GLAD: General Language to Annotate Data Using TOPO — version 3R6

Marcelino Veiga

Dpt. Ingeniería Telemática
E.T.S.I. de Telecomunicación
Univ. Politécnica de Madrid
E-28040 Madrid, SPAIN

<topo@dit.upm.es>

10 October, 1994

Abstract

It is a language, with its associated tool, that permits *external* annotation of LOTOS data. Rather than modifying the specification text, the required annotations are introduced by TOPO after semantics analysis, following an external description of user desires.

Contents

1 Introduction

TOPO is able to compile abstract data types following the user equations by means of generating a rewrite system specifically tailored to the specification subject to implementation. This may be valid for early prototypes, but is usually inadequate for real applications.

Then, TOPO permits to annotate the data. It is a mechanism to associate LOTOS names to external (user provided) objects that may be implemented much more efficiently.

Sometimes, describing a type by means of providing a set of equations is a boring activity that users wish to skip. Then, rather than specifying equations, the sort and/or operation(s) are said to be external, and the behavioral part will link to that externally provided data.

Other annotations permit user packages to refer to LOTOS data by providing name handles. It would be nice if LOTOS identifiers could be mapped to external identifiers preserving the lexical value; but it is usually infeasible due to both the rich set of characters that LOTOS accepts to build operation identifiers, and to the facilities provided for overloading names.

TOPO has also a data interpreter, useful for symbolic evaluation of data expressions, mainly for debugging purposes. In any case (compiler or interpreter), thinking in rewrite rules it is very useful to provide information about what are constructors. It allows to debug large specification much easier. So, TOPO also permit to annotation data for this purpose.

2 Why is GLAD needed?

The primitive mechanism to annotate LOTOS data consist on modifying the textual specification and write special comments syntactically linked to sort and/or operation identifiers.

Several problems are a consequence of this:

1. The specification has to be edited, and it is frequently impossible to keep the new text concentrated in a clearly distinguishable section. The annotations are poured all about the user specification.
2. It is not possible to annotate data types brought from the library. The implementer is forced provide those types needed in its specification, rather then bringing them from the standard library. Alternatively, but equally uneasy, a private copy of the `stdlib` may be edited by the user to annotate it.
3. Data operations as renaming and actualization manipulate sort and operation names. As a side effect, annotations are *propagated* with no change, but there is no way to annotate the resulting objects coming out from them, because it is unclear and perhaps too tricky.

4. It is cumbersome to do incremental annotations as the specification grows up. What does it mean?

Owed to these problems it was necessary to invent an alternative mechanism to annotate data: an annotation language.

3 Syntax

The syntax has been chosen deliberately close to LOTOS, to avoid the users have to pass long time learning a new language.

The language allows to the user to write templates compound with several sets of rules.

```
rule
  module ::= _template

rule
  _template ::= [ rule_set * ]
```

These sets of rules are divided in four classes. The first and second ones represent the beginning and the end of the LOTOS specification, and both ones consist on only one rule.

```
rule
  rule_set ::= _initial_rule

rule
  rule_set ::= _final_rule
```

And third and fourth classes correspond to sort and operation declarations, both of them compound of one or more rules.

```
rule
  rule_set ::= "sorts" _sort_rule_list

rule
  _sort_rule_list ::= [ _sort_rule + ]

rule
  rule_set ::= "opns" _operation_rule_list
```

```
rule
  _operation_rule_list ::= [ _operation_rule + ]
```

Every rule consists of two parts: a pattern and an annotation list.

```
rule
  _initial_rule ::= _initial_pattern
                    [ "=>" _initial_annotation_list ]
                    ';'

rule
  _final_rule ::= _final_pattern
                  [ "=>" _final_annotation_list ]
                  ';'

rule
  _sort_rule ::= _sort_pattern
                 [ "=>" _sort_annotation_list ]
                 ';'

rule
  _operation_rule ::= _operation_pattern
                      [ "=>" _operation_annotation_list ]
                      ';'
```

