BUILDING EXTENSIBLE PARSERS WITH CAMLP4

Asankhaya Sharma
asankhaya@nus.edu.sg

Technical Report
April 2011

BUILDING EXTENSIBLE PARSERS WITH CAMLP4

Abstract – *In this report we describe our experience of using Camlp4 to build an extensible parser for the Hip/Sleek verification system. Camlp4 is a software system for writing extensible parsers for programming languages in OCaml. The Hip/Sleek verification system consists of a core C like language along with a specification language to describe user defined predicates in separation logic. Sleek is an entailment prover for separation logic while Hip is an automated verification engine which can verify heap manipulating programs. The current system consists of two different parsers for Sleek and Hip; using Camlp4 we were able to build a single extensible parser. This enabled reuse of a large portion OCaml code and also provides better system for maintenance of the parser. This parser is then extended to support functions in user defined predicates (Higher Order). A grammar for higher order predicates is introduced which builds on the already existing Sleek/Hip grammar. The syntax extension of grammar closely matches with the parser extension done with Camlp4 to support it. Thus we show that Camlp4 allows us to build parsers that are easy to extend and reuse. We also discuss the issues of test case generation in extensible parsers. In particular we show how to modify Purdom's sentence generation algorithm for grammars, to create test cases which only check for the new productions in grammar. We find that this technique is useful to test parser extensions built using Camlp4.*

1. Introduction

Automated verification of heap-manipulating program a challenging problem, recent work [1] on structured specifications in the Hip/Sleek [4] system has shown that adding new structures to a specification framework for separation logic leads to more precise and better guided verification for heap manipulating programs. In order to add new features and support more expressive structures continuous changes are made to the parser for the Hip/Sleek system. Currently the system has two parsers - one each for Sleek and Hip, these parsers are written in OCaml using Ocamlyacc. Any non-trivial extension to the grammar is hard to implement in the existing system as there is no way to extend already existing grammar productions in Ocamlyacc. There is also the issue of maintenance of both parsers which happen to share quite a lot of common code. In order to resolve these problems and to enable more extensibility in future, we decided to implement a new parser front-end for Hip/Sleek using Camlp4.

Camlp4 is a software system for writing extensible parsers for programming languages [2]. It provides a set of OCaml libraries that are used to define grammars as well as loadable syntax extensions of such grammars. Camlp4 stands for Caml Preprocessor and Pretty-Printer and one of its most important applications is the definition of domain-specific extensions of the syntax of OCaml. It can also be used to build an extensible parser for new domain-specific languages. When compared to Ocamlyacc, Camlp4 also provides better error messages for parsing related errors. However, one important difference between Ocamlyacc and Camlp4 is that while the former works LALR grammars in a bottom up manner, the latter is a recursive decent top down parser which works best for LL(0) grammars. This meant that the existing grammar for Hip/Sleek had to be modified in order to parse it with Camlp4.

The new parser provides better reuse (between Sleek and Hip grammar), clearer error messages (for syntax errors in incorrectly entered programs) and ease of extensibility (to higher order predicates

for specification). The rest of the report is structured as follows in section 2 we discuss the experience of implementing the new parser using Camlp4, in section 3 we show how to extend the parser for higher order predicates by grammar extension, in section 4 we introduce an algorithm for test case generation in extensible parsers and finally we conclude in section 5.

2. Implementation

We present the description of our implementation by means of illustrative examples from the Hip/Sleek grammar and corresponding Camlp4 features. Our attempt here is not to provide an exhaustive list of features but to cover all interesting cases and pitfalls that one may encounter while writing a parser with Camlp4.

