PRACTICAL TYPE CHECKING OF FUNCTIONS DEFINED ON CONTEXT-FREE LANGUAGES*

Chen Haiming and Dong Yunmei
Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences
Beijing 100080, P R China
{chm,dym}@ios.ac.cn

ABSTRACT

LFC is a functional language based on recursive functions defined on context-free languages. In LFC context-free language (CFL) is used as data type to represent compound data structures. This makes LFC a dynamically typed language. To improve efficiency, we present a practical type checking method, which consists of both static and dynamic type checking. Although the inclusion relation of CFLs is not decidable, a special subset of the relation is decidable, i.e. the sentential pattern relation, which can be statically checked. Moreover, most of the expressions in actual LFC programs appear to satisfy this relation according to the statistic data of our experiments. So, despite the static type checking is not complete, it undertakes most of the type checking task. Consequently the run time efficiency is greatly improved.

Keywords: type checking, context-free language, algorithm.

1 INTRODUCTION

LFC is a functional language which is based on the theory of recursive functions defined on context-free languages[1]. The language has been employed as a specification language to support specification acquisition. It uses context-free language (CFL for short) as data type to represent compound data structures.

Type checking is an important issue of programming language research, which can improve the productivity and quality of the programming task.

Utilizing the new kind of recursive functions, LFC in nature is a dynamically typed language. It is natural to perform type checking dynamically in the implementation of LFC[2]. For CFL types this is accomplished by parsing values of expressions, which is expensive in time. The time cost of running LFC programs will become rather high if there are considerable amount of dynamic parsing. In order to improve the time efficiency of LFC we incorporated static type checking into LFC in the desire to maximally reduce the need for dynamic type checking.

Expressions of CFL types have implicit structures, which are disadvantageous for efficient implementation. So the type checking of LFC will convert expressions into explicitly structured representation.

The authors have not seen any other research on type checking with CFL types, we hope the paper will be of help to this issue.

This paper briefly describes the approach. A more detailed description can be found in [3]. Some knowledge about context-free language is assumed.

2 TYPES AND EXPRESSIONS

A context-free grammar G uniquely defines a CFL type $L(G)$. For simplicity, we consider the name of a CFL type the same as the start symbol of the grammar defining the CFL.

In LFC, CFL types represent structured data types, just like the compound data types in other languages. But the construction of CFL types is quite different from those of other compound data types. The basic operations for constructing a CFL type are union ($\cup$) and concatenation (., usually omitted).

Concatenation: A production of a context-free grammar has the form $X \rightarrow \alpha_1 \ldots \alpha_n$, where $\alpha_i \in V_T^*$ or $\alpha_i \in V_N$. (V_T and V_N denote respectively the disjoint sets of terminal and nonterminal symbols.)

Union: For a group of productions $X \rightarrow \alpha_1 \mid \ldots \mid \alpha_n$, where $\alpha_i \in (V_N \cup V_T)^*$, i=1,…,n, we have $L(X) = L(\alpha_1) \cup \ldots \cup L(\alpha_n)$. $L(X)$ is the union of $L(\alpha_i)$.

For a n-ary function f with type $L_1 \times \cdots \times L_n \rightarrow L$,

* This research was supported by the National Natural Science Foundation of China.

where the L_i and L are CFLs, the definition of f has the following form

$$f(p_1^1, \cdots, p_1^n) = e_1$$
$$\cdots$$
$$f(p_m^1, \cdots, p_m^n) = e_m$$

where each p_i^j is a *pattern* of L_j, which is either a variable of L_j, or derived from a term α of L_j by replacing every nonterminal symbol R in α with a distinct variable of $L(G_R)$ (note that in this case a pattern may also be a single variable, we should distinguish it from the former case in implementation); e_i are expressions. A term of $L(G)$ is the right-hand side of a production of which the left nonterminal symbol is the start symbol of G.

Basic expressions of CFL types have the following syntax.

c	Constant
v	Variable
f	Function name
$e ::= c \mid v \mid e_1 \cdot e_2 \mid f(e_1, \ldots, e_1)$	Expression

3 PRACTICAL TYPE CHECKING

There could be equality, inclusion, or intersection relations between two CFLs, so a type system containing CFL types excludes the popular Hindley-Milner system which is based around type equality constraints[4], and is close to a subtype system[5].

However, these relations of two CFLs are not decidable. This forms the foundation of characteristics of subtype systems for CFL types.

The main characteristics are that we cannot perform type inference, and the type systems are dynamic.

Since type inference cannot be performed, type checking of LFC is, given all typing information, to check if an expression is compatible with the declared type. However, because the inclusion relation of two CFLs is not decidable, the type system of LFC must be a dynamic system.

There would be two ways to solve the type problem of CFL types. One way is to impose constraints on expressions or on grammars so that we can statically check the compatibility of an expression with the declared type. One possible constraint is to restrict expressions to sentential patterns. The cost is to lose flexibility of the language.

Another way is to use dynamic type system, but perform static type checking when possible. This would decrease run time efficiency if compared with the previous way, but would be more efficient than totally dynamic type checking. We chose this way to design the type system. Although the inclusion relation of CFLs is generally not decidable, a special subset of the relation is decidable, i.e. the sentential pattern relation, which can be statically checked. We begin with the following facts.

Definition 1. We call a string of nonterminal symbols and terminal symbols a *grammatical string*.

It is easy to know that a grammatical string defines a context-free language.

