Document Semantics: two approaches

José Carlos Ramalho
jcr@di.uminho.pt

José João Almeida
jj@di.uminho.pt

Pedro Rangel Henriques
prh@di.uminho.pt

Universidade do Minho
Braga – Portugal
October 14, 1996

Contents

Abstract

SGML introduced DTD idea to formally describe document syntax and structure. One of its main characteristics is the fact of being purely declarative and fully independent of the future document's processing (typesetting, formatting, translation/transformation). In this context, SGML has become the international standard to be followed.

Sooner or later, a document has to be processed. In order to do that we need to associate semantics to the document's structure. In a compiler context, normally we separate semantics in two, static and dynamic. Establishing a parallelism with document processing, we can think of the document's decorated tree (as recognized by an SGML analyzer) as representing the static semantics and document's tree transformation as dynamic semantics.

Pursuing this idea, we will present and discuss a study of the relationship between SGML, DAST (Decorated Abstract Syntax Tree), and Algebraic Specification, in order to better understand how to formally process documents and how to specify and build generic document processing tools.

1 Introduction

In this paper we will be concerned with document semantics and document processing. We will explore this subject in different directions.

By description of document dynamic semantics we mean the association of transformations to the document that will enable to generate some other form of the document that will serve some specific purpose.

Document semantics is becoming a relevant issue due to the growing generalized used of SGML. SGML introduced the concept of document type and the formal specification of a document's structure. One of its main characteristics and advantages is the fact of being purely declarative and fully independent of the future document's processing. This also means that in order to do something with a specific SGML document one has to be able to associate behaviour to the document. That is what almost every SGML editor allows the user to do — to specify a style sheet, and associate it with a DTD. The problem is that each editor has its own style specification language, instead of following a standard (like SGML).

To put some order in this world the ISO committee launched a new standard, Document Style Semantics Specification Language (DSSSL), which many people is trying to implement.

In the following sections, after a brief review of basic concepts, we will present a case study — the literate programming example. We will use this example through the document to present, analyze and compare two approaches to the specification of document semantics.

2 Background

We begin with a brief knowledge introduction, necessary to understand the subsequent discussion.

2.1 Decorated Abstract Syntax Trees

An Attribute Grammar (AG in the sequel) is an well accepted formalism used by the compilers' community to specify the syntax and semantics of languages. Introduced by Knuth [Knu68], the AG appeared as an extension to the classic Context Free Grammar (CFG for short) to allow the local definition (without the use of global variables) of the meaning of each symbol in a declarative

style. Terminal symbols have intrinsic attributes (that describes their lexical information) and Nonterminal symbols are associated with generic attributes; semantic information can be synthesized up the tree (from the bottom to the root), but can also be inherited down the tree (from the top to the leaves), enabling explicit references to contextual dependencies.
Let G be a CFG defined as a tuple $G = < T, N, S, P >$, where:

- T denotes the set of terminal symbols (the alphabet)
- N is the set of nonterminal symbols
- S is the *start symbol*, or *grammar axiom*: $S \in N$
- P is the set of *productions*, or derivation rules, each one of the form

  $$A \to \delta,\ A \in N\ \wedge\ \delta \in (T \cup N)^*$$

  that we will represent in the sequel as

  $$X_0 \to X_1\ X_2\ \cdots\ X_n$$

For each *abstract production* $p \in P$ —a derivation rule without *keywords* in its *rhs*[1]— there is an associated *abstract tree* whose *root* is X_0, the nonterminal in its *lhs*[2], and its n descendents are the $X_i,\ i \geq 1\ \wedge\ 1 \leq n$ symbols in the *rhs*.
Given a sentence of $\mathcal{L}_\mathcal{G}$, the language generated by grammar G, an *Abstract Syntax Tree* (AST)[3] is a tree whose *root* is S, the grammar start symbol, and the *frontier* is composed by the terminal-classes that once concatenated from left to right form the given sentence. The AST is built pasting the *abstract trees* corresponding to each grammar rule $p \in P$ used to derive the sentence from the *axiom* S. An Attribute Grammar AG is a tuple $AG = < G, A, R, C >$, where:

