Facilitating Formal Specification Acquisition by Using Recursive Functions on Context-Free Languages *

Haiming CHEN and Yunmei DONG
Computer Science Laboratory, Institute of Software, Chinese Academy of Sciences
Beijing 100080, China

Abstract. Although formal specification techniques are very useful in software development, the acquisition of formal specification is a difficult task. This paper presents the formal software specification language LFC, which is designed to facilitate the acquisition and validation of formal specifications. LFC relies on a new kind of recursive functions, i. e. recursive functions on context-free languages, for semantic aspect and uses context-free languages for syntactic aspect of specifications. Specification in LFC and the validation are entirely machine-aided. Specification is mainly facilitated through grammatical learning technique and machine-aided function construction. Validation is facilitated by sample recognition and generation techniques and rapid prototyping technique. A formal specification acquisition system SAQ has been implemented, several non-trivial examples have been developed using SAQ.

1 Introduction

The employment of formal specification techniques in software development is very useful for improving quality and productivity of software. It is particularly important for high confidence software systems. Formal specifications are precise representations of software systems using mathematical notions [20]. Potential benefits of using formal specifications include higher product quality, earlier defect removal, automatic code generation, etc.

While researches have been making progress in software synthesis, verification, testing, and so on, *research in the acquisition of formal specifications has not been keeping pace. Formal specification of software remains an intricate, manually intensive activity* [6]. This difficulty has become an obstacle to the use of formal specifications by a broad range of users. In the literature there have been quite a few different approaches proposed to this issue (e. g. [16, 14, 15, 19, 6]). For example, in some work, methods are explored for inferring domain-specific formal specifications, depending on AI-based techniques and domain knowledge databases. But these are restricted to specific application domains, for example telecommunication. Some other researches provide CASE tools to derive formal specifications by utilizing semiformal representations such as graphical diagrams. However, transformations from informal models into formal specifications, using techniques such as stepwise refinement, are also difficult tasks.

*Work supported by the National Natural Science Foundation of China (Grants Nos. 60103008, 60273023). Correspondence author: Haiming Chen chm@ios.ac.cn.

Besides the many reasons having been mentioned for specification difficulties (see e. g. [14]), we think that lacking enough consideration early in the design of formal specification languages on supporting specification acquisition is also an important reason which has not attracted much attention of researchers. Obviously specification difficulty is closely related to the formal specification language itself. Equally important is the validation of acquired specifications, which confirms that the specifications meet the requirements.

The MLIRF method [7] was proposed to address the above issues. The central idea is to allow the utilization of machine-aided techniques and the reuse of known specification knowledge in the specification process to decrease difficulties. Two powerful formalisms, i. e. context-free (CF) languages and recursive functions, are employed in the method, which enable the use of machine learning and other machine-aided techniques in specification process, while possessing strong expressiveness for representing specifications.[1] For this particular purpose the theory of *recursive functions on CF languages* (CFRF) was proposed [10, 11].

The paper presents a formal software specification language LFC based on MLIRF method. The language is domain independent, however, we think, and just as shown from our experiments, it is most suited for specifying nonnumeric algorithms on complex data objects, especially for algorithms on phrase structures, e. g. graph algorithms, tree algorithms, programming language processing, and so on. Specification in LFC and the validation are entirely machine-aided. Specification is mainly facilitated through grammatical learning technique and machine-aided function construction. For example, grammar of complex data object can be acquired from user-given samples of the object by machine-learning method, and validated by generating new samples and/or parsing other user-given samples. Validation is also performed by testing through rapid prototyping technique. The dynamic behavior of the specifications can be demonstrated by rapid prototyping, hence users can decide if the specifications meet the requirements and detect potential defects in the specifications earlier.

We have developed system SAQ, the Specification AcQuisition interactive environment, to assist users to acquire and validate LFC specifications with support of specification bases. SAQ realizes supporting techniques for LFC, such as specification management, acquisition and validation of specifications, which we briefly overview in this paper and is described in more detail in [9].

The theoretical basis of LFC is the theory of CFRF. The name "LFC" stands for "Language For CFRF". CFRF has the same expressiveness as recursive functions on natural numbers. CF grammars are adopted in LFC as the construct to define data structures. They have been well studied and widely used as a powerful tool in many areas of software.

Section 2 introduces preliminaries needed for the paper, where we briefly review MLIRF method, and describes basic definition forms of CFRF. The main aspects of LFC are described in Section 3. Section 4 outlines the tool SAQ and how to acquire specifications. The process is illustrated by examples in Section 5. Discussion is given in Section 6. A number of non-trivial examples involving several different domains have been developed. These are sketched in Section 7, where we also discuss some problems and suggest directions for future work. Some knowledge about CF language is assumed, for which the reader is referred to [18].

