This paper introduces a pragmatic approach to deal with Data Description Languages.

What is a data description language (DDL)? The authors give a number of characteristics, most of which describe the absence of desirable descriptions or properties (grammar, parser, data contains errors, etc). I think the only unique concept of a DDL is that it may contain dependent types. Also, a DDL parser may have mechanisms for dealing with errors, but that holds for some parsers for context-free grammars too, although this is not specified in the grammar.

A data description language is defined as an element of DDC, which is a calculus/language for specifying types. From this type, the authors derive several components, in particular the abstract syntax in the host language, and a parser that parses sentences of the DDL into the abstract syntax. The representation type, the parser, and the error-handling mechanisms satisfy several desirable properties, such as that the generated parser is type correct.

I think this paper should be accepted after revision. I think the idea of generating types and parsers from type descriptions possibly containing dependent types and program code is new and interesting, and deserves to be published. I like the fact that advanced programming tecniques are used to produce software that is usually produced in an ad-hoc fashion. Furthermore, the authors have performed a rigorous study of the topic. I buy the argument that you need a separate tool to produce both a type, and a parser for the type. I think you could write these components in a dependently-typed programming language such as Agda, Epigram, DML, or in a generic programming language extension such as Generic Haskell, but it is maybe not realistic to expect ad-hoc data users to program in these languages. This paper might stimulate authors of advanced programming languages.

I do have some questions and comments, which I list below.

- Although the authors spent quite some time on explaining what a DDL is, I would like to see some final conclusion. I suppose currently the definition of a DDL is an element of the DDC calculus. I would like to see something like: a DDL is a context-free grammar extended with ... and restricted to ..., or a two-level grammar with the following restrictions, or a ... I would like to see a description of the relation between DDL's and existing formalisms early in the paper. The authors do give a comparison, but that appears in the end, and is more about tools than formalisms.

- `One explicit non-contribution of this work is the development of new techniques for recognizing any particularly interesting class of grammars. The parsers generated by DDC or PADS are completely ordinary recursive-descent parsers.' (p7) With recursive-descent parsers you have to watch out for left-recursion. This is nowhere mentioned in the paper. I suppose your program crashes with a left-recursive value of DCC? And I suppose nothing prevents the DDC author to define a type which puts matching tags around the file, resulting in a parser which can only start to produce output after parsing the entire document?

- Error-correction behaviour depends on the kind of parser you use.

For example, in their ICFP 2003 paper, Swierstra and Hughes show how you can use continuation-passing style to obtain better error messages faster. Other work of Swierstra also focusses on fast parsers that do error correction. You might spend some text explaining the consequences of taking a recursive-descent parser for error correction.

- Although I think it is good to perform a thorough study of the subject, I think the lemma's and theory at the end of the paper are a bit much. Of course I want the desirable properties such as generated parsers are type correct. But I didn't get much out of all the lemma's in 5. Lemma 26 should be removed, or added to the definitions on page 22/23. When I read this code, I infer the type, and I find it very strange to get a lemma with the types only 10 pages later.

Small comments:

p6: an clean := a clean
p6: prove our implementation technique was correct. := prove our implementation technique correct.
p8: write C (e) for a base type parameterized by a (host language) expression e. := write C (e) for a base type C parameterized by a (host language) expression e.
p9: closed-paren ... english? I'm not a native speaker.
p10/11: I've found an example of Prec alpha = t (tree_t), but not of Prec alpha.t Might be nice to add?
p11: Why do you add the alpha to your DDC? This suggests there also is a DDC without an alpha, which is not the case (at least not in this paper).
p11: i.e.the := i.e. the
p12: I find the sentence: `The type tau_t is used when data following the array will signal termination.' a bit strange.
p13: (and later): occuring := occurring
p13: Fig 8: don't you want to include all values in your expressions? Now you just include c and fun.
p15: `When appearing in F_omega judgments, such contexts may also contain type-variable bindings of the form alpha::kappa.' Strange. Where do you do that? Which judgements?
p16: rule Base := rule Const
p17: I only understood the `(implicitly)' after reading the explanation on p18.
p19: In Fig. 13 (and also at later points in your paper) I got the impression that you first want to do an inductive step over the kinds, and only then inductive definition over types, a la type-indexed functions have kind-indexed types from Hinze in Science of Computer Programming 2002. That would explain the funny translation of \alpha.x. Now you say, for example, that each PD type has a header and a body, which is not true of course, exactly because of the higher kinds. Can the semantics be split into two steps: first translating the higher-kinded types, and then types of kind T?
p19: What is the type-specific metadata?
p21: last line in def of tau seq(...): w' should be w. In the previous continue, it is probably better to take w' instead of w?
p22: In fun P_Sigma: where does that h come from? I think it becomes clear later, but it confused me here.
p26: B_opty is nowhere defined.
p27: Shouldn't the proof of lemma 3 mention normalization instead of evaluation?

p27: Explain the names of the lemmas: what do you mean with inversion, for example?
p27: `determinacy of sibngle-step evaluation' where did you state that?
p28: Lemma 9 (2): where does tau' go? I find the proof of the lemma very unconvincing: part 4 and parse_rep? [[tau]]_P in part 6? I think something has gone wrong in this proof.
p29: What does inverting kinding rules mean?
p29: invertable := invertible
p29: Lemma 14: the tau should be nu?
p30: Def 18, point 2. whenever H(tau' : T), we have H(tau tau' : kappa) := for all tau' . H(tau':T) => H(tau tau' : kappa)
p33/34: I expect some of the substitutions in the subscripts should be r_1,p_1 instead of r,p. For example in the dependent sum in Def 29, and the pair-case in Lemma 31.
p36: `intesection' `elobaration'
p37: I find the use of x at both the lhs and the rhs of the definition of Pdatatype confusing. The same in the typing rule below. Nonuniform datatypes are nested datatypes in the sense of Bird and Meertens (MPC'98)? Can you handle mutually recursive datatypes? I don't understand the second premise of the tau seq(... rule. What does \((len,elts),p).len=length mean?
p39: I find the bug hunting paragraph not very convincing. Wouldn't you have received the same result if you would have done a through code review?
p40: `highlighed'
p42: Parser combinator libraries are programmable, so some of the arguments comparing DDC to parser combinators are not valid. Also, the error-reporting mechanism of parser combinator libraries can be adapted. I think Swierstra's work is relevant here.
p43: Implementing a PADS-style language for any standard imperative, functional or object-oriented requires := Implementing a PADS-style language for any standard imperative, functional or object-oriented language requires