ParTS: Final Report

Chris Casinghino ccasinghino@draper.com
Cody Roux croux@draper.com

May 2020

Submitted to:

Dr. Stephen Kuhn
148 Electronic Parkway
Rome, NY 13441
stephen.kuhn@us.af.mil

Submitted by:

The Charles Stark Draper Laboratory, Inc.
555 Technology Sq.
Cambridge, MA 02139

Table of Contents

i Task Objectives

i.A Motivation: Insecure Parsers in Practice

Parsers are everywhere. The process of validating and translating input data is a core component of many pieces of software. Compilers must recover program structure from text files, network stacks must handle data packets according to a published standard, and word processing applications must process and display data according to any number of complex public and proprietary data formats.

Because parsers interact directly with data from a source outside the application, their vulnerabilities are often easily exploited by attackers. The last decade has seen numerous real-world attacks—examples include critical bugs in Cisco firewall XML parsers [1, 2], Android media parsers [3, 4], and Windows font parsers [5, 6], all of which resulted in arbitrary code execution exploits. It is clear that commercial software developers remain unable to secure this critical component.

These failures are surprising in light of the well-understood theory of formal languages and the availability of tools to automatically generate correct parsers. In practice, programmers are hand-writing custom parsers rather than using these tools. We believe programmers' unwillingness to use better parsing tools has three key causes, which motivate the ParTS project:

- Parser generator tools can be difficult to use. They require programmers to learn a custom language to express their format in terms of formal grammars and to integrate complex generated code with their development.
- Hand-written parsers often out-perform generated parsers. Generators typically use "one-size-fits-all" parsing algorithms, which may be more complex and slower than necessary for a particular format. Additionally, hand writing parsers can allow users to perform lexing, parsing, and validation steps in one pass, while these must typically be expressed and executed separately when using parser generator tools.
- Real-world formats have complex constraints and data dependencies. They often involve parse-time validation or computations. These requirements can be challenging or impossible to achieve with parsers generators.

i.B Project Goals

The core goal of ParTS is to build a parsing library that can be used in practice by overcoming the three limitations described above. We aim for:

- **Ease of use**: Our library must be more appealing to programmers than an external parser generator tool, and should not require programmers to express their format in a custom formal grammar language.
- **High performance**: Our library must out-perform parser generator tools.
- **Broad format support**: Our library must support a broad class of real-world formats and complexities like data dependencies and parse-time validation.

The ParTS project has delivered a new parser library that accomplishes these goals. We achieve this with a parser combinator approach that uses *partial evaluation* for performance, *type classes* to support formats of varying complexity without sacrificing speed on simpler ones, and *fusion optimizations* to turn declarative multi-stage specifications into efficient single-pass implementations. These advances are described in more detail in the remainder of this document.

To promote collaboration and impact the broader research community, we have made the ParTS library available publicly under a permissive open-source license:

`https://github.com/draperlaboratory/parts`

In Section ii, we examine a key piece of relevant prior work and the core technical challenges in adapting it to meet our goals. In Section iii, we outline the methodology and project plan we employed to overcome those challenges and demonstrate success. In Section v, we describe in detail the results of the project, including the technical innovations that resulted in success and our empirical evaluation. In Section v we summarize the important findings.

ii Technical Problems/Challenges

Describing the technical challenges associated with the ParTS project requires a small amount of background. We begin by reminding the reader of the advantages and disadvantages of parser combinators (Section ii.A), then describe a key piece of recent work that we build on in ParTS (Section ii.B), and finally identify the core challenges remaining in adapting that work to achieve our goals (Section ii.C).

ii.A Parser Combinators

To achieve the goal of being easy to use, we move away from the approach of using an external parser generator tool, and instead proposed to build a library of *parser combinators*. Parser combinators are an idea that dates back to the 1970s [7] but saw a major boost in popularity with the release of modern libraries for functional languages, like Haskell's Parsec [8, 9].

A parser combinator library provides the user with a collection of building blocks from which to construct their parser. Thus writing a parser has the feel of a programming activity rather than formal grammar specification, which is appealing to many programmers in practice. This has resulted in wide use of parser combinators in the functional programming world, despite their relatively poor performance. Even the original paper on monadic parser combinators in Haskell acknowledges that its technique "lacks the efficiency of bottom-up parsers generated by machine" [8], and we have already observed that machine-generated parsers are too slow themselves!

ii.B A Partial Solution

To overcome the challenge of retaining the appeal and ease-of-use offered by parser combinators while providing good performance, we are heavily inspired by the recent paper "A Typed Algebraic Approach to Parsing" [10], which we will refer to as "TAAP" in this document.

TAAP offers a parser combinator library with a type system that enforces that parsers use at most a single token of lookahead. Their library is implemented in the "MetaOCaml" metaprogramming variant of OCaml [11], which enables a *multi-stage programming* implementation where parser

evaluation occurs over several stages, some of which can take place at compile time. At a high level, this process has three key components:

- Their parser combinators are not implemented directly, but rather in terms of a custom "virtual machine" specialized for parsing. This machine has primitive operations like peeking at or dropping the next token in the input stream. The machine code can be thought of as an intermediate domain specific language for parsers.
- They implement an interpreter for this virtual machine in MetaOCaml. This is a function that translates virtual machine code into a runnable parser. Their interpreter makes crucial use of information provided by the static type system that enforces the "single token of lookahead" requirement to provide an efficient implementation.
- Using MetaOCaml's multi-staged programming facilities, they generate an implementation for a given parser by *partially evaluating* the application of the interpreter to the generated virtual machine code for a given parsers.

The TAAP authors themselves describe this approach slightly differently, combining the second and third bullets above into a single step they call code generation. We find it enlightening to distinguish the implementation of the virtual machine from the MetaOCaml compilation stage. This makes it clear that the approach is a use of the so-called "first Futamura projection," a classic technique in program optimization via partial evaluation [12].

ii.C Core Remaining Obstacles

The TAAP work demonstrates that parser combinators can achieve performance that is better than parser generators, with clever use of partial evaluation. However, there are three key limitations in this work that make it impractical for widespread use. Designing a new approach that overcomes these three limitations is the core technical challenge for ParTS.

Challenge 1: Support more formats without sacrificing performance. The TAAP combinators restrict the programmer to using a single token of lookahead. They also do not include a general notion of parser state, often found in parser combinator libraries and essential to implement context-dependent parsers. ParTS seeks to generalize the TAAP library to support arbitrary formats.

A core challenge here is that the TAAP type system which enforces the lookahead limitations also tracks key information required by their efficient implementation. To retain TAAP's high performance, we must find a middle ground where our implementation can statically identify when a format requires limited lookahead and use this information to generate highly performant implementations, while still allowing arbitrary formats with less performant implementations.

In ParTS, we seek to go even further: as we will describe in Section iv.A, we have observed that many real-world formats *mostly* use little lookahead, but have small sections that require more complex parsing strategies. Rather than selecting a single algorithm based on the most complex part of the format, we want to statically identify which parts of a format can be parsed efficiently and limit the use of expensive techniques to the portions of the format that actually requires them. This will allow us to generate code that resembles what a programmer hand-writing a parser for speed would do.

Our solution to this challenge is described in Section iv.E, where we show that typeclasses can be used to statically track the lookahead requirements of parsers and provide efficient implementations on an extremely find-grained basis.

Challenge 2: Enable clean, staged specifications with fusion. In practice, parsers often have several conceptual stages: First the input stream is transformed into a sequence of tokens, then these are parsed into a high-level data structure, then various validation steps may be performed. It is important to allow programmers to specify these stages separately—combining them manually into a single pass results in complicated, challenging-to-understand code which can lead to bugs and vulnerabilities.

However, the most performant hand-written parsers *do* perform all stages simultaneous, for two reasons. First, explicitly implementing a parser as a series of individual stream transformations has memory overhead for intermediate data structures and small performance penalties. Second, by implementing all stages in one pass, the parser can fail early if there is an error early in the input, or succeed early if the high-level client code requires only a portion of the data in the stream.

Thus, a key challenge for ParTS To enable clean, staged specifications while achieving the performance of single-pass implementations. Our solution to this challenge is described in Section iv.D.6, where we find that this can be accomplished by carefully structuring our partial evaluation approach to achieve an effect similar to *fusion* or *deforestation* compiler optimizations [13].

Challenge 3: Support multiple backends and verification. The TAAP work makes crucial use of the MetaOCaml multi-staged programming tool. MetaOCaml is a research tool only available as an extension to a limited number of specific versions of the OCaml compiler, and can be challenging to install and use. Its use limits the potential applicability of the TAAP library. At the same time, most programming languages do not have sophisticated staged programming or partial evaluation support. These are crucial aspects of the TAAP approach, so a key challenge for ParTS is to find an alternative strategy for achieving them and move away from MetaOCaml.

