Research Report: An *Optim*(*l*) Approach to Parsing Random-Access Formats

Mark Tullsen, Sam Cowger, Mike Dodds
Galois, Inc.
{tullsen,sam,miked}@galois.com

Peter Wyatt
PDF Association
peter.wyatt@pdfa.org

***Abstract*—We introduce a Domain Specific Language (DSL) that allows for the declarative specification of random access formats (formats that include offsets or require out of order parsing, e.g., zip, ICC, and PDF). Our DSL is composed by layering three distinct computational languages: first, we have the *base layer* of primitive parsers (these could use various "off the shelf" parsing technology). Second, upon this base we layer our *XRP* (*eXplicit Region Parser*) DSL, a pure functional language for declaratively specifying random-access formats. *XRP*, although similar to parser combinators, gives greater control and expressiveness by making parsing regions explicit. Third, we embed *XRP* into *Optim*(*l*), a novel DSL that describes Directed Acyclic Graphs (DAGs) of computations. A node represents a computation in *XRP*, an edge represents a dependency between the computations. From a declarative *Optim*(*l*) program we can generate an imperative module that provides an API for on-demand and incremental parsing of the random access format.**

I. Introduction

A. PDF Applications: Not Your Traditional Parser

In the DARPA SafeDocs project, much effort—by Galois and other teams—was spent researching and developing methods for writing safe, secure PDF parsers. Parsing PDFs (Portable Document Format) [9] is challenging on many fronts: the language has been evolving over many decades; the specification is informal, even ambiguous in places; PDF includes dozens of embedded formats; PDF has elements which are binary-formats, elements which are textual grammar-based languages; PDF has features that support incremental parsing. (We have previously addressed some of these difficulties in [16].)

It is understandable that one might view the problem of writing safe, secure PDF parsers as diagrammed in Fig. 1 and ask, **What kind of *parsing technology* is required to write a secure PDF parser?** With our "parsing tool" *hammers* in hand, PDF applications certainly look like *nails*. But, is the lack of better *parsing technology* the **primary** impediment to PDF security? We think not. In fact, there are many factors. Two that stand out are poor software engineering practices and the inadequacies of the PDF specification (changing, informal, ambiguous in places, etc.). But in this paper, we call attention to another major factor: the mismatch between format description tools and the actual needs of *PDF applications*. The problem needs to be viewed differently, see Fig. 2, and the question we *now* ask is, **What should a *multi-entry point* parser look like and how do we build one to be correct and performant?** It turns out that actual PDF tools are implemented primarily like Fig. 2 and much less like Fig. 1 : where the application code reads and parses the file on demand and incrementally. In other words—to use the terminology of Harel and Pneuli [4]—PDF tools are *Reactive systems*, not *Transformational systems*. "Shotgun parsers"[1] in the LangSec community have a bad reputation, but when the problem looks like Fig. 2, a correct, well-structured solution is not at all obvious; it turns out the many "shotgun parsers" in the wild are a reflection of an unsolved software engineering problem.

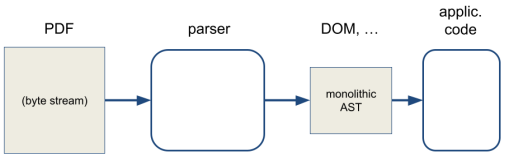

Fig. 1. PDF Application, viewed as "Transformational" system

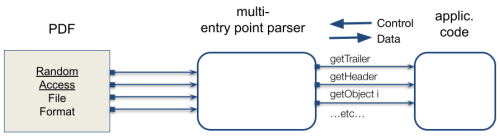

Fig. 2. PDF Application, viewed as "Reactive" system

Fig. 1 works for most parsing applications (e.g., the parsing of programming languages, most binary formats, etc.); so, why do all the PDF applications look like Fig. 2? It is because PDF is a random access format, which *by design* allows for both partial and incremental parsing.

[1] A shotgun parser is a parser in which the parsing code is scattered across the application code.

B. Random Access Formats: Opportunities and Problems

A *random access format* is a format that involves parsing which need not be sequential over the input, e.g., the format may include offsets or locations which are used by the parser to "jump" to new locations to parse. Examples of random access formats are ICC, ELF, Zip, and PDF. Random access formats are often binary, but could be textual (e.g., PDF). We use the term *region* to refer to a sub-range of the input that can—or must—be parsed out of sequence. A region's start could be defined by an absolute address, a relative address, or be known statically. A region often has a known length (either statically or known at the time of region parsing). We can view a sequential format as the degenerate case of a single region format.

