Languages and Problem Specification

Loutfi Soufi

Languages and Problem Specification

Loutfi SOUFI
(University of Bourgogne, LIRSIA
Fac. Sciences Mirande BP 47870
21078 Dijon France
soufi@crid.u-bourgogne.fr)

Abstract: Generally, programming problems are formally described as function computation problems. In this paper we want to see programming problems as language recognition problems. More precisely, we sugget to specify the former problems as systems of languages, i.e. recursive languages built from the concatenation and transformation of recursives languages. To understand our grammatical approach we illustrate it by means of typical examples of programming.

Keywords

programming, type specification, program specification, grammar systems, formal language.

1 Introduction

This paper tries to show how programming problems can be described as language recognition problems. Generally, formal methods of program development describe programming problems as function computation problems (see for example [1, 2, ?, 4]). As mentionned, for example, in [11] a function $f : N \to N$ (N being the natural numbers) can be specified by a grammar G. More generally, if f is a function $f : T \to S$, where T, S are any *type*, denumerable domains, then the elements of T and S can be put in a bijective correspondence with the words of Σ^*, i.e words (strings) over Σ (Σ being some alphabet).

Example 1. A type *integer*.
The usual integer type can be specified by $L(G) = \{1^n \mid n \geq 0\}$ where x^i denotes the word obtained by concatenation of x with itself i times, and $x^0 = \epsilon$ (the emtpy word). Thus one can imagine a programming environnment (a mechanical device) in which 1^n corresponds to the integer n.

Example 2. A type function $f : N \to N$
For instance $f(n) = n^2$ can be specified by the context-sensitive language
$L(G) = \{1^n m 1^{n^2} \mid n \geq 0\}$.
"given 1^n interpreted as an integer, the outcome of f is 1^{n^2} (the marker m separates the input from the output).

Our goal is to specify programming problems (problems for short) using formal languages. In our grammatical approach a problem is formally described as a "system of languages", i.e. a recursive language structured in a hierarchical

way. This hierarchy of language is only realized by using the concatenation and transformations of recursive languages and forms what we call a *language of concatenation of level n* depending on the number of transformations applied to a language. Thus we describe a problem P as a recursive set S. Since S is recursive then, by definition [11], its charateristic function is computable. As a consequence we can translate the language S into another formalism for solving P using mechanical devices such as programming (specification) languages, or formal methods of program construction, or the like. Here we do not tackle this translation of formalisms. We are, rather, concentred on the specification of problems using generative devices. Chomsky grammars and the Lindenmayer systems are such devices. Besides these two basic devices we can mention [14, 13, 15, 16, 10]. In this paper we will see how one can specify problems using DOL and *recurrent* sytems by means of typical examples of programming.
The use of generative devices liberates us of any particular syntax which appears in programming (specification) languages. For example there is no syntax of types in a generative device. Since we are liberated of any syntax then we augment our freedom of expressions for specifying problems. Such a freedom of specification (one can define an "exotic" generative device which can be suitably described our problem), and the non-definition of a particular syntax for describing a problem are the main positive aspects of our grammatical approach.
The rest of the paper is organized as follows. In Section 2, we introduce some notations and the DOL and recurrent systems. The section 3 deals with hierarchy of languages. The section 4 gives examples of problem specification. Finally, in Section 5, a short summary is given.

Related Work

At first, we can mention that the presents work is included in the general topic of the relationship between grammar systems and Programming. Examples of grammar systems are [13, 15, 16]. The particularity of our work is the use of a kind of hierarchy languages for specifying programming problems in a structured and modular way.
Finally, our work has nothing to do with work on tree grammars and related topics as discussed in [12]. In our approach a word over an alphabet is not viewed as a term in the algebraic sense.

