Matching Logic
A New Program Verification Approach

Grigore Rosu and Andrei Stefanescu
University of Illinois at Urbana-Champaign

(work started in 2009 with Wolfram Schulte at MSR)

Usable Verification …

- Relatively clear objectives:
  - 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

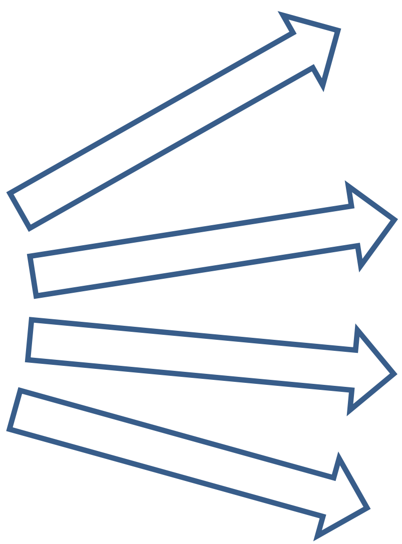
- Why would I trust any of these tools for L ?
- How do they relate to L ?
- What is L ?

- 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

L



Execution of L programs
(use for extensive testing)

Model checking of L programs

Proving L programs correct

…

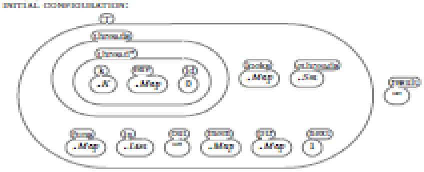
Our Approach

- 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

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
  KResult ::= List{Val}
  K ::= List{Exp} | List{PointerId} | List{DeclId} | StmtList | Pgm | String
      | restore( Map )
  Exp ::= Val
  List{Exp} ::= List{Val}
  Val ::= Int
        | & Id
        | void
  List{Val} ::= Val
              | List{Val} , List{Val} [id: () ditto assoc]
  List{K} ::= Nat .. Nat
  INITIAL CONFIGURATION:
```



K RULES:

CONTEXT: $* \square = \_\_$

RULE: $I_1 == I_2 \to \mathrm{Bool2Int}(I_1 ==_{Int} I_2)$

RULE: $I_1 \mathrel{!}= I_2 \to \mathrm{Bool2Int}(I_1 \mathrel{!}=_{Int} I_2)$

RULE: $I_1 + I_2 \to I_1 +_{Int} I_2$