A simple example of a random access format is ICC [6], see the diagram in Fig. 3. Note the "Tag Table" where entries contain byte offsets to various "Tag element data" (TED) regions later in the file.

Partiality and Parallelism

The presence of regions in a random access format can allow for partial, incremental, or on-demand parsing of sub-regions of the format. This is usually the *point* of having sub-regions in the format. Regions can also provide the possibility of parallel parsing. For example, note, in Fig. 3, the multiple "Tag element data" regions. An application could either parse just one needed TED region, or parse all the TED regions in parallel.

Partiality: Parsing Projections and Demand-Functions

We've mentioned "partial parsing," but what precisely do we mean? We use the term *parsing projection* to signify a program which implements the *function* of "parsing" and then (on success) projecting a subset of the resulting value. E.g., `getCnt` will parse an ICC file then extract the length of the TEDs data from the fully parsed ICC value (the same as the "Tag count" field).

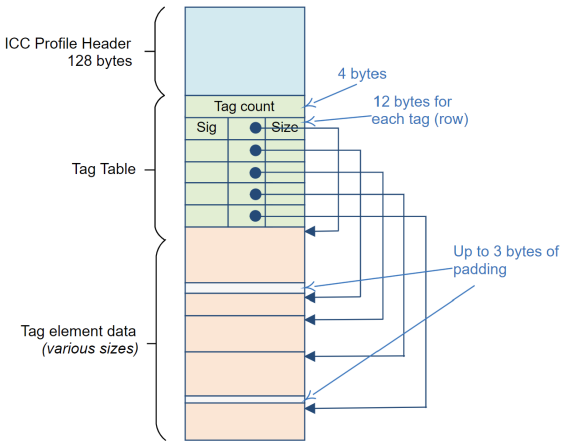

Fig. 3. ICC File Structure.

But the more useful concept is what we term a *demand-function*, which

- returns the same value as the equivalent *parsing projection*,
- may be defined on more inputs than the *parsing projection*, as it can be non-strict on the input; e.g., the `getCnt` demand-function could simply parse the "Tag count" in the ICC file but ignore the TEDS values (and is thus "robust" with respect to failures in the parsing of TED values).

When we have a set of demand-functions as part of a *multi-entry point* parser, the following would be ideal

- the minimal amount of computation is done;
- the computation is fully shared among the different demand-functions; and
- optimizations will preserve correctness with respect to the original *parsing projection*: i.e., if the *parsing projection* returns a value, then the *demand-function* must return with the same value.

So, *demand-functions* are both more efficient and "more defined". One might be a little nervous using parsers that ignore parts of their inputs, but in what follows we'll show how we can write *multi-entry point* parsers that provide both efficiency, and when desired, provide full validation. Note that this is how most PDF tools work and is the desired behavior in most cases: e.g., (1) we don't want the `pdftotext` tool to fail when there is a minor error in the color meta-data, (2) we want to display page one, even if there is an error in the independent definition of page two, and (3) we want to extract meta-data from the file even if the rest of the file is highly corrupted.

Regions and Cavities: Security Issues

When a random access format is fully parsed, the parser will generally "seek to" and parse various regions of the input; but what about the bytes that are not in any of these regions? We refer to such "negative space" regions as *cavities*, such byte ranges are not in any possibly parsed region, and thus should *not* be parsed. Cavities waste space, can indicate a format error, or even indicate malicious behavior. Regarding malicious behavior, a cavity might

- hold latent data for further 'exploits' such as a PDF shadow attack, see [13], or
- be the means for creating a 'polyglot', a document that is a member of two formats considered to be disjoint.

Another problem with *regions* is that when regions partially overlap, the proper behavior is unclear[2]: Are overlaps an error? If so, is a parser required to detect and reject them[3]? When lack of clarity exists as to what is *in* the format and what is *not*, we now have the possibility of "parser differentials":

[2]We are not aware of random access formats where this *is* specified.

[3]Ideally, format specifications should not merely state what is "in the format," they should indicate what is required of the format producer and what is required of the format consumer.

where multiple implementations of the same standard can vary one from one another; this is yet another security problem.

C. Our Approach

Note the problem statement from Sec. I-A: **What should a *multi-entry point* parser look like and how do we build one to be correct and performant?** Our approach is as follows:

1) Use a declarative DSL *LoCo* to describe the random access format.
2) From the *LoCo* specification, generate a stateful *multi-entry point* parser which provides all desired *demand-functions* for the format. This *multi-entry point* parser will be 'optimal' in the sense that it caches all sub-results and no parse or computation will be computed more than once.

