The pitfalls of protocol design

Attempting to write a formally verified PDF parser

Andreas Bogk
Principal Security Architect
HERE
Berlin, Germany
andreas.bogk@here.com

Marco Schöpl
Institut für Informatik
Humboldt-Universität
Berlin, Germany
schoepl@informatik.hu-berlin.de

Abstract

Parsers for complex data formats generally present a big attack surface for input-driven exploitation. In practice, this has been especially true for implementations of the PDF data format, as witnessed by dozens of known vulnerabilities exploited in many real world attacks, with the Acrobat Reader implementation being the main target. In this report, we describe our attempts to use Coq, a theorem prover based on a functional programming language making use of dependent types and the Curry-Howard isomorphism, to implement a formally verified PDF parser. We ended up implementing a subset of the PDF format and proving termination of the combinator-based parser. Noteworthy results include a dependent type representing a list of strictly monotonically decreasing length of remaining symbols to parse, which allowed us to show termination of parser combinators. Also, difficulties showing termination of parsing some features of the PDF format readily translated into denial of service attacks against existing PDF parsers—we came up with a single PDF file that made all the existing PDF implementations we could test enter an endless loop.

1. Introduction

The work we are presenting here was inspired by a combination of two research areas we are interested in: usage of dependently typed programming languages for formal verification of software, and aspects of language theory on IT security, an area by now known as LangSec.

Since the PDF file format is not only widely used, but also quite often used as an attack vector in IT security breaches, we focused on writing a PDF file format parser. As the platform for implementation, we chose Coq. Even though Coq is primarily a proof assistant, and a research platform for formal proofs based on the calculus of constructions and the Curry-Howard isomorphism, it doubles as a functional programming language with support for dependent types. It has been shown that writing real world software by using code extraction from Coq is entirely feasible, and interesting correctness results for such software have been shown.[4]

In this paper, we will first look into the general problem of writing parser combinators in Coq, and then look at the practical problems using them for parsing PDF files that we encountered.

2. Parser combinators in Coq

Our implementation of a PDF parser uses the well-known pattern of parser combinators[2] often used in functional languages. The implementation of parser combinators in Coq is mostly a straightforward exercise. However, certain constructs involving recursion lead to problems. In this section, we will look at our initial attempt that fails at recursion, and then examine the refinement that led to a successful implementation.

2.1. Initial attempt

Initially, we defined a parser for a type `T` to be a function taking a `list` of tokens (in our case, `ascii` characters), and returning either some `T` plus a list of remaining unparsed tokens, or a parse error with some suitable error message:

```
Require Import Ascii.
Require Import Coq.Lists.List.
```

```
Definition parser' (T : Type) :=
list ascii ->
  optionE (T * list ascii).
```

Here, `optionE` is the familiar option type (or `Maybe` monad to Haskell programmers) that supports an additional error message for the `None` case:

```
Inductive optionE (X:Type) : Type :=
| SomeE : X -> optionE X
| NoneE : string -> optionE X.
```

This definition already allows us to implement a number of interesting combinators, such as one that matches on a single arbitrary character:

```
Definition parse_one_char'
  : parser' ascii :=
fun xs =>
  match xs with
  | []     =>
    NoneE "end of token stream"
  | (c::t) => SomeE c
  end.
```

or one that sequences two parsers:

```
Definition sequential' {A B : Set}
  (a : parser' A) (b : parser' B)
  : parser' (A*B) :=
fun xs =>
  match a xs with
  | SomeE (val_a, xs') =>
    match b xs' with
    | SomeE (val_b, xs'') =>
      SomeE ((val_a, val_b), xs'')
    | NoneE err => NoneE err
    end
  | NoneE err => NoneE err
  end.
```

However, the following implementation of the Kleene star operator, which repeatedly applies the same parser for some type `T` to generate a list of `T`, although looking like the natural solution to the problem, fails:

```
(* Broken: the fixpoint recursion
   is not allowed *)
Definition many' {T : Set}
  (p : parser' T) : parser' (list T) :=
fun xs => many_helper' T p [] xs.
Fixpoint many_helper' (T : Set)
  (p : parser' T) (acc : list T)
  (xs : list ascii)
  : optionE (list T * list ascii) :=
match p xs with
| NoneE err      => NoneE err
| SomeE (t, xs') =>
  match many_helper _ p (t::acc) xs'
  with
  | NoneE _            =>
    SomeE (rev (t::acc), xs')
  | SomeE (acc', xs'') =>
    SomeE (acc', xs'')
  end
end.
```

The reason for this is that Coq doesn't allow unbounded recursion, it wants to see proof that the recursion terminates. This is because infinite recursion gives rise to circular and thus unsound proofs under the Curry-Howard isomorphism in Coq. The language specification explicitly requires structural induction on one of the parameters in order for the `Fixpoint` definition to succeed, or alternatively a measure on one of the arguments that provides an equivalent of structural induction.[5]

This restriction of arbitrary recursion might seem like a burden at a first glance. However, on further examination it becomes clear that guaranteed termination is actually an interesting and useful property of a parser algorithm. We're getting a form of correctness validation out of this behaviour of Coq.

