Tópicos em LP

Fabio Mascarenhas – 2018.1

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

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)

Regras de dedução - sequências

$$\frac{\Gamma \vdash e_1 : t \quad \ldots \quad \Gamma \vdash e_n : t \quad n > 0}{\Gamma \vdash new(e_1, \ldots, e_n) : \{t\}}$$

$$\frac{\Gamma \vdash s : \{t\} \quad \Gamma \vdash i : number}{\Gamma \vdash byte(s, i) : t}$$

$$\frac{\Gamma \vdash s : \{t\} \quad \Gamma \vdash i : number \quad \Gamma \vdash j : number}{\Gamma \vdash sub(s, i, j) : \{t\}}$$

Primitivas vs funções

- Classificamos `seq_new`, `seq_byte` e `seq_sub` como primitivas, ao invés de funções, por quê?
- Qual seria o tipo de `seq_new()` (uma sequência vazia)?
- Nosso sistema de tipos é *monomórfico*: cada termo do sistema pode ter apenas um tipo, e isso vale para as funções
- Logo, se `seq_byte` e `seq_sub` fossem funções só poderiam receber sequências de um tipo pré-determinado, e `seq_new()` sempre retornaria uma sequência do mesmo tipo
- Podemos resolver esses problemas com um sistema *polimórfico*

Parâmetros de tipos

- A ideia do *polimorfismo paramétrico* é poder ter *parâmetros* nos tipos, que são “buracos” onde podemos encaixar qualquer tipo

- Representamos esses parâmetros com *parâmetros de tipos*

- Um tipo polimórfico tem um ou mais parâmetros como parte dele (podemos usar o mesmo parâmetro mais de uma vez!)

Generalização

- Criamos um tipo polimórfico por um processo de *generalização*

- Devemos tomar cuidado para verificar se os parâmetros de tipos usados foram introduzidos

```
local foo = function <a> (x: a): {number}
    local y = seq_new(x)
    return seq_new(1) .. y
end

local bar = function <a> (x: a): number
    local z = function (y: a): a
                  return x
              end
    return z(1)
end
```

Especialização

- O inverso da generalização é a *especialização*, que precisamos fazer toda vez que queremos *usar* um valor de um tipo polimórfico: no nosso caso, chamar uma função polimórfica, ou passar uma função com tipos polimórficos como argumento

- Na especialização, substituimos os parâmetros livres por tipos concretos

- O problema é: quais?

- Uma alternativa é incluir uma sintaxe para especialização:

```
local e = seq_new<string>()
local s = seq_new<number>(1)
local n = seq_byte<number>(s, 1)
```

Unificação

- Em uma chamada de função polimórfica, a ideia é usar os tipos dos argumentos para dizer quais devem ser os tipos que vamos usar na especialização

- Para fazer a atribuição de parâmetros a tipos usamos um algoritmo que mistura casamento de padrões com atribuição: a *unificação*

Unificação - Variáveis

- A unificação é um algoritmo fácil de definir quando trocamos nossos parâmetros de tipo por *variáveis de tipo*
- Uma variável inicialmente não está associada a nenhum tipo, mas pode vir a ficar durante o processo de unificação
- Uma vez associada, ela não pode mais mudar; efetivamente ela assume aquele tipo
- Duas (ou mais) variáveis não associadas podem se unir: quando uma for associada, a outra também fica associada ao mesmo tipo

Unificação - pseudocódigo

```
function unify(t1, t2):
    t1 = prune(t1)
    t2 = prune(t2)
    case t1, t2:
        match var, _:
            if occurs(t1, t2):
                error("ciclo")
            else:
                t1.type = t2
        match _, var: unify(t2, t1)
        match base, base:
            if t1.tag ~= t2.tag:
                error("incompatible")
        match func(params1, ret1), func(params2, ret2):
            if #params1 ~= #params2:
                error("arity")
            zip(unify, params1, params2)
            unify(ret1, ret2)
        match seq(elem1), seq(elem2):
            unify(elem1, elem2)
        otherwise: error("incompatible")
```