The Parsley Data Description Language

Prashanth Mundkur[1] Linda Briesemeister[1]
Natarajan Shankar[1] Prashant Anantharaman[2]
Sameed Ali[2] Zephyr Lucas[2] Sean Smith[2]

[1]SRI International

[2]Dartmouth College

May 21, 2020

Problem

- What kinds of formal grammars can capture data formats?
  - ELF, OpenDocument, PDF
- How can untrusted data be parsed securely?
  - e.g. prevent parser exploits that manipulate parsing offsets

Limitations of Typical Formal Grammars

- Need for handling long-range context dependency
  - e.g. graph structures in document formats
- Very limited or no interface to parsing buffer
  - e.g. offset manipulations, buffer windowing, transformations

Issue: Handling Context Sensitivity

- Parsing and semantic actions can be controlled by context
- Distant context may need to be threaded through to current parsing operation
- Need a bounded mechanism to thread just the relevant context
- Relevant context can require structured datatypes

Issue: No First-class Parsing Buffers

- Binary data often specify offsets, lengths, etc.
  - Parsing cursor locations are data-dependent
- Network packet formats often require checksums
  - Requires windowed views into parsing buffer
- Data can often be optionally compressed, encrypted, etc.
  - Requires controlled transformations of windowed views into the parsing buffer

Design Goal: Adding Computation to Syntax

We are looking for a DDL that incorporates the computation needed

- to store and retrieve contextual information
- to evaluate contextual constraints
- to operate on parsing buffers

Other Design Goals

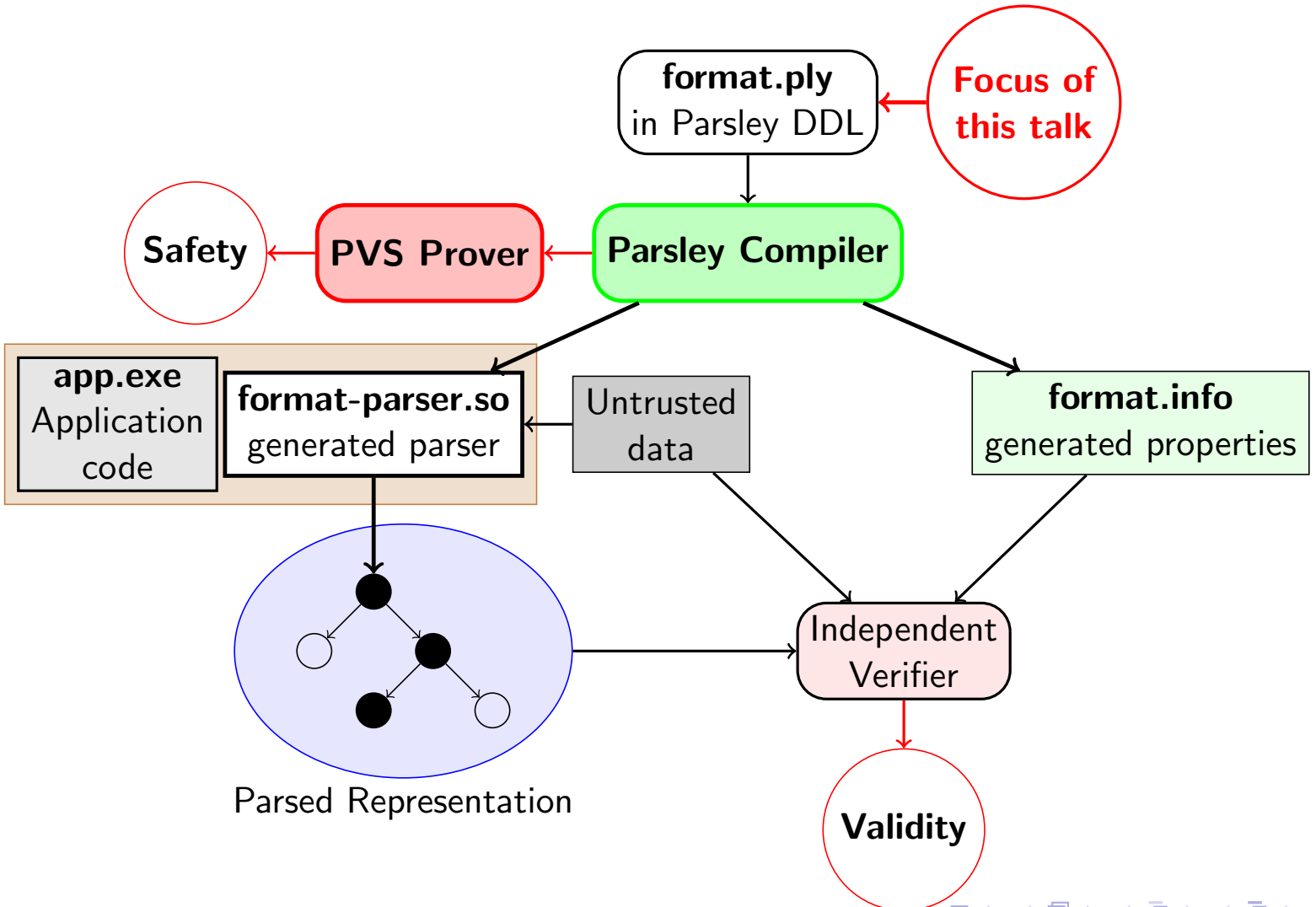
- Automatic extraction of parsers
- Construction of proofs-of-parse to validate correctness (Blaudeau, Shankar CPP 2020)

