Taming Type Recursion

Andreas Rossberg

Recap: Structural types

Current proposal features structural types

…to support modular compilation and linking

Current proposal *also* uses equi-recursive type equivalence

…least restrictive, but can lead to costly canonicalisation

But structural ≠ equi-recursive

…separate questions!

Recap: Iso-recursive types

Inductive, recursion groups explicit, type ASTs form trees

Can be canonicalised bottom-up in linear time using std techniques

Type equivalence sufficient for all MVP use cases

But concerns regarding subtyping

…standard rule is restrictive: applies point-wise to mutually recursive types

rec X. (T, U, V) <: rec X’. (T’, U’, V’)
iff T <: T’ and U <: U’ and V <: V’ assuming X <: X’

…would need more casts, impact on producers non-obvious

Example: “nested” subtyping

```
class C {
    func f(x : D) {}
}

class D {
    func g(x : C) {}
}

class E <: C {
    func h(x : E) {}
}
```

```
rec
  $C     = struct (ref $C-vt)
  $C-vt  = struct (ref $C-f)
  $C-f   = func (ref $C) (ref $D)

  $D     = struct (ref $D-vt)
  $D-vt  = struct (ref $D-g)
  $D-g   = func (ref $D) (ref $C)

rec
  $E     = struct (ref $E-vt)
  $E-vt  = struct (ref $C-f) (ref $E-h)
  $E-h   = func (ref $E) (ref $E)
```

~~$E <: $C~~

Observations

Generally, need to establish subtyping between individual types from multiple (or same) recursion group(s)

Works with equi-recursion
but not with simple iso-recursion rule

Generalising iso-recursive subtyping

Generalising iso-recursive subtyping sufficiently is possible

…but in the limit requires a co-inductive relation with implicit unrolling

$$T <: \mu X.\, T' \quad \text{iff} \quad T <: T'[X := \mu X.\, T']$$
$$\mu X.\, T <: T' \quad \text{iff} \quad T[X := \mu X.\, T] <: T'$$

However, such subtype rules do not affect type equivalence and canonicalization, so wouldn't have the same problem as equi-recursion

(NB: antisymmetry wouldn't hold, but that is not a problem)

But subtyping check can still have super-linear cost in pathological cases

Iso-recursive Expressiveness

Iso-recursive types can embed equi-recursive types [Abadi & Fiori, LICS 1996]

…i.e., every program using equi can be compiled to iso (compositionally)

…by inserting a coercion to bridge retyping/subsumption rule

The approach extends to subtyping

…since that can also be compiled with coercions

Operationally, these coercions are unobservable

…could introduce zero-cost use-site coercion instructions (suggested by Sam Lindley)

(NB: Strictly speaking, using coercions only works for pure types, but that can be solved)

Definition-site Annotations

Use-site annotations are rather unwieldy

Idea: not use-site but def-site annotations (suggested before by Luke Wagner)

…somewhat less flexible

…but looks fairly familiar

Effectively, nominal subtyping modulo iso-equivalence

Definition-site Annotations

deftype ::= (**func** *valtype* valtype**) | (**struct** *valtype**) | (**rec** *deftype*$^{+}$) | (**sub** *$t deftype*)

Subtyping rule can only use declared subtyping between defined types

$t <: $u
if $t == $u
or $t = (**sub** $t' dt) and $t' <: $u

Moreover, (**sub** *$t deftype*) is well-formed iff *deftype* structurally matches unrolled *$t*

(**sub** *$t deftype*) ok
iff *deftype* <: unroll(*$t*)

Shallow check, assumes declared subtyping in same recursion group are ok well-formed

Almost like nominal typing, except that identical type defs are still equivalent

Example revisited

```
class C {
    func f(x : D) {}
}

class D {
    func g(x : C) {}
}

class E <: C {
    func h(x : E) {}
}
```

```
rec
  $C     = struct (ref $C-vt)
  $C-vt  = struct (ref $C-f)
  $C-f   = func (ref $C) (ref $D)

  $D     = struct (ref $D-vt)
  $D-vt  = struct (ref $D-g)
  $D-g   = func (ref $D) (ref $C)

rec
  $E     = sub $C (struct (ref $E-vt))
  $E-vt  = struct (ref $C-f) (ref $E-h)
  $E-h   = func (ref $E) (ref $E)
```

$E <: $C

Example revisited

```
class C {
    func f(x : D) : C {…}
}

class D {
    func g(x : C) {}
}

class E <: C {
    func f(x : D) : E {…}
}
```

$E <: $C

```
rec
  $C     = struct (ref $C-vt)
  $C-vt  = struct (ref $C-f)
  $C-f   = func (ref $C) (ref $D) → (ref $C)

  $D     = struct (ref $D-vt)
  $D-vt  = struct (ref $D-g)
  $D-g   = func (ref $D) (ref $C)

rec
  $E     = sub $C (struct (ref $E-vt))
  $E-vt  = struct (ref $E-f)
  $E-f   = func (ref $C) (ref $D) → (ref $E)
```

Remarks

Structural in hierarchy

…same technique as used for RTTs

…effectively same rules that'd be needed for nominal subtyping when checking supplied imports against import signature

Can be extended to multiple supertypes

…unlike RTTs

…care required for imports

Summary of Possible Solutions

0. Accept the need for casts
1. Co-inductive subtyping (with inductive equivalence)
2. Use-site annotations (no-op coercion operators)
3. Def-site annotations (declared subtyping)

   …“nominal types modulo bottom-up canonicalisation”

   …combines relevant advantages of structural and nominal

Outtakes

Reminders

Wasm-level types describe data layout,
they are only tangentially related to source language.

They (should) have no operational relevance.

Types are a genuine language problem,
the language should provide a solution.

Aside: Casts & Runtime Types

Escape hatch for limitations of Wasm type system

Casts recover type information lost by static typing

Hence, casts over structural types must be structural

…otherwise we'd be back to square one

That's what rtt.canon (and rtt.sub) enable