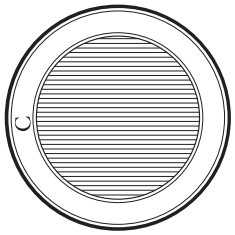
A Validated Parser for Stan

Brian Ward *Advisors: Joseph Tassarotti, Jean-Baptiste Tristan*

Boston College Computer Science Department



Abstract

Before any code can be interpreted, compiled, or otherwise processed, it must first be parsed. This process of mechanically reading and interpreting code is a well-trodden area of compiler design, and one that has many existing tools and methods. A programmer can succinctly specify the syntax of a language and use a variety of tools, known as parser generators, to create code which implements the translation from that syntax into a tree structure for evaluation or compilation.

Recent developments in the field allow for a way to be certain that this translation is done following the desired specification. Rather than simply trusting a handwritten parser, or trusting those who write parser generators such as

Contents

4. A set of *productions* that provide rules for translating from a non-terminal to terminal and non-terminal symbols. By convention, a set of productions is generally used as the complete specification of the language, with the start symbol's productions listed first. Productions are written as *nt ! derivation* or *nt* ::= *derivation*, in a format known as Backus-Naur form [2]. If there are multiple productions for a given non-terminal, the various options can be written as *x j y*.

This is usually easier to understand by an example. Perhaps the simplest non-trivial CFG is the following single production:

expr ::= `x' *j expr* `+' *expr*

This grammar contains one non-terminal (*expr*) and two terminal (x, +) symbols. It defines an infinite language of repeated additions. Some members of this language are *x*, *x* + *x*, and *x* + *x* + *x* + *x* + *x*. Because this production features the same non-terminal on both sides of the translation, it is said to be a recursive rule. It is possible to be more specific in some cases, e.g. a rule *a*

the preservation of semantic meaning during this translation. Stan programs must change meaning during compilation from a description of a random process into a program that simulates that randomness. It is worth noting that while the output programs may be pseudo-random, the compilation itself is deterministic.

While the nature of Stan as a PPL is not necessary to understand in order to parse it, this does provide one of the philosophical underpinnings for this project. Testing that the compilation translation was done properly can be very di cult because programs that simulate randomness are particularly hard to test by traditional means. Take for example a function that returns a value between 0 and 1 uniformly at random. How does one test that this function is well-behaved? It is hard to say much more about an individual run of

```
data {
        int<lower=0> N;
        vector[N] x;
        vector[N] y;
}
parameters {
        real alpha;
        real beta;
        real<lower=0> sigma;
}
model {
        y ~ normal(alpha + beta * x, sigma);
}
```

Fig. 3: An example linear regression Stan model [16, User Guide §1.1].

1.3 Formal Verification

Formal Verification is the practice of proving properties of a program through standard mathematical tools. Such proofs are often performed in a semi-automatic fashion with the aid of a *proof assistant*. These are tools that automate some basic parts of proofs while requiring the programmer to provide the remaining steps. Coq is both a proof assistant and a functional programming language [14]. This means one can both write programs and prove properties about them in the same language. That is to say one writes software that is *correct by construction*.

There are several intellectual hang-ups associated with formal verification of software. If software can be written correctly without testing, why is that not the standard practice? Is it even possible to have a proof written in a way computers can understand, check, and assist with?

The answer to both of these hinges on the work the programmer must do. It is easy to prove many simple things on a computer, and the computer can assist with repetitive, simple tasks | indeed, this is what computers do all the time in other facets of work. Unlike many proofs in mathematics, proofs of software correctness are often intellectually simple, it is just the size of the problem that makes it infeasible to prove `by hand'. There are few requirements for `tricks' or the invention of new techniques, just the repeated application of many basic principles such as induction or case analysis. The computer can be quite good at solving incredibly large and repetitive (but ultimately quite `dumb') proofs.

Writing proofs in such a way to leverage this capability can a difficult and time-consuming task for the programmer, but it is far from impossible. We will discuss in Section 1.3.1 one example of a large fully verified program which shows that it truly can be done in practice. Additionally, as discussed above, some problems lend themselves more naturally to formal methods; for problems that the more traditional means of testing would also require a good deal of work, or for which complete testing is infeasible, there is a prime opportunity to use the tools of formal verification to avoid these pitfalls.

