4

Memórias Transacionais e Troca de Mensagens: Duas Alternativas para Programação de Máquinas Multi-Core

Professor:

André Rauber Du Bois[1]
(dubois@atlas.ucpel.tche.br)

Resumo:

Para que programas possam tirar proveito das arquiteturas multi-core é necessário que estes possuam várias atividades concorrentes que possam ser alocadas aos *cores* disponíveis. Programas concorrentes para máquinas multicore são geralmente implementados usando *threads* que se comunicam através de uma memória compartilhada. Esse tipo de abstração é muito difícil de programar e sujeita a erros. Neste capítulo discutimos algumas alternativas de programação para máquinas multicore que não são baseadas na abstração de memória compartilhada. Em especial é apresentado modelo de *memórias transacionais* e sua programação na linguagem STM Haskell e a idéia de programação paralela por troca de mensagens na linguagem Erlang. Na programação usando memória transacional, os acessos à memória são feitos através de transações parecidas com as transações de banco de dados. Se não existiu nenhum acesso concorrente a uma área crítica de memória então a transação pode ser efetivada e as mudanças no estado do programa são gravadas na memória. Caso contrário a transação é abortada. Erlang é uma linguagem de programação desenvolvida pela *Ericsson* para programação tolerante a falhas através de troca de mensagens. Erlang vem sendo utilizada com sucesso na programação de várias aplicações comerciais altamente concorrentes. Como estudo de caso, este texto apresenta a implementação do clássico problema do *jantar dos filósofos* em STM Haskell e em Erlang.

1 Graduado em Bacharelado em Ciência da Computação pela Universidade Católica de Pelotas (1998), mestrado em Ciência da Computação pela Universidade Federal do Rio Grande do Sul (2001) e doutorado em Ciência da Computação pela Heriot-Watt University (2005). Atualmente é Professor Auxiliar da Universidade Católica de Pelotas. Tem experiência na área de Ciência da Computação , com ênfase em Teoria da Computação.

4.1. Introdução

Máquinas multi-core agora são uma realidade. Se antes, a cada mais ou menos dois anos a velocidade dos processadores dobrava, agora a tendência no desenvolvimento de processadores parece ser agregar um maior número de *cores* ao processador. Esses *cores* não serão muito mais velozes do que os *cores* disponíveis na versão anterior do processador. Dessa maneira, uma aplicação seqüencial não vai conseguir tirar proveito de uma nova versão de um processador, mesmo que esse tenha o dobro do número de *cores*. Para se explorar o potencial desta arquitetura, as aplicações devem ser concorrentes ou paralelas, de modo que existam tarefas para serem distribuídas entre os *cores*.

Tradicionalmente máquinas multi-core são programadas usando um modelo de memória compartilhada. Nesse modelo temos várias linhas de execução (*threads*) que se comunicam por meio de uma área de memória em comum. Para evitar que threads interfiram de maneira errada no trabalho de outras threads, as linguagens fornecem mecanismos de sincronização como por exemplo *locks*, ou *mutexes*. Esse modelo de programação é difícil e propenso a erros [Rajwar and James, 2003, Lee, 2006, Peyton Jones, 2007, Rigo et al., 2007]. A revolução causada pelas máquinas multi-core é na verdade uma revolução na área de software e não de hardware. Para que se possa desenvolver software confiável para esse tipo de máquina é necessário que se encontrem maneiras mais fáceis de se programar esses processadores.

Neste texto são apresentadas algumas alternativas para a programação de máquinas multi-core que não são baseadas no modelo de memória compartilhada. Primeiramente, na Seção 4.2, é apresentado o modelo de programação concorrente usando memória compartilhada e seus problemas. Apesar de poderem ser aplicados em qualquer paradigma de programação, os modelos de programação concorrente apresentados neste texto são baseados em caracteristicas de *programação funcional*. Por isso, na Seção 4.3, apresentamos os principais conceitos desse paradigma. Em seguida, na Seção 4.4, é apresentada a idéia de *memórias transacionais* e sua programação usando STM Haskell. Na Seção 4.5, a programação por troca de mensagens na linguagem Erlang é discutida. Para que se possa comparar as duas abordagens de programação concorrente, na Seção 4.7, usamos STM Haskell e Erlang para solucionar o clássico problema do *jantar dos filósofos*. Por fim, na Seção 4.8, outros modelos de programação concorrente são discutido e na Seção 4.9 é feito o fechamento do capítulo com algumas conclusões finais.

4.2. Programação Usando o Modelo de Memória Compartilhada

Threads são uma extensão da programação seqüencial que permite linhas de execução simultâneas em um programa. Nos programas tradicionais, existe apenas uma linha de execução definida pela função principal do programa, como a `main` em C e C++. Um programa com várias threads é mais ou menos como um programa que possui vários `main`s executando ao mesmo tempo. Linguagens de programação precisam de poucas modificações para suportar threads, e o hardware e sistemas operacionais disponíveis hoje em dia possuem em geral um bom suporte para a implementação desse tipo de abstração. Programar utilizando este recurso é simples quando existe pouca interação entre as threads. Em muitas aplicações as threads trabalham de forma indepente, não existindo áreas de memória compartilhada entre as mesmas. Porém, quando existe a necessidade de comunicação e sincronização, a programação de aplicações concorrentes não é tão fácil.

Para ilustrar o tipo de problema que pode ocorrer quando existem várias threads compartilhando uma mesma área de memória, vamos analisar o seguinte exemplo:

```
int contador = 0;
```

```
int incrementa(){
  contador++;
  return contador;
}

int decrementa(){
 contador--;
 return contador
}
```

O trecho de código apresenta um contador que é acessado por duas funções (ou métodos). Depois de inicializada a variável `contador` seu valor pode ser modificado pelas funções `incrementa()` e `decrementa()`.

Quando existem várias threads em um programa, essas podem acabar executando as suas instruções de forma intercalada. Essa execução intercalada de instruções pode resultar em resultados inesperados. Exemplo: supondo que existam duas threads *t1* e *t2* usando o contador, e que o estado atual do contador é `6`. Imagine que a thread *t1* chamou a função `incrementa()` e que exatamente quando a thread *t1* acabou de executar a instrução `contador++`, a thread *t2* chamou `decrementa()` executando a operação `contador-`. Dessa maneira, a thread *t1* que deveria receber o valor `7` como resultado da chamada à `incrementa()`, acaba recebendo o valor `6` como resposta. Da mesma maneira, a thread *t2* acaba recebendo o valor `6` como resultado da chamada de `decrementa()`. A intercalação das duas threads acabou gerando um resultado errado. Situações como essa, em que threads lêem e escrevem em uma área de memória compartilhada e a ordem em que as threads são intercaladas influi no resultado final, são chamadas de *condições de corrida*. Encontrar erros em programas que possuem condições de corrida não é uma tarefa simples, pois, mesmo que o resultado da maioria dos testes esteja correto, em algum momento algum resultado inesperado pode aparecer [Tanenbaum and Woodhull, 1997, Lee, 2006]. Os trechos do programa onde os dados compartilhados entre as threads são acessados são chamados de *regiões* ou *seções críticas*.

Para evitar esse tipo de problema, é necessário evitar que mais de um processo acesse uma região crítica por vez, o que chamamos de *exclusão mútua*. De alguma maneira as threads precisam ser *sincronizadas* para evitar um comportamento não esperado do programa. Com a sincronização do acesso das threads aos dados gerados por outras threads é realizada a coordenação das trocas de dados entre as threads evitando resultados imprevistos [Cavalheiro, 2006]. Através do uso correto dos métodos de sincronização podemos garantir que não irá ocorrer um resultado inesperado durante a execução de um programa. O problema é que nem sempre é fácil sincronizar threads de maneira correta, ou de forma a atingir um bom nível de paralelismo.

