Pragmas and Vortex

The Cecil Group

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

Abstract

This document describes the pragmas recognized by Vortex. We first specify the supported pragma syntax and document to which source constructs pragmas may be attached. We then describe the semantics of the pragmas currently recognized by Vortex.

1 Pragma Syntax

(Reprised from the Cecil language specification and the Vortex RTL specification.)

The syntax of a pragma is as follows:

```
pragma          ::=  “(**” exprs “**)”
```

In the body of a pragma, `exprs` is just a syntax tree. Individual pragmas will interpret this format in particular ways. `exprs` typically has the form `name`, optionally followed by `(exprs′)`, where `name` is the name of the pragma and `exprs′` is a list of arguments to the pragma.

In a Cecil source file, a pragma may occur either as a top-level declaration or as part of a method, field, or variable declaration:

```
top_decl_block ::=  { decl | pragma }
method_decl    ::=  [privacy] impl_kind method_name
                        “(” [formals] “)” [type_decl] {pragma}
                        “{” (body | prim_body) “}” [“;”]
field_decl     ::=  [field_privacy] [“shared”] [“var”] “field” field_kind
                        msg_name [formal_params] “(” formal “)” [type_decl]
                        {pragma} [“:=” expr] “;”
let_decl       ::=  [privacy] “let” [“var”] name [type_decl] {pragma}
                        “:=” expr “;”
```

In an RTL source file, a pragma may occur as a top-level declaration or as part of an object, method, field, or variable declaration:

```
rtl_file         ::=  { rtl_decl | pragma }
rtl_object_decl  ::=  [rtl_obj_role] “object” name [“isa” names]
                          {pragma|object_id_pragma} “;”
rtl_method_decl  ::=  “method” [rtl_visibility] rtl_name “(” {rtl_formals} “)”
                        “:” representation {pragma} “{” rtl_stmts “}” [“;”]
rtl_extern_method_decl::= “extern” “method” name “(” {rtl_formals} “)”
                        “:” representation {pragma} “;”
rtl_field_decl   ::=  [“var”] [array_field_info] “field” [rtl_visibility] name
                        “(” rtl_formal “)” “:” representation {pragma}
                        [“:=” literal] “;”
rtl_var_decl     ::=  “decl” [rtl_visibility] representation name {pragma}
                        [“:=” (literal | static_init)] “;”
```

2 Pragma Semantics

2.1 Pragmas for Describing External Primitives

Primitive methods written in a language other than Vortex RTL (e.g. C++) are called external primitives, since Vortex cannot reason about their behavior. Vortex needs to know their behavior, however, and this behavior is documented in a set of pragmas. Usually, pragmas can be omitted and Vortex will make conservative assumptions, but sometimes Vortex makes optimistic assumptions (that reflect the vast majority of cases), and some pragmas Vortex requires of all external primitives, at least when performing certain analyses such as interprocedural analysis. The following non-conservative characteristics are assumed of external primitives:

- External primitives don't instantiate or reference any objects that are not also created/referenced in some analyzable call graph node that is guaranteed to be included in the call graph whenever the primitive node is in the call graph.
- Unless a `sends` or `sends_anything` pragma is specified, an external primitive is assumed to not send any messages or otherwise invoke any methods or procedures compiled by Vortex.
- Unless a `has_nlr` or `has_no_nlr` pragma is present, an external primitive is assumed to possibly result in a non-local return/exception exactly if one of its callees does.
- Unless a `has_side_effects` or `has_no_side_effects` pragma is present, an external primitive is assumed to possibly modify exactly the union of those variables possibly modified by its callees.

To document less conservative behavior we support several pragmas on external primitive method declarations:

- `sends(v1 = m1([t11,t22],[t21]), m2([t11,t12,v1]), ...)` specifies the messages sent by the external primitive. The argument vectors specify the argument class sets. The notation `v1 = m1`… introduces a variable that represents the set of classes returned by `m1`. Each message has a list of vectors, each vector representing a set of classes for that argument. Each `t` in a vector may be `unknown`, the name of a formal variable, the name of a variable declared as a result of an earlier send in the `sends` pragma, or the name of a class.
- `sends_anything` specifies that the method may send any message with any possible argument classes.
- `return_type(t1,t2,t3...tn)` indicates that the primitive method is guaranteed to return one of `t1...tn`, with the same allowed form of the `ti`'s as in the `sends` pragma.
- `does_not_return` indicates that the primitive doesn't return normally (e.g., it always exits with a non-local return or error).
- `has_nlr` indicates that the external method may result in a non-local return/exception (needed only if no callee may result in a non-local return). `has_no_nlr` indicates that the method does not end in a non-local return, even if one of its callees may, e.g. the primitive catches and suppresses all non-local returns.
- `has_no_side_effects` indicates that the external primitive does not modify any variables, even if its callees might. `has_side_effects` indicates that the external primitive might modify (unknown) variables in addition to what its callees might modify.