Coq is written in OCaml and has many similar features as a programming language. It features the ability to be \extracted," a process of mechanically translating Coq code to OCaml or other languages. Coq's proof abilities are based on the formal language of the Calculus of Inductive Constructions [9]. It is not essential to understand these details {

input program" can take a massive amount of time to resolve, but \Missing) on line 32" is fixed almost instantly.

After successfully transforming the Stan reference parser into one suitable for verification by the Coq mode of Menhir, we considered how best to generate these error messages. Based on the existing CompCert example, and the rest of the Menhir ecosystem, we identified three options for proceeding:

1. Follow the lead of CompCert and use a second { unverified { parser that sits in front of the verified parser and uses Menhir's \incremental" (or \table") mode. This is a backend that, like the Coq mode, changes the style of parser produced and is used in Menhir's standard error messaging techniques. This would be a simple solution that clearly has been used before.

2. Modify Menhir's Coq mode significantly to allow it to be run in the same incremental style that enables the standard error messaging (and additional error recovery features) that are available in Menhir's \table" backend.

3. Modify Menhir's Coq mode to return the parser's state and other contextual information when an error state is entered.

We considered each choice in turn.

2.2.1 A Second Parser

The most straightforward choice would be to use a second parser to handle the error messaging. This is done in CompCert's C compiler, which also does lexical feedback through this initial parser [10]. The grammar structure of this additional parser would be the same, and Menhir supports flags for automatically generating grammar specification files with the semantic actions removed. This would allow someone who chooses this path to automatically create the second parser as part of their build system with relatively minimal effort.

The main arguments against this approach are twofold. First, this approach adds nothing to the existing tools and methods. It is entirely trodden terrain. Secondly, parsing twice is both inelegant and (at least for Stan) unnecessary.

2.2.2 Verified Incremental Mode

This second option would recreate the existing behavior of Menhir's incremental API, but in the verified mode. This alternative backend produces parsers that perform one step of parsing at a time and then yield their (partial) results to the calling code. Syntax errors can be handled while they occur, rather than the entire parse failing, and this allows both error-messaging and error-recovery to be done in a natural and elegant way. Two concerns made this option undesirable.

First, this is the approach that would require the most changes to Menhir. Any existing use of the tool would need significant changes. Furthermore, the ultimate benefits that those users received would be relatively limited | with a few notable exceptions, such as the Merlin language server [5], the incremental mode is often used as simply a very large hammer to

solve the very small problem of error messaging. The additional features it enables, such as allowing error *resolution*, not just recognition, are unnecessary for our (and many other's) use case.

It is these very features that actually lead to a second reason to not pursue this path. Incremental parsing is ultimately driven by the lexer or an external loop, not the parser itself. This means any true verification of the parser would require proof that the lexer and other code also maintain the invariants required by the parser's correctness and completeness theorems. It would be counter to the entire notion of verifying the parser to allow the lexer to feed back unverified data following each and every computation, and we therefore decided against this method.

2.2.3 Meaningful returns on error

Finally, there was the middle road. The parser generated by Menhir's Coq mode has a sum type for its output, as show in Section 2.1. By modifying the `Fail_pr` branch of this type to include information about the state of the parser during the time the error was detected, we can reconstruct a meaningful error message after failure. In particular, we can return the state of the parser and the last token seen. The state is the piece of information used by Menhir's existing error messaging functionality when in incremental mode, and clever use of the token types (as is employed in both CompCert and our parser) allows the retrieval of crucial context information.

This final option was ultimately selected for use. This required modifications to both Menhir and the coq-menhirlib library.

2.3 Modifications

There are two basic pieces of information that are useful for creating meaningful error messages: the state of the parser (which carries information about what was expected and therefore what went wrong) and the position of the error. The first is relatively simple - it is usually encoded as a number, and the existing Menhir tools rely on these numbers for picking which error message is displayed. The second is not immediately available to the parser, but must be given by the lexer. Luckily, it is quite reasonable to include position information in each token the lexer produces. The token causing the error can thereafter be used as a stand-in for the position of the error in the input.

The modifications to Menhir were motivated by making these two pieces of information available. This primarily required modification to the file `Interpreter.v` in the coq-menhirlib library that is linked to the generated Coq parsers. In particular, the return type of the parsing functions was updated to:

```
Inductive parse_result :=
| Fail_pr_full: state -> token -> parse_result
| Timeout_pr: parse_result
| Parsed_pr: symbol_semantic_type (NT (start_nt init)) ->
        Stream token ->
        parse_result.
```

This change was nearly enough on its own, but the existing proofs and usages of this type all ostensibly needed to be updated to recognize the new `Fail_pr_full`. However, none of these actually require this information (this should be obvious, as they predated its inclusion). Therefore, using the Coq notation functionality, `Fail_pr` was set up as an abbreviation for `Fail_pr_full _ _`. This meant no further modi cations to the library were needed to complete this change.

