

# Toward Robust Verification of PEG Parser Interpreters<sup>2</sup>

N. Shankar

Computer Science Laboratory  
SRI International  
Menlo Park, CA

Mar 6, 2024

---

<sup>2</sup>This work was supported by DARPA under agreement number HR001119C0075. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government. Joint work with Zephyr Lucas (Dartmouth).

- The number and complexity of data and file formats is growing by the day.
- Bad or missing input validation of the input in these formats is a major source of security vulnerabilities.
- The Parsley data definition language is a way of specifying such language formats based on
  - ➊ Parsing Expression Grammars (PEGs) as the basic grammar definition
  - ➋ Attribute grammars for capturing data dependencies using inherited and synthesized attributes over PEG grammars
  - ➌ A functional language for defining computations and constraints.
- The talk focuses on the PEG aspect of Parsley with a verification and proof of, and code extraction from, a parsing abstract machine for PEGs.

# Managing the Proof Lifecycle

- The 1992 CADE paper introducing PVS captures the motivation for many of the design choices in PVS for supporting the scalable construction of robust proofs:

*Our experience with mechanical verification of complex designs and algorithms has led us to conclude that, just as with software, there is a lifecycle to a mechanically-checked proof. In the initial exploratory phase of proof development, we are mainly interested in debugging the specification and putative theorems, and in testing and revising the key, high-level ideas in the proof. An important requirement in this phase is early and useful feedback when a purported theorem is, in fact, false. Once the basic intuitions have been acquired and the formalization is stable, the proof checking enters a development phase where we take care of the details and construct the proof in larger leaps. Efficiency of proof development is a key requirement here. In the third, presentation phase, the proof is honed and polished for presentation in order to be scrutinized by the social process. Readability and intellectual perspicuity of the output is the goal here. The final phase is generalization where we carefully analyze the finished proof, weaken and generalize the assumptions, extract the key insights and proof techniques, and make it easier to carry out subsequent verifications of a similar nature. Maintenance is a special application of generalization, where we adapt a verification to slightly changed assumptions or requirements. Robustness of the proof procedure is a useful attribute here.*

# The PVS Language in Brief

- A PVS specification is a collection of libraries.
  - Each library is a collection of files.
  - Each file is a sequence of theories.
  - Each theory is a sequence of declarations/definitions of types, constants, and formulas (Boolean expressions).
- Types include
  - 1 Booleans, number types
  - 2 Predicate subtypes:  $\{x : T \mid p(x)\}$  for type  $T$  and predicate  $p$ .
  - 3 Dependent function  $[x : D \rightarrow R(x)]$ , tuple  $[x : T_1, T_2(x)]$ , and record  $[\#a : T_1, b : T_2(x)\#]$  types.
  - 4 Algebraic and coalgebraic datatypes: lists, trees, ordinals.
- Expressions in PVS are
  - 1 Booleans, numbers
  - 2 Application :  $f(a_1, \dots, a_n)$
  - 3 Abstraction :  $\lambda(x_1 : T_1, \dots, x_n : T_n) : a$
  - 4 Tuples:  $(a_1, \dots, a_n), a'3$
  - 5 Records:  $(\#l_1 := a_1, \dots, l_n := a_n\#), a'l_i$
  - 6 Conditionals: IF  $a_1$  THEN  $a_2$  ELSE  $a_3$  ENDIF
  - 7 Updates:  $a$  WITH  $[(3)'1'age := 37]$ .

