Research Report: The Parsley Data Format Definition Language

Prashanth Mundkur
Linda Briesemeister
Natarajan Shankar
SRI International
Menlo Park, CA 94025

Prashant Anantharaman
Sameed Ali
Zephyr Lucas
Sean Smith
Dartmouth College
Hanover, NH 03755

***Abstract*—Any program that reads formatted input relies on parsing software to check the input for validity and transform it into a representation suitable for further processing. Many security vulnerabilities can be attributed to poorly defined grammars, incorrect parsing, and sloppy input validation. In contrast to programming languages, grammars for even common data formats such as ELF and PDF are typically context-sensitive and heterogenous. However, as in programming languages, a standard notation or *language* to express these data format grammars can address poor or ambiguous definitions, and the automated generation of correct-by-construction parsers from such grammar specifications can yield correct and type- and memory-safe data parsing routines. We present our ongoing work on developing such a data format description language. Parsley is a declarative data format definition language that combines grammars and constraints in a modular way. We show how it can be used to capture data formats such as MAVLink, PDF and ELF. We briefly describe the processing pipeline we are designing to generate verified parsers from these specifications.**

1. Introduction

Parsers serve as a firewall between any possibly adversarial input received by a program, and the rest of the program. Since such data can, maliciously or inadvertently, trigger vulnerabilities in a program, the data and any operations on it need to be validated before the data is copied or processed in any way. Quite often, programmers design their own communication protocols and data formats and implement *ad hoc* parsers for them in low-level languages such as C and Rust. But with imprecisely specified data formats and parsers written without adequate thought given to malicious data, applications are susceptible to attacks via their parsing modules. Some data definition languages have been developed (e.g., DFDL [1], PADS [2]) to address these issues of specification and parser implementation, but common data and document formats such as PDF and ELF [3] contain constructs that are not easily expressed in them.

Grammars and their parsing techniques have long been well established components of the toolbox used for specifying and implementing programming languages and their parsers. However, the language-processing toolbox is less developed when it comes to data, document and network protocol formats. Unlike the grammar of a programming language, the grammatical specifications of many data formats often show context sensitivity and constrained data-dependency (e.g. the value of the checksum field in a TCP packet depends on the packet contents with possible zero-padding). As a result, applications that handle document and network data use substantially different processing pipelines than those used for programming languages. For example, the actions of a parser for the packets in a network protocol can be driven by a global protocol session state. Similarly, document formats and streaming formats can have container structures within which content is described by independent data formats (e.g. image data within a document). Automatically generated parsers for such data formats, and the techniques involved in the generation of such parsers, are therefore different from those for programming languages.

We describe in §2 our current design of Parsley, an expressive, declarative, modular, data format definition language for data-dependent formats, that is intended to alleviate many of the above problems in data processing. The language has the following design goals:

- A useful foundational set of parser primitives and combinators,
- A capacity to capture context-sensitivity and data-dependency via a constraint system, and
- A module system that enables nested grammars and composes with the constraint system, and
- A formalizable static and dynamic semantics for parsing.

The language is not intended to explicitly support aspects not directly related to parsing, such as specifying protocol state machines for network protocols, or rendering state for document formats.

A format specification in the Parsley language is processed by the Parsley compiler, which performs semantic checks to detect the presence of unsafe or ambiguous constructions, and generates a safe verified parser. This is described in §3. We give examples of the use of Parsley to specify data formats in §4. We compare the Parsley approach to earlier and related work in §5.

2. Language Design

We now explain how the above goals motivate our language design choices.

- The core structure of a Parsley specification is provided by a *parsing expression grammar (PEG)* [4], concretely specified in notation similar to *extended BNF (EBNF)* for grammar productions. Although *context-free grammars (CFGs)* also use the EBNF notation, there are critical differences in its adaptation for PEGs: (a) the choice combinator in PEGs is ordered, as opposed to non-deterministic in CFGs, and (b) the PEG notation includes syntactic predicates that do not occur in CFGs. The choice of a PEG core provides us our base set of primitives and combinators. This choice is motivated by its deterministic execution and amenability to formalization [5] [6].
- Context management is provided by a fairly traditional attribute-grammar system, with a fully specified expression language for attribute computations. In Parsley, non-terminals have user-defined attributes, while terminals have a default attribute value of type byte or byte-string. This system provides Parsley with a mechanism to capture context-sensitivity.
- Additional context-sensitivity is provided by a constraint system that guards further processing within a non-terminal production. The constraint language uses the attribute system to perform context-sensitive checks, and employs primitives in the constraint expression language to perform data-dependent parsing. This constraint system is motivated by the inadequacy of the standard PEG constructs for data-dependent context sensitive parsing (e.g. detecting cyclic relationships when parsing the objects in a PDF file.)
- Parsley includes a module system to enable the composition of independent grammars. This is designed to allow the integration of various different packet formats into a single networking stack, and various different image and font formats into a single document format (such as PDF).

