POET Tutorial: Building A Small Compiler Project

Qing Yi
University of Texas At San Antonio
(qingyi@cs.utsa.edu)

December 2, 2010

NOTE: the example code can be found in POET/examples directory and are contained in files Parse.pt, BNFexp.code, and CFG.pt

1 Setting Up The Translation

A compiler is essentially a translator that reads in some some code in an input language and output the result in another language. The following illustrates the structure of a compiler in POET that translates between two languages.

```
include utils.incl

<********************* A compiler/translator project ******************>
<parameter inputFile default="" message="input file name"/>
<parameter outputFile  default="" message="output file name"/>

<* declares the start non-terminal of your grammar *>
<code Goal/>

<* read the inputFile, parse it using grammar syntax defined in file
   "BNFexp.code", and save the parsing result (the abstract syntax tree of
   the input code) into variable inputCode *>
<input from=(inputFile) syntax="BNFexp.code" parse=Goal to=inputCode/>

<*  define functions/routines to accomplish your task *>
<xform TypeCheckExp pars=(symTable, exp)>
......
</xform>

<eval table = MAP(STRING,IntType|FloatType|ArrayType);
           ....../>

<* perform whatever additional operations here *>
<eval TypeCheckExp(table, inputCode);/>
```

```
<* output your result to outputFile *>
<output to=outputFile syntax="BNFexp.code" from=table/>
```

The first line is an include command, which reads the *utils.incl* file (located in POET/lib directory) before reading any instructions in the current file. **NOTE: The include commands must be at the start of a POET program to be recognized. If appearing in middle of a program, they will be ignored (treated as source code of an input language).**

The following two lines are *parameter* declarations that define the values of $inputFile$ and $outputFile$. Specifically, each *parameter* keyword declares a global variable whose value can be modified through command-line options. The command-line to invoke the above translator therefore is the following (assuming the current directory is $POET/examples$.

```
> ../src/pcg [-pinputFile=<myInputFile>] [-poutputFile=<myOutputFile>]  Parse.pt
```

Here each command-line option $-p < var >=< value >$ specifies a new value for a global parameter <var>. The command-line options are optional — when no value is given for a global parameter, the parameter contains its default value (declared using the $default$ keyword within the *parameter* declaration).

Following the *parameter* declarations is a forward declaration that declares *Goal* as the name of a code template. This code template name should be defined in the file that contains grammar syntax definitions for the input file, the *BNFexp.code* file in this case.

NOTE: Except for the *include* **commands at the start of the program, all the commands in a POET program are first read in and saved in an internal representation before being evaluated. In particular, the** *input* **commands in the middle of a program are not evaluated until all the other commands and declarations have been read. Because of this, although the name** *Goal* **is defined in** *BNFexp.code*, **it needs to be declared in** $Parse.pt$ **as a code template name to avoid being treated as a regular global variable.**

Following the forward declaration for *Goal* is an input command. Specifically, the *input* command opens the input file (name specified by the $inputFile$ variable), reads in the input code (the content of the input file), and saves the input code as value of the global variable $inputCode$. The parsing of input code is based on the grammar definition contained in file *BNFexp.code* (specified by the *syntax* attribute). Additionally, the code template *Goal* will be used as the start non-terminal (the top-level variable) of the grammar to parse the input code (specified by the $parse$ attribute).

After the input command, a *xform* declaration defines a function/routine named $TypeCheckExp$. The *xform* declarations may be used to accomplish whatever task your compiler may need to accomplish. They may use all the capabilities that POET provides (e.g., pattern matching, AST traversal, and program transformations) to analyze or transform the input code.

NOTE: By default, all names in POET are assumed to be regular variable names unless they have already appeared as the name of a code template or xform routine declaration. For more details, see the language manual.

Following the $xform$ routine definitions, two *eval* commands are used to set up a symbol table (named *table*) and invoke the $TypeCheckExp$ routine with *table* and $inputCode$ as actual parameters.

