Vortex RTL Textual Description Grammar

The Cecil Group

Department of Computer Science and Engineering
University of Washington
Box 352350, Seattle, Washington 98195-2350
cecil@cs.washington.edu

February 6, 1996 5:13 pm

This document describes the grammar for the textual form of the Vortex compiler's RTL. This is the language used to communicate between the C++ and Modula-3 front-ends and Vortex

Warning: the RTL language has not been designed from scratch, but rather via accretion of features. It is in great need of some reworking to make the operators more general and language-independent.

1 Top-Level Entry Points

For languages that have separate front-ends, such as C++ and Modula-3, source files are translated by the front-end into RTL files. An RTL file is described by the following grammar:

```
rtl_file          ::=  { rtl_decl | pragma }
rtl_decl          ::=  rtl_method_decl
                   |   rtl_var_decl
                   |   rtl_init_decl
                   |   rtl_object_decl
                   |   rtl_assoc_decl
                   |   rtl_include_decl
                   |   rtl_top_level_expr
```

2 Global Declarations

```
rtl_method_decl  ::=  "method" [rtl_visibility] rtl_name "(" {rtl_formals} ")"
                        ":" representation [pragma] "{" rtl_stmts "}" [";"]
rtl_formals      ::=  rtl_formal {"," rtl_formal}
rtl_formal       ::=  [rtl_name] ["@" rtl_name] [":" representation]
rtl_visibility   ::=  "private" | "static" | "extern"
rtl_var_decl     ::=  "decl" [rtl_visibility] representation rtl_name ";"
rtl_init_decl    ::=  "static_init" rtl_name "{" {rtl_init} "}" ";"
rtl_init         ::=  "[" int "," representation "," rtl_init_value "]"
rtl_init_value   ::=  "pad"
                  |   "address" "(" rtl_name "," integer ")"
                  |   "istr" "(" string ")"
                  |   "str" "(" string ")"
rtl_object_decl  ::=  "object" rtl_name ["isa" rtl_names] [object_id_pragma] ";"
rtl_names        ::=  rtl_name {"," rtl_name}
rtl_assoc_decl   ::=  "associate" rtl_name "with"
                        msg_name "(" rtl_formals ")" ";"
msg_name         ::=  rtl_name
rtl_include_decl ::=  "include" string ";"
rtl_top_level_expr::="rtl" "{" rtl_stmts "}" [";"]
```

3 Data Representations

Representations are specified in declarations and in statements.

```
representation  ::=  "bytes" "[" integer "]"
                |    base_rep {"*"}
base_rep        ::=  "char" | "u_char" | "short" | "u_short"
                |    "int"  | "u_int"  | "long"  | "u_long"
                |    "float" | "double"
                |    "void"
                |    "wordInt" | "unsignedWordInt" | "wordFloat" | "OOP"
                |    "ReturnCode" | "PairType" | "reg"
```

4 Statements

RTL statements are the following:

```
rtl_stmts       ::=  { rtl_stmt ";" }
rtl_stmt        ::=  alu
                |    vector_op
                |    ptr_op
                |    allocation
                |    goto
                |    label
                |    fn_call
                |    return
                |    fatal
                |    proc_addr_op
                |    static_info
```

Many RTL statements may be prefaced with a declaration. These declarations are used to pass along type information that will be needed to declare variables in the C++ code we generate as output.

```
decl            ::=  "decl" ["local"] representation
```

Arithmetic operations and assignments:

```
alu             ::=  assign | unop | binop | cast
assign          ::=  [decl] rtl_name [":=" value]
unop            ::=  [decl] rtl_name ":=" unop_op value
binop           ::=  [decl] rtl_name ":=" value binop_op value
cast            ::=  [decl] rtl_name ":="
                       "cast" rtl_name representation

value           ::=  rtl_name | literal | "void" | "true" | "false"

literal         ::=  integer | float | character | string
```

Simple pointer operations

```
ptr_op          ::=  ptr_load | ptr_store | addr_op
ptr_load        ::=  [decl] rtl_name ":=" ptr_location ["#" representation]
ptr_store       ::=  ptr_location ":=" value ["#" representation]
ptr_location    ::=  "*" "(" rtl_name ["+" index] ")"
addr_op         ::=  [decl] rtl_name ":=" "&" rtl_name
```