Um método de sincronização muito utilizado são os bloqueios, também chamados de *locks* ou *mutexes*. *Locks* são usados para implementar duas funcionalidades diferentes: exclusão mútua (evitar que mais de uma thread acesse uma região crítica) e sincronização (fazer com que uma thread espere pelo trabalho de outra). Um *lock* é uma variável booleana que é modificada pelas operações atômicas `LOCK` e `UNLOCK`. A operação `LOCK` serve para alterar a variável booleana para verdadeira. Se no momento da chamada a `LOCK` a variável booleana está verdadeira, a thread que chamou `LOCK` fica bloqueada até que a variável seja modificada para `false` por outra thread.

Podemos usar *locks* para proteger as chamadas às funções `incrementa()` e `decrementa()`. Assim, toda a vez que uma thread desejar chamar um desses métodos, ela deve primeiramente adquirir o *lock*. Se todas as chamadas às funções do contador

forem protegidas pelo mesmo *lock* conseguimos evitar que threads chamem esses métodos ao mesmo tempo:

```
resposta r;
(...)
LOCK(&m)
r = incrementa()
UNLOCK(&m)
```

Para evitar o uso de bloqueios toda a vez que é feita a chamada da função, o programador poderia modificar a implementação do método `incrementa()` e `decrementa()` usando internamente o mesmo *lock* para proteger o acesso à variável contador. Por exemplo, a função `incrementa()` poderia, em uma primeira tentativa, ser implementada da seguinte maneira:

```
(...)
int incrementa()
{
 LOCK(&m)
 contador++;
 UNLOCK(&m)
 return contador;
}
(...)
```

Esse código ainda possui um problema: quando uma thread libera o *lock* outra thread pode adquiri-lo e modificar o contador antes da função `incrementa()` retornar o resultado. Para solucionar o problema, podemos usar uma variável local na função para retornar o resultado:

```
int incrementa()
{
 int tmp;

 LOCK(&m)
 contador++;
 tmp = contador;
 UNLOCK(&m)
 return tmp;
}
```

Para facilitar a implementação desse tipo de função, linguagens como Java permitem a implementação de métodos atômicos. Em Java, podemos implementar métodos atômicos usando a a palavra reservada `synchronized`:

```
public synchronized int incrementa()
{
 contador ++;
 return contador;
}
```

Uma classe pode possuir vários métodos `synchronized`. Os métodos `synchronized` de um objeto só podem ser acessados por uma thread de cada vez. Uma vez que uma thread começa a executar um método `synchronized`, este método é executado em regime de exclusão mútua em relação aos demais métodos `synchronized`

do mesmo objeto. Internamente essa sincronização é atingida por meio de bloqueios. Cada objeto Java possui apenas um *lock* e para chamar um métodos `synchronized` as thread precisam adquirir esse *lock*. Dessa maneira as outras threads devem esperar que essa thread termine para que possam obter o *lock* e executar um método `synchronized`. Métodos `synchronized` permitem que se implemente o mesmo tipo de sincronização fornecida por *monitores* [Tanenbaum and Woodhull, 1997].

Existem vários problemas com o uso de bloqueios para a sincronização de threads [Rajwar and James, 2003, Lee, 2006, Peyton Jones, 2007, Rigo et al., 2007]:

- **Dificuldade de Programação:** Programar usando bloqueios é complicado. É comum acontecerem erros como adquirir *locks* demais, o que acaba com a concorrência ou pode gerar um bloqueio eterno do programa (*deadlock*), adquirir menos *locks* do que o necessário, o que pode permitir que mais de uma thread entre na região crítica, adquirir os *locks* errados, erro de programação comum pois pode ser difícil de perceber qual os dados que estão sendo protegidos pelo *lock*, adquirir os *locks* na ordem errada, os *locks* devem ser adquiridos na ordem correta para evitar *deadlock* etc. Programar usando *locks* é tão baixo nível quanto programar em *assembly*, porém é ainda mais difícil de depurar pois todos os erros possíveis como condições de corrida e *deadlocks* são imprevisíveis e de difícil reprodução [Tanenbaum and Woodhull, 1997]. Por isso é muito difícil de se criar sistemas que sejam confiáveis e escaláveis.
- **Dificuldade de Reuso de Código:** Trechos de código corretamente implementados usando *locks*, quando combinados, podem gerar erros. No exemplo do contador, tínhamos as funções `incrementa()` e `decrementa()` que estavam corretamente implementadas usando *locks*. Porém, se o programador resolve incrementar duas vezes o contador usando o método `incrementa()`, não podemos garantir que o resultado estará correto. Como a função `incrementa()` libera o *lock* depois de ser chamada a primeira vez, alguma thread pode se aproveitar da oportunidade e realizar alguma operação que modifique a variável `contador` antes do método `incrementa()` ser chamado pela segunda vez.
- **Gargalo Serial:** Uma região crítica protegida por bloqueios permite o acesso de apenas uma thread de cada vez. Quando uma thread tenta acessar uma região de código protegida que já está sendo acessada, essa thread fica bloqueada até a outra liberar o bloqueio causando gargalos seriais nos programas. Um erro comum de programação é criar grandes regiões de código protegidas por *locks*, o que reduz com o paralelismo do programa. Porém, quando se trabalha com regiões menores, o código de sincronização fica mais difícil de se escrever e mais sujeito a erros.

O objetivo deste texto é apresentar ao leitor duas alternativas para programação concorrente, uma baseada na idéia de *memória transacional* (Seção 4.4) e outra baseada em *troca de mensagens* (Seção 4.5). Ambas oferecem abstrações de mais alto nível para a programação.

4.3. Linguagens Funcionais e Programação Paralela

O paradigma de programação funcional enfatiza a avaliação de expressões matemáticas. Utilizando este paradigma, o programador deve se preocupar em definir funções e construir expressões que combinem essas funções. Linguagens funcionais são linguagens de programação que facilitam, ou até obrigam, o programador a construir software usando apenas funções e expressões. O uso de linguagens funcionais traz várias vantagens como prototipação rápida e reuso de software, além de gerar programas compactos e com alto nível de expressão.

Nos últimos anos, as linguagens de programação modernas vêm adquirindo cada vez mais caracteristicas oriundas do paradigma funcional. Por exemplo, coleta automática de lixo, que se popularizou com a linguagem Java, é um recurso que surgiu com o Lisp [Winston and Horn, 1989], a primeira linguagem funcional. Funções anônimas ou *clousures* e funções de alta ordem, abstrações derivadas do cálculo lambda [Barendregt, 1984], e que são de grande importância na maioria das linguagens funcionais, hoje estão presentes em linguagens como Groovy, Python, PHP e C#. Polimorfismo universal, abstração que se popularizou em linguagens funcionais modernas como ML [Harper et al., 1986], Haskell [Jones, 2003] e O'Caml [OCaml, 2002], aparece em Java e C# com o nome de *tipos genéricos*. *Compreensão de listas*, abstração que permite a definição de funções sobre listas usando uma notação parecida com a notação matemática de conjuntos, está presente em linguagens de script modernas como LINQ e Java FX.

Com o surgimento de processadores multi-core para máquinas domésticas aparece a necessidade de linguagens de programação com abstrações de alto nível que facilitem a programação desses dispositivos. Como programas funcionais possuem apenas expressões matemáticas puras, sem atribuições ou acesso explícito à memória, as expressões de um programa podem ser avaliadas em qualquer ordem, facilitando a exploração do paralelismo.