NOTE: All POET expressions must be embedded within an eval command to be evaluated at the global scope.

After reading the input code and applying necessary analysis and transformations, the last *output* command writes the compilation result (in this case, the symbol table) to an external file whose name specified by the *outputFile* variable. In particular, the result is unparsed based on the grammar definition contained in file *BNFexp.code* (enforced by the *syntax* specification). If there is no syntax specification, the result will be output as a sequence of integer/string tokens.

NOTE: when the *inputFile* variable contains an empty string, the input code will be read from the standard input (the user will be prompted to type in the input code); when the *outputFile* variable contains an empty string, the input code will be written to the standard output (the screen).

NOTE: as illustrated by the *MyAST* variable, global variables in POET can be used without being declared. Command-line parameters are also considered as global variables.

2 Writing Grammar Specifications For Parsing

POET supports automatic parsing of arbitrary languages as long as the concrete syntax of the language is defined in a separate syntax file. The specification of syntax is through code templates in POET. There is a direct correlation between specifying the syntax using BNF and using POET. In particular, suppose we have the following grammar specification via BNF.

```
Goal ::= Exp
Exp ::= Exp + Factor | Factor
Factor ::= id | intval
```

After eliminating left-recursion (POET parses the input code in a top-down recursive descent fashion and therefore left-recursion in the grammar needs to be eliminated), we obtain the following grammar.

```
Goal ::= Exp
Exp ::=  Factor Exp'
Exp' ::= + Factor Exp' | epsilon
Factor ::= id | intval
```

The above grammar can be expressed using the following POET code template definitions.

```
<code Exp/>
<code Exp1/>
<code Exp2/>
<code Factor />

<code Goal pars=(content : Exp) >
@content@
</code>

<code Exp pars=(opd1 : Factor, rest : Exp1 )>
```

```
@opd1@ @rest@
</code>

<code Exp1 pars=(content : Exp2 | "") >
@content@
</code>

<code Exp2 pars=(opd2: Factor, rest : Exp1)>
+ @opd2@ @rest@
</code>

<code Factor pars=(content : Name | INT)>
@content@
</code>
```

Here an extra code template $Exp2$ is used to specify the first BNF production for Exp1 (i.e., $Exp' ::= +FactorExp'$), where Exp1 corresponds to non-terminal Exp' in the BNF specification. Specifically, the **name** of each code template corresponds to the left-hand non-terminal of each BNF production, and the **body** of each code template corresponds to the right-hand side of each BNF production. If any symbol in the right-hand side of a BNF production is itself a non-terminal or has a value that needs to be saved, a variable is created as parameter of the code template, and the variable (surrounded by a pair of @'s) is used inside the code template body instead. For example, the following code template definition corresponds to the BNF production $Exp ::= FactorExp'$.

```
<code Exp pars=(opd1 : Factor, rest : Exp1 )>
@opd1@ @rest@
</code>
```

Here two variables, $opd1$ and $rest$ are created for the non-terminals $Factor$ and Exp' in the original production respectively.

POET code templates are not only used to define the grammar of languages, they are also used to define the internal representation of the input code using an abstract syntax tree (AST). Suppose the following POET program is used to parse an arbitrary input code (see Section 1).

```
include utils.incl

<parameter inputFile message="input file name"/>
<parameter outputFile  message="output file name"/>

<code Goal/>

<input from=(inputFile) syntax="BNFexp.code" parse=Goal to=inputCode/>

<eval PRINT(inputCode)/>
```

Here $PRINT(inputCode)$ prints out the AST constructed from parsing the input code. If the POET code templates above is used to define grammars in the *inputlang* file, and if an expression $a + b + 5$ is used as the input code, the following AST will be printed out.

```
Goal#Exp#(
    Factor#"a",
    Exp1#Exp2#(
        Factor#"b",
        Exp1#Exp2#(
            Factor#3,
            Exp1#"")))
```