2 Preliminaries

We use the following general notation: Σ^* (possibly subscripted) is the set of all finite words (sentences, strings) over the alphabet Σ, ϵ is the empty word, and $\Sigma^+ = \Sigma^* - \epsilon$.
If w and v are words $w, v \in \Sigma^*$, then so is their contatenation, written wv or $w.v$.
For further elements of formal language theory we refer to [9, 10]. We suppose that the reader is familiar with the notions of regular expressions, and type-0, type-1, type-2, type-3 languages called respectively recursively enumerable, context-sensitive, context-free, and right-linear (regular) languages.
We recall that a grammar is termed *right-linear* if all productions have the form $A \to x$, $A \to xB$ with A, B non-terminals and x a terminal string. As showed in [8] every right-linear grammar can be transformed into a regular grammar generating the same language. This paper makes indistinct the regular and right-

linear languages.

DOL System

Now, we define a DOL system as given in [10]. First we recall that a mapping $h : \Sigma^* \to \Sigma_1^*$ is a morphism if

$$h(vw) = h(v)h(w), \quad \text{for all words } v \text{ and } w$$

A DOL system is a triple $D = (\Sigma,\ h,\ w)$, where $h : \Sigma^* \to \Sigma^*$ is a morphism and $w \in \Sigma^*$. The system D^i, where i is the number of word to be generated, defines the following sequence $S(D)$ of words:

$$w = h^0(w), h(w) = h^1(w), h(h(w)) = h^2, h(h(h(w))) = h^3, h^4, ...$$

the word h^0 is referred to as the first element of $S(D)$, denoted by D_1, and h^1 as the second, denoted by D_2, and h^2 as the third, D_3, and so on.

The system D defines the following language

$$L(D) = \{h^i(w) \mid i \geq 0\}.$$

Example 3. $D = (\{a, b\}, h, ab)$ where $h(a) = (ab)^2,\ h(b) = \epsilon$
$L(D) = \{(ab)^{2^n} \mid n \geq 0\}$.
$S(D) = ab,\ abab,\ abababab,\ ....$

Word Transformation

By a transformation $\boldsymbol{T}$ defined on Σ and Σ_1 we shall understand a single-valued correspondence between Σ^* and Σ_1^* (where possibly $\Sigma^* = \Sigma_1^*$). It is represented by:

$$[w1/v1, w2/v2, ..., wi/vi]$$

where $wi \in \Sigma^*$ and $vi \in \Sigma_1^*$ and the wis are all distinct and the same holds for the vis and such as we have $[\epsilon/\epsilon]$. Now, its effect on a word $\alpha \in \Sigma^*$ is a word belonging to $\Sigma^* \bigcup \Sigma_1^*$, written $\boldsymbol{T}\alpha$, that is the same as α but with all wis occuring in α replaced by the vis. This simultaneous replacement in α of subwords of α by words is called *word transformation.*

Note:
In general $[w1/v1, ..., wi/vi]\alpha$ is different from $[w1/v1]...[wi/vi]\alpha$, as illustrated by the following two word transformations.

Example 4. $[a/b, b/c]a = b \qquad [b/c][a/b]a = c.$

Recurrent Systems.

We introduce a special symbol @, called the *recurrent word*, in order to characterize regularities occuring in the words of a language. For instance, the words of the precedent language are generated from the word ab: it is a recurrent word. The word $abab$ is also a recurrent word and so on. The idea is when we derive a word w from a word v that means that v must become an occurence (a new subword) of w. The symbol @ is used to denote the derived word to be occured in the next word to be derived.
Formally, a recurrent system is a triple of the form:
$S = (V, R, P)$, where $V = \Sigma \bigcup \{@\}$, R is a finite language over the alphabet Σ,

and P is a finite language whose the words are of the form $u{:}v$, with $u, v \in V^*$. These words drive the derivation of the words of S. We define the relation $\Longrightarrow$ on V^* by

$$w \Longrightarrow z \quad \text{iff} \quad w = u,\ z = [@/w]v \text{ for } u{:}v \in P.$$

Then, the language generated by S is defined by

$$L(S) = \{z \in \Sigma^* \mid w \Longrightarrow^* z,\ w \in R\}.$$

A language generated by a recurrent system is called a recurrent language.

