FPL

Lars Hansen

MODULE FPL-VALUE

Special rules for call by value.

lambda derivation

SYNTAX $K ::=$ `lambda` *Ids* `->` *Block*(`ans`, *VExps*)

RULE $\dfrac{\texttt{lambda}\ X\texttt{,}\ Xs\ \texttt{->}\ B(E\texttt{,}\ Es)}{E \curvearrowright \texttt{lambda}\ X\texttt{,}\ Xs\ \texttt{->}\ B(\texttt{ans}, Es)}$

lambda read ans

RULE k: $\dfrac{\texttt{lambda}\ Xs\ \texttt{->}\ B(\texttt{ans}, Es)}{\texttt{lambda}\ Xs\ \texttt{->}\ B(V\texttt{,}\ Es)}$ ans: V

lambda partial application

RULE $\dfrac{\texttt{lambda}\ X\texttt{,}\ Xs\ \texttt{->}\ \{B\}(V\texttt{,}\ Es)}{\texttt{lambda}\ Xs\ \texttt{->}\ \{(X\ \texttt{:=}\ V)\ \texttt{;}\ B\}(Es)}$

lambda application

RULE k: $\dfrac{\texttt{lambda}\ \bullet_{Ids}\ \texttt{->}\ \{B\}(\bullet_{VExps}) \curvearrowright K}{B}$ env: $\dfrac{Env}{\bullet_{Map}}$ fstack: $\dfrac{\bullet_{List}}{(Env, K)}$

variable definition

RULE k: $\dfrac{(N\ \texttt{:=}\ V)}{V}$ env: $\dfrac{\bullet_{Map}}{(N \mapsto L)}$ store: $\dfrac{\bullet_{Map}}{(L \mapsto V)}$ requires `fresh` (L)

function definition

RULE k: $\dfrac{\texttt{function}\ N(Args)B}{\bullet_{K}}$ genv: $\dfrac{\bullet_{Map}}{(N \mapsto L)}$ store: $\dfrac{\bullet_{Map}}{(L \mapsto \texttt{lambda}\ Args\ \texttt{->}\ B)}$ requires `fresh` (L)

END MODULE

MODULE FPL-NAME

Special rules for call by name.

lambda partial application

RULE $\dfrac{\texttt{proc}\ X\texttt{,}\ Xs\ \texttt{->}\ \{B\}(E\texttt{,}\ Es)}{\texttt{proc}\ Xs\ \texttt{->}\ \{\ \texttt{set}\ X\ \texttt{=}\ (\ \texttt{\$arg}\ E)\ \texttt{;}\ B\}(Es)}$

lambda application

RULE k: $\dfrac{\texttt{proc}\ \bullet_{Ids}\ \texttt{->}\ \{B\}(\bullet_{VExps}) \curvearrowright K}{B \curvearrowright \texttt{execute}}$ env: $\dfrac{Env}{\bullet_{Map}}$ fstack: $\dfrac{\bullet_{List}}{(Env, K)}$ reading: $\dfrac{\texttt{false}}{\texttt{true}}$

variable definition

RULE k: $\dfrac{\texttt{set}\ N\ \texttt{=}\ (E)}{\bullet_{K}}$ env: $\dfrac{\bullet_{Map}}{(N \mapsto L)}$ store: $\dfrac{\bullet_{Map}}{(L \mapsto (\ \texttt{\%}, E))}$ requires `fresh` (L)

function definition

RULE k: $\dfrac{\texttt{sub}\ N(Args)B}{\bullet_{K}}$ genv: $\dfrac{\bullet_{Map}}{(N \mapsto L)}$ store: $\dfrac{\bullet_{Map}}{(L \mapsto \texttt{proc}\ Args\ \texttt{->}\ B)}$ requires `fresh` (L)

base variable lookup

RULE k: $\dfrac{N}{E}$ env: $N \mapsto L$ store: $L \mapsto (\ \texttt{\%}, E)$ reading: `false`

END MODULE

MODULE FPL-NEED

Special rules for call by need.

lambda partial application

RULE $\dfrac{\texttt{\textbackslash}\ X\texttt{,}\ Xs\ \texttt{->}\ \{B\}(E\texttt{,}\ Es)}{\texttt{\textbackslash}\ Xs\ \texttt{->}\ \{\ \texttt{let}\ X\ \texttt{=}\ (\ \texttt{\$arg}\ E)\ \texttt{;}\ B\}(Es)}$

lambda application

RULE k: $\dfrac{\texttt{\textbackslash}\ \bullet_{Ids}\ \texttt{->}\ \{B\}(\bullet_{VExps}) \curvearrowright K}{B \curvearrowright \texttt{execute}}$ env: $\dfrac{Env}{\bullet_{Map}}$ fstack: $\dfrac{\bullet_{List}}{(Env, K)}$ reading: $\dfrac{\texttt{false}}{\texttt{true}}$