Existem várias vantagens de se escrever programas paralelos/concorrentes usando linguagens funcionais [Hammond and Michaelson, 1999]:

- **Facilidade de particionar o programa paralelo**: Linguagens funcionais puras possuem a caracteristica de que todas as sub-expressões de um programa podem ser avaliadas em qualquer ordem, sempre retornando o mesmo resultado para o programa, já que não existe dependência de controle explícitas entre as sub-expressões, como por exemplo atribuições. O resultado do programa é independente do escalonamento das tarefas.
- **Ausência de deadlock**: Ao contrário de programas imperativos, é impossível introduzir um *deadlock* em um programa funcional ao paralelizar um programa seqüencial [Hammond and Michaelson, 1999]. Todo o programa que devolve um valor quando roda de forma seqüencial obrigatoriamente retorna o mesmo valor quando rodar em paralelo.
- **Possibilita a implementação de abstrações de alto-nível**: Linguagens funcionais possuem um alto poder de abstração. Recursos como funções de alta-ordem (funções que recebem funções como argumento), permitem que o programador defina o *esqueleto* de um algoritmo que pode ser configurado com o uso dos argumentos da função. Surge então a idéia de *Algorithmic Skeletons* [Cole, 1989] que são funções de alta-ordem que encapsulam padrões recorrentes de paralelismo. O programador apenas escolhe um esqueleto que implementa o comportamento paralelo desejado. Aspectos de *coordenação* (criação de threads, sincronização etc) da aplicação são totalmente controlados pelo esqueleto.

Nas seções 4.4 e 4.5 deste texto, duas linguagens de programação funcional para máquinas multi-core são apresentadas. A primeira, STM Haskell, é uma extensão da linguagem funcional Haskell para programação concorrente usando memória transacional. A segunda, Erlang, é uma linguagem desenvolvida pela Ericsson para programação concorrente usando troca de mensagens.

4.4. STM Haskell

4.4.1. Memórias Transacionais

Memória transacional é uma nova abstração para programação concorrente baseada na idéia de transações. Uma *transação de memória* é uma seqüência de operações que modificam a memória e que podem ser executadas completamente ou podem ter nenhum efeito (*podem ser abortadas*) [Herlihy and Moss, 1993]. Essas operações são *atômicas*, ou seja, são executadas por completo ou nunca são executadas. Uma transação de memória é parecida com as transações de bancos de dados: se uma transação é efetivada (*commit*) todas as alterações são executadas modificando o estado do programa. Se a transação aborta, nenhuma alteração no estado ocorre. Dessa maneira, transações em memória rodam como se estivessem isoladas, no sentido de que as threads executam modificações na memória como se estivessem modificando uma área de memória isolada do acesso de outras threads. As outras threads do sistema só conseguem ver o resultado de uma transação após o *commit*. Na verdade os acessos à memória são colocados em um *log da transação*. Quando o bloco da transação termina de executar, seu *log* é validado para verificar se sua visão da memória é consistente, e só então as modificações são realmente gravadas na memória, o que seria o *commit* da transação [Harris et al., 2005b][1].

Linguagens de programação que fornecem a abstração de memórias transacionais geralmente possuem um comando `atomic` ou `atomically` que serve para marcar um bloco de código que deve ser executado atomicamente com relação a todos os outros blocos atômicos do programa:

```
atomically{
  comando1;
  comando2;
  (...)
}
```

Os blocos *atomically* são executados usando *sincronização otimista* pois no momento em que rodam não adquirem *locks*, todos os blocos executam modificando seu *log* de acesso a memória. Após a execução, se o *log* é validado, as modificações são escritas em memória, caso contrário o código é executado novamente.

Voltando ao exemplo do contador, apresentado na Seção 4.2, podemos resolver o problema de incrementar o contador duas vezes, sem o uso *locks* internos ou externos, apenas incluindo as chamadas às funções dentro de um bloco `atomically`:

```
atomically {
  incrementa();
  incrementa();
}
```

O uso de memórias transacionais traz várias vantagens como ausência de *deadlock*, possibilidade de abortar e retornar ao estado original do programa em caso de exceções ou *timeouts* e ausência da tensão entre a granulosidade dos bloqueios e da concorrência, acabando com o gargalo serial das aplicações.

Nesta seção, uma extensão da linguagem funcional Haskell para a programação concorrente usando memórias transacionais é descrita. Primeiramente alguns conceitos

[1]Existem várias maneiras diferentes de se implementar memórias transacionais, tanto em hardware quanto em software. Para uma visão geral sobre o assunto, o leitor pode consultar [Adl-Tabatabai et al., 2006, Rigo et al., 2007].

de programação em Haskell são apresentados como entrada e saída, threads e variáveis mutáveis (Seção 4.4.2). Em seguida, a Seção 4.4.3 apresenta a STM Haskell, uma extensão da linguagem Haskell para programação concorrente usando memórias transacionais (Seção 4.4.3). O leitor não precisa nenhuma experiência prévia com programação funcional ou Haskell, pois todos os conceitos necessários para o entendimento dos exemplos são apresentados no texto.

4.4.2. Efeitos colaterais em uma linguagem funcional pura

Tradicionalmente linguagens funcionais puras não possuem efeitos colaterais (mudança de um valor na memória, entrada e saída etc). O resultado da aplicação de uma função depende exclusivamente dos valores que são passados como argumento. Isso é interessante pois preserva propriedades da linguagem e permite fazer provas matemáticas em cima dos programas. O problema é que programas na vida real precisam de efeitos colaterais. O objetivo de um programa é produzir algum efeito, nem que seja imprimir um resultado na tela do computador. Isso por muito tempo foi um problema para a comunidade que pesquisava essa área até que se descobriu uma maneira de se ter efeitos colaterais em uma linguagem funcional e mesmo assim manter as propriedades legais existentes nas linguagens funcionais puras. Em [Wadler, 1990], Phil Wadler propôs o uso de *monadas* para encapsular os efeitos colaterais dos programas funcionais. Monadas são estruturas da teoria das categorias [Blauth and Haeusler, 2002], um ramo bastante abstrato da matemática. Apesar da teoria por trás do uso de monadas ser complicada, o seu uso em programas funcionais é bastante simples. A idéia consiste em separar as ações puras das ações impuras pelos tipos das funções. Uma ação que pode gerar um efeito colateral possui tipo `IO a`. Uma ação do tipo `IO a` é uma ação que, quando executada, pode gerar um efeito colateral e produzir um valor qualquer de um tipo `a` qualquer. Por exemplo, uma ação de tipo `IO Int` é uma ação que, quando executada, retorna um valor de tipo inteiro. Todo o programa em Haskell define apenas uma única ação chamada de `main`. O tipo da ação `main` é `IO ()`. Ou seja, o `main` é uma ação que quando executada retorna um valor do tipo `()`. O tipo `()` funciona como o `void` da linguagem C ou Java, ou seja, não retorna valor:

```
main :: IO ()
main = print "Alô Mundo!"
```

A ação `main` é definida pela expressão `print "Alô Mundo!"`, que usa a função `print` para gerar o efeito de imprimir `"Alô Mundo!"` na tela. Seguem dois exemplos de funções que geram efeitos colaterais:

```
print :: String -> IO ()
getLine :: IO String
```

A primeira função, `print`, recebe como argumento uma `String` e gera como resultado uma computação que, quando executada, imprime a string na tela. Note que a computação gerada possui tipo `IO ()`, significando que essa ação não devole nenhum resultado. A função `getLine` é uma ação que, quando executada, lê uma `String` da entrada padrão do sistema. As ações do tipo `IO` podem ser combinadas para gerar ações maiores usando a notação `do`:

```
echo :: IO ()
echo = do
      a <- getLine
      print a
```