2.1 Stream Parsers in Camlp4

OCaml comprises a library type for streams (possibly infinite sequences of elements, that are evaluated on demand), and associated stream expressions, to build streams, and stream patterns, to de-structure streams [6]. Streams and stream patterns provide a natural approach to the writing of recursive-descent parsers. The Camlp4 system takes the input as a stream and then just runs through a nested pattern match. It seeks the next element from the stream and matches it with the defined grammar symbols (advancing the stream to next element). If there's no match on the first token, that's a Stream.Failure (the stream is not advanced, giving us the opportunity to try another branch); but once we have matched the first token, a subsequent match failure is a Stream.Error (we have committed to a branch, and advanced the stream; if the parse fails now we can't try another branch) [7]. As an example, consider the following grammar rule for disjunctive constraints, in this case if let's say the 2nd rule is been matched and after the `OPAREN token if a subsequent match is a failure then the parsing fails and we cannot try the branch at 3rd rule. Also the parser ignores tokens after a successful parse (unless we use the EOI token to mark end of input) so in this case it will terminate successfully after `OPAREN without any error if the match fails in rule 2.

```
disjunctive_constr:
[[
  dc=disjunctive_constr; `ORWORD; oc=disjunctive_constr ->
  F.mkOr dc oc (get_pos 2)] (*1*)

| `OPAREN; dc=disjunctive_constr; `CPAREN -> dc (*2*)

| `OPAREN; cc=core_constr; `CPAREN -> cc (*3*)
]];
```

Thus we see that we can't give arbitrary BNF-like grammars to Camlp4. Instead, the Camlp4's grammars, lets us specify a recursive-descent parser using a BNF-like syntax.

2.2 Left Factoring

As a consequence of how stream parsers works in Camlp4 (Section 2.1) we need to make the grammar left factored. The rules for a grammar are left-factored: when there is a common prefix of symbols among different rules, the Camlp4 stream parser doesn't choose which rule to use until the common prefix has been parsed [7]. The symbol which distinguishes the two rules can be thought of as the branching point for the parse tree. It is illustrated by the following rule for the heap constraint.

```
heap_constr:
[[
   `OPAREN; hrd=heap_rd; `CPAREN; `SEMICOLON; hrw=heap_rw ->
    F.mkPhase hrd hrw (get_pos 2)

| `OPAREN; hrd=heap_rd; `CPAREN -> F.mkPhase hrd F.HTrue (get_pos 2)

| hrw = heap_rw -> F.mkPhase F.HTrue hrw (get_pos 2)
]];
```

2.3 Local Backtracking and Look Ahead

From the description so far it may seem that backtracking is not possible with a Camlp4 parser. It does however support a limited form of backtracking - when the parser is at a branch point (e.g. a choice between two entries), and when the called entry does not itself commit and advance the stream (in which case Stream.Error is raised on a parse error instead of Stream.Failure) [7]. The following grammar rule for local variable declaration shows how local backtracking can work.

```
local_variable_declaration:
[[
  t1=local_variable_type; t2=variable_declarators ->  mkVarDecl t1 t2 (get_pos 1)
]];

variable_declarator:
[[
   `IDENTIFIER id; `EQ; t = variable_initializer -> (id, Some t, get_pos 1)

| `IDENTIFIER id -> (id, None, get_pos 1)
]];
```

Since `IDENTIFIER does not advance the stream we can match both the rules of variable_declarator in the production rule for local_variable_declaration. In addition, Camlp4 allows for explicit look ahead of the stream of tokens. We can use local backtracking with look ahead to resolve ambiguities in the rules. Consider the grammar rule to parse the disjunctive constraint again, here we use peek_dc token to look ahead and use rule 2 only if there is an `EXISTS in the stream or else the branch fails and the `OPAREN will be then checked with rule 3. The explicit look ahead doesn't advance the stream and only peeks at the next token.

```
let peek_dc =
   SHGram.Entry.of_parser "peek_dc"
       (fun strm ->
           match Stream.npeek 2 strm with
             | [OPAREN,_;EXISTS,_] -> ()
             | _ -> raise Stream.Failure)

disjunctive_constr:
[[
  peek_dc; `OPAREN;  dc = disjunctive_constr; `CPAREN -> dc] (* 1 *)

| `EXISTS; ocl= cid_list; `COLON; cc = core_constr -> (match cc with (* 2 *)
      | F.Base ({F.formula_base_heap = h;
                 F.formula_base_pure = p;
                 F.formula_base_flow = fl;
                 F.formula_base_branches = b}) -> F.mkExists ocl h p fl b (get_pos 1)
      | _ -> report_error (get_pos 4) ("only Base is expected here."))

| cc = core_constr -> cc (* 3 *)
]];
```

By having a left factored grammar and using backtracking with explicit look ahead it is possible to resolve most of the ambiguities in choice of branch points such that the Camlp4 can parse the

grammar (even with a recursive decent top down parser). In the next few sub sections we discuss some of the syntactical features of Camlp4 that aid in writing parsers.

