: () strict hybrid assoc]
  List{PointerId} ::= PointerId | List{Bottom}
                    | List{PointerId} , List{PointerId} [id: () ditto assoc]
  List{DeclId} ::= DeclId | List{Bottom}
                 | List{DeclId} , List{DeclId} [id: () ditto assoc]
  List{Exp} ::= Exp | List{PointerId} | List{DeclId}
              | List{Exp} , List{Exp} [id: () ditto assoc]
  DeclId ::= int Exp
           | void PointerId
  Stmt ::= Exp ; [strict]
         | {}
         | { StmtList }
         | if ( Exp ) Stmt
         | if ( Exp ) Stmt else Stmt [strict(1)]
         | while( Exp ) Stmt
         | DeclId List{DeclId} { StmtList }
         | DeclId List{DeclId} { StmtList return Exp ; }
         | #include< StmtList >
  Id ::= main
  Pgm ::= #include<stdio.h>#include<stdlib.h> StmtList
END MODULE
MODULE KERNELC-DESUGARED-SYNTAX
  IMPORTS KERNELC-SYNTAX
  MACRO: ! E => E ? 0 : 1
  MACRO: E1 && E2 => E1 ? E2 : 0
  MACRO: E1 || E2 => E1 ? 1 : E2
  MACRO: if ( E ) St => if ( E ) St else {}
  MACRO: NULL => 0
  MACRO: I () => I ( () )
  MACRO: DI L { Sts } => DI L { Sts return 0 ; }
  MACRO: void PI => int PI
  MACRO: int * PI => int PI
  MACRO: #include< Sts > => Sts
  MACRO: E1 [ E2 ] => * E1 + E2
  MACRO: int * PI = E => int PI = E
  MACRO: E ++ => E = E + 1
END MODULE
```



```
MODULE KERNELC-SEMANTICS
  IMPORTS PL-CONVERSION+K+KERNELC-DESUGARED-SYNTAX
  KItem ::= List{Val}
  K ::= List{Stp} | List{PointerId} | List{DeclId} | StmtList | Pgm | String
      | restore( Map )
  Stp ::= Val
  List{Stp} ::= List{Val}
  Val ::= Int
        | & Id
        | void
  List{Val} ::= Val
              | List{Val} , List{Val} [id: () ditto assoc]
  List{K} ::= Next . . Next
  INITIAL CONFIGURATION:
```



```
  K RULES:
    CONTEXT: * [] = ...
    RULE: I1 == I2 => BoolInt ( I1 ==Int I2 )
    RULE: I1 != I2 => BoolInt ( I1 !=Int I2 )
    RULE: I1 + I2 => I1 +Int I2
    RULE: I1 - I2 => I1 -Int I2
    RULE: I1 <= I2 => BoolInt ( I1 <=Int I2 )
    RULE: _?_:_ => if(_)else_
    RULE: if( I ) _ else St => St        when I =/=Int 0
    RULE: if( I ) St else _ => St        when notBool I =/=Int 0
    RULE: V ; => .
    RULE:
```



```
    RULE:
```



```
    RULE:
```



```
    RULE:
```



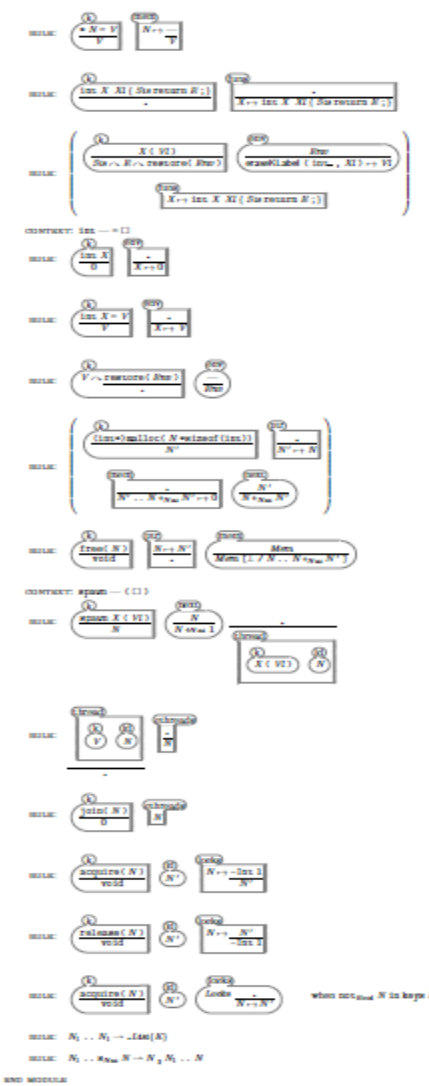
```
    RULE:
```



```
    RULE:
```



```
    RULE: #include<stdio.h>#include<stdlib.h> Sts => Sts
    RULE: { Sts } => Sts
    RULE: {} => .
    RULE: St Sts => St ↷ Sts
    RULE:
```

