Generation of indexes for compiling efficient parsers from formal specifications*

Carlos Gómez-Rodríguez[1], Miguel A. Alonso[1], and Manuel Vilares[2]

[1] Departamento de Computación
Universidade da Coruña, Spain
{cgomezr,alonso}@udc.es

[2] Departamento de Informática
Universidade de Vigo, Spain
vilares@uvigo.es

Extended abstract

Parsing schemata [4] provide a formal, simple and uniform way to describe, analyze and compare different parsing algorithms. The notion of a parsing schema comes from considering parsing as a deduction process which generates intermediate results called *items*. An initial set of items is directly obtained from the input sentence, and the parsing process consists of the application of inference rules (called *deductive steps*) which produce new items from existing ones. Each item contains a piece of information about the sentence's structure, and a successful parsing process will produce at least one *final item* containing a full parse tree for the sentence or guaranteeing its existence.

Their abstraction of low-level details makes parsing schemata very useful, allowing us to define parsers in a simple and straightforward way. Comparing parsers, or considering aspects such as their correction and completeness or their computational complexity, also becomes easier if we think in terms of schemata. However, when we want to actually use a parser by running it on a computer, we need to implement it in a programming language, so we have to abandon the high level of abstraction and worry about implementation details that were irrelevant at the schema level. To overcome this, we have developed a compiler that given a parsing schema is able to obtain an executable implementations of the corresponding parser [3, 2].

In this paper, we study how the compiler analyses the source parsing schema to decide what kind of indexes need to be generated in order to obtain an efficient parser. In particular, implementation of the following operations affects the resulting parser's computational complexity:

- Check if a given item exists in the item set.
- Search the item set for all items satisfying a certain specification: Once an antecedent of a deductive step has been matched, a search for items matching the rest of the antecedents is needed in order to make inferences using that step.

* Partially supported by Ministerio de Educación y Ciencia (MEC) and FEDER (TIN2004-07246-C03-01, TIN2004-07246-C03-02), Xunta de Galicia (PGIDIT05PXIC30501PN, PGIDIT05PXIC10501PN, PGIDIT05SIN044E) and Programa de Becas FPU of MEC.

In order to maintain the theoretical complexity of parsing schemata, we must provide constant-time access to items. In this case, each single deduction takes place in constant time, and the worst-case complexity is bounded by the maximum possible number of step executions: all complexity in the generated implementation is inherent to the schema.

In order to achieve this, we propose to generate two distinct kind of indexes for each schema, corresponding to the operations mentioned before:

- Items are indexed in *existence indexes*, used to check whether an item exists in the item set, and *search indexes*, which allow us to search for items conforming to a given specification.
- Deductive steps are indexed in *deductive step indexes*. These indexes are used to restrict the set of "applicable deductive steps" for a given item, discarding those known not to match it. Deductive step indexes usually have no influence on computational complexity with respect to input string size, but they do have an influence on complexity with respect to the size of the grammar, since the number of step instances depends on grammar size when grammar productions are used as side conditions.

A simple case. The generation of indexing code is not trivial, since the elements by which we should index items in order to achieve efficiency vary among schemata. For instance, if we are trying an Earley [1, 4] COMPLETER step

$$\frac{[A \to \alpha.B\beta, i, j], [B \to \gamma., j, k]}{[A \to \alpha B.\beta, i, k]}$$

on an item of the form $[B \to \gamma., j, k]$ which matches the second antecedent, we will need to search for items of the form $[A \to \alpha.B\beta, i, j]$, for any values of A, α, β and i, in order to draw all the possible conclusions from the item and step. Since the values of B and j are fixed, this search will be efficient and provide constant-time access to items if we have them indexed by the symbol that follows the dot and by the second string position (B and j). However, if we analyze the other Earley steps in the same way, we will find that their indexing needs are different, and different parsing schemata will obviously have different needs.

Therefore, in order to generate indexing code, we must take the distinct features of each schema into account. In the case of search indexes, we must analyze each deductive step just as we have analyzed the COMPLETER step: we must keep track of which variables are instantiated to a concrete value when a search must be performed. This information is known at schema compilation time and allows us to create indexes by the elements corresponding to instantiated variables. For example, in the case of COMPLETER, we would create the index that we mentioned before (by the symbol directly after the dot and the second string position) and another index by the symbol in the left side of productions and the first string position. This second index is useful when we have an item matching the first antecedent and we want to search for items matching the second one, and is obtained by checking which variables are instantiated when the first antecedent is matched.

The generation of existence indexes is similar to, but simpler than, that of search indexes. The same principle of checking which variables will be instantiated when the index is needed is valid in this case, but when an item is checked for existence it is always fully known, so all its variables are instantiated.

