Deductive Evaluation: Formal Code Analysis with Low User Burden

Ben Di Vito
NASA Langley Research Center
Hampton, Virginia  USA
15 May 2016
• Formal code verification is enjoying a resurgence
  – Improved deduction (SMT solvers, primarily)
  – Recent tools: Frama-C, VCC, SPARK Pro (Ada)

• BUT:
  – Industry strongly prefers push-button methods
  – Code verifiers require effort
  – Will software engineers use them?

• Meanwhile, static analysis is fully automated
  – Many software developers have embraced them
  – But they only check well-formedness
Opportunities

• Can we automatically deduce functionality?
  – Yes! Discover, derive, infer code’s execution behavior
  – Forgo traditional verification results
  – Challenge: Iteration is hard

• Our method analyzes code having loops
  – Adaptation of classical Floyd-Hoare verification methods
  – Loop invariant synthesis using iteration schemes
  – Annotation-free deductive evaluation of C functions
  – More complete form of symbolic evaluation/execution
  – Mechanized using PVS (Prototype Verification System)
  – Best-effort analysis; no guarantee of coverage
Opportunities (cont’d)

• Data-driven approach relies on a division of labor
  – Human assistance to create iteration scheme library
  – Full automation when applying them during evaluation

• Ease of use is a major goal
  – Encourages uptake by software engineers
  – Provides rigorous feedback on user’s code
  – Augments existing tools and practices

• Filling a gap, finding a niche:

<table>
<thead>
<tr>
<th>SA</th>
<th>AI</th>
<th>DE</th>
<th>MC</th>
<th>PV</th>
<th>TP</th>
<th>Custom</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lightweight</td>
<td>Heavyweight</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example of Deductive Evaluation

C function:

```c
int add_mult(
    unsigned int m,
    int n)
{
    int p = 0;
    unsigned int i = 0;
    while (i < m) {
        p += n;
        i++;
    }
    return p;
}
```

Evaluation result (PVS):

```pvs
add_mult_deval
[(IMPORTING
   iter_schemes@prog_types)
   m_0_: nat,
   n_0_: int] : THEORY
BEGIN
   . . .
   final: return_values =
       (# result_ :=
         m_0_ * n_0_ #)
   WFO: boolean = TRUE
END add_mult_deval
```
IMPORING iter_schemes@top

% Invariants for loop index i
% (scheme loop_index_recur):
%   (index_var_expr . i_1_ = k_1_)
%   (iter_k_expr . k_1_ = i_1_)
%   (initial_bound . TRUE)
%   (final_bound . i_1_ < 1 + m_0_)

% Invariants for variable p
% (scheme arith_series_recur):
%   p_1_ = (k_1_ * n_0_)

% Values of dynamic variables on
% (normal) loop exit:
%   k_2_ : nat = m_0_
%   i_3_ : nat = m_0_
%   p_3_ : int = m_0_ * n_0_

% End of for/while loop at depth 1.

IMPORING iter_schemes@top

p_0_ : int = 0
i_0_ : nat = 0
result_0_ : int
return_values: TYPE = [# result_: int #]

% Analyzing while loop at depth 1.
% Found dynamic variables: p, i
% Found static variables: m, n
% Found possible index variables: i

% Values at top of loop:
% k_1_ : nat % implicit loop index
% p_1_ : int % dynamic variable
% i_1_ : nat % dynamic variable

% Effects of loop body:
% p_2_ : int = p_1_ + n_0_
% i_2_ : nat = i_1_ + 1

% End of for/while loop at depth 1.
Features of PVS

• PVS (by SRI International) is both a language and a suite of deduction tools
  – Classical higher order logic with typing
  – Powerful interactive theorem prover
  – Prover also can be invoked programmatically
  – Tools hosted within the Emacs editor

• Relevant language features
  – Declarations grouped into parameterized theories
  – **Predicate subtypes** are crucial: \( \{ x : T \mid P(x) \} \)
  – Function types are versatile; used to model arrays:
    \[ \text{below}(n) \to \text{int} \]

• Uninterpreted constants model program values
  – Example: \( n_1 : \{n : \text{int} \mid 0 \leq n \text{ AND } n < q \} \)
C Features Supported

• Current fragment of C is modest
  – Types int, unsigned int and arrays of int
  – Function declarations and most statements
  – Function parameter mechanism