Example 5. The language $L = \{(ab)^{2^n} \mid n \geq 0\}$ can be defined by the recurrent system $S = \{\ \{a, b, @\}, \{ab\}, P\}$ with $P = \{ab{:}@, @{:}@@, @@{:}@@\}$.
$ab \Longrightarrow [@/ab]@ = ab$, now $@ = ab$
$ab \Longrightarrow @ \Longrightarrow [@/ab]@@ = abab$, now $@ = abab$
$ab \Longrightarrow @ \Longrightarrow @@ \Longrightarrow [@/abab]@@ = abababab$
$ab \Longrightarrow @ \Longrightarrow @@ \Longrightarrow @@ \Longrightarrow [@/abababab]@@$

The sequence of words generated by S^i, where i is the number of words to generated, is: ab, the first element, $abab$ the second one, ...We write S_i for the ith word generated by S.
We can put together many recurrent systems in order to form a *complex recurrent system* SS (a *complex* for short) as illustrated in section 4. In this case the ith element of SS is a recurrent system $S(k)^i$. Consequently $S(k)^i_n$ denotes the nth word of the recurrent system $S(k)^i$ of the complex SS. More formally , a complex is a construct of the form
$SS(i) = \{S(1), ..., S(i)\}$ where $S^j = \{V_j, R_j, P_j\}$ for $j = 1, 2, ..i$ are recurrent systems such that $S^k = \{V_k, \boldsymbol{T}\mathrm{k}R_{k-1}, \boldsymbol{U}\mathrm{k}P_{k-1}\}$ for $1 < k \leq i$ and $i > 1$.

The language generated by $SS(i)$ is defined by
$L(SS(i)) = \{w \mid w \in L(S^j)$ for $j = 1, 2, .., i\}$.

Theorem 1. *A language defined by a DOL system is recurrent.*

The proof of this theorem and those given in the next section can be found in [7].

3 Hierarchy of Languages

The goal of this section is to define the languages of concatenation of level n (languages of level n for short). Such languages are used for building languages of type $0 \leq i \leq 3$ from languages of type 3. Before defining what we mean by languages of level n, we come back to the notion of transformation on which these languages are based on.

Transformations

We have already defined what we mean by word transformations. Now, Suppose that we apply $\boldsymbol{T}= [w/v]$ on a word α. It could happen that α contain overlapping occurences of w. We decide that a transformation is to be applied to the leftmost occurence in α reading α from the left to right. Such transformations are called

leftmost transformations. From now we consider only leftmost transformations. We define the *product* of two transformations (do not confuse with the concatenation) $\boldsymbol{ST}$ by $(\boldsymbol{ST})\alpha = \boldsymbol{S}(\boldsymbol{T}\alpha)$, and the *sum* $\boldsymbol{S}+\boldsymbol{T}$ by $(\boldsymbol{T}+\boldsymbol{S})\alpha = \boldsymbol{T}\alpha+\boldsymbol{S}\alpha$ denoting either the word $\boldsymbol{T}\alpha$ or the word $\boldsymbol{S}\alpha$. Thus from such a sum we can construct the set $SUM = \{\boldsymbol{T}\alpha\} \bigcup \{\boldsymbol{S}\alpha\}$. If $\boldsymbol{T}=\boldsymbol{S}$ we can write that $\boldsymbol{T}(\alpha+\alpha) = \boldsymbol{T}\alpha$. More generally we can state that $\boldsymbol{T}(\alpha+\beta) = \boldsymbol{T}\alpha + \boldsymbol{T}\beta$, where $\beta \in \Sigma^*$. Obviously sum is associative and commutative. But the product is not commutative. Furthermore sum and product are not mutually distributive. We have only : $\boldsymbol{T}(\boldsymbol{U}+\boldsymbol{S}) = (\boldsymbol{TU}) + (\boldsymbol{TS})$.
The two operations, product and sum, are referred to as *tranformation operations.*

These operations are extended in a natural way to languages as follows. Let $\boldsymbol{T}$ be a transformation on Σ and $\Sigma 1$. For a language L over Σ we define

$$\boldsymbol{T}L = \{\alpha' \mid \alpha' = \boldsymbol{T}\alpha \text{ for all } \alpha \in L\}$$