In a lot of cases, when writing a Fixpoint recursive function definition, Coq can identify a function argument that is structurally decreasing. By the nature of inductive construction of objects, it follows that the function eventually terminates. For instance, in a function like:

```
Fixpoint plus (n m:nat) : nat :=
match n with
| O => m
| S p => S (plus p m)
end.
```

Coq will detect that the argument `n` is structurally decreasing and the function plus is guaranteed to terminate. It will complain that it can't find a structurally decreasing argument in our above definition of `many'`, though. There is a syntax to specify which of the arguments is structurally decreasing. It isn't of much help here, as indeed the above definitions allow for infinite recursion. Consider a construction that consists of a parser that matches the empty string ϵ:

```
Definition match_epsilon
  : parser' unit :=
fun xs => SomeE (tt, xs).
```

and applying the `many'` Kleene star operator to it:

```
Definition match_many_epsilon
  : parser' (list unit) :=
many' match_epsilon.
```

It should be obvious that when calling `match_many_epsilon` with any argument, it keeps matching the empty string indefinitely, without ever making any parsing progress. So in fact, Coq is right about complaining about `many'`, we have shown a case where it won't terminate.

2.2. Solution: dependent types

The usual workaround given in the literature[1] is so-called step-indexed recursion. The key idea here is that an extra argument, the step index, is passed to the recursive function. On the initial call, the maximum number of allowed recursion steps is passed to the function. On every recursion step, the index is reduced by one. When it reaches zero, the computation is aborted. The step index takes the role of the structurally decreasing argument that Coq wants to see.

Step-indexed recursion seemed like an easy enough way out. It even matches practical constraints of computation, like the finite amount of RAM found in computers, that put a bound to the maximally possible recursion steps anyways. However, it seemed a bit arbitrary, and it essentially also shadows real problems of non-termination as the above `match_any_epsilon` case. So we decided to see whether we can come up with a recursion approach that retains the analysis capabilities we wanted.

Our approach is based on the intuition of "making progress" during parsing, as used in the argument above. It turns out that the key to actually come up with a parser combinator construction and show termination lies within formalization of this idea. It is easy enough to reason that as long as every parser step keeps consuming input tokens, we will finally reach the end of our token list, and this parsing will terminate.

Revisiting our initial definition of the `parser'` type:

```
Definition parser' (T : Type) :=
list ascii
  -> optionE (T * list ascii).
```

we notice that the type doesn't express any relationship between the list of tokens used as the input to the function and the list of remaining tokens returned from the function. However, we do know that there is one: the returned list of tokens is supposed to be the non-consumed rest of the token list that was used as the input to the parser.

By making use of dependent types, we can actually come up with a type that formally captures this idea. Dependent types, generally speaking, are types depending on values. Of particular interest here are dependent pair types, also known as sigma types, which can be understood as existential quantifiers. Coq comes with a syntax for this that closely resembles mathematical notation: we can use `{ x : T | P x }` to express the type of all `T` for which the proposition `P` holds. For instance, given a type `nat` for natural numbers, and a proposition `prime : nat -> Prop` that holds when its argument is a prime number, we can express the type of all primes as `{ x : nat | prime x}`. Actual instances of this type can be understood as the product type of a value, and a proof that the proposition holds for any possible value at this moment. This proof is a first class object, and can be passed around and manipulated, e.g. to construct other instances of the same type.

Using dependent types, we can construct a better type for `parser`. The first step is a definition of a predicate `sublist l l'`, which expresses that a list `l` is the list remaining of another list `l'` after removing an arbitrary non-zero number of elements from the beginning of `l'`. We use an inductive definition of this predicate instead of a computational, to make proofs easier later:

```
Inductive sublist
  : list A -> list A -> Prop :=
| sl_tail : forall (c : A) l,
    sublist l (c::l)
| sl_cons : forall (c : A) l' l,
    sublist l' l -> sublist l' (c::l).
```

Given this predicate, we can now properly implement the `parser` type:

```
Definition parser (T : Type) :=
forall l : list ascii, optionE (T *
  {l' : list ascii | sublist l' l}).
```

This now formally captures our intuitive notion that every parser step should make progress.