Our solution to this challenge is described in Section iv.D, where we describe our use of the Coq programming language and interactive theorem proving environment. We find that the rich features of this language can be used to achieve a similar partial evaluation strategy. Additionally, this potentially enables us to offer multiple backends compatible with different programming languages, as Coq directly supports "extraction" of its code to OCaml and Haskell, and other work has found that it is possible to directly generate low-level code suitable for linking with arbitrary programs from within Coq [14]. Finally, working in Coq offers the opportunity to pursue verification of this library in the future.

iii General Methodology

The ParTS project took place in three steps, and we outline the approach for each.

- **Step 1: Format Survey.** As the goal of ParTS is to create a usable, practical tool, we began by examining nine real-world formats drawn from three broad categories: document formats, "on the wire" formats, and data exchange formats. We conducted a survey which identified the particular complexities of each format, some of which require techniques that go beyond what is traditionally considered "parsing" in the literature. The survey categorized

these complexities and considers suitable implementation techniques for a domain-specific language of parser combinators. This resulted in requirements for the design and implementation tasks that followed in the remainder of the project. The survey was delivered to DARPA early in the project, and its results are included in Section iv.A.

- **Step 2: Library Design and Implementation.** The second step was to build our library. We began by carefully reviewing the core elements of the TAAP work. We considered how best to translate them to our setting (Coq) and to add support for more complex lookahead and fusion. We iterated through several designs, conducting small experiments to evaluate feasibility and performance of alternate approaches, some of which are further described in Section v below. We discussed our work with the TAAP authors and integrated their advice. Eventually, we arrived at a complete implementation and began evaluation.

- **Step 3: Evaluation and Refinement.** Once we had a complete library that supported extraction and was sufficiently expressive to reimplement the core TAAP benchmarks, we began evaluation in earnest. To evaluate the library, we reimplemented two benchmarks from the TAAP paper with our combinators and ran the TAAP code itself on the same computer to compare results. Initial results were predictably quite poor, with large overheads relative to TAAP, and we proceeded by using profiling tools and comparing our generated code with the intermediate stages stages of the MetaOCaml TAAP compilation process. We adjusted our parser DSL interpreter incrementally, eventually out-performing TAAP on its benchmarks.

iv Technical Results

iv.A Format Survey Results: Design Considerations for Parsing Tools

Our first task in the ParTS project was conducting a survey of various real-world formats. The purpose of the survey was to identify key features that occur in real-world languages but are not handled well by current parser generator tools. As the goal of this project is to design a library suitable for real-world use, these features serve as requirements.

The survey covered three broad categories of formats and pulled several examples from each category. The intent was to capture a broad range of standard parser uses. The categories and specific examples were:

- **Document formats**: HTML, PDF and Markdown
- **"On the wire" formats**: DNS, UDP, and JPEG
- **Data exchange formats**: WAV, XML, JSON and Protobufs

The survey provided a high-level overview of the structure of each of these formats and highlighted key unusual or complex aspect in each format that could provide a challenge for parsing tools. We found that:

1. Many languages contain a simple *core* which is context free.
2. *Data dependencies* are the major source of context sensitivity, and can be classified into a few major types.
3. Parsing is naturally broken into *phases* which deal with different types of tasks. These phases may include lexing, parsing, validation and error recovery.

Rather than recapitulating here the details of each format, we elaborate on these findings and the describe the overall conclusions of the report.

iv.A.1 The Language Hierarchy

Given the maturity of the classic theory of language hierarchies, it is a useful exercise to examine which language features do not fall clearly into one of its portions. We briefly recall the categories of the Chomsky hierarchy:

1. Regular Grammars: These are the languages that can be parsed with an automaton, with no notion of state.
2. Context Free Grammars (or CFGs): These languages can be parsed using an automaton which has access to a finite stack.
3. Context Dependent Grammars: The machines which recognize these languages are more complex, but they roughly correspond to machines which have access to a finite amount of general purpose memory, bounded relative to the size of the input.

4. Recursively Enumerable languages: These languages are extremely general, and roughly correspond to the upper limit of what one may hope to recognize. Recursively enumerable languages are those where valid strings can be recognized by a Turing machine.

Despite the fact that parser generator tools are highly geared towards context free grammars, only one of the languages we surveyed (JSON) falls clearly into this category. Even JSON is no longer context free if one adds the requirement to check for duplicate keys.

We will elaborate on the failures of CFG to capture real-world formats in more detail, but the main obstruction is data-dependencies, which enable using parsed data to determine how to parse subsequent fields.

Much more flexible are the context-dependent grammars, but nevertheless there are features in most of our examined languages which are either impossible or inconvenient to capture using this formalism. Notable examples include data dependencies like integers representing byte-lengths of subsequent fields, and acyclicity requirements for references. Even the features within this class do not seem to be particularly naturally expressed in these terms, like fields defining character encodings, which look like a messy duplication of rules when expressed as a context-dependent grammar.

Thus, the real-world formats we encountered nearly all reside in the most expressive category of the Chomsky hierarchy. As a result, the hierarchy is not a very useful tool for categorizing complete formats. However, as we discuss below, it may be useful for categorizing *portions* of formats. This is not surprising, as real formats are often designed to minimize resource and time usage. They can often be parsed in a single pass, with higher-complexity constructs being restricted to small sub-sections of the total input.

iv.A.2 Context Free Parts

Most of the formats we considered actually contain a context-free *core*, despite not being context free overall. This roughly breaks the document or data into high-level *chunks*, which can then be either validated or further refined by non-context free operations.

We give some illustrative examples:

- XML: the description of the format explicitly describes a CFG (using BNF notation) which acts as a template for the first phase of parsing. Of course, this phase is not expected to be done entirely at once, but interleaved with the other aspects of the parsing task.
- JFIF: despite there being a length field for packets, there is a clear delimiter for the end of the packet itself, enabling to disregard this data dependency.
- PDF and JSON: Both formats contain dictionaries, which may either contain circular, undefined references or duplicate keys. However the formats explicitly do not enforce these "sanity checks", restricting themselves to the context-free fragments in order to give implementers the freedom to handle these more complicated aspects in the way that they desire, possibly as a post-hoc or optional validation phase.

iv.A.3 Data Dependencies

Data dependencies are the main source of complexities in the non context-free parts of the data. An interesting aspect of the data dependencies that appear in practice is that they fall into a small number of rough categories. These are:

- Size and depth information
- Ordering or sequencing information
- References linking to previous entries
- Finite state information which determines subsequent parsing actions

Here are a few examples where these constraints appear:

- Many parse steps have a data field that specifies an integer length for subsequent data. Examples include UDP bodies, JFIF packets, DNS labels and PDF stream lengths.
- In XML, tags may contain arbitrary data, but beginning and ending tags must correspond to each other, which means that beginning and ending tags must be balanced, in a first-in-last-out manner.
- JSON implicitly suggests that the structures described by maps and lists be turned into map-like and list-like data structures in the source language. Similar implicit (or explicit) demands appear in HTML and PDF. XML has references that associate a key to a value, to be processed as if the value itself appeared in that spot (with some caveats!).
- Many formats have conditional decisions. For example, "if a certain tag appears, process the remainder of the input in a certain way". HTML even may change the character encoding within the document! This configuration data tends to be *write once*, i.e. the data as read determines the subsequent behavior of the parser for all subsequent subtrees of the parsed data.

One of these items stands separate from the others: the ability to specify parsing "modes", i.e. state which determines the parsing action to be taken next, based on the previous results. This requires some conditional change of the *state* of the abstract machine which carries out the processing, in a way that is independent of the input token directly (as is the case with regular and context-free languages).

In contrast, actions like reference lookup or repeated actions bounded by some integer remain in the same high-level parser state, with a simpler state update, (e.g. locating the reference definition or decrementing a counter).

Somewhat compellingly, there are clear correspondences between the types of constraints outlined above and natural data structure which might be used to implement them, which we outline in table form:

Data constraint	Implementation
Size/depth	Integer
Sequencing	Stack/Queue
References	Dictionary
Finite state info	Case statement

These data structures can be maintained in a `Reader`- or `State`-style monad.

iv.A.4 Validation

The data dependencies outlined above require analyzing parsed data in order to determine which parsing *action* to take. A very common such action is to interrupt the parsing process, or continue without change. To put this another way, there are often implicit or explicit constraints on the parsed data in order to determine whether a given input is *valid* or not. These constraints are most easily specified as occurring on a completely parsed data structure, even when it is more efficient to perform them during the parse (e.g., checking for duplicated keys).

Examples include:

- Fixed bounds on sizes of objects (e.g. string lengths in PDF) or depths of object nesting (implementation defined quantity in JSON)
- Non-cyclicity of data (ID references in PDF)
- Ordering constraints (references in XML) or uniqueness constraints (in various formats)

These are quite similar to the data constraints from the previous section, except that they could theoretically be ignored to define a more permissive format. How to fail as a result of the checks is often defined by the specification (HTML embraces this practice wholeheartedly).