(In fact some α can remain unchanged that means ($\boldsymbol{T}\alpha = \alpha$. Such words α belong also to $\boldsymbol{T}L$.

$$[w/L1]L = \{z \mid z = [w/v]x, \text{ for all } x = ywz \in L, \text{ and } v \in L1\}$$

$$[L1/L2] = \{[w/L2] \text{ for all } w \in L1\}$$

$$(\boldsymbol{S}+\boldsymbol{T})L = \{\boldsymbol{T}L\} \bigcup \{\boldsymbol{S}L\}$$

Theorem 2. *For a given transformation $\boldsymbol{T}$ if L is a regular language then $\boldsymbol{T}L$ is a regular language.*

Generalized Transformations

Generalized transformations are transformations with DOL and recurrent systems, i.e transformations of the form either

$[K_i/K_j]$ where K is a DOL or recurrent system,
or $[K_i)/K^j]$ (corresponding to j transformations)
or $[K^i/K^j)]$ (corresponding to $i * j$ transformations)

Theorem 3. *If L is a regular language and $\boldsymbol{T}$ a generalized transformation then L can be transformed into another language*
$\boldsymbol{T}L = \{\alpha' \mid \alpha' = [sk/si]\alpha$ *for all* $\alpha \in L\}$ *of type* $0 \leq i \leq 3$.

Example 6. The language $L1 = \{a^{n^2} \mid n \geq 1\}$ can be constructed from the regular language $L = \{a^n \mid n \geq 2\}$ using the transformations
$\boldsymbol{T}= [b/a]$,
$\boldsymbol{S}= [a/(\{a,b\}, a, h_n)]$, where $h(a) = ab$, $h(b) = b$, and
$\boldsymbol{U}= [a/ab]$.
We can prove that $L1 = \boldsymbol{T}(\boldsymbol{S}L+\boldsymbol{U}\{a\})$.
Given n, $R1 = \boldsymbol{S}L = \{(ab^n)^n \mid n \geq 2\}$, so we have
$\boldsymbol{T}R1 = \{a^{n^2} \mid n \geq 2\}$ and $\boldsymbol{T}\{ab\} = \{a^2\}$.

In the sequel the term "transformation" will also denote generalized transformations.

Structure

We recall that our objective is to express complex languages in terms of simpler ones as illustrated in the above example. The theorem 3 asserts that some languages can be obtained by applying transformations on regular languages. Now we are going to strength that theorem. This strengthening is our main result (theorem 4) and is based on a manner to classify the languages.
We construct hierarchies by considering the concatenation product and the transformation operations only.
A regular expression over Σ^* is termed a *regular expression of level* 0. The regular languages over Σ^* and the empty language are regular expressions of level 0, denoted by $\mathbf{RE(0)}$ (more shortly $\mathbf{RE(0)}$ languages). Obviously, the concatenation of languages $\mathbf{RE(0)}$ is a $\mathbf{RE(0)}$ language.
If E is a $\mathbf{RE(0)}$ language and $\boldsymbol{T}$ a transformation then the language $\boldsymbol{T}E$ is a $\mathbf{RE(1)}$ language. The concatenation of $\mathbf{RE(0)}$ languages and $\mathbf{RE(1)}$ languages is a $\mathbf{RE(1)}$ language (1 is the higher level) in this product of concatenation.
If L is a $\mathbf{RE(1)}$ language then $\boldsymbol{S}L$ is a language $\mathbf{RE(2)}$ and so on. More generally, in a product of languages, the higher level n is determined by a finite combination of n transformation operations. A $\mathbf{RE(n)}$ language is of the form:

$$L1L2...Lk$$

where $L1$, $L2$, ..., LK are $\mathbf{RE(m)}$ languages for $0 \leq m \leq n$, $k \geq 1$ and there exists at least a language Li, $1 \leq i \leq k$, of level n.