The *LoCo*[4] DSL is composed by layering *three* distinct computational languages: (1) the *base layer* of primitive parsers; (2) the *XRP* (*eXplicit Region Parser*) layer invokes the primitive parsers. *XRP* is a pure functional language DSL that works similar to parser combinators but it makes the regions *explicit* to which parsers are applied, for random access formats this provides needed control and expressiveness; and last, (3) the *Optim(ℓ)* layer, a novel DSL that describes Directed Acyclic Graphs (DAGs) of computations (in our case, *XRP* computations). When viewing *Optim(ℓ)* as a DAG, a node represents a computation—in *XRP*—and an edge represents a dependency between *XRP* computations.

D. Scope and Summary

At this point in our development, we consider *LoCo* to be at the "concept language" stage. It is currently implemented as DSLs in Haskell and we have found it elegantly useful[5]. Our focus in this paper is to communicate the primary ideas, applicability, and benefits of our approach. The creation of a more robust tool-set is future work; a more formal semantics is also, alas, future work.

In the following section Sec. II we explain *LoCo* using examples; in Sec. III we go into further details of *LoCo*'s multiple layers (base layer, *XRP*, and *Optim(ℓ)*); and in Sec. IV we assess and conclude.

II. *LoCo* BY EXAMPLE

The current section attempts to describe by example the main features of *LoCo*. While further details of the layers of *LoCo* will be discussed in the following section (Sec. III), in *this* section, for the sake of preciseness, we may distinguish the layers of the *LoCo* code into *Optim(ℓ)* constructs and into *XRP* constructs.

We have implemented a *LoCo* parser, compiler, and interpreter inside Haskell[10] via Template Haskell[15]. This has allowed us to quickly get *LoCo* working and type-checking, so we can focus on exploring and developing the features and concepts of *LoCo*.

[4]*LoCo* adds "Location Constraints" to sequential primitive parsers.
[5]*LoCo* can be downloaded from [8]

```
pICC : Parser [TED]
pICC = do
  cnt ← pInt4Bytes
  tbl ← pMany cnt pTblEntry  -- parse cnt Table Entries
  rsTeds ← except $ mapM getSubRegion tbl
  teds   ← mapM applyPTED rsTeds
  return teds

-- parse a Tagged Element Data (TED):
applyPTED :: Parser TED
applyPTED (sig,offset,size) =
  withParseRegion offset size (pTED sig)
```

Fig. 4. ICC: The Traditional Approach

```
[optimal|
icc : Region → ICC
icc rFile =
 { (cnt,rRest) =<| pInt4Bytes                    @!  rFile    |>
 , tbl         =<| pManySRPs (v cnt) pTblEntry   @!- rRest    |>
 , rsTeds      =<| except $ mapM (getSubRegion rFile) (v tbl)  |>
 , teds        =<| mapM applyPTED rsTeds                       |>
 }
|]

applyPTED r = pTED (region_width r) `appSRP` r
```

Fig. 5. ICC: Using *LoCo*

```
[optimal|
icc_lazyVectors : Region → ICC
icc_lazyVectors rFile =
 { (cnt,rRest) =<| pInt4Bytes                      @!  rFile  |>
 , rsTbl       = generate (v cnt)
                   <| λi→ regionIntoNRegions
                            (v cnt) rRest (width pTblEntry) i  |>
 , tbl       = map rsTbl <| λr→ pTblEntry @$$ r                |>
 , rsTeds    = map tbl   <| λr→ except $ getSubRegion rFile r  |>
 , teds      = map rsTeds <| applyPTED                         |>
 }
|]
```

Fig. 6. ICC: Using *LoCo* with Lazy Vectors

```
icc rFile = {
  ...
  , crsFile     =<| makeCanonicalRegions (r cnt : r tbl : rsTeds) |>
  , isCavityFree=<| hasNoCavities $ R.complementCRs rFile crsFile |>
  , teds_safe   =<| if isCavityFree
                      then return teds
                      else throwE ["teds_not_safe"] |>
}
```

Fig. 7. ICC: Extending With Constraints

A. A Quick Introduction to LoCo Syntax

In the *LoCo* (Fig. 5) there are a few syntactic matters to cover.

Primitive parsers, and other code, are defined in Haskell as top level definitions. The *LoCo* code is defined inside a Template

Compare Fig. 4—where ICC is implemented in Haskell as a traditional parser—to Fig. 5 where ICC is implemented using *LoCo* "inside the Haskell."