- G is a CFG as defined above
- A is the union of $A(X)$ for each $X \in (T \cup N)$, and denotes the set of all attributes (each one has a name and a type); the attributes of a terminal symbol are called *intrinsic* and their value do not need to be evaluated; for each nonterminal X, its set of attributes $A(X)$ is splitted into two disjoint subsets: the *inherited attributes* $AI(X)$, and the *synthesized attributes* $AS(X)$
- R is the union of R_p, the set of *attribute evaluation rules* for each production $p \in P$
- C is the union of C_p, the set of *contextual conditions* for each production $p \in P$

[1] Right Hand Side of the production.
[2] Left Hand Side of the rule.
[3] Also called, *Abstract Derivation Tree.*

Let p be a CFG production ($p \in P$), R_p its set of *attribute evaluation rules*, and C_p its set of *contextual conditions*; in this context:

- $a(X_i)$, $i \geq 0$, represents the attribute a associated to the symbol X that occurs in position i of production p
- an *attribute evaluation rule* is an expression of the form

  $$a(X_i) = fun(\cdots, b(X_j), \cdots)$$

  with: $a \in AS(X_0) \ \vee a \in AH(X_i),\ i > 0$; $b \in AH(X_0) \ \vee b \in AS(X_j),\ j > 0$; and fun any function of type V_a (the type of attribute a)
- a *contextual condition* is a predicate of the form

  $$pred(\cdots, a(X_i), \cdots),\ i \geq 0$$

Given a sentence of $\mathcal{L}_{\mathcal{AG}}$ —the language generated by the attribute grammar AG— let AST be its *Abstract Syntax Tree*, as defined above.
Each tree *node* is, originally, labeled by the corresponding grammar symbol and the identifier of the production that was applied to derive it. Assume, now, that each tree *node* is enriched with the attributes (either *inherited* or *synthesized*) associated to that symbol.
A *Decorated Abstract Syntax Tree* (DAST) is the initial AST after the *attribute evaluation process*, with all the attribute occurrences in each *node* associated with an actual value (belonging to its proper domain). To be a DAST it is necessary that all the *contextual conditions*, related to the production labeling each node, are satisfied[4].

2.2 Algebraic Specification: CAMILA approach

From school physics we got used to a basic problem solving strategy: *create a mathematical model, reason on it, calculate a solution.* The CAMILA approach is an attempt to make such a strategy available at the software engineering level. Based on a notion of *formal software component* it encompasses a set-theoretic notation, a prototyping environment, fully connectable to external applications and equipped with communication facilities, and an inequational refinement *calculus.*

CAMILA aims to be both a learning tool for Computer Science students and a working tool for software engineers. At the first level it provides a smooth way to programming. At the second a rigorous way to develop complex systems and to promote the use of formal methods in software industry.

CAMILA[5] was originally devised as a collection of interrelated support tools for teaching different parts of the Computer Science and Software Engineering

[4] Evaluate to `true` for the actual attribute occurrence values.

[5] CAMILA is named after a Portuguese 19^{th}-century novelist — Camilo Castelo-Branco (1825 - 1890) — whose immense and heterogeneous writings, deeply rooted in his own time experiences and controversies, mirrors a passionate and difficult life.

curricula. The project affiliates itself, but is not restricted to, to the research in exploring Functional Programming as a *rapid prototyping* environment for formal software models, whose origin can be traced back to P. Hendersen's me too [Hen84].

In the way, some new theoretical and technological results — namely a component classification and reification calculus and a notion of connectable high-level prototyping environment — were achieved and incorporated in the project.

As a working tool for software engineers it offers a simple set-theoretic notation and a fully connectable environment. As a learning tool supporting a Computer Science curriculum, it aims to be easy to understand and use, and to stimulate a kind of abstract and compositional reasoning which paves the way to sound methodological principles.

The CAMILA platform is organized around 5 main components:

- An executable (functional) specification language directly based on *naive* set theory.
- An inequational calculus [Oli90, Oli92] — SETS — for refining and classifying software formal models. In particular it enables the synthesis of target code programs by transformation of the initial specifications.
- A flexible rapid prototyping kernel which bears "full citizenship" at C/C++ programming level (C may call CAMILA services and CAMILA may also invoke external C functions). It is available at both UNIX, LINUX and MS/DOS operating systems and may provide services under X-WINDOWS or as a WINDOWS 3.1 DLL. Furthermore the prototyping environment provides a set of communication facilities to animate systems built by composition of independent and concurrent software components.
- A formal software components *repository* which catalogues available models and a compositional notation based on "software-circuit" diagrams (a shorthand for some piece of mathematics), suggestively resembling the conventional hardware notation.
- An approach to the specification and generation of structural Human-Machine Interfaces, independent of but mirroring the application semantics.

