Do We Really Need Axiomatic Semantics?

Grigore Rosu
University of Illinois, USA

Thesis

Axiomatic semantics (Hoare logic) unnecessary for program verification.

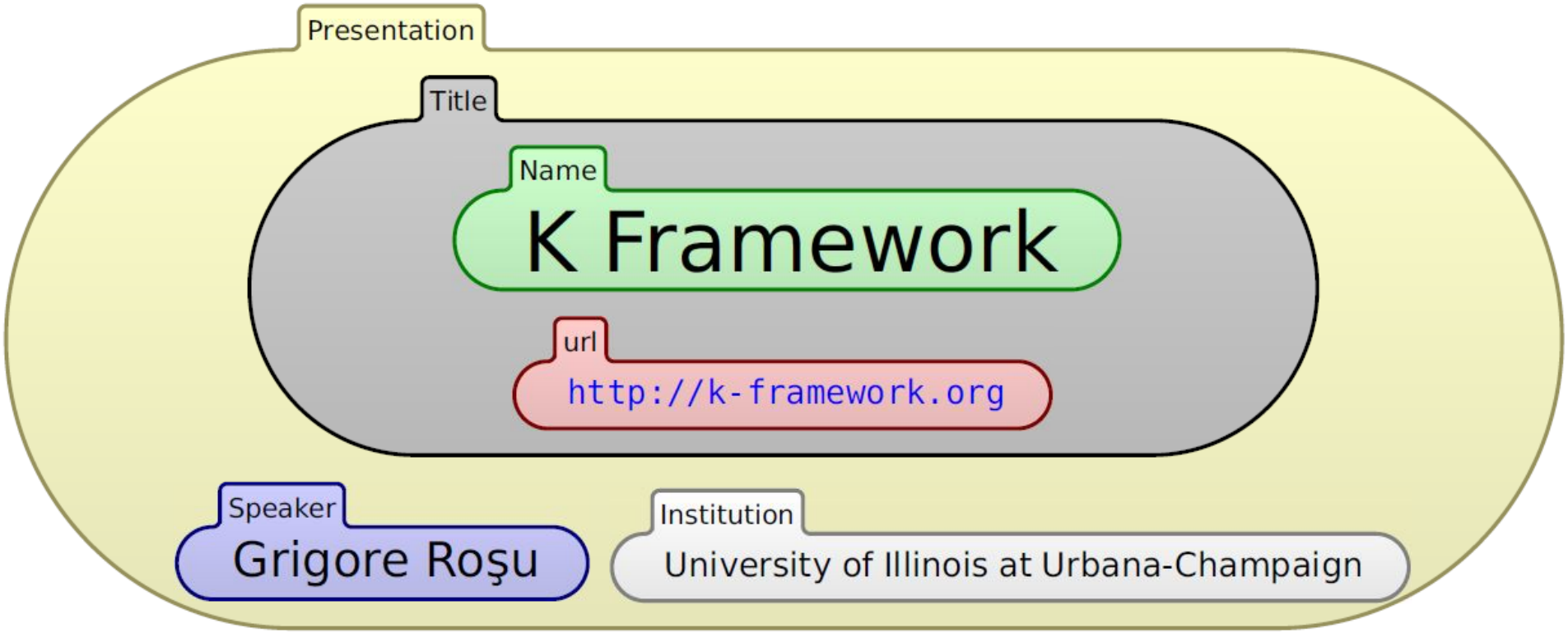
Operational semantics suffices.

- without paying any price!
- on the contrary, with advantages!

Overview

- Demo of MatchC
- K (operational semantics framework)
- Reachability Logic
  - Sound and complete language-independent proof system for reachability
  - Takes operational semantics as “axioms”

MatchC Demo



Joint project between
the FSL group at UIUC (USA) and
the FMSE group at UAIC (Romania)

K Team

 UIUC, USA

- **Grigore Rosu (started K in 2003)**
- Cansu Erdogan
- Dwight Guth
- David Lazar
- Patrick Meredith
- Andrei Stefanescu

Former members

- Kyle Blocher
- Peter Dinges
- Chucky Ellison
- Mike Ilseman
- Traian Serbanuta

 UAIC, Iasi, Romania

- **Dorel Lucanu**
- **Traian Serbanuta**
- Andrei Arusoae
- Denis Bogdanas
- Stefan Ciobaca
- Gheorghe Grigoras
- Radu Mereuta
- Raluca Necula
- Emilian Necula

Former Members

- Irina Asavoae
- Mihai Asavoae

Current State-of-the-Art in PL Design, Implementation and Analysis

Consider some programming language, L

- Formal semantics of L?
  - Typically skipped: considered expensive and useless
- Implementations for L
  - Based on some adhoc understanding of what L is
- 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
- …

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
    - Otherwise they are adhoc and likely wrong
- 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 want a semantic framework which makes it easy and fun to define programming languages, no matter how complex or large they are!

