Parsing of Context-Sensitive Languages

Lukáš Rychnovský

Dept. of Information Systems, Faculty of Information Technology,
Brno University of Technology, Božetěchova 1, 612 66 Brno, Czech Republic
rychnov@fit.vutbr.cz

Abstract. This article presents some ideas from parsing Context-Sensitive languages. Introduces Scattered-Context grammars and languages and describes usage of such grammars to parse CS languages. Also there are presented additional results from type checking and formal program verification using CS parsing.

Keywords: Turing Machines, Parsing of Context-Sensitive Languages, Formal Program Verification, Scattered-Context Grammars.

1 Introduction

This work results from [Kol–04] where relationship between regulated pushdown automata (RPDA) and Turing machines are discussed. We are particulary interested in algorithms which may be used to obtain RPDAs from equivalent Turing machines. Such interest arises purely from practical reasons because RPDAs provide easier implementation techniques for problems where Turing machines are naturally used. As a representant of such problems we study context-sensitive extensions of programming language grammars which are usually context-free. By introducing context-sensitive syntax analysis into the source code parsing process a whole class of new problems may be solved at this stage of a compiler. Namely issues with correct variable definitions, type checking etc.

2 Scattered context grammars

Definition 1. *A scattered context grammar (SCG) G is a quadruple (V, T, P, S), where V is a finite set of symbols, $T \subset V$, $S \in V \backslash T$, and P is a set of production rules of the form*

$$(A_1, \ldots, A_n) \rightarrow (w_1, \ldots, w_n), n \geq 1, \forall A_i : A_i \in V \backslash T, \forall w_i : w_i \in V^*$$

Definition 2. *Let $G = (V, T, P, S)$ be a SCG. Let $(A_1, ..., A_n) \rightarrow (w_1, ..., w_n) \in P$. Then we define a derivation relation $\Rightarrow$ as follows: for $1 \leq i \leq n+1$ let $x_i \in V^*$. Then*

$$x_1 A_1 x_2 A_2 ... x_n A_n x_{n+1} \Rightarrow x_1 w_1 x_2 w_2 ... x_n w_n x_{n+1}$$

$\Rightarrow^*$ is reflexive, transitive closure of $\Rightarrow$.
Language generated by the grammar G is defined as $L(G) = \{w \in T^* | S \Rightarrow^* w\}$.

Theorem 1. *Let $L_n(SC)$ be the family of languages generated by SC grammars whose number of productions that contains two or more context-free productions (degree of context sensitivity) is n or less. $L(RE)$ denotes the family of recursively enumerable languages.*

$$L_2(SC) = L_\infty(SC) = L(RE)$$

Proof. See [Med–03] Lemma 1 and Theorem 3.

3 Parsing of Context-Sensitive Languages

The main goal of regulated formal systems is to extend abilities from standard CF LL-parsing to CS or RE families with preservation of ease of parsing.

In [Kol–04] and [Rych–05] we can find some basic facts from theory of regulated pushdown automata (RPDA). We figured that regulated pushdown automata can in some cases simulate Turing machines so we could use this theory for constructing parsers for context-sensitive languages or even type-0 languages.

We have also demonstrated the basic problem of this concept: complexity. Almost trivial Turing machine was transformed to regulated pushdown automata with almost 6 000 rules.

As we have seen in previous lines, converting deterministic (linear bounded) Turing machine or scattered-context grammar to deterministic RPDA is very complex task. For the most simple context-sensitive languages corresponding deterministic RPDA has thousands of rules. If we want to use these algorithms for creating some practical parser for real context-sensitive programming language it may result in millions of rules. Therefore, we are looking for another way to parse context-sensitive languages. We would like to extend some context-free grammar of any common programming language (such as Pascal, C/C# or Java). After extending context-free grammar to corresponding context-sensitive grammar, parsing should be straightforward.

3.1 KontextZAP03

As an example of a context-free language we use a language called ZAP03 [ZAP–03] which has very similar syntax to Pascal. We will use following program as an example program in ZAP03 language.

```
int : a, b, c, d;
string : s;

begin
   a = 1;
   b = 2;
   c = 10;
   d = 15;
   s = "foo";
```

```
  a = c;
end
```

Now we will define KontextZAP03, the context-sensitive extension of ZAP03. In the first phase we will enrich KontextZAP03 by variable checking. If the variable is undefined or assigned before initialized, the parser of KontextZAP03 will finish in error state. We will need to analyze three fragments of ZAP03 code where variables are used (variable `c` for example).
Variable definition
`int : a, b, c, d;`
Assignment statement
`c = 10;`
And using variables in commands
`a = c;`
Corresponding grammar fragments from ZAP03 are following.
Variable definition
`DCL` → `TYPE [:] [id] ID_LIST`
`ID_LIST` → `[,] [id] ID_LIST`
`ID_LIST` → ε
Fragment of assignment statement
`COMMAND` → `[id] CMD COMMAND`
`CMD` → `[=] STMT [;]`
And usage variable in command
`STMT` → `[id] OPER`
`OPER` → ε.
Symbols in brackets `[,]` are terminals. Complete ZAP03 grammar has about 70 context-free grammar rules.