The CAMILA approach to programming technology claims to provide a smooth way to teaching and using (constructive) formal methods in software engineering. Its roots on functional prototyping of information models [Hen84] has already been referred. Similar motivations may be found either in the research on *formal specification methods*, such as VDM, Z, RAISE [Hax90], COLD-K [FJ92] or LARCH [GH93], or on *functional programming languages* such as ML [HM86] or MIRANDA [Tur86].

In contrast with the former group one could stress the lighter notation of CAMILA, borrowed from set theory, and the direct correspondence to the prototyping language. But what is, to our knowledge, new is the associated calculus

for model reasoning and refinement. On the other hand, CAMILA lacks the sophisticated interface and documentation management features available, for instance, in RAISE.

CAMILA, or at least its prototyping language, may also be compared with other functional languages which achieved a high degree of clarity and expressive power. Although some features of more elaborated languages (*eg*, effective polimorphism) are absent in CAMILA, we would point out as original features CAMILA's flexibility in being fully connectable to other "galaxies" of the computation universe and easily suited to different application domains.

A brief summary of CAMILA language can be found in appendix A. For a more detailed description look in [BA95] lecture notes.

3 The study of a case

To illustrate our ideas we will use a case study, very dear to us (it is good to demonstrate several aspects of the problem), the literate programming type of document [Knu92]. This type of document was created to mix in the same text file programs and reports, providing the necessary commands to distinguish those two kinds of components so that it would be possible to extract the relevant parts later on.

Though, a literate programming document is a text with a mixed content of special text fields (like title and section), program code and definitions. A definition is an association between an identifier and piece of text/code; a program can embed references to the defined identifiers.

Our text example contains a small DTD and short text written according to the former DTD.

The DTD below defines formally the structure of a *literate programming type of document*.

```
<!doctype litprog -- public or system --
[
<!element litprog  - -  ((#PCDATA | prog | def | id | sec | tit)*) >
<!element prog     - -  ((#PCDATA | id)*) >
<!element def      - -  (prog) >
<!attlist def ident  ID #REQUIRED >
<!element sec      - -  (#PCDATA) >
<!element tit      - -  (#PCDATA) >
<!element id       - o  EMPTY     >
<!attlist id refid IDREF #REQUIRED >

]>
```

The following text is a concrete example of a document written according to the DTD above.

```
<litprog>
  <tit>Example of Literate Programming</tit>
<sec>Stack - FAQ</sec>
<def ident="main">
  <prog>
    main(){
    int S[20]; sp=0;
    <id refid="push">
    <id refid="pop">
    }
  </prog>
</def>

<sec>Pushing Elements</sec>
To push elements onto the stack you can use this function.
<def ident="push">
    <prog>void push(int x)
         {S[sp++]=x;}
    </prog>
  </def>

<sec>Popping Elements</sec>
This function is not yet completed ...
<def ident="pop">
    <prog>int pop(x)
         {/* to be continued */}
    </prog>
  </def>
</litprog>
```

In the next two subsections we will be concerned with semantics, and we will discuss two approaches. The *literate programming example* will be used in the sequel to illustrate our ideas. First, we will specify the static meaning of this type of documents; then, we will define an operation `getprogram()` intended to extract the program code from a literate programming text, replacing all identifiers by their definitions.

3.1 Document Semantics as DAST

The operations we normally want to perform over documents include: translation, text formatting, interpretation, and information retrieval . Looking at those operations, the parallelism between document processing and formal language processing is obvious.

In this section, we apply to document processing, the same technic we are used to apply to formal language processing. Though, we represent document semantics as a DAST. DAST is formally specified by an attribute grammar. The first step

in the development of such semantic specification of documents, is the design of the underlying CFG. This CFG can be automatically derived from the DTD.

The following grammar has been obtained systematically from the DTD previously introduced to define what is a *literate programming document*.

```
p1  litprog  -->  "<litprog>" X  "</litprog>"
    X        -->  TXT X | prog X | def X | id X | sec X | tit X
                    |
p2  prog      -->  "<prog>" Y "</prog>"
    Y        --> TXT Y | id Y
                    |
p3  def     -->  "<def ident=" ID ">" prog  "</def>"
p4  sec      -->  "<sec>" TXT "</sec>"
p5  tit      -->  "<tit>" TXT "</tit>"
p6  id       -->  "<id refid=" ID ">"
```

After writing the CFG, it is necessary to associate attributes to the CFG symbols, and write the appropriate attribute evaluation rules (in the context of each derivation rule).