While the *definition* of these validation constraints is naturally separate from the definition of the parsing task proper, it is clear that the *operational* behavior of the parser should be to reject the input as soon as possible (if the validation is to be enforced). Traditional parser generator tools do not have support for the kind of "fusion" optimizations that could turn staged specifications into single-pass implementations.

iv.A.5 Error and Recovery

Several of the formats we considered have specific requirements for how certain kinds of "invalid" data is handled. These error recovery mechanisms can be organized into two basic categories.

In the first category, we have specifications that require all implementations to correct "invalid" data in a particular way. These invalid encodings are really part of the "de facto" language. Is a serialized protobuf payload with a duplicated field really "invalid" if all legal parsers must accept it and handle the duplicated data in a uniform way? In our view, no. In these cases, "error recovery" may be a convenient way to describe to a human reader what the format allows, but no special mechanism is required for a parser—we can simply add the recoverable cases to the grammar and parse them as specified.

In the second category, we have recovery mechanisms that are outside the scope of a parser, typically because additional information is required. In the case of the JFIF format, an invalid packet

triggers a network communication requesting a re-send of the data, and then parsing starts again from the last synchronization point.

This requires two key things from a parsing tool: support for a rich notion of *semantic actions*, including here network communication, and support for *backtracking* (or restarting). A parsing tool can provide safe support for common semantic actions, but in the most general case the semantic actions required by a format could involve arbitrary computation and network communication. To support real-world formats, parsing tools must allow this potentially unsafe behavior, but should make it possible to disable or mediate the feature in safety-critical settings.

Backtracking, on the other hand, is straightforward to support safely. However, formats that require backtracking cannot be parsed by the simplest and most efficient algorithms, which use finite lookahead. Therefore it is still important to limit its use and alert the user when unbounded backtracking may occur in a parser, or to enable an up-front resource limit.

iv.A.6 Survey Conclusions

Our overview of real-world formats revealed a number of complexities that make them fit quite poorly into the classic approach of regular expression lexer followed by a context-free LL(k) or LALR(1) parser. As we mentioned earlier, many languages contain context-free cores which cannot exactly recognize the full language but serve as a useful implementation guideline. Parsing is often conceived as a 3 part process:

1. Lexing, which usually consists of recognizing a regular language, plus (possibly) some limited non-context-free constructions.

2. Parsing, often of a language which is *almost* context free, but has some data dependencies requiring state.

3. Validation and semantic actions, which is done using the full power of a general purpose language.

In addition, it often is the case that while a certain field is data dependent, it is possible to at least *bound* the data dependency to a certain subsection of the input by using data-independent techniques, like the JFIF null termination of fields, or DNS names.

A characteristic of the implementations of these languages is a tendency to be able to process a stream of data continuously, consuming a finite amount of input before taking a semantic action (or produce a token for the following processor). This view has advantages in terms of data locality and space usage. Some standard techniques in data processing and functional programming seem particularly appropriate for this goal.

We believe that this compositional structure can be made both more precise and more efficient by fleshing out each phase with the specific capabilities described in the previous sections. We plan to build a DSL that implements and optimizes this model.

iv.B High-level Parser Combinators

We next introduce the high-level interface for our parser combinators and show some example parsers. This interface is similar to other parser combinator libraries, with a few tweaks for the Coq setting and our efficient evaluation strategy.

We begin with the types. The type of parsers in our library is:

```
Machine rv st tok tag out
```

We select the name `Machine` to reflect the evaluation approach—parsers actually evaluate to programs in a small virtual machine. We save the definition of this type and the virtual machine for the next section, and here focus on examples from the user's perspective.

The `Machine` type has five parameters. Three of them will be familiar to users of other parser combinator libraries:

- `st`—The user-defined state carried by the parser. Our machines implements a state monad, much like Haskell's Parsec library. Including state permits context-sensitive decision making in the parser, including data dependencies.
- `tok`—The type of tokens the parser consumes.
- `out`—The result type of the parser.

The two other parameters are more technical. The first `rv`, is due to our use of the "PHOAS" technique [15] to encode variable binding in the internals of the virtual machine. This technique is common in Coq encodings of languages, and will be described further in the next section. The user never needs to interact with this type, and can always leave it abstract. The second, `tag`, relates to our implementation of an optimization from the TAAP library. We explain it in detail in the next section. For the examples in this section, and for most parsers, `tag` can be the same as the token type `tok`.

To illustrate our library, we will show how to build one of the parsers used as a benchmark in the TAAP paper: an s-expression parser that counts the number of atoms. We will do this using a two-stage approach, first lexing into tokens and then parsing the tokens to check for a valid s-expression and count the number of atoms.

iv.B.1 A Lexer for S-Expressions

We begin by defining a type of tokens for our lexer to produce, using a standard Coq datatype. There are three possible tokens: atoms, and left and write parentheses.

```
Inductive sexp_tok : Set :=
| Atom : sexp_tok
| LParen : sexp_tok
| RParen : sexp_tok.
```

We next define a parser that recognizes each of the tokens. We need three simple combinators from our library. First, "^", which we pronounce "exact". It recognizes a specific token and returns it [1]:

```
(^) : ∀ rv st tok tag,    tok → Machine rv st tok tag tok
```

We see here that ^ works for any selection of the rv, st, tok and tag type parameters, has one argument (the token it recognizes) and returns a token (because the out type parameter is tok).

Second, we need a sequencing combinator @>. This infix combinator takes two parsers as arguments. It executes the first parser followed by the second (if the first is successful), and throws away the result of the first.

Finally, we also need a way to return a result. The combinator Return consumes no input and returns its argument as the result of the parse.

We can now build parsers for the left and right parentheses tokens as follows:

```
Definition tok_left {rv st} : Machine rv st ascii ascii sexp_tok :=
  ^"(" @> Return LParen.

Definition tok_right {rv st} : Machine rv st ascii ascii sexp_tok :=
  ^")" @> Return RParen.
```

Each parser recognizes a specific character (either "(" or ")") using ^ and then "return"s the appropriate token. The types of these parsers indicate that they parse characters (ascii is Coq's name for characters) and return sexp_toks. They work for any selection of the rv and st type parameters, hence the {rv st} parameters.

Next we build a parser that recognizes atoms. Atoms are identifiers—sequences of letters and number beginning with a letter. Parsers for these primitives are included in our library, but we show how they are defined to provide an example. The library has a one_of combinator that recognizes any token from a given list:

```
oneof : ∀ rv st, list tok → Machine rv st tok tok tok
```

We can use this, combined with a Coq function to turn a string into a list of characters, to build a combinator that recognizes any character from a given string:

```
Definition charset {st rv} (s : string)
           : Machine rv st ascii ascii ascii :=
  one_of (list_ascii_of_string s).
```

This combinator can then be used, for example, to build a parser for any letter:

```
Definition alpha {st rv}
           : Machine rv st ascii ascii ascii :=
  charset "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ".
```

[1] In this section, we elide the type class parameters when showing the types of combinators. These type class parameters appear in signatures but are automatically inferred by Coq when the combinators are used, so we defer their introduction until the next section.

Following the Parsec combinator library, we define variants that recognize but throw away the relevant tokens, for convenience. We name these variants with an additional underscore. We also have similar parsers for digits and any alphanumeric characters. The library includes all the combinations:

```
charset : ∀ rv st,  string → Machine rv st ascii ascii ascii
charset_ : ∀ rv st, string → Machine rv st ascii ascii unit

alpha : ∀ rv st, Machine rv st ascii ascii ascii
alpha_ : ∀ rv st, Machine rv st ascii ascii unit

digit : ∀ rv st, Machine rv st ascii ascii ascii
digit_ : ∀ rv st, Machine rv st ascii ascii unit

alphanum : ∀ rv st, Machine rv st ascii ascii ascii
alphanum_ : ∀ rv st, Machine rv st ascii ascii unit
```

Finally we introduce a combinator that iterates a parser. Again we have two variants, one that returns a list of the results, and another that discards them, for convenience:

```
star : ∀ rv st tok a, Machine rv st tok tok a
                    → Machine rv st tok tok (list a)

star_ : ∀ rv st tok a, Machine rv st tok tok a
                     → Machine rv st tok tok unit
```

Now we can define a parser for atoms:

```
Definition tok_atom {rv st} : Machine rv st ascii ascii sexp_tok :=
  alpha @> (star_ alphanum) @> return_ Atom.
```

This parses a letter followed by a sequence of alphanumeric characters and returns the token `Atom`. Because we will only count the number of atoms, we do not save the string in this example, but a similar parser could be defined to do so.

We can put our definitions together to parse any token:

```
Definition lex_one {rv st}
           : Machine rv st ascii ascii sexp_tok :=
  tok_left <|> tok_right <|> tok_atom.
```

Here, <|> is a combinator that parsers either of two alternatives. Finally, to build a tokenizer, we also want to allow for whitespace. Our library includes a whitespace parser, defined similarly to `alpha` and `alphanum`:

```
whitespace : ∀ {rv st}, Machine rv st ascii ascii unit
```

We allow each token to be preceded by whitespace:

```
Definition sexp_lexer {rv st}
           : Machine rv st ascii ascii sexp_tok :=
  whitespace @> lex_one.
```

Sequencing of this into multiple tokens could be done with `star`, but when composing parsers this is handled for the user by our library. We first show the parser for s-expressions, and then show how it is composed with the lexer and compiled.

iv.B.2 A Parser for S-Expressions

We now parse s-expressions. In these parsers, our token type will be `sexp_tok` rather than `ascii`, since our parser operates on token streams constructed with the use of the lexer from the previous section.