[1] These also make specifications having a functional style and executable. Although there was a debate on whether formal specifications should be executable in the research community some years ago, both kind of specifications are used today.

2 Preliminaries

2.1 An Overview of MLIRF method

The MLIRF method [7] is one for representation, acquisition, and reusing of formal specifications. A main goal of the method is to develop a representation of formal specifications both with high expressiveness and feasible for the acquisition and validation of the specifications.

To express various kinds of data objects used on computers, CF grammars are adopted in the method to describe structures of data objects, i. e. each kind of data objects, called a *concept*, is represented as a CF language. Initially being used to define syntax of programming languages since 1960's, CF grammars have been widely accepted as a powerful and well-studied tool to specify structures of various problems in many areas of software. More recently, CF grammars are used in XML as semi-structured data exchange format.

To express operations or semantics of specifications, recursive functions on CF languages (CFRF) are chosen in the method. It is well known that recursive functions (on natural numbers) have the same expressiveness as Turing machines. It is also proved that CFRF has the same expressiveness as recursive functions on natural numbers [10, 11]. CFRF can be regarded as a natural extension of recursive functions on words (which was first introduced to computer science by J. McCarthy in Lisp [21] and as models of programs [22]) — its domain and range are CF languages, or sets of words with phrase structures.

It is possible to obtain CF grammars by machine learning techniques, which is called *grammatical identification* in the literature. Many learning algorithms for CF grammars exist, however, they either arbitrarily produce a grammar, without meaningful structure, defining the language, or are special for particular subclasses of CF grammars, such as precedence grammar or regular set. On the other hand, we require the grammars to reflect naturally the structures of the data, and the acquisition to work efficiently. For this purpose a reuse-based novel algorithm for CF grammar acquisition was devised [8]. In the algorithm, acquisition of grammars is an interactive process between user and machine, where the user offers samples(sentences) of the grammars and decides whether to accept or reject guesses on request, and the machine infers grammars by inductive learning method and by reuse of specification knowledge (which denotes known grammars here). Details about it can be found in [9]. This not only reduces the difficulty of writing grammars by novices, but also is helpful for defining grammars of data objects whose structures are complex or even unclear at first.

The construction of recursive functions can also be assisted by machine. As we know, there have been studies on how to obtain definitions of recursive functions by machine learning techniques. However, because of the inherent difficulties in the inference of functions, we do not intend to follow this approach. Instead, we take a more practical way, in which machine assists users to inductively construct definitions of functions according to grammars of the concepts. The benefit is, completeness of function definitions is assured, as well as many kind of errors can be detected in the construction by syntactic and context-sensitive analyses.

Validity of CF grammars can be checked by sample recognition and generation techniques, i. e. users may offer new samples conforming to their intention and check by a general CF grammar parser if the samples are valid sentences of the obtained grammar, and, on the other hand, new samples of a grammar can be generated from

the grammar as well for users to check if they are desired.

CFRF can be validated by prototyping techniques. Since CFRF is executable, we already get a prototype after we complete a specification. Through an evaluator of CFRF, we may validate CFRF by testing.

2.2 Basic definition forms of Recursive Functions on CF Languages (CFRF)

2.2.1 Notations and Conventions

Let V_N denote the set of nonterminal symbols, V_T the set of terminal symbols ($V_N \cap V_T = \emptyset$). P denote the set of productions $P = \{X \to \alpha \mid X \in V_N, \alpha \in (V_N \cup V_T)^*\}$.

For any production $Y \to \alpha$, α is called a *term* of Y. A term without any nonterminal symbol is call a *base term*, otherwise it is called a *compound term*.

We can define a CF grammar $G_X = (V_N, V_T, X, P)$ for each nonterminal symbol $X \in V_N$, where X is called the start symbol. G_X produces a CF language written $L(G_X)$. Terms of X are also called terms of $L(G_X)$. Denote $Term(L(G_X)) = Term(X) = \{\alpha \mid X \to \alpha \in P\}$.

2.2.2 Basic definition forms of CFRF

CFRF is recursive function on CF languages, i. e. the domain and range of the function are CF languages. It is impossible to give a formal presentation of the theory here. Instead we briefly introduce the main results in informal manner, with emphasis on the meanings of various definitions. A formal treatment is in [10, 11] and [2].