RULE: $I_1 - I_2 \to I_1 -_{Int} I_2$

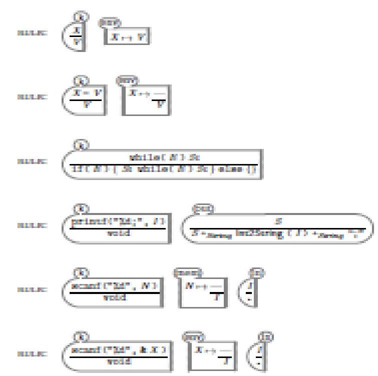
RULE: $I_1 <= I_2 \to \mathrm{Bool2Int}(I_1 \leq_{Int} I_2)$

RULE: `_?_:_` → `if(_)_else_`

RULE: `if( I ) — else St` → St   when $I ==_{Int} 0$

RULE: `if( I ) St else —` → St   when $\mathrm{not}_{Bool}\ I ==_{Int} 0$

RULE: `V ;` → `.`

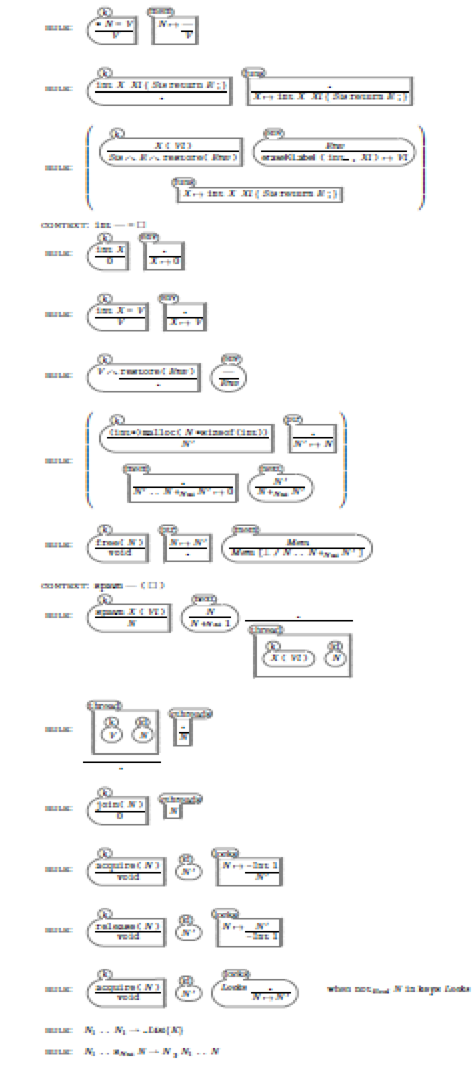


RULE: `#include<stdio.h>#include<stdlib.h> Sts` → Sts

RULE: `{ Sts }` → Sts

RULE: `{}` → `.`

RULE: `St Sts` → `St ↷ Sts`



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!
- Extends the PL semantics with matching logic specifications, or *patterns*; for example,

$$\langle \quad \langle \mathsf{root} \mapsto \mathsf{?root},\ \mathsf{E} \rangle_{\mathrm{env}} \quad \langle \mathsf{tree}(\mathsf{?root})(\mathsf{T}),\ \mathsf{H} \rangle_{\mathrm{heap}} \quad \mathsf{C} \quad \rangle_{\mathrm{config}}$$

specifies all configurations in which program variable root points to a tree T in the heap

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

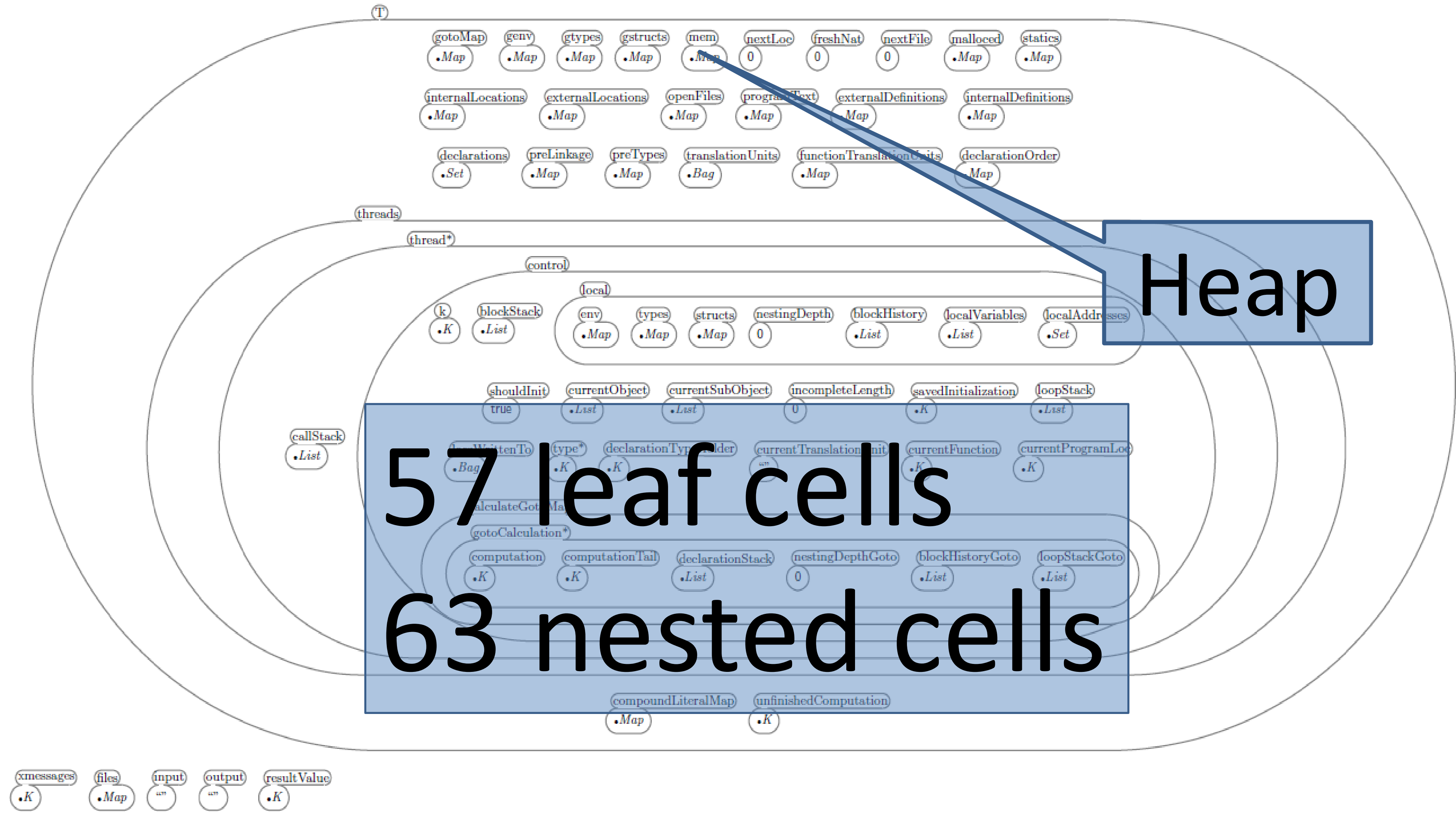
- Nothing special needs to be done for separation or for framing!

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

MODULE C-CONFIGURATION
IMPORTS C-SYNTAX
IMPORTS COMMON-C-CONFIGURATION
INITIAL CONFIGURATION:



END MODULE

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