Haskell quasi-quote [12] like this:

```
[optimal|
-- The Optim(l) layer of LoCo is here:
module1 .. = { ... }
module2 .. = { ... }
...
|]
```

An *Optim(ℓ)* module is defined in Haskell-like syntax as a set of bindings

```
[optimal|
icc rFile =
  { pat1 =<| Haskell XRP code here |>
  , pat2 =<| Haskell XRP code here |>
  , ...
  }
|]
```

The right hand side of a binding is where we see the *XRP* layer of *LoCo*, this Haskell *XRP* code must be surrounded by delimiters. An individual binding is written thus:

```
(var1,var2,...) = <| YOUR HASKELL XRP CODE HERE |>
```

The left hand side of the binding `(var1,var2,...)`, is similar to a Haskell irrefutable pattern.

The Haskell code on the right hand side (RHS) of *Optim(ℓ)* bindings can refer to any Haskell definition in scope at the top level as well as all *Optim(ℓ)* bindings in the same *Optim(ℓ)* module: one needs no extra syntax or programming construct to refer to these bindings. Note that *Optim(ℓ)* bindings cannot be defined recursively.

B. ICC Traditionally and "Optimally"

The reader, hopefully armed sufficiently to read the *LoCo*, can now observe three key differences between the sequential Haskell and the *LoCo*:

- In the sequential Haskell, the ordering is top to bottom and is strict.

  In *LoCo*, we have a set of *bindings* in which there is no ordering implied, any sequencing is only that implied by the *data dependencies* between *bindings*.
- In the sequential Haskell, we have a single parser function `pICC` which returns the final result of type `[TED]`.

  In *LoCo*, we define a *module* `icc` which has no designated or main result, any of the bindings could be demanded, e.g., `icc.teds` or `icc.cnt`.
- In the sequential Haskell, the point at which parsing starts is implicit, a parser starts where the previous one ended. When there is a "seek" (jump, or change of location), this implicit location needs to be overridden; in Fig. 4, the `withParseRegion` primitive is used for this: it changes the "point of parse" and also limits the extent of the parse.

  In *LoCo*, the regions are always explicit, e.g., `rFile`, `rRest`. One must have a region in order to apply a primitive parser to the input.

C. Compiling LoCo to Multi-Entry Point Modules

The `icc` module is compiled to a stateful *multi-entry point* module with the following API:

```
Monad m ⇒
  applyICC   :: String → M  -- no computation done, never fails
  get_cnt    :: M → m Int
  get_tbl    :: M → m [TblEntry]
  get_teds   :: M → m [TED]
```

(If so desired, we could also export the region variables and have these added: `get_rRest`,`get_rsTeds`, ...)

Each of the *demand functions* in the API will only execute the computation required to evaluate its associated binding. For instance, the `get_tbl` demand will

1) force all dependent bindings of `tbl`; thus forcing the `(cnt,rRest)` binding (thus evaluated and cached).
2) evaluate the RHS of the `tbl` binding, and cache it.
3) return that value to the user.

This works similarly to lazy evaluation in non-strict languages.

D. Fine Tuning With Lazy Vectors

As an example, if we only need the second value returned by `get_teds`, all the TEDs in the ICC file would still be parsed and the list returned to the user. If any of the other TEDs resulted in a parsing error, `get_teds` would fail. This is not always what we want: this makes our tools brittle when we'd prefer them to be robust. Inspecting the format, we see that to parse the second *TEDs* value, we only need to parse the *Tag count*, the second *Tag Table* entry, and the region referenced by the second *Tag Table* entry. To achieve this greater laziness, Haskell's laziness doesn't help: *Optim(ℓ)* has no knowledge or requirements on the values bound and it uses strict evaluation when it requires values. So we need to add lazy vectors directly into *Optim(ℓ)*. Lazy vectors will allow us to capture element-wise dependencies between two vector bindings. Fig. 6 shows how we can write `icc` to use *Optim(ℓ)*'s lazy vectors, When we compile the `icc_lazyVectors` we get a stateful *multi-entry point* module with this API:

```
Monad m ⇒
  applyICC   :: String → M  -- no computation done, never fails
  get_cnt    :: M → m Int
  get_tbl    :: M → Index → m TblEntry  -- vector indexing
  get_teds   :: M → Index → m TED       -- vector indexing
```

Where the "vector" elements `tbl` and `teds` now generate demand functions that require an `Index` argument. With these, we can *optimally* read the second value of `get_teds` by calling the `icc_lazyVectors.get_teds 1` demand function, which demands

- `teds.1`, which demands
- `rsTeds.1`, which demands `rFile` and
- `tbl.1`, which demands
- `rsTbl.1`, which demands `cnt` and `rRest`

So, `icc_lazyVectors.get_teds 1` requires very little of the ICC file to be traversed and parsed.