The purpose of those attributes is two fold. On one hand, they must enable us to state the semantic conditions that necessary to restrict the set of valid sentences (predicates that must be verified in the context of some productions). On the other hand, the attributes must carry all the information, directly or indirectly inferred from the source document, necessary to perform the translation task. An attribute can, obviously, be useful for both purpose. The first purpose is intended to define the so-called *static semantics*, and the second aims at the definition of the *dynamic semantics*.

The *static semantics attributes* derive easily from the `attlist` clauses present in the DTD.

Considering again our case-study, it is immediate to define the following attributes

```
def: syn {ident: word}

id:  syn {refid: word}
```

whose evaluation rules are

```
p3  def     -->  "<def ident=" ID ">" prog  "</def>"
    ident( def ) = lexval( ID )

p6  id       -->  "<id refid=" ID ">"
    refid( id ) = lexval( ID )
```

Notice that we assume that each terminal symbol (in our example `ID` and `TXT`) has an intrinsic attribute called `lexval` of type `word`.

The formulation of semantic restrictions comes directly from the specification of SGML, used to write the DTD. In our case-study, we must say:

```
p3  def     -->  "<def ident=" ID ">" prog  "</def>"
    CC: not exists( ident( def ), itab( def ))

p6  id       -->  "<id refid=" ID ">"
    CC: exists( refid( id ), itab( id ))
```

To write the precedent contextual conditions, we are faced with the need to associate with the symbols `def` and `id` an inherited attribute, `itab`. This attribute shall record all the definitions made along the document. To keep up-to-date this table of declared identifiers, we also need to introduce the following attributes

```
def: inh {itab: IdentTAB}
     syn {pros: PEleList}

id:  inh {itab: IdentTAB}

X:   inh {itab: IdentTAB}
     syn {stab: IdentTAB}

prog:syn {pros: PEleList}
```

The evaluation rules below are then added to the AG.

```
p3  def     -->  "<def ident=" ID ">" prog  "</def>"
    pros( def ) = pros( prog )

    X       -->  def X
    itab( def ) = itab( X0 )
    itab( X1 ) = update( itab( X0 ), ident( def ), pros( def ) )
    stab( X0 ) = stab( X1 )

    X       -->  id X
    itab( id ) = itab( X0 )
    itab( X1 ) = itab( X0 )
    stab( X0 ) = stab( X1 )
```

The development of the complete AG proceeds precisely in this way, until all the semantic restrictions are formulated and all the attributes used in those conditions are specified.

The *dynamic semantics attributes* are associated to the non-terminal symbols by need (just as it happened in the example above concerning the `itab` and `stab` attributes).

Looking again to our case-study, if we declare the attributes below

```
 litprog: syn {tab: IdentTAB; pros: PEleList}

 X:   syn {pros: PEleList}
```

and the evaluation rules

```
     X       -->  prog X
     pros( X0 ) = append( pros( prog ), pros( X1 ) )

p1  litprog  -->  "<litprog>" X  "</litprog>"
     tab( litprog ) = stab( X )
     pros( litprog ) = pros( X )
```

We have, at last, all that we need to be able to specify any aimed processing of the document formally described by the AG.

The `getprogram()` operation, that we have announced in the beginning of this section as the aim of the problem we are studying, can be written as follows

```
p1  litprog  -->  "<litprog>" X  "</litprog>"
     getprogram( pros( litprog ), tab( litprog ) )
```

3.2 Document Semantics in CAMILA

As a DTD is a type definition, from an algebraic point of view it can be seen as a model, as previously discussed in [RAH95].

The implicit model, for the example written in CAMILA, is:

```
model
type
  litprog = X-seq
  X       = (TXT | prog | def | id | sec | tit)
  prog    = Y-seq
  Y       = (TXT | id)
  def     = ID x prog
  sec     = TXT
  id      = ID
endtype
```

In this approach, the operation `getprogram()` is then expressed as a function over that model. It transforms a literate programming text in a program (`cprog`).

```
type
  cprog = TXT-seq
endtype

  func getprogram(t:litprog):cprog
  returns explode('main,mkindex(t));
```

getprogram()is a recursive substitution (explode) of the identifiers(id) by their associated programs, starting with main identifier.

```
func mkindex(t:litprog): id -> prog
return [ i(x) -> v(x) | x <- t : is-def(x) ] ;

func explode(i:id, d: id-> prog) : cprog
pre i in dom(d)
returns CONC(
  < if(is-id(x) -> explode(x,d),
       else     -> <x>   )  | x <- d[i]>);
```

This approach is good for prototyping semantics: it is compact to write, we can play with getprogram() independently of the concrete syntax and before building any parsing tool.