As mentioned above, the state is often encoded as a simple integer, but in this return type it is provided as a sum type, `state`

over the documentation in areas where they disagreed, as this is believed to represent Stan `in the real world'. We mainly di ered from stanc3's parser to conform to the requirements

The driver code in the larger compiler was modified to display syntax errors, and the relevant portion of this code is available in Appendix A. Finally, the error messages were written for each parser state that can lead to an error. While the `.messages` file from stanc3 was used as a reference, the differences between the grammars meant that these needed to largely be done by hand. Our final parser has 237 possible error-causing states, which correspond to 164 unique error messages.

4 Results

This thesis successfully created a verified parser for Stan for use in a formally verified compiler. This required rewriting the grammar and semantic actions for use with the verified mode provided by the Menhir parser generator. Furthermore, changes were made to Menhir to allow error messaging capability from the verified parser alone, and these changes were included upstream by the Menhir developers.

This work allows the further development of the Stan verified compiler, and it provides greater functionality to anyone seeking to use Menhir to produce realistic, validated parsers.

Another avenue for useful extension of Menhir's Coq mode which was identified, but not pursued, would be the addition of the associativity and precedence declarations which are available in the more typical usage of Menhir. A large amount of manual work was put in to translating the grammar specification from one that used these annotations to one that did not, and this required additional testing to ensure that the grammars still agreed with one another. This serves as both a source of potential human error and a barrier to the adoption of validated parsing. Implementing these would make transitioning from an existing Menhir specification to one which could be used with the Coq backend considerably simpler.

5 Acknowledgments

I would like to thank Joseph Tassarotti and Jean-Baptiste Tristan for their guidance and mentorship throughout the development and writing of this thesis. Additionally, thanks are owed to Robert Muller, for introducing me to them both, and Howard Straubing, for advising me throughout my undergraduate studies and insisting that I write a thesis. Outside of the Boston College Computer Science department, thank you to both Francois Pottier and Jacques-Henri Jourdan for their assistance in modifying Menhir and their eagerness to include my contributions into the tool.

The work described in this thesis was supported by a gift from Oracle Labs.

Finally, I would like to thank my family, my roommates, and my girlfriend, Olivia, for all helping me finish my studies in this very odd year and nodding intently while I described details of my work that were decidedly outside of their interests and areas of expertise.

References

[1] A new Stan-to-C++ compiler, stanc3. https://github.com/stan-dev/stanc3

[15] Francois Pottier and Yann Regis-Gianas. *Menhir Reference Manual*. INRIA, November 2020. https://gallium.inria.fr/~fpottier/menhir/manual.html.

[16] Stan Development Team. *Stan Modeling Language Users Guide and Reference Manual*, 2019. https://mc-stan.org.

A Error Messaging Code

This code is part of my contributions to the full parser project, built on CompCert [13].

```
let location t : Lexing.position * Lexing.position =
  match t with
  (* These four tokens have a payload we ignore *)
  | STRINGLITERAL sp | REALNUMERAL sp | INTNUMERAL sp | IDENTIFIER sp ->
    snd sp
  (* All of the following tokens have no payload, just a position *)
  | WHILE p |
```

B Final Grammar

The following grammar was generated from the Menhir description file using Obelisk [6]. It is written in a common extension of Backus-Naur form, which allows several shorthand

hbasic_type i ::= `INT'
| *j* `REAL'
| *j* `VECTOR'
| *j* `ROWVECTOR'
| *j* `MATRIX'

hunsized_dims i ::= `LBRACK' `COMMA' `RBRACK'

hvar_decl i ::= *hsized_basic_type i* *hdecl_identi er i* [*hdims i*] [`ASSIGN' *hexpression i*] `SEMICOLON'

hsized_basic_type i ::= `INT'
| *j* `REAL'
| *j* `VECTOR' `LBRACK' *hexpression i* `RBRACK'
| *j* `ROWVECTOR' `LBRACK' *hexpression i* `RBRACK'
| *j* `MATRIX' `LBRACK' *hexpression i* `COMMA' *hexpression i* `RBRACK'

htop_var_decl_no_assign i ::= *htop_var_type i* *hdecl_identi er i* [*hdims i*] `SEMICOLON'