E. Constraining Modules

In the `icc` module specification, we can extend the module with further bindings, refer to Fig. 7 where we now have safe (`get_teds_safe`) and unsafe (`get_teds`) ways to access the same data, their equivalence (when the file is `cavityFree`) is clear from the *LoCo* code.

Using explicit regions, we can use library functions to check for sanity among *all* the regions parsed: `makeCanonicalRegions` ensures no region overlaps with another; `hasNoCavities` ensures there are no undue cavities in the file. These functions allow us to address the security issues with regions and cavities referred to in Sec. I-B.

III. Deconstructing *LoCo*: *Optim(ℓ)* and *XRP*

In this section we will dive into further details of *LoCo*.

A. The Three Layers of LoCo

In Sec. I-C we introduced the three layers of *LoCo*:

1) The base layer of primitive parsers.
2) The *XRP* (*eXplicit Region Parser*) layer invokes the primitive parsers. *XRP* is a Haskell "parsing combinator library" with some unique features: parsers must be passed regions explicitly, the regions are kept abstract, and the *XRP* code is in the Reader plus Exception monad (a commutative monad).
3) The *Optim(ℓ)* layer (written inside a custom Template Haskell quasi-quote [12]). *Optim(ℓ)* is effectively a syntax for writing DAGs of computations in the computation language *l*. The name *Optim(ℓ)* tries to suggest the parameterization of *Optim(ℓ)* over many *l* languages.

   Declaratively, the purpose of *Optim(ℓ)* is to abstract over the dependencies between sub-computations (sub-parsers or the like) and thus to avoid over sequentializing our specifications. From the implementation point of view, *Optim(ℓ)* will be compiled into lazy, on-demand *multi-entry point* APIs, i.e., optimal, stateful demand functions.

We've been showing all three layers in our *LoCo* code, but let's make it more explicit. Note the following snippet of our previous code:

```
f = return 5
[optimal|
icc : Region -> ICC
icc rFile =
 { (cnt,rRest)= <| pInt4Bytes @! rFile |>
 , tbl = <| pManySRPs (v cnt) pTblEntry @!- rRest |>
 ...
 }
|]
```

We see the top level Haskell code in black. The *Optim(ℓ)* is next, defined at the top level and delimited by our quasi-quotes [optimal| ... |]. *Optim(ℓ)* doesn't look inside the <| ... |> quasi-quotes, this is where the Haskell *XRP* code goes, *XRP* is Haskell of the proper type. And, in this example, there's only one primitive parser called (what we call the base layer), indicated by pInt4Bytes.

B. The Optim(ℓ) DAG computation DSL

Note the simplicity of *Optim(ℓ)*, it doesn't do much:

- *Optim(ℓ)* does little computation of its own, the computation is embedded as *l* expressions in the bindings;
- *Optim(ℓ)* *lacks* any control flow operators;
- *Optim(ℓ)* *lacks* recursion: the bindings **must** form a DAG.

We consider *Optim(ℓ)*'s simplicity as an advantage: for instance, all data dependencies are statically determinable from the syntax of a module definition.

The irrefutable pattern matching of tuples on the left hand sides of bindings is merely syntactic sugar; this could be transformed away easily using standard techniques.

An *Optim(ℓ)* binding can reference Haskell top-level definitions, but it also references other *Optim(ℓ)* bindings. These references to bindings are the edges in our DAG of computations. When an *Optim(ℓ)* binding is demanded, it will demand (force a thunk if you will) the bindings of *every* *Optim(ℓ)* binding in its definition. There is no laziness here, all bindings will be demanded.

C. Lazy Vectors in Optim(ℓ)

As we demonstrated in Sec. II-D, *Optim(ℓ)* allows for lazy vectors. When we are thinking of the code as a DAG, we can view these as *node vectors* where we have element-wise dependencies between equal length *node vectors*. The following bindings show the four primitives for creating, transforming, and accessing lazy vectors:

```
pat = generate int  <| l code |> -- :: Int → (Int → a) → Vec<a>
pat = replicate int <| l code |> -- :: Int → a → Vec<a>
pat = map fieldName <| l code |> -- :: Vec<a> → (a→b) → Vec<b>
pat = index fieldName int        -- :: Vec<a> → Int → a
```

Optim(ℓ) uses the syntax `Vec<a>` (which is not Haskell) to refer to a lazy vector containing values of type `a`. As lazy vectors are a special built-in type of *Optim(ℓ)*, and abstract, we use the non-Haskell syntax `Vec<a>` to suggest this intention.