2.4 Self-Calls

The entry being defined in Camlp4 can refer to itself using the SELF symbol; it gives calls to the entry being defined ("self-calls") special treatment. The rules of an entry actually generate two parsers, the "start" and "continue" parsers [7]. When a self-call appears as the first symbol of a rule, the rest of the rule goes into the continue parser; otherwise the whole rule goes into the start parser. An entry is parsed starting with the start parser; a successful parse is followed by the continue parser. As an example consider the rules for parsing arithmetic expressions in cexp_w, when the parsing starts, the last rule is used to begin with and then first four rules are used (based on the arithmetic expression) on the continue parser and finally the whole rule goes into the start parser.

```
cexp_w:
[[
  c1 = SELF; `PLUS; c2 = SELF ->
  apply_cexp_form2 (fun c1 c2-> P.mkAdd c1 c2 (get_pos 2)) c1 c2

| c1 = SELF; `MINUS; c2 = SELF ->
  apply_cexp_form2 (fun c1 c2-> P.mkSubtract c1 c2 (get_pos 2)) c1 c2

| t1 = SELF; `STAR; t2 = SELF ->
  apply_cexp_form2 (fun c1 c2-> P.mkMult c1 c2 (get_pos 2)) t1 t2

| t1 = SELF; `DIV; t2 = SELF ->
  apply_cexp_form2 (fun c1 c2-> P.mkDiv c1 c2 (get_pos 2)) t1 t2

| t = cid -> Pure_c (P.Var (t, get_pos 1)) (* Last Rule *)
]];
```

2.5 Levels and Precedence

In a Camlp4 grammar the entries are lists of levels, in single square brackets, and each level consists of a list of rules, also in single square brackets. The idea with levels is that parsing begins at the topmost level; if no rule applies in the current level, then the next level down is tried. Furthermore, when making a self-call, call is made at the current level (or the following level) rather than at the top [7].This gives a way to implement operator precedence: order the operators top to bottom from loosest to tightest binding. The associativity of a level can be specified by prefixing it with the keywords NONA, LEFTA, or RIGHTA. Hence the arithmetic expression example above should be better written as follows.

```
cexp_w:
[[
  c1 = SELF; `PLUS; c2 = SELF ->
  apply_cexp_form2 (fun c1 c2-> P.mkAdd c1 c2 (get_pos 2)) c1 c2

| c1 = SELF; `MINUS; c2 = SELF ->
  apply_cexp_form2 (fun c1 c2-> P.mkSubtract c1 c2 (get_pos 2)) c1 c2

]|[
  t1 = SELF; `STAR; t2 = SELF ->
  apply_cexp_form2 (fun c1 c2-> P.mkMult c1 c2 (get_pos 2)) t1 t2

| t1 = SELF; `DIV; t2 = SELF ->
  apply_cexp_form2 (fun c1 c2-> P.mkDiv c1 c2 (get_pos 2)) t1 t2

]|[
  t = cid -> Pure_c (P.Var (t, get_pos 1)) (* Last Rule *)
]];
```

2.6 Special Symbols

As we saw in section 2.4, SELF is a special symbol to refer to the entry being defined; similarly there are many other special symbols in Camlp4. NEXT refers to the entry being defined, at the following level regardless of associativity or position. A list of zero or more items can be parsed with the syntax LIST0 elem, where elem can be any other symbol. The return value has type 'a list when elem has type 'a [7]. To parse separators between the elements we can use LIST0 elem SEP. An optional item can be parsed with OPT elem; the return value has type 'a option [7]. The following example shows how to parse a list of zero or more expressions separated by a comma

```
opt_cexp_list:
[[
  t = LIST0 cexp SEP `COMMA -> t
]];
```

2.7 Error Messages

The error printing facility in Camlp4 is much better than Ocamlyacc; the error message during a parse failure describes the exact token which was expected and the point where the parser is after parsing the input. The following shows the errors printed by the current and new version of Hip/Sleek parser for the same file.