• Limitations and unsupported features
  – Integer types are unbounded
  – No side effects in expressions
  – No parameter aliasing (e.g., overlapping arrays)
  – No pointers (yet)
  – No declarations other than functions
Prototype Tool Chain

Evaluator, Synthesizer: Common Lisp
AST Translator: Python
C Parser: Open-source tool (Python)
Emacs Interface: Emacs Lisp
Invariant Concepts

• Non-iterative code segments can be analyzed via:
  – Predicate transformation
  – Proof rules from a program logic (e.g., Hoare logic)
  – Symbolic evaluation/expression

• Invariants are needed to capture loop behavior
  – In verification tools, normally provided by users
  – Generally considered a tedious, error-prone activity

• Typical proof rule for while-loop:
  – Given: \( P \rightarrow Q \land \{B \land Q\} S \{Q\} \land Q \rightarrow (R \lor B) \)
  – Infer: \( \{P\} \text{ while } B \text{ do } S \{R\} \)

• Derivation of invariants is undecidable in general
  – Use tractable domains, heuristics or predefined schemes
Analysis Approach

• Invariant synthesis based on recurrence relations
  – Generalized for predicates
  – Iteration schemes expressed as PVS theories
  – Templates and patterns derived from theories
  – Applied during analysis using matching and proving

• Deductive evaluation of C code
  – Based on Floyd-Hoare verification concepts
  – No verification conditions
  – Instead, perform on-the-fly analysis and proof
  – Predicate subtypes play a key role
  – Iteration schemes are searched, invariants are derived
  – Fully automatic, strongest-postcondition analysis
Predicate Recurrence Relations

- Schemes formalize generalized recurrence relations
  - Recurrence: \( I(u,0): u = 1; \ R(u,v,k): v = 2^u \)
  - Solution: \( P(u,k): u = 2^k \)
  - Prove: \( I(u,0) \rightarrow P(u,0); \ P(u,k) \land R(u,v,k) \rightarrow P(v,k+1) \)
  - Enables solutions to be Boolean expressions

- PVS formulation uses structured predicate definition
  - Labeled conditions and solution components
  - Implicit loop index \( k \) used in every scheme
  - Optional declaration for auxiliary facts
  - Inductive proof that solution satisfies recurrence
  - Meta-model expressed in separate theories
arith_series_recur : THEORY
BEGIN
    dyn_vars: TYPE = int
    stat_vars: TYPE = int
    IMPORTING recur_pred_defn[dyn_vars, stat_vars]
    k: VAR nat
    I,U,V: VAR dyn_vars
    S,W: VAR stat_vars
    recur_type: recurrence_type = var_function

    solution(I, S)(U, k): invar_list = . . .

    recur_satis: LEMMA sat_recur_rel(solution, recurrence)
END arith_series_recur
Example Scheme 1 (cont’d)

arth_series_recur : THEORY

.
.
.

recurrence(I, S)(U, V, k): recur_cond =

LET s0 = I, d = S, u = U, v = V IN

(# each := (: (iter_effect, v = u + d) :),
 once := (: :)
 #)

.
.

solution(I, S)(U, k): invar_list =

LET s0 = I, d = S, u = U IN

(: (func_val_expr, u = k * d + s0),
 (initial_bound,
  IF d < 0 THEN u <= s0 ELSE u >= s0 ENDIF)
 :)

.
.

END arith_series_recur
Example Scheme 2

```plaintext
loop_index_recur : THEORY

. . .

dyn_vars:  TYPE = int
stat_vars:  TYPE = [nzint, int, real_rel]

. . .

recurrence(I, S)(U, V, k): recur_cond =

  LET i0 = I, (d, n, R) = S, i = U, v = V IN
  (# each := (: (iter_effect, v = i + d),
   (while_cond, R(i, n)) :),
  once := (: (dyn_init, R(i0, n + d)),
   (stat_cond,
    R = reals.< OR R = reals.>) :) )

  #)

. . .

END loop_index_recur
```
Example Scheme 2 (cont’d)

solution(I, S)(U, k): invar_list =
  LET i0 = I, (d, n, R) = S, i = U IN
  (: (index_var_expr,
       i = id(LAMBDA (k: nat): k * d + i0)(k)),
  (iter_k_expr,
       k = id(LAMBDA (i: int): (i - i0) / d)(i)),
  (initial_bound,
       IF d < 0 THEN i <= i0 ELSE i0 <= i ENDIF),
  (final_bound,
       R(i0, n + d) IMPLIES R(i, n + d)) :)

facts(I, S)(U, k): aux_fact_list =
  LET i0 = I, (d, n, R) = S, i = U IN
  (: (final_index_value,
       R(0, d) AND NOT R(i, n) IMPLIES
       i = n + mod(i0 - n, d)),
  (final_k_value,
       R(0, d) AND NOT R(i, n) IMPLIES
       k = ceiling((n - i0) / d)) :)