The grammar for s-expressions we will parse is:

```
sexp ::= atom | ( sexp * )
```

The main novelty of the parser is the need to introduce an explicit recursion, to handle the nested nature of s-expressions. Our library has a simple combinator for defining recursive parsers: `parse_fix`. This is a standard fixed-point combinator—its argument is a function that constructs a parser given the result of recursive call. It is perhaps best explained by example, so here is our parser to compute the number of atoms in an s-expression:

```
Definition sexp_parser {st rv} : Machine rv st sexp_tok sexp_tok nat :=
  parse_fix (fun sexp ⇒
        (^Atom @> Return 1)
    <|> (sum <$$> delimited_list LParen (Var sexp) RParen)
  ).
```

This parses an lone atom (returning 1), or a sequence of s-expressions within parentheses producing a list of integers which are then summed to arrive at the total number of atoms. The variable `sexp` is used for the recursive call, which here will compute the number of atoms in a nested s-expression. The recursion is constructed by the `parse_fix` combinator. This parser uses `^`, `@>` and `<|>`, which also appeared in the lexer, and introduces a few new combinators:

- The `<$$>` combinator is *map*. It applies a function to the result of a parser. In this case, we use the Coq function `sum` which computes the sum of a list of numbers to accumulate the number of atoms in each s-expression in the sequence.
- The `delimited_list` combinator is similar to `star`. It parses a list delimited by provided tokens (here the left and right parenthesis tokens).
- The `Var` constructor is how the a parser defined using `parse_fix` makes its recursive call.

iv.B.3 Composing and the Parser and Lexer

The final step is to compose the parser and lexer. It is possible to run these machines individually and send the output from the lexer into the parser. However, for maximum efficiency, we include a composition operation that can fuse the two parsers together and execute them in lockstep, sometimes eliminating intermediate data structures. The details of these optimizations are discussed below—here we simply show how to compose.

```
Definition sexp_complete : stream ascii → nat :=
  compose_simple sexp_lexer sexp_parser.
```

The composition operation `compose_simple` handles sequencing the lexer and pasing its output into the parser. Note it also transforms our machine into a function from streams to results. Here we have used a simplified composition operation for the common case where neither the lexer or parser use state. The library also includes a generic composition that handles arbitrary parsers and requires the user to provide the initial state.

The implementation of the virtual machine evaluation, fusion, and composition are discussed in Section iv.D. The performance of this s-expression parser is quite good, substantially beating the TAAP parser. Our benchmarks are discussed in Section iv.F.

iv.C An Intermediate Virtual Machine for Parsers

The parser combinators described in the previous section actually evaluate to programs in our intermediate virtual machine, which is subsequently executed using a partially evaluated interpreter and extraction to Coq. In this section, we show the machine's definition and explain its core operations. In Section iv.D we describe intepreters and partial evaluation for efficient machine execution.

The virtual machine is defined as a Coq datatype, with constructors representing each "instruction". Its definition is shown in Figure 1.

In the remainder of this section, we will describe each of the core machine operations and how they are used by combinators. We described most of the parameters of the `Machine` type in the previous section, and will examine `rv` and `tok` in more detail when we examine the constructors that use them. The `Machine` defintion closely resembles the intermediate languages used in TAAP, and we will highlight some of the differences.

```
Inductive Machine (rv : RecVar) (st tok tag out : Type) : Type :=
(* Ending execution *)
| Return : out → Machine rv st tok tag out
| Error : string → Machine rv st tok tag out

(* Manipulating the input stream *)
| Peek   : (SetCompl tag)
         → Machine rv st tok tag out
         → Machine rv st tok tag out
         → Machine rv st tok tag out
         → Machine rv st tok tag out
| Return_Drop_Tok : (tok → out)
                  → Machine rv st tok tag out
| Drop : Machine rv st tok tag out → Machine rv st tok tag out

(* Manipulating the state monad *)
| Read : (st → Machine rv st tok tag out) → Machine rv st tok tag out
| Write  : (st → st)
          → Machine rv st tok tag out
          → Machine rv st tok tag out

(* Composition and recursion *)
| Call : ∀ out' , Machine rv st tok tag out'
               → (out' → Machine rv st tok tag out)
               → Machine rv st tok tag out
| Fix  : (rv st tag out → Machine rv st tok tag out)
       → Machine rv st tok tag out
| Var : rv st tag out → Machine rv st tok tag out .
```

Figure 1: Definition of `Machine`.

iv.C.1 Operations for Terminating and Producing Output

The first two operations allow a parser to conclude execution successfully or unsucessfully:

```
| Return : out → Machine rv st tok tag out
| Error : string → Machine rv st tok tag out
```

The `Error` operation immediately terminates execution of the `Machine`, and can not be caught. The `Return` operations ends execution and yields a value. Parsers are often constructed by *sequencing* many smaller parsers, each of which yields a value that is used by the next. For example, our s-expression lexer from the previous section was constructed out of smaller pieces that each yielded a value individually, like this parser for left parentheses:

```
Definition tok_left {rv st} : Machine rv st ascii ascii sexp_tok :=
  ^"(" @> Return LParen.
```

The `Call` and `Fix` constuctors can both be used to obtain the result of a succesfully terminated machine and use it in a subsequent parser, and are described below.

iv.C.2 Operations for Manipulating the Input Stream

These three operations permit a parser to examine and manipulate the input stream:

```
| Peek   : (SetCompl tag)
         → Machine rv st tok tag out
         → Machine rv st tok tag out
         → Machine rv st tok tag out
         → Machine rv st tok tag out
| Return_Drop_Tok : (tok → out)
                   → Machine rv st tok tag out
| Drop : Machine rv st tok tag out → Machine rv st tok tag out
```

The simplest is `Drop`. The parser `Drop p` advances one token in the input stream and then continues by executing the parser `p`.

The parser `Peek br p1 p2 p3` examines the next token in the input stream and makes a choice of which of `p1`, `p2` and `p3` to execute next based on the user-provided branch condition `br`. The machine executes `p1` if the branch condition is true, `p2` if it is false, and `p3` if we are at the end of the input stream.

The branch condition's type, `SetCompl tag`, bears some explanation. In our first versions of the library, the branch condition was instead expressed as a list of tokens `list tok`. If the stream's next token was in the list, the branch condition was true, and otherwise false. We quickly discovered a key problem with this approach. The `SetCompl tag` type adapts and generalizes a technique we found in the TAAP library for dealing with it.

The problem is that, in general, token types are not finite. For example, in our JSON parser, tokens contain arbitrary strings and integers. Thus, it is often impossible to build an explicit list or set that captures the collection of possible tokens we want for our branch condition. An obvious alternative is to use a function `tok → bool` to make the branching decision. However, this interacts poorly with partial evaluation. With a finite branch condition, partial evaluation can produce an

efficient branching structure and propogate information from each branch into subsequent parsers (for example, eliminating the need to peek twice at the same stream).

The solution, therefore, is to *map* from the potentially infinite token type to a finite domain that captures the information relevant for branching decisions. In the case of JSON, for example, we may want to know whether the next token is a string to make a parsing decision, but we do not need to know exactly which string it is. Therefore, we can project from the actual type of JSON tokens to a type where the data is removed and only the token shapes remain, and use this for branching decisions.

We call the finite projection of the token type used for branching the *tag* type, explaining the `tag` parameter to the `Machine`. When the token type is finite (as in the case of characters), it can be used as the tag directly. Otherwise, a separate tag type and a projection from the token type to tags is defined. The TAAP work also uses this approach.[2] Their `peek` operation takes a set of tags. We generalize this somewhat with the `SetCompl` type, which expresses a finite set or a finite set complement:

```
Inductive SetCompl (tag : Type) : Type :=
| is_set : list tag → SetCompl tag
| is_compl : list tag → SetCompl tag.
```

We find it convenient to also allow the use of set complements for branch decisions, to encode cases where we want to take parsing decision unless some specific token or tokens are seen. For example, the `parse_delimited_list` combinator from our s-expression parser example is implemented with the `is_compl` constructor to parse a list until the specific end delimiter token is seen. This smaller encoding of complements may also contribute to part of our performance advantage over TAAP, although we have not measued it in isolation.

To provide the mapping between tokens and tags, we use a type class we call `BInfo`, for "branch information":

```
Class BInfo tok tag : Type :=
{
  take_branch : tok → (SetCompl tag) → bool;
  intersect : SetCompl tag → SetCompl tag → SetCompl tag;
  union : SetCompl tag → SetCompl tag → SetCompl tag;
  compl : SetCompl tag → SetCompl tag;
  subset : SetCompl tag → SetCompl tag → bool
}.
```

A `BInfo` class instance defines how to compare a token with a `SetCompl` of tags and arrive at a boolean parsing decision (the `take_branch` field). It also contains functions for manipulating `SetCompl`s with standard set operations like union and intersection. These are used by the combinators and machine evaluation code to propogate branching information for efficiency.

