Journal of Universal Computer Science, vol. 5, no. 9 (1999), 622-631
submitted: 10/6/99, accepted: 15/7/99, appeared: 28/9/99 © Springer Pub. Co.

Type Specification by Regular Expressions

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 they are viewed as language recognition problems. More precisely, we sugget to specify types, and programs using the concept of languages of concatenation of level n, i.e, languages built from regular languages on which language transformation operations are applied to them. Regular languages denoted by regular expressions allow an easy connection between those languages of concatenation and programming. This connection is naturally done via recurrence relations. We explain our approach through some examples.

1 Introduction

The formal methods of program development, as for example the algebraic methods, (see the different approaches in [San91, LUO95, ST95, PAR90]), describe programming problems as function computation problems. They propose specification and/or programming languages incorporating mechanisms and concepts (such as parametrisation, modularisation..., and subtyping, dependent types, higher-order functions etc.) for specifying and implementing data types and functions. We can view these languages, and also those formal methods as "mechanical" devices to solve programming problems.
This paper suggests to describe programming problems as language recognition problems (in the sense of formal language theory). Such languages could be transformed into executable functions by mechanical devices. As mentionned in [MG87] a function $f : N \to N$ (N being the natural numbers) can be specified by a grammar G. Consequently a type (program) specification can be stated as a problem of recognizing a language in Σ^* (Σ being some alphabet). Indeed, generally, in programming a type is a collection of elements. We can put these elements in a bijective correspondence with the words of Σ^*, i.e strings over Σ (see again [MG87] for more details).

Let us illustrate our talk by the following simple examples.

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 *directory*
The elements of this type can be given by the words of
$L(G) = \{(1^n a)^k \mid n > 0,\ k \geq 0\}$.
In our mind 1^n is an integer, and a a string of characters.

Example 3. A type function $f : N \rightarrow N$
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).

Generally, in programming one, gradually, specify complex types in a structured and modular way. This way is reflected in our grammatical approach to type specification. Our approach consists in specifying types as a "system" of languages based on regular languages. It proposes to build complex languages up from regular languages as stated by the theorem 6 (section 3) which is our main result. It suggests to construct languages from simpler ones using what we call *word transformations.* These transformations explicit the construction steps of a language:
given a complex language, C, we extract regular languages (such an extraction is decidable). Suppose that we extract only one language L such that the words of L are subwords of words of C and there is a bijection between M and L. The problem, now, is to reconstruct C from L. Then we apply to it a transformation, $\boldsymbol{T}1$. We then obtain a new language $\boldsymbol{T}1L$ (not necessarily regular). To $\boldsymbol{T}1L$ we apply $\boldsymbol{T}n$: we, obtain the language $\boldsymbol{T}2(\boldsymbol{T}1L)$, and so on. At the end of this construction process we get a language $M = \boldsymbol{T}n...(\boldsymbol{T}2(\boldsymbol{T}1L))$. The transformations are choosen in such a way that one can prove that $M = L$.

Since the regular languages are very simple then their use and transformation into another languages make easy and natural the specification of types as formal languages. They facilitate the translation of type specifications into objects of a mechanical device. Here, we propose to translate the former specifications into recurrent specifications.
Now, if we have a tool for constructing type specifications in our sense then it will become possible to make accessible programming to unskilled persons. This is our hope in the use formal languages for describing programming problems as language recognition problems.