3 Rebuilding AST

Unless specified otherwise, POET automatically builds an AST node (code template object) for each code template definition used in the parsing process. To build a more elegant AST, you can reconfigure the parsing process by defining a *rebuild* attribute for some code templates. For example, the context-free grammar definitions in section 2 need to be extended with the following attribute grammar to build an AST.

```
Goal ::= Exp       { Goal.ast = Exp.ast; }
Exp ::=  Factor  {Exp'.inh = Factor.ast; } Exp'  { Exp.ast = Exp'.ast; }
Exp' ::= + Factor { Exp'1.inh = BinaryOperator("+", Exp'.inh, Factor.ast); } Exp'1
        {  Exp'.ast=Exp'1.ast;}
        | epsilon { Exp'.ast = Exp'.inh; }
Factor ::= id {Factor.ast = Variable(id.name); }
        | intval   { Factor.ast = ICONST(intval.val); }
```

The above attribute grammar can be translated to the following POET implementations.

```
include utils.incl

<code Exp/>
<code Exp1/>
<code Exp2/>
<code Factor />
<xform BuildExp1 pars=(inh, content)   />

<code Goal pars=(content : Exp) rebuild=content >
@content@
</code>

<code Exp pars=(opd1 : Factor, rest : Exp1 ) rebuild=(BuildExp1(opd1,rest))>
@opd1@ @rest@
</code>

<code Exp1 pars=(content : Exp2 | "") rebuild=content >
@content@
</code>

<code Exp2 pars=(opd2: Factor, rest : Exp1) >
+ @opd2@ @rest@
```

```
</code>

<code Factor pars=(content : Name | INT) rebuild=content>
@content@
</code>

<code Bop pars=(op : STRING, opd1 : Exp, opd2 : Exp)>
(opd1 op opd2)
</code>

<xform BuildExp1 pars=(inh: Exp, content : Exp1)>
switch (content) {
  case Exp2#(opd2, rest) : BuildExp1(Bop#("+", inh, opd2), rest)
  case "" : inh
 }
</xform>
```

Each *rebuild* attribute of a code template holds an expression that defines what AST node to return as the result of parsing the given code template. After POET successfully parses the input program using each code template, it evaluates the corresponding *rebuild* attribute using all the known parameters and attributes of the code template. For example, in the above POET code, the *BuildExp*1 routine is invoked in the *rebuild* attribute of code template *Exp* whenever the *Exp* template has been used to successfully parse a fragment of the input code.

Because the POET built-in code template parser provides limited support for inherited attributes (you can only define a single INHERIT attribute, which holds the previous code template object built from input, for each code template), it is more convenient to convert inherited attributes into synthesized attribute evaluation. In particular, you can save all the synthesized attributes relevant for building an AST until both the synthesized and inherited attributes can be collectively used. For example, the *BuildExp*1 routine is invoked to the rebuild function for *Exp* instead of *Exp*1 because the inherited attribute for *Exp*1 is not available until the parsing of *Exp* has succeeded.

4 Building A Symbol Table

POET provides a compound data type, MAP, to support the need of associative tables that map a collection of values to their attributes. Each symbol table is a map that associate type information with variable names. Suppose the following code templates define the data structure for storing type information.

```
<code IntType/>
<code FloatType />
<code ArrayType pars=(elemType, arraySize) />
```

The following POET statement constructs an empty symbol table and stores it into a local variable *table*.

```
table = MAP(STRING,IntType|FloatType|ArrayType);
```

The following statements check whether a variable named x is already in in the symbol table; if not, it inserts x into the symbol table as a $IntType$ variable. This can be used as an action for type checking when seeing type declarations such as "int x,y,z;".

```
if (table[x] != "")
     ERROR( "variable multiply declared: " x);
else table[x] = IntType;
```

The following statements insert three integer variables, "a", "b", and "c", into the symbol table.

```
foreach ( ("a","b","c") : (x=STRING) : TRUE) {
  if (table[x] != "")
     ERROR( "variable multiply declared: " x);
  else table[x] = IntType;
}
```

To find out more about the use of the $foreach$ loop and associative tables, see POET manual.

