REGLO (V2)

Pascal RAYMOND
Verimag
CNRS/Université Grenoble Alpes
Pascal.Raymond@univ-grenoble-alpes.fr
http://www-verimag.imag.fr/ raymond/

2000/2019

Reglo compiles regular expressions into programs that recognize (accept) the corresponding regular languages. Regular expressions are expressed in the ad hoc reglo source language. Generated programs are intrinsically synchronous circuits, made of logical gates and registers. Concretely, they are expressed in the synchronous dataflow language Lustre. This is the second version of the language and tool, re-implemented in ocaml.

Contents

1 Reglo source language

A regular expression denotes a language whose vocabulary is the set of valuation of a some finite set of Boolean variables. We call *memory* such a finite set $M = \{x_1, \cdots, x_m\}$.

A *valuation* of the memory is a tuple of m Boolean values. The set of all possible valuations can be viewed as a vocabulary (with 2^m symbols).

A sequence of valuations is called a *trace* over the memory: indeed, it is a word over the vocabulary 2^m.

A set of traces is called a *history* of the Boolean memory.

The reglo source language allows the description of histories using classic regular constructs (sequence, iteration and union).

1.1 Lexical aspects

The comments are C++ like:

```
// line comment
/* multi
lines
comment */
```

The following strings are keywords of the source language:

- `eps`
- `prefix`

Any other string made of digits, letters and underscore can be used as identifier.

1.2 History declaration

A source program consists of a list of history declarations. A declaration is of the form:

<*declaration*> ::= <*ident*>`(` <*ident-list*>`)` `=` <*expression*>`;`

and means that <*ident*> denotes the history over <*ident-list*> described by the regular expression <*expression*>.

1.3 Monomials

A basic item in a regular expression is a set of valuations, denoted as a Boolean function of the memory variables. This function may use logical *and* and *not*, but not the logical *or*, which is useless since we have the regular union operator. This means that basic items are boolean monomials over the memory variables:

<*monomial*> ::= <*atom*>
| `[` <*atom-list*> `]`

<*atom*> ::= <*ident*>
| `-` <*ident*>
| `~` <*ident*>

A monomial is a bracket enclosed list of *atoms*. The comma in the list stands for the logical and operator.

Atoms are conditions on a single variable. Each variable may appear at most once in the monomial. The `-` prefix stands for the logical not operator. The `~` prefix can be used to outline the fact that a variable is "don't care", but it can be avoided: `[X, -Y, ~Z ]` is equivalent to `[X, -Y ]`.

If the monomial contains a single atom, brackets can be avoided: `[-X]` is equivalent to `-X`.

The list of atoms in a monomial can be empty: the empty monomial `[]` stands for the "always true" function.

1.4 Expressions

The syntax is the following:

<*expression*> ::= <*monomial*>
| <*expression*>`.` <*expression*>
| <*expression*>`*`
| <*expression*>`+` <*expression*>
| `~~`
| `eps(` <*expression*>`)`
| `prefix(` <*expression*>`)`

The first three operators are classical: concatenation, finite iteration and union.
The `~~` macro denotes the set of all possible traces; it is equivalent to `[]*`.
The `eps` operator adds the empty trace to an history.
The expression "`prefix(` E `)`" denotes all the prefixes of the traces in "E". For instance:

```
prefix(a.b.(a + b)*)
```

is equivalent to:

```
eps(a + a.b + a.b.(a + b)*)
```

1.5 Call of a regular expression

When it is defined, a regular expression can be called in the definition of an other regular expression. It allows the user to write programs which "look like" regular grammar. Note that recursive definitions are forbidden, even though they are semanticaly correct (like left or right-recursive grammars).
The syntax of call is the following:

<*expression*> ::= `<` <*ident*>`(` <*monomial-list*>`)>`
| `<` <*ident*>`>`

Definition order In order to (simply) reject recursive definitions, a regular expression must be defined before it it used.