A função `echo` é uma ação de entrada e saída que quando executada lê uma `String` da entrada padrão (usando `getLine`) e depois imprime essa `String` na tela usando a função `putStr`. Note que cada linha de um bloco `do` deve obrigariamente possuir uma ação `IO`. Segue um exemplo de ação `IO` que retorna um resultado:

```
leInverte :: IO String
leInverte = do
           a <- getLine
           return (reverse a)
```

A função `leInverte` é uma ação que, quando executada, lê uma `String` da entrada padrão e retorna a mesma `String` invertida. Para inverter essa `String`, foi usada a função pré-definida `reverse`:

```
reverse:: String -> String
```

Note que a função `reverse` é uma função pura, ou seja, nenhum dos seus argumentos possui tipo `IO`. Para uma ação retornar um valor, usamos a função pré-definida `return`:

```
return :: a -> IO a
```

Essa função recebe um valor de qualquer tipo, o valor `a` é uma *variável de tipo* que pode representar qualquer tipo da linguagem Haskell, e devolve uma ação que quando executada retorna esse mesmo valor.

Note que ações `IO` são valores de *primeira ordem* em Haskell. Dessa maneira, ações podem ser passadas como argumentos para funções e retornadas como resultado da computação de uma função. Usando ações e recursão podemos implementar estruturas de controle, como por exemplo repetições. Exemplo:

```
repete :: Int -> IO () -> IO ()
repete 0 acao = return ()
repete n acao = do
                 acao
                 repete (n-1) acao
```

A função `repete` recebe um inteiro `n` e uma `acao` e executa essa `acao` `n` vezes. Quando o inteiro recebido for igual a `0` (*zero*) o programa não faz nada, ou seja, retorna `()`. Caso contrário, o programa executa a `acao` uma vez e então chama `repete` novamente para executar a `acao` n-1 vezes. Exemplo:

```
main :: IO ()
main = repete 10 (print "Alo Mundo")
```

Todos os programas apresentados até agora são programas seqüências. Podemos criar *threads* em Haskell usando a primitiva `forkIO`:

```
forkIO :: IO () -> IO ()
```

A primitiva `forkIO` recebe como argumento uma ação de `IO` e cria uma nova thread para executar essa ação. Essa nova thread roda concorrentemente com outras threads. Por exemplo:

```
main :: IO ()
main = do
      forkIO (print "Alô da nova thread")
      print "Alô do main"
```

Nesse programa foi criada uma thread para imprimir a *string* `"Alo da nova thread"` enquanto a thread principal imprime `"Alô do main"`.

Variáveis mutáveis Em Haskell, como as funções são funções matemáticas puras, todos os argumentos são passados por valor [2], significando que não existem referências a valores, todos os valores passados como argumentos para uma função são copiados. A única maneira de se trabalhar com áreas de memória mutáveis é dentro das ações `IO`.

Por exemplo, a seguinte função incrementa um valor inteiro que está em uma posição de memória:

```
incrementaRef :: IORef Int -> IO ()
incrementaRef ref = do
                    val <- readIORef ref
                    writeIORef ref (val +1)
```

Um valor do tipo `IORef Int` é uma referência a uma área de memória que pode guardar valores do tipo `Int`. Um valor do tipo `IORef Int` é como se fosse um ponteiro para uma região de memória que contém um inteiro, mais ou menos como declarar uma variável do tipo `int*` em C. A ação primeiramente lê o valor que está na posição de memória `ref` e depois escreve esse valor incrementado na mesma posição de memória. Para ler e escrever em posições de memória usamos as funções `readIORef` e `writeIORef`:

```
readIORef :: IORef a -> IO a
writeIORef :: IORef a -> a -> IO ()
```

Note que as funções que trabalham com regiões de memórias mutáveis são polimórficas. Um valor do tipo `IORef a` representa uma região de memória que contém um valor de um tipo qualquer `a`. Esse polimorfismo de tipos é parecido com o polimorfismo de tipos presente em tipos genéricos em Java.

4.4.3. A linguagem STM Haskell

Uma linguagem de programação funcional, tipo Haskell, é perfeita para se implementar memórias transacionais por dois motivos [Harris et al., 2005b]:

- O sistema de tipos consegue separar as ações que possuem efeitos colaterais das que não possuem. Nem todo o tipo de ação pode ser realizada dentro de uma transação. O sistema de tipos de linguagens como o Haskell consegue garantir que somente as ações corretas serão executadas dentro de uma transação.
- Em uma linguagem funcional, a maior parte das computações são *puras* no sentido de que não possuem efeitos colaterais. Esse tipo de ação não modifica a memória e não precisa ser logada pelo sistema transacional. As ações *puras* da linguagem nunca precisam ser *desfeitas*, elas podem simplesmente ser repetidas no caso de uma transação abortar.

Quanto à primeira vantagem, observe o seguinte exemplo:

```
atomically{
   if (n>k) then
       lancar_bomba_atomica()
}
```

Uma transação, caso falhe, pode ser re-executada. A transação apresentada no exemplo, quando executada novamente, pode acabar lançando uma nova bomba atômica.

[2] Na verdade a passagem de parâmentros usa *lazy evaluation*.

É necessário garantir que dentro de uma transação apenas operações que executam modificações na memória serão executadas. STM Haskell, com seu sistema de tipos, garante que operações de entrada e saída não serão executadas dentro de uma transação.

A primitiva `atomically` do STM Haskell possui o seguinte tipo:

```
atomically :: STM a -> IO a
```

A primitiva `atomically` recebe como argumento uma *ação transacional* de tipo `STM a` e gera uma ação de entrada e saída (`IO`) que quando executada também executa a transação de forma atômica. Uma ação do tipo `STM a` é parecida com uma ação de `IO` pois contém efeitos colaterais, porém esses efeitos colaterais são mais restritos. Basicamente, dentro de uma ação `STM a` pode-se apenas ler e escrever em váriaveis transacionais (`TVar a`).

Um valor do tipo `TVar a` é como se fosse uma posição de memória mutável. Essa posição de memória pode ser modificada usando duas operações:

```
readTVar :: TVar a -> STM a
writeTVar :: TVar a -> a -> STM ()
```

As primitivas `readTVar` e `writeTVar` funcionam de forma parecida com as funções para modificar valores do tipo `IORef`. A primitiva `readTVar` recebe como argumento uma variável transacional `TVar` e devolve como resposta uma *ação transacional* que quando executada lê o valor contido na variável. De forma similar, a função `writeTVar` serve para escrever um valor em uma `TVar`.

Uma ação executada com a primitiva `atomically` deve garantir duas propriedades [Peyton Jones, 2007]:

- **Atomicidade**: O efeito de executar `atomically comp` torna-se visivel para outras threads como um todo. Nenhuma thread consegue ver os estados intermediários da execução de `comp`.
- **Isolamento**: Durante a execução de `atomically comp`, a ação `comp` não pode ser afetada por outras threads. É como se `comp` usasse uma cópia do estado do programa para executar.

Para que se possa entender o funcionamento das ações transacionais e de suas primitivas, esta seção apresenta a implementação de um sistema simples para a venda de ingressos para um show. Para se comprar os ingressos deve-se realizar duas ações. Primeiro deve-se retirar o número de ingressos sendo comprados dos ingressos disponíveis e em seguida deve-se depositar o valor sendo pago pelos ingressos na conta da loja. Essas ações devem ser realizadas de forma atômica e indivisível. Não se deseja que em algum momento o estado do sistema apresente uma visão inconsistente, i.e., mais dinheiro na conta do que ingressos vendidos ou menos ingressos do que dinheiro.