The K Framework

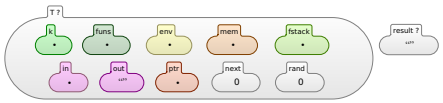
k-framework.org

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

Complete K Definition of KernelC

```
MODULE KERNELC-SYNTAX
  IMPORTS K-LATEX+PL-ID+PL-INT
  SYNTAX  Exp ::= Exp + Exp [strict]
                | DeclId
                | Id
                | Int
                | Exp - Exp [strict]
                | Exp ++
                | Exp == Exp [strict]
                | Exp != Exp [strict]
                | Exp <= Exp [strict]
                | Exp < Exp [strict]
                | Exp % Exp [strict]
                | ! Exp
                | Exp && Exp
                | Exp ? Exp : Exp
                | Exp || Exp
                | printf("%d;", Exp ) [strict]
                | scanf("%d",& Exp )
                | scanf("%d", Exp ) [strict]
                | NULL
                | PointerId
                | (int*)malloc( Exp *sizeof(int)) [strict]
                | free( Exp ) [strict]
                | * Exp [strict]
                | Exp [ Exp ]
                | Exp = Exp [strict(2)]
                | Id ( List{Exp} ) [strict(2)]
                | Id ( )
                | random()
                | srandom( Exp ) [strict]
  SYNTAX  Stmt ::= Exp ; [strict]
                | {}
                | { StmtList }
                | if( Exp ) Stmt
                | if( Exp ) Stmt else Stmt [strict(1)]
                | while( Exp ) Stmt
                | return Exp ; [strict]
                | DeclId List{DeclId} { StmtList }
                | #include< StmtList >
  SYNTAX  StmtList ::= StmtList StmtList
                | Stmt
  SYNTAX  Pgm ::= StmtList
  SYNTAX  Id ::= main
  SYNTAX  PointerId ::= * PointerId [ditto]
                | Id
  SYNTAX  DeclId ::= int Exp
                | void PointerId
  SYNTAX  StmtList ::= stdio.h
                | stdlib.h
  SYNTAX  List{Bottom} ::= List{Bottom} , List{Bottom} [assoc hybrid id: () strict]
                | ()
                | Bottom
  SYNTAX  List{PointerId} ::= List{PointerId} , List{PointerId} [ditto]
                | List{Bottom}
                | PointerId
  SYNTAX  List{DeclId} ::= List{DeclId} , List{DeclId} [ditto]
                | DeclId
                | List{Bottom}
  SYNTAX  List{Exp} ::= List{Exp} , List{Exp} [ditto]
                | Exp
                | List{DeclId}
                | List{PointerId}
END MODULE

MODULE KERNELC-DESUGARED-SYNTAX
  IMPORTS K-LATEX
  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  int * PointerId = int PointerId

  MACRO  #include< Stmts > = Stmts

  MACRO  E1 [ E2 ] = * E1 + E2

  MACRO  scanf("%d",& * E ) = scanf("%d", E )

  MACRO  int * PointerId = E = int PointerId = E

  MACRO  int X = E ; = int X ; X = E ;

  MACRO  stdio.h = {}

  MACRO  stdlib.h = {}

END MODULE
```