Combinators that make branching decisions, like <|>, have types which require a `BInfo` instance for the token and tag types used by the parser. We have provided a collection of common instances in

[2]This approach is not described in their paper, but we found it in their code while exploring efficiency differences with early versions of our library.

our library (for example, handling the case where the token type is finite and can be used directly as the tag type).

The generality of the `BInfo` approach also offers some opportunities for optimization. In inspecting the TAAP code we noticed that they generate extremely efficient branch conditions for character sets. For example, the high-level definition of the `alpha` combinator from our s-expression is a nested alternative that is 52 branches deep (one for each upper and lower-case letter). We observed that they use the same high-level definition, but were somehow generating an extremely efficient branch condition in OCaml: for a token `t`, their branch condition was:

```
('a' <= t && t <= 'z') || ('A' <= t && t <= 'Z')
```

This branch condition is substantially faster than the naive implementation of a deeply nested "or." Looking back at their code, we discovered they have a machine optimization pass that looks for deeply nested alternatives over a continuous range and collapses them into the efficient branches above.

After discovering this optimization in the TAAP code, we found we were able to implement the same idea using our generic `BInfo` interface. As a result, we also generate extremely efficient branch conditions for character parsers.

One limitation of the `Peek` interface is that it does not provide the specific value of next token to the subsequent parsers. This limitation is also present in the TAAP machine, and is a key contributor to efficiency. In many situations, the knowledge of whether the next token is within some some specific set is sufficient to continue the parse without examining its exact value. Performing computation on the specific token value has the potential to limit how much information propogation can be done during partial evaluation.

We limit the use of the specific token value to where it is strictly needed by using a separate operation for this purpose: `Return_Drop_Tok`. This operation applies a user-provided function to the token and returns the result. It is used where parsers read specific data values from the input stream, like strings and integers values in JSON.

We also have support for lookahead and backtracking in the input stream. This allows languages that require arbitrary lookahead, but we have taken care to restrict its use so that we still generate efficient parsers for languages that do not need its generality. The `Machine` operations for general lookahead are describe in Section iv.E as an extension of this `Machine` type.

iv.C.3 Operations for Manipulating the State Monad

Our parser combinator language implements a state monad for arbitrary user-defined state. We have two `Machine` operations for interacting with the state:

```
(* Manipulating the state monad *)
| Read : (st → Machine rv st tok tag out) → Machine rv st tok tag out
| Write  : (st → st)
          → Machine rv st tok tag out
          → Machine rv st tok tag out
```

The `Read` operation allows the user to provide a function that examines the current state and produces a parser, which is then executed. The `Write` operation allows the user to provide a function

that transforms the state, and a parser to execute once it has been updated.

State monads are common in parser combinator languages. They provided a generic interface for context-dependent parsing tasks. For example, a parser which checked whether a JSON record had duplicated field names could keep track of the preveiously-seen names in the state.

iv.C.4 Operations for Composition and Recursion

Finally, we have operations for sequencing parsers and defining recursive parsers:

```
(* Composition and recursion *)
| Call : ∀ out',  Machine rv st tok tag out'
              → (out' → Machine rv st tok tag out)
              → Machine rv st tok tag out
| Fix  : (rv st tag out → Machine rv st tok tag out)
       → Machine rv st tok tag out
| Var : rv st tag out → Machine rv st tok tag out
```

The `Call p f` operation executes the parser `p`, then provides its result to the function `f` to obtain a second parser which is then executed. This both provides a means to sequence parsers and provides a means for parsers to depend on previous results.

The `Fix` and `Var` constructors implement recursive parsers. Here we use the "PHOAS" technique [15] to overcome the limitations of Coq's powerful but sound type system. The most natural type for `Fix` would be:

```
| Fix  : (Machine st tok tag out → Machine st tok tag out)
       → Machine st tok tag out
```

However, this definition is rejected by Coq. The type is not *positive*, a technical restriction that is required to maintain the consistency of Coq's logic. The PHOAS technique overcomes this limitation by adding an extra type parameter, which we call `rv`, which explicitly encodes "variables" representing the result of recursion. These are included in terms via our `Var` type.

The details of the PHOAS technique are complex, but its complexities only arise when defining recursive functions that pattern match on the `Machine` type. This is necessary for our `Machine` interpreter, but parsers and combinators themselves never introspect machines. As a result, users of the library never need to cope with the complexity of PHOAS, and can use the straightforward interface we have provided in combinators like `parse_fix` from the s-expression example to build recursive parsers.

iv.D Virtual Machine Interpreter and Partial Evaluation in Coq

Writing a simple interpreter for the virtual machine for the purposes of running within Coq is straightforward exercise. Each constructor corresponds to a natural operation on the stream (which can be implemented as a list) and the state of the machine. However, with the goal extracting these machines to efficient partially evaluated OCaml code, the situation is more complicated and a number of key tricks.

iv.D.1 Runtime

There are some very useful features of OCaml that are less direct to model in or to extract from Coq. These include opening files, mutable references, unbounded recursion, array indexing, and error throwing. Using these features is essential in order to achieve competitive performance in comparison with libraries that do use them. In order to use these features, a small OCaml runtime library was used whose interface was modeled by Coq axioms. Coq extraction to OCaml provides a mechanism for replacing the axioms with their OCaml implementations.

iv.D.2 Unbounded Recursion

One big difference between OCaml and Coq is that Coq requires functions to be terminating by default. Unfortunately, our machine interpreter is not obviously terminating. The input streams it processes are potentially infinite, and the parsers themselves are defined with potentially unbounded recursion.

A standard technique to achieve this in situations where termination is not obvious is to use "gas," an integer that is decremented on every step of the computation, which terminates when it reaches zero. An efficient OCaml parser will of course not carry around this unnecessary data nor do this unnecessary decrement computation. A simple method by which to express this nontermination in Coq is to axiomatize a fix function in Coq and extract it to an OCaml implementation in the runtime library. It was determined upon experimentation that this resulted in significant performance overhead, so instead the Coq fix operator was used with guard checking unset in the extraction evaluator code.

In the future, we plan to explore alternatives, which will be necessary to verify the implementation. For example, gas could be used, with more careful extraction that removes it. There is also an argument to be made for parsers being modelled naturally by co-recursion rather than recursion. However, this approach may complicated the development, hurting usability.

iv.D.3 Continuation Passing Style Evaluator

Continuation passing style (CPS) transformations allow explicit representation of control flow and return points for expressions and are a well known technique in compiler implementation. CPS gives one the necessary flexibility to achieve more partial evaluation [16, 17]. The core idea of CPS is that funtions do not directly return their results—instead, they take an extra argument that tells them what to do with the result. This extra argument, the continuation, is itself a function whose input is the result that would otherwise be returned. This continuation argument intuitively represents "the rest of the program," and makes explicit the program's dependency on the current function's result.

Here is an example of transforming a simple function into CPS. The variable "k" is often used as

the name the continuation:

```
Definition example (x : bool) : nat := 4 + (if x then 2 else 3).

Definition example_cps (a : Type) (x : bool) (k : nat → a) : a :=
  let k' := fun n ⇒ k (4 + n) in
  if x then k' 2 else k' 3.

Eval cbv in example.
(*
     = fun x : bool, S (S (S (S (if x then 2 else 3))))
     : bool → nat
*)

Eval cbv in example_cps.
(*
     = fun (a : Type) (x : bool) (k : nat → a), if x then k 6 else k 7
     : forall a : Type, bool → (nat → a) → a
*)
```

We found that writing our machine interpreter in continuation passing style gave us more control over how it interacted with partial evaluation by making the control flow explicit. This is also discussed in the TAAP work, and is related to an observation in the literature that CPS helps to disentangle the static and dynamic aspects of a program [18].

It is folklore in the functional programming community that writing libraries in continuation passing style results in performance benefits. Popular combinator libraries like Parsec already make that choice. One perspective on why this is the case that compilers for functional languages include partial-evaluation style optimizations. It is possible that because the ParTS parser combinators are built in a style suited for the purposes of partial evaluation in Coq, that the OCaml compiler has more opportunity for optimization as well. More investigation is required.

The CPS transformation has the unfortunate effect of making code less intuitive. Luckily, the CPS transformation based code in our library is contained completely in the evaluator function we provide. It does not appear in combinator code, the Machine data type, or any other user-facing element.

iv.D.4 The Trick

There is a well known technique for partial evaluation known simply as "The Trick" [19]. Dynamic data can sometimes be made available for static optimizations by enumerating its possible values with a pattern match or if-then-else statement. For example, rewriting a call on a boolean `b` such as `f b` with the semantically equivalent code `if b then (f true) else (f false)` can allow non trivial partial evaluation and optimization to occur within each of the branches. This is at the expense of increasing the size of the code, perhaps exponentially if used without care.