If nested scopes are allowed, a list of symbol tables can be used to represent the context information of each block. The following POET routine can be invoked to determine the type information of a variable in a nested block.

```
<xform LookupVariable pars=(symTableList, varName)>
if (symTableList == "")
    ERROR("Variable not found: " varName);
curTable = HEAD(symTableList);
res = curTable[varName];
if (res != "") RETURN res;
LookupVariable(TAIL(symTableList), varName)
</xform>
```

5 Type Checking

To apply type checking to an input program, the AST representation of the program must be traversed, and a type must be determined for each expression within the program. For example, suppose we have the following translation scheme for type checking.

```
Exp ::= {Exp1.symTable=Exp.symTable;} Exp1 + {Exp2.symTable=Exp.symTable; } Exp2
    { if (Exp1.type == INT && Exp2.type == INT) Exp.type = INT;
       else if (Exp1.type == FLOAT && Exp2.type==FLOAT) Exp.type=FLOAT;
       else ERROR; }
  | id { Exp.type = lookup_type(Exp.symTable,id); }
  | intval { Exp.type = INT; }
```

It can be implemented using the following POET function.

```
<xform TypeCheckExp pars=(symTable, exp)>
  switch(exp)
  {
    case Bop#("+", exp1, exp2):
```

```
          type1 = TypeCheckExp(symTable, exp1);
          type2 = TypeCheckExp(symTable, exp2);
          if  (type1 : IntType && type2 : IntType)  returnType=IntType;
          else if (type1 : FloatType && type2 : FloatType) returnType=FloatType;
          else ERROR("Type checking error: " exp);
          returnType
     case STRING:  symTable[exp]
     case INT : IntType
  }
</xform>
```

The type information of each expression may be additionally saved in the symbol table of each scope, so that the information can be used for later code generation and optimizations. This is implemented by the following POET routine.

```
<xform TypeCheckExp pars=(symTable, exp)>
  switch(exp)
  {
    case Bop#("+", exp1, exp2):
          type1 = TypeCheckExp(symTable, exp1);
          type2 = TypeCheckExp(symTable, exp2);
          if  (type1 : IntType && type2 : IntType)  returnType=IntType;
          else if (type1 : FloatType && type2 : FloatType) returnType=FloatType;
          else ERROR("Type checking error: " exp);
          symTable[exp] = returnType;   <<* saving the type of exp in symbol table
          returnType
     case STRING:  (symTable[exp])
     case INT : IntType
  }
</xform>
```

Note that in POET, the last expression evaluated is the value returned by a xform routine. For details on how to define *xform* routines and how to use the pattern pattern capabilities in POET, see the language manual.

6 Building A Control Flow Graph

There are two steps involved in building a control flow graph: identifying basic blocks, and using the basic blocks as nodes to build an arbitrary graph. Because each basic block is a sequence of statements or expressions, you can implement each basic block using a list of AST nodes. Suppose you have already identified a set of statements/expressions that should be included as components of basic blocks and a subset of these ASTs that serve as the starting points of different basic blocks. The following POET function can be invoked to build a list of all the basic blocks (Need poet.1.02.04. see POET/examples/Anal.pt).

```
include Loops.incl

<code BasicBlock pars=(label, stmts)>
```

```
@label@[@stmts@]
</code>

<xform BuildBasicBlocks pars=(input)>
   blocks = MAP{};
   foreach (input : (cur=Nest#(CLEAR loop, CLEAR body)) : FALSE)
     { blocks[cur] = blocks[body]=1; }
   start=""; curBlock = ""; count=0;
   foreach (input :  (cur=_) : FALSE) {
     if (blocks[cur] == 1) {
        if (curBlock != "") {
           blocks[start]=BasicBlock#(count, ReverseList(curBlock));
           count = count + 1; curBlock = "";
        }
        start=cur;
     }
     if (cur : Loop|ExpStmt) { curBlock = cur :: curBlock; }
  }
  if (curBlock!="")
     blocks[start] = BasicBlock#(count+1, curBlock);
  blocks
</xform>
```