Pode-se modelar o problema com uma função chamada `compraIngressos`. Essa função recebe como argumentos uma referência aos ingressos disponíveis, uma conta, um inteiro representando o número de ingressos sendo comprados, e um `Float` representando o valor pago pelos ingressos. Como resultado `compraIngressos` retorna uma ação que, quando executada, subtrai os ingressos comprados do número de ingressos disponíveis e deposita o valor da compra na conta bancária.

```
compraIngressos::Ingressos-> Conta -> Int ->  Float -> IO()
compraIngressos ing cont q v =
                    atomically ( do
                                   retiraIngressos ing q
                                   deposito cont v
                                )
```

Tanto `retiraIngressos` quanto `deposito` são ações transacionais:

```
retiraIngresso :: Ingressos -> Int -> STM ()
deposito :: Conta -> Float -> STM ()
```

Note que podemos compor ações transacionais da mesma maneira que combinavamos ações `IO` usando a notação `do`. O leitor deve observar que dentro de um mesmo bloco `do` *não* podemos ter ações `STM` e `IO` ao mesmo tempo.

A função `retiraIngressos` recebe uma `TVar`, que contém o número de ingressos disponíveis no momento, e o número de ingressos a serem vendidos para o cliente como argumentos e retorna uma ação transacional que retira os ingressos vendidos da variável transacional:

```
type Ingressos = TVar Int

retiraIngresso :: Ingressos -> Int -> STM ()
retiraIngressos ingressos v = do
              n <- readTVar ingressos
              writeTVar ingressos (n-v)
```

Dentro do bloco `do` o número de ingressos disponíveis é lido usando `readTVar` e um novo valor é gravado na variável transacional usando `writeTVar`. A função `deposito` é parecida, ela primeiramente lê o saldo atual da `conta` e então deposita o `valor` pago pelo ingresso:

```
type Conta = TVar Float

deposito :: Conta -> Float -> STM ()
deposito conta valor = do
                  saldo <- readTVar conta
                  writeTVar conta (saldo + valor)
```

Haskell fornece duas outras primitivas para trabalhar com ações transacionais: `retry` e `orElse`. Essas primitivas lidam com a idéia de como *bloquear* threads e como *escolher* entre os resultados computados por transações. A primitiva `retry` possui o seguinte tipo:

```
retry :: STM a
```

Quando o `retry` é executado, a transação corrente é abandonada, sendo executada novamente no futuro. Podemos usar essa primitiva para tratar o caso em que o número de ingressos disponíveis é menor do que o número de ingressos que se deseja comprar:

```
compraIngressos :: Ingressos -> Int -> STM ()
compraIngressos ingressos comprar = do
              disponivel <- readTVar ingressos
              if (disponivel < comprar)
              then retry
              else writeTVar ingressos (n-v)
```

Se o número de ingressos que se deseja comprar é maior do que o número de ingressos disponíveis então a transação é suspensa e reiniciada mais tarde. A transação poderia ser executada imediatamente, mas isso seria ineficiente já que o número de ingressos disponíveis provavelmente demora para mudar, por exemplo, quando alguma devolução ocorrer. Uma implementação eficiente do `retry` só deve tentar novamente a transação quando o valor da variável `ingressos` for modificada por uma outra thread.

O padrão *testar uma condição e se esta não for satisfeita então chamar* `retry` é tão comum que fica mais fácil usar uma função auxiliar como `check`:

```
check :: Bool -> STM ()
check True = return ()
check False = retry
```

Dessa maneira, podemos escrever a função `comprarIngressos` de maneira mais elegante:

```
compraIngressos :: Ingressos -> Int -> STM ()
compraIngressos ingressos v = do
              disponivel <- readTVar ingressos
              check (comprar <= disponivel)
              writeTVar ingressos (n-v)
```

O último combinador da biblioteca de ações transacionais é o `orElse`, que serve para descrever transações alternativas. A idéia é permitir que uma ação alternativa seja tomada caso uma ação chame o `retry`. O combinador `orElse` possui o seguinte tipo:

```
orElse :: STM a -> STM a -> STM a
```

`orElse` recebe como argumentos duas ações transacionais e devolve como resposta uma outra ação transacional que faz uma *escolha* entre as ações que recebeu como argumento. Uma chamada para `orElse a1 a2` tenta primeiro executar `a1`, se `a1` chamar `retry` então `a2` será executada. Se `a2` chamar `retry` então toda a transação é executada novamente.

Como exemplo de uso do `orElse` podemos pensar no caso em que, se os ingressos de um show acabaram, então o usuário pode comprar os ingressos para um outro show:

```
compraIngressos2 :: Ingressos -> Ingressos -> Int -> STM ()
compraIngressos2 i1 i2 n =
             orElse (compraIngressos i1 n)
                    (compraIngressos i2 n)
```

Nesta seção foram apresentadas as principais primitivas para programação usando memórias transacionais na linguagem STM Haskell. Na Seção 4.7 é descrita a implementação do problema do *jantar dos filósofos* usando STM Haskell.

4.5. Erlang

4.5.1. Troca de mensagens

Troca de mensagens é um mecanismo para comunicação quando não existe memória compartilhada entre processos. Bibliotecas para comunicação usando troca de mensagem geralmente fornecem duas primitivas, `send` (envia) e `receive` (recebe):

```
send (destino, mensagem)

mensagem = receive(remetente)
```

A primeira primitiva envia uma `mensagem` para um processo `destino` e a segunda recebe uma `mensagem` enviada por um processo `remetente`. Existem várias semânticas possíveis para essas operações. Por exemplo, o `send` e o `receive` podem ser *bloqueantes* ou *não-bloqueantes*. Um `send` bloqueante faz com que o processo fique

bloqueado até que a mensagem chegue no destino. No `send` não-bloqueante o processo continua executando enquanto a mensagem está sendo enviada. Se o `receive` fica bloqueado quando tenta receber uma mensagem que ainda não chegou ele é dito bloqueante. Se o processo prossegue executando mesmo sem ter recebido a mensagem ele é dito não bloqueante. O `send` e o `receive` também podem ter *nomeação explícita* e *implícita*. Na nomeação explícita, tanto o `send` quanto o `receive` devem indicar para qual processo vão enviar/receber a mensagem. Na nomeação implícita as primitivas não precisam indicar para qual processo a mensagem será enviada/recebida. No caso do `send`, a mensagem pode ser enviada para um grupo de processos interessados. No caso do `receive`, a mensagem pode vir de qualquer processo.

Troca de mensagens geralmente é utilizada para a comunicação entre processos localizados em *máquinas diferentes* em uma rede. Esse paradigma de comunicação é explorado por linguagens como PVM [Geist et al., 1994] ou MPI [Gropp et al., 1999].

4.6. A linguagem de Programação Erlang

Erlang é uma linguagem para programação concorrente, distribuída e tolerante a falhas desenvolvida pela Ericsson para a programação de *switches* de telefonia. A história da linguagem começa em 1981, com a criação de um laboratório de pesquisa em Ciência da Computação dentro da Ericsson para o desenvolvimento de novas tecnologias. A partir de 1986 começou o trabalho na área de linguagens de programação com o objetivo de projetar uma linguagem que facilitasse a programação concorrente. Surge então a linguagem Erlang que inicialmente era apenas uma extensão de Prolog e que com o passar do tempo ganhou sua própria sintaxe, máquina virtual e características de programação funcional.