Both for initial an final rules, the pattern is simple enough.

```
rule
  _initial_pattern ::= "specification"

rule
  _final_pattern ::= "endspec"
```

For sorts, the pattern is like a LOTOS sort declaration. Only one remark, there is a special sort name, *any*, to represent any sort.

```
rule
  _sort_pattern ::= _sort_descriptor

rule
```

```
    _sort_descriptor ::= _any_sort

rule
    _sort_descriptor ::= _sort_identifier

rule
    _any_sort ::= "any"

rule
    _sort_identifier ::= IDENTIFIER
```

And for operations, the pattern is also like a LOTOS operation declaration, with some extensions: a special operation name, *any*, represents any operation; for arguments and result, the sort name *any* is allowed too; and only for arguments the key word *forall* represents any kind and any number of arguments.

```
rule
    _operation_pattern ::= _operation_descriptor
                           ':' [ _argument_descriptor ]
                           "->" _result_descriptor

rule
    _operation_descriptor ::= _any_operation

rule
    _operation_descriptor ::= _operation_identifier

rule
    _operation_descriptor ::= '_' _operation_identifier '_'

rule
    _any_operation ::= "any"

rule
    _operation_identifier ::= IDENTIFIER

rule
    _operation_identifier ::= SPECIAL

rule
    _argument_descriptor ::= _any_argument_list

rule
```

```
    _argument_descriptor ::= _argument_list

rule
    _any_argument_list ::= "forall"

rule
    _argument_list ::= [ argument + ',' ]

rule
    argument ::= _any_sort

rule
    argument ::= _sort_identifier

rule
    _result_descriptor ::= _any_sort

rule
    _result_descriptor ::= _sort_identifier
```

And here they are the annotations. For initial rules,

```
rule
    _initial_annotation_list ::= [ initial_annotation + ]

rule
    initial_annotation ::= _ldc_annotation

rule
    initial_annotation ::= _ldcinit_annotation
```

For final rules,

```
rule
    _final_annotation_list ::= [ final_annotation + ]

rule
    final_annotation ::= _ldc_annotation
```

For sort rules,

```
rule
  _sort_annotation_list ::= [ sort_annotation + ]

rule
  sort_annotation ::= _name_annotation

rule
  sort_annotation ::= _lexical_annotation

rule
  sort_annotation ::= _extern_annotation

rule
  sort_annotation ::= _free_annotation

rule
  sort_annotation ::= _nofree_annotation

rule
  sort_annotation ::= _equal_annotation

rule
  sort_annotation ::= _draw_annotation

rule
  sort_annotation ::= _nodraw_annotation

rule
  sort_annotation ::= _parse_annotation

rule
  sort_annotation ::= _noparse_annotation
```

And for operation rules,

```
rule
  _operation_annotation_list ::= [ operation_annotation + ]

rule
  operation_annotation ::= _using_annotation

rule
  operation_annotation ::= _usingsort_annotation
```

```
rule
  operation_annotation ::= _name_annotation

rule
  operation_annotation ::= _lexical_annotation

rule
  operation_annotation ::= _lexicalifpossible_annotation

rule
  operation_annotation ::= _internal_annotation

rule
  operation_annotation ::= _extern_annotation

rule
  operation_annotation ::= _call_annotation

rule
  operation_annotation ::= _partial_annotation

rule
  operation_annotation ::= _constructor_annotation

rule
  operation_annotation ::= _nonconstructor_annotation
```

The format of all annotations is the same as in LOTOS specifications.

```
rule
  _annotation ::= "(*|" annot_name annot_size "|*)"
```

And even their size, of course.

