Linguagens de Programação

Fabio Mascarenhas - 2013.1

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

try/catch e throw com continuações

- Podemos refazer as exceções usando continuações ao invés de um valor de erro, para isso temos que manter uma “pilha de tratadores de exceção” em `Comp`:

```
case class Handlers(l: List[(End, Cont)]) {
  def pushHandler(sp: End, k: Cont): Handlers = Handlers((sp, k) :: l)
  def popHandler: (End, Cont, Handlers) = l match {
    case (sp, k) :: t => (sp, k, Handlers(t))
  }
}
```

- `erro` abandona a continuação atual para chamar a que está no topo da pilha, enquanto `trycatch` empilha uma continuação que executa o bloco catch e depois usa a sua continuação

- Botamos um sentinela na pilha de tratadores que “aborta” a execução

Threads

- Podemos representar uma thread com uma tripla de um valor pendente que vai ser passado pra thread, seu stack pointer, e uma continuação

- Agora podemos manter nas computações um *relógio*, que conta quantos tiques temos até trocar de thread, e uma fila de threads em espera

- Onde ficaria o escalonador? Uma ideia é fazer `bind` agir como um escalonador, decrementando o relógio e trocando de thread quando necessário

```
def bind(a1: Acao, f: Valor => Acao): Acao = k =>
  a1(v => (tick, sp, ths, mem) =>
            if (tick > 0) f(v)(k)(tick – 1, sp, ths, mem)
            else {
              val (tv, tsp, tk, nths) = changeThread(v, sp, f, k, ths)
              tk(tv)(TICKS, tsp, nths, mem)
            })
```

Threads, take 2

- Uma implementação de threads através de uma mudança em `bind` poderia ser feita sem expor as continuações para o interpretador

- Usando continuações podemos deixar `bind` como estava, e embutir o escalonador nas *primitivas*, pois elas controlam “o que fazer depois” pela escolha de usar *k* ou não

- É como o que fizemos com as exceções

- Se a primitiva faz o relógio chegar a 0 então guardamos a continuação dela e usamos a da primeira thread em espera