Linguagens de Programação

Fabio Mascarenhas - 2015.2

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

Exceções

- Vários erros podem acontecer em nossos programas: fazer aritmética com valores que não são números, chamar coisas que não são funções, ou com o número de parâmetros errados, tentar atribuir ou dereferenciar valores que não são referências...

- Em uma semântica checada, todos esses erros abortariam a execução, retornando um valor de erro

- Mas e se quisermos poder detectar e recuperar esses erros na própria linguagem?

Erros

- Uma `Acao[T]` não vai produzir mais `T`, mas um valor `Talvez[T]`, que é como `Option[T]` com um valor associado ao caso `None`:

```
trait Talvez[T]
case class Ok[T](v: T) extends Talvez[T]
case class Erro[T](msg: String) extends Talvez[T]
```

- Um valor `Erro[T]` faz *bind* entrar em curto circuito, e não continuar com a sua outra ação

- As primitivas *id* e *le* produzem valores `Ok`, e uma nova primitiva `erro` produz um valor `Erro` com alguma mensagem de erro

- O interpretador ainda precisa ser reescrito para checar todas as possíveis condições de erro e chamar `erro` nos locais certos

try/catch/throw - throw

- Uma vez no interpretador, o mecanismo de erros pode ser exposto à linguagem
- A expressão `throw` produz um com a mensagem passada

```
exp : ...
    | THROW STR
```

```
case class Throw(msg: String) extends Exp
```

- A avaliação de `throw` usa apenas usa a primitiva `erro` que já definimos

try/catch/throw – try/catch

- A expressão `try/catch` executa a expressão no corpo do `try`, e caso o resultado seja um erro executa a expressão no corpo do `catch`

```
exp : ...
    | TRY exp CATCH exp END
```

```
case class TryCatch(etry: Exp, ecatch: Exp) extends Exp
```

- try/catch pode ser implementado em termos de uma primitiva `trycatch` que é em essência o dual de `bind`: entra em curto circuito no caso de etry ser `Ok`, mas executa ecatch se etry der Erro

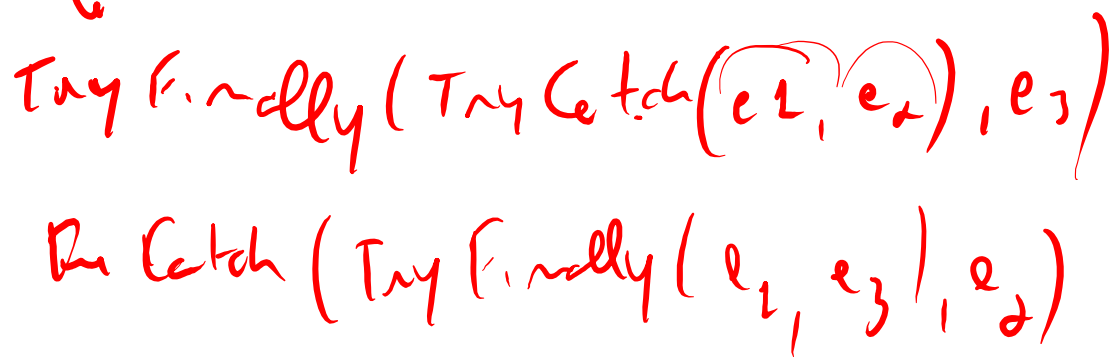
try/catch/throw – finally

- Uma cláusula finally associada a um comando try garante que sua expressão sempre é avaliada independente de um erro acontecer ou não

```
exp : ...
    | TRY exp FINALLY exp END
    | TRY exp CATCH exp FINALLY exp END
```

```
case class TryFinally(etry: Exp, efin: Exp) extends Exp
case class TryCatchFinally(etry: Exp, ecatch: Exp, efin: Exp) extends Exp
```

- Primitivas similares a `bind` e `trycatch` definem como `finally` funciona



Exceções small-step

- Não é difícil adicionar tratamento de exceções ao interpretador small-step
- Precisamos tratar `Throw(msg)` como um possível valor em todas as expressões que têm múltiplas partes, com a diferença de que a presença de Throw faz a expressão toda reduzir para `Throw` no passo corrente
- `TryCatch`, `TryFinally` e `TryCatchFinally` simplesmente tratam `Throw` de outro modo: um `Throw` em `etry` faz `TryCatch` e `TryCatchFinally` reduzirem para a expressão `ecatch`, e `TryFinally` reduzir para a expressão `efin`
- Um `Throw` no bloco `catch` de `TryCatchFinally` faz ele reduzir para a expressão `efin` também

MicroC

- Para poder explorar outras formas de efeitos colaterais e estruturas de controle, vamos mudar o foco para uma linguagem imperativa simples

- MicroC tem sintaxe parecida com a de *fun*, mas abandona funções anônimas e tem apenas um único tipo de valor, números inteiros

- MicroC também não tem referências de primeira classe: toda variável pode ser usada como lado esquerdo da atribuição. Isso também vale para parâmetros de funções, mas a passagem ainda é por valor:

```
fun troca(a, b)
  let tmp = a in
    a := b;
    b := tmp
  end
end  -- não vai trocar os valores de x e y!
let x = 1, y = 2 in troca(x, y) end
```

Ponteiros

- Para compensar a falta de referências de primeira classe, MicroC tem *ponteiros*
- Um ponteiro é um número inteiro tratado como um endereço na memória
- MicroC tem dois operadores para lidar com *ponteiros*:
  - * (Deref) trata o valor de sua expressão como um endereço, e o dereferencia; ele também pode ser usado esquerdo de uma atribuição
  - & (Ender) pode ser usado com variáveis, e dá o endereço da variável

*((&x) + 5)

Passagem por referência com ponteiros

- Usando ponteiros podemos escrever uma função *troca* que funciona:

```
fun troca(a, b)  -- a e b são ponteiros
  let tmp = *a in
    *a = *b;
    *b = tmp
  end
end
-- troca os valores x e y!
let x = 1, y = 2 in troca(&x, &y) end
```

- Quando promovemos esse padrão para a linguagem temos a *passagem por referência* presente em linguagems como C++ e Pascal

- Também podemos usar ponteiros para ter estruturas de dados em MicroC, como vetores

Variáveis em MicroC

- MicroC só tem um tipo de valor, então expressões relacionais funcionam como em C, produzindo 0 no caso de falso e 1 se verdadeiro
- Como toda variável pode ser atribuída, sempre substituímos variáveis por endereços ao invés de valores
- O fato de todas as variáveis serem endereços vai afetar tanto *let* quanto as chamadas de função
- Todo let e toda chamada de função precisa alocar endereços na memória para a variável do let e os parâmetros