Tópicos em LP

Fabio Mascarenhas – 2018.1

http://www.dcc.ufrj.br/~fabiom/comp2

SmallLua

```
bloco <- stat* ret? -> list
   stat  <- # "while" exp "do" bloco "end" -> nwhile
       / # "local" id '=' exp -> nlocal
       / # "if" exp "then" bloco ("else" bloco)? "end" -> nif
       / # id '=' exp -> nassign
       / # pexp -> nstatexp
ret   <- # "return" exp -> nret
ids   <- id (',' id)* -> list
exps  <- exp (',' exp)* -> list
exp   <- lexp (# {"or"} -> op lexp)* -> chainl
lexp  <- rexp (# {"and"} -> op rexp)* -> chainl
rexp  <- cexp (# rop -> op cexp)* -> chainl
cexp  <- aexp # '..' cexp -> nconcat  / aexp
aexp  <- mexp (# aop -> op mexp)* -> chainl
mexp  <- sexp (# mop -> op sexp)* -> chainl
sexp  <- '-' sexp -> nunm / "not" sexp -> nnot / "false" -> nfalse / "true" -> ntrue
    / number -> nnumber / lmb / "nil" -> nnil / string -> nstring / pexp
lmb   <- # "function" '(' (ids? -> opt) ')' bloco "end" -> nlmb
pexp  <- ('(' exp ')' / # id -> nvar) (# '(' -> ncall (exps? -> opt) ')')* -> chainl
rop   <- {'<'} / {'=='} / {'~='}
aop   <- {'+'} / {'-'}
mop   <- {'*'} / {'/'}

string <- sp { string = [squote] (![squote] .)* [squote] / [dquote] (![dquote] .)* [dquote] }
number <- sp { number = [isdigit]+ (['.'] [isdigit]+)? }
id     <- !kws sp { id = [isidstart] [isidrest]* }
sp     <- [isblank]*

kws    <- "while" / "end" / "if" / "then" / "else" / "local" / "true" / "false" / "nil"
   / "function" / "not" / "and" / "or" / "do" / "return"
```

Regras de dedução

- As regras de dedução de um sistema de tipos dão um esquema de como podemos *deduzir* o tipo de uma expressão dados os tipos de suas subexpresões

- Tradicionalmente usamos uma notação "barra" para as regras de dedução, em que as hipóteses da regra ficam acima de uma barra horizontal e a conclusão abaixo dessa barra

- Tanto as hipóteses quanto a conclusão são escritas da forma $\vdash e : t$, onde *e* é uma expressão, *t* um tipo e o símbolo $\vdash$ é a "roleta"

  - Lê-se "pode-se provar que *e* tem tipo *t*"

literal numérico

$$\vdash n : \text{number}$$

$$\frac{\vdash e_1 : \text{number} \qquad \vdash e_2 : \text{number}}{\vdash e_1 + e_2 : \text{number}}$$

Tipagem de variáves

- Qual o tipo de uma variável?
- Não podemos determinar esse tipo sintaticamente, ele depende do *contexto*
- Vamos dar esse contexto usando um ambiente de tipos que associa nomes de variáveis a seu tipo:

$$\frac{(x \to t) \in T}{\mathrm{T} \vdash x : t}$$

- Declarações de variáveis *estendem* o ambiente:

$$\frac{\mathrm{T} \vdash e : t}{\mathrm{T} \vdash local\ x = e : T[x \to t]}$$

Comandos e blocos

- Comandos não têm tipo, mas podem mudar o ambiente
- Em uma sequência de comandos, vamos “costurando” o ambiente por cada comando da sequência
- Para o comando `return`, o tipo de retorno fica em uma variável especial no ambiente
- Comandos que têm blocos (`if` e `while`) “jogam fora” o ambiente ao final de seus blocos, para manter as regras de escopo

$$\frac{T_1 \vdash s_1 : T_2 \quad T_2 \vdash s_2 : T_3}{T_1 \vdash s_1\ s_2 : T_3}$$

$$\frac{T \vdash e : t \quad T(\$) = t}{T \vdash \texttt{return}\ e : T}$$

Funções

- Para tipar a declaração de uma função, estendemos o ambiente associando o nome de cada parâmetro a seu tipo, assim como seu tipo de retorno

- Depois tipamos o corpo da função, e checamos se o tipo bate com o tipo de retorno declarado

- Tipar a chamada de uma função é mais simples, bastando verificar que o número de argumentos bate com o de parâmetros (aridade), e o tipo de cada argumento bate com o de cada parâmetro

$$\frac{T[\overline{x \to t}, f \to t_r] \vdash b : T_b}{T \vdash \text{function}(\overline{x:t})\ b\ \text{end} : (\overline{t}) \to t_r}$$

$$\frac{T \vdash e_f : (\overline{t_p}) \to t_r \qquad T \vdash \overline{e_a : t_a} \qquad \overline{t_p = t_a}}{T \vdash e_f(\overline{e_a}) : t_r}$$

Exercício

- Implementar o restante do verificador de tipos de SmallLua, como uma função definida por casos

- O verificador deve produzir o ambiente com as declarações do programa, ou lançar um erro com o primeiro erro de tipo encontrado

Funções recursivas

- Podemos especializar a regra de `local` para permitir que ela defina funções recursivas

- Em Lua essa especialização tem uma sintaxe especial

Sequências

- Para o sistema de tipos de SmallLua ficar interessante, vamos acrescentar um tipo estruturado: *sequências*

- Um tipo `{ t }` é uma sequência de itens de tipo t

```
tseq  <- # "{" type "}" -> ntseq
```

- Construímos sequências com os operadores `seq_new(...)`, que recebe os elementos da sequência como argumentos, e `..`, que concatena duas sequências

- Decompomos sequências com os operadores `seq_byte(s, i)`, que retorna o i-ésimo elemento da sequência, e `seq_sub(s, i, j)`, que retorna a subsequência entre os elementos i e j (inclusive)