- Quantification and equality over higher types are not executable, but many definitions are.
- Mapping PVS to Common Lisp is relatively direct: each construct is mapped to its Lisp counterpart and types are transformed in to compiler declarations.
- Array, record, tuple updates are a bit challenging: need to minimize copying by performing destructive updates.
- PVS2CL uses a static analysis to identify whether a reference is live in the context of its update.<sup>3</sup>
- *Code generation is different from implementation — can't create a bespoke runtime.*
- *A (principled) specification logic like PVS is not a (pragmatic) programming language.*
- *Executable specifications are very useful for animation, testing, validation, proof, and system building.*

---

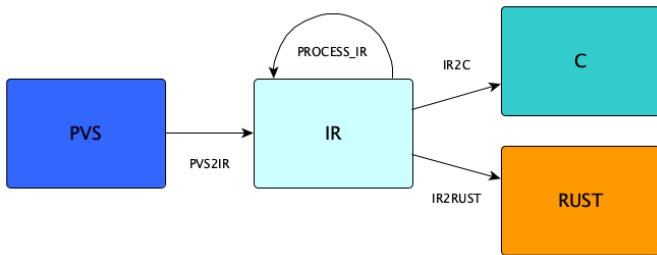
<sup>3</sup>Shankar, N.: Static analysis for safe destructive updates in a functional language, LOPSTR 2001.

# Functional vs. Imperative

- Functional and imperative languages are quite different, both syntactically and semantically.
- With functional languages
  - Referentially transparency is a key property
  - Both manual and mechanical verification are more civilized
  - Evaluation is pure (updates are nondestructive)
  - Memory management is implicit.
- With imperative languages
  - Scoping is tricky: No counterpart of let-expressions in many imperative languages
  - Evaluation is triggered by control flow not value flow
  - Semantically closer to machine execution
  - Integrated with extensive libraries and application software
  - Memory management may be explicit, e.g., through `free` and `malloc`.

# The PVS2C Code Generator

- PVS2C generates safe, efficient, standalone C code for a full functional fragment of PVS.
- Each PVS theory `foo.pvs` generates a `foo.h` and `foo.c`.<sup>4</sup>
- The translation is factored through an intermediate language that represents PVS expressions in A-normal form and performs a light static analysis to identify the *release points* for references.



<sup>4</sup>Férey, G., Sh\_, N.: Code Generation using a formal model of reference counting, NFM 2016. See also Wolfram Schulte, Deriving reference count garbage collectors, *6th International Symposium on Programming Language Implementation and Logic Programming*, 1994.

# PVS2C: Putting Theory to Use

- The theoretical outline above deals with an idealized first-order language with (unbounded) integers and arrays.
- The full PVS2C implementation covers
  - 1 Multi-precision rational numbers and integers, and floats
  - 2 Fixed-size arithmetic: uint8, uint16, uint32, uint64, int8, int16, int32, int64, with safe casting
  - 3 Dependent (dynamically sized) and infinite arrays
  - 4 Dependent records and tuples
  - 5 Higher-order functions and closures (with updates)
  - 6 Characters (ASCII and Unicode) and strings
  - 7 Algebraic datatypes
  - 8 Parametric theories with type parameters (unboxed polymorphism)
  - 9 Memory-mapped File I/O
  - 10 Semantic attachments
  - 11 JSON representation for data
- PVS2C captures a functional subset of PVS that is usable as a safe programming language - a well-typed program cannot fail (modulo resource limitations).



# Experiments in Code Generation: HMAC from Wikipedia

```
function hmac is
  input:
    key:      Bytes    // Array of bytes
    message:  Bytes    // Array of bytes to be hashed
    hash:     Function // The hash function to use (e.g. SHA-1)
    blockSize: Integer // The block size of the hash function
                                //(e.g. 64 bytes for SHA-1)
    outputSize: Integer // The output size of the hash function
                                //(e.g. 20 bytes for SHA-1)

  // Keys longer than blockSize are shortened by hashing them
  if (length(key) > blockSize) then
    key <- hash(key) // key is outputSize bytes long

  // Keys shorter than blockSize are padded to blockSize by padding
  //with zeros on the right
  if (length(key) < blockSize) then
    key <- Pad(key, blockSize) // Pad key with zeros to make it
                                // blockSize bytes long
  o_key_pad <- key xor [0x5c * blockSize] // Outer padded key
  i_key_pad <- key xor [0x36 * blockSize] // Inner padded key
  return hash(o_key_pad || hash(i_key_pad || message))
```

