Compiling to iso-recursive types
Experience with Waml and Wob

Andreas Rossberg

Recap

Iso-recursive types – syntax

strtype ::= (**func** *valtype** *valtype**) | (**struct** *valtype**) | …

deftype ::= (**sub** *$t** *strtype*)
| (**rec** (**sub** *$t** *strtype*)$^{+}$)

For now, allowing at most 1 supertype

(**sub** ε *strtype*) can be abbreviated to just *strtype*

Iso-recursive types – semantics

Subtyping rule only uses declared subtyping (constant time)

(**sub** *$t strtype*) is well-formed iff *strtype* matches *$t*

(**rec** *subtype*$^+$) is well-formed iff all *subtype*$^+$ are

…shallow check, recursively assumes declared subtyping

Rules almost like nominal typing, except that identical type defs are equivalent

…nominal up to type canonicalisation (bottom-up)

…rec definitions only equivalent if types in them are, in same order

rtt.sub instruction removed, **rtt.canon** on subtype takes its role

Iso-recursive types – example

```
(type $t1 (struct i32))
(type $t2 (struct i32))                          ;; $t1 == $t2

(rec
  (type $u1 (struct (ref $v1)))
  (type $v1 (sub $t1 (struct i32 (ref $u1))))    ;; $v1 <: $t1
)

(rec
  (type $u2 (struct (ref $v2)))                  ;; $u1 == $u2
  (type $v2 (sub $t2 (struct i32 (ref $u2))))    ;; $v1 == $v2,  $v2 <: $t2
)

(rec
  (type $v3 (sub $t2 (struct i32 (ref $u3))))    ;; $v3 =/= $v2
  (type $u3 (struct (ref $v3)))                  ;; $u3 =/= $u2
)
```

Iso-recursive types – example

```
(rec
  (type $t1 (struct i32))
  (type $t2 (struct i32))                          ;; $t1 =/= $t2

  (type $u1 (struct (ref $v1)))
  (type $v1 (sub $t1 (struct i32 (ref $u1))))      ;; $v1 <: $t1

  (type $u2 (struct (ref $v2)))                    ;; $u1 =/= $u2
  (type $v2 (sub $t2 (struct i32 (ref $u2))))      ;; $v1 =/= $v2,  $v2 <: $t2

  (type $v3 (sub $t2 (struct i32 (ref $u3))))      ;; $v3 =/= $v2
  (type $u3 (struct (ref $v3)))                    ;; $u3 =/= $u2
)
```

single global **rec** degenerates into nominal semantics

Porting from equi to iso

General recipe

1. Replace uses of rtt.sub instruction with sub definitions
2. If not making use of structural typing:
   (a) Wrap entire type section into a single rec
3. Otherwise:
   (a) Wrap each recursion group into separate rec
   (b) Possibly, normalise type order in each rec
   (if compiling for separate compilation)

Separating concerns

1. During code generation, emit type definitions without concern for recursion
2. In the end, run SCC over types to insert **rec**'s
3. If necessary, also normalise order of type definitions

Essentially, (2-3) is an algorithm for translating a set of equi-recursive to iso-recursive definitions

Waml

Changes to Waml

1. Replace uses of rtt.sub with sub definitions
2. Other than that, emit type definitions as before
3. In the end, run a simple SCC algorithm to group types

125 line change to Waml compiler, +70 lines SCC module

Took approximately 3 hours to implement

Why Waml was easy

Main use of type recursion in Waml is for function and functor closures

…mutual recursion between code pointer **func** and closure environment **struct**

…recursive data types do *not* lower to recursive types, due to uniform representation

Compiler always emits type recursion in fixed order

…source code does not affect order within recursion

…no need for additional normalisation

SCC only needed for rediscovering recursion group boundaries

…could have avoided even that by refactoring some code

Wob

Changes to Wob

1. Replace uses of rtt.sub with sub definitions
2. Other than that, emit type definitions as before
3. In the end, run a simple SCC algorithm to group types
4. Then, normalise order of types

260 line change/addition to Wob compiler

- +70 lines SCC module (reused from Waml)
- +130 lines generic type renumbering code (reused from Wln)

Took 1-2 days to implement, main challenge was defining a suitable ordering algorithm

Why Wob was more work

Type recursion in Wob comes from classes

…mutual recursion between instance **struct**, vtable **struct**, and method pointer **func**s

…but also, possibly mutual recursion between multiple classes, injected by source

When generating imports, compiler only has type signature of imported module, not source

…lost ordering information between mutually recursive classes

Three possible fixes:

(A) extend signature information to allow reconstructing order in code generator

(B) normalise order in a generic fashion in module builder

(C) use type imports for nominal source types (but type imports not yet implemented in GC branch)

Took second route – probably was more work, but a fully general, language-agnostic solution

…represents expected upper bound of complexity to support separate compilation with iso

Normalising “Recify” algorithm

1. Run SCC algorithm
2. Define a total ordering on Wasm type definitions (NB: not related to subtyping)

   …pick some total order on type constructors

   …lift lexicographically to (lists of) *strtype* trees

   …extended to graphs for recursive SCCs (essentially, comparing tree keys)
3. Sort SCC’s according to ordering function

   …incrementally, along SCC dependency frontier

   …recursive groups are first sorted internally
4. Apply renumbering and insert **rec**’s

About 180 lines of code, once I knew what I was doing :)

Tools

WIn

1. Adapt renumbering code to type syntax changes

25 line change

Took less than 1 hour to implement

JS runner

No changes

Summary of Impact

Minimal impact on whole-program compilers

…wrapping module in a big **rec** is enough

Small impact on separate compiler for language with explicit type recursion

…either produce appropriate **rec** grouping by construction,

…or run SCC in the end

Larger impact on separate compiler for language with liberal type recursion

…either ensure appropriate **rec** grouping and type order by construction,

…or run SCC and order normalisation in the end

Practically no impact on loaders and linkers

Porting everything took about 2 days

Open Questions

How much does it affect other tools?

Are there cases of structural source languages it can’t handle?

https://github.com/WebAssembly/gc/tree/iso/interpreter (Reference interpreter)

https://github.com/WebAssembly/gc/tree/iso.waml/proposals/gc/wob (Wob)

https://github.com/WebAssembly/gc/tree/iso.waml/proposals/gc/waml (Waml)

https://github.com/WebAssembly/gc/tree/iso.waml/proposals/gc/wln (Linker)

outtakes

Summary of Recify

Recify algorithm is generic, independent of source language

…can be encapsulated in module builder

…frees code generators from thinking about type recursion

Essentially, maps a set of equi-recursive type equations into iso-recursive definitions that are minimal and normalised