We define grammar of language KontextZAP03 in the following way. Substitute previous rules with these scattered-context ones:
`(DCL, S')` → `(TYPE [:] [id] ID_LIST, D)`
`(ID_LIST, S')` → `([,] [id] ID_LIST, D)`
`(ID_LIST)` → (ε)
assignment statement
`(COMMAND, D)` → `([id] CMD COMMAND, DL)`
`(CMD)` → `([=] STMT [;])`
and usage variable in command
`(STMT, D)` → `([id] OPER, DR)`
`(OPER)` → (ε).

Parsing now proceeds in the following way: starting symbol is `S S'` and derivation will go as usual until there is `DCL S'` processed and `(DCL, S')` → `(TYPE [:] [id] ID_LIST, D)` rule is to applied. At this moment `S'` is rewritten to `D` indicating that variable [id] is defined. When variable [id] is used on left resp. right side of assignment `D` is rewritten to `DL` resp. `DR` according to second resp. third previously shown fragment. If variable [id] is used without being defined beforehand, a parse error occurs because `S'` is not rewritten to `D` and `S'` cannot

be rewritten to DL or DR directly. When the input is parsed S is rewritten to program code and during LL parsing is popped out of the stack. S' is rewritten onto D{LR}* and this is only string that remains.

S S' $\Rightarrow^*$ DCL S' $\Rightarrow$ TYPE [:] [id] ID_LIST D $\Rightarrow^*$ COMMAND D $\Rightarrow$
$\Rightarrow$ [id] CMD COMMAND DL $\Rightarrow^*$ DL

If the only remaining symbol is D, it means that variable [id] was defined but never used. If DL^+ is the only remaining symbol, we know that variable [id] was defined and used only on the left sides of assignments. Finally if there is the only remaining DR(LR)*, we know that the first occurrence of variable [id] is on the right side of an assignment statement and therefore it is being read without being set. In all these cases the compiler should generate a warning. These and similar problems are usually addressed by a data-flow analysis phase carried out during semantic analysis.

Using this algorithm we can only process one variable at a time. But the proposed mechanism can be easily extended to a finite number of variables by adding new S'$\cdots$' every time we discover a variable definition. Parsing of described SC grammar can be implemented by pushdown automaton with finite number of pushdowns. The first pushdown is classic LL pushdown. The second one is variable specific and every [id] holds its own.

Because original ZAP03 grammar is LL1 and using described algorithm wasn't any rule added, KontextZAP03 grammar has unambiguous derivations.

A few examples can clear the idea. This program is well-formed according to ZAP03 grammar, but it's semantics is not correct and parsing it as KontextZAP03 program should reveal this error.

```
int : a, b, c, d;
string : s;

begin
   a = 1;
   b = 2;
   d = 15;
   s = "foo";
   a = c;
end
```

Corresponding stack to variable c will be DR what lead to warning:

```
Variable c read but not set.
```

Second example shows another variation

```
int : a, b, c, d;
string : s;

begin
   a = 1;
   d = 15;
   s = "foo";
   a = c;
```

```
end
```
Corresponding stack to variable `b` will be `D` what lead to warnings (together with previous one):
```
Variable c read but not set.
Variable b is defined but never used.
```

3.2 Type checking

By using scattered-context grammars we can describe type set of language (INT, STR) and type check rules directly in grammar. Almost trivial language with type check using 5 stacks can look like this:

$(s) \rightarrow (program)$
$(program) \rightarrow (dcl)$
$(program) \rightarrow ([begin]command[end])$
$(dcl) \rightarrow (type[:]dcl2)$
$(dcl2, S, INT, ,) \rightarrow$
$([id]id\_list[;]next, D, INT, INT)$
$(dcl2, S, STR, ,) \rightarrow$
$([id]id\_list[;]next, D, STR, STR)$
$(id\_list) \rightarrow ([,]id_{l}ist2)$
$(id\_list2, S) \rightarrow ([id]id\_list, D)$
$(id\_list) \rightarrow (\varepsilon)$
$(next) \rightarrow (dcl)$
$(next) \rightarrow (program)$
$(type, ,) \rightarrow ([int], , INT)$
$(type, ,) \rightarrow ([string], , STR)$
$(command, D, INT, ,) \rightarrow$
$([id]\ cmd\ command, D\ L, INT, INT)$
$(command, D, STR, ,) \rightarrow$
$([id]\ cmd\ command, D\ L, STR, STR)$
$(command) \rightarrow (\varepsilon)$
$(cmd) \rightarrow ([=]stmt[;])$
$(stmt, D) \rightarrow ([id], D\ R)$
$(stmt, , , , INT) \rightarrow ([digit], , , ,)$
$(stmt, , , , STR) \rightarrow ([strval], , , ,)$

Stacks at even positions are the variable specific stacks as mentioned in previous chapter. The first stack is classic LL stack and the rest of stacks at odd positions are temporary used stacks for additional information. Although the underlying context-free grammar is not LL grammar because there are several identic rules (command → [id] cmd command), this grammar is unambiguous.