Definition 2. From an expression e of a CFL type we can derive a grammatical string by substituting the type of each variable in e for that variable. We call the language defined by this grammatical string the *least type* of expression e, denoted by $L(e)$.

Obviously for any CFL type l, e has type l if and only if $L(e) \subseteq l$.

Definition 3. Context-free language l is called a *checkable type* of expression e if the grammatical string derived from e is a sentential pattern of l.

Definition 4. l_1 is called a *subtype* of l_2 if $l_1 \subseteq l_2$.

Definition 5. We say that type of e can be *coerced* to l if the least type of e is a subtype of l.

It is easy to derive the following properties from the above definitions.

Property 1. Checkable type is not unique.

Property 2. It can be statically determined whether an expression e has a checkable type l.

Property 3. There exists l, such that l is not a checkable type of e, but the type of e can be coerced to l.

Property 4. If l is a checkable type of expression e, then $L(e)$ is a subtype of l.

Property 5. Whether the type of an expression can be coerced to an arbitrary CFL type is not decidable.

From the above discussion we can obtain the type checking method. The idea is described as follows.

First we statically check if the declared type l is a checkable type of expression e at compilation time, using sentential pattern parsing technique. If not, we then check if e can be dynamically coerced to l, i.e. determine if the value of e is a sentence of l at run time.

It can be seen that, if $L(e)$ is a subtype of l, then the type of e can be dynamically converted to l; if $L(e)$ dose not intersect with l, then the type of e cannot be dynamically converted to l. But dynamic conversion permit the case of $L(e) \not\subset l \wedge L(e) \cap l \neq \phi$. Consequently if an expression need dynamic type checking, the correctness of the type of this expression may depend on run time environment, so type correctness cannot be guaranteed. To remedy this defect, a type-warning message will be given when an expression cannot be statically type checked, so that the user can decide if the type is correct.

In order to be more efficient, before checking if e has checkable type l by using sentential pattern parser we can first check if an expression e corresponds to a

Table 1 Statistic data of sentential patterns in expressions

	Functions	Expressions	SP (term)	SP (not term)	Others
Number	135	2720	2207	399	114
Ratio			0.81	0.15	0.04

term of *l*, if not then call the parser. According to our statistic data of a large amount of instances, over 80% of the expressions that are sentential patterns correspond to terms. Since term checking is more efficient than sentential pattern parsing, the efficiency of type checking can be considerably improved.

Recall the definition of expressions in section 2, we can see that a CFL expression is basically formed by concatenation. This form cannot reflect the CFL type of an expression, therefore is a representation of implicit structure.

In order to achieve efficient implementation of LFC, an intermediate representation is necessary that can reflect the phrase structures of expressions. We have designed such an intermediate representation, and implemented structure reconstruction in type checking.

According to the above idea, a type checking algorithm is developed. For the limited size of paper, the algorithm will not be introduced here. The formal presentation of the algorithm as well as other details are in [3].

4 IMPLEMENTATION

We have realized the type checking algorithm as part of the implementation of language LFC.

The CFL types used in LFC do not have any constraints, so a general CFL parser is required. The sentence parser is based on Earley's algorithm[6] which generates the right(left) parse of a sentence. For a sentence of length *n*, the worst parse time is $O(n^3)$. The sentential pattern parser is a variant of sentence parser and is derived from Earley's algorithm. An algorithm to construct the intermediate representation from right parse produced by sentential parser is devised.

Experiments have been made on many trivial and nontrivial examples, such as string sorting and formal differentiation of elementary functions. Table 1 gives the statistic data. In the table, “SP” denotes expressions satisfying the sentential pattern relation, i.e. their least types are sentential patterns of the declared types. These expressions are further separated into two parts, denoted by “term” and “not term” respectively, one for expressions whose least types are terms of the declared types, another for other expressions. It shows that most of the expressions satisfy the sentential pattern relation. Most of the type checking work is then done statically, therefore, despite the static type checking is not complete, the run time efficiency is greatly improved. From our experiments the execution times are reduced by about two to twenty times, if compared with the previous implementation. How much the efficiency is gained mainly depends on how many expressions are sentential patterns of the declared types.

5 CONCLUSION

We present a practical type checking method for LFC. In this method both static and dynamic type checking are involved. Structures of expressions are also reconstructed to represent phrase structures. Experiments show that the run time efficiency can be greatly improved.

REFERENCES

[1]. Dong Yunmei. Recursive functions defined on context-free languages (I). Technical Report ISCAS-LCS-2k-03, Computer Science Laboratory, Institute of Software, Chinese Academy of Sciences, August 2000.

[2]. Chen Haiming. Function definition language FDL and its implementation. Journal of Comput. Sci. & Technol., Vol.14, No.4, 1999, pp.414-421.

[3]. Chen Haiming, Dong Yunmei. Practical type checking of functions defined on context-free languages. Technical Report ISCAS-LCS-2k-08, Computer Science Laboratory, Institute of Software, Chinese Academy of Sciences, Dec. 2000.

[4]. R. Milner, M. Tofte and R. Harper, The definition of Standard ML. The MIT Press, 1990.

[5]. J. C. Mitchell. Type inference with simple subtype. Journal of Functional Programming, 1(3):245-285, July 1991.

[6]. A. V. Aho, J. D. Ullman. The theory of parsing, translation, and compiling. Volume 1: parsing. Prentice-Hall, Inc., 1972.