Iso-recursive Types

Andreas Rossberg

Recap: Equi-recursive Types

Type definitions are equations, may refer to each other arbitrarily

Type ASTs form regular trees (representable as graphs)

Unrolling is implicit in type equivalence

Type equivalence is greatest fixpoint over equations (co-induction/recursion)

Canonicalization requires graph minimization

Alternative: Iso-recursive Types

Type definitions are ordered equations, may only refer to earlier ones

Recursion groups are defined explicitly, and use explicit self reference

Type ASTs form finite trees

Unrolling is explicit in certain typing rules

Type equivalence is smallest fixpoint over equations (induction/recursion)

Canonicalization is just classic bottom-up hash-consing

Equi Example

```
type $t = struct (ref $u)
type $u = struct (ref $t)

type $v = struct (ref $t)

;; $u == $v
```

Equi Example

```
type $t = struct (ref $u)
type $u = struct (ref $t)

type $v = struct (ref $t)

;; $u == $v == $t

type $w = struct (ref $w)

;; $t == $u == $v == $w == μ($self). struct (ref $self)
```

Core Rule of Equi-Recursion

$$\mu(\text{self}).\ t \quad == \quad t\ [\text{self} := \mu(\text{self}).\ t]$$

e.g., $\mu(\text{self}).\ \text{struct (ref self)} \quad == \quad \text{struct (ref (}\mu(\text{self}).\ \text{struct (ref self)))}$

$\$t \ = \ \text{struct (ref } \$t) \ == \ \text{struct (ref (struct (ref } \$t))) \ == \ \ldots$

Core Rule of Equi-Recursion

$$\mu(\text{self}).\ t \quad == \quad t\ [\text{self} := \mu(\text{self}).\ t]$$

Equivalence Rule for Iso-Recursion

$\mu(\text{self}).\ t_1 \quad == \quad \mu(\text{self}).\ t_2 \quad \text{iff} \quad t_1 == t_2$

No implicit unrolling

Recursive type only equal to other recursive types

μ is an explicit type constructor that needs to be handled

Mutual Iso-Recursion

When unrolling isn't implicit, mutual recursion requires extra support

Generalize μ to a tuple of types:

$\mu(\text{self}).\ \langle t_1, \ldots, t_N \rangle$

Introduce a projection operator for type tuples:

$t.\mathrm{i}$

Wasm Extensions for Iso-recursion

New form of recursive type definition:

deftype ::= **func** *valtype** | **struct** *fieldtype** | **array** *fieldtype* | **rec** *deftype**

A *deftype* may only refer to smaller type indices, or to itself when inside **rec**

(NB: no real need to allow nested **rec**)

Extension of concrete heap types with projection:

heaptype ::= **any** | **data** | **extern** | **func** | … | **ref** *typeidx*.*n* | **rtt** *typeidx*.*n*

Refers to the *n*-th type of a recursion group (can be omitted when 0)

(NB: alternative would be "alias definitions" similar to module linking proposal)

Example

```
type $tu = (rec
  struct (ref $tu.1)
  struct (ref $tu.0)
)

type $v = struct (ref $tu.0)

;; $tu.1 ≠ $v
```

Type Equivalence

$(\text{ref } \$t1.n_1) == (\text{ref } \$t2.n_2)$ iff $\$t1 == \$t2$ and $n_1 = n_2$

type $\$t1 = (\text{rec } \mathit{deftype}_1{}^*)$
type $\$t2 = (\text{rec } \mathit{deftype}_2{}^*)$

$\$t1 == \$t2$ iff $(\mathit{deftype}_1 == \mathit{deftype}_2[\$t2 := \$t1])^*$

Subtyping

(ref $\$t1.n_1$) <: (ref $\$t2.n_2$) iff $\$t1$ <: $\$t2$ and $n_1 = n_2$

type $\$t1$ = (rec $deftype_1{}^*$)
type $\$t2$ = (rec $deftype_2{}^*$ $deftype_3{}^*$)

