Matching Logic

- A New Program Verification Approach -

Grigore Rosu and **Andrei Stefanescu**

University of Illinois at Urbana-Champaign

Question

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

Floyd-Hoare logic

$$\{\pi_{\mathsf{pre}}\}\ \mathtt{code}\ \{\pi_{\mathsf{post}}\}$$

Question

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

Floyd-Hoare logic

?

$$\{\pi_{\text{pre}}\}\ \texttt{code}\ \{\pi_{\text{post}}\}$$

Limitations of Floyd-Hoare Logic

- 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

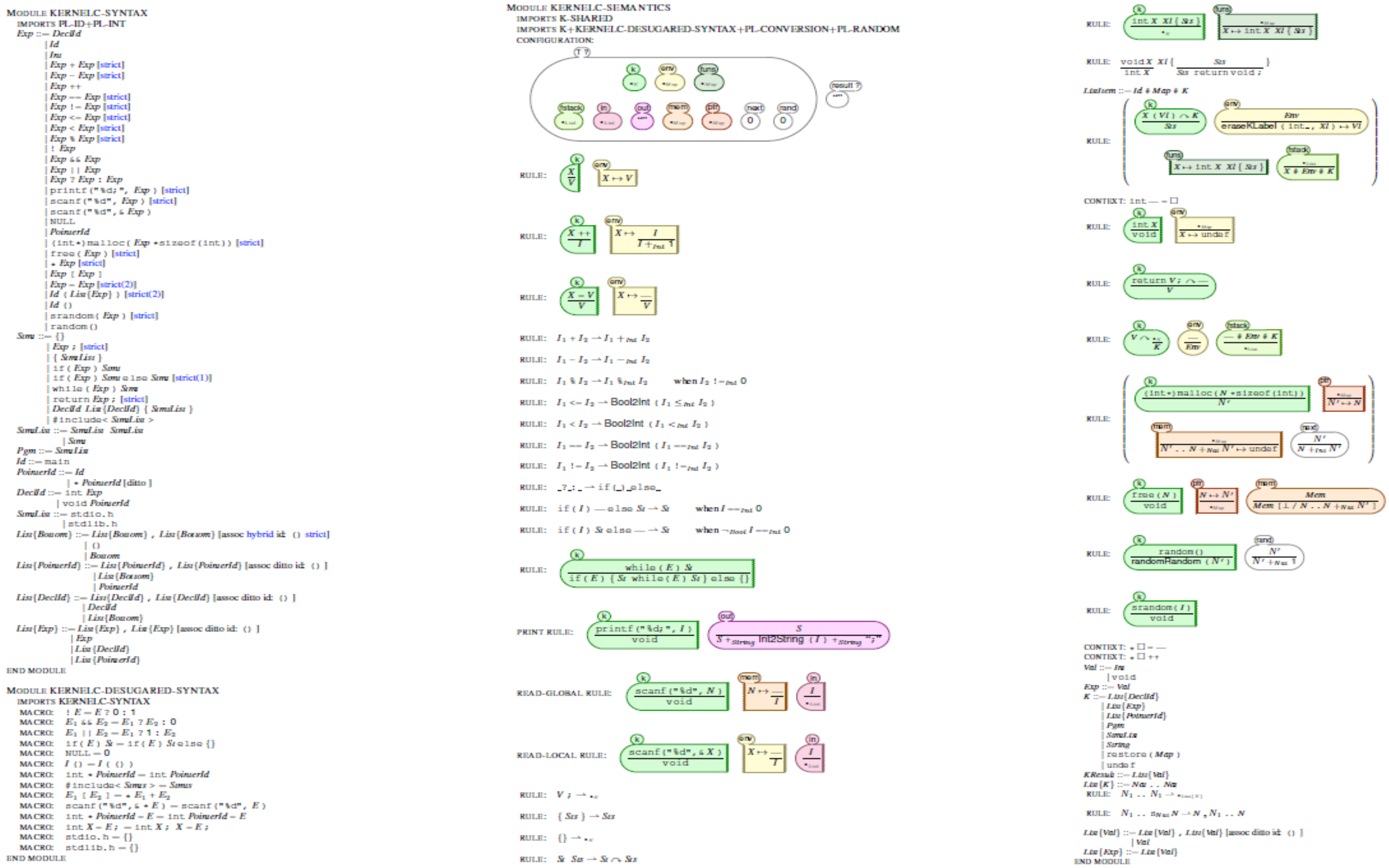
Ideal Program Logic

- Based on a formal *executable* semantics
  - So we can test it by executing 1000's of programs
  - Sound "by construction"
- Allows us to state any structural properties about configurations
  - Heap, stacks, input/output, etc.
  - Framing would be straightforward; nothing special
- Leads to immediate implementations of program verifiers, based on the executable semantics

Matching Logic

- A logic for reasoning about configurations
- Builds upon executable/operational semantics
    - Provides ground configurations and transitions
- Matching Logic
    - Formulae / Specifications
        - FOL over configurations with variables, called patterns
    - Models
        - Ground configurations
    - Satisfaction
        - Matching for configurations, plus FOL for the rest

Formal Executable Semantics of KernelC



MODULE KERNELC-SYNTAX
IMPORTS PL-ID+PL-INT
Exp ::= DeclId
| Id
| Int
| Exp + Exp [strict]
| 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) [strict]
| scanf("%d",& Exp)
| 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 ()
| srandom(Exp) [strict]
| random()
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 >
StmtList ::= StmtList StmtList
| Stmt
Pgm ::= StmtList
Id ::= main
PointerId ::= Id
| * PointerId [ditto]
DeclId ::= int Exp
| void PointerId
StmtList ::= stdio.h
| stdlib.h
List{Bottom} ::= List{Bottom} , List{Bottom} [assoc hybrid id: () strict]
| ()
| Bottom
List{PointerId} ::= List{PointerId} , List{PointerId} [assoc ditto id: ()]
| List{Bottom}
| PointerId
List{DeclId} ::= List{DeclId} , List{DeclId} [assoc ditto id: ()]
| DeclId
| List{Bottom}
List{Exp} ::= List{Exp} , List{Exp} [assoc ditto id: ()]
| Exp
| List{DeclId}
| List{PointerId}
END MODULE

MODULE KERNELC-DESUGARED-SYNTAX
IMPORTS KERNELC-SYNTAX
MACRO: $! E = E ? 0 : 1$
MACRO: $E_1 \&\& E_2 = E_1 ? E_2 : 0$
MACRO: $E_1 || E_2 = E_1 ? 1 : E_2$
MACRO: if(E) St = if(E) St else {}
MACRO: NULL = 0
MACRO: I () = I (())
MACRO: int * PointerId = int PointerId
MACRO: #include< Stmts > = Stmts
MACRO: $E_1 [E_2] = * E_1 + E_2$
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: k $\frac{X}{V}$ env $X \mapsto V$