These choices have resulted in two sublanguages constituting Parsley: an expression sublanguage and a grammar sublanguage. The expression sublanguage used in attribute updates and constraints is a strongly typed polymorphic functional language supporting user-defined types and functions. It includes a standard library of common types and utility functions. The expression and data type sublanguage is a total language by construction, that ensures that recursive and iterative computations always terminate. The grammar sublanguage contains the parsing primitives and combinators, and the attribute system. A fragment of the abstract syntax of Parsley is shown in Figure 1.

Paths	p	::=	$x \mid p.x$
Constants	c	::=	0,1,... $\mid$ 'A', 'B', ... $\mid$ ...
Constructors	C	::=	C_i
Attributes	l	::=	$l^i \mid l^s$
Base types	ν	::=	$\texttt{unit} \mid \texttt{uint8} \mid \texttt{int32} \mid \ldots$
Monotypes	τ	::=	$\alpha \mid \nu \mid (\tau_i) \mid \tau \rightarrow \tau \mid \sum_i C_i\tau_i$
			$\mid \{l_i : \tau_i\} \mid \texttt{typeof}(N)$
Types	σ	::=	$\tau \mid \forall\alpha.\sigma \mid \sigma \rightarrow \sigma$
Expressions	e	::=	$p \mid \texttt{c} \mid Ce \mid (e_i) \mid e\ e \mid e\ \texttt{op}\ e$
			$\mid (e : \tau) \mid e.l \mid \texttt{case}\ \ e\ \{\rho_i, e_i\}$
			$\mid \texttt{let}\ \ x = e\ \ \texttt{in}\ \ e \mid e \sim C \mid f\ \ e$
Patterns	ρ	::=	$x \mid (\rho_i) \mid C\rho$
Functions	f	::=	$f := (x_i : \sigma_i) \rightarrow \sigma\{e\}$
Statements	s	::=	$p = e$
Constraints	ϕ	::=	e
Actions	a	::=	$\{s_i\}$
Rules	r	::=	$\epsilon \mid \texttt{c} \mid \texttt{scan}(c) \mid \texttt{scan}^R(c)$
			$\mid (x =)^? N\{l_i = e_i\}$
			$\mid \phi\ r \mid r\ r \mid r/r \mid r^{*e?}\ \mid !r \mid a$
Productions	P	::=	$N\{l_i : \tau_i\} := r$
Format	F	::=	$\{\tau_i, N\{l_i : \tau_i\}, P_i\}$

Figure 1. Abstract syntax of the Parsley specification language

Expression sublanguage

In Figure 1, x ranges over identifiers, l over inherited l^i and synthesized l^s attributes, α ranges over type variables, c over constants, and *op* over standard arithmetic and boolean operators. Paths p are used to access identifiers across modules. As discussed below, attributed non-terminals are represented as records, and we denote record fields as well as attributes using l. C ranges over the constructors used for values of sum types.

The type system is a conventional polymorphic type system with records. It is equipped with a set of standard base types ν, type constructors like tuples (τ_i), functions $\tau \rightarrow \tau$, sums $\sum_i C_i\tau_i$, records $\{l_i : \tau_i\}$ and polymorphic type schemes $\forall\alpha.\tau$. The Parsley library defines standard data types such as polymorphic lists, sets, and maps.

The expression language is also standard, with tuples (e_i), sums C_ie_i, function application $e\ e$, attribute or record field selection $e.l$, type constraints $(e : \tau)$, and let bindings and case expressions that bind patterns within expressions. The $e \sim C$ is a boolean expression that tests if e is a sum with the C constructor; this is a useful primitive within grammar constraints.

Parsley supports user-defined higher-ranked polymorphic functions f, where the function arguments have polymorphic types.

Statements are the only imperative constructs, and consist of assignments to (synthesized) attributes.