```
  RULE:
```



```
  RULE:
```



```
  RULE:
```



```
  CONTEXT: int _ = []
  RULE:
```



```
  RULE:
```



```
  RULE:
```



```
  RULE:
```



```
  RULE:
```



```
  CONTEXT: spawn _ = ( [] )
  RULE:
```



```
  RULE:
```



```
  RULE:
```



```
  RULE:
```



```
  RULE:
```



```
  RULE:
```



```
when notBool N in keys Locks
  RULE: N1 . . N2 => .List{K}
  RULE: N1 . . N +Int N => N , N1 . . N
END MODULE
```

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, *patterns*
- Configurations *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

$$\langle\langle \texttt{x := 1 ; y := 2} \rangle_k \ \langle \texttt{x} \mapsto ?a,\ \texttt{y} \mapsto ?a,\ ?\rho \rangle_{env} \ \langle ?a \geq 0 \rangle_{form} \rangle$$

configuration term with variables — constraints

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

configuration term with variables — constraints

Pattern Matching

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

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

$$\langle\langle [\mathtt{x}] := 5 ; \mathtt{z} := [\mathtt{y}] \rangle_k \langle \mathtt{x} \mapsto 2,\ \mathtt{y} \mapsto 2 \rangle_{env}\ \langle 2 \mapsto 7 \rangle_{mem} \rangle \models$$
$$\langle\langle [\mathtt{x}] := 5 ; \mathtt{z} := [\mathtt{y}] \rangle_k \langle \mathtt{x} \mapsto ?a,\ \mathtt{y} \mapsto ?a,\ ?\rho \rangle_{env}\ \langle ?a \mapsto ?v,\ ?\sigma \rangle_{mem}\ \langle ?a \geq 0 \rangle_{form} \rangle$$

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

$$\frac{\left\langle \langle \mathtt{e} \rangle_k \; C \right\rangle \Downarrow \left\langle \langle \mathtt{v} \rangle_k \; C' \right\rangle}{\left\langle \langle \mathtt{x = e} \rangle_k \; C \right\rangle \Downarrow \left\langle \langle \cdot \rangle_k \; C'[x \leftarrow v] \right\rangle} \quad \text{(ML-ASGN)}$$