CFRF consists of two classes of recursive functions: primitive recursive functions CFPRF, and recursive functions CFRF. Functions in CFPRF can be obtained from three initial functions[2], and by a limited times of applications of composition and/or mutual recursion. Mutual recursion has the following form:

Let $f_i : L_1 \times \cdots \times L_n \longrightarrow L$, $i = 1, \ldots, m$, be m functions to be defined, where L_j, L are CF languages. For each $\alpha \in Term(L_1)$, f_i has one equation of the following form:

$$\begin{aligned} f_i(\alpha, y_2, \ldots, y_n) &=_{df} h_{\alpha i}(y_2, \ldots, y_n), \text{ if } \alpha \text{ is a base term; or} \\ f_i(\alpha, y_2, \ldots, y_n) &=_{df} h_{\alpha i}(Z_1, \ldots, Z_r, y_2, \ldots, y_n, \ldots, \\ & \quad f_k(Z_j, y_2, \ldots, y_n), \ldots), \quad \text{if } \alpha \text{ is a compound term.} \end{aligned}$$

where $h_{\alpha i}$ are known functions in CFPRF, $Z_1, \ldots, Z_r$ are the nonterminal symbols occurring in α. f_k $(1 \le k \le m, 1 \le j \le r)$ may occur in $h_{\alpha i}$ only if $L(G_{Z_j}) = L_1$.

The above functions are defined by *structural induction* on CF language L_1. L_1 is called the *inductive language* of the functions. Because there is only one inductive language in mutual recursion, it is also called a *single induction form*.

Functions in CFRF are obtained if the minimization operator is also permitted in the construction of functions. Because this operator is not used in LFC, we do not introduce it in this paper.

The entire theory of CFRF is built upon the above basic operators. It has been proved that CF(P)RF is equivalent to the class of (primitive) recursive functions on numbers [11].

[2] They are the constant, projection and concatenation functions.

3 Main Aspects of the Language LFC

In LFC, structures of concepts are defined by CF grammars, which are represented by a BNF like notation, where each nonterminal symbol denotes the name of a concept. Operation upon concepts is represented by CFRF. A specification is composed of a set of concepts and functions.

3.1 Concept Definition and Sample Representation

Concepts in LFC can be divided into decomposable concept (i. e. non-atomic languages, simply called concept) and indecomposable concept (i. e. atomic languages). Samples of indecomposable concepts are always used as whole, i. e. no structural induction is conducted on these samples when defining functions, and no structural decomposition is conducted when evaluating functions. Predefined indecomposable concepts in LFC are called *basic concepts*.

We require a CF grammar to naturally reflect the inner structure of a data object. Therefore general CF grammars, which have no restriction on grammars, are supported in LFC. On the other hand, in other applications of CF grammars, usually a subclass is used, such as LALR grammar or precedence grammar. Without restrictions, the CF grammars in LFC have not only powerful expressiveness, but also more freedom in the forms of grammars, therefore are able to naturally represent the inner structures of data objects. In implementation a general CF grammar parsing algorithm, such as the Earley's algorithm [13], is required.

In the following example, the concept of binary numbers is defined by the grammar:

```
<Bin>->0|1|<Bin>0|<Bin>1
```

This grammar defines a concept `Bin`, with four productions, saying that a binary number would be either zero, or one, or a binary number with a zero appended, or a binary number with a one appended. Another grammar for binary numbers is

```
<Bin>-><Bit>|<Bin><Bit>
<Bit>->0|1
```

which actually defines two concepts, `Bin` and `Bit`, each has two productions, and `Bin` depends on `Bit`.

Samples or data constants of concepts are represented as character strings quoted by double quotes in function definitions. For example "10" and "100" are valid binary numbers. In evaluation, structure recognition and decomposition of samples of concepts are automatically done by the language's processor[3]. A parser for general CF grammars [13] should be embedded in any implementation of the language.

As mentioned before, the most notable advantage of using CF grammars to represent concepts is that the grammars can be acquired in SAQ from a few samples of concepts. For example, the first of the above grammars for binary numbers can be learned from samples "0", "1", "10", "11". SAQ also provides many other facilities for dealing with CF grammars. By this way, complex concepts can be defined more easily. Of course, user can define simple grammars directly.

[3]In implementation, most of the work can be done at compilation, not evaluation time, to improve efficiency. See Section 3.4

3.2 Function Definition and Evaluation

Although mutual recursion is theoretically sufficient as function definition means, it may sometimes result in tedious and inefficient definitions of functions. So in practice efficient definition means are necessary. In addition, presently there has been no efficient implementation technique for the minimization operator, the need for practical way to define functions in CFRF is also urgent. To address these problems, we proposed several efficient operators. Together with mutual recursion, they can generate quite a lot of function forms. Furthermore, they form a new class of functions which includes CFPRF as its proper subclass. We have proved that this class is equivalent to CFRF [2]. Therefore functions in CFRF can be defined without using minimization operator.

Based on the operators of CFRF having been described, LFC provides complete construction form, direct definition form and partial construction form for function definition. A function has a declaration part (which begins with `dec` and `var`) and a definition part (which begins with `def`). In general, for a n-ary function f with type $L_1 \times \ldots \times L_n \longrightarrow L$, where the L_i and L are CF languages, equations in the definition part of f has the following form

$$\begin{aligned} &f(p_1^1,\ \ldots,\ p_1^n) = e_1 \\ &\cdots \\ &f(p_m^1,\ \ldots,\ p_m^n) = e_m \end{aligned}$$

where each p_i^j is a *pattern* of L_j, which is either a variable of L_j, or corresponds to a term of L_j (which means p_i^j will be a term of L_j after each variable in p_i^j is replaced by its corresponding nonterminal); e_i are expressions.