Grammar sublanguage

We now come to the grammatical constructions: productions, rules, and actions. These are based on parsing expression grammars (PEGs) [4], but extended with an attribute system and constraints to capture context sensitivity. A grammar production defines the parsing expression for a non-terminal N. Each N is typed as a record, named $\texttt{typeof}(N)$, with its attributes as the record fields. Rules r are elemental parsing expressions and are combined with parsing combinators. Actions a are used to update the parsing state during parsing, and consist of a sequence of statements where synthesized attributes are assigned. The primitive rules are ϵ, which successfully matches without consuming any input, and literals $\texttt{c}$, which match if the input has a c prefix, upon which that prefix is consumed. The $N\{l_i = e_i\}$ rule matches if its production matches an input prefix provided its inherited attributes l_i are initialized with expressions e_i evaluated within the parsing context at that point. The $x = N$ construct names the matched value (of type $\texttt{typeof}(N)$), and enables access to the values of its attributes as $x.l$ in expressions appearing within any subsequent constraints, actions, or attribute initializations.

These primitive rules can be combined with combinators, as in PEG, but with some extensions. $r\ r$ denotes the PEG sequence operator, while $r\ /\ r$ is the PEG ordered choice. The $!r$ construct is the *not* syntactic predicate in PEG. $r^{*e?}$ denotes the Kleene star, but with an optional bound e that is context sensitive. If the bound e is present, it is an integer valued expression that limits the number of times that r is matched.

A parsing rule r can be guarded with a constraint expression ϕ, written as $\phi\ r$: the boolean expression ϕ is evaluated before deciding to proceed parsing with rule r. This is a Parsley extension of PEG, and could be thought of as a *contextual* predicate: ϕ can check the current parsing state, using the attribute values of syntactic elements in its context. If this guard fails, parsing backtracks to the most recent choice point, and continues with the next alternative.

Parsley is equipped with a module system that enables splitting a specification into multiple files, and allows data specifications to be re-used in different contexts. Paths support the module system (not shown) by enabling cross-module use of types and syntax elements.

Extensions under development

The Parsley language is a work-in-progress. It is being adapted as we attempt to use it to capture more data formats, in order to gauge the extent to which existing data formats can be specified in our current formalism or extensions to it. We also intend to use it to define the data formats needed for our other research projects.

We are also considering extending Parsley with primitives to capture manipulations of the current parsing offset by actions such as seeking. Such offset manipulation could either derive declaratively from the format specification, or procedurally under the control of the application driving the parser. A design challenge is to ensure both types of manipulations are supported and compose well. In addition, this is one of the most security-critical aspects of parsing: ensuring that offsets derived from untrusted data are used in valid ways.

Declarative offset manipulation is required for capturing formats such as PDF, where parsing a PDF file involves seeking to the end of the file (or a specific marker) and searching backwards for a syntactic element. This motivated the $\texttt{scan}(c)$ and $\texttt{scan}^R(c)$ constructs, which scan forwards or backwards for a literal c, and when successful adjust the current parsing offset to the match point. The scans are not explicitly bounded[1], though it should be straightforward to introduce bounded scans if required.

3. The Parsley Processing Pipeline

The typical use of a Parsley specification is via the workflow illustrated in Figure 2. The Parsley compiler performs type and semantic checks on a given Parsley specification. The type checks correspond to standard polymorphic type checking [7] [8], with additional typing rules to handle grammar productions. In addition, attribute usage checks are performed, such as for L-attributedness (see below).

The compiler generates prover definitions that are used to perform safety checks: for example, data-dependent values are used as indices into structures such as arrays provided they are suitably constrained. We are developing the safety properties that can be used by a theorem prover (PVS in our case) to verify that the Parsley specification is a ‘safe’ grammar. The compiler also flags the use of constructs such as $\texttt{scan}(c)$, which can create the potential for polyglot formats [9].

We are also defining the parsing automaton that will be generated by the compiler. This automaton needs to interleave constraint and attribute computation with parsing actions that construct the internal data representation and manipulate the parsing buffer. When used for efficiently parsing data, especially network protocols, attribute computations have to be done on partially constructed parse trees (unlike the typical use of attribute grammars in programming languages, where the entire parse tree of a program can be available before processing). In our experience of using Parsley to define data formats, the computations in constraints and attribute updates require access to a parsing context (via inherited attributes) that is best handled by L-attributed grammars, which are compatible with the top-down processing of the “core” PEG specification. L-attributed grammars are attribute grammars where attributes in the nodes in the parse tree to the left of the current node have been assigned their final values. This eases the implementation of PEG backtracking, as individual attribute values do not need to be unwound. The generated automaton is compiled into a library that can be linked into application code. We plan to use a Rust code generator to ensure that the library is type and memory safe, and can be linked into C applications, and is compatible with other languages using the appropriate wrappers.