$\$t1$ <: $\$t2$ iff $(\$t1 <: \$t2 \vdash deftype_1 <: deftype_2)^*$

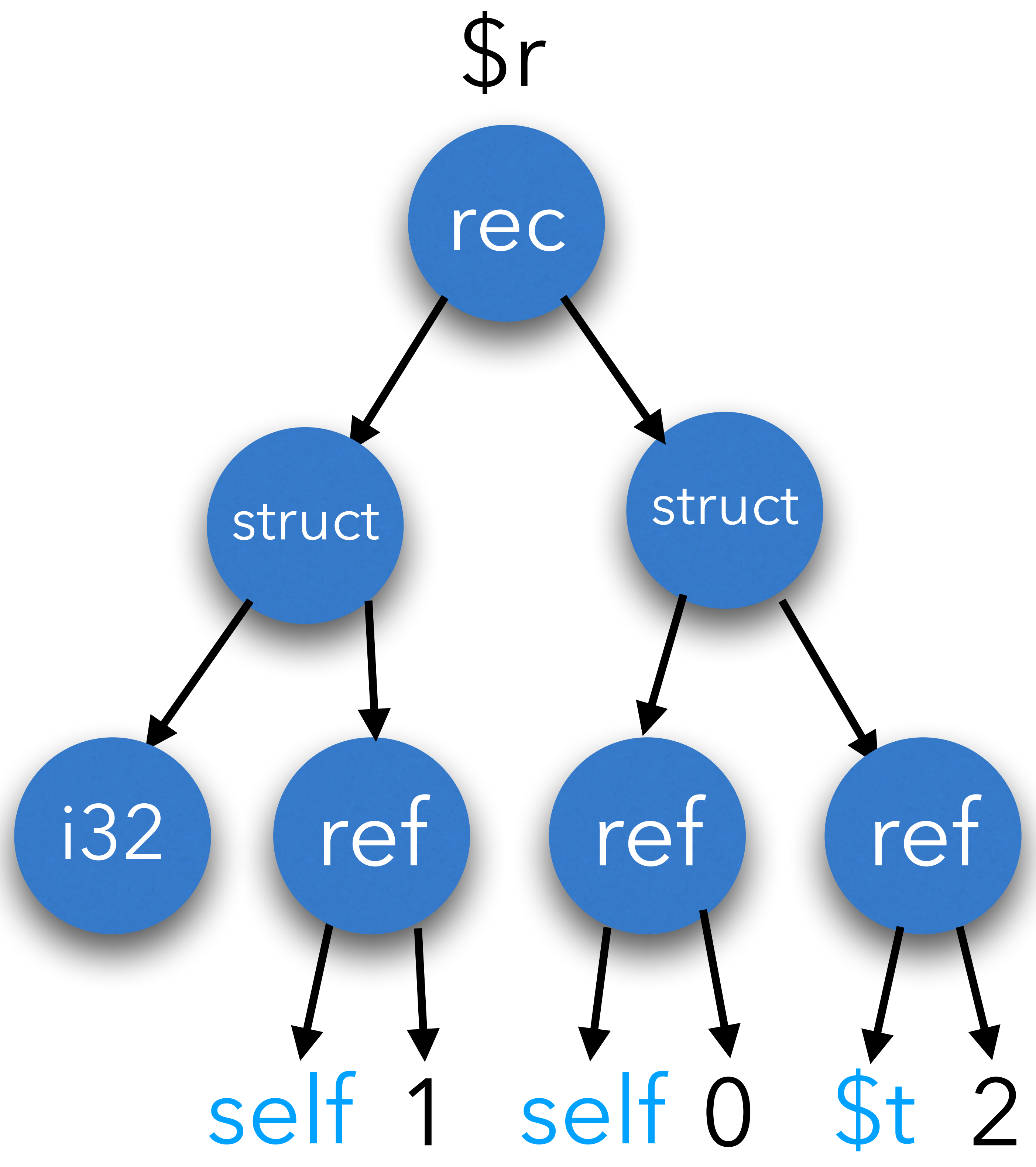
Canonicalization

Unique representations can be built with bottom-up hash-consing

Provided recursive self-references are represented relatively

Canonicalization

```
type $r = (rec
  (struct i32 (ref $r.1))
  (struct (ref $r.0) (ref $t.2))
)
```