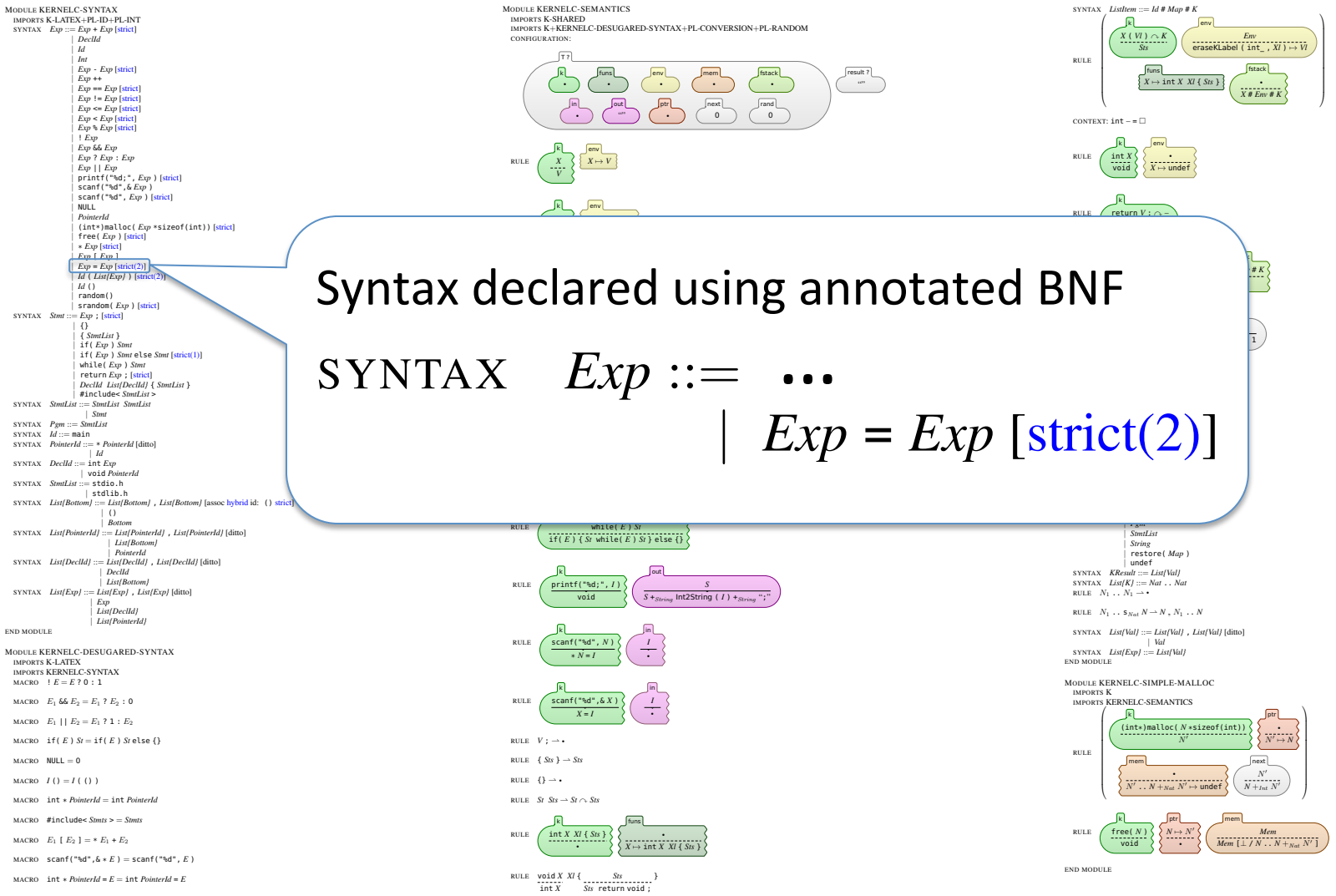
```
MODULE KERNELC-SEMANTICS
  IMPORTS K-SHARED
  IMPORTS K+KERNELC-DESUGARED-SYNTAX+PL-CONVERSION+PL-RANDOM
  CONFIGURATION:
```