A linguagem Erlang fornece um modelo de programação chamado de *Programação Orientada a Concorrência* [Armstrong, 2003]. Esse modelo surge da visão de que no mundo real estamos cercados por atividades concorrentes. Quando programamos tentamos modelar uma parte do mundo. Se o mundo é formado por atividades concorrentes, concorrência deveria ter uma importância maior no modelo de programação. Concorrência significa que *coisas* acontecem ao mesmo tempo. No mundo real, existem sempre atividades concorrentes. Quando caminhamos em uma rua existem milhares de atividades acontecendo em nossa volta. Linguagens de programação orientadas a objeto teriam, a princípio, a idéia de que objetos são entidades concorrentes que conversam por troca de mensagens. Porém, quando olhamos para linguagens de programação como Java, objetos são estáticos. Quando desejamos entidades concorrentes e ativas como por exemplo um *clock*, temos que criar threads que são abstrações de mais baixo nível.

As principais características de uma *linguagem orientada a concorrência* são:

- Suporte a *processos* concorrentes. Cada processo funciona como se rodasse em sua própria máquina virtual.
- Processos devem ser isolados. Não existe memória compartilhada entre processos.
- Processos se comunicam apenas por troca de mensagens. O protocolo que o processo entende funciona como a sua interface.
- Mensagens devem ser assíncronas, sem garantia de entrega
- Processos devem poder de alguma maneira detectar a falha de outros processos.

Quando adicionamos threads em um programa Java ou C é porque realmente necessitamos de concorrência na aplicação. Em Erlang, programamos definindo processos que trocam mensagens. Essa é a maneira natural de escrever um programa em Erlang. Modularidade é atingida pela definição de vários processos cuja interface são as mensagens que eles conseguem receber e processar. Da mesma maneira que programadores Java dividem seu programa em uma série de objetos que se comunicam através

de chamadas de métodos, o programador Erlang divide seu programa em vários processos concorrentes que conversam por troca de mensagens. Processos em Erlang são de baixo custo. Segundo Joe Armstrong [Heller, 2003], um dos criadores desta linguagem, em um teste de escalabilidade viu-se que leva menos de um milisegundo para se criar um processo em Erlang e quando existem alguns milhares de processos o tempo de criação fica por volta de 5ms. Em comparação, no mesmo hardware, a criação de processos em C# e Java leva 250ms até 2000 e 1000 processos respectivamente, e um maior número de processos não pode ser criado. Processos em Erlang são completamente independentes. É como se cada processo rodasse em sua própria máquina. Como não existe memória compartilhada entre processos, ou memória mutável dentro de um processo, não existe a necessidade de se sincronizar o acesso a áreas de memória. Programas podem possuir milhares de processos sem que isso afete a performance. Dessa maneira, um programa Erlang sempre consegue uma melhora de desempenho quando roda em uma máquina com vários processadores pois sempre existem atividades concorrentes para serem alocadas aos processadores existentes. O fato dos processos se comunicarem apenas por troca de mensagem evita a maioria dos problemas que são inerentes da programação concorrente com memória compartilhada. Para evitar gargalos, como por exemplo entrada e saída, mensagens são assíncronas o que adiciona um grau maior de concorrência as aplicações. Além disso existem *behaviours* que são esqueletos de aplicações que podem ser reutilizados, mais ou menos como um padrão de projeto, para garantir que a aplicação seja escalável e tolerante a falhas. Erlang foi projetada para se construir sistemas que rodam por anos sem nenhuma parada. O switch AXD 301 da Ericsson, que possui 1.7 Milhões de linhas de código Erlang, possui 99.999999999% de confiabilidade (downtime de 31ms por ano). Uma característica importante da linguagem, que permite tanta confiabilidade, é que partes de um programa podem ser modificadas enquanto o programa está rodando, sem a necessidade de parar a aplicação.

Nesta seção a linguagem Erlang é apresentada. Primeiramente, na Seção 4.6.1, programação seqüencial em Erlang é descrita. Na Seção 4.6.2 as principais construções para programação concorrente são mostradas. Na Seção 4.7, a implementação do problema do Jantar dos Filósofos em Erlang é apresentada.

4.6.1. Programação seqüencial

Programas em Erlang são escritos dentro de módulos. Exemplo:

```
-module(mex1).
-export([soma/2,incrementa/1,maiorIdade/1]).

soma(X,Y) -> X + Y.

incrementa(X) -> soma(1,X).

maiorIdade(X) -> X>= 18.
```

A primeira linha indica o nome do módulo sendo definido (`mex1`). A segunda indica as funções que são exportadas pelo módulo sendo definido. A lista de funções sendo exportadas contém o nome das funções e o número de argumentos que cada uma recebe (ex: a função `soma` recebe dois argumentos).

A terceira parte de um módulo contém definições de funções. A definição:

```
soma(X,Y) -> X + Y.
```

nos diz que `soma` é uma função que recebe dois argumentos `X` e `Y` e devolve como resposta a expressão `X + Y`. A definição de uma função, além de operadores aritméticos, também pode usar outras funções. Por exemplo, a função `incrementa` recebe um valor numérico e soma o valor `1` a esse valor. Sua definição no módulo `mex1` usa a função `soma`:

```
incrementa(X) -> soma(1,X).
```

ou seja, a função `incrementa` recebe como argumento um valor qualquer `X` e devolve como resposta a `soma` de `X` ao valor `1`. Funções podem receber qualquer tipo de valor como argumento e devolver qualquer tipo de valor como resposta. A última função do módulo `mex1` recebe um valor numérico como argumento e devolve como resposta um valor booleano que diz se o argumento é igual ou maior que `18`:

```
maiorIdade(X) -> X>= 18.
```

Os programas escritos em Erlang rodam em um interpretador. Quando rodamos o interpretador, ele fornece um *prompt* onde podem ser digitadas expressões que usam as funções definidas nos módulos. Nos exemplos apresentados neste texto, o prompt do interpretador será representado pelo símbolo (`>`). Para usar as funções do módulo `mex1` devemos primeiramente compilar este módulo dentro do interpretador. Exemplo:

```
> c(mex1).
{ok,mex1}
```

O comando `c(mex1)` compila o módulo `mex1`. Quando o processo de compilação termina, o interpretador devolve a mensagem `{ok,mex1}` significando que o módulo foi compilado com sucesso. Depois da compilação do módulo, podemos usar as funções no interpretador:

```
> mex1:soma(1,2).
3
```

Para se chamar uma função no interpretador devemos fornecer também o nome do módulo em que ela está definida. Dessa maneira, a expressão `mex1:soma(1,2)` representa a chamada da função `soma` definida no módulo `mex1` passando para ela dois valores como argumentos: `1` e `2`. Como resposta a essa chamada o interpretador retorna o valor `3`.

Outros exemplos de uso das funções definidas no módulo `mex1`:

```
> mex1:incrementa(10).
11

> mex1:maiorIdade(19).
true
```

Note que os valores booleanos são representados em Erlang pelos átomo `true` e `false`.

