14-1 $R_3 \Rightarrow R_4$

dom rng

ty := ... | (→ (ty ...) ty)

e := ... | (app e e ...)

def := (define var [var : ty] ...) : ty e)

p := (program info (def ...) e)

(program _

((define (even? [x : S64]) : Bool (even? 99))

(if (== x 0) true

(odd? (- x 1))))

(define (odd? [x : S64]) : Bool

(if (== x 0) false

(even? (- x 1)))))

14-2) old: $\Gamma \vdash e : \tau$

new: $\Delta, \Gamma \vdash e : \tau$

Δ: top-level funs; Γ: local

$$\frac{\vdash (\text{program}\ \_\ [\text{def}_n)\ e) : T}{\begin{array}{l}\Delta \vdash \text{def} \\ \Delta, \emptyset \vdash e : T \\ \Delta = \emptyset\,[\text{def}_0 \mapsto (\mapsto (\text{dom}_{0,0} \cdots \text{dom}_{0,n})\ \text{rng}_0)] \cdots\end{array}}$$

$$\frac{\begin{array}{l}\Gamma = \emptyset\,[x_0 \mapsto \tau_0] \cdots [x_n \mapsto \tau_n] \\ \Delta, \Gamma \vdash e : r\tau\end{array}}{\Delta \vdash (\text{define}\ (v\ [x_0 : \tau_0] \cdots [x_n : \tau_n]) : r\tau\ e)}$$

143/

$$\frac{\Gamma(x) = +}{\Delta, \Gamma \vdash x : +} \qquad \frac{\Gamma(x) = \bot \quad \Delta(x) = +}{\Delta, \Gamma \vdash x : +} \rightarrow \text{(fun-ref x)}$$

$$\frac{\Delta, \Gamma \vdash e_{rator} : (\Rightarrow (t_{y0} \ldots t_{yn}) \; r_{ty}) \quad \Delta, \Gamma \vdash e_{r0} : t_{y0} \quad \ldots \quad \Delta, \Gamma \vdash e_{rn} : t_{yn}}{\Delta, \Gamma \vdash (\text{app } e_{rator} \; e_{r0} \ldots e_{rn}) : r_{ty}}$$

```
(define (f [x : S64]) : S64 (+ x 1))
(define (g [x : S64]) : S64 (* x x))
( (if (== (read) 0) f g) (read))
```

14-4/

randp :

- sometimes generate N random funs
- each one returns a random type
- takes a few arguments

f : int * int -> int

g : bool * bool -> int

h : int * bool -> bool

14-5

optimization — inlining

(app erator e0 ... en)

know that (after opt) erator =

⟹

(let x0 = e0
...
xn = en
in
e.body)

(define (_ [x0 : ty0]
...
[xn : tyn])
: rty
e.body)

(define (f [x : S64]) : S64 (f 0)) ⟶ (+ x 1)

14-6) uniquify treats each fun seperately

reveal-funs : find all references to funs
and replace them with (fun-ref x)

⇒ uniquify / reveal funs ⟶ typec
typed / reveal-funs → uniquify
typec → uniquify → reveal-funs

14-7/ limit-funs

On x86-64 funs always have 6 arguments:

%rdi %rsi %rdx %rcx %r8 %r9

Ry: (f a b c d e f g h)

⇓

(f a b c d e (vector f g h))

① modify fun defns

② modify fun app

14-8 /

```
(define (f [a:ta] [b:tb] [c:tc] [d:td] [e:te]
           [f:tf] [g:tg]) : rty  e_body)
```

⇓

```
(define (f [a:ta] [b:tb] [c:tc] [d:td] [e:te]
           [rest : (vector tf tg)]) : rty
  (let f = (vector-ref rest 0)
       g = (vector-ref rest 1)
  in
    e_body))
```

14-9)

(f a_1 a_2 a_3 a_4 a_5 a_6 a_7)

⇓

(f a_1 a_2 a_3 a_4 a_5 (vector a_6 a_7))

14-10 / rco

1) let fun-refs be a new simple cat

2) old (program info e) ⇝ (program info e')

now: (program info (def ...) e)

⇓

(program info (def' ...
(define (main) : ty e)) (main))

3) deal w/ app

14-11 /

rcoa σ tail? (app rator rand0 ... randn) =

let rand_res = map (rcoa σ false) (rand0 ... randn)

nvrator, arator = rcoa σ false rator

nvrands = append (map fst rand_res)

arands = map snd rand-res

nvs = nvrator ++ nvrands

if tail? then

(nvs, (app arator arands))

o.w.

(nvs ++ [(ans, (app arator arands))], ans)