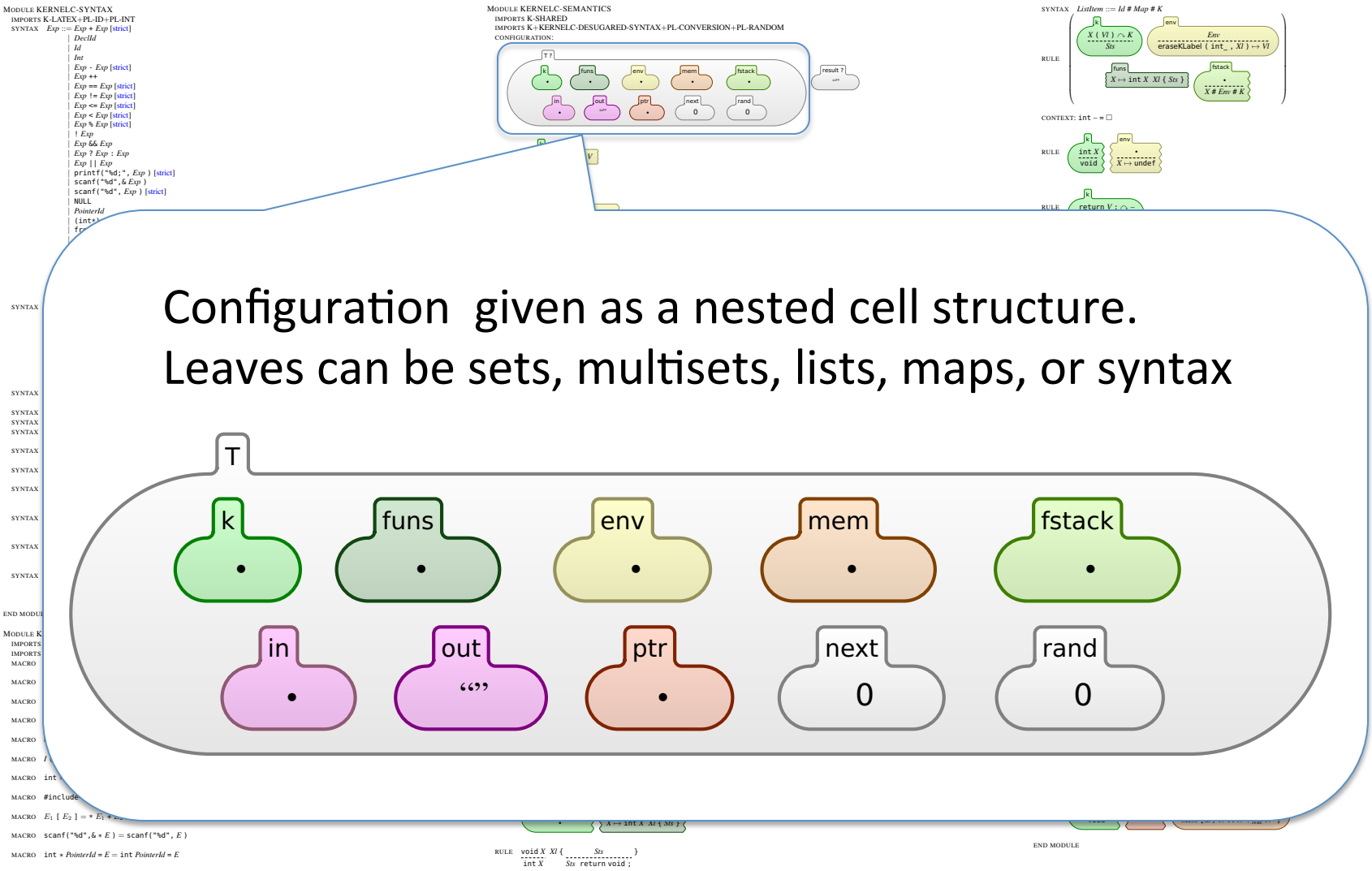

```
  RULE  X  V        env: X ↦ V

  RULE  X ++  I     env: X ↦ I  /  I +Int 1

  RULE  X = V  V    env: X ↦ _  /  V

  RULE  I1 + I2 ⇀ I1 +Int I2

  RULE  I1 - I2 ⇀ I1 −Int I2

  RULE  I1 % I2 ⇀ I1 %Int I2        when I2 !=Int 0

  RULE  I1 <= I2 ⇀ Bool2Int ( I1 ≤Int I2 )

  RULE  I1 < I2 ⇀ Bool2Int ( I1 <Int I2 )

  RULE  I1 == I2 ⇀ Bool2Int ( I1 ==Int I2 )

  RULE  I1 != I2 ⇀ Bool2Int ( I1 !=Int I2 )

  RULE  _?_:_ ⇀ if(_)_else_

  RULE  if( I ) _ else St ⇀ St       when I ==Int 0

  RULE  if( I ) St else _ ⇀ St       when ¬Bool I ==Int 0

  RULE  k: while( E ) St  /  if( E ) { St while( E ) St } else {}

  RULE  k: printf("%d;", I )  /  void        out: S  /  S +String Int2String ( I ) +String ";"

  RULE  k: scanf("%d", N )  /  * N = I        in: I  /  •

  RULE  k: scanf("%d",& X )  /  X = I         in: I  /  •

  RULE  V ; ⇀ •

  RULE  { Sts } ⇀ Sts

  RULE  {} ⇀ •

  RULE  St Sts ⇀ St ↷ Sts

  RULE  k: int X Xl { Sts }  /  •        funs: •  /  X ↦ int X Xl { Sts }

  RULE  void X Xl { Sts }  /  int X        Sts return void ;
```

```
  SYNTAX  ListItem ::= Id # Map # K

  RULE  k: X ( Vl ) ↷ K  /  Sts        env: Env  /  eraseKLabel ( int_ , Xl ) ↦ Vl
        funs: X ↦ int X Xl { Sts }        fstack: •  /  X # Env # K

  CONTEXT: int – = □

  RULE  k: int X  /  void        env: •  /  X ↦ undef

  RULE  k: return V ; ↷ –  /  V

  RULE  k: V ↷ •  /  K        env: –  /  Env        fstack: – # Env # K  /  •

  RULE  k: random()  /  randomRandom ( N' )        rand: N'  /  N' +Nat 1

  RULE  k: srandom( I )  /  void

  CONTEXT: * □ = –

  CONTEXT: * □ ++

  SYNTAX  Val ::= Int
                | void
  SYNTAX  Exp ::= Val
  SYNTAX  K ::= List{DeclId}
                | List{Exp}
                | List{PointerId}
                | Pgm
                | StmtList
                | String
                | restore( Map )
                | undef
  SYNTAX  KResult ::= List{Val}
  SYNTAX  List{K} ::= Nat .. Nat
  RULE  N1 .. N1 ⇀ •

  RULE  N1 .. sNat N ⇀ N , N1 .. N

  SYNTAX  List{Val} ::= List{Val} , List{Val} [ditto]
                | Val
  SYNTAX  List{Exp} ::= List{Val}
END MODULE

MODULE KERNELC-SIMPLE-MALLOC
  IMPORTS K
  IMPORTS KERNELC-SEMANTICS

  RULE  k: (int*)malloc( N *sizeof(int))  /  N'        ptr: •  /  N' ↦ N
        mem: •  /  N' .. N +Nat N' ↦ undef        next: N'  /  N' +Int N

  RULE  k: free( N )  /  void        ptr: N ↦ N'  /  •        mem: Mem  /  Mem [⊥ / N .. N +Nat N']

END MODULE
```

