Extending the TPTP Language to Higher-Order Logic with Automated Parser Generation

Allen Van Gelder[1] and Geoff Sutcliffe[2]

[1] University of California at Santa Cruz, USA, `http://www.cse.ucsc.edu/~avg`
[2] University of Miami, USA, `geoff@cs.miami.edu`

Abstract. A stable proposal for extending the first-order TPTP (Thousands of Problems for Theorem Provers) language to higher-order logic, based primarily on lambda-calculus expressions, is presented. The purpose of the system is to facilitate sharing of theorem-proving problems in higher-order logic among many researchers. Design goals are discussed. BNF2, a new specification language, is presented. Unix/Linux scripts translate the specification document into a *lex* scanner and *yacc* parser.

1 Introduction

The goal of this work is to extend the current TPTP (Thousands of Problems for Theorem Provers) language [SZS04] to include adequate support for higher-order logic, while continuing to recognize the existing first-order language. It was motivated by a panel discussion at the Workshop on Experimentally Successful Automated Reasoning in Higher-Order Logic held in conjunction with LPAR-12, December 2005. The panel discussion conveyed the desire to have a common language in which various researchers could express benchmark problems in higher-order logics that would be contributed to a common library along the lines of the TPTP problem library [SS98].

The new language developed in this project is tentatively named HOTPTP. Some of the design goals were

1. The rules of the language should be simple and regular, so that humans can understand them without too much trouble.
2. The rules of the language should be presented in a specification document that has sufficient formality and rigor to be unambiguous, yet is not so technical and complicated that its meaning is obscured.
3. The language should be amenable to *straightforward* automated parser generation, with established tools such as *lex* and *yacc*, or *flex* and *bison*. These tools accept LALR-1 languages. It would be undesirable to rely on tricks and extensions that might be supported in one tool and not another.
4. It should be straightforward to set up a Prolog parser for the language, using Prolog's `read()` procedure to accomplish most of the parsing drudgery.

The rules of the TPTP language, as released with TPTP v3.0.0, already achieved goals (1), (2), and (4) above quite well. The first step of the project was to achieve goal (3). During this initial phase, a few ambiguities were discovered in the existing language, and minor revisions of the rules were implemented to remove these ambiguities without changing the underlying language. Following that work, the task of extending the language to accommodate higher order constructs began.

Briefly, the contributions arising from this project are:

1. The development of BNF2, a new variant of Backus-Naur form. BNF2 is oriented toward the modern practice of two-level syntax for programming languages and is easy for humans to read.
2. Unix/Linux scripts to translate BNF2 into input readable by *lex* and *yacc*.
3. Stable BNF2 rules that extend the TPTP language and accept a variety of higher-order logic expressions in a human-readable language.

Software and documents are at `http://www.cse.ucsc.edu/~avg/TPTPparser`.

2 Specifications with BNF2

The TPTP language was specified in TPTP v3.0.0 using the original standard Backus-Naur form (BNF) [Ne60], with informal explanations to get over some rough spots. In this simple and easy-to-read format, which is found in many programming-language texts, grammar symbols are enclosed in $<$ $>$, and the only meta-symbols are the production symbol "`::=`", the alternative symbol "`|`", and the repetition symbol "`*`"; any other character sequence stands for itself, and is called a *self-declared token*. More sophisticated variants have been proposed over the years; see Section 5.

While trying to write scripts to translate BNF into inputs for *lex* and *yacc* it was realized that standard BNF is ill-suited for specifying *tokens*. That is, the modern two-level style of programming-language specification defines *tokens* using regular expressions, and defines *grammar symbols* using context-free production rules. A lexical analyzer parses the raw input into tokens, while the production rules treat tokens as terminal symbols. This distinction is blurred in standard BNF. Another aspect of the TPTP language that was observed was that some production rules went beyond specifying the *form* of the input, and specified a list of acceptable words. This presented a conflict in that such words became self-declared tokens. Without making a context-sensitive lexical analyzer, such words became unavailable for user identifiers.

To overcome these limitations of standard BNF we designed BNF2, a simple extension of BNF that preserves the easy-to-read format for production rules and adds different formats to specify semantic rules, tokens, and macros for tokens. *Semantic* production rules are ignored for purposes of syntactic parsing, but are available to specify more detail about the semantic content of certain sentential forms. All the extensions are implemented by using additional meta-symbols to specify various rule types, according to the following table. As the table shows, a symbol that has a *semantic* rule must also have a normal grammar rule if it appears on the right side of any normal grammar rule. The string "`<=>`" and following strings are self-declared tokens: grammar symbols must consist of alphanumerics. The right sides of token and macro rules are *lex*-ready regular expressions, except that "`< >`" need to be converted to "`{ }`".