Functions can also be viewed as being defined by pattern-matching, where patterns correspond to structures of CF languages. According to the definition above, the patterns in functions concerns only the root structures of the parse trees of parameter values, without dealing with nested structures of more than one level. Therefore they are a kind of *simple* patterns. This kind of patterns is suitable for machine-aided construction of functions.

The evaluation rule of the functions is as follows.

When evaluating a function application $f(v_1, \ldots, v_n)$, where v_i are values of L_i, equations of f are examined in a top-down ordering, and the first equation (say, equation i) satisfying the following conditions is selected:

p_i^j either is a variable of L_j, or corresponds to the right-hand side of the first production in the derivation of v_j.

The above rule is just the structural pattern-matching rule of functions. LFC adopts eager or strict evaluation semantics, i. e. the call-by-value manner of function evaluation in which functions are evaluated after all arguments are evaluated.

Note that the order of equations of f is significant only when f is defined in partial construction form, which can be automatically detected. In fact the partial construction form, used for optimal evaluation purpose, is just an abbreviation of non-partial construction form, and can be converted to non-partial construction form when necessary. Thus the existence of partial construction form does not affect any formal treatment to functions.

Completeness of functions can be assured by the machine-aided construction, and can be detected automatically.

LFC supports simultaneous definition of mutually recursive functions. These mutually defined functions always occur as a whole when being defined or used, and are

called a definition group, which is also the basic unit of function access.

As a simple example of function, the following code defines a function that increases a binary number by one, using the first grammar for binary numbers in Section 3.1.

```
dec inc: Bin -> Bin;
var b : Bin;
def inc("0")    = "1";
    inc("1")    = "10";
    inc(b[]"0") = b[]"1";
    inc(b[]"1") = inc(b)[]"0";
```

This function is defined in complete construction form (here is mutual recursion), where `[]` is the infix operator for string concatenation.

To define a function, the cases of each equation can be generated automatically; users only need to offer the definition of each expression. In the above example, the four cases for the inductive language `Bin` are easy to generate from productions. After definition of a function is completed, refinement can be done to merge several equations into one and/or reduce variable numbers if the definition has redundancy. This cases-driven forms the basis of machine-aide construction of functions.

Besides the inductive way, some functions can be defined without induction, for example the above function has an equal version in direct definition form.

```
dec inc: Bin -> Bin;
var b : Bin;
def inc(b) = if eq(b,"0") then "1" else
             if eq(b,"1") then "10" else
             if term(b,"Bin",2) then b[]"1"
             else inc(b)[]"0";
```

In the above definition two built-in functions are used. `eq` determines if two strings are identical. `term` determines if `b` is the third term (terms of a nonterminal symbol are indexed from 0) of `Bin`.

3.3 Other Features

LFC is executable, so it supports rapid prototyping as a way to validate specifications. In function construction, each function group can be tested independently after it is defined. Specifications are developed interactively and incrementally.

Although LFC is based on a new kind of recursive functions, the theory of the recursive functions itself is no prerequisite to users. Like recursive functions on numbers, the theoretical development of CFRF is not easy, but the use of CFRF is not difficult for ordinary users. Users only need to learn basic knowledge of CF languages (which most users are probably already familiar with) and basic skills in using LFC (which is no more than writing mathematical functions), and can be assisted by machine tool, i. e. SAQ, to complete tasks.

3.4 About Implementation

Any implementation of the evaluation of CFRF should combine evaluation and parsing techniques. As mentioned above, since LFC supports general CF grammars, parsing technique for general grammars, such as the Earley's algorithm, is needed.

In principle the evaluation of functions can be straightforwardly implemented by incorporating a general CF grammar parser and performing structure recognition while evaluation, which is not efficient. Several algorithms have been proposed.

[10, 11] proposed a bottom-up, branch-pruning algorithm for CFPRF, which evaluates a group of mutually recursive functions by repeatedly evaluating the values of functions at the nodes of a parse tree in a bottom-up order and deleting evaluated branches, until root is reached. However, this algorithm cannot be extended to CFRF. [1] presented a simple algorithm for evaluation of CFRF with only one inductive language, which has been used in an earlier implementation of LFC. Recently, we have developed a structural evaluation method for CFRF, which makes use of the structural information existed in the definitions of functions, and includes an efficient representation of parse trees [3], type checking and reconstruction of expressions into an intermediate representation reflecting phrase structures [4], pattern-matching compilation [5], and so on. Experiments show that this method can effectively increase efficiency.

