NONA (code selector description translator)

Vladimir Makarov, vmakarov@gcc.gnu.org Apr 5, 2001

This document describes NONA (a code selector description into code for solving code selection and possibly other back-end tasks).

Contents

1 Introduction

NONA is a translator of a code selector description (CS) into code for solving code selection and possibly other back-end tasks. The code selector description is mainly intended for describing code selection task solution, i.e. for determining by machine-independent way a transformation of a low level internal representation of source program into machine instruction level internal representation. But the code selector description can be used also to locate machine dependent code for solving other back-end task, e.g. register allocation. To describe code selector description a special language is used.

An code selector description describes mainly tree patterns of low level internal representation with associated costs and semantic actions. NONA generates the tree matcher which builds cover of low level internal representation by the tree patterns with minimal cost on the first bottom up pass and fulfills actions associated with the choiced tree patterns on the second bottom up pass. Usually the actions contain code to output assembler instruction.

Analogous approach for solving code selection task is used by modern generator generators such as BEG, Twig, Burg and Iburg. The tree matcher generated by NONA uses algorithm similar to one of BEG and Iburg, i.e. the algorithm is based on dynamic programming during fulfilling code selection.

Although the algorithm used by BURG and based on dynamic programming during tree pattern matcher generation time is considerably more fast, it is not acceptable for us. Its main drawback which is to need usage of less powerful code selector description results in necessity of usage of more machine-dependent low level internal representation. For example, the special internal representation node types for 8-bits, 16-bits constants besides 32-bits constants would be needed. Also the algorithm used by BURG is considerably more complex.

Tree pattern matchers generated by NONA also can work with directed acyclic graphs besides trees. This feature is useful when target machine instruction is generated from the internal representation which is result of some optimizations such as common sub-expression elimination.

2 Code selector description language

An code selector description describes mainly tree patterns of low level internal representation with associated usage constraints, costs and semantic actions. The tree pattern may contains terminal representing the low level internal representation node. Nonterminals serves to designate the tree patterns. Each terminal or nonterminal may have attribute which is evaluated in actions.

2.1 Layout of code selector description

Code selector description structure has the following layout which is similar to one of YACC file.

```
DECLARATIONS
%%
RULES
%%
ADDITIONAL C/C++ CODE
```

The '%%' serves to separate the sections of description. All sections are optional. The first '%%' starts section of rules and is obligatory even if the section is empty, the second '%%' may be absent if section of additional C/C++ code is absent too.

The section of declarations contains declarations of terminals by names of constants which represent modes of the low level internal representation nodes. The section also may contain declarations of commutativeness of some terminals. Type and union declarations can be used to declare types of terminal and nonterminal attributes. And finally the section may contain subsections of code on C/C++.

The next section contains rules. Each rule describes tree pattern, nonterminal designating the pattern, the pattern cost, its usage constraint, and associated action. The rule list can be empty. In this case the code selector description is used to locate machine dependent code and generated code of the tree pattern matcher can not be used.

The additional C/C++ code can contain any C/C++ code you want to use. Often functions which are not generated by the translator but are needed to work with code selector description go here. This code without changes is placed at the end of file generated by the translator.

2.2 Declarations

The section of declarations may contain the following construction:

```
%term <TYPE> IDENTIFIER ...
```

All terminals must be defined in constructions of such kind (or in construction '%commutative'). The same identifier can be defined repeatedly. The identifier is a name of constant which represents mode of the low level internal representation node. If the user needs usage of attributes of different types, the terminal attribute type can be given in this construction. The union declaration can be used for this.

```
%union {
  any thing that can go inside a 'union' in C/C++
}
```

The last such construction in the declarations section specifies the entire collection of possible types for attribute values. For example:

```
%union {
  IR_node_t node;
  int number;
}
```

This means that the two alternative types are 'IR_node_t' and 'int'. They are given by names 'node' and 'number'. These names are used in the type and commutativeness declarations to instruct type of attribute for a terminal or nonterminal. The type declarations has the following form:

```
%type <TYPE> IDENTIFIER ...
```