Instruction Typing

Recursive types no longer equal their unrolling

So they must be handled explicitly in typing rules for instructions

Fortunately, separate instructions for unrolling are unnecessary,
can be integrated into typing rules for existing instructions

Instruction Typing

Previously:

$\mathbf{struct.get}$ i : [(ref $s)] → [$t$] iff $s = $\mathbf{struct}\ t_1^{i}\ t\ t_2^{*}$

Now:

$\mathbf{struct.get}$ i : [(ref $s.n)] → [$t$] iff unroll($s)[n] = $\mathbf{struct}\ t_1^{i}\ t\ t_2^{*}$

where:

unroll(**struct** *fieldtype**) = (**struct** *fieldtype**)
unroll(**rec** *deftype**) = *deftype**

Instruction Typing

Previously:

$$\textbf{struct.new\_default} : [(\text{rtt } \$s)] \to [(\text{ref } \$s)] \quad \text{iff} \quad \$s = \textbf{struct } t^*$$

Now:

$$\textbf{struct.new\_default} : [(\text{rtt } \$s.n)] \to [(\text{ref } \$s.n)] \quad \text{iff} \quad \text{unroll}(\$s)[n] = \textbf{struct } t^*$$

…similarly for other relevant instructions

Import/Export

Recursive groups of types can be ex/imported

either as a whole

or their projections individually

But no after-the-fact recursion between imports and other definitions

Recursion after the fact

```
(module $A
  (type $t (import "t"))
  (type $u (struct (field (ref $t))))
)

(module $B
  (type $t (struct (field (ref $u))))
  (type $u (struct (field (ref $t))))

  ;; possible under equi-recursion, but not expressible with iso-recursion:
  (instance (module $A) (import "t" (type $t)))
)
```

Post-MVP: Type Parameters

```
type $pair = (param $a) struct (ref $a) (ref $a)
```

Iso-recursion does not equate type with its unrolling,
hence readily allows for non-uniform recursion

```
type $u = (rec
  (param $a) struct (ref (type $u $a))    ;; uniform recursion
)

type $t = (rec
  (param $a) struct (ref (type $t (type $pair $a)))   ;; non-uniform recursion
)
```

"Higher-order" Iso-Recursion

Generalize μ to "higher-order"

$\mu(\mathrm{self}).\ \langle \lambda(\alpha_1{}^*)\ t_1,\ \ldots\ ,\ \lambda(\alpha_N{}^*)\ t_N \rangle$

Unrolling performs type application on μ types:

struct.get i : [(ref (type $t.*n* *ht**))]   iff   unroll($t)[n](ht*) = struct $t^i\ t\ t^*$

Can be implemented without performing substitutions under μ

Iso-recursion, alternate design

New form of recursive type definition listing only indices:

deftype ::= **func** *valtype** | **struct** *fieldtype** | **array** *fieldtype* | **rec** *typeidx**

A *deftype* other than **rec** may only refer to smaller type indices,
or type indices from a **rec** it occurs in

A **rec** *deftype* must list a consecutive range of type indices

Pro: No extension to heap types necessary, recursive types flattened into index space

Cons:

- structure less apparent, more work to reconstruct it (in both engines and semantics/spec)
- difference of meaning of type “equations” inside and outside recursion not apparent

Iso-recursion, alternate design

```
type $r = rec $t $u
type $t = struct (ref $u)
type $u = struct (ref $t)

type $v = struct (ref $t)

;; $v ≠ $u
```

Does $x refer to a recursive type? Looking it up does not suffice.