In the ParTS parsers, "The Trick" is used for the branching conditions occurring in the `m1 <|> m2` alternative combinator. While the next token is not known statically, knowing which branch is taken reveals some information about its possible values. We use the trick to propogate that information

into the branches, potentially allowing further optimization. Alongside branch information, the runtime token value is also propagated through the evaluator when possible, in order to eliminate repeated peeking at the same input stream.

iv.D.5 Opacity

The built-in Coq evaluation mechanisms are powerful and engineered to be efficient. However, partial evaluation for the purposes of extraction was not their designed intent. Put another way, Coq is sophisticated at evaluating code, but it is less sophisticated at selectively evaluating code. This issue arises because there are some functions which have both Coq and OCaml implementations where the OCaml version is substantially faster. We do not want the Coq evaluator to "unfold" these functions into their Coq definitions, so we must find a way to make the "opaque." One key example is ascii character equality—Coq's internal representation of characters is an inductive data type with 8 bool fields. If this were to be extracted, it would result in inefficient parsers. Coq does have extraction libraries that correctly translate some of its inefficient datatypes to their more efficient OCaml counterparts. However, this only works if the evaluation mechanism has not unfolded some definition that relies on the inefficient Coq implementations.

Our first approach was to use more conservative evaluation tactics within Coq, like `simpl` and `cbn`. We found that this has two key disadvantages: it produces inefficient code, because not all possible partial evaluation has taken place, and it causes long compile times, because these simpler tactics are not as carefully optimized as their more complex counterparts.

Another approach controling partial evaluation is to explicitly list the functions to be protected from unfolding in the evaluation command `cbv -[ascii_eq_dec]`. We found this to be unacceptable for two reasons: First, we do want functions like ascii equality to unfold *sometimes*, specifically in the cases where they can be completely evaluated at compile time. Second the most performant evaluation tactics `vm_compute` and `native_compute` do not support these annotations.

The most fool-proof method to ensure opacity of definitions is to assert them as Coq Axioms. The Coq system is fundamentally incapable of unfolding these definitions, and they can be annotated to extract to efficient OCaml counterparts. We found that with a few carefully chosen axioms, one can isolate the areas of code that are intended to evaluate at compile time vs runtime.

One particularly useful application of this technique is to define a variant of function application that can not be reduced by Coq evaluation. We call this `let_`, and extract it to the following simple OCaml code:

```
let let_ x f = f x
```

The idea is that the Coq expression `let_ e f` has the same meaning as `f e` or as `let x = e in f x`. However, because we have defined `let_` as an axiom, Coq can not unfold it and perform the evaluation of `f`. Complete inlining at every single possible opportunity was found to lead to excessive code bloat, which in fact decreased the performance of the parsers.

Because eta expansions and inlining of small functions are essential components of any optimization functional compiler, a combinator as small as `let_` will be inlined the OCaml compiler, resulting in efficient code. This can also be enforced by [@@inline] annotations in OCaml.

iv.D.6 Stream Fusion

The most naive implementation of stream processing can result in large intermediate data structures for every stage in the stream processor. This can be avoided by stream fusion techniques.

A stream can be "push" or "pull" based. In a push stream, the stream producer has control of the rate at with the stream consumer executes, while with a pull stream the consumer has control. The ParTS parsers are implemented as pull based streams. The later stage parser requests the processing and production of tokens from an earlier stage lexer on demand.

We also apply existing work on how to achieve partial evaluation across multiple stages of a stream processor [20]. The ParTS library uses a CPS stream representation inspired by the representation found in that work.

```
Class stream (str : Type) (a : Type) : Type :=
  {
    state : str;
    peek_st : forall b, str → (option a → b) → b;
    drop_st : forall b, str→ (str → b) → b;
  }.
```

We found that our partial evaluation techniques sometimes achieve complete "fusion" of parsing phases when used with this stream representation and the CPS-based evaluator. When this occurs, intermediate token datatypes are completely eliminated, resulting in fewer allocations and therefore more performant parsers. We have observed that parsers with complex recursion patterns and those which use many tokens are once are less likely to be fused.

We have also observed that fusion in this sense is not always desirable. In particular, fusion may result in substantial code bloat, where logic that would have been used to identify a token is duplicated at the many locations where the parser expects that token. Our current implementation has attempted to balance these trade offs, but further research and empirical evaluation is required to identify the best strategy.

iv.E Lookahead: Support for More Language Classes

Real-world formats are both heterogeneous, consisting of sub-formats with wildly different complexities, and performance sensitive, making it a requirement to always use efficient and context-aware parsing strategies. Therefore, we want to statically select the most appropriate parsing strategy for each portion of the language, so that the implementation can switch between fast and expressive parsing modes when needed.

Our mechanism for achieving this focuses on how we manage backtracking. While our flexible handling of state enables us to do much while examining a single token at a time in a top-down manner, there are situations where more look-ahead is required in order to make a decision involving the next productions.

To this end, we add a new constructor to our `Machine` type. The code examples in this section are from a branch in our repository that is missing a few updates to the `Machine` type. In particular, the `tok` and `tag` type parameters are replaced by a single `br` parameter that encodes both. This difference is not fundamental, and we plan to merge the two branches in the immediate future, so the examples should be straightforward to understand:

```
Inductive Machine (var : VarType) (st br out : Type) : Type :=
...
| Lookahead : Machine var st br br
              → Machine var st br out
              → Machine var st br out
              → Machine var st br out
              → Machine var st br out
...
```

Contrast this with the type of `Peek`:

```
| Peek : br
         → Machine var st br out
         → Machine var st br out
         → Machine var st br out
         → Machine var st br out
```

The main difference is that instead of having a statically known branch condition, we allow for a machine to *dynamically compute* the branch condition, which will then be used to decide which branch to take.

It may seem like overkill to use the `br` type here, rather than a simple boolean: indeed, since we have all the power of dynamic execution at our command, why bother returning the more complex type, which is only useful at static compilation time to reduce the redundancy in the decisions?

The answer is that we can sometimes *statically* determine the result of the lookahead machine, and in that case we want to be able to propagate this information as usual within the compilation process. As we will see, this will be particularly useful to handle the LL(1) case seamlessly along side more complex cases.

We therefore define the notional semantics of `Lookahead` as follows:

```
Fixpoint eval_m ... :=
```

```
...
| Lookahead m_br m_true m_false m_eof ⇒
    match hd_error input with
    (* No use looking ahead if we're at EOF. *)
    | None ⇒ eval_m n m_eof state input
    | Some t ⇒
      (* We throw away the updated stream here *)
      let '( b_err, _, state') := eval_m n m_br state input in
      match b_err with
      | inl err ⇒ (write_err err, input, state')
      | inr b ⇒
        (* We propagate state, but undo progress in the input to backtrack *)
        if take_branch t b then
          eval_m n m_true state' input
        else
          eval_m n m_false state' input
      end
    end
  end
  ...
```

So we evaluate the `m_br` machine, which will either fail, in which case the whole machine fails, or succeed along with a branch condition `br`, in which case we proceed as in the static `Peek` situation, and evaluate the appropriate branch, but taking care to operate on the *initial* input, rather than the remainder of the stream! This is the crucial point: we cannot save and restore the stream state using any other constructor.

In the extracted OCaml, however, we have an imperative stream state, which is typically just an integer cursor into a buffer. In that case, we of course need to take more care when restoring the state of a stream, since simply passing around a reference will be insufficient.

We work around this issue by adding an explicit operator for saving and restoring stream state to our abstract stream interface. Concretely, we have:

```
Class stream (str : Type) (a : Type) : Type :=
{
  state : str;
  peek_st : forall b, str → (option a → b) → b;
  drop_st : forall b, str→ (str → b) → b;
  (* This operator allows saving and restoring stream state *)
  shift_reset_st : forall b, str → (str → b) → b;
}.
```

Where `shift_reset_st b str k` saves the stream state `str` at the current execution point, and passes it to the continuation `k`, which may then use it as desired.

In the extraction process, we send this constructor to the following function:

```
type 'a stream_t = { mutable pos : int; byte_stream : bytes }
```

```
let ocaml_shift_reset : 'a stream_t → ('a stream_t → 'b) → 'b =
  fun st k →
  let st_restore = { pos = st.pos; byte_stream = st.byte_stream } in
  k st_restore
```

We add the type of streams for clarity. Here we simply do a deep-copy of the stream datastructure, so we can safely pass it around without fear that it will be mutated by another reference holder.