The construction can be used when union declaration is present. Here 'TYPE' is a name of type given in union declaration. If the type of terminal or nonterminal is not given its type is believed to be 'CS_TYPE' (see next section). The identifier is one of terminal or nonterminal. The construction

```
%commutative <TYPE> TERMINAL_IDENTIFIER ...
```

describes terminal and commutativeness of operator represented by the terminal besides optional instruction for the terminal attribute type. For example, if the terminal 'plus' is described as commutative then pattern 'plus (register, constant)' simultaneously designates pattern 'plus (constant, register)'. There may be also the following constructions in the declaration section

```
%local {
   C/C++ DECLARATIONS
}

%import {
   C/C++ DECLARATION
}

and
```

```
%export {
   C/C++ DECLARATION
}
```

which contain any C/C++ declarations (types, variables, macros, and so on) used in sections. The local C/C++ declarations are inserted at the begin of generated implementation file (see code selector description interface) but after include-directive of interface file. C/C++ declarations which start with '%import' are inserted at the begin of generated interface file. For example, such C/C++ code may redefine some internal definitions, e.g. macro defining type of attributes. C/C++ declarations which start with '%export' are inserted at the end of generated interface file. For example, such C/C++ code may contain definitions of of external variables and functions which refer to definitions generated by NONA. All C/C++ declarations are placed in the same order as in the section of declarations.

2.3 Rules

The section of declarations is followed by section of rules . A rule is described the following construction

```
nonterminal : pattern cost %if constraint action
```

The rule connects tree pattern with nonterminal. Several tree patterns can be connected to a nonterminal. Nonterminal is given by its identifier. The rest constructions cost, constraint with keyword '%if', and actions are optional. Cost is a integer expression on C/C++ in brackets '[' and ']'. If the cost is omited in the rule, then its value is believed to be zero. The cost value has to be non-negative. Constraint which is given as a boolean expression on C/C++ in brackets '[' and ']' defines a condition of the tree pattern usage. In the case of absent constraint its value is believed to be nonzero. The tree pattern will be never in a minimal cost cover if constraint value is equal to zero during testing the tree pattern as a candidate on inclusion into minimal cost cover. For example, the following rule with constraint describes Am29k instruction CONST

```
register : integer_constant [1]
           %if [IR_value ($1) >=0 && IR_value ($1) < 256]
```

The actions can contain any statements on C/C++ in figure brackets '{' and '}'. Constructions '$$' and '$n' (n is a integer number here) denote attributes of correspondingly nonterminal before ':' and terminal or nonterminal in the pattern with order number equal to 'n'. The order numbers start with 1. These constructions may be present in the cost, the constraint and the action. But it should be remembered that definition of the moment of the cost and the constraint evaluation is quite difficult. Moreover there is not guaranty that given cost and constraint will be evaluated. Therefore side effects should be not in the cost and in the constraint, and usage attributes of nonterminals should be not in the cost and in the constraint. The cost and the constraint may be fulfilled on the first bottom up pass. It is known only that the cost is fulfilled after evaluation of constraint give nonzero result. The action of the tree pattern which is a part of result minimal cost cover is fulfilled on the second bottom up pass. Tree pattern is given by one of the following forms:

```
TERMINAL ( PATTERN , ...)
TERMINAL
NONTERMINAL
```

For example, the following pattern describes Am29050 instruction DMSM

```
double_addition (twin, double_multiplication (twin,
                                              accumulator0))
```

It should be remembered that each terminal must have fixed arity in all patterns. It means that usage of the following patterns in a code selector description will be erroneous.

```
integer_addition (register, register)
integer_addition (register)
```

Full YACC syntax of internal representation description language is placed in Appendix 1.

3 Generated code

A specification as described in the previous section is translated by code selector description translator (NONA) into code selector description (CS) interface and implementation files having the same names as one of specification file and correspondingly suffixes '.h' and '.c' for C or '.cpp' for C++ (see option '-c++').

3.1 C Interface objects

C interface file of CS consists of the following definitions of generated type and functions:

1. There is constant with nonterminal name and prefix 'CSNT_' for each nonterminal. 'CSNT' is abbreviation of code selector description nonterminal. The constants are defined as macros (with the aid of C preprocessor directive '#define'). The constants are used as parameter of functions 'CS_it_is_axiom' and 'CS_traverse_cover' (see below).

2. Macro 'CS_NODE' must represent type of node of low level internal representation. This macro can be redefined (only in a C declaration section which starts with '%import'). By default this macro is defined as follows:

   ```
   #define CS_NODE           struct IR_node_struct *
   ```

3. Macro 'CS_TYPE' defines type of terminal and nonterminal attributes. This macro can not be redefined if the corresponding code selector description has a construction '%union' which already describes terminal and nonterminal attribute types because in this case the default macro definition is absent. By default this macro is defined as follows:

   ```
   #define CS_TYPE           CS_node
   ```

4. Type 'CS_node' is defined as follows:

   ```
   typedef CS_NODE CS_node;
   ```

5. Type 'CS_cover' describes a minimal cost cover. The values of this types are created by function 'CS_find_cover' and deleted by function 'CS_delete_cover'.