1. They are implicitly bounded by the runtime system which knows the size of the parsing buffer.

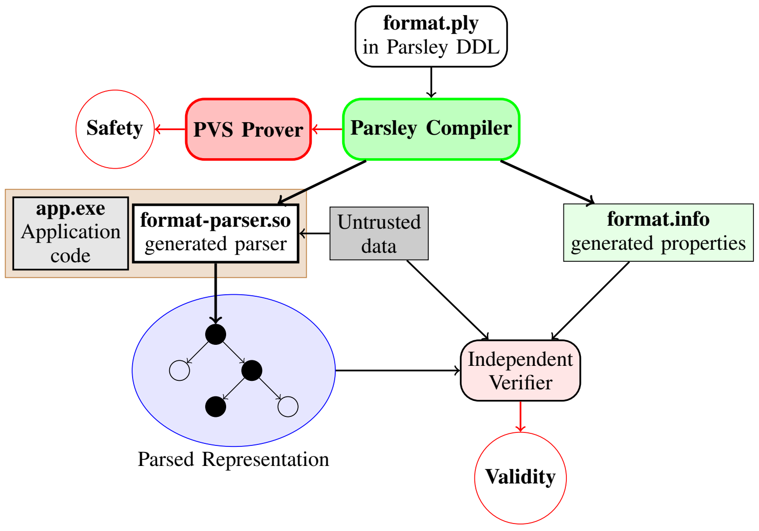

Figure 2. Parsley in context.

The data representation constructed by the automaton contains a *proof of parse*. We are building on the approach used in [5], which formalized PEG parsing and the generation of PEG parsers. These parsers annotate the constructed data representation with grammar metadata that can be used to verify that the representation is a valid parse of the given input data for the given Parsley specification. This verification is done in a generic (i.e. grammar-independent) verifier, which is simple enough to include in the application itself. In addition to ensuring parser correctness, the verifier provides tamper-resistance for the parse tree, e.g. across serialize-deserialize operations.

Unresolved issues

The API between the application and the parsing library will dictate crucial implementation choices, and hence impact the code generation pipeline. This API might differ across applications parsing the same format, and these types of API often differ across different types of formats (e.g. streaming network formats versus document formats).

A crucial requirement is to ensure that the computation performed by the parsing automaton is bounded and well-characterized; e.g. arbitrary input data cannot cause unrestricted computation within the parsing library. We are attempting to provide formal guarantees that this will be the case.

Our language definition and processing infrastructure has focused more on the parsing problem, and less on the encoding or serialization aspect. We have left this for later investigation.

4. Case Studies with Parsley

We have found the language design above has allowed the succinct specification of several data formats as well as packet formats in a few network protocols. We show below snippets of the packet formats in the MAVLinknetwork protocol and fragments of the ELF and PDF data formats.

4.1. MAVLink

The Micro Air Vehicle link (MAVLink) protocol is an open source standard for unmanned aerial vehicles. The packet payload lengths can span from 0 bytes to 255 bytes that can use one of 18 different message types and one of the 91 MAVLink commands [10]. As shown in Figure 3, the packets include two integrity mechanisms, with a third optional mechanism. First, the packets include a length field—that shows the total length of the rest of the packet. Second, the packets include a 2-byte checksum after the payload. The packet has an optional trailing 13-byte signature.

Listing 1 shows the Parsley syntax corresponding to the format in Figure 3. The *mavLinkPkt* non-terminal is represented by a record with typed synthesized attributes corresponding to the packet fields of interest. The production for this non-terminal first checks for the literal magic bytes `\xfd` identifying a MAVLink packet. It then extracts the payload length into the *pll* local variable using the definition *Int8* of an 8-bit integer interpretation of a byte from the Parsley standard library. The following scalar fields are similarly extracted into variables, which are used in the terminating action to assign the values of the corresponding synthesized attributes. The extraction of the payload bytes into *pld* uses a byte array specified by the bounded Kleene star combinator, whose data-dependent length is computed from *pll* using a standard library function *$int_of_byte*. After the checksum field *crc* is extracted, the following constraint uses a user-defined function *compute_checksum* to ensure that the extracted checksum is valid. A similar sequence of parsing and constraint rules is used to verify the packet signature. Parsley allows this sequence to be wrapped in the optional construct `?`, which returns a value in the option type defined in the standard library.

