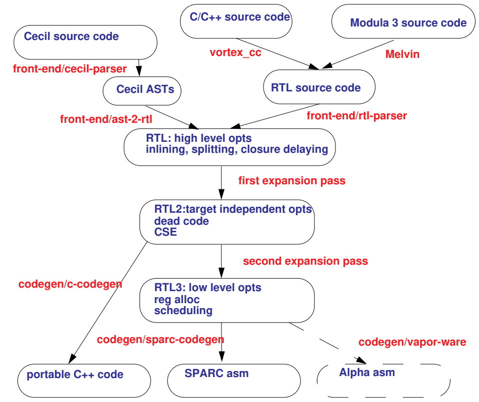
Overview

Vortex was designed to:

- Support whole program analysis
- Be easily extensible
- Support optimization of object-oriented languages

Compiler Structure



What Happens When you ‘Make’?

Parse all out-of-date files

Rebuild file & variable tables (`compiler-inc.cecil`)

For each file to be codegened

... *some initialization*...

For each function in that file

build CFG from textual RTL

optimize & codegen (`codegen-cfg.cecil`)

... *wrap up the file* ...

Output makefile information

Important Hierarchies

RTL - Defines kinds of nodes in control flow graphs (in `internal-representation/rtl`)

rtl_symbols - Operands for RTL nodes (in `internal-representation/rtl`)

Scopes - Symbol tables. Keep track of what variables were defined in each scope, and of the nesting relationship among scopes. Lots of different kinds. (in `internal-representation/scopes`)

CFG_context - Defines control flow graph. Basically a graph of RTL nodes, plus info which nodes are the head and tails of the graph (in `internal-representation/rtl`)

rtl_symbols

used as operands of RTL nodes

Literals

- int, string, char, float, vector
- funky runtime values (uninit, delayed, maps)

Variables

Registers

Useful operations defined on RTL

(see `rtl.cecil` and `rtl-dataflow.cecil`)

```
print_string(RTL):string
copy(RTL):RTL
hash(RTL, int):int

simple_uses(RTL):indexed[rtl_symbol]
replace_uses(RTL, indexed[rtl_symbol]):void

simple_defs(RTL):indexed[rtl_symbol]
replace_defs(RTL, indexed[rtl_symbol]:void
```

Useful operations defined on rtl_symbols

```
print_string(rtl_symbol):string
=(rtl_symbol, rtl_symbol):bool
hash(rtl_symbol, int):int
```

IDFA Framework

Framework manages iteration, convergence, CFG mutation

Client provides:

- analysis_info
  - merge, copy, print_string, (compatible), (set_scope)
- analysis closure
  - &(n:RTL, i:analysis_info):analysis_result { ... *do work* ... }

Possible analysis results:

- continue with (modified) analysis_info
- delete node
- replace node with a node or a whole graph, analyze replacement

Key file is cfg-traversal-interface.cecil

```
traverse(g:CFG_context,
         dir:traversal_direction,
         mode:traversal_mode,
         initial_info:analysis_info,
         cls: &(RTL, analysis_info):analysis_result):void
```

dir: `forward_traversal` or `reverse_traversal`

mode: `iterative` or `noniterative`

Clients should never modify the graph directly.

What does cls usually look like?

```
&(node:RTL, info:analysis_info) { node.do_analysis(info) }
```

```
method do_analysis(t@:RTL, info) { ....... }

method do_analysis(t@:rtl_move, info){ ..... }

method do_analysis(t@:rtl_send, info) { ...... }
```


Cecil Tips

At `debug>` or `Eval>`, can type lots of useful things

Use the evaluator to speed up development

- paste methods in instead of recompiling
  - does not always work
  - cannot paste in object creation expressions (e.g., `object isa rtl_move`), object or field declarations
- create temporary variables (discarded once you exit the evaluator; not accessible from pasted-in methods)
- insert `breakpoint()` to get the debugger when inside a method
  - use `debug> adopt_env` command when debugging pasted-in methods

More Cecil Tips

Use dot graphs to debug optimizations

- option: `Cecil> set make_dot_graphs 2`
- method: `debug> g.make_dot_graph`

Mix optimized & unoptmized code

- set `Cecil> no_optimize` for incremental compiles
- alternatively, insert `(** debug **)` pragma in files you are debugging
- run `Cecil> make_optimized` periodically

Things to avoid if possible (potentially long inc. recompiles)

- deleting fields from `comp` or `options`
- don't modify 'if' 'plus' etc.