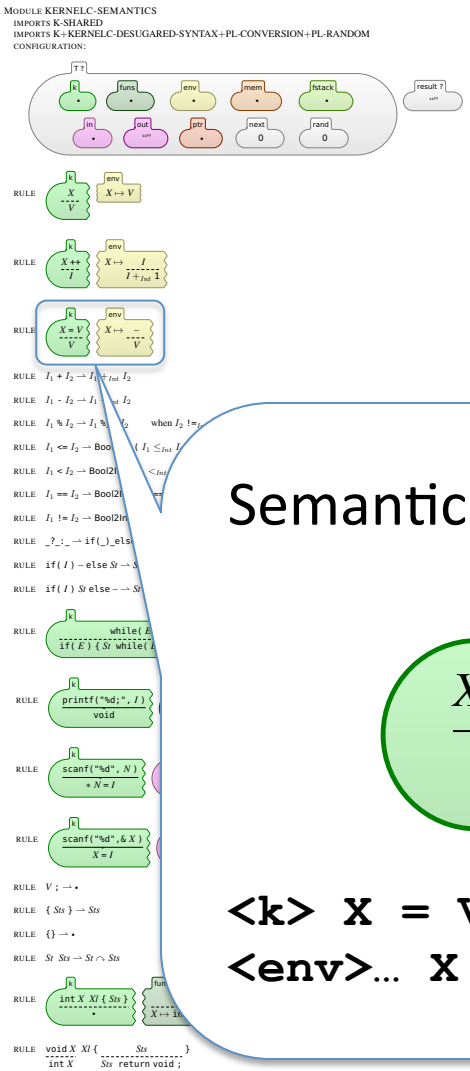
Complete K Definition of KernelC

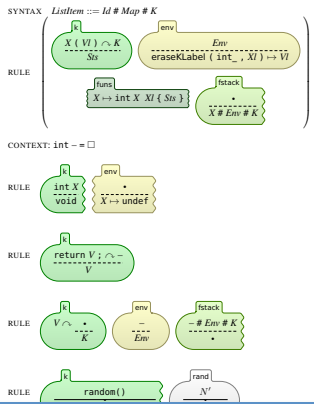


Complete K Definition of KernelC

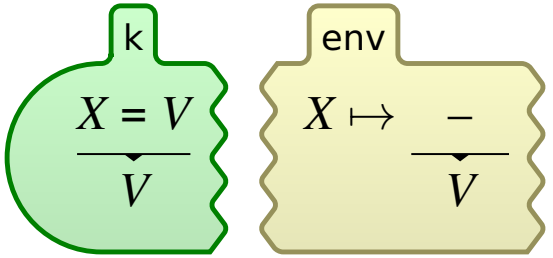


Complete K Definition of KernelC



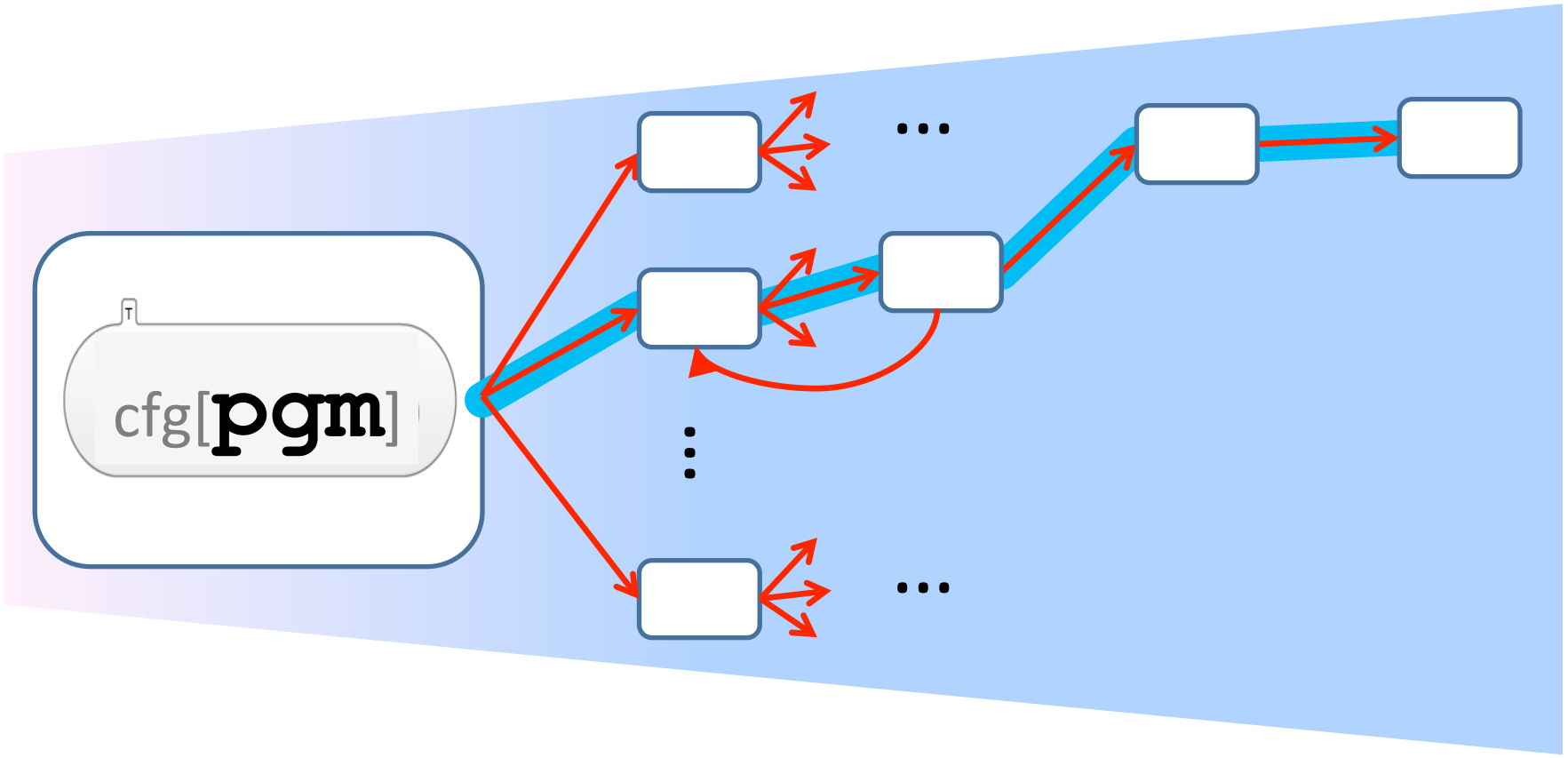


Semantic rules given contextually




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

What does the K Tool Offer?