The rest of the paper is organized as follows. In Section 2, we introduce some notations and the DOL systems. The section 3 deals with hierarchy of languages. In section 3, we discuss type specification and give the example of a tree. The section 4 shows how type specifications can be represented by recurrent specifications. 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 [CS94, DAS98, WOR97]. The particularity of our work is the use of a kind of hierarchy languages for specifying types in a structured and modular way.
The expressive power of the recurrence relations, and of the induction as well are well known [DB95, GKP94, LOC70]. That is why we suggest the use of recurrent specification in our approach. Finally, our work has nothing to do with work on tree grammars and related topics as discussed in [TOM99]. 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, $\epsilon =$ is the empty word, and $\Sigma^+ = \Sigma^* - \epsilon$.
If w and v are words $w, v \in \Sigma^*$, then so is their contatenation, written xy or $w.v$. For further elements of formal language theory we refer to [SAL81, SAL73]. We suppose that the reader is familiar with the notion 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 $A \rightarrow x,\ A \rightarrow xB$ with A, B non-terminals and x a terminal string. As showed in [HOW91] 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.
It is well known that regular languages are denoted by regular expressions. For instance, the regular expression $a(bb)^*$ denotes the language $\{ab^{2i}\} \mid i \geq 0\}$.
Now, we define a DOL system as given in [SAL73]. First we recall that a mapping $h : \Sigma^* \rightarrow \Sigma_1^*$ 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 $h : \Sigma^* \rightarrow \Sigma^*$ is a morphism and $w \in \Sigma^*$. The system D 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 h_0, and h^1 as the second, denoted by h_2, and h^2 as the third, h_3, and so on.

The system D defines the following language

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

Example 4. $D = (\{a, b\}, h, ab)$ where $h(a) = (ab)^2,\ h(b) = \epsilon$
$L(D) = \{(ab)^{2n} \mid n \geq 0\}$.
$S(D) = ab,\ abab,\ abababab,\ ....$

The proofs of theorems given in the next section can be found in [SOU99b].

3 Hierarchy of Languages

The goal of this section is to define the languages of concatenation of level n, for $n \geq 0$ in order to build languages of type $0 \leq i \leq 3$ from languages of type 3 (regular languages). First we introduce the notion of transformation and we explain how to structure languages as languages of concatenation of level n. We conclude this section by a theorem related to this structure.

Transformations

By a transformation $\boldsymbol{T}$ defined on Σ and $\Sigma 1$ we shall understand a single-valued correspondence between Σ^* and $\Sigma 1^*$ (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*. We define the *product* (do not confuse with the concatenation) $\boldsymbol{ST}$ by $(\boldsymbol{ST})\alpha = \boldsymbol{S}(\boldsymbol{T}\alpha)$. In general $[w1/v1, ..., wi/vi]\alpha$ is different from $[w1/v1]...[wi/vi]\alpha$, as illustrated by the following two word transfromations:

Example 5. $[a/b, b/c]a = b \qquad [b/c][a/b]a = c.$

The *sum* $\boldsymbol{S}+\boldsymbol{T}$ is the transformation such as $(\boldsymbol{T}+\boldsymbol{S})\alpha = \boldsymbol{T}\alpha+\boldsymbol{S}\alpha$ 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 have $\boldsymbol{T}(\alpha + \alpha)$ or more generally $\boldsymbol{T}(\alpha + \beta) = \boldsymbol{T}\alpha + \boldsymbol{T}\beta$, 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 reffered to as *tranformation operations*.

Leftmost Transformations

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. For a language L over Σ and given a transformation $\boldsymbol{T}$, on Σ and $\Sigma 1$ 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 $(\boldsymbol{T}\alpha = \alpha$. And also these words α belong to $\boldsymbol{T}L$. In addition to that it is important to note that $\boldsymbol{T}$ is the letfmost transformation).

Theorem 1. *For a given transformation $\boldsymbol{T}$ if L is a regular language then $\boldsymbol{T}L$ is a regular language.*

Corollary 2. *If a language L is obtained from regular languages by transformation operations then L is regular.*

Proof. The product is an associative operation and, consequently according to the theorem 1 we obtain a regular language. Given a regular language, since the sum corresponds to the union of two regular languages then we obtain a regular language:

$(\boldsymbol{S}+\boldsymbol{T})L = \{\alpha' \mid \alpha' = \boldsymbol{T}\alpha \text{ for all } \alpha \in L\} \bigcup \{\alpha'' \mid \alpha'' = \boldsymbol{S}\alpha \text{ for all } \alpha \in L\}$.