To use the capabilities of backtracking, we define the disjunction combinator (the only one which actually requires making run-time decisions) in terms of lookahead. As we have discussed, it is important to support different strategies for lookahead, since backtracking is not always necessary. Therefore, we use a Coq typeclass to encode a strategy for selecting between two parsers `m1` and `m2`:

```
  Class Branch {var st br out} (m1 m2 : Machine var st br out) :=
  { lookahead_machine : Machine var st br br }.
```

We can define an efficient instance of this class when `m1` requires only a single token of lookahead. We have another type class `First` that records the "first set" for such parsers. First sets, which are commonly used in LL(1) parsing, record the tokens that may appear first in a valid input.

```
  Global Instance first_branch {var st tok out}
      (m1 : Machine var st (SetCompl tok) out)
      (m2 : Machine var st (SetCompl tok) out)
      `{EqDecision tok}
      {F : First tok m1} : Branch m1 m2 | 0 :=
  {| lookahead_machine := Return (is_set (first_set (m:=m1))) |}.
```

Here, the lookahead machine can simply check whether the first token of the input stream is in the first set of `m1`. Since first sets can be statically determined, this enables our partial evaluation to make use of it.

Coq's type class system allows us to assign "weights" to each instance. These weights determine which instance is used when multiple instances are available. Here, we assign the `first_branch` instance the lowest possible weight, 0, so that it will be chosen whenever possible.

The less work a lookahead machine has to do, the better. However, in the worst case, we can *always* find a lookahead machine for an arbitrary pair of parsers, simply by fully running `m1`, and picking it if it succeeds, and backtracking otherwise. This gives rise to the following instance:

```
    Instance slow_branch {var st br out} `{BInfo tok br}
          (m1 m2 : Machine var st br out) : Branch m1 m2 | 100 :=
    {| lookahead_machine :=
        bind_inline
          (machine_to_opt m1)
          (fun o ⇒ return_ (match o with | None ⇒ false_b | _ ⇒ true_b end))
    |}.
```

Here `machine_to_opt` is a function which simply replaces every failure instance of a machine with `return_ None`, and wraps every success with `Some`.

We see that this machine accurately predicts whether a branch should be taken, but at the cost of potentially catastrophic backtracking! This is exactly what we want to avoid in parser combinators in general, so we add the weight of 100 to this instance, so that it is chosen only as a last resort.

In this way we leverage Coq's type class mechanism to perform search of `Branch` instances at each choice point, encouraging the choice of faster instances when possible, but without failing if a fast algorithm cannot be found for a particular occurrence of `m1 <|> m2`. Note that the `Branch` instance is chosen indepdently *every time the parser uses* `<|>`, not once for the entire parser Thus, we achieve maximally fine-grained choice of parsing strategies.

As a simple example of such a situation, we demonstrate the classic example of the "optional else" problem: in many languages, an "`if ... then ... else ...`" construct may optionally omit the else clause, with the implicit behavior being simply a "no-op", or fallthrough. Such a language may be easily written in our system:

```
Inductive stmt_tok :=
| If
| Then
| Else
| Seq
| Exp
| Stmt.

...

Definition parse_prog {rv} : Machine rv () (SetCompl stmt_tok) () := fun _ ⇒
  Fix
    (fun rec ⇒
       ^Stmt <|>
        (^If @> ^Exp @> ^Then @> Var rec @> ^Seq) <|>
        (^If @> ^Exp @> ^Then @> Var rec @> ^Else @> Var rec @> ^Seq)).
```

We do not recover the semantic values (ASTs) of programs here for simplicity, simply producing `unit`.

We then correctly synthesize a parser, which correctly uses first-sets for the atomic case, but performs the backtracking search for the two forms of the if-then statements.

```
Definition test_input1 := [If; Exp; Then; Stmt; Else; Stmt; Seq].
Definition test_input2 := [If; Exp; Then; Stmt; Seq].
Definition test_input3 := [If; Exp; Then; Stmt; Stmt].

Eval compute in eval_M prog () test_input1. (* = (inr (), [], ()) *)
Eval compute in eval_M prog () test_input2. (* = (inr (), [], ()) *)
Eval compute in eval_M prog () test_input3.
(* = (inl "parse_exact: Unexpected Token!", [Stmt], ()) *)
```

iv.F Evaluation and Benchmarks

We evaluated our development in two ways. First, we implemented a collection of combinators and parsers for a variety of languages, like the s-expression parser described above. We found the library to be similar in complexity to the TAAP library and other parser combinator libraries. At the same time, our approach enables development of parsers for a larger class of languages than the TAAP library, as described in Section iv.E.

Second, we evaluated the library's performance by re-implementing two of the key benchmarks from the TAAP paper (s-expressions and JSON). We benchmarked our parsers and theirs using the data sets from the TAAP repositories. For our parsers, we evaluated two different OCaml backends—the default 4.10.0 OCaml compiler and the "flambda" 4.10.0 OCaml compiler, which implements additional optimizations. For the TAAP parsers, we used the provided docker image, which contains version 4.07.1 of the BER MetaOCaml compiler. Unfortunately, no more recent version of the BER MetaOCaml compiler is available, and our Coq and OCaml code is not compatible with older versions of OCaml, so we were unable to compare on exactly the same compiler.

The results of these benchmarks are shown in Figure 2. We found that our parsers substantially outperformed the TAAP parsers, running 29% faster in the s-expression benchmark and 9% faster in the json benchmark, when compiled with the flambda compiler. When compiled with the default OCaml compiler, the sexp benchmark runs slightly slower (9%) than the TAAP parsers, and the JSON benchmark runs substantially slower (147%).

This performance difference highlights a key advantage of our approach: support for multiple backends. Because we use Coq for metaprogramming and generate standard OCaml code, we are not restricted to the use of the BER research compiler. Indeed we plan to explore other backends in future work, including other programming languages and FPGA implementations.

To understand the performance differences, we have inspected the OCaml code that we extract from Coq and the intermediate OCaml code generated by the BER compiler for the TAAP parsers. We can see that our extracted OCaml could be improved in some cases. As a simple example, Coq string literals are currently extracted as lists of characters, which is an extremely inefficient representation. A fix for this is expected in the next version of Coq. We can also see some intermediate

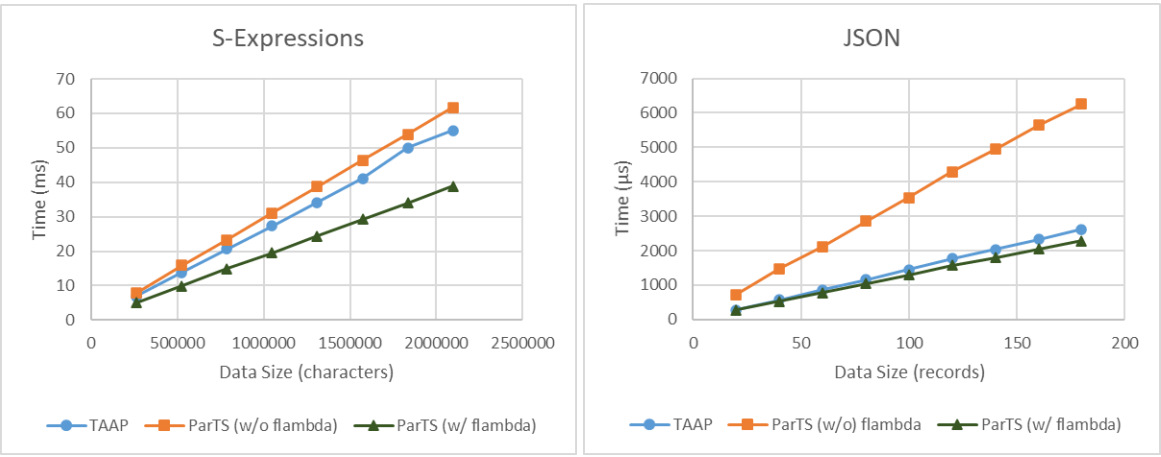

Figure 2: Performance Benchmarks

computations that could be simplified or eliminated given more time to work on our partial evaluation implementation. This is perhaps unsurprising, as the MetaOCaml research project is two decades old, and we have implemented our Coq partial evaluator techniques over the last seven months.

The large performance hit in the JSON example with the unoptimized compiler appears to come from overly-aggressive fusion optimizations in our evaluator resulting in the lexer code being massively duplicated within the parser. We suspect the flambda compiler is able to eliminate this duplication with common subexpression elimination or other optimizations. Given more time, we could develop smarter fusion techniques that weigh potential code bloat against other considerations.

While we are already outperforming TAAP, our inspection of the generated code has suggested several avenues for additional performance gains. These include several pieces of "low-hanging fruit" that could be reached quickly now that we have an end-to-end, functioning library. We highlight a few examples in Section viii.A.

v Important Findings and Conclusions

This report has described the design and implementation of the ParTS parser combinator library. This library presents a user-friendly interface for implementing parsers while offering substantial improvements compared with existing state-of-the-art with both better performance and support for many more classes of languages.