Error Message Current Parser (Ocamlyacc) -

```
asankhs@loris-7:~/hg-sleekex$ ../hg/sleekex/hip ex.ss
Processing file "ex.ss"
Parsing...
ex.ss:7:14: parse error in constraints disjunctive
Exception occurred: Failure("Error detected")
Error(s) detected at main
```

Error Message New Parser (Camlp4) -

```
asankhs@loris-7:~/hg-sleekex$ ./hip ex.ss
Processing file "ex.ss"
Parsing...
File "ex.ss", line 7, characters 13-14
 --error: Stream.Error("[cexp] expected after COMMA (in [opt_heap_arg_list])")
```

With this we conclude our discussion on implementing the Hip/Sleek parser in Camlp4, in the next section we introduce a grammar for higher order predicates and show how it is possible to extend the parser to handle the new grammar. Camlp4 makes it easy to extend the new parser to support higher order predicates.

3. Extending the Parser

The specification language in the current Hip/Sleek verification system can be extended to include functions in user defined predicates. This will allow for a composition of various predicates to express shape properties of data structures. In order to support this, the Camlp4 based parser is extended to include the grammar of higher order predicates. The complete set of grammar rules are listed in the appendix, here we only describe how it is possible to do syntax extension using Camlp4.

Any of the grammar entries already defined in the parser can be extended. Also we can extend any existing level and insert new levels. The syntax of an entry extension is as follows.

```
entry-extension ::= optional-position

identifier : [ list-of-levels-separated-by-bars ] ;

position ::= FIRST | LAST | BEFORE label | AFTER label | LEVEL label
```

In order to refer to level in the production of an entry it is important to label it. As an example we can extend the heap constraint with functions to create higher order predicates.

```
EXTEND

Simple_heap_constr: AFTER "heap_formula"

[ “higher order predicate”

[ t = ho_fct_header ->
  F.mkHeapNode ("",Primed) "" false false false false [] None  (get_pos 1)

]];

END;;
```

The rest of the rules of the higher order predicate grammar can be similarly added to the parser to extend it. The Camlp4 syntax extension mechanism greatly increases code reuse and provides an easy way to develop and maintain a parser. In the next section we discuss some of issues of testing an extensible parser.

4. Testing Extensible Parsers

The new Camlp4 parser was evaluated by testing against a benchmark of test suite for the Hip/Sleek system. But while making extensions to the parser how can we ensure that the new rules are tried and tested properly. Similar to various forms of testing coverage criteria, (statement, block, condition etc.) for a parser we may be interested to know how many of the rules does a given test suite exercise. Based on this one can define a production coverage criteria [3] as the percentage of grammar productions used by a given set of test cases. To achieve production coverage we are interested in generating a minimum set of input test cases which will use all the production rules of the underlying grammar.

Purdom’s algorithm [5] provides a way for generating the shortest sentences from a context free grammar which cover all the productions. Figure 1, describes the steps of the algorithm in detail, the output of the algorithm is a set of strings which cover all productions in the grammar. This algorithm can be used to test top down parsers by generating inputs to the parser using the sentences generated based on the underlying grammar [8].

Phase I (Shortest String Length)	
SLEN	Shortest terminal string length for each symbol
RLEN	The length of shortest string derivable from each production rule
SHORT	For every non-terminal, it contains the rule numbers which leads to shortest string derivation.
Phase II (Shortest Derivation Length)	
DLEN	For each non-terminal, the length of shortest terminal string which used it in its derivation
PREV	Rule number which introduces a non-terminal in shortest string derivation

Phase III (Generate Sentence)	
ONST	The number of occurrences of each non-terminal on the stack.
ONCE	Information about each non-terminal whether to use it or not and how to rewrite it. Contains anyone of the following values: 1. READY 2. UNSURE 3. FINISHED 4. INTEGER
MARK	Information whether a rule has been used or not. Contains boolean values.
STACK	Every symbol is pushed on the stack starting with the starting symbol "S".A terminal is being popped every time it comes on the top and non-terminal is rewritten by an appropriate rule.