Languages transformed into $\mathbf{RE(n)}$ languages, $n \geq 0$, without using intersection, difference, union, and the complementation operations, are called *languages of concatenation of level* n. **Note:**
One can use transformation operations in order to get the same effect that the set operations (union,...). For example let's take the difference operation.
$\{a1, a2, ..., an, b1, ..., bn\} - \{b1, b2, ..., bn\} = \{[b1/\epsilon, ..., bi/\epsilon]\{a1, a2, ..., an, b1, ..., bn\}$.

Theorem 4. *The languages of type* $0 \leq i \leq 3$ *are languages of concatenation of level* n, $n > 0$.

We have not yet an algorithm which transforms a language of type i into a $\mathbf{RE(n)}$ language. However this theorem suggests a bottom-up method. Given a language L we decompose L into $\mathbf{RE(0)}$ languages, $R1$, $R2$, ..., Rk for $k \geq 1$, such that the words of Rj are subwords of words of L and there is a bijection between Rj and L, for $j = 1, 2, ..., k$. Then we construct a language of concatenation of level 1. After that we construct a language of concatenation of level 1, and so on until we obtain the words of L. So at each step i of our construction process of L we build a $\mathbf{RE(i)}$ language.

Example 7. The sensitive-context language $L = \{a^n b^n c^n \mid n \geq 1\}$ can be obtained as follows.
Step 0 from L we choose to extract $L0 = \{a^n \mid n \geq 1\}$.
Step 1 $L1 = [a/(\{a, b\}, h_n, a)]L0$ where $h(a) = ab$,
Step 2 $L2 = [a/b]L1$ and $L2_1 = [b/(\{b, c\}, h_n, b)]L2$ where $h(b) = bc$
Step 3 $L3 = [b/c]L2_1$ and $M = L0.L2.L3$.

By construction we have $M = L$:
From $L0$ we obtain $L1 = \{ab^{n-1}\}$. Then we construct $L2 = \{b^n\}$. .
From $L2$ we obtain $L2_1 = \{bc^{n-1}\}$. Finally we construct $L3 = \{c^n\}$.

4 Examples of Problem Specification

This section presents four examples of specification of programming problems: The factorial, the Fibonacci numbers, the type tree, and the Hanoi towers.

Example 8. The factorial
The factorial problem is specified by the language FAC using a DOL system.
$S = (\{a, b, c\},\ h,\ abc)$ with $h(a) = ab,\ h(b) = b,\ h(c) = abc$

$$FAC = [n/\{\{a, b, c\},\ h_n,\ abc\}]\{n\}$$

For instance the word $abbabc$ in our mind denotes $2 * 1$, and $abbbabbabc$ denotes $3 * 2 * 1$.

Example 9. The Fibonacci numbers
This problem can be specified by the language FIN over $\{f0, f1\}$ $f1,\ f0$ are terminal symbols.

$$FIN = [f0/(\{f0, f1\}, h^i, f0)]\{f0\}$$

where $h(f0) = f1,\ h(f1) = f0.f1$
The elements of FIN are $f0, f1, f0.f1, f1.f0.f1$. ....
The length of a word of FIN is a fibonacci number.

Example 10. How to represent a tree
This problem of representation can be specified by using the recurrent system,

$$S = (\{a, c, e, @\}, \{a\}, ac{:}@a, @a{:}@a, @a{:}@e, @a{:}@c, @e{:}@a, ca, ae{:}\epsilon\}$$

The symbols c, e (opened and closed parentheses respectively) play the role of levels in a tree, and the as are nodes. The node of the root is optional.
The word $(a(aa)a)a$ is an element of $TREE$.
Read the tree from the right to the left. The first a is the node root.

Our goal is to construct $TREE$ by induction on the words. This question is not tackled in this paper. Let us give a second example.