Generalized Transformations

Generalized transformations are transformations with DOL systems, say D, i.e transformations of the form either

$[(\Sigma, v, h_k)/(\Sigma, v, h_i)] = [sk/si]$ where sj is the jth element of the sequence $S(D)$, or $[(\Sigma, v, h_k)/(\Sigma, v, h^i)] =$

$[(\Sigma,v,h_k)/(\Sigma,v,h_i)][(\Sigma,v,h_k)/(\Sigma,v,h_{i-1})]...[(\Sigma,v,h_k)/(\Sigma,v,h_1)]$
or $[(\Sigma,v,h^k)/(\Sigma,v,h^i)] =$
$[(\Sigma,v,h_k)/(\Sigma,v,h^i)][(\Sigma,v,h_{k-1})/(\Sigma,v,h^i)]...[(\Sigma,v,h_1)/(\Sigma,v,h^i)]$

Note 3. A DOL system, D, in a transformation is such that $L(D)$ defines a regular language. As proved in [SAL73], given a regular language, R, it is decidable if $L(D)$ is contained in R.

Theorem 4. *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* $1 \leq i \leq 3$.

Note 5. About the theorem 4, we do not know if $\boldsymbol{T}L$ can be a language of type0.

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. Consequently the corollary 2 holds if the transformations are not 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 4 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 6) 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 **RE(0)** (more shortly **RE(0)** languages). Obviously, the concatenation of languages **RE(0)** is a **RE(0)** language.
If P is a **RE(0)** language and $\boldsymbol{T}$ a transformation then the language $\boldsymbol{T}P$ is a **RE(1)** language. The concatenation of **RE(0)** languages and **RE(1)** languages is a **RE(1)** language (1 is the higher level) in this product of concatenation.
If L is a **RE(1)** language then $\boldsymbol{S}L$ is a language **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 **RE(n)** language is of the form:

$$L1L2...Lk$$

where $L1$, $L2$, ..., LK are **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 6. 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. $\{a1, a2, ..., an\} - \{b1, b2, ..., bn\} = \{[b1/\epsilon, ..., bi/\epsilon](A + B)\}$.

Theorem 7. *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 $L3 = \{c^n\}$.

Now we are in the position to give a relationship between the languages of concatenation of level n and programming.

4 Type Specification

This section deals first with the use of $\mathbf{RE(n)}$ languages to specify types, and computation and then with the representation of type specifications by *recurrent specifications*.

A type (a program) is just a collection of elements (all the possible outcomes) . In our grammatical approach, elements (outcomes) of types (programs) are seen as words of languages. Consequently, type (program) specification become language recognition problems. Thus in our approach “type specification” and “program specification” are interchangeable. We will use the term “type specification”.
A type specification is given by a language of concatenation of level n, $n \geq 0$.

Example 8. A type $TREE$ over $\{a, (,)$ can be specified by the following structure. Let R be a type: $R = \{((a^+)^+\}$ R is a $\mathbf{RE(0)}$ language.

$TREE = R.[a/(\{a\}, h^i, a)][(/)]R$
where $h(a) = aa, h(a) = \epsilon, h(a) = a$.

The parentheses play the role of levels in a tree, and the as 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 9. A type FIN (for the Fibonacci numbers) 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.

The connection between languages of concatenation of level n and programming is done via recurrence relations. These relations form what can be called a *recurrent specification*.

Recurrent Specifications

To construct a recurrent specification from a type specification in a systematic way we first adopt the following convention:

A symbol formed by capital letters, say T, denotes a type specified by a language. When we rewrite such a symbol in lower-case with a barre then it denotes any element of T, written $\bar{t}$. We call barred symbols *typed symbols*.