Figure 1: Purdom's Algorithm

In our case, we already had a test suite to validate the new Camlp4 parser with, but for the higher order predicates extension we still need to test the parser with new inputs. Ideally when doing extensions using Camlp4 we would want to generate test cases which will exercise the new grammar rules and productions. Since the extensions are done on an already existing parser which is either tested or has a regression suite. Now we present a modified Purdom's algorithm that will generate the sentences corresponding to only the new grammar rules. The key idea here is to just generate the shortest terminal string for each symbol of the original grammar and use it while generating sentences for the extended grammar. The old symbols in the extended grammar are just treated as fixed constant sized strings of their shortest terminal string. This will ensure that the new sentences that are generated are also of shortest length. The following describes the modified algorithm for generating test cases in extended grammar.

Modified Purdom's Algorithm for Extended Grammar

```
For the Original Grammar:
let SLEN_O be the list of shortest terminal string for each symbol;
let RELN_O be the list of shortest string derivable from each production rule;
let SHORT_O be the list of rule numbers which leads to the shortest string
derivation;
```

```
For the Extended Grammar:
let SLEN_E be the list of shortest terminal string for each symbol;
let RELN_E be the list of shortest string derivable from each production rule;
let SHORT_E be the list of rule numbers which leads to the shortest string
derivation;

For all the symbols in the Original grammar
- do Phase 1 of Purdom's Original Algorithm
- SLEN_E = SLEN_O;
- RLEN_E = RLEN_O;
- SHORT_E = SHORT_O;

For all the new symbols in the Extended grammar
- do Phase 1,2 and 3 of the Purdom's Original Algorithm
```

Applying this algorithm on the grammar extension for the higher order predicates we get the following set of test cases which provide full production coverage of the new set of grammar rules. Table 1 lists the sentences generated by the algorithm and the corresponding test cases for the parser. Note that as per the modification in the Purdom's algorithm the symbols from the original grammar are replaced by their shortest terminal string (e.g. ID with node). Our parser is able to parse all the test cases in Table 1, thus ensuring that each production of the grammar is tested at least once.

Sentence Generated by Modified Purdom Algorithm	**Test Case for the Extensible Parser**
HPRED ID OSQUARE ID CSQUARE LT ID GT OSQUARE ID CSQUARE EQEQ TRUE SEMICOLON	ho_pred node [node] <node> [node] == true ;
HPRED ID OSQUARE ID OSQUARE ID CSQUARE CSQUARE LT ID GT OSQUARE ID CSQUARE EQEQ TRUE SEMICOLON	ho_pred node [node[node]] <node> [node] == true ;
HPRED ID OSQUARE SET OSQUARE ID CSQUARE CSQUARE LT ID GT OSQUARE ID CSQUARE EQEQ TRUE SEMICOLON	ho_pred node [set[node]] <node> [node] == true ;
HPRED ID OSQUARE ID CSQUARE LT ID GT OSQUARE ID COLON ID CSQUARE EQEQ TRUE SEMICOLON	ho_pred node [node] <node> [node:node] == true ;
HPRED ID OSQUARE ID CSQUARE LT ID GT OSQUARE ID OSQUARE ID CSQUARE CSQUARE EQEQ TRUE SEMICOLON	ho_pred node [node] <node> [node[node]] == true ;
HPRED ID OSQUARE ID CSQUARE LT ID GT OSQUARE SET OSQUARE ID CSQUARE CSQUARE EQEQ TRUE SEMICOLON	ho_pred node [node] <node> [set[node]] == true ;
HPRED ID OSQUARE ID CSQUARE LT ID GT OSQUARE ID OSQUARE ID CSQUARE COLON ID CSQUARE EQEQ TRUE SEMICOLON	ho_pred node [node] <node> [node[node]:node] == true ;
HPRED ID OSQUARE ID CSQUARE LT ID GT OSQUARE SET OSQUARE ID CSQUARE COLON ID CSQUARE EQEQ TRUE SEMICOLON	ho_pred node [node] <node> [set[node]:node] == true ;
HPRED ID LT GT EXTENDS ID LT GT WITH OBRACE ID OPRAEN ID CPAREN EQ TRUE CBRACE	ho_pred node <> extends node <> with { node(node) = true}