Comparing this approach with section 3.1, we can see that the algebraic model describes simultaneously both the structure and the values defined by the AG; the attribute evaluation rules correspond to the type-constructors in the algebraic approach. For instance functions like mkindex() in CAMILAcorrespond to the partial evaluation rules to build the tt stab attribute; in the AG approach such a function (that takes complex arguments) is splitted into parts (according to the CFG) that process simple arguments.

In this approach, we are looking at the DTD in a semantic point of view.

Another point of view is to consider element definition as production of a CFG as we did in the previous subsection. In this case, the DTD is only a syntactic way to define the document.

It is well known the mapping between context free grammars (CFG) and algebraic signatures[GTWW77].

The signature associated with our case-study is:

```
sorts: litprog, prog, def, sec, tit, id, TXT, ID,
operators:
  p1: (TXT | prog | def | id | sec | tit)* -> litprog
  p2: (TXT | id)* -> prog
  p3: ID x prog   -> def
  p4: TXT         -> sec
  p5: TXT         -> tit
  p6: ID          -> id
```

In order to have semantics, we have to define models for the sorts and definitions for the operators.

It is now clear how to relate the algebraic specification with the attributed grammar(AG) approach:

- the model for all the sorts corresponding to the T and NT symbols, is a cartesian product of attributes.
- the operators p1..pn are defined by a set of equations over the attributes associated with each productions

4 Conclusion

Throughout this paper we have made some reasoning about ways to represent document semantics in order to further process them using tradicional environments.

DAST (and AG) seamed to be a natural process do enrich a DTD with dynamic semantics definitions. Since we are working with SGML documents, the process of grammar derivation starting from the DTD is almost automatic. As SGML documents can have some kind of static semantics, that can be easily converted into some attribute equations (as we have shown in our case study). Dynamic semantics can also be specified through attributes. Currently we are working on a project that takes this approach and aims at implementing DSSSL (this work is based on Synthesizer Generator[RT89a, RT89b]).

Formal specification in general, and CAMILA in particular, seamed to be useful for prototyping document processing and for defining intended behavior of tools.

Some work is being done in order to build a SGML input device to CAMILA.

Algebraic programming proved to be useful for comparing different approaches.

Comparing SGML language with CAMILA specifications one difference comes out. In SGML there is a clear distinction between what is an element and what is an attribute, in our approach this difference is almost invisible. This issue could be an advantage in some cases: many times it is hard to understand why someone has specified some object as an attribute instead of specifying it as an element; there is no criteria, when developing a DTD, to decide what should become an element and what should become an attribute. The relevance of this issue is that for a parser that difference is very important, and this leads to errors even when we are dealing with small documents.

References

[BA95] Luis Barbosa and J.Joao Almeida. *System Prototyping in CAMILA*. University of Minho, 1995. Lecture notes for the system Design Course, Computer System Engineering, University of Bristol.

[FJ92] L. Feijs and H. Jonkers. *Formal Specification and Design*. 35. Cambridge Tracts in Theoretical Computer Science, 1992.

[GH93] J. Guttag and J Horning. LARCH*: Languages and Tools for Formal Specification*. Springer-Verlag, 1993.

[GTWW77] Joseph Goguen, James Thatcher, Eric Wagner, and Jesse Wright. Initial algebra semantics and continuous algebras. *Journal of the Association for Computing Machinery*, 24(1):68–95, January 1977.

[Hax90] A. Haxthausen. A Tutorial on RAISE. Technical Report RAISE/CRI/DOC/1-2-3-9, CRI A/S (Denmark), 1990.

[Hen84] P. Hendersen. me too: A Language for Software Specification and Model Building -- Preliminary Report. Tecnhical Report, University of Stirling, 1984.

[HM86] R. Harper and K. Mitchell. Introduction to Standard ML. Technical Report, University of Edimburgh, 1986.

[Knu68] Donald E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, 1968.

[Knu92] Donald E. Knuth. *Literate Programming*. Distributed by Unversity of Chicago Press, 1992. CSLI–27.