This convention allows us to use typed symbols as terminal symbols. To establish recurrence relations we introduce special identifiers called *linear symbols*.
A *linear symbol* is an identifier of the form $a1{:}a2{:}...{:}an$ for $n > 1$ where "**:**" is a linear symbol constructor and each ai denotes either an identifier, elements of types, or typed symbols. The special symbol ϵ is the neutral element of "**:**": $u{:}\epsilon$ is equivalent to u and to $\epsilon{:}u$. Now, we can translate words of a type into linear symbols.

Example 10. A type directory $DIR = \{empty(t\ \overline{name}\ \overline{tel})^*\}$, where *empty* and t are terminal symbols, can be translated into the recurrence relation :

$$dir_i \ == \ dir_{i-1}{:}t{:}\overline{name}{:}\overline{tel}$$

(Read: The words of DIR are defined by $dir_0,\ dir_1, ...,\ dir_n$).
One can define $dir_0 == empty$. Thus *empty* is the first value computed by means of these recurrence relations; dir_1 defines the word $empty{:}t{:}\overline{name}{:}\overline{tel}$, dir_2 the word $empty{:}t{:}\overline{name}{:}\overline{tel}{:}t{:}\overline{name}{:}\overline{tel}$, and so on.
The type DIR is built from the types $NAME$ and TEL. Thus we have
$DIR \subseteq \{emty, t\}^* \bigcup NAME \bigcup TEL$.

Now, we give the construction rules of a recurrent specification from a type specification.
Let S be type specified by a language L. We denote by ω the first value of recurrence relations.

$$R1 \qquad S = L \rightarrow S =_{rep} S$$

(Read $A \rightarrow B$: A is translated into B. Thus $R1$ says that:
"the type specification S, specified by L, is represented by the recurrent specification S -the names should be the same-."
It remains to see how to translate L into a recurrent specification S.

$$R2 \qquad S = \{\alpha^*\} \rightarrow s_0 == \omega,\ s_i == s_{i-1}\mathbf{:}\alpha$$

This rule establishes a relationship between regular expressions of the form α^* and recurrence relations as follows:
α^0 corresponds to s_0 the initial value,i.e. ω
α^1 corresponds to the first value, s_1, generated by means of those relations, i.e. $s_{i-1}\mathbf{:}\alpha$, and so on.

Note 8. We do not write $\overline{s_i}$ since s_i denotes a particular word of S.

$$R3 \qquad S = L.R \rightarrow s == \overline{l}\mathbf{:}\overline{r}$$

Parametrized Recurrent Specification
A transformation of a language, $R = \boldsymbol{T}L$, means that R depends on L and, consequently the recurrent specification L becomes a *parameter* of R.

$$R4 \qquad R = \boldsymbol{T1T2}...\boldsymbol{Tn}L \rightarrow$$
$$Pn = \boldsymbol{Tn}L, Pn-1 = \boldsymbol{Tn-1}Pn, ..., P1 = \boldsymbol{T1}P2$$

$$R =_{rep} \ R(Pn, Pn-1, ..., P1)$$
$$r == \overline{pn}\mathbf{:}\overline{pn-1}\mathbf{:}...\mathbf{:}\overline{p1}$$

$$Pn =_{rep} \ Pn(L)$$
$$pn == \boldsymbol{Tn}\overline{l}$$
....

$$P1 =_{rep} \ P1(P2)$$
$$p1 == \boldsymbol{T1}\overline{p2}$$

Read: The type specification R parametrized by the type specifications $P1, ...Pn$ is represented by the parametrized recurrent specification, $R(P1, ..., Pn)$ where $P1, ...Pn$ are now recurrent specifications.

Note 9. The transformations and the trasformation operations are used as primitive operations in the recurrent specifications.