RULE: k $\frac{X\ ++}{I}$ env $X \mapsto \frac{I}{I +_{Int} 1}$

RULE: k $\frac{X = V}{V}$ env $X \mapsto \frac{\_}{V}$

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

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

RULE: $I_1 \% I_2 \rightarrow I_1 \%_{Int} I_2$ when $I_2 \mathrel{!=_{Int}} 0$

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

RULE: $I_1 < I_2 \rightarrow$ Bool2Int $(I_1 <_{Int} I_2)$

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

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

RULE: _?_:_ → if(_)_else_

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

RULE: if(I) St else _ → St when $\neg_{Bool} I ==_{Int} 0$

RULE: k $\frac{\texttt{while(} E \texttt{) } St}{\texttt{if(} E \texttt{) \{ } St \texttt{ while(} E \texttt{) } St \texttt{ \} else \{\}}}$

PRINT RULE: k $\frac{\texttt{printf("\%d;", } I \texttt{)}}{\texttt{void}}$ out $\frac{S}{S +_{String} \text{Int2String} (I) +_{String} \text{";"}}$

READ-GLOBAL RULE: k $\frac{\texttt{scanf("\%d", } N \texttt{)}}{\texttt{void}}$ mem $N \mapsto \frac{\_}{I}$ in $\frac{I}{\cdot_{List}}$

READ-LOCAL RULE: k $\frac{\texttt{scanf("\%d",\& } X \texttt{)}}{\texttt{void}}$ env $X \mapsto \frac{\_}{I}$ in $\frac{I}{\cdot_{List}}$

RULE: $V ; \rightarrow \cdot_K$

RULE: $\{ Sts \} \rightarrow Sts$

RULE: $\{\} \rightarrow \cdot_K$

RULE: $St\ Sts \rightarrow St \curvearrowright Sts$

RULE: k $\frac{\texttt{int } X\ Xl \texttt{ \{ } Sts \texttt{ \}}}{\cdot_K}$ funs $\frac{\cdot_{Map}}{X \mapsto \texttt{int } X\ Xl \texttt{ \{ } Sts \texttt{ \}}}$

RULE: $\frac{\texttt{void } X}{\texttt{int } X}\ Xl \{ \frac{Sts}{Sts \texttt{ return void ;}} \}$

ListItem ::= Id # Map # K

RULE: k $\frac{X (Vl) \curvearrowright K}{Sts}$ env $\frac{Env}{\text{eraseKLabel} (\texttt{int\_}, Xl) \mapsto Vl}$ funs $X \mapsto \texttt{int } X\ Xl \texttt{ \{ } Sts \texttt{ \}}$ fstack $\frac{\cdot_{List}}{X \# Env \# K}$

CONTEXT: int _ = □

RULE: k $\frac{\texttt{int } X}{\texttt{void}}$ env $\frac{\cdot_{Map}}{X \mapsto \texttt{undef}}$

RULE: k $\frac{\texttt{return } V \texttt{;} \curvearrowright \_}{V}$

RULE: k $V \curvearrowright \frac{\cdot_K}{K}$ env $\frac{\_}{Env}$ fstack $\frac{\_ \# Env \# K}{\cdot_{List}}$

RULE: k $\frac{\texttt{(int*)malloc(} N \texttt{ *sizeof(int))}}{N'}$ ptr $\frac{\cdot_{Map}}{N' \mapsto N}$ mem $\frac{\cdot_{Map}}{N' .. N +_{Nat} N' \mapsto \texttt{undef}}$ next $\frac{N'}{N +_{Int} N'}$

RULE: k $\frac{\texttt{free(} N \texttt{)}}{\texttt{void}}$ ptr $\frac{N \mapsto N'}{\cdot_{Map}}$ mem $\frac{Mem}{Mem [\bot / N .. N +_{Nat} N']}$

RULE: k $\frac{\texttt{random()}}{\text{randomRandom} (N')}$ rand $\frac{N'}{N' +_{Nat} 1}$

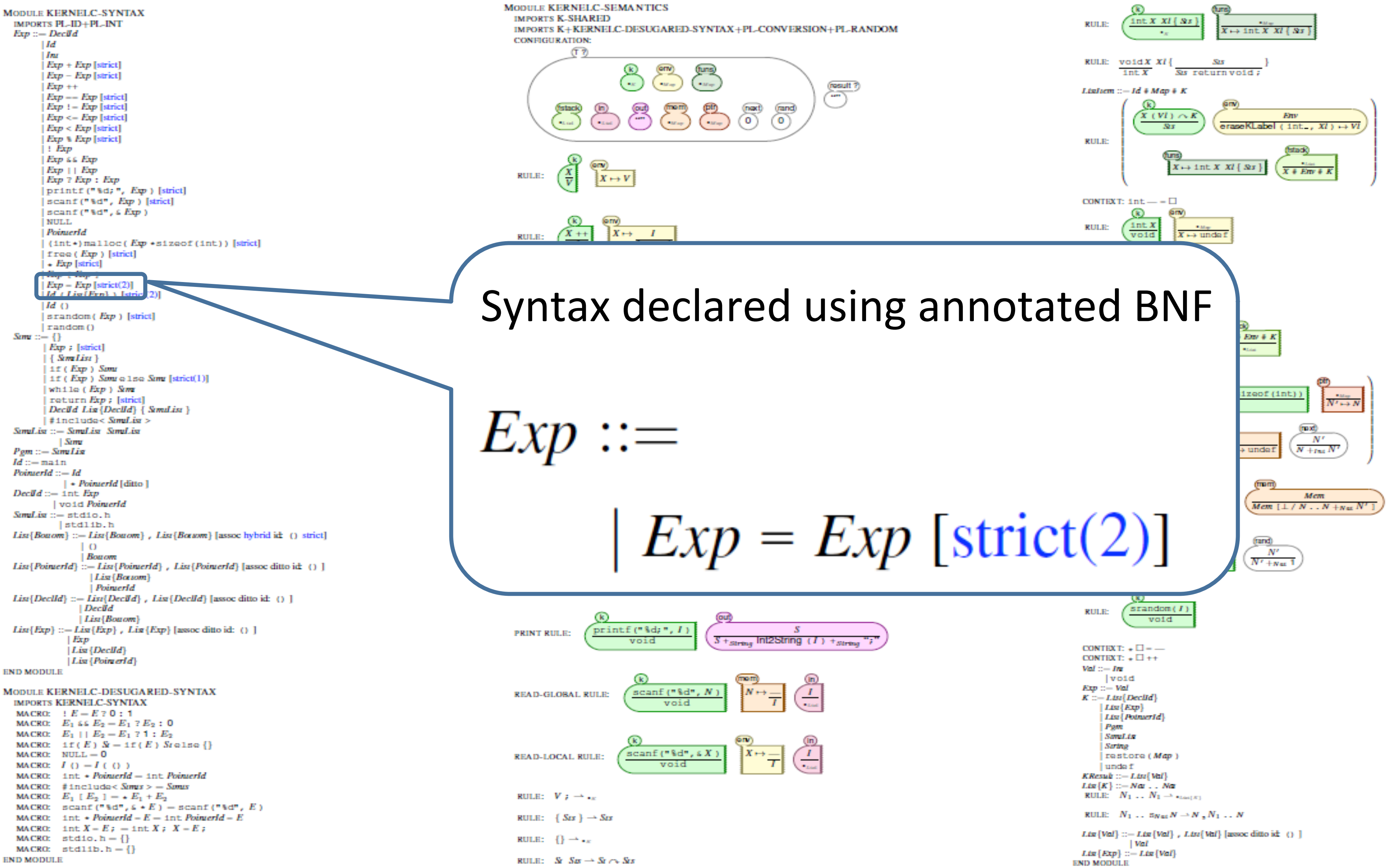
RULE: k $\frac{\texttt{srandom(} I \texttt{)}}{\texttt{void}}$