The primitive `replicate n <|x|>` creates a constant vector of length `n`, `generate n <|f|>` creates a vector containing the values `[f i|i←[0..n-1]]`, and `map fld <|f|>` creates a new lazy vector with the values of the Haskell list `map f (toList fld)`. The primitive `index fld i` indexes into `fld` which needs to be a lazy vector. Note that `index` may fail if `i` is out of bounds.

D. The l in Optim(ℓ): Commutative Computation Languages

Optim(ℓ), as the name tries to suggest, is parameterized over the language *l* of computations, where *l* must be a *commutative monad*. A commutative monad is a monad in which the order of actions does not matter, i.e., a monad where this law holds for all program contexts `A,B,C`:

```
  do {a ← A; b ← B; C[a,b]}
= do {b ← B; a ← A; C[a,b]}
```

This is the key design decision in *LoCo*: **Separate the DAG evaluation language from the commutative computation language!**

By restricting our computation language *l* to be a commutative monad[6] *Optim*(ℓ) becomes a deterministic language: the order of demands cannot change the results we get from our demand functions. It turns out that this "restriction" is satisfied in many useful cases and provides the key to parallel, optimal, and deterministic behavior. Examples of commutative monads in Haskell are `Identity` (i.e., pure code), `Maybe` (i.e., exceptions), and `Reader` (i.e., read-only globals), as well as combinations thereof. Monads which are not commutative are `StateM` (mutable globals) and `IO`. However, one could have a subset of `IO` that is commutative, e.g., using `IO` in a "read-only" manner, such as reading configuration files or program input. Although the restriction of *l* to commutative monads lessens the applicability of *Optim*(ℓ), we note that giving up a clean, deterministic, and functional semantics is not a price we choose to pay to achieve "greater applicability."

The next section will show the *XRP* language, a commutative monad as it is effectively Haskell's Reader plus Exception monad.

E. The XRP DSL: explicit, abstract regions for random access formats

In hindsight, the design of *XRP* is pretty simple and obvious:

- Start with a parser combinator library designed with the `State` (holds the point of next parse) and the `Exception` monads (we want to be able to fail).
- Rewrite that library to be a `Reader` (holds the input) and `Exception` monad where we explicitly plumb the region through all the parsers.
- Ensure regions are *abstract* by designing an algebra of regions, the *XRP* program does not have access to the concrete region.
- At the base level, primitive parsers must always be applied to the region.

A key design decision for the *XRP* parser combinator library was **not** to add a 'seek' primitive (which would necessitate a stateful implementation), combinator library, but to "lift" non-seeking parsers into the explicit region language.

In the code in Sec. II, we saw a few examples of *XRP* region combinators:

- parsers that don't consume their whole region and which return a remaining region,
- the `getSubRegion` combinator which extracts one region out of another,
- `regionIntoNRegions n` which can slice a region into `n` equal width regions.

Even the authors can find writing explicit regions in *XRP* irksome at times. (Hint: when you *do* have sequentiality of parsers, you can use a combinator for that, no explicit regions needed.) However, compared to the sequential implementation, we find many things pleasant about writing the ICC format in *Optim*(ℓ): (1) the `icc` module is more declarative, no ordering is needed, (2) the code generated from `icc` is optimal and more general, (3) constraints can be added without cluttering the main parser code.

The explicit regions are essential to allowing achieving commutativity and essential to accurately determining the dependencies. *Optim*(ℓ) needs to see *all* the dependencies between the bindings, *especially* when those dependencies are region variables.

The alternative, an implicit location with some form of overriding primitive, is inherently stateful and imperative. We'd consider it harmful, similar to a *goto* statement [3] insofar as it precludes any form of local reasoning, any parser is now at liberty to "go seeking" anywhere and as many times as it wants.

In other words, the explicit region approach is pure functional programming, the implicit location approach is programming with a mutable global variable (sometimes convenient, but it can bite).

F. Implementation: Current and Future

Implementing *Optim*(ℓ) in a meta-language in Haskell using Template Haskell [15] and quasi-quotation [12] gave us many advantages

- the ability to write the *Optim*(ℓ) parser easily,
- easy access to Haskell syntax,
- the ability to borrow Haskell's type-checking, and
- proof that *Optim*(ℓ) can be separated from our *l* computation language in an elegant fashion.