Deductive step indexes are generated by taking into account those step variables which will take a value during instantiation, i.e. which variables appear on side conditions. Since these variables will have a concrete value for each instance of the step, they can be used to filter instances in which they take a value that will not allow matching with a given item.

A general case. For a more general view of how the adequate indexes can be determined by a static analysis of the schema prior to compilation, we can analyze the general case where we have a deductive step of the form

$$\frac{[a, d, e, g], [b, d, f, g]}{consequent} \quad c\ e\ f\ g$$

where each lowercase letter represents the set of elements (be them grammar symbols, string positions or other entities) appearing at particular positions in the step, so that a stands for the set of elements appearing only in the first antecedent item, e represents those appearing in the first antecedent and side condition, g those appearing in both antecedents and side condition, and the rest of the letters represent the other possible combinations as can be seen in the step. We can easily see that any deductive step with two antecedent items can be represented in this way (note that the element sets can be empty, and that the ordering of elements inside items is irrelevant to this discussion).

In this case, the following indexes are generated:

- One deductive step index for each antecedent, using as keys the elements appearing both in the side condition *and* in that particular antecedent: therefore, two indexes are generated using the values (e, g) and (f, g). These indexes are used to restrict the set of deductive step instances applicable to items. As each instance corresponds to a particular instantiation of the side conditions, in this case each step instance will have different values for c, e, f and g. When the deductive engine asks for the set of steps applicable to a given item $[w, x, y, z]$, the deductive step handler will use the values of (y, z) as keys in order to return only instances with matching values of (e, g) or (f, g). Instances of the steps where these values do not match can be safely discarded, as we know that our item will not match any of both antecedents.
- One search index for each antecedent, using as keys the elements appearing in that antecedent which are also present in the side condition *or* in the other antecedent. Therefore, a search index is generated by using (d, e, g) as keys in order to recover items of the form $[a, d, e, g]$ when d, e and g are known and a can take any value; and another index using the keys (d, f, g) is generated and used to recover items of the form $[b, d, f, g]$ when d, f and g are known. The first index allows us to efficiently search for items matching

the first antecedent when we have already found a match for the second, while the second one can be used to search for items matching the second antecedent when we have started our deduction by matching the first one.
– One existence index using as keys all the elements appearing in the consequent, since all of them are instantiated to concrete values when the step successfully generates a consequent item. This index is used to check whether the generated item already exists in the item set before adding it.

As this index generation process must be applied to all deductive steps in the schema, the number of indexes needed to guarantee constant-time access to items increases linearly with the number of steps. However, in practice we do not usually need to generate all of these indexes, since many of them are repeated or redundant. For example, if we suppose that the sets e and f in our last example contain the same number and type of elements, and elements are ordered in the same way in both antecedents, the two search indexes generated would in fact be the same, and our compiler would detect this fact and generate only one. In practical cases, the items used by different steps of a parsing schema usually have the same structure, so most indexes can be shared among several deductive steps and the amount of indexes generated is small.

In our example, we have considered only two antecedents for the sake of simplicity, but the technique is general and can be applied to deductive steps with an arbitrary number of antecedents.

Although all the cases we have seen so far correspond to context-free parsing, our technique is not limited to working with context-free grammars, since parsing schemata can be used to represent parsers for other grammar formalisms as well. All grammars in the Chomsky hierarchy can be handled in the same way as context-free grammars. For example, we have generated implementations for some of the most popular parsers for tree adjoining grammars [2].

References

1. J. Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102, 1970.
2. C. Gómez-Rodríguez, M.A. Alonso, and M. Vilares. Generating XTAG parsers from algebraic specifications. In *Proceedings of the 8th International Workshop on Tree Adjoining Grammar and Related Formalisms. Sydney, July 2006*, pp. 103-108, Association for Computational Linguistics, East Stroudsburg, PA, 2006.
3. C. Gómez-Rodríguez, J. Vilares, and M. Vilares. Automatic Generation of Natural Language Parsers from Declarative Specifications. In L. Penserini, P. Peppas and A. Perini (eds.), *STAIRS 2006 - Proceedings of the Third Starting AI Researchers' Symposium, Riva del Garda, Italy, August 28-29, 2006*, volume 142 of *Frontiers in Artificial Intelligence and Applications*, pp. 259-260, IOS Press, Amsterdam, 2006.
4. K. Sikkel. *Parsing Schemata — A Framework for Specification and Analysis of Parsing Algorithms*. Springer-Verlag, Berlin/Heidelberg/New York, 1997.