CONTEXT: * □ = _
CONTEXT: * □ ++
Val ::= Int
| void
Exp ::= Val
K ::= List{DeclId}
| List{Exp}
| List{PointerId}
| Pgm
| StmtList
| String
| restore(Map)
| undef
KResult ::= List{Val}
List{K} ::= Nat .. Nat
RULE: $N_1 .. N_1 \rightarrow \cdot_{List\{K\}}$

RULE: $N_1 .. s_{Nat} N \rightarrow N , N_1 .. N$

List{Val} ::= List{Val} , List{Val} [assoc ditto id: ()]
| Val
List{Exp} ::= List{Val}
END MODULE

Formal Executable Semantics of KernelC

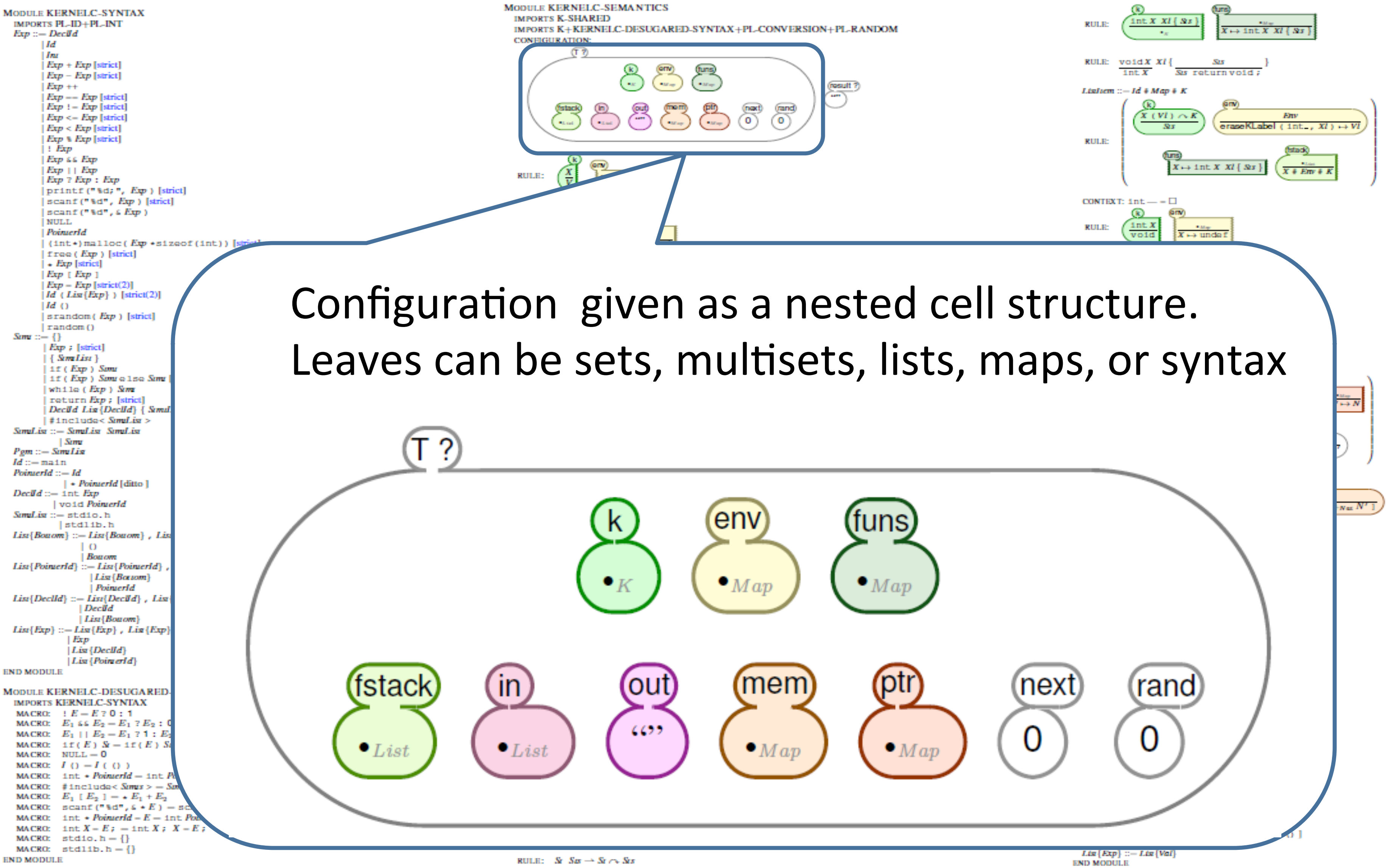


Syntax declared using annotated BNF

$$Exp ::=$$

$$\mid Exp = Exp\ [\textcolor{blue}{\text{strict}(2)}]$$