- We have demonstrated that it is possible to build a Coq parser combinator library with an extremely performant backend, using the advanced features of Coq to achieve the metaprogramming and partial evaluation techniques offered in research tools like BER MetaOCaml.
- We have narrowed the gap between secure parser construction tools by showing that declarative specifications of separate lexing, parsing and validation phases can be fused to single-pass implementations. Fusion optimizations enable these parsers to succeed or fail as soon as the relevant data is available, just as a hand-written parser would, rather than performing complete lexing and parsing passes before the data can be used.
- We have shown that these techniques can be extended to an extremely broad class of languages, including those that require arbitrary lookahead and state, without sacrificing performance. Our technique uses Coq type classes to select a new lookahead strategy each time the parser must make a choice, enabling maximally fine-grained parsing algorithm selection.

vi Significant Hardware Development

No hardware development was performed on the ParTS project.

vii Special Comments

The ParTS Team has no additional special comments.

viii Implications for Further Research

Our encouraging performance numbers, along with the ability to use the combinator library to effectively write high-level declarative parsers, suggests the possibility of exciting future work.

viii.A Performance

There are many things which could additionally improve the performance of our extracted parsers. We can leverage the fact that we are working with a relatively restricted domain of applications (producing values from a single input stream, in an incremental manner) to apply many optimizations which are not beneficial in general purpose computing.

One example is our use of stack space for recursion: in general, it is possible to generate extremely efficient in-place imperative code which can even out-perform naive hand-written code, in a manner that is a natural extension of our work [20].

Another example is our use of a functional monadic state: we could easily replace this with an imperative, in-place implementation at extraction. In addition, because we expect the state to have a very limited set of potential instantiations (integer counters, stacks, queues and maps), there are many opportunities for generating efficient code for these limited cases, in an *ad hoc* manner.

Finally, while we do perform fusion for composed parsers, there remains work to evaluate the trade offs of fusion and code bloat, and to achieve more complete elimination of intermediate data structures.

viii.B Proofs

While we work within the Coq proof and specification language, we have not yet explored formal verification of our library. There are several appealing properties we would like to prove.

The first is the correctness of the efficient CPS evaluator on abstract streams with respect to a more straightforward evaluator with lists as input. This property is both natural, and creates a high degree of confidence with respect to additional tweaks and experiments with propagation and transformation into continuation-passing style.

The next is the correctness of the parser composition. Even stating this property is surprisingly challenging. For example, a naive statement might be:

> If the parser "`compose lex parse`" accepts a given input, with result `a`, then the result of taking the output of `star lex` and feeding it into `parse` also succeeds, with result `a`.

However, this version of the property does not hold. One advantage of fusion optimizations it he ability to end parsing early if the later fused stages do not need all the input. This means that, for example, if the lexer would fail on input in the stream that occurs after the parser has finished, a fused version would succeed.

It would be an interesting research project to determine the correct property for fused parser. One idea would be to weaken the property as follows:

> If the parser "`compose lex parse`" accepts a given input, with result `a` **and the parser `star lex` succeeds on the input**, then the result of taking the output of `star lex` and feeding it into `parse` also succeeds, with result `a`.

This would require additionally that the lexer does not fail on the entire input. We believe that this is still a reasonable safety property, since the failure of the lexer in this case will only occur on input that is not even examined, and can therefore not adversely influence the final result. Examining this question in more detail is left for future work.

A third goal is to prove the correctness of the parser combinators with respect to a more abstract definition of languages, where a (non-deterministic) language is simply defined as a subset $\mathcal{L} \subseteq \text{streams} \times A$, where A is the type of outputs and streams is the type of all possible streams.

Then the correctness of a given parser p with respect to a language $\mathcal{L}$ would state that, on input str, $p(str)$ succeeds and returns a iff $(str, a) \in \mathcal{L}$.

viii.C Backends

While targeting OCaml is both a natural fit for Coq and a good trade-off of expressiveness and speed, it is quite tempting to want to target different runtimes for our tool. We consider two possibilities: a C backend and an FPGA backend.

Targeting C would require several steps:

- Having a notion of C computation in Coq. This can either be handled via existing C semantics formalizations, or simply treated as an opaque black box of semantic actions, with simple inclusions of base types like `int`, `char`, etc.
- Using a C AST for control-flow decisions in the `eval_M_stream` partial evaluator, rather than Coq builtins. We may only need function calls and if-then-else statements.
- Building a pretty-printer for the control-flow constructs and the abstract stream data-type. This is straightforward.

In general, state needs to be restricted to the primitive data-types as well. This approach could build on similar work in the Bedrock and Fiat Coq libraries [21].

Targeting hardware uses similar ideas, but with the following caveats:

- Hardware, particularly FPGAs, tends to shine when given massively parallel, independent tasks in lockstep. More investigation would be needed to understand the trade-offs here, but an obvious opportunity for this would be in our notion of composition, where the tokenizer/lexer might work independently of the parser, possibly filling a queue of tokens.
- In general, it may be difficult or costly to branch. We could explore modifications of the library where we remove branch typeclass instances that are inefficient in hardware to provide static checking that an FPGA implementation is suitable.
- Lookahead and backtracking are particularly subtle—it may make sense to limit parsers to a fixed lookahead depth.

- State must be restricted to a constant amount of memory.

Here we may use the Kami [22] framework as our model for hardware.

ix Other Items of Interest

The ParTS team does not have any additional items of interest to bring to the attention of the government.

x References

References

[1] NIST, "CVE-2018-0101." National Vulnerability Database. https://nvd.nist.gov/vuln/detail/CVE-2018-0101.

[2] Cisco, "Cisco adaptive security appliance remote code execution and denial of service vulnerability." https://tools.cisco.com/security/center/content/CiscoSecurityAdvisory/cisco-sa-20180129-asa1, 2018.

[3] NIST, "CVE-2015-3864." National Vulnerability Database. https://nvd.nist.gov/vuln/detail/CVE-2015-3864.

[4] Zimperium, "Experts found a unicorn in the heart of android." https://blog.zimperium.com/experts-found-a-unicorn-in-the-heart-of-android, 2015.

[5] NIST, "CVE-2011-3402." National Vulnerability Database. https://nvd.nist.gov/vuln/detail/CVE-2011-3402.

[6] M. Mimoso, "Of truetype font vulnerabilities and the windows kernel." https://threatpost.com/of-truetype-font-vulnerabilities-and-the-windows-kernel/101263, 2013.

[7] W. H. Burge, *Recursive Programming Techniques*. Addison-Wesley, 1975.

[8] G. Hutton and E. Meijer, "Monadic parsing in haskell," *Journal of Functional Programming*, vol. 8, no. 4, pp. 437–444, 1998.

[9] D. Leijen and E. Meijer, "Parsec: Direct style monadic parser combinators for the real world," Tech. Rep. UU-CS-2001-27, July 2001.

[10] N. R. Krishnaswami and J. Yallop, "A typed, algebraic approach to parsing," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, (New York, NY, USA), p. 379âĂŞ393, Association for Computing Machinery, 2019.

[11] O. Kiselyov, "The design and implementation of berÂămetaocaml," in *Functional and Logic Programming* (M. Codish and E. Sumii, eds.), pp. 86–102, Springer International Publishing, 2014.

[12] Y. Futamura, "Partial evaluation of computation process–an approach to a compiler-compiler," *Higher-Order and Symbolic Computation*, vol. 12, pp. 381–391, 1999.

[13] P. Wadler, "Deforestation: transforming programs to eliminate trees," *Theoretical Computer Science*, vol. 73, no. 2, pp. 231 – 248, 1990.

[14] A. Chlipala, "The bedrock structured programming system: combining generative metaprogramming and hoare logic in an extensible program verifier," in *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*, pp. 391–402, ACM, 2013.

[15] A. Chlipala, "Parametric higher-order abstract syntax for mechanized semantics," in *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, ICFP âĂŹ08, (New York, NY, USA), p. 143âĂŞ156, Association for Computing Machinery, 2008.

[16] O. Kiselyov, "Why a program in cps specializes better." Staging, Program Generation, Meta-Programming. http://okmij.org/ftp/meta-programming/Generative.html#bti.

[17] A. Bondorf, "Improving binding times without explicit cps- conversion.," in *ACM SIGPLAN Conference on LISP and Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*, ACM, 1992.

[18] A. Bondorf, "Improving binding times without explicit cps-conversion," *SIGPLAN Lisp Pointers*, vol. V, p. 1âĂŞ10, Jan. 1992.

[19] O. Danvy, K. Malmkjær, and J. Palsberg, "Eta-expansion does the trick," *ACM Trans. Program. Lang. Syst.*, vol. 18, p. 730âĂŞ751, Nov. 1996.

[20] O. Kiselyov, A. Biboudis, N. Palladinos, and Y. Smaragdakis, "Stream fusion, to completeness," in *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, (New York, NY, USA), p. 285âĂŞ299, Association for Computing Machinery, 2017.

[21] A. Erbsen, J. Philipoom, J. Gross, R. Sloan, and A. Chlipala, "Simple high-level code for cryptographic arithmetic - with proofs, without compromises," in *2019 IEEE Symposium on Security and Privacy (SP)*, pp. 1202–1219, 2019.

[22] J. Choi, M. Vijayaraghavan, B. Sherman, A. Chlipala, and Arvind, "Kami: A platform for high-level parametric hardware specification and its modular verification," *Proc. ACM Program. Lang.*, vol. 1, Aug. 2017.