Fixed start	Length	Incomp. Flags	Comp. Flags	Sequence number	System ID	Comp. ID	Message ID	Payload	Checksum	Signature
1 byte	1 byte	1 byte	1 byte	1 byte	1 byte	1 byte	1 byte	0-255 bytes	2 bytes	13 bytes

Figure 3. MAVLink packet structure.

```
type option (’a) = Some of ’a | None

mavLinkPkt mvl {
  { magic         : u8,
    payloadLength : u8,
    incompatibilityFlags : u8,
    compatibilityFlags   : u8,
    packetSequenceNumber : u8,
    systemID      : u8,
    componentID   : u8,
    messageID     : u32,
    payload       : [u8],
    crc           : u16,
    sig           : option<[u8]> }
  :=
    "\xfd"
    pll = Int8   icf = Int8
    cf  = Int8   ps  = Int8
    sid = Int8   cid = Int8
    mid = Int32

    pld = [Byte * $int_of_byte(pll)]

    crc = Int16
    [ crc == compute_checksum(pld) ]

    sig = ( [Int8 * 13]
            [ verify_sig(sig) ] )?

  { mvl.magic = "\xfd";
    mvl.payloadLength = pll;
    ...
    mvl.payload = pld;
    mvl.crc = crc;
    mvl.sig = sig }
```

Listing 1. MAVLink Syntax in Parsley.

```
type endianness = Big   | Little;
type wordsize   = ELF32 | ELF64;

type elf_ctxt = {
  endian   : endian,
  wordsize : wordsize
}

elfIdent eident {
  magic      : [u8],
  version    : u8,
  osabi      : u8,
  abiversion : u8,
  ctxt       : elf_ctxt
} :=
  "\x7FELF"
  c = Int8   d = Int8
  v = Int8   o = Int8
  a = Int8   p = [Int8 * 7]

  [c == 1 || c == 2]
  [d == 1 || d == 2]
  [v == 1]
  [o >= 0x00 && o <= 0x11]

  { eident.magic = "\x7FELF";
    eident.class = c;
    eident.ctxt =
      { wordsize = (case c of
                    | 1 -> ELF32
                    | 2 -> ELF64),
        endian  = (case d of
                   | 1 -> Little
                   | 2 -> Big) };
    eident.version = v;
    eident.osabi = o;
    eident.abiversion = a;
  }
```

Listing 2. ELF ident parser

This shows how Parsley interleaves computation and parsing using the the expression and grammar sublanguages to declaratively define the MAVLink packet format.

4.2. ELF

The Executable Linkable Format (ELF) is used by GNU/Linux systems to describe executable binaries and shared libraries. It is designed to be generic with respect to machine architectures. As a result, the widths and endianness of the fields of various ELF structures in the ELF file are determined only after some initial parsing of the ELF header. In particular, the ELF file header contains fields whose widths are data-dependent. We illustrate how this can be captured in a Parsley format specification.

The ELF header starts with 16 byte identification block called an `ident`. The `ident` block contains the architecture independent fields which control how the respect of the header is parsed. The `ident` block is specified in listing 2, and starts with the `\x7FELF` magic bytes, followed by a byte each for wordsize class, data endianness, an ELF format version identifier, an OSABI type, and an OSABI version. The block terminates with padding bytes. Since the endianness and wordsize control subsequent parsing, their values are stored in the *ctxt* attribute as a record of a user-defined type *elf_ctxt*. The parsing of the `ident` block is fairly straightforward, with some constraints to ensure that the field values conform to the standard.