Default parameters If actual parameters are omited, the formal parameters of the caller are given, in the same order as they are declared. As a consequence, the call is correct if and only if the caller and the called have the same number of parameters.

```
toto(A, B) = (A + A.B)* ;
titi(X, Y) = <toto>.X ;  //equivalent to : <toto(X,Y)>.X
tutu(B, A) = <toto>.A ;  //equivalent to : <toto(B,A)>.A
bibi(B, A, C) = <toto>.C ; //wrong : not same number of parameters
```

Actual parameters Each actual parameter must be a monomial:

```
toto(A, B) = (A + A.B)* ;
titi(X, Y, Z) = <toto([X,-Y],[Z])>.X ;
```

Call semantics A call can be viewed as a macro: it is equivalent to the regular expression obtained by substituing the formal parameters to the actual parameters of the call. In the previous example, the call `<toto([X,-Y],[Z])>` is equivalent to `([X,-Y] + [X,-Y].[Z])*`.

2 Use of the compiler

2.1 Simple mode

The syntax of the command is:
`reglo` <*file-name*>
where <*file-name*> is a file in the reglo format.
For each declaration in the source program:
<*ident*>`(` <*ident-list*>`) =` <*expression*>`;`
the compiler builds a LUSTRE node which header is:
`node` <*ident*> `(` <*ident-list*> `: bool) returns ( OK : bool);`
N.B. The actual name for the output ("`OK`" in the example) is not guaranted.

2.2 Modular mode

If the source file contains expression calls, or if the option `-m` is given to the command, the compiler builds two nodes.

- A "core" node whose header is:
  `node` <*ident*>`_core (INIT, PFX,` <*ident-list*>`: bool) returns ( OK : bool);`
- A main node whose header is:
  `node` <*ident*> `(` <*ident-list*> `: bool) returns (OK : bool);` which is simply a call of the prevvious one.

2.3 Options

`-v` **(verbose)** This option tells the compiler to output some informations on `stderr`.

`-o` <*file-name*> **(output file)** The command "`reglo name.rg`" normally produces the file "`name.lus`".

`-os` **(stdout)** This option forces the compiler to output its result on `stdout`.

`-m` **(modular compilation)** This option forces the modular mode.

`-eqs` **(language equations)** The default is to produce a LUSTRE recognizer. With this option, the compiler outputs a set of language equation (in a readable format).

2.4 Errors

Sorry, but syntax errors are not well handled... On the contrary, static semantics errors produce meaningful messages:

- `regular '`<*ident*>`' must have parameters` *(a regular expression must have at least one parameter)*
- `regular '`<*ident*>`' defined twice`
- `In '`<*ident*>`' header: parameter '`<*ident*>`' declared twice`

- `In '<ident>' de'finition: param '<ident>' used twice in a monomial`
- `In '<ident>' de'finition: undeclared parameter '<ident>'`

3 Examples

3.1 Temporal logic

Reglo was developped for programming of *synchronous observers*. Such an observer checks if a sequence of configurations meets some safety property [**?**]. Here are some simple observers:

Once Let `a` be a Boolean variable, we want to describe the set of sequences containing at least one configuration where `a` is true:

```
Once(a) = [-a]*.a.~~ ;
```

The preceding equation outlines the first occurence of `a`. The following definition denotes the same property, but outlines the last occcurence of `a`:

```
Once(a) = ~~.a.[-a]* ;
```

Finally, the following definition outlines “some” occurence, but it is still equivalent:

```
Once(a) = ~~.a.~~ ;
```

After We now want to define the set of traces where `a` has been true at least once in the past. In order to express the delay, we can use the “true” monomial `[]`, which represents any configuration (do not confuse with the “any sequence” macro `~~= []*`):

```
After(a) = [-a]*.a.[].~~ ;
After(a) = ~~.a.[].~~ ;
```

Annexe : syntax

```
<reglo-file>    ::=  <declaration-list>
    <declaration-list>   ::=  <declaration>
                          |   <declaration>  <declaration-list>
    <declaration>        ::=  <ident>( <ident-list>) = <expression>;
    <ident-list>         ::=  <ident>
                          |   <ident>, <ident-list>
    <monomial>   ::=  <atom>
                  |   [<atom-list>]
    <atom-list>  ::=  <atom>  |  <atom>, <atom-list>

    <atom>       ::=  <ident>
                  |   - <ident>
                  |   ~<ident>
    <expression>  ::=  <monomial>
                  |    <expression> . <expression>
                  |    <expression>*
                  |    <expression> + <expression>
                  |    ~~
                  |    eps( <expression>)
                  |    prefix( <expression>)
                  |    < <ident>( <monomial-list>) >
                  |    < <ident>>
```