Efficient and interactive execution (interpreters)
State-space exploration (search and model-checking)
Deductive program verification (in progress)

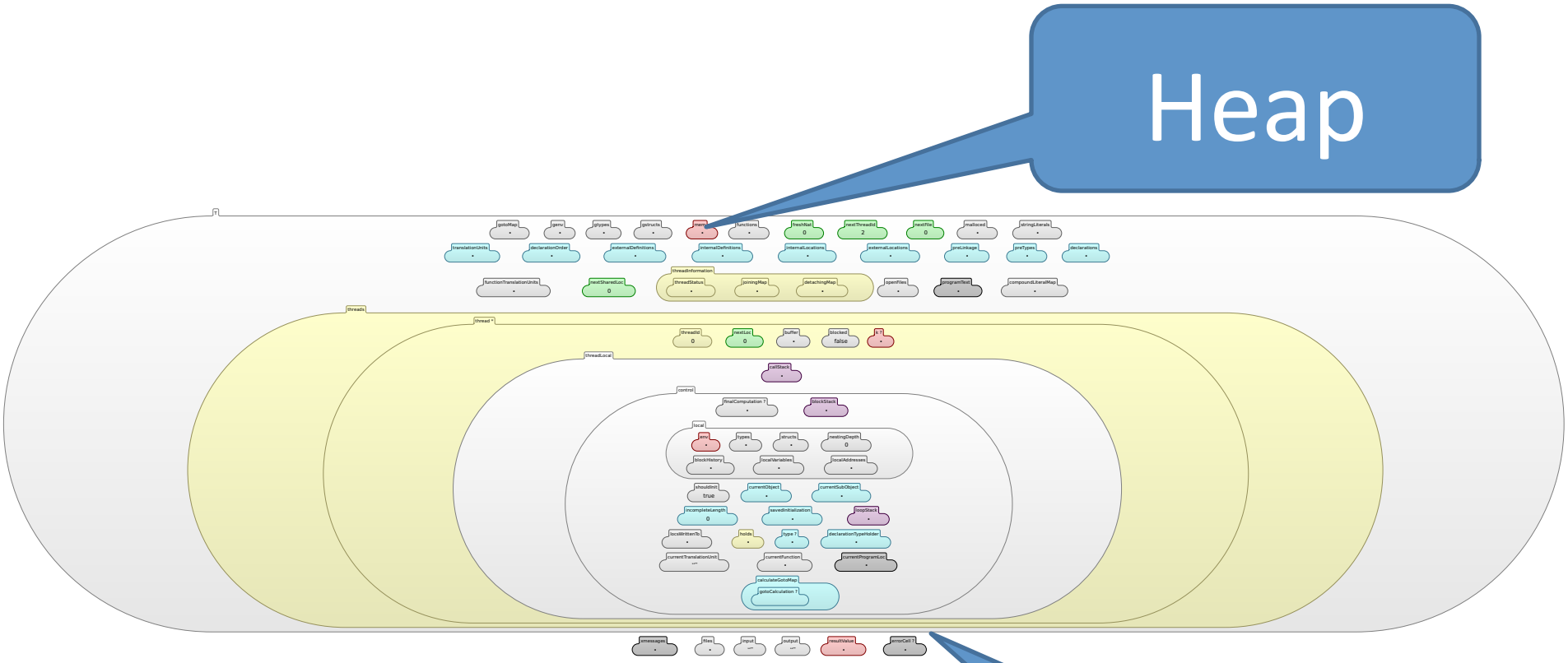
K Scales

Besides smaller and paradigmatic teaching languages, several larger languages were defined

- Java 1.4 : by Chen [CAV'06]
- Verilog : by Meredith&Katelman [MEMOCODE'10]
- C : by Chucky Ellison [POPL'12]

etc.

