Uma linguagem de programação com tipos de sessão independentes do contexto

Bernardo Almeida e Vasco T. Vasconcelos

LASIGE, Faculdade de Ciências, Universidade de Lisboa

4 de Setembro de 2018

Motivação

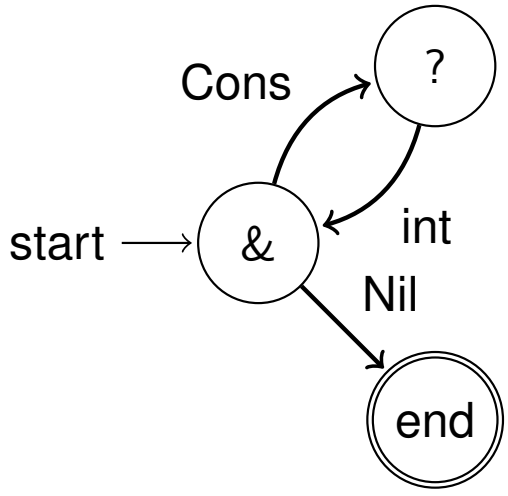
- Tipos de sessão tradicionais têm inúmeras aplicações
- Exemplo: Transmitir uma lista num canal de comunicação

```
type List = Nil | Cons Int List

type ListServer = &{
  Nil : end
  Cons : ?int . ListServer
}
```

Motivação

- Sequência de operações é definida por um autómato finito:



- Linguagem regular: $(\&\text{Cons}\ ?\text{Int})^*\ \&\text{Nil}$

Motivação

- E se o objetivo for transmitir uma árvore?

  type Tree = Leaf | Node Int Tree Tree

- Enviar sequências de Node, Leaf e Int
  ex: Node Leaf 2 Node Leaf

- Comunicação restrita ao envio de tipos básicos e sem passar novos canais

- Os tipos de sessão devem garantir que as árvores estão bem formadas

Motivação

Facto

A linguagem produzida pela gramática que descreve um tipo de sessão é:

- Reconhecida por um autómato finito
- Uma linguagem (ω-) regular

- **Gramática:** N ::= Leaf | Node int N N
- **Consequência:** Os tipos de sessão tradicionais não podem descrever estas estruturas
- **Solução:** Tipos de sessão independentes do contexto propostos por Thiemann e Vasconcelos.

A linguagem

- Sintaxe semelhante à do Haskell
- Acrescida de primitivas para
  - Criação de canais
  - Envio de valores nos canais
  - Receção de valores nos canais
- Primitivas de comunicação: troca de mensagens e escolhas
- Canais de comunicação síncronos e bidirecionais
- Os processos podem escrever numa das extremidades do canal ou ler na outra.

A linguagem

Os tipos disponíveis na linguagem são:

$$
\begin{array}{lll}
B & ::= \mathbf{Int} \mid \mathbf{Char} \mid \mathbf{Bool} \mid () & \text{Tipos básicos} \\
T & ::= \mathbf{Skip} \mid T; T \mid \,!\, B \mid ?B & \text{Tipos} \\
 & \quad \mid \oplus \{l_i\colon T_i\}_{i \in I} \mid \,\&\{l_i\colon T_i\}_{i \in I} & \\
 & \quad \mid B \mid T \to T \mid T \multimap T & \\
 & \quad \mid (T, T) \mid [l_i\colon T_i]_{i \in I} \mid \mathbf{rec}\, \alpha \,.\, T \mid \alpha & \\
\mathcal{C} & ::= T \mid \mathbf{forall}\, \alpha \Rightarrow \mathcal{C} & \text{Esquemas de tipos}
\end{array}
$$

A linguagem - Exemplo

- **Objetivo**: Transmitir uma árvore binária num canal
- **Tipos de dados:**

```
data Tree = Leaf | Node Int Tree Tree
type TreeChannel =
  +{LeafC: Skip,
    NodeC: !Int; TreeChannel; TreeChannel}
```

- **Tipo da função que envia a árvore**

```
sendTree :: forall a =>
  Tree → (TreeChannel; a) → a
```

A linguagem - Envio de uma árvore

O código da função para enviar uma árvore é:

```
sendTree :: forall a =>
  Tree → (TreeChannel; a) → a
sendTree t c =
 case t of
   Leaf → select LeafC c
   Node x l r →
     let c1 = select NodeC c in
     let c2 = send x c1 in
     let c3 = sendTree[TreeChannel;a] l c2 in
     let c4 = sendTree[a] r c3 in
     c4
```