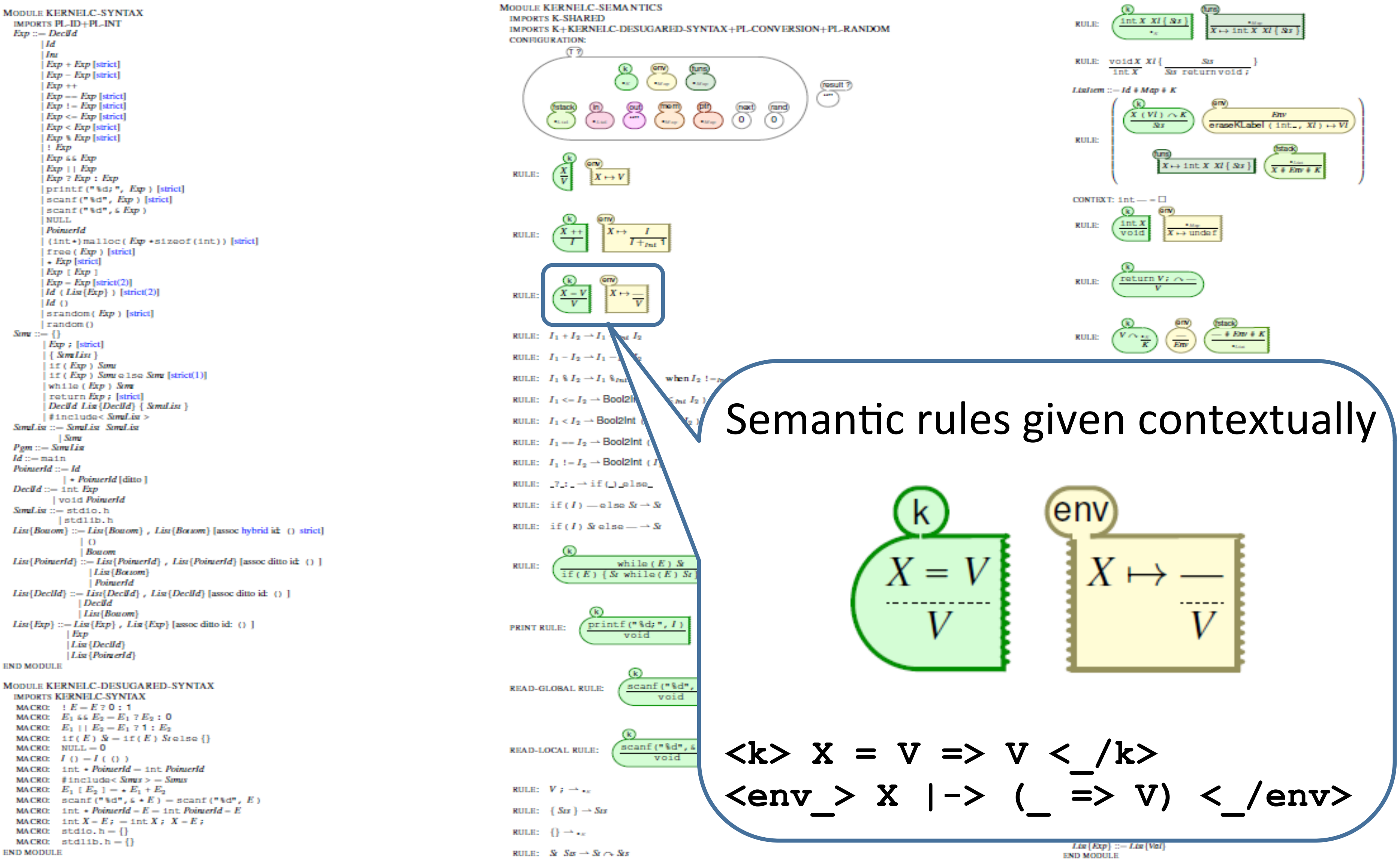
Formal Executable Semantics of KernelC



Configuration given as a nested cell structure.

Leaves can be sets, multisets, lists, maps, or syntax

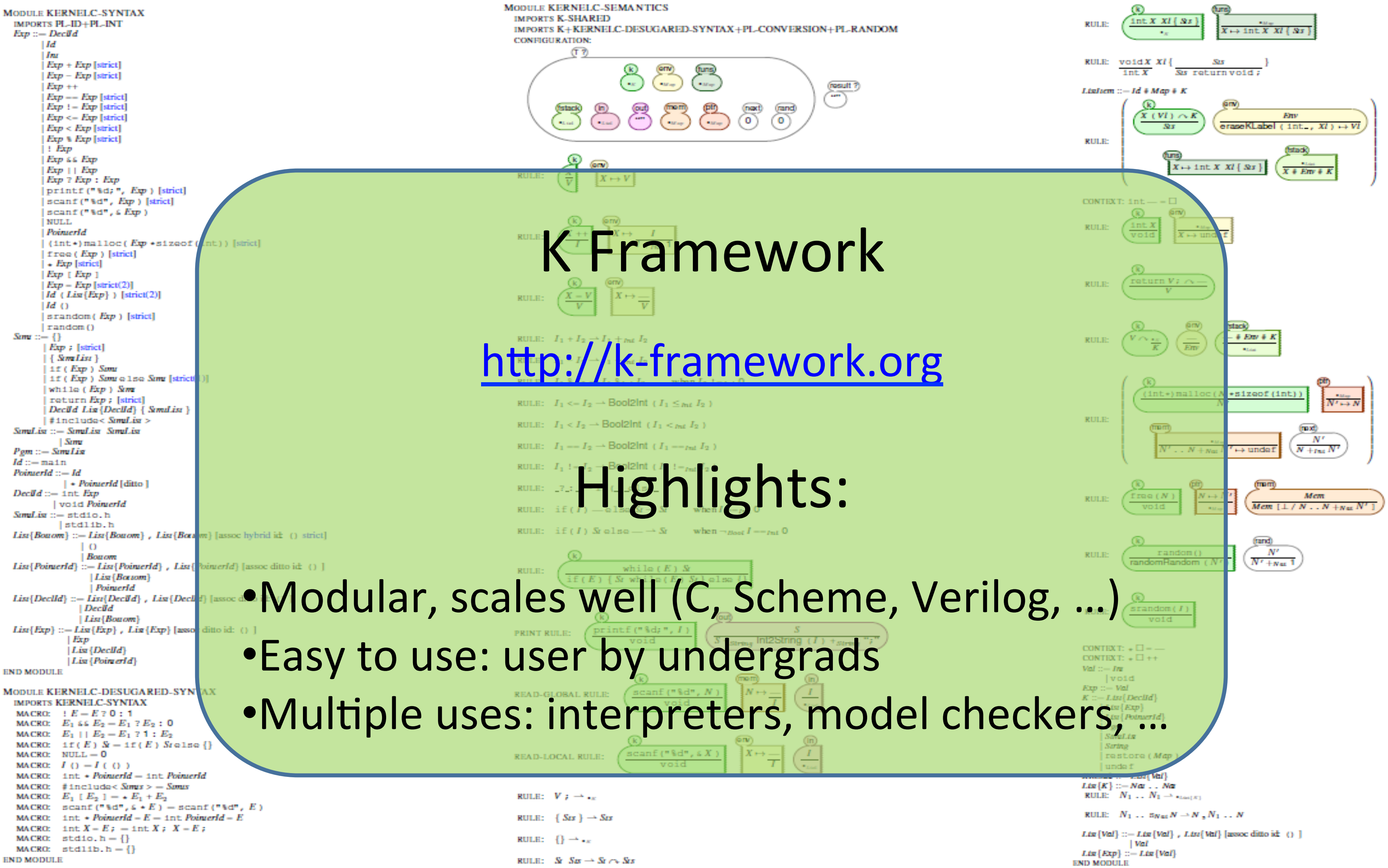
Formal Executable Semantics of KernelC



Semantic rules given contextually

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

Formal Executable Semantics of KernelC



K Framework

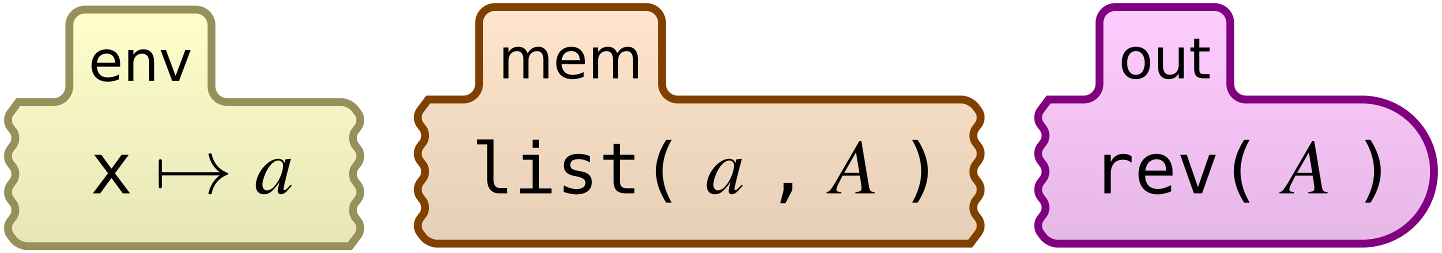
http://k-framework.org

Highlights:

- Modular, scales well (C, Scheme, Verilog, …)
- Easy to use: user by undergrads
- Multiple uses: interpreters, model checkers, …