The K Configuration of C



Statistics for the C definition

- Total number of rules: **~1200**

- 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

Program Reasoning using Operational Semantics: -- Reachability Logic --

Reasoning About Programs

- Virtually all operational semantics can be defined with rewrite rules of the form

$$cfg \Rightarrow cfg' \text{ if } b \wedge cfg_1 \Rightarrow cfg'_1 \wedge b_1 \wedge \ldots \wedge cfg_n \Rightarrow cfg'_n \wedge b_n$$

- We would like to reason about programs using precisely such operational semantics!

Current State-of-the-Art

- Redefine the language using a different semantic approach (Hoare/separation/dynamic logic)
- Very language specific, error-prone; e.g.:

$$\frac{\mathcal{H} \vdash \{\psi \wedge \mathtt{e} \neq 0\}\, \mathtt{s}\, \{\psi\}}{\mathcal{H} \vdash \{\psi\}\, \mathtt{while(e)}\, \mathtt{s}\, \{\psi \wedge \mathtt{e} = 0\}}$$

$$\frac{\mathcal{H} \cup \{\psi\}\, \mathtt{proc()}\, \{\psi'\} \vdash \{\psi\}\, \mathtt{body}\, \{\psi'\}}{\mathcal{H} \vdash \{\psi\}\, \mathtt{proc()}\, \{\psi'\}}$$

Current State-of-the-Art

- Thus, these semantics need to be proved sound, sometimes also relatively complete, wrt trusted, operational semantics of the language
- Verification tools developed using them
- So we have an inherent gap between trusted, operational semantics, and the semantics currently used for program verification

Our Proposal: Matching Logic

- Use directly the trusted operational semantics!
  - Has been done before (ACL2), but proofs are low-level (induction on the transition system) and language-specific
- We propose a language-independent proof system based on matching logic, which
  - Takes operational semantics as axioms
  - Derives reachability properties
  - Is sound and relatively complete

Matching Logic

- Logic for specifying properties about program configurations and reasoning about them
  - Generalizes separation logic
- Key insight:
  - Configuration terms with variables are allowed to be used as predicates, called patterns!
  - Semantically, their satisfaction means matching

More Formally: Configurations

- For concreteness, assume configurations having the following syntax:

$$\langle\langle \ldots \rangle_{k}\ \langle \ldots \rangle_{env}\ \langle \ldots \rangle_{heap}\ \langle \ldots \rangle_{in}\ \langle \ldots \rangle_{out}\ \ \ldots \rangle_{cfg}$$

(matching logic works with any configurations)

- Examples of concrete (ground) configurations:

$$\langle\langle \texttt{x=*y; y=x; } \ldots \rangle_{k}\ \langle \texttt{x} \mapsto 7,\ \texttt{y} \mapsto 3,\ \ldots \rangle_{env}\ \langle 3 \mapsto 5 \rangle_{heap}\ \ \ldots \rangle_{cfg}$$

$$\langle\langle \texttt{x} \mapsto 3 \rangle_{env}\ \langle 3 \mapsto 5,\ 2 \mapsto 7 \rangle_{heap}\ \langle 1, 2, 3, \ldots \rangle_{in}\ \langle \ldots, 7, 8, 9 \rangle_{out}\ \ \ldots \rangle_{cfg}$$

More Formally: Patterns

- Concrete configurations are already patterns, but very simple ones, ground patterns
- Example of more complex pattern

$$\exists c : Cells,\ e : Env,\ p : Nat,\ \ i : Int,\ \sigma : Heap$$
$$\langle\langle \mathbf{x} \mapsto p,\ e\rangle_{\text{env}}\ \langle p \mapsto i,\ \sigma\rangle_{\text{heap}}\ c\rangle_{\text{cfg}}\ \wedge\ i > 0\ \wedge\ p \neq i$$

- Thus, patterns generalize both terms and [FOL]

More Formally: Reasoning

- We can now prove (using [FOL] reasoning) properties about configurations, such as

$$
\begin{aligned}
\models \quad & \forall c : Cell,\ e : Env,\ p : Nat \\
& \langle \langle \mathbf{x} \mapsto p,\ e \rangle_{\mathsf{env}}\ \langle p \mapsto 9 \rangle_{\mathsf{heap}}\ c \rangle_{\mathsf{cfg}}\ \wedge\ p > 10 \\
& \rightarrow \exists i : Int,\ \sigma : Heap \\
& \qquad \langle \langle \mathbf{x} \mapsto p,\ e \rangle_{\mathsf{env}}\ \langle p \mapsto i,\ \sigma \rangle_{\mathsf{heap}}\ c \rangle_{\mathsf{cfg}}\ \wedge\ i > 0\ \wedge\ p \neq i
\end{aligned}
$$