A linguagem - Receção de uma árvore

O código da função complementar que permite receber uma árvore é:

```
type TreeChannelR =
  &{LeafC: Skip, NodeC: ?Int; TreeChannelR ;
      TreeChannelR}
receiveTree :: forall a =>
  (TreeChannelR; a) → (Tree, a)
receiveTree c =
  match c with
    LeafC c1 → (Leaf, c1)
    NodeC c1 →
      let x, c2 = receive c1 in
      let left, c3 = receiveTree [TreeChannelR;a]
          c2 in
      let right, c4 = receiveTree [a] c3 in
      (Node x left right, c4)
```

A linguagem - Sintaxe das expressões

A sintaxe das expressões disponíveis na linguagem é:

$e ::= () \mid \textbf{Int} \mid \textbf{Char} \mid \textbf{Bool}$	Expressões básicas
$\mid x \mid \textbf{let}\ x = e\ \textbf{in}\ e$	Variáveis e Let
$\mid ee \mid e[T]$	Aplicação
$\mid \textbf{if}\ e\ \textbf{then}\ e\ \textbf{else}\ e$	Condicional
$\mid (e, e) \mid \textbf{let}\ x, y = e\ \textbf{in}\ e$	Pares
$\mid \textbf{new}\ T \mid \textbf{send}\ e\ e \mid \textbf{receive}\ e$	Operações de comunicação
$\mid \textbf{select}\ e \mid \textbf{match}\ e\ \textbf{with}\ \{l_i \rightarrow e_i\}_{i \in I}$	
$\mid \textbf{fork}\ e$	Fork
$\mid C \mid \textbf{case}\ e\ \textbf{of}\ \{C_i \rightarrow e_i\}_{i \in I}$	Tipos de dados

A linguagem - Validação

- Sistema de *kinding*:
  - Boa formação dos tipos
  - Classifica os tipos nas categorias de tipos de sessão ou tipos gerais
  - Associa multiplicidades (linear ou partilhado)
- Exemplos:
  - $!\mathbf{Int}$: Bem formado (tipo de sessão, linear)
  - $(\mathbf{Int} \to \mathbf{Bool});\mathbf{Int}$ não é bem formado
  - $\mathbf{rec}\ x.a;x$
    - Mal formado se a não estiver no ambiente de *kinding*
    - Mal formado se a não for um tipo de sessão
- Verificação de tipos

A linguagem - Geração de código

- A linguagem alvo da geração de código é Haskell
- Quatro desafios do processo de tradução:

1. *Call-by-value* VS. *Call-by-name*
   Solução: *BangPatterns*
   ex: `fun x = e` quando traduzida fica `fun !x = e`

2. Canais de comunicação implementadas através de duas *MVar* por canal.
   - putMVar - Operação **send**
   - takeMVar - Operação **receive**

A linguagem - Geração de código

3. *MVar* só têm um tipo que se mantém inalterado durante a computação
   - Canais necessitam que o tipo possa variar (ex: !**Int**;?**Bool** progride para ?**Bool**)
   - Sistema de tipos do Haskell não pode verificar os tipos dos canais (*unsafeCoerce*)

```
_send x (m1, m2) = do
  putMVar m2 (unsafeCoerce x)
  return (m1, m2)

_receive (m1, m2) = do
  a <- takeMVar m1
  return ((unsafeCoerce a), (m1, m2))
```

A linguagem - Geração de código

4. Operações de comunicação (**fork**, **send**, **receive** e **new**) são operações sobre um monáde
   - Quando traduzir uma expressão para código de um monáde?
   - Anotação da árvore sintática com valores booleanos

Geramos código com base na seguinte tabela:

Valor esperado (anotação da árvore sintática)	Valor encontrado (na função de tradução)	Código gerado (Haskell)
False	**False**	e
True	**False**	return e
True	**True**	e
False	**True**	e >>= x → x

A linguagem - Função traduzida

Resultado:

```
sendTree !t !c =
  case t of
    Leaf → _send "LeafC" c
    Node x l r →
      _send "NodeC" c >>=
      \c1 → _send x c1 >>=
      \c2 → sendTree l c2 >>=
      \c3 → sendTree r c3 >>=
      \c4 → return c4
```

Conclusão e Trabalho futuro

Conclusão:

- Linguagem concorrente e explicitamente tipificada
- Comunicação exclusiva por troca de mensagens
- Canais síncronos descritos por tipos de sessão independentes do contexto.

Trabalho futuro:

- Reduzir a verbosidade da linguagem
- Abreviar tipos: **type** SendInt = !**Int**
- Inferência de tipos em alguns cenários (as aplicaçoes de tipos e[T])
- Canais partilhados
- Operador de **dualof**