

The "simplification" algorithm

- Purpose:

- flatten name space
- remove λ , i.e. (do abstraction)
- put everything in normal form
- ensure that r.h.s.s of defs are functions

- Functionality:

$S(\text{p-tree}, \text{bindings}, \text{def-array}) \rightarrow (\text{r-node}, \text{def-array})$

Data structures:

- 1) a - array of definitions, each of which is a structure of the form:

lhs	name (charstring)
<u>rhs</u>	r-node - decorated with:
isfnf	flag for "function in normal form"; (set <u>false</u> for λ -vars)
isnf	flag for "is in normal-form"

- 2) b - list of bindings, each element of which is a pair of the form:

("n" r-node)

note usually will be simply

("n" (fn #))
or ("n" (pfu #))

but the more general form is used to support lambda-removal.

Definition of S , by cases of p -trees

Atoms: $S(p\text{-tree}, b, a) = (r\text{-tree}, a)$

where $\frac{p\text{-tree}}{t} \quad (mk\text{-bool-node } :t)_{\text{FNF}}^{NF+}$

$f \quad (mk\text{-bool-node } :f)_{\text{FNF}}^{NF+}$

(no "number") $(mk\text{-num-node } (char2num \text{ num } b))_{\text{FNF}}^{NF+}$

$\# \backslash c \quad (mk\text{-char-node } \# \backslash c)_{\text{FNF}}^{NF+}$

Expressions

$$S((app\ e_1\ e_2), b, a) = ((app\ e_1'\ e_2'), a'')$$

where $(e_1', a') = S(e_1, b, a)$

$$(e_2', a'') = S(e_2, b, a')$$

$$S((seq\ e_1, \dots, e_n), b, a) = \text{"similar"}$$

$$S((\text{fn "name"}), b, a) =$$

$$\begin{cases} ((\text{get "name" } b), a) & \text{if "name" } \in b \\ ((\text{fn \#new}), a') & \text{o.w.} \end{cases}$$

where $a' = a ++ \text{ new entry} :$

lhs = "name"

rhs = memoize["name", "undef", 10]

To handle environments:

$$S_{aux} (env, b, a) \rightarrow (b', a')$$

$$e.g.: env \equiv \{ (rec f_1, e_1), (rec f_2, e_2) \}$$



2) i) $set (e_1', a'') = S(e_1, b', a')$

ii) add $\overline{F}(e_1')$ as rhs of $\#f_1$ in a''

3) i) $set (e_2', a''') = S(e_2, \underline{b}, a'')$

ii) add $\overline{F}(e_2')$ as rhs of $\#f_2$ in a'''

4) return (b', a''')

$$S((\text{whereexp } e \text{ env}), b, a) = S(e, b', a')$$

$$\text{where } (b', a') = S_{\text{aux}}(\text{env}, b, a)$$

$$S((\text{func } e), b, a) = (\widehat{F}(e'), a')$$

$$\text{where } (e', a') = S(e, b, a)$$

WWP?

$S((\text{lamb env } e), b, a)$

- 1) set $(b', a') = S.\text{aux}(\text{env}, b, a)$
- 2) for each function in $b' - b$, apply $\hat{\tau}$ and γ to its rhs in a' .
- 3) set $a'' = a++ (\#next = (\text{pfu} * \text{id}))^{Fnt = \text{false}}$
- 4) set $b'' = b'$ with all bindings in $b' - b$
($\hookrightarrow (b_i, r_i)$) replaced by $(b_i (\text{app } r_i \#next))$
- 5) set $(r', a''') = S(e, a'', b'')$
- 6) set $a'''' = a'''$ where each rhs of $a''' - a''$
has first been $\hat{\tau}$ ed and then
 $\text{abs}_{(\#next, \#s \text{ of } a''' - a'')}$ 
- 7) return $(\text{abs}_{(\#next, \#s \text{ of } a''' - a'')}(r'), a''')$

WWP8

Definition of $\mathcal{F}$ using auxiliary $\mathcal{K}$:

note: both map $r\text{-nodes} \rightarrow r\text{-nodes}$

r	$\mathcal{F}(r)$	$\mathcal{K}(r)$
atoms	$(K:r) : 'id$	$K:r$
$\#f$	$(K:r) : 'id$ if $\text{is_nf} = \text{false}$ r o.w.	$K:r$
$\#pf$	r	$K:r$
$\langle r_1, \dots, r_n \rangle$	$\mathcal{K}(r) : 'id$	$[\mathcal{K}(r_1), \dots, \mathcal{K}(r_n)]$ $(K:\langle r_1, \dots, r_n \rangle$ if all the r_i in <u>nf</u>)
$\langle \text{app } r_1, r_2 \rangle$	$\mathcal{K}(r) : 'id$	$\mathcal{K}(r_1) : ' \mathcal{K}(r_2)$ note: $(r_1, o(K:r_2))$ if both in <u>nf</u> $(r_1, o(\mathcal{K}(r_2)))$ if just r_1 in <u>nf</u>

wwp9

Abstracting wrt a list of function #s :

$$\text{abs}_Q(a) = k:a$$

$$(f) = \begin{matrix} f & \text{if } f \in Q \\ k:f & \text{o.w.} \end{matrix}$$

$$(x_1 : x_2) = \text{abs}_Q(x_1) : \text{abs}_Q(x_2)$$

$$(\langle x_1, \dots, x_n \rangle) = \text{cons} : \langle \text{abs}_Q(x_1), \dots, \text{abs}_Q(x_n) \rangle$$