6. Function

   ```
   'CS_cover CS_find_cover (CS_node node)'
   ```

makes the first bottom up pass on directed acyclic graph given by its node and finds a minimal cost cover for each nonterminal. The minimal cost cover is returned. If error occurs during the pass then macro 'CS_ERROR' (see below) is evaluated. For example, the cause of error may be meeting undeclared terminal code. During building cover, each directed acyclic graph (DAG) node is visited only once.

7. Function

   `'int CS_it_is_axiom (CS_cover cover, int nonterminal)'`

   returns 1 if there is a minimal cost cover for given nonterminal in given cover, 0 otherwise.

8. Function

   `'CS_TYPE CS_traverse_cover (CS_cover cover, int nonterminal)'`

   makes the second bottom up, left to right pass and fulfills actions corresponding a minimal cost cover for given nonterminal in cover found by call of function 'CS_find_cover'. A minimal cost cover for given nonterminal must be in given cover before the function call. This condition can be checked by function 'CS_it_is_axiom'. The function returns attribute of given nonterminal after fulfilling the actions. Remember that action (and attribute evaluation) associated with tree pattern is fulfilled only once for each occurrence of the tree pattern in given cover of a directed acyclic graph.

9. Function

   `'void CS_delete_cover (CS_cover cover)'`

   frees memory allocated to 'cover' which was to be created by call of function 'CS_find_cover'. Of course this cover can not be used in function 'CS_traverse_cover' after that. The memory is freed in the reverse order therefore allocation macros (see below) can use stack strategy of the memory allocation.

10. Function

    `'void CS_start (void)'`

    is to be called before any work with code selector description. The function initiates code selector description storage management (see below).

11. Function

    `'void CS_finish (void)'`

    is to be last. The function finishes code selector description storage management. Therefore only function 'CS_start' can be called after this function call.

3.2 C++ Interface objects

C++ interface file (see option -c++) of CS consists of the following definitions of generated type and functions:

1. There is constant with nonterminal name and prefix 'CSNT_' for each nonterminal. 'CSNT' is abbreviation of code selector description nonterminal. The constants are defined as macros (with the aid of C preprocessor directive '#define'). The constants are used as parameter of functions 'CS_it_is_axiom' and 'CS_traverse_cover' (see below).

2. Macro 'CS_NODE' must represent type of node of low level internal representation. This macro can be redefined (only in a C declaration section which starts with '%import'). By default this macro is defined as follows:

```
#define CS_NODE           struct IR_node_struct *
```

3. Macro 'CS_TYPE' defines type of terminal and nonterminal attributes. This macro can not be redefined if the corresponding code selector description has a construction '%union' which already describes terminal and nonterminal attribute types because in this case the default macro definition is absent. By default this macro is defined as follows:

```
#define CS_TYPE           CS_node
```

4. Type 'CS_node' is defined as follows:

```
typedef CS_NODE CS_node;
```

5. Type 'CS_cover' describes a minimal cost cover. It is pointer objects of a class. The class has not public constructors and desctructors. The class objects can be created only by function 'CS_find_cover' and can be deleted by function 'CS_traverse_cover'. This class has the following friend functions:

   (a) Function

   ```
   'CS_cover CS_find_cover (CS_node node)'
   ```

   makes the first bottom up pass on directed acyclic graph given by its node and finds a minimal cost cover for each nonterminal. The minimal cost cover is returned. If error occurs during the pass then macro 'CS_ERROR' (see below) is evaluated. For example, the cause of error may be meeting undeclared terminal code. During building cover, each directed acyclic graph (DAG) node is visited only once.

   (b) Function

   ```
   'void CS_start (void)'
   ```

   is to be called before any work with code selector description. The function initiates code selector description storage management (see below).

   (c) Function

   ```
   'void CS_finish (void)'
   ```

   is to be last. The function finishes code selector description storage management. Therefore only function 'CS_start' can be called after this function call.

   This class has also the following class functions:

   (a) Function

   ```
   'int CS_it_is_axiom (int nonterminal)'
   ```

   returns 1 if there is a minimal cost cover for given nonterminal in given cover, 0 otherwise.