Both an interpreter and a compiler of LFC have been implemented. For the limited size of the paper, the implementation of LFC will not be further introduced. An earlier implementation can be found in [9], which includes the implementation of a subset of the LFC described in this paper.

4 How to Work Out LFC Specifications

Specification in LFC is supported by using system SAQ. We have developed both a Solaris version and a Microsoft Windows version of SAQ. An introduction to SAQ can be found in [9]. Here we give a brief view of the system.

SAQ mainly has the following functionalities: specification base management, concept learning and validation, and function construction and validation, together with GUI and online hypertext help.

Grammars of concepts are obtained by using the Concept Learning and Checking subsystem of SAQ. The core of this subsystem is a reuse-based CF grammar acquisition algorithm mentioned in Section 2.1. Samples of the concepts are offered by the user in a form-filling manner, grammars can be learned from the samples automatically or interactively. The acquired grammars then are validated by sample recognition and generation facilities provided by this subsystem. Of course the user can write grammars directly. They also can offer sentential patterns and some other constraints to accelerate the acquisition.

Functions defined on known concepts are constructed by using the Function Construction and Checking subsystem of SAQ. To define a function inductively, the user can select one or more inductive languages in the domain of the function. The subsystem automatically generates left-hand side of each equation of a function according to inductive languages, and interactively request the user to reply the right-hand side in a dialogue manner. The whole definition of a function will be formed after the construction for it is finished. Inter-related recursive functions are supported to be defined simultaneously. Completeness of function definitions can be assured with machine-aided construction. Facilities are provided to do syntactical and many context-sensitive anal-

Table 1: Samples of concepts

Concept Name	Samples
elemFunc	1 1+1
term	1 1*1
factor	1 1^1 -1
primExpr	1 x (1+1) exp(1+1) ln(1+1) sin(1+1) aSin(1+1)
Var	x

yses of functions at the same time in the construction process. Facility is also provided for the user to write or update definitions of functions directly. When a group of mutually recursive functions are constructed, they can be tested by using the interpreter of LFC embedded in the subsystem. The user may then continue to construct and test new functions until all are defined.

Definitions of concepts and functions can be stored in specification bases. They can be reused in subsequent specification processes.

5 Examples

As illustration, this section introduces two examples. Nontrivial examples having been developed with LFC are sketched in Section 7.

Example 1. *The formal differentiation of elementary functions.* This example also appears in [9]. Elementary functions are commonly used mathematic functions, such as polynomial functions, trigonometric functions, anti-trigonometric functions, exponential functions, logarithmic functions, and so on. We will give a specification of this kind of functions and the formal differentiation operation on the functions. We have developed several versions of specifications for this problem, here we give a revised one which is the most simple and concise among them. A more complex one would do some simplification on expressions, resulting simpler expression form as the result of differentiation.

By giving samples listed in Table 1, which are separated by blank space, with one simple concept: `FuncName` given directly as shown below:

```
<FuncName>->exp|ln|sin|cos|tg|ctg|sec
    |csc|aSin|aCos|aTg|aCtg|aSec|aCsc
```

and two known concepts in specification base: `addOp, mulOp`, whose definitions are:

```
<addOp>->+|-
<mulOp>->*|/
```

the learned grammars are defined as follows.

```
<elemFunc>-><term>
<elemFunc>-><elemFunc><addOp><term>
<term>-><factor>
<term>-><term><mulOp><factor>
<factor>-><primExpr>
```

```
<factor>->-<primExpr>
<factor>-><factor>^<primExpr>
<primExpr>-><Num>
<primExpr>-><Var>
<primExpr>->(<elemFunc>)
<primExpr>-><FuncName>(<elemFunc>)
<Var>-><id>
```

To define formal differentiation on elementary functions, we first give the signature of the function : $Diff : elemFunc \rightarrow elemFunc$. Then the interactive construction process of the function begins. Because the concept `elemFunc` has two terms, two cases are generated in the process. After passing several syntactic and semantic checking, the definition is finally as follows.

```
dec Diff: elemFunc -> elemFunc;
var x0,x3 :term;
    x1 :elemFunc;
    x2 :addOp;
def Diff(x0)=tmDiff(x0);
  Diff(x1[]x2[]x3)=scat(Diff(x1),x2,tmDiff(x3));
```

In the definition, `scat` is the built-in function of concatenation.

Since unknown function `tmDiff` is introduced in the above, the process continues, until no unknown function exists. The complete functions are listed in Appendix B.