Example of error program code can be
```
int : a, b, c, d;
string : s;

begin
   a = 1;
   b = 15;
   c = "foo";
   a = c;
end
```
because `c = "foo"` is not type correct (STR is assigned to INT) and parsing will fail.

3.3 Other applications of CS languages

It is obvious, that using a simple scattered-context extension of CF languages, we obtain a grammar with interesting properties with respect to analysis of a programming language source code. We provide some basic motivation examples.

1. Errors related to usage of undefined variables may be discovered and handled at parse time without the need to handle them by static semantic analysis.
2. A CS extension of a grammar of the Java programming language, which copes with problems like mutual exclusion of various keywords, such as `abstract` and `final`, reflecting the fact that abstract methods cannot be declared final and vice versa. This situation can be handled quite easily, by introducing additional symbol S'' and two corresponding rules S'' → A (corresponds to `abstract`) and S'' → F (corresponds to `final`). Obviously only one of the rules can be used at a time.
3. Introducing an `observer` keyword for methods in Java, which indicates that this method does not modify the state of *this* object (similar to defining method as `const` in C++). Handling of such keyword in the language grammar is similar to approach taken in the previous example.
4. Accounting of statements in a program in a ZAP03 language by introducing new `size` keyword, which defines upper bound on the number of statements in current scope. The parser is than extended such that when the keyword is discovered, the number of specialized nonterminals (say `X`) is generated on the stack – as specified by the keyword occurrence and the grammar of the language is modified accordingly. The rule:
   `COMMAND` → `[id] CMD COMMAND`
   changes to:
   `(COMMAND, X)` → `([id] CMD COMMAND,` ε `)`
   Then, when a statement rule is used, one `X` nonterminal is eliminated from the stack. If there are no remaining `X` nonterminals, the parsing immediately fails. The context-sensitive language used in this example is:

$$L(G) = \{w.|w|_{10}\},$$

   where w is word and $|w|_{10}$ is the length of w written as a decimal number.

4 Formal Program Verification

One of the promising applications of the described concept is introducing a notion of *preconditions* and *postconditions* to the ZAP03 programming language. Preconditions and postconditions are essentially sets of logical formulae, which are required to hold at entering resp. leaving a program or method. For example, when computing a square root of x, we can require a positivity of x using precondition $\{x >= 0\}$. A computation of sinus function $\{y = sinx\}$ a natural postcondition $\{(y <= 1) \wedge (y >= -1)\}$ arises.

Statements of a programming languages then induce transformation rules on precondition and postcondition sets. For example, assignment statement in the form $V := E$ defines a transformation:

$$\{P[E/V]\}V := E\{P\},$$

where V is a variable, E is an expression, P is precondition and P[E/V] denotes a substitution of V for all occurrences of E in P. Other transformation examples may be found in [Gor–98].

The purpose of introducing preconditions and postconditions into a language is to be able to derive postconditions from specified preconditions using transformation rules in a particular program. Such program than carries a formal proof of its correctness with it, which is a desirable property.

In the ZAP03 language we can implement the described concept by introducing two new keywords `pre` and `post` and by extending the rules of the language grammar with above mentioned transformation rules. When the parsing of a program is initiated, the precondition set is constructed using the `pre` declarations and the transformation rules are applied to it as the parsing progresses through the source code. When the parsing terminates, the resulting set of transformed preconditions is compared with the declared postconditions. If these two sets match, the parsed program is correct with respect to the specified preconditions and postconditions.

However, for practical reasons we must define constraints on the possible preconditions and postconditions. Postconditions must be generally derivable from preconditions or (as a corollary of the Gödels incompleteness theorem) the postconditions need not be provable from the preconditions at all, but may still hold. In this particular case we have encountered a program which may be correct, but we cannot verify this fact.

5 Conclusions

In this article are presented some ideas from parsing of context-sensitive languages based on scattered-context grammars. Very powerful extension of classic parsing techniques is introduced and this context-sensitive extension let us discover some common errors such as undefined or unused variables in parsing time. The extension suggested in this work is far more beyond. In some conditions we are able to add some features to language that enables us to check formal program verification.

References

[Aho–72] Aho, A., Ullman, J.: The Theory of Parsing, Translation, and Compiling, Prentice-Hall INC. 1972.

[Gor–98] Gordon, J. C. M.: Programming Language Theory and its Implementation, 1998.

[Kol–04] Kolář, D.: Pushdown Automata: New Modifications and Transformations, Habilitation Thesis, 2004.

[Med–00] Meduna, A., Kol, D.: Reguleted Pushdown Automata, Acta Cybernetica, Vol. 14, 2000. Pages 653-664.

[Med–03] Meduna, A., Fernau, H.: On the degree of scattered context-sensitivity, Elsevier, Theoretical Computer Science 290 (2003), Pages 2121-2124.

[Sal–73] Salomaa, A.: Formal Languages, Academic Press, New York, 1973.

[Rych–05] Rychnovsky, L.: Relation between regulated pushdown automata and Turing mascines, Report from semestral project, 2005.

[ZAP–03] http://www.fit.vutbr.cz/study/courses/ZAP/public/project/