In the future, we plan to implement *LoCo* as a stand-alone compiler. There is nothing in the design of *Optim*(ℓ) that precludes us from having a stand-alone compiler from *Optim*(ℓ) with $l = L$ that generates code in the language L, provided L is relatively complete. For a desired language L, we would need to write a library in L to support the basic glue (thunks and tracking dependencies) between the computations. In fact, as future work, we would like to break the tie between *Optim*(ℓ) and Haskell by implementing the system as a stand-alone compiler in which we can target a set of computation languages; e.g., Haskell, C, and Rust. Not only would a stand-alone compiler allow for additional target languages, it would allow for easier extensions to our simple compiler, e.g., analyze and optimize the generated code, such as removing unneeded thunks when a binding has only one dependent.

[6]This is enforced by convention, our Haskell based implementation cannot enforce it.

IV. Conclusion

A. Assessments

Although *LoCo* is at the "concept language" stage, we have an implementation in Haskell has kept us honest. *LoCo* can be downloaded from [8], a toy implementation of ICC is included.

We have struggled with the ramifications of the current design tradeoffs: *LoCo* can feel a little complicated, wouldn't adding a seek primitive to a traditional parser be simpler? Explicit regions make *LoCo* less readable (than a parser defined with implicit, sequential regions) and code with explicit regions is more tedious to write. But, as we have gained familiarity with our new language, we feel that the high road (pure, no seek with state, commutativity) is a better road, or at least a road that should be explored further. Further observations are in Sec. III-E. It should be noted that *LoCo* was designed for real world random access formats, that's where it should shine.

We have found that it is possible to reduce the need for explicit regions with effort, and certainly where there is no desire for a "demand function" for an element, one can eliminate an *Optim*(ℓ) binding.

However, we think the advantages of *LoCo* are compelling

- *Optim*(ℓ) modules are declarative;
- the separation of declarative semantics and the complex imperative interface;
- with lazy vectors and regions, *Optim*(ℓ) can provide optimal and fine-grained *demand functions* which would be infeasible to do by hand;
- *Optim*(ℓ) gives a tool that is more general and useful than a parser;
- with one clear, simple *LoCo* specification, *LoCo* allows for accessing sub-results, full results, and constrained results from the parser;
- more secure regions (as mentioned in Sec. II-B): detecting cavities and region overlaps

B. Alternatives

What are other approaches to handling random access formats?

The Classic Shotgun

It appears that the most common approach is rolling ones own (i.e., shotgun parsers):

- If the primitive parsers are of the "scanf" variety, then the use of ad hoc code to orchestrate the parsers makes sense.
- If the number of "demand functions" are small, even if the parsers are complicated, this is another case where ad hoc code will probably be sufficient.

The biggest disadvantage here is that correctness is not at all clear, and the parser is not localized to a single place in the code where it can be checked for correctness.

Retrofitting Parsing Technologies with 'seek', etc.

Galois' highly expressive DaeDaLus parser [7] does exactly this. DaeDaLus has the features of a combinator library and grammar based parser, it supports seeks and more. DaeDaLus is a great tool for generating basic parsers even for very complex formats. The gap only became clear when we tried to use DaeDaLus to create a large set of *demand functions* for various parts of the PDF; we then realized it's still not solving the "keep track of the dependencies and cache the result" problem, and to do this we were just writing everything by hand, except the primitive parsers themselves. Retrofitting needs more than 'seek', it needs to create a stateful *multi-entry point* module; and creating such a module requires knowing the full format.

KaiTai Struct

KaiTai struct [14] is an excellent tool *suite* for describing *binary* formats. It goes beyond *LoCo* as it supports both reading and writing formats: and it has features that support *regions*. Also, it generates imperative code that implements caching *demand functions*.

Some things that distinguish our approach from KaiTai struct are

- *LoCo* uses real programming languages (vs. YAML) and thus we have abstraction (no duplication of code), more programming language types and sum types.
- In our opinion, *LoCo* has fewer and simpler concepts than KaiTai struct and is more declarative.
- *LoCo* supports the notion of lazy vectors.
- *LoCo* is less focused on binary formats, and can work, as originally designed, with both binary and textual (PDF) formats.

C. Related Work

We are not aware of work particularly close to our topic here: **writing declarative descriptions of random access formats from which we can generate code for optimal *demand functions*.** But there is related work further afield:

Tools. There is the previously discussed (Sec. IV-B) KaiTai Struct tool [14].

Format Description. Zhang et al [17] deal with defining region based parsers as we do, though from a more theoretical perspective. They focus on the specification of formats, not dealing with *demand functions* or optimality. Their *Related Work* section is an excellent survey: they note how frameworks for data-dependent grammars are generally not sufficient to describe region based parsing, e.g., PADS [11]. They also discuss tools that *can* describe region based parsing, KaiTai Struct [14], Nail [2], DataScript [1], and FlexT [5]. Similar to our discussion in Sec. II-B, they note the imperative, *goto* like, nature of these descriptions.