In the above code, we define a code template *BasicBlock* to be composed of a label together with a sequence of statements. A function *BuildBasicBlocks* is then defined to build all the basic blocks. In particular, we use an associative map (the *blocks* variable) to remember which AST nodes should start a new basic block. For each AST node *cur* within the given code *input*, if *cur* is not included in *blocks* (that is, $blocks[cur] != ""$), then *cur* should be included in the current basic block being built; otherwise, if $blocks[cur] == 1$, then the current AST node *cur* starts a different basic block, so we include *cur* as the first statement of a new basic block. Note that as *input* code is traversed, each statement is prepended to the current basic block, and the *start* variable keeps track of the last statement added to the basic block. For details on the POET statements, see POET language manual.

The second step of control-flow graph construction is to connect the basic blocks with edges. The following defines code templates that support the construction and output of a control-flow graph.

```
<code GraphEdge pars=(from,to) >
"@(from)@"->"@(to)@"
</code>
<code CFG pars=(label, flow : LIST(GraphEdge,"\n"))>
digraph @label@
{
  @flow@
}
</code>
```

The following function adds a control-flow edge between each pair of adjacent basic blocks in a list and then output the generated control flow graph.

```
<xform BuildCFG pars=(input, blocks) prev="" edges="">
    cur=blocks[input];
    if (cur != "")
     {
       if (prev != "")
         edges = GraphEdge#(prev,cur) :: edges;
       prev=cur;
     }
   switch (input) {
    case (first second):
       (e1, b1) = BuildCFG(first,blocks);
       if (second != "") { BuildCFG[prev=b1;edges=e1](second,blocks)}
       else { (e1,b1) }
    case Nest#(loop, body):
       (e1,b1) = BuildCFG(body, blocks);
       (GraphEdge#(b1,cur)::e1, b1)
    case ExpStmt: (edges, prev)
    default:
       foreach (input : (cur=_) : FALSE) { <<* look inside?
           if (HEAD(cur) : ExpStmt|Nest)
              { RETURN (BuildCFG[prev=""](cur,blocks)); }  <<* yes
       }
       (edges,prev) <<* no need to look inside
}
</xform>
```

The following is an example output of the constructed control-flow graph.

```
digraph CFG
{
  "2[int i,j,l;C[j*ldc+i] = beta*C[j*ldc+i];]"
   ->
   "1[C[j*ldc+i] = C[j*ldc+i]+alpha*A[i*lda+l]*B[j*ldb+l];]"
}
```

The graph can be output into a file named, say *output.dot*, and then viewed using *graphviz*, the open-source graph visualization project from AT&T research (you can download the software for free by googling *graphviz* or *dot*).

7 Traversing The Control-flow Graph

Building a control-flow graph simply means identifying all the basic blocks and then adding a sequence of graph edges to represent the flow of control among the basic blocks. However, control-flow graph often needs to be traversed in efficient ways. For example, to support data flow analysis, we need to quickly find all the successors (or predecessors) of an arbitrary basic block.