# HMAC in PVS

```
hmac(blockSize: uint8,  
      key : bytestring,  
      (message : bytestring |  
        message'length + blockSize < bytestring_bound),  
      outputSize: upto(blockSize),  
      hash: [bytestring->lbytes(outputSize)]: lbytes(outputSize)  
= LET newkey = IF length(key) > blockSize THEN hash(key) ELSE key ENDIF,  
  newerkey: lbytes(blockSize)  
  = IF length(newkey) < blockSize  
    THEN padright(blockSize)(newkey)  
    ELSE newkey  
  ENDIF,  
  oKeyPad = lbytesXOR(blockSize)(newerkey, nbytes(0x5c, blockSize)),  
  iKeyPad = lbytesXOR(blockSize)(newerkey, nbytes(0x36, blockSize))  
  IN hash(oKeyPad ++ hash(iKeyPad ++ message))  
  
hmac256((blockSize: uint8 | 32 <= blockSize),  
        key : bytestring,  
        (message : bytestring |  
          message'length + blockSize < bytestring_bound))  
: lbytes(32)  
= hmac(blockSize, key, message, 32, sha256message)
```

```
bytestrings__bytestring_t
HMAC__hmac256(uint8_t ivar_2, bytestrings__bytestring_t ivar_3,
              bytestrings__bytestring_t ivar_4){
    bytestrings__bytestring_t result;
    uint8_t ivar_18;
    ivar_18 = (uint8_t)32;
    HMAC_funtype_0_t ivar_19;
    HMAC_closure_3_t cl1230;
    cl1230 = new_HMAC_closure_3();
    ivar_19 = (HMAC_funtype_0_t)cl1230;
    bytestrings__bytestring_t ivar_14;
    ivar_14 = (bytestrings__bytestring_t)
HMAC__hmac((uint8_t)ivar_2, (bytestrings__bytestring_t)ivar_3,
           (bytestrings__bytestring_t)ivar_4, (uint8_t)ivar_18,
           (HMAC_funtype_0_t)ivar_19);
    //copying to bytestrings__bytestring
    //from bytestrings__bytestring;
    result = (bytestrings__bytestring_t)ivar_14;
    if (result != NULL) result->count++;
    release_bytestrings__bytestring(ivar_14);
    return result;
}
```

# Parsing Expression Grammars (PEG)

- PEG grammars are a natural class that capture *recursive descent* parsers.
- The constructs and their semantics are shown below.  
 $\sigma \models e, \sigma'$  means that the expression  $e$  recognizes  $\sigma$  leaving a (unique) suffix  $\sigma'$ . Otherwise,  $\sigma \not\models e$ .
  - 1 Empty:  $\varepsilon$ , where  $\sigma \models \varepsilon, \sigma$
  - 2 Terminal:  $c$ , where  $c\sigma \models c, \sigma$
  - 3 Any:  $any$ , where  $c\sigma \models any, \sigma$
  - 4 Ordered choice:  $E_1/E_2$ , where  $\sigma \models E_1/E_2, \sigma'$  iff  $\sigma \models E_1, \sigma'$  or if  $\sigma \not\models E_1$  and  $\sigma \models E_2, \sigma'$
  - 5 Iteration:  $E^*$  and  $E^+$  where  $\sigma \models E^*, \sigma'$  iff  $\sigma \not\models E$  and  $\sigma' = \sigma$  or  $\sigma \models E, \sigma''$  and  $\sigma'' \models E^*, \sigma'$
  - 6 Optional:  $E?$ , where  $E \models E?, \sigma'$  iff  $\sigma \models E, \sigma'$  or  $\sigma' = \sigma$
  - 7 And:  $\&E$ , where  $\sigma \models \&E, \sigma$  iff  $\sigma \models E, \sigma'$  for some  $\sigma'$
  - 8 Negation:  $!E$ , where  $\sigma \models !E, \sigma$  iff  $\sigma \not\models E$
- Example: The non-CFG expression  $a^n b^n c^n$  is PEG-expressible, but the exact relationship between PEGs and CFG is open.