variable definition

RULE k: $\dfrac{\texttt{let}\ N\ \texttt{=}\ (E)}{\bullet_{K}}$ env: $\dfrac{\bullet_{Map}}{(N \mapsto L)}$ store: $\dfrac{\bullet_{Map}}{(L \mapsto E)}$ requires `fresh` (L)

function definition

RULE k: $\dfrac{\texttt{def}\ N(Args)B}{\bullet_{K}}$ genv: $\dfrac{\bullet_{Map}}{(N \mapsto L)}$ store: $\dfrac{\bullet_{Map}}{(L \mapsto \texttt{\textbackslash}\ Args\ \texttt{->}\ B)}$ requires `fresh` (L)

thunk variable lookup

RULE k: $\dfrac{N}{E \curvearrowright N\ \texttt{<-}\ \texttt{ans}}$ env: $N \mapsto L$ store: $L \mapsto E$ reading: `false`

thunk variable override

RULE k: $\dfrac{N\ \texttt{<-}\ \texttt{ans}}{V}$ ans: V env: $N \mapsto L$ store: $L \mapsto \dfrac{—}{V}$

END MODULE

MODULE FPL-SYNTAX

This is a functional language with three different methods of function evaluation:

1. Call by Value: Parameters are reduced to base values, then assigned to the argument variables. 2. Call by Name: In the function body, argument names are substituted with parameter values. 3. Call by Need: Parameters are assigned to argument values and only evaluated on usage. Once evaluated, the value is stored instead of the expression.

Call by Value functions are eagerly evaluated, Call by Name and Call by Need are lazyly evaluated.

FPL currently has the following features: - variable assignment (no override, all 3 evaluation methods) - variable lookup - standard arithmetics (+,-,*,/) (call by need) - branching (if, including comparison operator <) - anonymous functions (all 3 evaluation methods) - named functions (all 3 evaluation methods) - eager lists (including predefined functions empty,head and tail)

To demonstrate the different evaluation styles, this language also includes pre-increment ++x, thereby introducing side effects to the language. It is noteworthy that pre-increment may also be skipped in lazy-evaluated contexts. This is somewhat counter-intuitive, so let's analyze the alternatives: 1. execute on read: does not comply with project specification, as call-by-name square(++x) would be equal to call-by-value square(++x) (see test.fpl) 2. execute on variable read: this requires the data structure to tell numeric variables from other kinds and mark them as tainted, thus creating considerable overhead

top level program expressions:
Expressions, variable and function definitions, halt program and sequential composition

SYNTAX *PgmExp* ::= *VExp*
| *Let*
| *Funct*
| `halt`
| *PgmExp* `;` *PgmExp*

values or expressions which return values

SYNTAX *VExp* ::= *Exp*
| *Val*

expressions which return values:
`Id` variable lookup
`++Id` pre-increment (with side-effects!
`AOp VExp VExp` arithmetics in polish notation
`VExp : VExp` haskell-style list extension
`VExp ( VExps )` function / lambda call
`if VExp then Block else Block` if condition
`$arg VExp` evaluate expression in caller context (e.g. function arguments)
pre-defined list functions:
`empty ( VExp )` check if list is empty
`head ( VExp )` get head of list
`tail ( VExp )` get tail of list

SYNTAX *Exp* ::= *Id*
| `++` *Id*
| *AOp VExp VExp*
| *VExp* `:` *VExp*
| `empty` (*VExp*)
| `head` (*VExp*)
| `tail` (*VExp*)
| *VExp*(*VExps*)
| `if` *VExp* `then` *Block* `else` *Block*
| `$arg` *VExp*
| (*VExp*) [bracket]

variable assignment
`let Id = ( VExp )` for call by need
`set Id = ( VExp )` for call by name
`(Id := VExp)` for call by value

SYNTAX *Let* ::= `let` *Id* `=` (*VExp*)
| `set` *Id* `=` (*VExp*)
| *Id* `:=` *VExp* [strict(2)]

function definition
`def Id ( Ids ) Block` for call by need
`sub Id ( Ids ) Block` for call by name
`function Id ( Ids ) Block` for call by value

SYNTAX *Funct* ::= `def` *Id*(*Ids*)*Block*
| `sub` *Id*(*Ids*)*Block*
| `function` *Id*(*Ids*)*Block*

anonymous function definition
`\\ Ids -> Block` for call by need
`proc Ids -> Block` for call by name
`lambda Ids -> Block` for call by value

SYNTAX *Lambda* ::= `\` *Ids* `->` *Block*
| `proc` *Ids* `->` *Block*
| `lambda` *Ids* `->` *Block*

list [eager]

SYNTAX *Li* ::= [*VExps*] [strict]

code block for conditions and functions

SYNTAX *Block* ::= {*PgmExp*}

predefined functions

SYNTAX *Predef* ::= `empty`
| `head`
| `tail`

SYNTAX *Ids* ::= $List\{Id, \text{“,”}\}$

SYNTAX *VExps* ::= $List\{VExp, \text{“,”}\}$ [strict]

arithmetic operations (call by need)

SYNTAX *AOp* ::= `*`
| `/`
| `+`
| `-`
| `<`

base values (string used for comments only)