Example 2. *Check if a sentence belongs to a CF language.*

This requires to define a parser for the CF language. We can make use of the characteristic of LFC to get a simple solution. For a CF language `A`, we define a function `belongA`, which will check if a given string is a sentence of `A`. First we construct the following grammar

```
<GenTyp> -> <A> | <String>
     ... (productions for A is omitted)
```

where `String` is a basic concept representing strings. Then we define the following function

```
dec belongA : GenTyp -> Bool;
var x : A;
    y : GenTyp;
def belongA(x)="True";
    belongA(y)="False";
```

The above definition makes use of the automatic structure recognition function of LFC. If the argument of `belongA` belongs to `A`, `"True"` is returned. Otherwise, because any string is an element of `String`, `"False"` will be returned .

6 Related Work

LFC specifications use a restricted form of equations, i. e. recursive functions. They have similarity with algebraic specifications, which use equation system as the semantics of the specifications. In particular, LFC has some similar aspects with the ASF+SDF formalism [12]. For example, they all use CF grammar as part of the formalism, and are especially suitable for specifying programming language processing. However, from the point of view of specification construction and validation, recursive functions can be constructed more easily in the assistance of machine. On the other hand, equation system may be more appropriate for specifying properties of data types.

There have been other work related to formal specification acquisition, as mentioned in Section 1. Compared with these work, we focus on the acquisition phase, and address the problem of developing a formal specification language to support specification acquisition.

From the point of view of language design, recursive functions are used as computation model in functional languages such as Lisp, ML, and Haskell. Yet there are some major differences between LFC and these languages. They have different design purposes. LFC supports CFRF and CF languages which make machine support for specification possible. LFC is a specification language, and what we are most concerned with in LFC includes the easiness of specification, the expressiveness, completeness of specifications, and the consistency between specifications and requirements. For example, to raise expressiveness, we may introduce expressive operators of CFRF (including quantifiers and bounded minimization currently not in LFC) and other possible formalisms into LFC[4]. On the other hand, in functional languages efficiency and other conveniences for programming are mostly concerned, thus, for example we may introduce imperative components into the languages.

Most functional languages support compound data types in which constructors are utilized to represent data in tree structures. Grammar productions and compound data type definitions have similarities; indeed we have proved they have the same expressive power. However, the former are suitable for external representation of data and the other internal representation of data. For example "abc" is more suitable as external representation than "cons(a cons(b cons (c nil)))", this is especially the case for language processing. In the implementation of LFC, the representation of parse trees of grammars has the same form as the constructor representation of compound data types. Then the implementation techniques for compound data types can be applied to LFC. To support CF languages, a parser is embedded in LFC's processor. This also makes LFC extremely powerful for dealing with syntax-related problems. For example, if we are to write a program in other languages for the example 2 in Section 5, we should either resort to a YACC-like parser generator or write a parser manually.

LFC is directly based on the model of CFRF, which brings some advantages for it. For example, primitive recursive functions are surely terminated in evaluation, and more efficient implementation techniques exist for this class of functions. On the other hand, although functional languages are said to base on and can be translated to λ–calculus, they lack higher level models suited directly for the languages which reflect specific nontrivial features of the languages, e. g. compound data type, pattern matching.

Another formalism based on CF languages is attribute grammars. In an attribute grammar, the grammar serves as the main body, with each production of the grammar attached with a semantic rule. The calculation is carried out by "syntax directed"

[4]This does not necessarily mean LFC is inefficient for implementation.

attribute evaluation. In LFC, grammars specify the types of the objects processed by functions. Therefore LFC and attribute grammars are two different kinds of applications of CF languages, each having its own suitable applications.

7 Concluding Remarks

In this paper, a formal specification language LFC is presented. It is based on recursive functions on CF languages, and uses CF grammars to specify data objects. As far as we know, CF grammars were ever adopted to specify data in only one existing programming language, namely the SNOBOL, known for its power of string manipulation.

We have implemented a supporting tool for LFC, i. e. the formal specification acquisition system SAQ. Several nontrivial examples have been developed using SAQ, including formal specifications for a Java to C++ convertor, a Basic to C convertor, a Lisp interpreter, a music notation system, and formal differentiation of elementary functions. In all of the examples, the specifications were defined in short time. For example, the music notation system, which is composed of 21 concepts and 51 functions, was finished by one student in two weeks. Several pieces of music have also been composed by that system. The results suggest that LFC together with SAQ ease specification acquisition.

Because LFC directly supports operations on CF languages, which means the related operations such as lexical analysis, syntactic analysis and conversion of internal representations of parse tree all are done automatically, LFC is especially competent to problems involving phrase structure processing, such as programming language processing. This is testified by the fact that three of the above nontrivial examples are about programming languages. Furthermore, we expect LFC to apply to more areas. For example, structures of molecules may be represented by CF languages, then chemical reactions may be specified by CFRF. Similarly, LFC may also be applied to molecular biology. LFC may be suited for problems with extremely complex data structures.