Examples of Patterns

- x points to sequence *A* with |*A*|>1, and the reversed sequence rev(*A*) has been output



env

$x \mapsto a$

mem

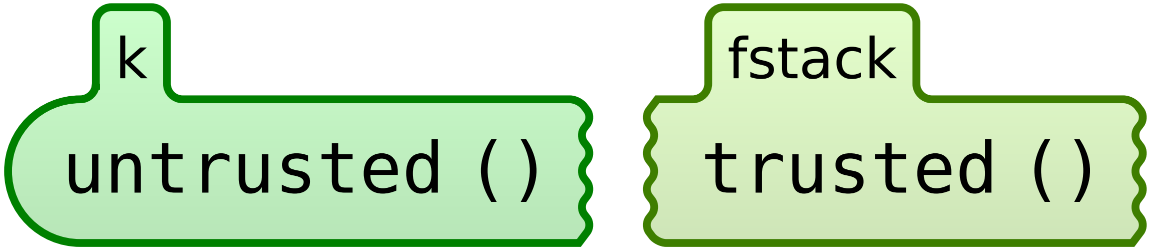
`list(` a `,` A `)`

out

`rev(` A `)`

$\wedge \ |A|>1$

- **`untrusted()`** can only be called from **`trusted()`**



k

`untrusted()`

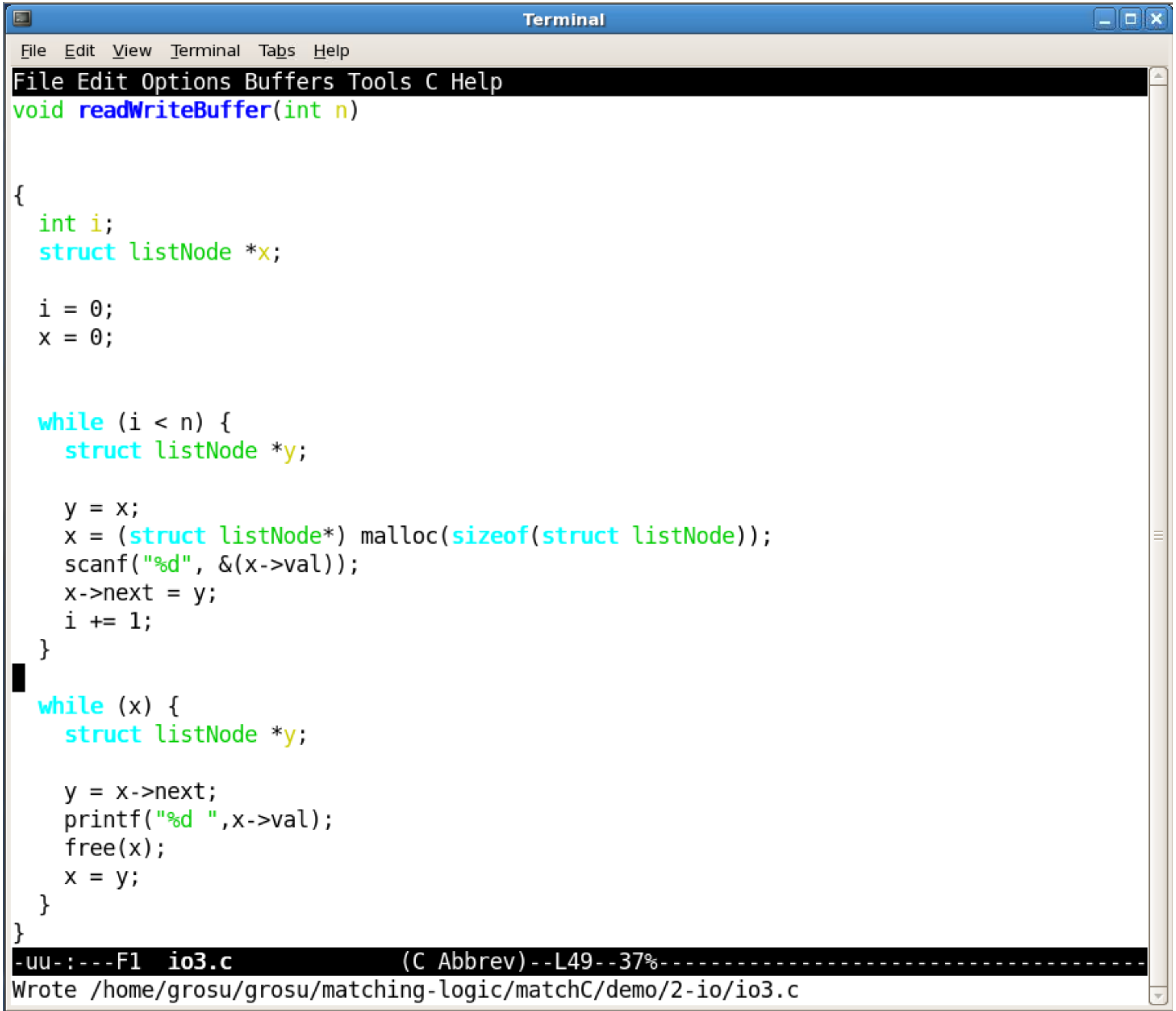
fstack

`trusted()`

Does it Really Work?

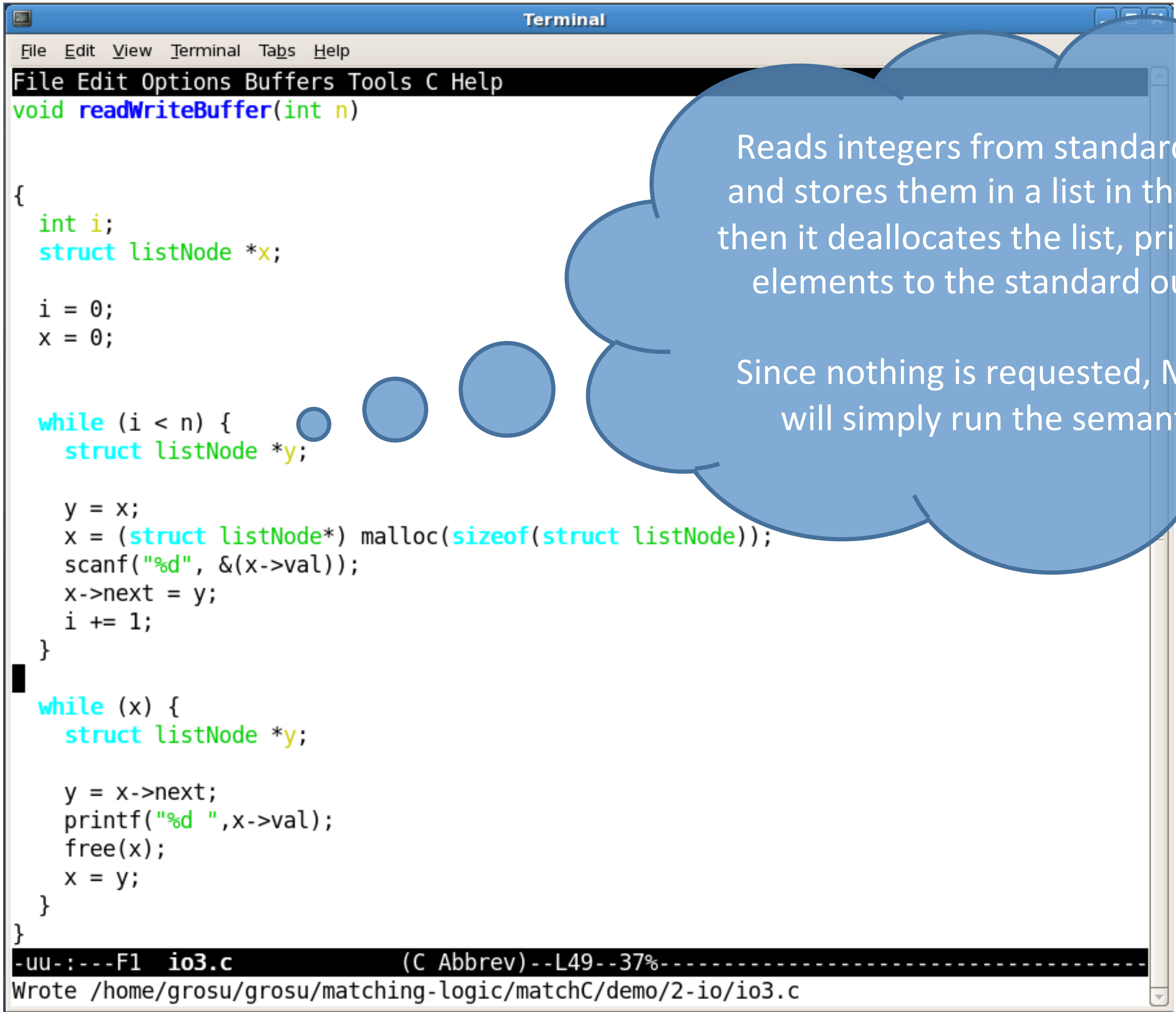
- We implemented a proof-of-concept matching logic verifier for a fragment of C
- Could verify all properties currently verifiable with existing separation logic based verifiers (for C-like languages)
- Could also verify properties that cannot be expressed in separation logic
- See Matching Logic poster tomorrow
  - Demo possible