May work, but is a bit more hacky.

Summary: Equi-recursion

Pros

- Canonical notion of equivalence, natural generalisation of semantics of function types
- Maximally permissive, no “artificial” restrictions — cannot get in the way
- mutual recursion comes for free
- no additional constructs or constraints needed for type section in MVP

Cons

- More expensive worst-case for checking equivalence (O(n log n) vs O(n))
- More expensive type canonicalization (O(n + e log n) vs O(n), worse incrementally)
- Additional constructs/constraints still arise once we add type parameters

Summary: Iso-recursion

Pros

- Inductive definitions of equivalence and subtyping
- Cheap bottom-up canonicalization
- Can readily be extended with type parameters and supports non-uniform recursion

Cons

- More bureaucracy, supporting mutual recursion requires extra features
- Mutual recursion becomes order-dependent, producers may need to canonicalize order
- Recursion cannot be introduced after the fact
- Problematic for compiling source languages that actually have equi-recursive types

Extended Play

Syntax

deftype ::= … | **rec** *deftype**

heaptype ::= … | *typeidx* . *n*

Type Validation

$C \vdash (\textbf{rec}\ \mathit{deftype}^k)\ \mathit{deftype}^*$ ok
  iff $(C, \text{type}\ (\mathit{deftype}^k) \vdash \mathit{deftype})^k$
  and $C, \text{type}\ (\mathit{deftype}^k) \vdash \mathit{deftype}^*$ ok

$C \vdash \mathit{deftype}\ \mathit{deftype}^*$ ok
  iff $C \vdash \mathit{deftype}$ ok
  and $C, \text{type}\ \mathit{deftype} \vdash \mathit{deftype}^*$ ok

$C \vdash x.n$ ok
  iff $n < |C(x)|$

Subtyping

$C; A \vdash x_1.n <: x_2.n$
  iff $C; A \vdash x_1 <: x_2$

$C; A \vdash x_1 <: x_2$
  iff $x_1 <: x_2 \in A$
  or $C; A, x_1 <: x_2 \vdash C(x_1) <: C(x_2)$

$C; A \vdash (\mathbf{rec}\ deftype_1{}^*) <: (\mathbf{rec}\ deftype_2{}^*\ deftype_3{}^*)$
  iff $(C \vdash deftype_1 <: deftype_2)^*$

Instruction Validation

$C \vdash$ **struct.new_default** : [(rtt $x.n$)] $\rightarrow$ [(ref $x.n$)]
    iff unroll(x)[n] = **struct** ($mut\ t$)*

$C \vdash$ **struct.get** i : [(ref null$^?$ $x.n$)] $\rightarrow$ [t]
    iff unroll(x)[n] = **struct** ($mut_1\ t_1$)^i ($mut\ t$) ($mut_2\ t_2$)*

unroll(**rec** $deftype^*$) = $deftype^*$
unroll($deftype$) = $deftype$

Outtakes

Mutual Equi-Recursion

type \$t = struct i32 (ref \$u)
type \$u = struct f64 (ref \$t)

\$t = μ(t). struct i32 (ref (struct f64 (ref t)))

\$u = μ(u). struct f64 (ref (struct i32 (ref u)))

\$t == struct i32 (ref (struct f64 (ref \$t))) == struct i32 (ref \$u)

Mutual Iso-Recursion

```
type $tu = (rec
  struct i32 (ref $tu.1)
  struct f64 (ref $tu.0)
)
```

$tu = μ(tu). ⟨ struct i32 (ref tu.1), struct f64 (ref tu.0) ⟩

$t = $tu.0
$u = $tu.1

unroll($tu.0) = struct i32 (ref $tu.1)
unroll($tu.1) = struct f64 (ref $tu.0)