One problem with using CF grammars is the ambiguity. If the grammar of a CF language is ambiguous, a sentence of the language may have more than one parse tree, and a function may have different values on the same input. One way to solve this problem is to replace CF languages by regular languages. But this will lose some expressive power. Note some programming languages like C have ambiguous grammars, yet they are widely used in practice. In SAQ, grammars are required to naturally reflect structures of data objects. This is achieved by using the grammar acquisition algorithm of SAQ and by interaction with users. The acquired grammars with natural structures are usually better than arbitrarily selected grammars. Besides this, SAQ utilizes a fixed parsing algorithm, so each sentence will have a unique parse tree, although it is implementation-dependent. Another way is to use generalized LR-parsing technique [17], which covers the full class of CF grammars, and use priority and associativity declarations in grammars.

Currently the patterns of functions in LFC correspond to terms of CF languages to facilitate machine-aided construction of function definitions (see Section 3.2). To further improve efficiency of function definition, extension can be made to patterns so that patterns may correspond to sentential forms of CF languages.

Besides above, future directions of LFC also include: (1) In theoretical aspect, continuing developing efficient operators for CFRF; and (2) adding more built-in types, and type polymorphism. Although LFC presently dose not support polymorphism, it is not difficult to achieve polymorphism in LFC by using parameterized CF grammars.

References

[1] H. Chen. Design and implementation of function construction and checking system FC. (in Chinese) Journal of Software, 9(10), 755–759, 1998.

[2] H. Chen, Y. Dong. Definition forms of recursive functions defined on context-free languages. Technical Report ISCAS-LCS-99-15, Computer Science Laboratory, Institute of Software, Chinese Academy of Sciences, Dec. 1999. (In Chinese)

[3] H. Chen, Y. Dong. A representation of parse tree for context-free language. Journal of computer research & development, 37(10), 2000. (in Chinses)

[4] H. Chen, Y. Dong. 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.

[5] H. Chen, Y. Dong. Pattern matching compilation of functions defined on context-free languages. J. Comput. Sci. & Technol., 16(2), 159–167, 2001.

[6] S. A. DeLoach. A theory-based representation for object-oriented domain models. IEEE Transactions on Software Engineering, 26(6), 500–516, 2000.

[7] Y. Dong. MLIRF method for specification acquisition and reuse. (in Chinese) Proc. of the 9th National Conf. of China Computer Federation. 21–27, May 1996.

[8] Y. Dong. An interactive learning algorithm for acquisition of concepts represented as CFL. J. Comput. Sci. & Technol., 13(1), 1–8, 1998.

[9] Y. Dong, K. Li, H. Chen, et al. Design and implementation of the formal specification acquisition system SAQ. Proceedings of Conference on Software: Theory and Practice, IFIP 16th World Computer Congress 2000. Beijing, 201–211, August 2000.

[10] Y. Dong. Recursive functions of context free languages (I) — The definitions of CFPRF and CFRF. Science in China, Series F, 45(1), 25–39, 2002.

[11] Y. Dong . Recursive functions of context free languages (II) — Validity of CFPRF and CFRF definitions. Science in China, Series F, 45(2), 1–21, 2002.

[12] A. van Deursen. Introducing ASF+SDF using the λ-calculus as example. Chapter 2 of *Executable Language Definitions — Case Studies and Origin Tracking Techniques*, PhD Thesis, University of Amsterdam, 1994.

[13] J. Earley. An efficient context-free parsing algorithm. Comm. ACM. 13, 94–102, 1970.

[14] M. D. Fraser, K. Kumar, and V. K. Vaishnavi. Strategies for incorporating formal specifications in software development. CACM Vol. 37, No. 10, 74–86, 1994.

[15] M. Harada. An automatic programming system SPACE with highly visualized and abstract program specification. IEICE Transactions on Information and Systems, E78-D(4), 403-19, 1995.

[16] M. R. Lowry and R. D. McCartney (eds). Automating Software Design. AAAI Press, 1991.

[17] J. Rekers. Parser Generation for Interactive Environments. PhD thesis, University of Amsterdam, 1992.

[18] G. Rozenberg, A. Salomaa (Eds.). Handbook of Formal Languages. Volume 1 Word, Language, Grammar. Springer-Verlag, 1997.

[19] A. Takura, Y. Ueda, T. Haizuka, and T. Ohta. Requirement specification acquisition of communications services. IEICE Transactions on Information and Systems, E79-D(12), 1716–25, 1996.

[20] J. M. Wing. A specifier's introduction to formal methods. IEEE Computer, 23(9), 8–24, 1990.

[21] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine, Part 1. CACM 2(4), 1960.

[22] J. McCarthy. A basis for mathematical theory of computation. In P. Braffort and D. Hirshberg Ed. Computer Programming and Formal Systems. North-Holland, 33–70, 1963.