- While

$$\frac{\begin{array}{c}\left\langle \langle \mathtt{e} \rangle_k \; C \right\rangle \Downarrow \left\langle \langle \mathtt{v} \rangle_k \; C' \right\rangle \\ \left\langle \langle s \rangle_k \; (C' \wedge (v \neq 0)) \right\rangle \Downarrow \left\langle \langle \cdot \rangle_k \; C \right\rangle\end{array}}{\left\langle \langle \mathtt{while\,(e)\,s} \rangle_k \; C \right\rangle \Downarrow \left\langle \langle \cdot \rangle_k \; (C' \wedge (v = 0)) \right\rangle} \quad \text{(ML-WHILE)}$$

2. Configuration Axioms

- For example, lists in the heap:

$$\langle\langle \mathrm{list}(p,\alpha),\ \sigma\rangle_{mem}\ \langle\varphi\rangle_{form}\ C\rangle$$

$$\Leftrightarrow\ \langle\langle\sigma\rangle_{mem}\ \langle p=0\wedge\alpha=\epsilon\wedge\varphi\rangle_{form}\ C\rangle$$

$$\vee\ \langle\langle p\mapsto[?a,?q],\ \mathrm{list}(?q,?\beta),\ \sigma\rangle_{mem}\ \langle\alpha=?a{:}?\beta\wedge\varphi\rangle_{form}\ C\rangle$$

- Sample configuration properties:

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

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

$$\begin{aligned} \mathsf{rev}(\mathsf{nil}) &= \mathsf{nil} \\ \mathsf{rev}([a]) &= [a] \\ \mathsf{rev}(\mathsf{A}_1 @ \mathsf{A}_2) &= \mathsf{rev}(\mathsf{A}_1) @ \mathsf{rev}(\mathsf{A}_2) \end{aligned}$$

Matching is Powerful

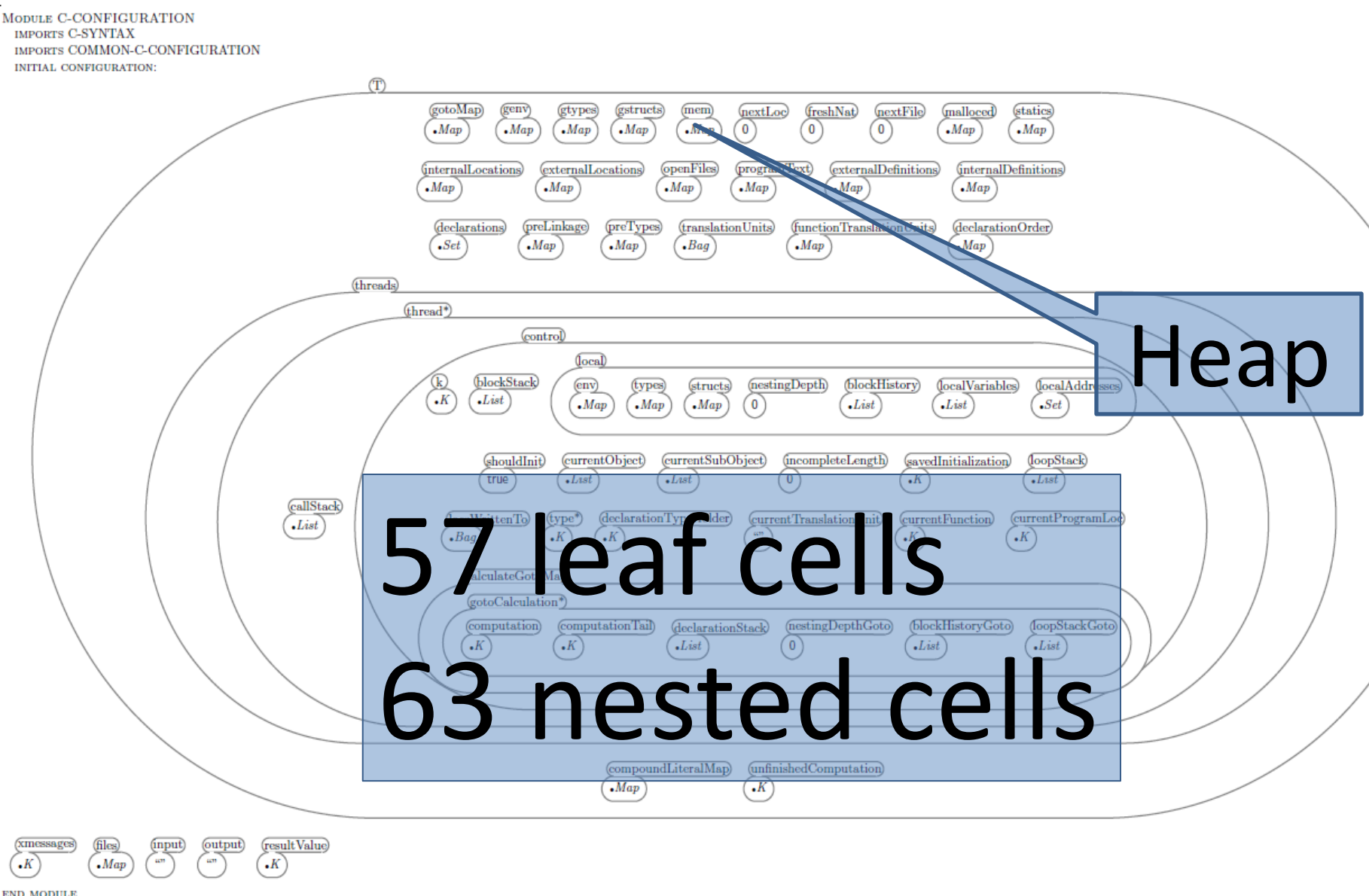
- The underlying rewrite machinery of K works by means of matching
  - So programming language semantics, which is most of the matching logic rules, is matching
- Pattern assertion reduces to matching
- Framing reduces to matching
- Separation reduces to matching

- 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



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