Nail [2] is worth calling out individually, it is focused on generating parsers and generators (unparsers or encoders) using PEG grammars. Like KaiTai struct, it allows for generating both the parser and the generator from the protocol definition.

Nail falls into the add a seek primitive language design camp and Nail does not attempt to support partial parsing.

In comparing *LoCo* with many of the above approaches, a key difference is that *LoCo* focuses on the *Optim(l)* partial parsing capabilities (few do) and its method of adding "seeking" (most don't allow or add primitives); the *LoCo* approach is quite agnostic with respect to particular parsing technologies used in the "base layer", e.g., using PEG parsers as Nail does.

D. Future Work

As mentioned in Sec. III-F, we have plans to implement *Optim(l)* as a stand-alone compiler, this would allow us to

- Target further *l*'s as mentioned in Sec. III-D.
- Do further analysis and optimization of the *Optim(l)* code (e.g., remove unneeded thunks when a node has only one dependent).

In order to gain more experience by which to improve *LoCo*, we plan to implement more random access formats. Coming full circle from where we started this paper, an excellent stress test of *LoCo* would be to use it to implement a PDF parser.

A more ambitious research topic to explore is using *LoCo* to generate encoders as well as decoders. In many cases the constructs in the bindings are bijective and the use of *Optim(l)* variables is linear, so this certainly gives us a reasonable start.

Acknowledgments

This research was supported in part by DARPA awards HR001119C0073 and HR001119C0079.

References

[1] Godmar Back. DataScript- a specification and scripting language for binary data. In Don Batory, Charles Consel, and Walid Taha, editors, *Generative Programming and Component Engineering*, pages 66–77, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[2] Julian Bangert and Nickolai Zeldovich. Nail: A Practical Interface Generator for Data Formats. In *2014 IEEE Security and Privacy Workshops*, pages 158–166, San Jose, CA, May 2014. IEEE.

[3] E. W. Dijkstra. Go To Statement Considered Harmful. In *Edsger Wybe Dijkstra: His Life,Work, and Legacy*, volume 45, pages 315–318. Association for Computing Machinery, New York, NY, USA, 1 edition, July 2022.

[4] D. Harel and A. Pnueli. *On the development of reactive systems*, page 477–498. Springer-Verlag, Berlin, Heidelberg, 1989.

[5] Alexei Hmelnov Hmelnov and Andrei Mikhailov. Generation of Code for Reading Data from the Declarative File Format Specifications Written in Language FlexT. In *2018 Ivannikov Ispras Open Conference (ISPRAS)*, pages 23–30, November 2018.

[6] ICC. Specification ICC.2:2019 (Profile version 5.0.0 - iccMAX) Image technology color management - Extensions to architecture, profile format and data structure [REVISION of ICC.2:2018], 2019.

[7] Galois Inc. Daedalus. https://github.com/GaloisInc/daedalus.

[8] Galois Inc. Loco. https://github.com/GaloisInc/LoCo.

[9] ISO TC 171 SC 2 WG 8. *ISO 32000-2:2020 Document management - Portable Document Format - Part 2: PDF 2.0*, volume 2 of *ISO 32000*. ISO, December 2020.

[10] Simon Peyton Jones. *Haskell 98 language and libraries: the revised report*. Cambridge University Press, 2003.

[11] Kathleen Fisher and David Walker. The PADS Project: An Overview, 2011.

[12] Geoffrey Mainland. Why it's nice to be quoted: Quasiquoting for haskell. In *Proceedings of the ACM SIGPLAN Workshop on Haskell Workshop*, Haskell '07, pages 73–82, New York, NY, USA, September 2007. Association for Computing Machinery.

[13] NDSS Symposium. NDSS 2021 Shadow Attacks: Hiding and Replacing Content in Signed PDFs, February 2021.

[14] KaiTai Project. Kaitai Struct: Declarative binary format parsing language. https://kaitai.io/.

[15] Tim Sheard and Simon Peyton Jones. Template meta-programming for Haskell. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell*, Haskell '02, pages 1–16, New York, NY, USA, October 2002. Association for Computing Machinery.

[16] Mark Tullsen, William Harris, and Peter Wyatt. Research report: Strengthening weak links in the pdf trust chain. In *2022 IEEE Security and Privacy Workshops (SPW)*, pages 152–167, 2022.

[17] Jialun Zhang, Greg Morrisett, and Gang Tan. Interval parsing grammars for file format parsing. *Proceedings of the ACM on Programming Languages*, 7(PLDI):1073–1095, June 2023.