Parsley Design

- Composition of conventional techniques
  - Parsing expression grammars (PEG)
    - for determinism and unambiguity
  - Attribute grammars
    - for structured handling of context-sensitivity
    - specifying semantic actions
  - Functional languages
    - for type-safe computation of constraints and semantic actions

- Capture syntax using PEG and attribute grammars
- Capture computation using a functional language
- Capture contextual and computational state using attributes

Overall Parsley Architecture



Sublanguages of the Parsley DDL

- Parsing expression grammar for production rules
  - Specified in notation similar to EBNF
    `NonTerm := rule1 ; rule2 ; ... ;;`
- Typed functional expression language
  - Used for constraint expressions
    `[guard1(args) && (guard2(args) || guard3(args))]`
  - and semantic actions
    `{let val := expr(args) in nt.syn_attr := val}`

Grammar sublanguage

- Standard attribute system
  - `NonTerm (inh_attr:type)` := *rules*
    Communicates context-sensitive information using inherited attributes
  - `NonTerm {syn_attr:type}` := *rules*
    Stores information computed during *local* parsing in synthesized attributes
- Boolean constraints can guard parsing progress of a rule

  ... := *prev rule* ; ... `[bool_expr(attrs)]` ... ; *next rule*
  - Expressed in terms of the computed attributes in scope
  - Rewinds on guard failure to the last ordered choice (;) and continues with the next choice
- Standard library contains basic primitives like character classes

  `[:ascii:]`, `[:alphanum:]`

Expression sublanguage

- Simple typed functional language with user-defined types and functions

  ```
  fun f(a: arg_type) -> result_type = {...}
  ```

- Types of grammar non-terminals are represented by records with synthesized attributes as field names

  ```
  type non_term_type = {attrname: type}
  ```

- Standard library contains basic functions such as conversions of parsed data into typed values, and basic data structures like lists, sets, and maps

  `string_to_int()`, `byte_to_int()`, ...

  `List.append`, `Map.extend`, `Set.add`, ...

PDF Examples

- Null

```
Null := "null" ;;
```

- Comments

```
Comment := "%" ([:char:] \ "\n")* "\n" ;;
```

- Booleans

```
Boolean b {val : bool} := "true"  { b.val := true }
                        ;  "false" { b.val := false } ;;
```

- Name object

```
Name n {val : string} :=
  "/" s=([:alphanum:]+)
  { n.val := normalize_name(s) } ;;
```

PDF Examples

- Context-sensitive Whitespace

```
Whitespace (allow_empty : bool) :=
  [allow_empty]  // boolean constraint
  (" " | "\0" | "\t" | "\r" | "\n"
   | "\x0c" // form-feed
   | Comment
  )*
; [!allow_empty]
  (" " | "\0" | "\t" | "\r" | "\n"
   | "\x0c" // form-feed
   | Comment
  )+
;;
```

Tagged-Length-Value (TLV) constructs

```
type tlv =
   TTL  of int
 | Port of int
 | Unknown of byte * [byte]
fun ttl_to_int(b : [byte; 1]) -> int =
{ byte_to_int(b[0]) }
fun port_to_int(b : [byte; 2]) -> int =
{ 256* byte_to_int(b[1]) * byte_to_int(b[0]) }

TLV t { val: tlv } :=
tg=([:byte:]) l=([:byte:]) v=([[:byte:] * byte_to_int(l)])
 ( [tg = 1] [l = 1] { t.val := tlv::TTL(ttl_to_int(v))   }
 | [tg = 2] [l = 2] { t.val := tlv::Port(port_to_int(v)) }
 |                  { t.val := tlv::Unknown(tg, v)       }
 )
```

ELF Fragment

```
type elf_file_hdr = {ph_offset : int}

ELF_File := e=ELF_File_Hdr
            [e.ph_offset < ParseBuffer.size($buffer)]
            @(ParseBuffer.start($buffer) + e.ph_offset)
            p=ELF_Prog_Hdr
            ...
```

Design challenges

- Backtracking in ordered choice requires rewinding attribute updates
  - This is easier with *L*-attributed grammars
- Defining how backtracking composes with operations on the parsing buffer

Acknowledgements

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.

Questions?

Thank you!