The common architecture-dependent field types are specified in listing 3. *ELFInteger16* and *ELFInteger32* define fields that are always 2 bytes or 4 bytes in width, respectively, but differ in endianness. *ELFWord* similarly defines a field that can differ in both width and endianness.

```
ELFInteger16 i (ctxt : elf_ctxt)
  { val : u16 }
:=
  [ i.ctxt.endianness == Little ]
  j=Int16LE { i.val = j }
/ [ i.ctxt.endianness == Big ]
  j=Int16BE { i.val = j }

ELFInteger32 i (ctxt : elf_ctxt)
  { val : u32 }
:=
  [ i.ctxt.endianness == Little ]
  j=Int32LE { i.val = j }
/ [ i.ctxt.endiannness == Big ]
  j=Int32BE { i.val = j }

ELFWord w (ctxt : elf_ctxt)
  { val : int }
:=
  [ w.ctxt.endianness == Little &&
    w.ctxt.wordsize == ELF32 ]
  i=Int32LE { w.val = i }
/ [ w.ctxt.endianness == Little &&
    w.ctxt.wordsize == ELF64 ]
  i=Int64LE { w.val = i }
/ [ w.ctxt.endianness == Big &&
    w.ctxt.wordsize == ELF32 ]
  i=Int32BE { w.val = i }
/ [ w.ctxt.endianness == Big &&
    w.ctxt.wordsize == ELF64 ]
  i=Int64BE { w.val = i }
```

Listing 3. Parsley specification of ELF integers.

Ordered choice and constraints are employed in their definitions in an idiom similar to a switch statement, where the constraints use inherited attributes to perform context-sensitive parsing decisions. For instance, *ELFInteger16* is defined with an inherited attribute *ctxt* that contains the relevant architecture context. It uses the context endianness to first check whether a little-endian integer parse of 2 bytes is appropriate; if not, it proceeds with a big-endian parse. The parsed integer value is stored in the *val* synthesized attribute. *Int16LE* and *Int16BE* are parsers defined in the standard library. *ELFInteger32* and *ELFWord* are defined similarly.

These definitions now allow us to parse the architecture dependent fields in the rest of the ELF file header, as shown in listing 4. The *elfHeader* is provided the architecture context in the *ctxt* inherited attribute, which it relays to the architecture specific parsers of each field. The new syntax in the listing shows how the inherited attributes of a non-terminal are initialized in a production.

4.3. PDF

We have specified the core PDF object syntax in Parsley. The snippet in Listing 5 illustrates the use of attributes and constraints in specifying the syntax of indirect objects in PDF.[2]

2. For brevity, we have elided the specification of whitespace handling.

```
elfHeader h (ctxt : elf_ctxt) {
  typ       : u16,
  machine   : u16,
  version   : u32,
  entry     : int,
  phoff     : int,
  shoff     : int,
  flags     : u32,
  ehsize    : u16,
  phentsize : u16,
  phnum     : u16,
  shentsize : u16,
  shnum     : u16,
  shstrindx : u16
 }
 :=
  ty  = ELFInteger16 <ctxt = h.ctxt>
  mc  = ELFInteger16 <ctxt = h.ctxt>
  vr  = ELFInteger32 <ctxt = h.ctxt>
  ent = ELFWord <ctxt = h.ctxt>
  pho = ELFWord <ctxt = h.ctxt>
  sho = ELFWord <ctxt = h.ctxt>
  flg = ELFInteger32  <ctxt = h.ctxt>
  esz  = ELFInteger16 <ctxt = h.ctxt>
  phsz = ELFInteger16 <ctxt = h.ctxt>
  phnm = ELFInteger16 <ctxt = h.ctxt>
  shsz = ELFInteger16 <ctxt = h.ctxt>
  shnm = ELFInteger16 <ctxt = h.ctxt>
  shix = ELFInteger16 <ctxt = h.ctxt>

  [ ty.val >=0 && ty.val <= 4
  || ty.val == 0xFE00
  || ty.val == 0xFEFF
  || typ.val == 0xFF00
  || typ.val == 0xFFFF]
  [ mc.val >= 0 && mc.val <= 100 ]
  [ vr.val == 0 || vr.val == 1 ]
  [ ehsz.val >=
        (case h.ctxt.wordsize of
        | ELF32 -> 52
        | ELF64 -> 64) ]
  { h.typ         = ty.val
    h.machine     = mc.val
    h.version     = vr.val
    ...
    h.shnum       = shnm.val
    h.shstrindx   = shix.val
  }
```

Listing 4. The ELF Header parser

An indirect object, specified as *IndirectObj*, has one inherited attribute *ctxt* and four synthesized attributes, *id*, *gen*, *val*, and updated context in *ctxt_updt*. *ctxt* contains a context of user-defined type *obj_ctxt* that maps pairs of integers representing object identifiers to the type representing the *PDFObj* non-terminal, named as *i* for convenience. *val* contains the PDF object defined by the indirect object, and is of type *typeof(PDFObj)*, which is the type representing the non-terminal *PDFObj*.