- `formals_escape(...,t,...,f,...)` indicates which formals of the method may escape, i.e., may be referenced by some other external primitive after this external primitive returns. The `formals_escape` pragma has a list of `t`s and/or `f`s, one per formal of the external primitive, with `f` indicating that the corresponding formal definitely doesn't escape. [A better syntax would list the names of possibly-escaping formals.]

2.2 Pragmas for Controlling Compilation

2.2.1 Pragmas on Method Declarations

- The `optimize` and `no_optimize` pragmas enable and disable, respectively, compilation of a method with optimization.
- The `debug` pragma on a method disables optimizations like static binding and inlining that would prevent the method from being replaced by the evaluator or caught by a breakpoint.
- The `inline` and `no_inline` pragmas specify that the annotated method should be inlined or not inlined, respectively, when statically bound. (If no such pragma is present, an automatic inlining heuristic is used.)
- The `no_typecheck` pragma on a method disables typechecking of the annotated method.

Specifying any of these pragmas at the top-level of a file affects all methods declared in the file.

2.2.2 Pragmas on Object Declarations

- The `ID([[obj_name, ID_name], ...])` pragma on an object declaration an in RTL program specifies that when the annotated object is referred to under the `obj_name`$_i$ static type, `ID_name`$_i$ names its corresponding class representation (a.k.a. virtual-function table). (See the Vortex RTL specification for slightly more information.)

2.2.3 Pragmas at Top-Level

- The `no_typecheck` pragma at the top-level of a file disables typechecking of all declarations in the file (not just the methods in the file).
- The `specialize("msg", num_msg_params, [[obj_name`$_1$`, num_obj_params`$_1$`], ...,[obj_name`$_N$`, num_obj_params`$_N$`]])` pragma causes the method named `msg` with `num_msg_params` type parameters and `N` formals to be specialized, where its i^{th} formal is specialized on the object named `obj_name`$_i$ with `num_obj_params`$_i$ type parameters.
- The `library("name")` pragma identifies the containing file as being the root file for the library named `name`. All files included from this file lacking their own library pragmas are included in the library. When the Vortex option `use_shared_libraries` is on, the files in a library will be compiled separately from all including applications (without optimization) and shared by all including applications.
- The `include_dir("-Idir1 -Idir2 ...")` pragma adds its argument to the include-directory list part of the command line used for `cc`, `as`, and `ld`. (Note that the directories should be absolute path names, since the directory used for compiling and linking the generated code is usually not the same as that containing the file referring to the include directories.)
- The `compile_with("...")` pragma adds its argument to the end of the flags part of the command line used for `cc` and `ld`. (Any `-Idir` options in the argument list are extracted and added to the include-directory list.)

- The `lib_dir("-Ldir1 -Ldir2 ...")` pragma adds its argument to the library-directory list part of the command line used for `ld`. (Note that the directories should be absolute path names, since the directory used for linking the generated code is usually not the same as that containing the file referring to the library directories.)
- The `lib("-llib1 -llib2 lib3.a ...")` pragma adds its argument to the library list part of the command line used for `ld`. (Note that any path names of libraries should be absolute path names, since the directory used for linking the generated code is usually not the same as that containing the file referring to the library.)
- The `link_with("...")` pragma adds any `-llib` arguments to the library list, adds any other `-...` options to the linker flags list, and adds all other arguments to the end of the command line used for `ld`.

2.3 Pragmas for Controlling Interprocedural Analysis

- The `recursive_customization` pragma on a method marks it as being potentially a source of unbounded recursive context-sensitive analysis.
- The `control_structure` pragma on a method indicates that the method should be analyzed separately for each of its callers, even in otherwise context-insensitive algorithms.

2.4 Pragmas for Controlling Language Parameters

- The `language("language_name")` pragma at the top-level of an RTL file documents the source language of the file.
- The `lang_info(expression)` pragma at top-level causes `expression` to be evaluated at the start of compilation, with the intention of allowing language-specific properties to be specified.