- PEG expressions are formalized as an algebraic datatype in PVS.
- A PEG grammar assigns PEG expressions  $e$  to nonterminals  $N$ :  $N \leftarrow e$ .
- Not all such grammars are well-formed since direct or indirect left-recursive rules would lead to nonterminating parsers, e.g.,  $S \leftarrow S/a$  is ill-formed.
- The grammar rules are statically analyzed for well-formedness.
- Naïve PEG parsing can be inefficient since the tests ( $E?$ ,  $\&E$ ,  $!E$ ) and ordered choice ( $E_1/E_2$ ) can involve repeated parsing of substrings for the same nonterminal.'

# Formalizing PEG Parsing in PVS

- We have formalized the metatheory of PEG grammars in PVS.
- We defined a static analysis on PEG grammars to identify whether a PEG expression can possibly fail, possibly succeed without consuming any input, or succeed only by consuming some input.
- This is used to define a reference PEG parser and prove its termination, e.g., by constraining a rule  $N \leftarrow e$  so that  $e$  must contain only smaller nonterminals in positions that can be reached (through success or failure) without consuming input.
- This check is necessarily conservative since termination of PEG grammars is undecidable.
- Well-formed grammars yield a terminating parser that constructs a parse tree.
- The parse tree datatype captures the *proof of parse-correctness*, including proof of failure.

# Chart Parsing of PEGs

- Our PEG expression language consists of productions in Chomsky Normal Form:

Empty	$A \leftarrow \epsilon$
Failure	$A \leftarrow f$
Any	$A \leftarrow .$
Terminal	$A \leftarrow c$
Concatenation	$A \leftarrow B C$
Ordered Choice	$A \leftarrow B / C$
Check	$A \leftarrow \&B$
Negation	$A \leftarrow !B$

- Grammars associate a PEG expression with each nonterminal — unlike the previous proof, there is no static analysis for well-formed grammars; loops are detected dynamically during parsing.

# An Example

- We want to parse (a prefix of) strings with linearly nested matched parentheses:  $(((\dots)))$ .
- The grammar and initial scaffold are shown below (with '?' for pending):

<i>NT</i>	<i>Position</i>				
	0 '('	1 '('	2 ')''	3 ')''	4
$S \leftarrow P/E$	?	?	?	?	?
$P \leftarrow BP\ CP$	?	?	?	?	?
$CP \leftarrow S\ EP$	?	?	?	?	?
$BP \leftarrow \text{term}('(')$	?	?	?	?	?
$EP \leftarrow \text{term}(')')$	?	?	?	?	?
$E \leftarrow \varepsilon$	?	?	?	?	?



## Example: Unfolding the Start Query

- We evaluate the top query  $(S, 0)$ , i.e., parse nonterminal  $S$  at nonterminal 0, where  $\mathbf{p}(\phi, 0)$  represents the null stack, and  $\mathbf{p}(S, 0)$  indicates the next stack position is  $a(0)(S)$ .

NT	Position				
	0 '('	1 '('	2 ')' '	3 ')' '	4
$S \leftarrow \boxed{P} / E$	$\mathbf{p}(\phi, 0)$	?	?	?	?
$P \leftarrow \boxed{BP} CP$	$\mathbf{p}(S, 0)$	?	?	?	?
$CP \leftarrow S EP$	?	?	?	?	?
$BP \leftarrow \text{term}('(')$	$\mathbf{p}(P, 0)$	?	?	?	?
$EP \leftarrow \text{term}(')')$	?	?	?	?	?
$E \leftarrow \varepsilon$	?	?	?	?	?