Branches & labels:

```
goto            ::=  cond_goto | uncond_goto | case_goto
cond_goto       ::=  "if" test "goto" label_name ["backedge"] [loopexit]
```

```
test             ::=  unop_op value
                 |    value binop_op value
loopexit         ::=  "true_loopexit" | "false_loopexit"
uncond_goto      ::=  "goto" label_name ["backedge"]
case_goto        ::=  "case" value "goto" "[" label_names "]"
label_names      ::=  label_name {"," label_name}
label            ::=  ("label" | "loop") label_name
label_name       ::=  rtl_name
```

Various kinds of procedure calls

```
fn_call          ::=  ...| call
call             ::=  [[decl] rtl_name ":="] call_desc "(" [values] ")"
call_desc        ::=  "call" rtl_name
                 |    "call_external" rtl_name
                 |    "call_indirect" value
                 |    "call_nested" rtl_name
values           ::=  value {"," value}
static_type_info ::=  "static_type" "(" static_type ")"
static_type      ::=  rtl_name
```

Return statements

```
return           ::= "return" [value]
```

Fatal errors

```
fatal            ::=  "fatal" "(" string ")"
```

Assertions about the static properties of a value

```
static_info      ::=  "assert_type" rtl_name static_type_info
```

Currently, static information specifies either an exact class for the variable, or specifies a cone of possible classes (and the value could be any subtype of the class). The optimizer uses this information to reason about the possible values that could be stored in the given variable.

```
static_type_info ::=  "map" "(" rtl_name ")"
                 |    "cone" "(" rtl_name ")"
```

5 Tokens

The token syntax for RTL is the same as for Cecil, with the addition of `rtl_name`, which has the same syntax as a C identifier:

```
rtl_name         ::=  (letter | "_") {letter | digit | "_"}
```

6 Unary and Binary Operators

These are the unary and binary operators currently defined.

unop_op name	description
unary_-_int	integer negation
unary_-_float	float negation
~_int	bitwise complement
convert_int_to_float	convert an integer into a float

unop_op name	description
convert_float_to_int	convert a float into an integer, rounding as with C (wordInt) cast
round	**I don’t think this is implemented fully**
trunc	**I don’t think this is implemented fully**
ceiling	**I don’t think this is implemented fully**
floor	**I don’t think this is implemented fully**

binop_op name	description
+_int	Add two signed integers
-_int	Subtract ...
*_int	Multiply ...
/_int	Divide ...
%_int	Modulus ...
&_int	Bitwise and two integers
|_int	... or ...
^_int	... xor ...
xnor_int	... xnor ...
<<_int	bitwise left shift one integer by another
>>_int	... arithmetic right shift ...
>>_logical_int	... logical right shift ...
+_ptr	Add two values, one of which is assumed to be a pointer and the other an integer byte count, to produce a new pointer
-_ptr	Subtract an integer byte count from a pointer to produce a new pointer
-_ptrdiff	Subtract two pointers to produce an integer difference, in bytes
+_float	Add two floats
-_float	Subtract ...
*_float	Multiply ...
/_float	Divide ...
=_int_log	Test whether two integers are equal
!=_int_log	... not equal
>_int_log	Test whether one signed integer is greater than another
>=_int_log	... greater than or equal to ...
<_int_log	... less than ...
<=_int_log	... less than or equal to ...
>_unsigned_log	Test whether one unsigned integer is greater than another
>=_unsigned_log	... greater than or equal to ...

binop_op name	description
<_unsigned_log	... less than ...
<=_unsigned_log	... less than or equal to ...
=_ptr_log	Test whether two pointers are equal
!=_ptr_log	... not equal
>_ptr_log	Test whether one pointer is greater than another
>=_ptr_log	... greater than or equal to ...
<_ptr_log	... less than ...
<=_ptr_log	... less than or equal to ...
=_float_log	Test whether two floats are equal
!=_float_log	... not equal
>_float_log	Test whether one float is greater than another
>=_float_log	... greater than or equal to ...
<_float_log	... less than ...
<=_float_log	... less than or equal to ...