Example 11. The Hanoi towers
We are given n disks of different sizes and three pegs ,0, 1, 2. At the beginning all disks are stocked in decreasing order of size on peg 0. The objective is to move all disks from this peg to another one moving just one disk at a time and never moving a larger disk (greater number) onto a shorter one (lower number). Let us call the shorter disk 1.
We observe that the moves of the disk 1 can be characterized by the word :
$w1 =$ 01 12 20 01 12 20 01 .... such that $|w1| = 2^n$.
"move 0 to 1, no move, move 1 to 2, no move,...
The moves of the disk 2 can be characterized by the word :
$w2 =$ 0022 2211 1100 0022 2211 1100 ... such that $|w2| = 2^n$

$w3 = 00001111\ 11112222\ 222220000\ 00001111...$ such that $|w3| = 2^n$
Now, how to generate the words $w1,\ w2,\ w3, ...$?
We set $T = [0/1, 1/2, 2/0]$, and $S = [0/2, 1/0, 2/1]$. Then
$w1 = 01T(01)S(01T(01))T(01T(01)S(01T(01)))...$
So we can define the words $w1$ by the following recurrent system:
$RR = \{\{0, 1, 2, @\}, \{01, R\}, \{(01, @T(@)), (@T(@), @S(@)), (@S(@), @T(@))\}\}$
$S^1 = \{\{0, 1, 2, @\}, \{R^1\}, \{(00, @S(@)), (@S(@), @T(@)), (@T(@), @S(@))\}\}$ with
$R^1 = \{00\}$.
$S^i = \{\{0, 1, 2, @\}, \{[0/00]R^{i-1}\}, [@(S)@/@(T)@, @(T)@/@(S)@]P^{i-1}\}$
$SS^k = (S^1, ..., S^k)$
$HANOI = [n/RR]\{n\}.[n/SS^n]\{n\}$

The strategy for building languages, for specifying problems is based on inductions to be invented.

5 Conclusion

We have presented a grammatical approach to problem specification based on a notion of hierarchy languages. We have said that any language of type $0 \leq i \leq 3$ can be characterized by languages of concatenation of level n. Then we have illustrated our approach by means of examples.
Finally, our intention using our grammatical approach is to bridge the gap the concepts naturally associated with a problem and those of any particular mechanical devices in which a problem will be solved.

References

1. Bidoit M., Kreowski H., Orejas F., Lescanne P., Sannela D. A Comprehensive Algebraic Approach to System Specification and Development Annoted Bibliography LNCS,1991
2. LUO,Z Program Specification and Data Refinement in Type Theory, Report LFCS Edinbourg,1995,
3. SANNELLA D. and TARLECKI A., Foundations for program development basic concepts and motivation, LFCS Edinbourg Report, 1995
4. PARTSCH,A., Specification and Transformation of Programs, Springer Verlag, 1990
5. SOUFI,L, A language of Transformations, Rapport lirsia Dijon, 1999
6. SOUFI,L, Designing Types as Automata, BCTS15 Keele University April, 1999
7. SOUFI,L, Formal Languages and Programming, draft lirsia , 1999
8. HOWIE,J.M., Automata and Languages, Oxford Science , 1991
9. SALOMAA,A, Jewels of Formal Language Theory, Pitman Publishing , 1981
10. SALOMAA,A, Formal Languages, AP , 1973
11. MANDRIOLI,D. and GHEZZI,C., Theoritical Foundations of Computer Science, John Wiley Sons , 1987
12. COMMON,H. DAUCHET,M. GUILLERON,R. JACQUEMARD,F. LUGIEZ,D. TISON,S. TOMMASI,M. , Tree Automata Techniques and Applications, web Ecole Normale Sup , 1999
13. CSUHAJ-VARJ E.DASSOW J.KELEMEN .JPAUN G., Grammar Systems: A grammatical approach to distribution and Cooperation, Gordon and Breach London , 1994

14. PAUN G.ROZENBERG G.A.Salomaa Grammars Based on the Shuffle Operation Journa Universal Computer Science, 1995
15. DASSOW J.PAUN G.ROZENBERG G., title =Grammar Systems, Chapter in G.Rozenberg and A. Salomaa Handbook of Formal Languages Springer-Verlag , 1998,
16. Workshop on Grammar Systems, Opava Gzech Republic , 1997