   (b) Function

   ```
   'CS_TYPE CS_traverse_cover (int nonterminal)'
   ```

makes the second bottom up, left to right pass and fulfills actions corresponding a minimal cost cover for given nonterminal in cover found by call of function 'CS_find_cover'. A minimal cost cover for given nonterminal must be in given cover before the function call. This condition can be checked by function 'CS_it_is_axiom'. The function returns attribute of given nonterminal after fulfilling the actions. Remember that action (and attribute evaluation) associated with tree pattern is fulfilled only once for each occurrence of the tree pattern in given cover of a directed acyclic graph.

(c) Function

```
'void CS_delete_cover (void)'
```

frees memory allocated to the cover which was to be created by call of function 'CS_find_cover'. Of course the function 'CS_traverse_cover' can not be used after that (because the object is removed). The memory is freed in the reverse order therefore allocation macros (see below) can use stack strategy of the memory allocation.

3.3 Implementation file macros

CS implementation file uses the following macros generated by NONA. These macros can be redefined (in any C/C++ declaration section).

1. Macro 'CS_OPERATION(node)' must return value defined by a constant which represents mode of the low level internal representation node (see description of construction '%term').

2. Macros

```
'CS_OPERAND_1_OF_1(node)',
'CS_OPERAND_1_OF_2(node)',
'CS_OPERAND_2_OF_2(node)', and so on.
```

   These macros define the structure of the low level internal representation for which is needed to build minimal cost cover. The structure must be a directed acyclic graph (DAG).

   Macro of kind 'CS_OPERAND_k_OF_n' defines k-th arc from the DAG node given as the macro argument which has 'n' such arcs. Let us consider tree pattern

```
'integer_plus (register, const8)'
```

   To access to children of node represented by terminal 'integer_plus' the macros 'CS_OPERAND_1_OF_2' and 'CS_OPERAND_2_OF_2' will be used.

3. Macros

```
'CS_STATE(node)',
'CS_SET_STATE(node, state)'
```

   define access to the a DAG node field. The field is used by tree pattern matchers. The type of this field must be any pointer type. Macro 'CS_STATE' has one argument of type 'CS_node'. Macro 'CS_SET_STATE' has two arguments of types correspondingly 'CS_node' and 'void *'.

4. Macro 'CS_ATTRIBUTE(node)' must return value of type 'CS_TYPE'. This value is attribute of the terminal corresponding to given node of low level internal representation.

   This macro for given terminal may be evaluated several times because the terminal attribute can be in cost or constraint constructions. The value of the macro for the terminal must be l-value (variable in other words) when construction '%union' is present and given terminal is declared with type because in this case the operation '.' of C/C++ language is applyed to the macro value in order to get access to the terminal attribute.

5. Macro 'CS_ERROR(str)' is a reaction on fixing an error by tree pattern matcher. The macro argument is tree pattern matcher message about the error.

By default these macros are defined as follows:

```
#define CS_OPERATION(node)        IR_NODE_MODE (node)
#define CS_OPERAND_1_OF_1(node)   IR_operand (node)
#define CS_OPERAND_1_OF_2(node)   IR_operand_1 (node)
#define CS_OPERAND_2_OF_2(node)   IR_operand_2 (node)
#define CS_OPERAND_1_OF_3(node)   IR_operand_1 (node)
#define CS_OPERAND_2_OF_3(node)   IR_operand_2 (node)
#define CS_OPERAND_3_OF_3(node)   IR_operand_3 (node)
...
#define CS_STATE(node)            IR_state (node)
#define CS_SET_STATE(node, state) IR_set_state (node, state)
#define CS_ATTRIBUTE(node)        (node)
#define CS_ERROR(str)             fprintf (stderr, "%s\012", str)
```

3.4 Storage management macros

NONA completely automatically generates macros of storage management for the cover. Usually, the user does not have to care about it. The predefined storage management uses C standard functions for global storage with free lists. The user can create own storage manager by redefinition of one or more the following macros and placing them in a C declaration section. But the user should know that all functions generated by NONA believe that the storage can not change its place after the allocation.

1. Macros 'CS_START_ALLOC()', 'CS_FINISH_ALLOC()' are evaluated in functions correspondingly 'CS_start' and 'CS_finish' and are to be used to start and finish work with the storage manager.

2. Macros 'CS_ALLOC(ptr, size, ptr_type)', 'CS_FREE(ptr, size)' allocate and free storage whose size is given as the second parameter. The first parameter serves to return pointer to memory being allocated or to point to memory being freed.