Name	Size
ldc	*text*
ldcinit	*text*
using	*word*
usingsort	–
name	*word*
lexical	–
lexicalifpossible	–
internal	–
extern	–
free	*word*

Name	Size
nofree	–
equal	*word*
draw	*word*
nodraw	–
parse	*word*
noparse	–
call	*text*
partial	*text*
constructor	–
nonconstructor	–

Besides, there are comments like in LOTOS, than can be nested.

```
rule
  comment ::= "(*" comm_text "*)"
```

4 Semantics

4.1 Annotations

At the beginning of the specification:

ldc.- It is a piece of code to be included verbatim in the generated code, and is devoted to insert links to external provisions.

ldcinit.- It is also a piece of code, and is to be invoked during start up of data run-time support. It is typically used to initialize some structures of hand coded data.

At the end of the specification:

ldc.- It is like the same one at the very beginning, but is devoted to insert some definitions to be used in data part of the specification.

For sorts we have the following annotations:

name.- Use the given name rather than a name generated by TOPO.

lexical.- Use the lexical value to refer to the sort.

It is the default option to generate sort names.

extern.- This sort is externally implemented.

free.- The user provides a function to deallocate memory.

nofree.- The user does not provide a function to deallocate memory, so data of such a sort do not need to deallocate memory.

It is the default, but it avoids a warning.

equal.- The user provides a function to decide equality, overtaking the one provided by the initial model of the algebra.

draw.- The user provides a function to draw values of this sort, rather than using system provided pretty–printing.

nodraw.- The user does not provide a function to draw values of this sort, so data of such a sort are not printable.

It is the default, but it avoids a warning.

parse.- The user provides a parsing function a string to get a value of this sort. It is used instead of the internal one.

noparse.- The user does not provide any parsing function, so this sort is not parseable.

It is the default, but it avoids a warning.

And for operations:

using.- The user provides a prefix for the operation name. Provided that `operation` is the name decided for the operation, the full name of the operation will be `prefix_operation`.

This is very handy when there is plenty of overloading, either explicit or implied by renaming and/or actualizing.

usingsort.- Similar to previous one, but the prefix is the name decided for the sort.

name.- Use the given name rather than a name generated by TOPO.

lexical.- Use the lexical value to refer to the operation.

lexicalifpossible.- The lexical value is used if the target language accepts it, otherwise the internal value is used.

internal.- Use an internal name to refer to the operation.

It is the default option to generate names.

extern.- This operation is externally provided.

call.- It is like a macro to be called instead of generating code for the operation.

partial.- A predicate is applied to function arguments in run-time, every time the operation is instantiated. It raises an exception if the predicate does not hold.

constructor.- This operation may be a constructor, and if every pattern for rewriting fails, it is legal to build a term using this operation.

This is the default option.

nonconstructor.- This operation cannot be a constructor: the patterns for rewriting should consider every case, and provide adequate rewriting. If all those fail, an exception is raised in run-time.

Everyone of these annotations will be available both via GLAD, or via direct association to operations in the source text.

4.2 Relationships

The next table an initial classification of annotations.

Name	Class	Group
ldc	*external*	–
ldcinit	*external*	–
using	*external*	*using*
usingsort	*external*	*using*
name	*external*	*name*
lexical	*external*	*name*
lexicalifpossible	*external*	*name*
internal	*external*	*name*
extern	*external*	*extern*
free	*external*	*free*

Name	Class	Group
nofree	*external*	*free*
equal	*external*	*equal*
draw	*external*	*draw*
nodraw	*external*	*draw*
parse	*external*	*parse*
noparse	*external*	*parse*
call	*external*	*call*
partial	*external*	*partial*
constructor	*symbolic*	*constructor*
nonconstructor	*symbolic*	*constructor*

As you can see, there are two classes of annotations: *symbolic* and *external* ones. First class correspond specially (but not only) to annotations for symbolic tools, i.e., there is nothing out of LOTOS (e.g. a data interpreter); and second class, *only* to annotations for non-symbolic tools, i.e., LOTOS specification can be enriched with some code in another language (e.g. a code generator). In other words, symbolic tools only can use *symbolic* annotation, but non-symbolic tools can use both *symbolic* and *external* annotations.