HPRED ID LT GT REFINES ID LT GT WITH OBRACE ID OPRAEN ID CPAREN EQ TRUE CBRACE	ho_pred node <> refines node <> with { node(node) = true}
HPRED ID LT GT JOIN ID LT GT	ho_pred node <> join node <>
HPRED ID LT GT JOIN ID LT GT SPLIT ID LT GT	ho_pred node <> join node <> split node <>
HPRED ID LT GT JOIN ID LT GT COMBINE ID LT GT	ho_pred node <> join node <> combine node <>

Table 1: Test Case Generation for Extensible Parser

5. Conclusions

The task of building and maintain parsers is made easy by the use of Camlp4. Not only does it provide a clean and systematic way to do syntax extensions but also has better error messages for parse errors. Using Camlp4 we were about to build a new parser for Hip/Sleek verification system, this new parser was then extended for higher order predicates. The extended parser was tested for production coverage by using modified Purdom's algorithm. Our experience of building, extending and testing a new parser shows that Camlp4 is really a powerful system for creating parsers.

6. References

[1] C. Gherghina, C. David, S. Qin, W.-N. Chin. Structured Specifications for Better Verification of Heap-Manipulating Programs. In FM 2011

[2] Camlp4 Wikipedia Entry. http://en.wikipedia.org/wiki/Camlp4. Retrieved on April 3, 2011.

[3] J. Riehl. Grammar Based Unit Testing for Parsers. In Master's Thesis, Department of Computer Science, University of Chicago.

[4] H.H. Nguyen, C. David, S. Qin, and W.-N. Chin. Automated verification of shape, size properties via separation logic. In 8th VMCAI, 2007.

[5] P. Purdom. A sentence generator for testing parsers. In BIT 12(3) (1972) 366–375.

[6] Camlp4 Manual. http://caml.inria.fr/pub/docs/manual-camlp4. Retrieved on April 17, 2011.

[7] http://ambassadortothecomputers.blogspot.com/2010/05/reading-camlp4-part-6-parsing.html. Retrieved on April 17, 2011.

[8] A.M. Paracha and F. Franek. Testing Grammars for Top-Down parsers. In CIS2E 2008.

7. Appendix

Grammar Extension for Higher Order Predicates

```
hopred_decl ->
    `HPRED hpred_header `EXTENDS ext_form
  | `HPRED hpred_header `REFINES ext_form
  | `HPRED hpred_header `JOIN split_combine
  | `HPRED hpred_header `EQEQ shape opt_inv `SEMICOLON

shape -> formulas*

split_combine ->
    hpred_header
  | hpred_header `SPLIT split_combine
  | hpred_header `COMBINE split_combine
```

ext_form -> hpred_header `WITH `OBRACE ho_fct_def_list `CBRACE

ho_fct_header -> `IDENTIFIER `OPAREN fct_arg_list `CPAREN

ho_fct_def -> ho_fct_header `EQ shape

ho_fct_def_list -> LIST1 ho_fct_def

hpred_header -> `IDENTIFIER opt_type_var_list `LT opt_typed_arg_list `GT opt_fct_list

typed_arg ->
    typ*
  | typ* `COLON typed_arg
  | typ* `OSQUARE typ* `CSQUARE
  | typ* `OSQUARE typ* `CSQUARE `COLON typed_arg
  | `SET `OSQUARE typ* `CSQUARE
  | `SET `OSQUARE typ* `CSQUARE `COLON typed_arg

opt_typed_arg_list -> LIST0 typed_arg SEP `COMMA

type_var ->
    typ*
  | typ* `OSQUARE typ* `CSQUARE
  | `SET `OSQUARE typ* `CSQUARE

type_var_list -> `OSQUARE LIST1 type_var SEP `COMMA `CSQUARE

opt_type_var_list -> OPT type_var_list

fct_arg_list -> LIST1 cid* SEP `COMMA

fct_list -> `OSQUARE fct_arg_list `CSQUARE

opt_fct_list -> OPT fct_list

Tokens start with ` and existing grammar symbols are marked with *.OPT, SEP and LIST are default Camlp4 symbols, they stand for Optional Symbol, Symbol Separator and a List of Symbols respectively.