SYNTAX *Val* ::= *Int*
| *Bool*
| *String*
| *Lambda*
| *Li*

END MODULE

MODULE FPL-CONFIG

SYNTAX *KResult* ::= *Val*

additional syntax for internal processes: *execute* starts evaluation in functions with lazy evaluation
ans allows insertion of value from ans cell (like HOLE)
continue returns evaluation in another scope
(%,K) stores call-by-name values
Id <- VExp overrides memory of variables

SYNTAX K ::= `execute`
| `ans`
| (*Map*, *K*)
| `continue` (*Map*, *K*)
| (`%`, *K*)
| *Id* `<-` *VExp*

k program (eager evaluation in main scope)
env scoped environment
genv global environment (functions only)
store heap memory
fstack function stack, each element with (environment,remaining commands)
reading reading mode for lazy evaluation
ans answer value of last expression

CONFIGURATION:

T: [k: $\$PGM$ | env: $\bullet_{Map}$ | genv: $\bullet_{Map}$ | store: $\bullet_{Map}$ | fstack: $\bullet_{List}$ | reading: `false` | ans: `false`]

END MODULE

MODULE FPL

pre-increment (with side-effects!)

RULE $\dfrac{\texttt{++}\ N}{(\ \texttt{+}\ \ N\ \ \texttt{1}) \curvearrowright N\ \texttt{<-}\ \texttt{ans}}$

halting

RULE k: $\dfrac{\texttt{halt}}{\square}$ fstack: $\dfrac{\bullet_{List}}{\texttt{"halt"}}$

sequential composition

RULE $\dfrac{X\ \texttt{;}\ Y}{X \curvearrowright Y}$ [structural]

end of reading program

RULE k: $\dfrac{\texttt{execute}}{A}$ ans: A reading: $\dfrac{\texttt{true}}{\texttt{false}}$

base case integer rules

RULE $\dfrac{(\ \texttt{+}\ \ X\ \ Y)}{X +_{Int} Y}$

RULE $\dfrac{(\ \texttt{-}\ \ X\ \ Y)}{X -_{Int} Y}$

RULE $\dfrac{(\ *\ \ X\ \ Y)}{X *_{Int} Y}$

RULE $\dfrac{(\ \texttt{/}\ \ X\ \ Y)}{X \div_{Int} Y}$ requires $Y =/=_{Int} 0$

comparison

RULE $\dfrac{\texttt{<}\ \ X\ \ Y}{X <_{Int} Y}$

conditional

RULE $\dfrac{\texttt{if true then}\ \{E\}\ \texttt{else}\ —}{E}$

RULE $\dfrac{\texttt{if false then}\ —\ \texttt{else}\ \{E\}}{E}$

lambda - end of execution, garbage collect

RULE k: $\bullet_{K}$ env: $\dfrac{(N \mapsto L)}{\bullet_{Map}}$ store: $\dfrac{(L \mapsto —)}{\bullet_{Map}}$ fstack: —

lambda - end of execution, change scope

RULE k: $\dfrac{\bullet_{K}}{V \curvearrowright K}$ env: $\dfrac{\bullet_{Map}}{Env}$ fstack: $\dfrac{(Env, K)}{\bullet_{List}}$ ans: V

necessary for HOLE in eager lists

list extension

RULE $\dfrac{X\ \texttt{:}\ [Els]}{[X\ \texttt{,}\ Els]}$

predef functions

RULE $\dfrac{\texttt{empty}\ ([\bullet_{VExps}])}{\texttt{true}}$

RULE $\dfrac{\texttt{empty}\ ([E\ \texttt{,}\ Es])}{\texttt{false}}$

RULE $\dfrac{\texttt{head}\ ([E\ \texttt{,}\ Es])}{E}$

RULE $\dfrac{\texttt{tail}\ ([E\ \texttt{,}\ Es])}{[Es]}$

write ans

RULE k: $\dfrac{(V \curvearrowright E)}{E}$ ans: $\dfrac{—}{V}$

skip over all expressions while reading program

RULE k: $\dfrac{X \curvearrowright Y}{Y}$ ans: $\dfrac{—}{X}$ reading: `true`

get value from different scope

RULE k: $\dfrac{\texttt{\$arg}\ E}{E \curvearrowright \texttt{continue}\ (CurEnv, K)}$ env: $\dfrac{CurEnv}{Env}$ fstack: $\dfrac{(Env, K)}{\bullet_{List}}$

return to scope

RULE k: $\dfrac{\texttt{continue}\ (Env, K)}{\bullet_{K}}$ env: $\dfrac{CurEnv}{Env}$ fstack: $\dfrac{\bullet_{List}}{(CurEnv, K)}$

base variable lookup

RULE k: $\dfrac{N}{V}$ env: $N \mapsto L$ store: $L \mapsto V$ reading: `false`

global variable lookup

RULE k: $\dfrac{N}{E}$ genv: $N \mapsto L$ store: $L \mapsto E$ reading: `false`

END MODULE