The production rule for *IndirectObj* first parses its identifier into a local variable *n*, using a parser *IntObj* for PDF integer objects defined elsewhere. The definition of *IntObj* is trivial and is omitted, but has an attribute *val* containing its numerical value. The subsequent constraint expression ensures that the parsed numerical value is positive, as man-

```
type obj_ctxt =
  map<(int, int), typeof(PDFObj)>

IndirectObj i (ctxt: obj_ctxt)
  { id: int, gen: int,
    val: typeof(PDFObj),
    ctxt_updt: obj_ctxt }
:=
  n=IntObj [ n.val >= 0 ]
  g=IntObj [ g.val >= 0 &&
    !i.ctxt.contains((n.val, g.val)) ]
  'obj' o=PDFObj<ctxt=i.ctxt> 'endobj'
  { i.id := n.val;
    i.gen := g.val;
    i.val := o;
    i.ctxt_updt[(n.val, g.val)] := i
  }

PDFObj o (ctxt : obj_ctxt) {} :=
  DictObj     <ctxt = o.ctxt>
/ ArrayObj    <ctxt = o.ctxt>
/ IndirectObj <ctxt = o.ctxt>
/ ...
```

Listing 5. The PDF indirect object in Parsley.

dated by the PDF specification. If the constraint is not satisfied, the parse of the production fails. Similarly, *g* expects another positive integer object, representing the generation of the indirect object. The PDF specification mandates that an indirect object is uniquely identified by its identifier and generation. The constraint following *g* ensures that this pair has not been defined yet in the context, using the *contains* method of the *map* datatype in the Parsley standard library. Once the literal `obj` is parsed, the rule expects to parse a *PDFObj* (whose inherited *ctxt* is initialized), followed by the literal `endobj`. On success, the terminating action sets the values of the synthesized attributes of *i*, and collects the updates to the context in *ctxt_updt*, which include the identity of the newly parsed indirect object.

The definition of a general *PDFObj* is specified as an ordered choice among the various types of PDF objects.

5. Related Work

We build on the core PEG ideas developed in [4], and the attribute grammar approach initiated in [11]. The Parsley contextual predicates, or constraints, are very similar to the semantic predicates discussed in [12] and implemented in ANTLR, although their context is $LL(k)$ grammars. *L*-attributed grammars [13] were motivated by their support for one-pass translation; in the Parsley context, they are also compatible with the backtracking requirements of PEG combinators. We are considering the approach to modularity in [14] in our design of the Parsley module system.

In complex formats such as PDF, parsing involves the construction of a representation of the input data, along with the use and enforcement of non-local constraints on the parsed objects; e.g., the page tree object in a document should not have cycles. The data description languages considered below do not address such data constraints.

Nail [15] implements a description format that supports dependent grammars and dependent fields. However, it does not support a rich constraint system, which requires a constraint expression language. Nail takes an interesting approach to nested formats using its notions of streams and transformations; we intend to address format nesting using our attribute and module system to perform the required parameter passing and handling of shared state. The Data Format Description Language (DFDL [1], pronounced as Daffodil), is an XML-based system for data description schemas, with limited support for user-specifiable constraints.

PADS [16] is an expressive strongly-typed data description language with support for data-dependent parsing and user-specified data constraints. However, we consider inherited attributes to provide a more convenient and modular access to distant parsing context than the let-binding constructs in PADS. The Yakker system [17] allows user-defined attributes to contain context to be used for later parsing, or attribute-directed parsing. But they do not include this attribute system in their abstract syntax, leaving it unclear if all such attributes are inherited, with no upward communication via synthesized attributes.

Narcissus [18] is a parser combinator framework for specifying encoders and decoders embedded in Coq that is able to handle a wide range of network packet formats, including support for typical constructs such as checksums and constrained packet fields. It ensures that the encoders and decoders for a format are inverses of each other by relying on the format specifier to provide suitable proof derivation rules in Coq. The Parsley framework attempts to provide a more user-friendly and flexible format specification environment, but does not currently address format serialization and its correspondence to parsing.

Our Parsley formalization builds on the PEG formalization in [5], including the use of an embedded proof-of-parse that can be used for independent verification of a parse result.

Acknowledgments

This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) under Contract No. HR001119C0075. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Defense Advanced Research Projects Agency (DARPA).