# Example: Unwinding the Parse

- We evaluate the top query  $(S, 0)$ , i.e., parse nonterminal  $S$  at nonterminal 0, where  $\mathbf{p}(\phi, 0)$  represents the null stack, and  $\mathbf{p}(S, 0)$  indicates the next stack position is  $a(0)(S)$ .

NT	Position				
	0 '('	1 '('	2 ')' '	3 ')' '	4
$S \leftarrow \boxed{P} / E$	$\mathbf{p}(\phi, 0)$	$\mathbf{p}(CP, 1)$	?	?	?
$P \leftarrow \boxed{BP} \boxed{CP}$	$\mathbf{p}(S, 0)$	$\mathbf{p}(S, 1)$	?	?	?
$CP \leftarrow \boxed{S} EP$	?	$\mathbf{p}(P, 0)$	?	?	?
$BP \leftarrow term('(')$	$\mathbf{g}(0, 1)$	$\mathbf{p}(P, 1)$	?	?	?
$EP \leftarrow term(')')$	?	?	?	?	?
$E \leftarrow \varepsilon$	?	?	?	?	?

## Example: Completing the Parse

- The scaffold for the full parse is thus filled out with entries as shown below.

<i>NT</i>	<i>Position</i>				
	0 '('	1 '('	2 ')' '	3 ')' '	4
$S \leftarrow P/E$	$g(7, 4)$	$g(4, 2)$	$g(1, 0)$	?	?
$P \leftarrow BP \ CP$	$g(6, 4)$	$g(3, 2)$	$f(1)$	?	?
$CP \leftarrow S \ EP$	?	$g(5, 3)$	$g(2, 1)$	?	?
$BP \leftarrow term('(')$	$g(0, 1)$	$g(0, 1)$	$f(0)$	?	?
$EP \leftarrow term(')')$	?	?	$g(0, 1)$	$g(0, 1)$	?
$E \leftarrow \varepsilon$	?	?	$g(0, 0)$	?	?

# (Chomsky-normal form) PEGs in PVS

```
len, max: VAR index %the length of the input
byte: TYPE = below(256)
strings(len): TYPE = ARRAY[below(len) -> byte]
num_non_terminals: byte = 255
non_terminal: TYPE = below(num_non_terminals)
```

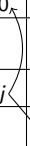
```
peg : DATATYPE
  BEGIN
    epsilon : epsilon?
    failure : failure?
    any(p : [byte -> bool]) : any?
    terminal(a: byte) : terminal?
    concat(e1, e2: non_terminal) : concat?
    choice(e1, e2: non_terminal) : or?
    check(e: non_terminal) : and?
    neg(e: non_terminal) : not?
  END peg
```

# The Scaffold

The scaffold is a two-dimensional array mapping each position/nonterminal to an entry.

```
scaffold(len) : TYPE
= ARRAY[pos: upto(len) ->
  ARRAY[non_terminal -> (nice_entry?(len, pos))]]
```

	0	...	$i$	...	$L$
$n0$	$a_{00}$	...	$a_{i0}$	...	$a_{L0}$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$nj$	...	...	$a_{ij}$	...	$a_{Lj}$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$nN$	$a_{0N}$	...	$a_{iN}$	...	$a_{LN}$



# Step Function

```
step(len, G, (s: strings(len)), (start: upto(len)),
      (rootnt: non_terminal))
  (St : state(len, G, s, start, rootnt))
  : state(len, G, s, start, rootnt)
= (LET scaf = St'scaf,
   stack = St'stack,
   depth = St'depth,
   lflag = St'lflag
   IN
   IF empty?(stack)
   THEN St
   ELSE LET pos = pos(stack),
          cur = nt(stack),
          rest = scaf(pos)(cur)
   IN CASES G(cur) OF
       epsilon: St WITH ['scaf(pos)(cur) := good(0, 0),
                          'stack := rest,
                          'depth := depth - 1],
       :
       :
```