Recursão Repetições são implementadas usando recursão. Por exemplo, o fatorial de um número `N` é igual a

```
N * N-1 * N-2 * ... * 2 * 1
```

ou seja, o fatorial de 4 é igual a:

```
4 * 3 * 2 * 1
```

O fatorial de um número é geralmente implementado em uma linguagem de programação tradicional usando uma estrutura de repetição. Por exemplo, em Java, calculamos o fatorial de um número `n` qualquer usando o seguinte `for`:

```
int fat = 1;
for(int i = 1; i<=n; i++)
   fat = fat * i;
```

Em Erlang, esse tipo de loop é definido usando recursão:

```
fat(0) -> 1;
fat(N) -> N * fat(N-1).
```

Essa definição recursiva é formada por duas cláusulas separadas por `;` (ponto e vírgula). A primeira cláusula diz que o fatorial do número `1` é o próprio `1`. A segunda cláusula diz que o fatorial de um número `N` qualquer, `diferente de 1`, é `N * fat(N-1)`. Quando fazemos uma chamada da função fatorial no interpretador Erlang, o interpretador verifica, usando *casamento de padrões*, qual das cláusulas da definição será usada para para avaliar a expressão. As cláusulas são analisadas seqüencialmente, começando na primeira, até que uma se encaixe na entrada passada para a função. Exemplo:

```
> fat(3)
   3 não é 0, então N=3 em fat(N)
   = 3 * fat(2)
     2 não é 0, então N=2 em fat(N)
     = 3 * 2 * fat(1)
       1 não é 0, então N=1 em fat(N)
       = 3 * 2 * fat(0)
         pela definição, fat(0)=1
         = 3 * 2 * 1

Resposta do interpretador:
6
```

Átomos e variáveis Átomos são seqüências de caracteres que denotam valores únicos. Átomos são diferentes de variáveis no sentido de que o valor de uma variável é seu conteúdo e o valor de um átomo é o próprio átomo. Átomos começam sempre com letras *minúsculas* enquanto variáves sempre começam com letras *maiúsculas*. Qualquer caracter pode ser usado em um átomo desde que o átomo seja definido entre aspas simples. Exemplos de átomos:

```
andre
janeiro
'Letras maiusculas e espacos'
```

Variáveis são usadas para armazenar valores e, assim como as variáveis usadas na matemática, podem receber valores *apenas uma vez*. Variáveis em Erlang não represetam uma posição de memória e sim um *nome* que é dado para um valor. Exemplo de variáveis:

```
X
Y1
Uma_variavel_longa
```

Tuplas Tuplas são estruturas que podem conter um número fixo de elementos de qualquer tipo, incluindo funções e tuplas. As tuplas podem ter qualquer tamanho. Tuplas funcionam como um agrupamento de valores, parecido com as estruturas (`structs`) da linguagem C.

```
{30,'Andre'}
{carro, vectra, gasolina}
{medico, {joao, pediatra}}
{}
```

Podemos definir funções que recebem tuplas como argumento e devolvem tuplas como resposta:

```
-module(tup).
-export([somaTupla/1,criaTupla/4]).

somaTupla({A,B}) -> A+B.

criaTupla(A,B,C,D) -> {A,B,C,D}.
```

A função `somaTupla` recebe como argumento uma tupla com dois valores `A` e `B` e devolve como resposta a soma desses dois valores. A função `criaTupla` recebe como argumento quatro valores e devolve como resposta uma tupla formada por esses valores. Exemplo de uso:

```
> tup:somaTupla({33,22}).
55


> tup:criaTupla(22,andre,33,1).
{22,andre,33,1}
```

O *padrão* que representa o argumento de uma função pode conter valores específicos:

```
area({quadrado,Lado}) -> Lado * Lado;
area({circulo,Raio}) -> 3.14 * Raio * Raio;
area(Outro) -> {objeto_invalido, Outro}.
```

Se a função área receber como argumento uma tupla cujo primeiro elemento é o átomo `quadrado` e o segundo argumento um valor qualquer (um valor não definido é sempre representado por uma variável, nesse caso a variável `Lado`), a função devolve a área do quadrado, ou seja, a expressão `Lado * Lado`. Se a função `area` recebe como argumento uma tupla cujo primeiro elemento é o átomo `circulo` e o segundo um valor qualquer `Raio`, a função retorna a área do círculo. Se a função receber como argumento um outro valor, que não é um quadrado nem um círculo, a função retorna uma tupla que representa uma mensagem de erro. Exemplos:

```
> tup:area({quadrado,22}).
484

> tup:area({circulo, 33}).
3419.46
```

```
> tup:area({idade,33}).
{objeto_invalido,{idade,33}}
```

Listas Listas são estruturas de dados que podem conter um número variável de elementos de qualquer tipo. Podem ter qualquer tamanho e, ao contrário das tuplas, o tamanho pode aumentar durante a execução do programa (ex: inserindo novos elementos na lista). Exemplos de listas:

```
[andre, joao, maria]
[abc, {medico, {joao, pediatra}}, 129]
"abc" --> [97,98,99]
[]
```

A expressão:

```
[1| [1,2,3]]
```

insere o valor 1 na lista [1,2,3]. Exemplos:

```
> [1|[1,2,3]].
[1,1,2,3]

> [[1,2,3,4]|[1,2,3]].
[[1,2,3,4],1,2,3]

> [atomo|[1,2,3]].
[atomo,1,2,3]
```

Funções que percorrem listas são construídas usando recursão. Para declarar recursão sobre listas usamos novamente *casamento de padrões*, sempre considerando que as listas podem estar em dois formatos: ou a lista está *vazia*, nesse caso a lista é representada pelo padrão **[]**, ou a lista está *não vazia*, nesse caso ela é representada pelo padrão **[H|T]**, onde **H** representa o primeiro elemento da lista, e **T** representa o resto da lista sem o primeiro elemento. Exemplos:

```
-module(listas).
-export([somaLista/1,tamanho/1,pega/2]).

somaLista([]) -> 0;
somaLista([H|T]) -> H + somaLista(T).

tamanho([]) -> 0;
tamanho([H|T]) -> 1 + tamanho(T).
```

A função `somalista`, recebe uma lista como argumento e soma todos os seus elementos. A definição nos diz, no primeiro caso, que a soma dos elementos de uma lista vazia é 0 (*zero*). O segundo caso nos diz que a soma dos elementos de uma lista não vazia é a soma do primeiro elemento (`H`) como a soma de todos os outros elementos (representado por `somaLista(T)`). A função `tamanho` também é implementada em dois casos: o primeiro nos diz que o tamanho de uma lista vazia é sempre 0 (*zero*). O segundo caso diz que o tamanho de uma lista não vazia é 1 mais o tamanho da lista sem o seu primeiro elemento. Exemplos de uso:

```
2> listas:tamanho([3,4,5,3,1]).
5
3> listas:somaLista([3,4,5,3,1]).
16
```

4.6.2. Programação concorrente

A programação concorrente em Erlang é baseada na idéia de *processos* e *mensagens*:

- **Processos:** Uma atividade concorrente. Um sistema pode possuir vários processos rodando ao mesmo tempo.
- **Mensagem:** Método de comunicação entre processos.

Para se criar um novo processo, usamos o comando `spawn`:

```
Pid = spawn (Modulo, Func, Args)
```

que recebe os seguintes argumentos:

- `Modulo`: o nome do módulo em que está definido o código a ser executado pelo novo processo.
- `Func`: uma função (código) que será executada pelo novo processo.
- `Args`: uma lista contendo os argumentos que serão aplicados é função para que o processo comece a executar.
- `Pid`: variável que, depois da chamada a `spawn`, recebe o nome do processo criado com `spawn`, ou seja, o seu *pid*.

O *pid* de um processo é um valor único que o identifica. Usamos o *pid* de um processo para lhe enviar mensagens. Para enviar uma `Mensagem` para um processo usamos o operador exclamação (`!`):

```
Pid!Mensagem
```

Essa expressão envia a `Mensagem` para o processo identificado pelo *pid* contido na variável `Pid`.