Implementation of parser combinators becomes a bit more intricate now. We have to make sure to properly manipulate proof objects for the dependent pair. We do that by creating dependent pairs and destructuring them in pattern matching using the `exist` constructor. For instance, the definition of `sequential` now becomes:

```
Definition sequential {A B : Set}
  (a : parser A) (b : parser B)
  : parser (A*B) :=
fun xs =>
  match a xs with
  | SomeE (val_a, exist xs' H) =>
   match b xs' with
   | SomeE (val_b, exist xs'' H') =>
     SomeE ((val_a, val_b),
             exist _ xs''
               (sublist_trans H' H))
   | NoneE err => NoneE err
   end
 | NoneE err => NoneE err
end.
```

What's going on here is that running the first parser `a` on the input returns a list of remaining tokens `xs'` and a proof object `H` that states that `sublist xs' xs` holds. Calling the parser `b` on `xs'` in turn produces a list of finally remaining tokens `xs''`, plus a proof object `H'` stating that `sublist xs'' xs'` holds. Using a theorem stating that `sublist` is transitive, easily shown using induction:

```
Theorem sublist_trans
  : forall l l' l'',
  sublist l l' -> sublist l' l''
  -> sublist l l''.
Proof.
  intros l l' l'' H0 H;
    generalize dependent l.
  induction H; intros.
    constructor; assumption.
    constructor;
      apply (IHsublist _ H0).
Qed.
```

we can construct a proof object showing that `sublist xs'' xs` holds, and thus create the dependent pair for the return type by calling the `exist` constructor.

Using the refined type for `parser` now allows us to express recursion by passing an argument to the `Function` definition that provides Coq with the necessary information to show that the recursion is well-founded and thus will terminate. In our case, it is a measure on the length of the list of tokens passed as an input that we can now show to be structurally decreasing:

```
Definition many {T:Set} (p : parser T)
  : parser (list T) :=
fun xs => many_helper T p [] xs.

Function many_helper (T:Set)
  (p : parser' T) (acc : list T)
  (xs : list ascii)
  {measure List.length xs }
  : optionE (list T *
  {l'' : list ascii | sublist l'' xs})
:= ...
```

The actual implementation and proof mechanics is omitted here for brevity, the interested reader can find it online. [1]. The intuition is that since every parser step consumes tokens, the length of the list will decrease with every step, and so the recursion is well-founded.

An interesting result follows directly from the definition of the `parser` type: we can't write a parser for the empty string that returns something else than a parse error:

```
Lemma parser_nil_none : forall t
  (p : parser t), exists err,
    p [] = NoneE err.
Proof.
  intros. remember (p []) as H.
  destruct H.
    inversion p0. inversion H.
        inversion H0.
    exists s. reflexivity.
Qed.
```

In other words, we can't have ϵ in our parser universe!

Another interesting result is that the `many` operator matches one or more successful parses, not zero or more as common in other implementations. This follows directly from our definition of a parser to always consume tokens.

In practice, not having the ability to match on empty strings or zero repetitions doesn't seem to be a problem. In all instances, we could find alternative constructions that would suffice. For instance, when we need to match on some `X` preceded by zero or more whitespace characters, we would use a construction that matches on either an `X`, or one or more whitespace characters followed by an `X`.

The most important result we achieved in this project, however, is showing that we can use dependent types to properly prove termination of parsers built using our parser combinators, without resorting to tricks like step-indexed recursion.

1. http://github.com/andreas23/pdfparser

3. Parsing PDF

We attempted to use our parser combinator library to implement a parser for the PDF file format. We succeeded to come up with a parser for a subset of PDF, enough to understand the object structure of the PDF file and extract byte streams for objects out of it.

Once our parser combinators were in place, using them to implement most parts of the PDF parser was easy and straightforward enough, e.g. parsing numbers, strings, object references and other primitive parts of a PDF. However, difficulties arose once we tried to put everything in place to parse complete PDF files. The reason again was that there were some recursive functions that we had a hard time showing termination for.

More precisely, the PDF format consists, simplifying, of a byte stream consisting of a header, a trailer, a number of objects between them, and a cross-reference table. Each object can be referred to by an ID number inside the PDF, and the cross-reference table maps ID numbers to byte offsets inside the PDF file. In order to read a specific object, the parser needs to look up its byte offset in the cross-reference table.

The format for the objects themselves can vary, one possible version involves a notation for a "stream". This is a sequence of arbitrary bytes, preceded by a length specification. The termination problem arises from the fact that this length may not only be specified using a direct integer value, but also by indirectly referencing another object that will contain the length value. In such cases, in order to parse an object, we first need to parse another object, giving rise to a recursive function call.