# Outer Parsing Loop

```
parse(len, G, (s: strings(len)), (start: upto(len)),  
      (root: non_terminal), St : state(len, G, s, start, root))  
: RECURSIVE endstate(len, G, s, start, root)  
= (IF St'depth = 0  
   THEN St  
   ELSE parse(len, G, s, start, root,  
              step(len, G, s, start, root)(St))  
   ENDIF)  
   MEASURE size(len, G, s, start, root)(St) BY <
```

# Proof-of-Parse Verification

Defined a proof-of-parse format for success, failure, loop outcomes and extracted valid proofs from parser output.

```
buildproof(len, G, (s: strings(len)),
           (rootpos: upto(len)), (rootnt: non_terminal))
  (st: endstate(len, G, s, rootpos, rootnt),
   n, (i | i <= len)):
  {P | good_parsetree?(len, G, s)(qempty, n, i, P)
   AND entry(P) = st'scaf(i)(n)}
= (IF loop?(st'scaf(i)(n))
   THEN (LET pendfun = (LAMBDA (n: non_terminal): pending),
          A = (LAMBDA (i: upto(len)): pendfun)
          IN buildloop(len, G, s, rootpos, rootnt)
              (st, A, qempty, n, i))
   ELSE buildtree(len, G, s, rootpos, rootnt)(st, qempty, n, i)
   ENDIF)
```



# Introducing let-then-else

Even with memoization (packrat parsing), the PEG grammar can be slow because it is not backtracking efficiently from failure. A variant `let x = A then B else C` can be used more efficiently and replaces several PEG constructors.

```
prepeg: DATATYPE
BEGIN
  epsilon : epsilon?
  failure : failure?
  any(p : [byte -> bool]) : any?
  terminal(a: byte) : terminal?
  lte(fst: non_terminal, lthen : non_terminal, lelse: non_terminal)
    : lte?
END prepeg
```

The original proofs were hard: took around four to six weeks with over a 150 nontrivial proof obligations.

Proofs replayed robustly even though the change was significant.

But the speed-up was only around 30%

# Extending the Base Case to DFAs

One source of slowness is that the base case handles only one character at a time.

The proposed remedy was to make the base case cover any DFA.  
(This is tricky DFA and PEG semantics are not compatible.)

```
prepeg: DATATYPE
BEGIN
  epsilon : epsilon?
  failure : failure?
  any(dfa : dfa) : any?
  terminal(a: byte) : terminal?
  lte(fst: non_terminal, lthen : non_terminal, lelse: non_terminal)
    : lte?
END prepeg
```

With this, the parsing time is halved.

- Change is the only constant.
- No special effort was made to make the above proofs robust.
- The pre-packaged automation (built-ins and strategies) made the proofs reusable across similar proof obligations and across the evolution of the specification/algorithm.
- There are many sources of non-robustness:
  - 1 Proof commands that mention formula numbers are sensitive to reordering/shifting of formulas in a sequent
  - 2 Proof commands that include expressions are sensitive to changes in these expressions, e.g., Skolem names.
  - 3 If the order of cases changes, then proofs can lose synchrony.
  - 4 If definitions change, the proof obligations can get detached from their proofs.
- A lot of post-processing is needed to define proof strategies and refactor proofs to make them robust.

# Conclusions

- Parsing is a key gatekeeper for software systems; parsing verification needs to be streamlined.
- More generally, beyond data formats, modeling dominates the software development activity for critical systems.
- Refinement and code generation bridge the semantic gap between models and code.
- Robust proofs make it easier to evolve better models and implementations.
- The efficiency of code generated from verified high-level models can be competitive with *hand-written code*.
- Code generation can be used to create standalone components or define entire systems.
- *The key value proposition is the use of one unified system and language for specification, modeling, programming, and proof.*