MatchC at Work – Heap and I/O



```
void readWriteBuffer(int n)


{
  int i;
  struct listNode *x;

  i = 0;
  x = 0;


  while (i < n) {
    struct listNode *y;

    y = x;
    x = (struct listNode*) malloc(sizeof(struct listNode));
    scanf("%d", &(x->val));
    x->next = y;
    i += 1;
  }

  while (x) {
    struct listNode *y;

    y = x->next;
    printf("%d ",x->val);
    free(x);
    x = y;
  }
}
-uu-:---F1  io3.c          (C Abbrev)--L49--37%-----------------------------------
Wrote /home/grosu/grosu/matching-logic/matchC/demo/2-io/io3.c
```

MatchC at Work – Heap and I/O

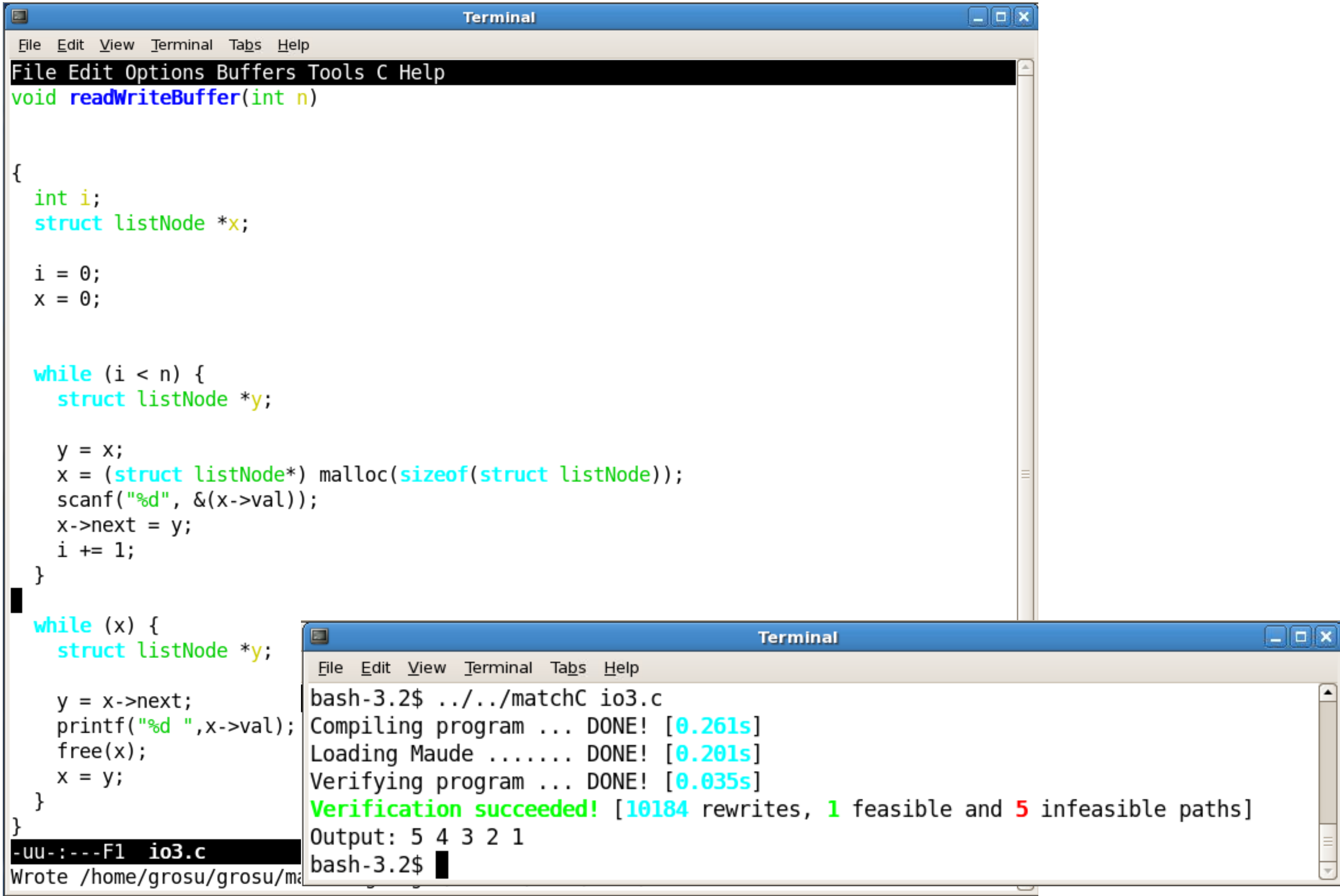


```
void readWriteBuffer(int n)


{
  int i;
  struct listNode *x;

  i = 0;
  x = 0;


  while (i < n) {
    struct listNode *y;

    y = x;
    x = (struct listNode*) malloc(sizeof(struct listNode));
    scanf("%d", &(x->val));
    x->next = y;
    i += 1;
  }

  while (x) {
    struct listNode *y;

    y = x->next;
    printf("%d ",x->val);
    free(x);
    x = y;
  }
}
-uu-:---F1  io3.c          (C Abbrev)--L49--37%------------------------------------
Wrote /home/grosu/grosu/matching-logic/matchC/demo/2-io/io3.c
```

Reads integers from standard input and stores them in a list in the heap; then it deallocates the list, printing its elements to the standard output.