Evaluator Operation

• Deductive evaluator accepts C in intermediate form
  – ASTs rendered as Lisp s-expressions

• Evaluator processes C statements within a function
  – Process is similar to symbolic execution
  – Handles extra paths due to {if, return, break} statements
  – PVS theory built incrementally during evaluation
  – PVS constants model C variables at change points
  – Predicate subtypes used to express constraints

• Loop handler finds invariants for dynamic variables
  – Iteration schemes searched
  – Matching applied to effects of loop body
  – Prover checks conditions and performs simplification
  – Final variable values at end of loop are derived
  – Schemes can depend on invariants found earlier
Evaluation Example 2

C function:

```c
int add_mult_exp(
    unsigned int m, int n) {
    int p = 0;
    unsigned int d = m;
    int y = n;
    while (d > 0) {
        if (d % 2 == 1)
            p += y;
        y += y;
        d /= 2;
    }
    return p;
}
```

Evaluation result (PVS):

```plaintext
... ...
% Invariants for variable d
% (scheme div2_exp2_recur):
%   d_1_ = floor((m_0_ / (2 ^ k_1_)))

% Invariants for variable y
% (scheme double_exp2_recur):
%   y_1_ = (n_0_ * (2 ^ k_1_))

% Invariants for variable p
% (scheme exp2_mult_recur):
%   p_1_ = m_0_ * n_0_ -
%       floor((m_0_ / (2 ^ k_1_)))
%       * (2 ^ k_1_) * n_0_

... ...
```
Array Handling

• Array indexing leads to well-formedness concerns
  – Ensure that index expressions are within bounds
  – Two declaration cases in C: (1) `int A[N]` and (2) `int A[]`
  – For (1), check that `i < N` (well-formedness condition, WFC)
  – For (2), add an implicit size parameter, then generate a well-formedness obligation (WFO) to ensure `i < size`

• Invariants help constrain array accesses within loops
  – When `i < n` for all iterations, can generate WFO: `n <= size`
  – Special schemes are provided to establish the bounds
  – WFOs must be enforced in the calling environment
C function:

```c
void array_init(
    int A[],
    unsigned int n,
    int v
)
{
    unsigned int i;
    for (i=0; i<n; i++)
        A[i] = v;
}
```

Evaluation result (PVS):

```plaintext
array_init_deval
[(IMPORTING
    iter_schemes@prog_types)
    A_size_: posnat,
    A_0_: int_array(A_size_),
    n_0_: nat, v_0_: int ] : THEORY
BEGIN
    . . .
    val_A: {r_: int_array(A_size_) |
        FORALL (q: below(n_0_)):
            r_(q) = v_0_}
    final: return_values =
        (# A := val_A #)
    WFO: boolean = n_0_ <= A_size_
END array_init_deval
```
Conditional Loop Exits

• Loops can be exited via return and break statements
  – Give rise to additional exit paths

• In some contexts, loop exits can induce invariants
  – When exit condition is P, can often infer “not P” holds at the top of every iteration
  – One sufficient condition is that the loop index is the only dynamic variable P references
  – Allows us to conclude the following:
    – FORALL (j: below(k)): NOT P(j)
  – An iteration scheme is provided to handle this case
C function:

```c
int linear_search(
    const int A[],
    unsigned int n,
    int v) {
    int i = 0;
    while (i < n) {
        if (A[i] == v)
            return i;
        i += 1;
    }
    return -1;
}
```