By default these macros are defined as follows:

```
#define CS_START_ALLOC()
#define CS_FINISH_ALLOC()
#define CS_ALLOC(ptr, size, ptr_type) \
                      ((ptr) = (ptr_type) malloc (size))
#define CS_FREE(ptr, size)  free (ptr)
```

4 NONA Usage

NONA(1) User Manuals NONA(1)

NAME

nona - code selector description translator

SYNOPSIS

nona [-c++ -v -debug -export -pprefix] specification-file

DESCRIPTION

Command nona translates code selector description (CS) which is described in specification file into code for solving code selection (tree matcher) and possibly other back-end tasks. The specification file must have suffix '.nona'.

Tree matcher determined by CS builds cover of low level internal repâĂŘresentation (which must be a directed acyclic graph) by the tree patâĂŘterns with minimal cost on the first bottom up pass and fulfills actions associated with the choiced tree patterns on the second bottom up pass.

Generated code consists of interface and implementation files having the same names as one of specification file and correspondingly sufâĂŘfixes '.h' and '.c' (C code) or '.cpp' (C++ code).

Full documentation of OKA is in OKA User's manual.

OPTIONS

The options which are known for NONA are:

-c++ NONA generates C++ code instead of C code (default).

-v NONA outputs statistic information to standard output stream.

-debug NONA outputs debugging information during execution of function 'CS_find_cover' and 'CS_traverse_cover' (see generated code).

-export
NONA generates macros defining identifiers of terminals as inteâĂŘger constants and inclusion of their in the interface file. Usually the user himself declares the identifiers in a C declaâĂŘrations section.

-pprefix
NONA generates names starting with prefix 'prefix' instead of 'CS'.

```
FILES
       file.nona
              NONA specification file
       file.c
              generated C implementation file
       file.cpp
              generated C++ implementation file
       file.h
              generated interface file

       There are no any temporary files used by NONA.

ENVIRONMENT
       There are no environment variables which affect NONA behavior.

DIAGNOSTICS
       NONA diagnostics is self-explanatory.

AUTHOR
       Vladimir N. Makarov, vmakarov@gcc.gnu.org

SEE ALSO
       msta(1), shilka(1), sprut(1), oka(1).  NONA manual.

BUGS
       Please, report bugs to https://github.com/dino-lang/dino/issues.



COCOM                              5 Apr 2001                              NONA(1)
```

5 Appendix 1 - Syntax of code selector description language

YACC notation is used to describe full syntax of code selector description language.

```
  %token PERCENTS LEFT_PARENTHESIS RIGHT_PARENTHESIS
         LEFT_ANGLE_BRACKET RIGHT_ANGLE_BRACKET COMMA COLON SEMICOLON
         IDENTIFIER CODE_INSERTION EXPRESSION ADDITIONAL_C_CODE

  %token COMMUTATIVE LOCAL IMPORT EXPORT UNION TERM TYPE IF

  %start description

  %%

  description : declaration_part  PERCENTS  rule_list  ADDITIONAL_C_CODE
              ;

  declaration_part :
                   | declaration_part  commutative_declaration
```

```
                | declaration_part  term_declaration
                | declaration_part  type_declaration
                | declaration_part  LOCAL CODE_INSERTION
                | declaration_part  IMPORT CODE_INSERTION
                | declaration_part  EXPORT CODE_INSERTION
                | declaration_part  UNION CODE_INSERTION
                ;

term_declaration : TERM  optional_type
                | term_declaration  term_identifier
                ;

commutative_declaration : COMMUTATIVE  optional_type
                        | commutative_declaration  term_identifier
                        ;

type_declaration : TYPE  type
                | type_declaration  term_or_nonterm_identifier
                ;

optional_type :
              | type
              ;

type : LEFT_ANGLE_BRACKET  IDENTIFIER  RIGHT_ANGLE_BRACKET
     ;

rule_list :
          | rule_list  rule
          ;

rule : nonterm_identifier  COLON  pattern
       optional_cost  optional_constraint  optional_action  SEMICOLON
     ;

optional_action :
                | CODE_INSERTION
                ;

optional_constraint :
                    | IF  EXPRESSION
                    ;

optional_cost :
              | EXPRESSION
              ;

nonterm_identifier : IDENTIFIER
                   ;

term_identifier : IDENTIFIER
```

```
                ;

term_or_nonterm_identifier : IDENTIFIER
                           ;

pattern : IDENTIFIER  LEFT_PARENTHESIS  pattern_list  RIGHT_PARENTHESIS
        | term_or_nonterm_identifier
        ;

pattern_list : pattern
             | pattern_list COMMA pattern
             ;
```