This is precisely the point where we had difficulties showing termination. Often, when a proof for something in Coq fails to go through, it helps to try to come up with a counterexample for the theorem in question. So contemplating the problem at hand, we could indeed come up with constructions of PDF files where the naive recursive parsing of objects would lead to infinite recursion. The two constructions we tried were the case where the length of an object was an indirect reference to that object itself, and the case where two objects would mutually reference each other indirectly for their length value.

```
6 0 obj
  << /Length 6 0 R >>
stream
  foobar
endstream
endobj

7 0 obj
  << /Length 8 0 R >>
stream
  foobar
endstream
endobj

8 0 obj
  << /Length 7 0 R >>
stream
  foobar
endstream
endobj
```

In this PDF snippet, object 6 references itself for the length value of its stream, object 7 and 8 mutually refer to each other for their respective stream lengths. A naive recursive parser will attempt to parse both objects in alternation, and hang in an endless loop.

A related problem exists with the support for updates of the cross-reference table. In order to support incremental updates of PDF files, cross reference tables can point to older versions of a cross reference table inside the same file, so additional objects can be added to the PDF file without having to rewrite the entire file. The pointer to the older cross reference table is again a byte offset into the file, and again we can construct two cross reference tables pointing to each other:

```
% at offset 2342:
xref
0 7
0000000000 65535 f
0000000596 00000 n
0000000686 00000 n
% ... (other entries omitted)
trailer
  <<  /Size 7  /Root 1 0 R
      /Prev 4223  >>
startxref
2342
%%EOF
% ...
% at offset 4223:
xref
% ... (table contents omitted)
trailer
  <<  /Size 7  /Root 1 0 R
      /Prev 2342  >>
startxref
4223
%%EOF
```

Interestingly enough, a PDF file containing all these constructions would hang every single PDF reader we tried it on in an endless loop[2]. Despite this, these constructions are allowed by the PDF standard, see [3], Chapters 7.3.8.2, 7.3.10 and 7.5.8.

We were able to come up with an implementation that would correctly determine that parsing the above constructs would be impossible. In order to do so, we implemented a mechanism that would remember the file offsets that were already being used to attempt parsing an object or xref table. This would return an error if a parsing attempt was made at at offset where an unfinished attempt has been made before. The correctness proof for this construction was non-trivial, and spans 500 lines.

4. Conclusion

We have shown how to implement parser combinators with a formal proof of termination that does not rely on step-indexed recursion, using dependent types. We have also shown how making use of formal termination proofs helps to find weaknesses in protocol specifications, which would be hidden when relying on step-indexed recursion. More precisely, it allowed us to come up with constructions of PDF files which are not in violation of the PDF standard, but which are nonetheless impossible to parse, because they contain cyclic structures.

Since this was only an academic exercise, we have not finished the implementation of the full PDF standard. It became obvious that there are more constructions in this format that give rise to cyclic structures, such as the `offset` and `reference` primitives, links from older generations of objects to newer generations, or incorrect backlinks from pages to their parents. Also, we haven't even looked into overlapping structures: due to the arbitrary file offsets, objects can be parts of each other, overlap with cross reference tables or with each other. All these give rise to similiar problems as the one described here.

It also became clear that the attempt to write a formally verified PDF parser is an extremely hard exercise. Even our very limited form of correctness, showing termination, quickly becomes tedious and frustrating. The reason for that lies entirely in the PDF specification itself, which defines a language that is not context free, and which contains constructs that are legal according to specification, but which have no meaningful representation which can be parsed.

2. Some of them, such as Acrobat Reader, have since been fixed

It is highly advisable, for all future definitions of protocols, file formats and other data exchange languages, to make sure that the format specification is complete, unambiguous and doesn't allow unparseable constructions. Furthermore, it should be shown that the language is context-free. Otherwise, any kind of formal analysis will run into problems, which are mirrored by reliability problems in real world implementations of these protocols.

References

[1] Benjamin Pierce et al. *Software Foundations*. July 2013. URL: http://www.cis.upenn.edu/~bcpierce/sf/.

[2] Graham Hutton. "Higher-order functions for parsing". In: *Journal of Functional Programming* 2.3 (1992), 323–343. URL: http://eprints.nottingham.ac.uk/221/1/parsing.pdf.

[3] ISO. *Document management—Portable document format—Part 1: PDF 1.7*. ISO 32000–1:2008. Geneva, Switzerland: International Organization for Standardization, 2008. URL: http://wwwimages.adobe.com/www.adobe.com/content/dam/Adobe/en/devnet/pdf/pdfs/PDF32000_2008.pdf.

[4] Xavier Leroy. "Formal verification of a realistic compiler". In: *Communications of the ACM* 52.7 (2009), pp. 107–115. URL: http://gallium.inria.fr/~xleroy/publi/compcert-CACM.pdf.

[5] The Coq Development Team. *The Coq Proof Assistant Reference Manual*. 2012. URL: http://coq.inria.fr/refman/.