TYPE CHECKING BY CONTEXT-SENSITIVE LANGUAGES

Lukáš Rychnovský
Doctoral Degree Programme (1), FIT BUT
E-mail: rychnov@fit.vutbr.cz

Supervised by: Dušan Kolář
E-mail: kolar@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. The main goal of this article is to present results from type checking using CS parsing.

1 INTRODUCTION

This work results from [Kol–04] where relationship between regulated pushdown automata (RPDA) and Turing machines is 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 2.1 *A scattered context grammar (SCG) G is a quadruple (V,T,P,S), where V is a finite set of symbols, $T \subset V$, $S \in V \setminus T$, and P is a set of production rules of the form*

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

Definition 2.2 *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*

$$x_1A_1x_2A_2...x_nA_nx_{n+1} \Rightarrow x_1w_1x_2w_2...x_nw_nx_{n+1}$$

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

Theorem 2.1 *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.
From [Rych–05] it is obvious that converting Turing machine to corresponding RPDA is very complex task. Even simple Turing machine (20 states, 10 symbols) can result in many (thousands) rules in RPDA.
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 KONTEXT-ZAP03

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 Kontext-ZAP03, the context-sensitive extension of ZAP03. In the first phase we will enrich Kontext-ZAP03 by variable checking. If the variable is undefined or assigned before initialized, the parser of Kontext-ZAP03 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 Kontext-ZAP03 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 $\texttt{D}\{\texttt{LR}\}^*$ and this is only string that remains.

$$\texttt{(S,S')} \Rightarrow^* \texttt{(DCL,S')} \Rightarrow \texttt{(TYPE [:] [id] ID\_LIST,D)} \Rightarrow^* \texttt{(COMMAND,D)} \Rightarrow$$
$$\Rightarrow \texttt{([id] CMD COMMAND,DL)} \Rightarrow^* (\varepsilon, \texttt{DL})$$

If the only remaining symbol is `D`, it means that variable [id] was defined but never used. If $\texttt{DL}^+$ 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 $\texttt{DR(LR)}^*$, 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'···'` 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, Kontext-ZAP03 grammar has unambiguous derivations.
An example can clear the idea. This program is well-formed according to ZAP03 grammar, but it's semantics is not correct and parsing it as Kontext-ZAP03 program should reveal this error.

```
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 the first warning and corresponding stack to variable `c` will be `DR` what lead to another warning:

```
Variable b is defined but never used.
Variable c read but not set.
```

4 TYPE CHECKING

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

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

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 :  c;

begin
  c = "foo";
end
```

Because `c = "foo"` is not type correct (STR is assigned to INT) and parsing will fail.

$$(S, S, \varepsilon, \varepsilon, \varepsilon) \Rightarrow^* (DCL, S, \varepsilon, \varepsilon, \varepsilon) \Rightarrow (TYPE\ [:]\ DCL2, S, \varepsilon, \varepsilon, \varepsilon)$$
$$\Rightarrow([int]\ [:]\ DCL2, S, INT, \varepsilon, \varepsilon) \Rightarrow^2 (DCL2, S, INT, \varepsilon, \varepsilon) \Rightarrow$$
$$\Rightarrow([id]\ ID\_LIST\ [;]\ NEXT, D, INT, INT, \varepsilon) \Rightarrow^*$$
$$\Rightarrow^* (COMMAND\ [end], D, INT, INT, \varepsilon) \Rightarrow$$
$$\Rightarrow ([id]\ CMD\ COMMAND\ [end], DL, INT, INT, INT) \Rightarrow^*$$
$$\Rightarrow^* (STMT\ [;]\ COMMAND\ [end], DL, INT, INT, INT) \Rightarrow$$
$$\Rightarrow ([digit]\ [;]\ COMMAND\ [end], DL, INT, INT, INT)$$

Now parsing will fail because the input character is [*strval*] instead of [*digit*].

5 CONCLUSION

In this article we described some possibilities of Context-Sensitive languages and advantages of parsing such languages. Injecting type rules directly into Context-Free grammar is one of this advantages. Another usage of CS parsing techniques and languages can be found in [Rych–05].

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., Kolář, 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.

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

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

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