Example 11. The type specification $TREE$ is translated into the recurrent specification $TREE$ as follows.

$$TREE =_{rep} \ TREE(P1, P2)$$
$$tree == \overline{r}\mathbf{:}\overline{p1}\mathbf{:}\overline{p2}$$

$$P1 =_{rep} \ P1(R)$$
$$p1 == [(/)]\overline{r}$$

$$P2 =_{rep} \ P2(P1)$$
$$p2 == [\overline{a}/(\{\overline{a}\}, h^i, a)]\overline{p1}$$

$$R =_{rep} \ R$$
$$r_0 == \epsilon, \quad r_i == ri-1\mathbf{:}(\mathbf{:}\overline{a}\}$$

In this recurrent specification we have left unspecified the type A $(\overline{a})$. We come to realize a polymorphism.
The polymorphism rule ($R5$): In a recurrent specification any typed symbol not specified can be specified by any recurrent specification.

About the correctness

The repetitive application of the above four rules ($R1$, $R2$, $R3$, $R4$) assure the correctness of a recurrent specification wrt a type specification.

Proof. It is trivial. Our rules are purely syntactic, except the rule $R2$ which is rather a semantical rule. Concerning $R2$ we have already justified the connection between regular expressions of the form a^* and the recurrence relations.

In order to experiment the representation of type specification we defined a language for writing recurrent specifications [SOU99c, SOU99a].

5 Conclusion

We have presented a grammatical approach to type 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 used these languages to specify types, i.e. collection of elements. We saw how to translate type specifications into recurrent specifications using syntactical rules. Finally, we hope to bridge the gap between the intuitive concepts of a programmer and the concepts defined in any particular programming (specification) languages using formal languages.

References

[CS94] CSUHAJ-VARJ E.DASSOW J.KELEMEN .JPAUN G. Grammar systems: A grammatical approach to distribution and cooperation. *Gordon and Breach London*, 1994.

[DAS98] DASSOW J.PAUN G.ROZENBERG G. Grammar systems. *Chapter in G.Rozenberg and A. Salomaa Handbook of Formal Languages Springer-Verlag*, 1998.

[DB95] H. DOORNBOS and R.C. BACKHOUSE. Induction and recursion datatypes. *LNCS 947*, 1995.

[GKP94] R. GRAHAM, D. KNUTH, and T. PATASHNIK. Concrete mathematics a fundation for computer science. *Addison-Wesley*, 1994.

[HOW91] J.M. HOWIE. Automata and languages. *Oxford Science*, 1991.

[LOC70] BELLMAN R. COOKE K. LOCKETT J.A. Algorithms, graphs, and computers. *Academic Press*, 1970.

[LUO95] Z LUO. Program specification and data refinement in type theory. *Report LFCS Edinbourg*, 1995.

[MG87] D. MANDRIOLI and C. GHEZZI. Theoritical foundations of computer science. *John Wiley Sons*, 1987.

[PAR90] A. PARTSCH. Specification and transformation of programs. *Springer Verlag*, 1990.

[SAL73] A SALOMAA. Formal languages. *AP*, 1973.

[SAL81] A SALOMAA. Jewels of formal language theory. *Pitman Publishing*, 1981.

[San91] Bidoit M., Kreowski H., Orejas F., Lescanne P., Sannela D. . A comprehensive algebraic approach to system specification and development annoted bibliography. *LNCS*, 1991.

[SOU99a] L SOUFI. Designing types as automata. *BCTS15 Keele University April*, 1999.

[SOU99b] L SOUFI. Formal languages and programming. *draft lirsia*, 1999.

[SOU99c] L SOUFI. A language of transformations. *Rapport lirsia Dijon*, 1999.

[ST95] SANNELLA D. and TARLECKI A. Foundations for program development basic concepts and motivation. *LFCS Edinbourg Report*, 1995.

[TOM99] COMMON,H. DAUCHET,M. GUILLERON,R. JACQUEMARD,F. LUGIEZ,D. TISON,S. TOMMASI,M. . Tree automata techniques and applications. *web Ecole Normale Sup*, 1999.

[WOR97] WOR. Workshop on grammar systems. *Opava Gzech Republic*, 1997.