Evaluation result (PVS):

```plaintext
linear_search_deval
[IMPOR TED iter_schemes@prog_types)
A_size_: posnat,
A_0_: int_array(A_size_),
n_0_: nat, v_0_: int] : THEORY
BEGIN
    ...
    val_result_: {r_: int |
        (((r_ = -(1)) AND
            (FORALL (j: below(n_0_)):
                NOT A_0_(j) = v_0_)) OR
            (A_0_(r_) = v_0_ AND
                (r_ < n_0_) AND (0 <= r_) AND
                (FORALL (j: below(r_)):
                    NOT A_0_(j) = v_0_)))
    final: return_values =
        (# result_ := val_result_ #)
    WFO: boolean = n_0_ <= A_size_
END linear_search_deval
```
Nested Loops

- Inner loop completed first
  - Outer loop evaluation encounters inner loop on main path within body
  - Inner loop is processed independently, resulting in derived effects
  - Those effects used to match a scheme for outer loop
  - Inferred invariants for outer loop reflect combined behavior

C function:

```c
void bubble_sort(
    int A[],
    unsigned int nml) {
    unsigned int i, j;
    int t;
    for (i=0; i<nml; i++) {
        for (j=i+1; j<1+nml; j++) {
            if (A[j] < A[i]) {
                t = A[i];
                A[i] = A[j];
                A[j] = t;
            }
        }
    }
}
```
Evaluation Example 5

Evaluation result (PVS):

\[
\text{bubble_sort_deval} \\
\text{[(IMPORTING iter_schemes@prog_types)]} \\
\text{A_size_: posnat,} \\
\text{A_0_: int_array(A_size_),} \\
\text{nm1_0_: nat] : THEORY} \\
\text{BEGIN} \\
\text{. . .} \\
\text{A_6_:} \\
\text{{A: int_array(A_size_) |} } \\
\text{(FORALL} \\
\text{(p: below((nm1_0_ - i_1_)):} \\
\text{(A(i_1_) <= A(1 + p + i_1_)))} \\
\text{AND permutation_of?(A, A_1_) AND} \\
\text{(FORALL (p: below(A_size_)):} \\
\text{((p < i_1_) OR (nm1_0_ < p)) IMPLIES A(p) = A_1_(p))}} \\
\text{. . .} \\
\text{val_A:} \\
\text{{r_: int_array(A_size_) |} } \\
\text{((FORALL (p: below(nm1_0_)):} \\
\text{(r_(p) <= r_(1 + p))) AND} \\
\text{permutation_of?(r_, A_0_))}} \\
\text{final: return_values =} \\
\text{(# A := val_A #) } \\
\text{WFO: boolean =} \\
\text{1 + nm1_0_ <= A_size_} \\
\text{END bubble_sort_deval}
\]
Inferring End-to-End Behavior

- Example: Lossless data compression
  
  ```c
  void data_comp(const int A[1000],
                  unsigned int n, int C[1000]) {
    int B[1000];
    unsigned int m;
    m = compress(n, A, B);   /* B’s format derived */
    decompress(m, B, C); }
  ```

- Try to evaluate `decompress` in context
- Two possible techniques:
  - Expand the function `decompress` in-line and evaluate
  - Set the type of formal parameter `B` in `decompress` to match constraint produced by evaluation of `compress`

- Expected inference is that `C = A`
Limitations

• Current prototype
  – Subset of C supported; no other languages yet
  – Small scale, slow performance
  – Matching is syntactic; canonical forms help
  – Too many TCCs (type correctness conditions) spawned
  – Need multi-pass evaluation for full treatment
  – NASA PVS libraries can help

• Overall method
  – Could support verification tools; not addressed yet
  – Synthesize PVS functions to mitigate code complexity
  – Need to populate iteration scheme library (> 1K ?)
  – Large scheme library is a design challenge for tools
Potential Uses, Outlook

• **Usage possibilities**
  - Development aid, symbolic debugging
  - Complement to unit testing
  - Reverse engineering of source code
  - Analyzer for component libraries, specialized software domains
  - Synthesis of invariants for verifiers and other tools

• **Future outlook**
  - Promising, but much work lies ahead
  - Could benefit from:
    - Tighter PVS integration
    - Data mining to help create iteration schemes
    - Use of SMT solvers and computer algebra systems
    - Integration with IDEs
  - Concepts should be portable to other theorem provers
Questions?

Ben Di Vito
NASA Langley Research Center
Hampton, Virginia 23681  USA
b.divito@nasa.gov
+1-757-864-4883