htop_var_decl i ::= *htop_var_type i* *hdecl_identi er i* [*hdims i*] [`ASSIGN' *hexpression i*] `SEMICOLON'

htop_var_type i ::= `INT' *hrange_constraint i*
| *j* `REAL' *htype_constraint i*
| *j* `VECTOR' *htype_constraint i* `LBRACK' *hexpression i* `RBRACK'
| *j* `ROWVECTOR' *htype_constraint i* `LBRACK' *hexpression i* `RBRACK'
| *j* `MATRIX' *htype_constraint i* `LBRACK' *hexpression i* `COMMA' *hexpression i* `RBRACK'
| *j* `ORDERED' `LBRACK' *hexpression i* `RBRACK'
| *j* `POSITIVEORDERED' `LBRACK' *hexpression i* `RBRACK'
| *j* `SIMPLEX' `LBRACK' *hexpression i* `RBRACK'
| *j* `UNITVECTOR' `LBRACK' *hexpression i* `RBRACK'
| *j* `CHOLESKYFACTORCORR' `LBRACK' *hexpression i* `RBRACK'
| *j* `CHOLESKYFACTORCOV' `LBRACK' *hexpression i* [`COMMA' *hexpression i*] `RBRACK'
| *j* `CORRMATRIX' `LBRACK' *hexpression i* `RBRACK'
| *j* `COVMATRIX' `LBRACK' *hexpression i* `RBRACK'

htype_constraint i ::= *hrange_constraint i*
| *j* `LABRACK' *ho set_mult i* `RABRACK'

hrange_constraint i ::= [`LABRACK' *hrange i* `RABRACK']

hrange i ::= `LOWER' `ASSIGN' *hconstr_expression i* `COMMA' `UPPER' `ASSIGN' *hconstr_expression i*
| *j* `UPPER' `ASSIGN' *hconstr_expression i* `COMMA' `LOWER' `ASSIGN' *hconstr_expression i*

⟨leftdivide_expression⟩ ::= *⟨pre_x_expression⟩*$^{+}_{\text{`LDIVIDE'}}$

⟨pre_x_expression⟩ ::= `BANG' *⟨exponentiation_⟩*

hprintables i ::= *hprintable i*$^{+}_{\texttt{`COMMA'}}$

hprintable i ::= *hexpression i*
| *hstring_literal i*

hlhs i ::= *hidenti er i*
| *hlhs i* `LBRACK' *hindexes i* `RBRACK'

hstatement i ::= *hclosed_statement i*
| *hopen_statement i*

hatomic_statement i ::= *hlhs i hassignment_op i hexpression i* `SEMICOLON'
| *hidenti er i* `LPAREN' *hexpression i*$_{\texttt{`COMMA'\$SEMICOLON'}}$

hopen_statement i ::= `IF' `LPAREN' *hexpression i* `RPAREN' *hsimple_statement i*
| *j* `IF' `LPAREN' *hexpression i* `RPAREN' *hopen_statement i*
| *j* `IF' `LPAREN' *hexpression i* `RPAREN' *hclosed_statement i* `ELSE' *hopen_statement i*
| *j* `WHILE' `LPAREN' *hexpression i* `RPAREN' *hopen_statement i*
| *j* `FOR' `LPAREN' *hidenti er i* `IN' *hexpression i* `COLON' *hexpression i* `RPAREN' *hopen_statement i*
| *j* `FOR' `LPAREN' *hidenti er i* `IN' *hexpression i* `RPAREN' *hopen_statement i*

hclosed_statement i ::= `IF' `LPAREN' *hexpression i* `RPAREN' *hclosed_statement i* `ELSE' *hclosed_statement i*
| *j* `WHILE' `LPAREN' *hexpression i* `RPAREN' *hclosed_statement i*
| *j* *hsimple_statement i*
| *j* `FOR' `LPAREN' *hidenti er i* `IN' *hexpression i* `COLON' *hexpression i* `RPAREN' *hclosed_statement i*
| *j* `FOR' `LPAREN' *hidenti er i* `IN' *hexpression i* `RPAREN' *hclosed_statement i*

hsimple_statement i ::= `LBRACE' *hvardecl_or_statement i* `RBRACE'
| *j* *hatomic_statement i*

hvardecl_or_statement i ::= *hstatement i*
| *j* *hvar_decl i*

htop_vardecl_or_statement i ::= *hstatement i*
| *j* *htop_var_decl i*