Since nothing is requested, MatchC will simply run the semantics.

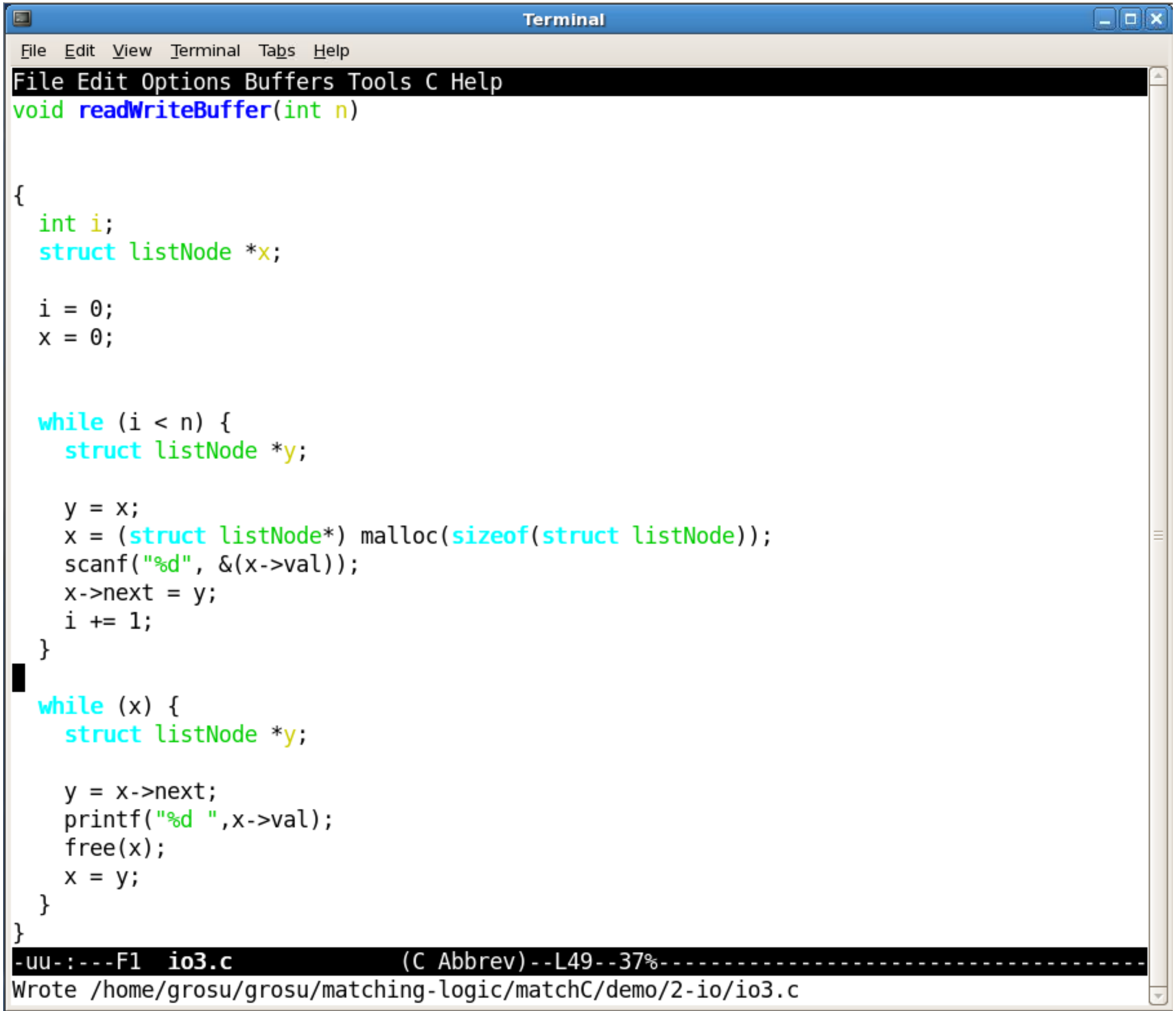
MatchC at Work – Heap and I/O



```
void readWriteBuffer(int n)


{
  int i;
  struct listNode *x;

  i = 0;
  x = 0;


  while (i < n) {
    struct listNode *y;

    y = x;
    x = (struct listNode*) malloc(sizeof(struct listNode));
    scanf("%d", &(x->val));
    x->next = y;
    i += 1;
  }

  while (x) {
    struct listNode *y;

    y = x->next;
    printf("%d ",x->val);
    free(x);
    x = y;
  }
}
-uu-:---F1  io3.c
Wrote /home/grosu/grosu/ma
```

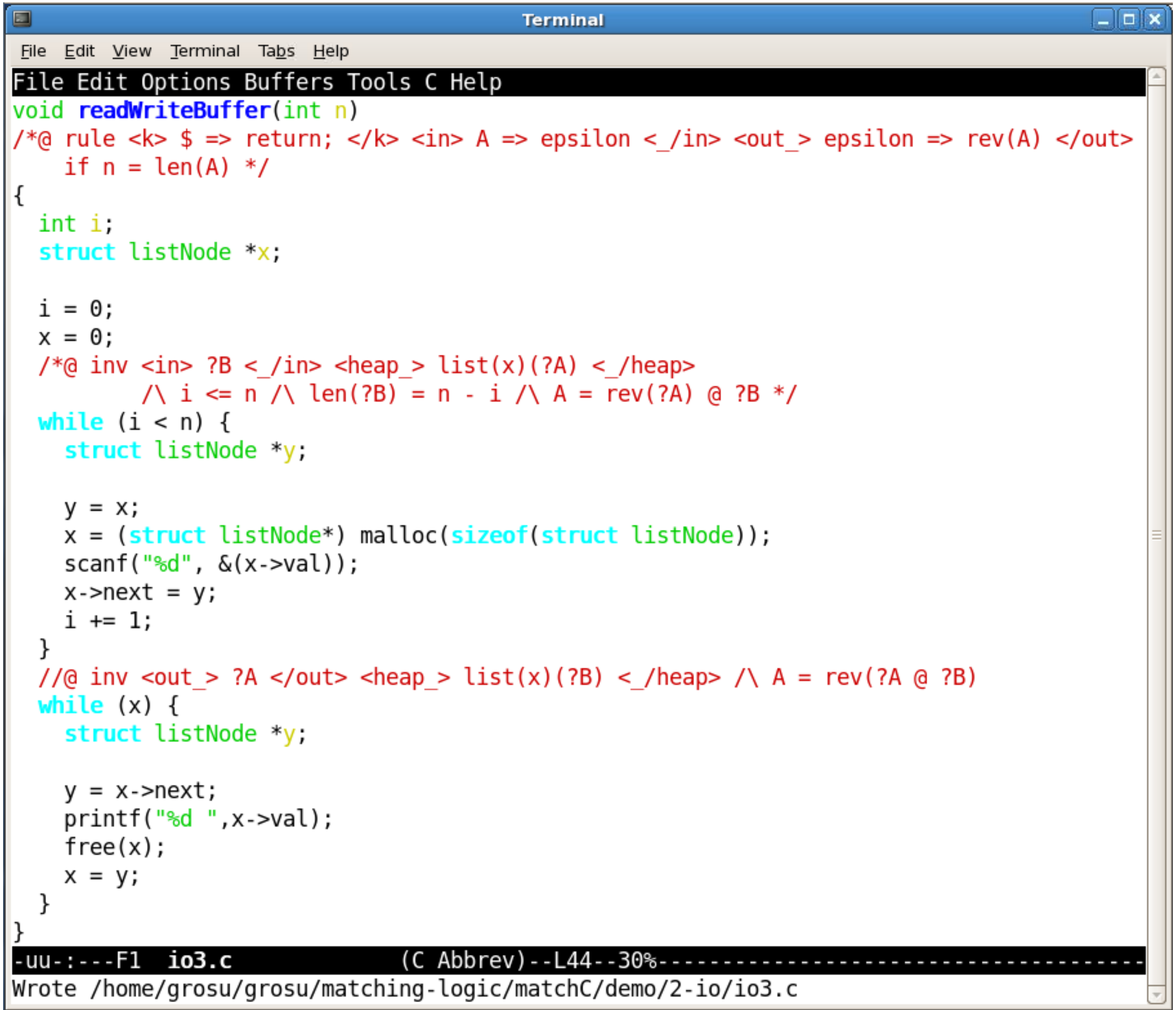
```
bash-3.2$ ../../matchC io3.c
Compiling program ... DONE! [0.261s]
Loading Maude ....... DONE! [0.201s]
Verifying program ... DONE! [0.035s]
Verification succeeded! [10184 rewrites, 1 feasible and 5 infeasible paths]
Output: 5 4 3 2 1
bash-3.2$
```