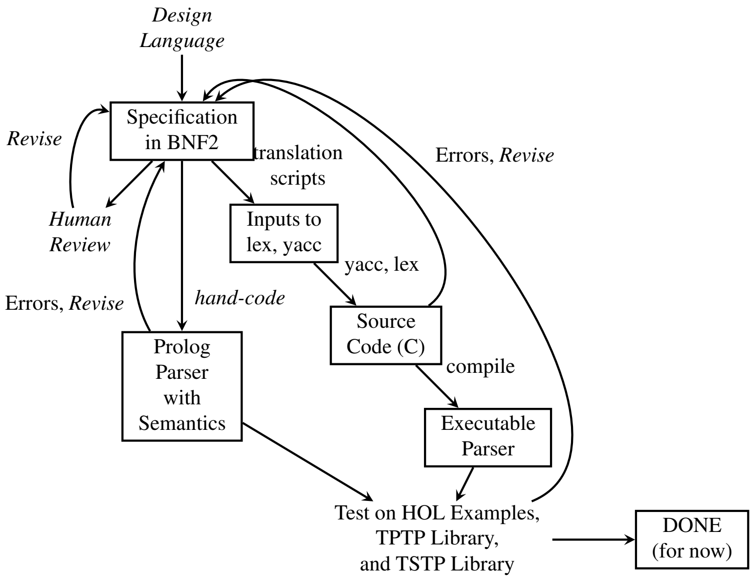

Fig. 1. Overview of the system evaluate a proposed HOTPTP language. Human activities are shown in *italics*.

<table>
<tr><th>Meta-
Symbol</th><th>Rule
Type</th><th colspan="3">Examples (some are simplified from the TPTP language)</th></tr>
<tr><td>::=</td><td>Grammar</td><td><TPTP input></td><td>::=</td><td><annotated formula> | <comment></td></tr>
<tr><td></td><td></td><td><nonassoc op></td><td>::=</td><td><=> | => | <= | <~></td></tr>
<tr><td></td><td></td><td><formula role></td><td>::=</td><td><lower word></td></tr>
<tr><td>:==</td><td>Semantic</td><td><formula role></td><td>:==</td><td>axiom | conjecture | lemma |
theorem | negated_conjecture</td></tr>
<tr><td>::-</td><td>Token</td><td><lower word></td><td>::-</td><td><lower><alphanum>*</td></tr>
<tr><td>::=</td><td>Macro</td><td><lower></td><td>:::</td><td>[a-z]</td></tr>
<tr><td></td><td></td><td><alphanum></td><td>:::</td><td>[A-Za-z0-9_]</td></tr>
</table>

3 System Description

This section describes the system that evaluates a proposed HOTPTP language, based on a BNF2 specification document produced manually. The system includes both manual and automated elements. This is not a system to process an arbitrary BNF2 document; its main purpose is to support TPTP-related development. Figure 1 provides an overview.

The primary automated part of the system generates an executable parser from a BNF2 specification document for HOTPTP, following the right branch of the diagram. This parser is extremely simple, to ensure that the input being checked is really in the language of the specification document. The first step to generate a parser is to translate the (ASCII text) BNF2 specification document, say `hotptp-bnf2.txt`, into a pair of files, `hotptp-1.lex0` and `hotptp-1.y`, which are input files for *lex* (or *flex*) and *yacc* (or *bison*). Unix/Linux scripts accomplish this translation, invoking *sed*, *awk*, *grep*, *sort*, etc. No errors are detected during this step. There is a clear correspondence between grammar rules in `hotptp-bnf2.txt` and `hotptp-1.y`. Tokens have mnemonic names and are easy to locate in `hotptp-1.lex0`.

The analysis and compilation of `hotptp-1.y` by *yacc* or *bison* is a critical step. Grammar errors and ambiguities are often located here after the BNF2 document has passed human inspection. The default library routines are used for all procedures that are expected to be supplied by the programmer. A standard semantic action is attached to each grammar rule, which builds a naive parse tree for each sentence, which may be printed in verbose mode. A syntax error causes "syntax error" on the `stderr` stream and a nonzero exit code; the exit code is zero upon success.

Testing against the full TPTP Problem library requires several minutes. All files with extension "`.p`" should be accepted, whereas the TSTP library contains files that are known to have syntax errors. The most volatile files are the HOL examples, which use the higher order extensions. When a syntax error or unexpected parse tree occurs, analysis is needed to determine if fault lies with the formula or the language specification. Based on available examples, the HOTPTP language has stabilized after about ten iterations of the left branch of the system diagram.