[Oli90] J. N. Oliveira. A Reification Calculus for Model-Oriented Software Specification. *Formal Aspects of Computing*, (2):1–23, 1990.

[Oli92] J. N. Oliveira. Software Reification Using the SETS Calculus (invited communication). In *Theory and Practice of Formal Software Development*. BCS FACS 5th Refinement Workshop, London, 1992.

[RAH95] J.C. Ramalho, J.J. Almeida, and P.R. Henriques. David - algebraic specification of documents. In A.Nijholt, G.Scollo, and R.Steetskamp, editors, *TWLT10 - Algebraic Methods in Language Processing - AMiLP95*, number 10 in Twente Workshop on Language Technology, Twente University - Holland, Dec. 1995.

[RT89a] Thomas Reps and Tim Teitelbaum. *The Synthesizer Generator: A System for Constructing Language-Based Editors*. Texts and Monographs in Computer Science. Springer-Verlag, 1989.

[RT89b] Thomas Reps and Tim Teitelbaum. *The Synthesizer Generator Reference Manual.* Texts and Monographs in Computer Science. Springer-Verlag, 1989.

[Tur86] D. A. Turner. MIRANDA: A Non-Strict Functional Language with Polymorphic Types. *Jour. Comp. Sys. Sci.*, (19):27–44, 1986.

A CAMILA Language

A CAMILA specification is a set of software components. Each one is a *model* that includes *type*, *function* and *state* definitions.

```
Model --> MODEL id
           TypeDef
           FunDef
           StateDef
          ENDMODEL
```

Where a type definition has the following form:

```
TypeDef --> TYPE
            ( id = TypeModel )*
            ENDTYPE
```

The basic data type models predefined in CAMILA are:

Data Models	CAMILA
Sets	X-set
Lists	X-seq
Finite functions	X $\longmapsto$ Y
Binary Relations	X $\longleftrightarrow$ Y
Alternatives	X | Y | ...
	[X]
tuples	T = X : A
	Y : B
Integers	INT
Strings	STR
Tokens	SYM
Universe	ANY

CAMILA also provides some other primitive types which do not bear a direct mathematical correspondence but are inherent to its programming environment.

A function definition has the following form:

```
FunDef --> FHeader FPredCond FState FBody
```

```
FHeader--> FUNC fid (ParamLst) : typeid
FPreCond--> PRE CondExp
FState --> STATE exp
FBody  --> RETURNS Exp
```

Finally, a state definition is written according to the rule:

```
StateDef  --> STATE sid : typeid
```

The basic collections of functions associated with CAMILA type constructors (*eg*, intersection or union of two sets, domain or range of binary relations, application or overwrite of mappings, concatenation of sequences and reduce operators, structure definition by enumeration or comprehension, etc.) are available as primitive functions in the language. So are the propositional connectives and quantifiers. To exemplify, a synopsis of some collections is presented above in the form of tables showing the CAMILA syntax, a brief informal description and the corresponding set theoretic notation.

(Finite) Functions –- $X \longmapsto Y$

CAMILA	Description	Semantics
dom(f)	Domain	$\mathsf{dom}\, f$
ran(f)	Co-domain	$\mathsf{rng}\, f$
f[x]	Application	$f[x]$
f/s	Dom. restriction	$f \mid s$
f\s	Dom. subtraction	$f \setminus s$
f + g	Overwrite	$f \dagger g$
$[\_ \mapsto \_, \ldots]$	Map. enum.	$[\ldots]$
$[x \to e \mid x \leftarrow s : p]$	Map. compreh.	$[e \mid x \in s \wedge p]$

Sequences –- X-seq

CAMILA	Description	Semantics
hd(s)	Head	$\mathsf{hd}\, s$
tl(s)	Tail	$\mathsf{t}\mid s$
nth(i,s)	Elem. by pos.	$s(i)$
s^r	Concatenation	$s \frown r$
<x:s>	Appending	$<x> \frown s$
CONC(s)	concatenation	$s_1 \frown s_2 .. \frown s_n$
elems(s)	Set of elements	$\{x \mid x \leftarrow s\}$
inds(s)	Domain	$\overline{\mathsf{dom}\, s}$
plusq(s,f)	overwrite	$s \dagger f$
<e|x<-s:p>	Seq. compreh.	$< e \mid x \in s \wedge p >$
o-orio(e,s)	Distribut. form	