A Syntax of the Language LFC

The abstract syntax for a kernel of LFC is as follows.

$$
\begin{array}{lcll}
c & & & \text{Constant} \\
x & & & \text{Variable} \\
f & & & \text{Function name} \\
e & ::= & c \mid x \mid e_1 \cdot e_2 \mid f(e_1, \ldots, e_n) & \text{Expression} \\
 & \mid & \text{IF } e_1\, e_2\, e_3 & \\
 & \mid & e \text{ where } x_1 = e_1 \ldots x_n = e_n & \\
p & ::= & c \mid x \mid p_1 \cdot p_2 & \text{Pattern} \\
prog & ::= & f_1(\overline{p}_{1,1}) = e_{1,1} & \text{Program} \\
 & & \ldots & \\
 & & f_1(\overline{p}_{1,m_1}) = e_{1,m_1} & \\
 & & \ldots & \\
 & & f_n(\overline{p}_{n,1}) = e_{n,1} & \\
 & & \ldots & \\
 & & f_n(\overline{p}_{n,m_n}) = e_{n,m_n} &
\end{array}
$$

where '$\cdot$' is the concatenation operator, $\overline{p}$ denotes a sequence of patterns $p_1, \ldots, p_n$. A pattern corresponds to a term of a CF language (see page 5, Section 3.2).

B Complete functions of formal differentiation of elementary functions (results of machine-aided function construction)

```
dec Diff: elemFunc -> elemFunc;
var x0,x3 :term;
    x1 :elemFunc;
    x2 :addOp;
def Diff(x0)=tmDiff(x0);
   Diff(x1[]x2[]x3)=scat(Diff(x1),x2,tmDiff(x3));

dec efNeg : elemFunc -> elemFunc;
var t : term; e : elemFunc; a : addOp;
def efNeg(t) = tmNeg(t);
    efNeg(e[]a[]t) =
    if eq(a,"+") then scat(efNeg(e),"-",t)
    else scat(efNeg(e),"+",t);

dec ftDiff: factor -> elemFunc;
var x0,x1,x3 :primExpr;
    x2 :factor;
def ftDiff(x0)=peDiff(x0);
    ftDiff("-"[]x1)=efNeg(peDiff(x1));
    ftDiff(x2[]"^"[]x3)=scat(x2,"^",x3,"*(",
       tmDiff(scat(x3,"*","ln(",x2,")")),")");

dec peDiff: primExpr -> elemFunc;
var x0 :Num; x1 :Var;
    x2,x4 :elemFunc;
    x3 :FuncName;
def peDiff(x0)="0";
    peDiff(x1)=if eq(x1,"x") then "1" else "0";
    peDiff("("[]x2[]")")=Diff(x2);
    peDiff(x3[]"("[]x4[]")")=
```

```
if eq(x3,"exp") then
    scat(x3[]x4,"*",Diff(x4)) else
if eq(x3,"ln") then
    scat("(",Diff(x4),")","/","(",x4,")") else
if eq(x3,"sin") then
    scat("cos(",x4,")*",Diff(x4)) else
if eq(x3,"cos") then
    scat("-(","sin(",x4,")*",Diff(x4),")") else
if eq(x3,"tg") then
    scat("sec(",x4,")^2*",Diff(x4)) else
if eq(x3,"ctg") then
 scat("-(","csc(",x4,")^2*",Diff(x4),")")
else if eq(x3,"sec") then
 scat("tg(",x4,")*sec(",x4,")*",Diff(x4))
else if eq(x3,"csc") then
 scat("-(","ctg(",x4,")*csc(",x4,")*",
      Diff(x4),")")
else if eq(x3,"aSin") then
 scat("(",Diff(x4),")/","(1-",x4,"^2)^(1/2)")
else if eq(x3,"aCos") then
 scat("-((",Diff(x4),")/",
      "(1-",x4,"^2)^(1/2))")
else if eq(x3,"aTg") then
 scat("(",Diff(x4),")/(1+",x4,"^2)")
else if eq(x3,"aCtg") then
 scat("-((",x4,"^2*",Diff(x4),
       ")/(1+",x4,"^2))")
else if eq(x3,"aSec") then
 scat("(",Diff(x4),")/(",
       x4,"^2*((1-","1/",x4,"^2)^(1/2)))")
else scat("-",peDiff(scat("aSec(",x4,")")));

dec tmDiff: term -> elemFunc;
var x0,x3 :factor;
    x1 :term;
    x2 :mulOp;
def tmDiff(x0)=ftDiff(x0);
    tmDiff(x1[]x2[]x3)=if seq(x2,"*") then
 scat(tmDiff(x1),"*",x3,"+",x1,"*",ftDiff(x3))
else
   scat("((",tmDiff(x1),")*",x3,"-",
        x1,"*",ftDiff(x3),")/(",x3,"^2)");

dec     tmNeg : term -> elemFunc;
    var t : term;
def     tmNeg( t ) = "-"[]t;
```