References

[1] M. J. Beckerle, S. M. Hanson, and A. W. Powell, "Data format description language (DFDL) v1.0 specification," *Apache*, 2014. [Online]. Available: https://daffodil.apache.org/docs/dfdl

[2] K. Fisher and R. Gruber, "PADS: A Domain-Specific Language for Processing Ad Hoc Data," *SIGPLAN Not.*, vol. 40, no. 6, p. 295–304, Jun. 2005. [Online]. Available: https://doi.org/10.1145/1064978.1065046

[3] M. Dunckley and S. Rankin, "The use of file description languages for file format identification and validation," in *Proc. PV 2007 Conference: Ensuring the Long-Term Preservation and Value Adding to Scientific and Technical Data*, 2007.

[4] B. Ford, "Parsing Expression Grammars: A Recognition-Based Syntactic Foundation," *SIGPLAN Not.*, vol. 39, no. 1, p. 111–122, Jan. 2004. [Online]. Available: https://doi.org/10.1145/982962.964011

[5] C. Blaudeau and N. Shankar, "A verified packrat parser interpreter for parsing expression grammars," *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, Jan 2020. [Online]. Available: http://dx.doi.org/10.1145/3372885.3373836

[6] A. Koprowski and H. Binsztok, "TRX: A formally verified parser interpreter," *Logical Methods in Computer Science*, vol. 7, no. 2, pp. 1–26, 2011. [Online]. Available: https://doi.org/10.2168/LMCS-7(2:18)2011

[7] J. Dunfield and N. R. Krishnaswami, "Complete and easy bidirectional typechecking for higher-rank polymorphism," in *International Conference on Functional Programming (ICFP)*, Sep. 2013, arXiv:1306.6032[cs.PL].

[8] J. Zhao, B. C. d. S. Oliveira, and T. Schrijvers, "A mechanical formalization of higher-ranked polymorphic type inference," *Proc. ACM Program. Lang.*, vol. 3, no. ICFP, Jul. 2019. [Online]. Available: https://doi.org/10.1145/3341716

[9] J. Magazinius, B. K. Rios, and A. Sabelfeld, "Polyglots: Crossing origins by crossing formats," in *Proceedings of the Conference on Computer and Communications Security (CCS)*, 2013, p. 753–764.

[10] Y. Kwon, J. Yu, B. Cho, Y. Eun, and K. Park, "Empirical analysis of mavlink protocol vulnerability for attacking unmanned aerial vehicles," *IEEE Access*, vol. 6, pp. 43 203–43 212, 2018.

[11] D. E. Knuth, "Semantics of context-free languages," in *In Mathematical Systems Theory*, 1968, pp. 127–145.

[12] T. Parr and R. Quong, "Adding semantic and syntactic predicates to ll(k): pred-ll(k)," in *International Conference on Compiler Construction*, vol. 786. Springer, 05 1994, pp. 263–277.

[13] P. M. Lewis, D. J. Rosenkrantz, and R. E. Stearns, "Attributed translations," *Journal of Computer and System Sciences*, vol. 9, 1974.

[14] H. Zhang, H. Li, and B. C. d. S. Oliveira, "Type-safe modular parsing," in *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering*, ser. SLE 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 2–13. [Online]. Available: https://doi.org/10.1145/3136014.3136016

[15] J. Bangert and N. Zeldovich, "Nail: A practical tool for parsing and generating data formats," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. Broomfield, CO: USENIX Association, Oct. 2014, pp. 615–628. [Online]. Available: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/bangert

[16] M. Daly, Y. Mandelbaum, D. Walker, M. Fernández, K. Fisher, R. Gruber, and X. Zheng, "PADS: An end-to-end system for processing ad hoc data," in *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '06. New York, NY, USA: ACM, 2006, pp. 727–729. [Online]. Available: http://doi.acm.org/10.1145/1142473.1142568

[17] T. Jim, Y. Mandelbaum, and D. Walker, "Semantics and algorithms for data-dependent grammars," in *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'10.* ACM, 2010, pp. 417–430.

[18] B. Delaware, S. Suriyakarn, C. Pit-Claudel, Q. Ye, and A. Chlipala, "Narcissus: Correct-by-construction derivation of decoders and encoders from binary formats," *Proc. ACM Program. Lang.*, vol. 3, no. ICFP, Jul. 2019. [Online]. Available: https://doi.org/10.1145/3341686