Because POET was designed to target code transformation instead of program analysis, the code templates in POET can be used to easily build flexible tree data structures (e.g., the AST) but currently does not support cyclic data structures. Therefore, you need to use additional data structures, e.g., associative maps, to compensate such limitation. For example, to quickly find the successors of an arbitrary basic block, the following code builds an associative table that maps each basic block to a list of its successors.

```
<xform MapSuccessors pars=(cfgEdges)>
  res = MAP(_,_);
  foreach (cfgEdges : GraphEdge#(CLEAR from, CLEAR to) : TRUE)
  {
     res[from] = BuildList(to, res[from]);
  }
  res
</xform>
```

8 Data-flow Analysis

When performing data-flow analysis, each basic block needs to be associated with a set of information (e.g., a set of expressions or variables). These sets of information can be implemented using either the built-in compound type *lists* in POET or using associative maps in POET. In general, you need to first traverse the entire input, find the set of all the entities in your analysis domain. The set of information associated with each basic block is then a subset of this domain.

Since set intersection, union, and subtraction are often applied frequently to subsets of the analysis domain, it is often convenient to implement each subset of the analysis domain using the POET associative map. In particular, if an item is mapped to the empty string (i.e., "") in the associaitive map, then the item is not in the subset; otherwise, it is in the subset.

9 Modifying the AST

The *REPLACE* operator is the most useful operation in POET to support code transformation. In particular, the *REPLACE* operator can be invoked using two different syntax, as illustrated int the following.

```
REPLACE("x","y", SPLIT("","x*x-2"));
REPLACE( (("a",1) ("b",2) ("c",3)), SPLIT("","a+b-c"));
```

The first REPLACE invocation replaces all occurrences of string "x" with string "y" in the third argument of *REPLACE* (the result of *SPLIT*("", "$x * x - 2$")); the second REPLACE invocation makes a sequence of replacement. In particular, it first looks for the occurrence of string "a". When found, it replaces "a" with 1, and then moves on and tries locate string "b", the origin of the second replacement. Once the string "b" is found, it is replaced with 2 and then we proceeds by trying to look for "c". Each replacement is applied only once, and a warning message is issued if the entire list of replacement specifications are not exhausted when reaching the end of the input argument (the last argument of REPLACE).

Note that each REPLACE operation performs the replacements by returning a new data structure. The original input (i.e., the last arguments of the operation) is never modified (unless a trace

handle is used, which is out of the scope of this discussion). More generally, POET is a functional programming language. Except for tracing handles (which are more advanced topics and are not discussed here), none of the compound data structures in POET can be modified. Therefore, during the process of redundancy elimination, you should try to build a list of all the replacement and then apply all the replacements in the end. The following is an example illustrating this process (see POET/examples/C2F.pt)

```
repl="";
foreach_r (input : (VarTypeDecl#(CLEAR type,_)) :FALSE)
   { repl=BuildList( (type, TranslateCtype2Ftype(type)), repl); }
input = REPLACE(repl, input);
```

Here to translate variable type declarations in C syntax to those in Fortran, the above code first build an empty replacement list and uses the variable *repl* to keep track of this list. It then go over the entire input in reverse order and locate all the $VarTypeDecl$ objects (the reverse traversal order almost always needs to used so that the replacement list is in the correct order). The code then extends the *repl* list by pre-pending a new replacement (a (source, dest) tuple) pair to it. After traversing the entire input, the list of replacements in *repl* is then applied collectively by invoking the $REPLACE$ operation on *input*.

10 Other Tasks

Value Numbering. As value number is a combined process of hash table management and AST modification, it is obvious that the POET associative map should be used to implement the hashing of value numbers and variable names, and that AST modification should be implemented by invoking the REPLACE operation. Note that **you cannot modify ASTs in POET, see Section 9).**

Three-address or machine code generation . This means translating the input program to another language. Here you need to define a collection of code templates for the new languages (in the case of three-address code, a single code template for formating three-address instructions). The translation can then proceed by translating the input AST to a three-address AST (or machine code AST). The output AST can then be unparsed in the proper format using the POET output command, as illustrated in the following (see POET/examples/TranslatorDriver.pt).

```
<output to=(outputFile) syntax=(outputSyntax) from=resultCode/>
```