Besides, the annotations are divided in several groups. All the annotations in the same group have a similar function, so they are mutually exclusive. And even more, an annotation that belong to any group cannot be repeated. Only annotation belonging to no group are compatible to any other one, even itself.

Apart of this relationships, there are several ones more:

1. If a sort is *extern*, it is mandatory to provide an *equal* annotation, otherwise an error is issued. And it is convenient to provide *free*, *draw* and *parse* annotations (or their counterpart *nofree*, *nodraw* and *noparse*), otherwise a warning is issued. Moreover, if a *parse* is provided, a *draw* annotation is also needed.
2. If a sort is not *extern*, it is unusual to provide *free*, *nofree*, *equal*, *draw*, *nodraw*, *parse* and *noparse* annotations. A warning is issued if any of them is present.
3. If a sort is *extern*, every *constructor* operation returning a value of that sort is expected to be *extern* too. An error is issued otherwise.
4. If a sort is not *extern*, every *constructor* operation returning a value of that sort is expected to be not *extern* too. An error is issued otherwise.
5. If a sort is *extern* and there one or more operations of such sort, one of them must be *extern* at least.
6. If a sort is not *extern* and there one or more operations of such sort, one of them must be a *constructor* at least.
7. Pattern matching may fail on values of external sorts. In general, the rewrite system may not make any assumption on the structure of values of external sorts. In these cases, an error will be issued in a later phase.
8. For *extern* operations, it is needed either a known *name* or a *call* annotation.
9. For *extern* operations, the *partial* annotation does not make sense.
10. For not *extern* operations, the *call* annotation does not make sense.

4.3 Rules

GLAD applies on the output of the semantics analysis, where overloading has been sorted out, and there is a unique (and flat) canonical data type.

The rules of a GLAD specification are applied sequentially. The system tries to apply rule N to every object in the canonical data type; and it is applied as many times as the pattern applies; when no other application is possible, rule $N + 1$ gets its opportunity.

This mechanism permits to provide general rules (with generic patterns) at the beginning, and later on, more specific ones to refine the annotations.

But there is one exception to this mechanism: the annotations in LOTOS specifications have priority. In other words, no annotation in a GLAD specification can override another one in the LOTOS specification. It avoids surprises to the users, and allows to use GLAD only to change default annotations.

5 Examples

Let's show a few examples of GLAD use.

1. Default generation of operation names corresponds to a simple specification:

```
opns
  any: forall -> any => (*| internal |*) ;
```

2. It is useful to generate lexical names for debugging:

```
opns
  any: forall -> any => (*| lexical |*) ;
```

3. But LOTOS names for operations are not always identifiers in the target language, so it is better to generate lexical names only when possible:

```
opns
  any: forall -> any => (*| lexicalifpossible |*) ;
```

4. It is very usual that a check for constructors is required. The following specification provides a means of annotating NaturalNumbers:

```
opns
  any: forall -> nat => (*| nonconstructor |*) ;
  0  :        -> nat => (*| constructor |*) ;
  s  : nat    -> nat => (*| constructor |*) ;
```

5. And even more general, all the operations can be marked as non-constructors, to specify the constructors later on:

```
any: forall -> any => (*| nonconstructor |*) ;
```

6. When an data is renamed, plenty of overloading is usually introduced. It may be completely dealt with by qualifying every object of the resulting sort:

```
any: forall -> SetOfNats => (*| usingsort |*)
                            (*| lexicalifpossible |*) ;
```

6 Limitations

Although the approach proposed above is much more powerful than the current approach, still it has some limits.

The most notorious one is that it works on the canonical data type, what means that no scope information is available to distinguish data specified in different contexts.

This limitation seems to be rarely applicable in actual practice: most specifiers concentrate all the data in a single scope, the outmost one. This seems to be a good specification style, although resource oriented specifications might be more interested on local definitions of data.