Reachability Rules

- "Rewrite" rules over matching logic formulae:

$$\varphi \Rightarrow \varphi'$$

(generalize to conditional rules)

- Since patterns generalize terms, matching logic reachability rules capture term rewriting rules
- Moreover, deals naturally with side conditions:

$l \Rightarrow r \text{ if } b$ turn into $l \wedge b \Rightarrow r$

Expressivity of Reachability Rules

- Capture operational semantics rules:

$$\langle\langle \mathtt{x} = i\,;\mathtt{s}\rangle_{\mathsf{k}}\ \langle \mathtt{x} \mapsto j,\ e\rangle_{\mathsf{env}}\ c\rangle_{\mathsf{cfg}}$$
$$\Rightarrow \langle\langle \mathtt{s}\rangle_{\mathsf{k}}\ \langle \mathtt{x} \mapsto i,\ e\rangle_{\mathsf{env}}\ c\rangle_{\mathsf{cfg}}$$

- Capture Hoare Triples: $\{\psi\}\,\mathsf{code}\,\{\psi'\}$

$$\exists X_{\mathsf{code}}(\langle \mathsf{code}, \sigma_{X_{\mathsf{code}}}\rangle \wedge \psi_X)$$
$$\Rightarrow \exists X_{\mathsf{code}}(\langle \mathtt{skip}, \sigma_{X_{\mathsf{code}}}\rangle \wedge \psi'_X)$$

Reachability Logic

- Language-independent proof system for deriving sequents of the form

$$\mathcal{A} \vdash_C \varphi \Rightarrow \varphi'$$

where A (axioms) and C (circularities) are sets of reachability rules

Proof System for Reachability

- Works with any operational semantics
  - Language independent
- Can prove anything that Hoare logic can
  - Plus more
- It is sound (partially correct) and relatively complete

$$\varphi \Rightarrow \varphi' \text{ if } \varphi_1 \Rightarrow \varphi_1' \wedge \cdots \wedge \varphi_n \Rightarrow \varphi_n' \in \mathcal{A}$$
$$\psi \text{ is a structureless pattern}$$

Axiom : $\dfrac{\mathcal{A} \cup C \vdash \varphi_1 \wedge \psi \Rightarrow \varphi_1' \quad \cdots \quad \mathcal{A} \cup C \vdash \varphi_n \wedge \psi \Rightarrow \varphi_n'}{\mathcal{A} \vdash_C \varphi \wedge \psi \Rightarrow \varphi' \wedge \psi}$

Reflexivity : $\mathcal{A} \vdash \varphi \Rightarrow \varphi$

Transitivity : $\dfrac{\mathcal{A} \vdash_C \varphi_1 \Rightarrow \varphi_2 \qquad \mathcal{A} \cup C \vdash \varphi_2 \Rightarrow \varphi_3}{\mathcal{A} \vdash_C \varphi_1 \Rightarrow \varphi_3}$

Consequence : $\dfrac{\models \varphi_1 \rightarrow \varphi_1' \qquad \mathcal{A} \vdash_C \varphi_1' \Rightarrow \varphi_2' \qquad \models \varphi_2' \rightarrow \varphi_2}{\mathcal{A} \vdash_C \varphi_1 \Rightarrow \varphi_2}$

Case Analysis : $\dfrac{\mathcal{A} \vdash_C \varphi_1 \Rightarrow \varphi \qquad \mathcal{A} \vdash_C \varphi_2 \Rightarrow \varphi}{\mathcal{A} \vdash_C \varphi_1 \vee \varphi_2 \Rightarrow \varphi}$

Abstraction : $\dfrac{\mathcal{A} \vdash_C \varphi \Rightarrow \varphi' \qquad \text{where } X \cap FV(\varphi') = \emptyset}{\mathcal{A} \vdash_C \exists X\, \varphi \Rightarrow \varphi'}$

Circularity : $\dfrac{\mathcal{A} \vdash_{C \cup \{\varphi \Rightarrow \varphi'\}} \varphi \Rightarrow \varphi'}{\mathcal{A} \vdash_C \varphi \Rightarrow \varphi'}$

Results

- Soundness (partial correctness) [ICALP'12]
- Generalizes Hoare logic for IMP [FM'12]
  - So also relatively complete for IMP
- Relatively complete in general [OOPSLA'12]
  - The fixed, language-independent proof system can prove any reachability property, including Hoare triples as special case, of any programming language

Conclusion

- Axiomatic semantics is unnecessary for program verification
- Operational semantics suffices
- Language-independent reachability proof system, which is sound and complete
  - Circularity generalizes the invariant rules