4 The HOTPTP Syntax Proposal

The complete BNF2 document for the proposed HOTPTP syntax is available as `hotptp-bnf2.txt` at the URL given at the end of Section 1. The following example illustrates many of the features added to express higher order constructs. TPTP follows the Prolog convention that variables begin with capital letters, and uses "?" for ∃ and "!" for ∀.

```
hof(1, definition,
    set_union := lambda [A: $type]: lambda [D: ((A-> $o)-> $o), X: A]:
                 ? [S: (A-> $o)]: ( (D @ S) & (S @ X) ) ).
```

The new operator "`:=`" permits a definition at the top level of a "formula" (Hudak uses "≡" [Hud89]). Other new operators are: "`lambda`" for lambda abstraction, "`@`" for application, and "`->`" for type mappings. The colon "`:`" operator has several new meanings, for typing and lambda expressions. Also, "`^`" is a synonym for `lambda` and "`>`" is a synonym for "`->`". We call these new operators the HOF operators.

Logical operators and HOF operators can be mixed in λ/`@` expressions,, subject to using parentheses as needed. Following the general principle in the TPTP language, an *apply expression*, using one or more binary "`@`" operators, must be parenthesized; however, the unary operator λX and its argument need not be. Note that "`@`" is left-associative, "`->`" is right-associative, and "`:`" is right-associative, following usual lambda-calculus conventions. Associativities of existing TPTP operators carry over. The lambda expression shown is: $\lambda A{:}\tau.\ \lambda D{:}((A{\to}o){\to}o).\ \lambda X{:}A.\ \exists S{:}(A{\to}o).\ ((D \mathbin{@} S) \wedge (S \mathbin{@} X))$.

Variables can be typed at the point where they are bound, but not elsewhere. Typing is not required. Builtin base types are `$type` (the set of types), `$i` (the set of individuals), and `$o` (the set of truth values). User-defined base types can be constants or functional terms. Compound types may be built from base types with "`->`". Constants can be typed where they occur in a formula; the syntax is `(c: (int-> int))` or `(c: A)`, etc. Other expressions cannot be typed. For example, `(g(U, V) : (int-> int-> int))` is impossible (but the apply expression `(g: (int-> int-> int)) @ U @ V` is accepted).

Operators other than the HOF operators can be treated as constants by enclosing them in parentheses, as in `(&)` or `(~)` or `(=)` etc. The expression `( (&) @ X @ Y )` is accepted.

A first-order style functional term can appear where a λ/`@` expression is needed, but λ/`@`/$\rightarrow$ expressions cannot appear inside a functional term. That is, `p((S @ X))` is impossible, but `(S @ p(X))` is accepted.

5 Related Work and Acknowledgments

Other variants of BNF have been proposed before BNF2. Extended BNF (EBNF) was designed by a standards committee to have great generality but is quite complicated, with about a dozen meta-symbols, and does not distinguish tokens from grammar symbols. Labeled BNF (LBNF) is designed to generate parsers automatically [FR05], and distinguishes tokens from grammar symbols, but is even more complicated than EBNF.

HOTPTP requirements were culled from Hudak's exposition of lambda calculus [Hud89], and descriptions of *Coq* [FeRL], *LF* [HHP93], and *ELF* [Pfe94]. We thank Chad Brown and Chris Benzmüller for contributing examples of formulas that should be expressible in HOTPTP syntax, based on their work [BB05].

References

[BB05] C. Benzmüller and C. Brown. A Structured Set of Higher-Order Problems. In *Proc. 18th Theorem Proving in Higher Order Logics*, pages 66–81, 2005.

[FeRL] D. Felty *et al.* The Coq Proof Assistant. http://pauillac.inria.fr/coq, URL.

[FR05] M. Forsberg and A. Ranta. The Labelled BNF Grammar Formalism. Technical report, Chalmers, Gothenburg, Sweden, 2005. http://www.cs.chalmers.se/∼markus/BNFC.

[HHP93] R. Harper, F. Honsell, and G. Plotkin. A Framework for Defining Logics. *Journal of the ACM*, 40(1):143–184, 1993.

[Hud89] P. Hudak. Conception, Evolution, and Application of Functional Programming Languages. *ACM Computing Surveys*, 21:359–411, 1989.

[Ne60] P. Naur *et al.* Report on Algorithmic Language ALGOL 60. *Communications of the ACM*, 3:299–314, 1960.

[Pfe94] F. Pfenning. ELF: A meta-language for deductive systems (system description). In *CADE*, 1994.

[SS98] G. Sutcliffe and C.B. Suttner. The TPTP Problem Library: CNF Release v1.2.1. *Journal of Automated Reasoning*, 21(2):177–203, 1998.

[SZS04] G. Sutcliffe, J. Zimmer, and S. Schulz. TSTP Data-Exchange Formats for Automated Theorem Proving Tools. In *Distributed Constraint Problem Solving and Reasoning in Multi-Agent Systems*, pages 201–215. IOS Press, 2004.