MatchC at Work – Heap and I/O



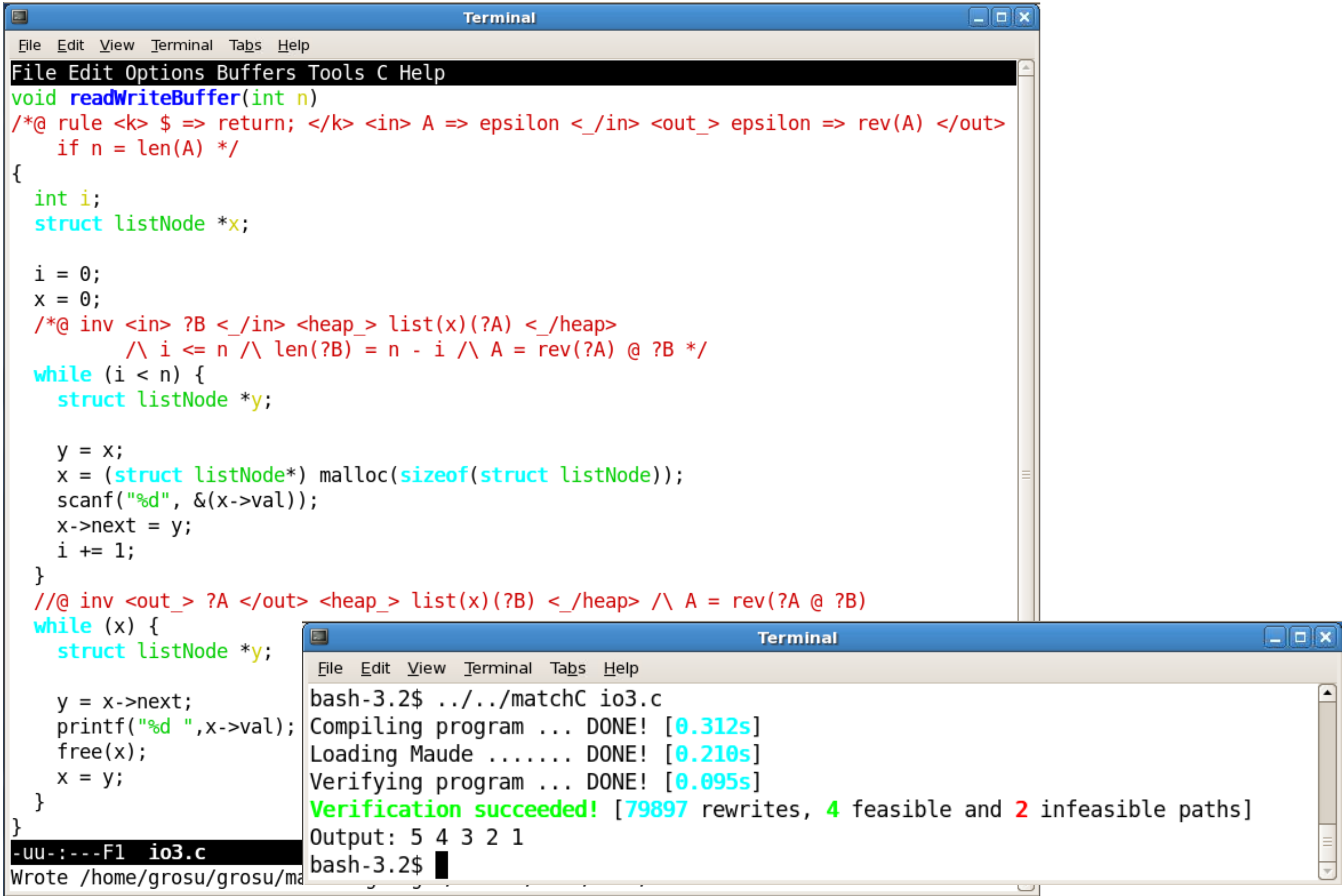
```
void readWriteBuffer(int n)


{
  int i;
  struct listNode *x;

  i = 0;
  x = 0;


  while (i < n) {
    struct listNode *y;

    y = x;
    x = (struct listNode*) malloc(sizeof(struct listNode));
    scanf("%d", &(x->val));
    x->next = y;
    i += 1;
  }

  while (x) {
    struct listNode *y;

    y = x->next;
    printf("%d ",x->val);
    free(x);
    x = y;
  }
}
-uu-:---F1  io3.c          (C Abbrev)--L49--37%-----------------------------------
Wrote /home/grosu/grosu/matching-logic/matchC/demo/2-io/io3.c
```

MatchC at Work – Heap and I/O



```
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);
    x = y;
  }
}
-uu-:---F1  io3.c            (C Abbrev)--L44--30%------------------------------------
Wrote /home/grosu/grosu/matching-logic/matchC/demo/2-io/io3.c
```

MatchC at Work – Heap and I/O



```
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);
    x = y;
  }
}
-uu-:---F1  io3.c
Wrote /home/grosu/grosu/ma
```

```
bash-3.2$ ../../matchC io3.c
Compiling program ... DONE! [0.312s]
Loading Maude ....... DONE! [0.210s]
Verifying program ... DONE! [0.095s]
Verification succeeded! [79897 rewrites, 4 feasible and 2 infeasible paths]
Output: 5 4 3 2 1
bash-3.2$
```

Conclusion

- Hoare Logic may not be the ultimate answer to the problem of program verification!
- In Matching Logic, we use an executable semantics of a language *as is* for verification
  - As opposed to redefining it as a Hoare logic
  - Executable semantics is *testable* and *reusable*
  - Giving an executable semantics is not necessarily painful, it can be fun if one uses the right tools