A função `self()` retorna o *pid* de um processo. A seguinte expressão

```
Pid2! {self(), mensagem}
```

envia uma tupla contendo o *pid* do rementente e um átomo (`mensagem`) ao processo `Pid2`. Todo o processo possui um *mailbox* ou caixa postal. Cada mensagem enviada para um processo fica guardada dentro dessa caixa postal até que o processo chame o comando `receive`. O comando `receive` retira uma mensagem da caixa postal, ou fica bloqueado esperando pelo recebimento de uma mensagem. As mensagens são retiradas da caixa postal por *casamento de padrões*. Por exemplo, para receber a mensagem enviada no exemplo anterior, podemos usar o seguinte `receive`:

```
receive
   {From, Msg} -> From ! alo
end
```

O comando receive é formado por uma seqüência de *padrões* e *ações*, separados pelo símbolo `->`:

```
receive
   Mensagem1 -> Acao1;
   Mensagem2 -> Acao2;
   ...
end.
```

Se uma mensagem na caixa postal se encaixa em um dos padrões especificados no `receive` então a mensagem é retirada da caixa postal, o padrão é instânciado, e a ação é executada. No exemplo, o primeiro programa envia a mensagem `{self(),`

`mensagem}`. Como a mensagem se encaixa no padrão do `receive`, esta é retirada da caixa postal e o padrão instânciado, ou seja, a variável `From` recebe o *pid* do processo que enviou a mensagem e a variável `Msg` recebe o átomo `mensagem`. O programa continua executando a expressão `From ! alo` que envia o átomo `alo` para o processo que enviou a primeira mensagem.

Segue um exemplo simples de troca de mensagens entre dois processos:

```
-module(conc1).
-export([proc1/0,proc2/0]).

proc1 () ->
          Pid2 = spawn (conc1, proc2, []),
          Pid2 ! self(),
          receive
               Mesg ->  io:format("Recebi: ~w~n",[Mesg])
          end.

proc2() ->
     receive
          Remetente ->  Remetente ! estouVivo
     end.
```

O processo `proc1` cria um novo processo para executar a função `proc2`. Em seguida envia para `proc2` seu próprio *pid*:

```
    Pid2 ! self()
```

O processo `proc2` no início de sua execução faz um `receive` e fica esperando pela mensagem com o *pid* de `proc1`. Quando recebe o *pid*, `proc2` envia de volta o átomo `estouVivo`. O processo `proc1` então recebe esse átomo e o imprime usando o comando `io:format`. O caractere de escape `~w` serve para imprimir o conteúdo de `Mesg` e o caractere de escape `~n` significa nova linha.

O comando `receive` pode possuir vários padrões mas cada `receive` retira apenas uma mensagem da caixa de mensagens do processo. Considere o seguinte trecho de programa:

```
Pid2 = spawn(...),
Pid3 = spawn(...),
receive
     {Pid2, M2} ->  io:format("Recebi do proc 2: ~w~n",[M2]);
     {Pid3, M3} ->  io:format("Recebi do proc 3: ~w~n",[M3])
end.
```

Nesse trecho de código, após criar dois processos, o programa faz um `receive` com dois padrões, um que permite receber uma mensagem do processo 2 (`{Pid2, M1}`) e outro que permite receber uma mensagem do processo 3 (`{Pid3, M3}`). Note que o casamento de padrões ocorre pelo *pid* dos processos que estão nas variáveis `Pid2` e `Pid3`. O `receive`, apesar de possuir dois padrões, permite o recebimento de apenas uma mensagem. O que significa que a primeira mensagem que chegar será retirada da caixa postal. Já o seguinte trecho de código:

```
Pid2 = spawn(...),
Pid3 = spawn(...),
receive
```

```
     {Pid2, M2} ->  io:format("Recebi do proc 2: ~w~n",[M2])
end,
receive
     {Pid3, M3} ->  io:format("Recebi do proc 3: ~w~n",[M3])
end.
```

permite o recebimento de duas mensagens, primeiramente uma mensagem do processo 2 (`Pid2`) e logo em seguida uma mensagem do processo 3 (`Pid3`).

4.6.3. Erlang distribuído

Como os programas em Erlang já são construídos usando a abstração de troca de mensagens, fica muito fácil modificar um programa concorrente para que este funcione em um *cluster* ou rede de larga escala, como a Internet. O comando `spawn` pode ser usado para iniciar um processo em uma máquina remota:

```
Pid = spawn(Fun@Node)
```

Para que uma máquina na rede possa participar de um sistema distribuído esta deve usar a primitiva `alive`:

```
alive(Node),
```

A primitiva `alive` faz com que o nodo Erlang fique visível para outros nodos. A primitiva `not_alive` torna um nodo invisível:

```
not_alive(Node)
```

Dessa maneira o `Nodo` não pode receber mensagens de outros nodos.

4.6.4. Substituindo código em tempo de execução

Um recurso poderoso da linguagem Erlang é a possibilidade de substituir código em tempo de execução. Como Erlang é uma linguagem funcional, funções são os elementos principais da linguagem. Dessa maneira funções podem receber funções como argumento, funções podem devolver funções como a resposta de uma computação e funções podem ser comunicadas através de mensagens:

```
loop(F)
  receive
        {requisicao, Pedido,  Pid} ->
              Resultado = F(Pedido),
              Pid ! Resultado,
              loop(F);
        {muda_codigo, F2} -> loop(F2)
  end
```

Nesse exemplo, quando o programa recebe uma mensagem com o átomo `requisicao`, ele processa o pedido enviado pelo cliente usando uma função `F`. Quando o programa recebe uma mensagem com o átomo `muda_codigo`, o código (função) que processa os pedidos é substituído pelo código contido na mensagem (`F2`).

4.6.5. Tolerância a falhas

Para tratamento de erros a linguagem Erlang fornece as primitivas `catch`, `exit` e `throw`. A primitiva `catch` serve para converter uma exceção em um valor:

```
Y = (catch 1/0).
```

O conteúdo da variável `Y` é uma tupla que descreve a exceção gerada pela divisão por zero:

```
{'EXIT',{badarith,[{erl_eval,eval_op,3},
                   {erl_eval,expr,5},
                   {erl_eval,expr,5},
                   {shell,exprs,6},
                   {shell,eval_loop,3}]}}
```

Quando uma expressão do tipo `(catch Expr)` avalia para `{'EXIT', W}` significa que ocorreu uma exceção na avaliação da expressão e o motivo está descrito em `W`. A primitiva `exit` serve para explicitamente gerar uma exceção. Se na implementação de uma função `F` uma exceção é gerada pela expressão `exit(E)`, então a avaliação de `catch F` vai gerar `{'EXIT', E}`:

```
> catch exit(erroNaExpressao).
{'EXIT',erroNaExpressao}
```

A primitiva `throw` serve para mudar o formato das exceções geradas. A expressão `catch throw(E)` avalia sempre para `E`.

Monitorando processos Processos podem ser agrupados (*ligados*) em conjuntos chamados de *link sets*. Quando um processo do conjunto falha, todos os outros que pertencem ao mesmo conjunto são notificados. Um processo `A` pode ser *ligado* ao processo `B` avaliando a primitiva `link`:

```
link(B).
```

A operação `link` é simétrica de modo que se um processo `A` está agrupado com o processo `B` então `B` também está ligado ao `A`. Processos podem ser agrupados diretamente no momento de criação:

```
Pid = spawn_link(...)
```

Usando `spawn_link` o processo criado fica automaticamente ligado ao processo que o criou. Processos ligados em grupos podem receber mensagens que descrevem o motivo de falha de outros processos, ou podem ser programados para terminar no momento em que ocorre um falha em algum dos processos do grupo.

A primitiva `monitor` é usada para agrupar processos de forma assimétrica. Por exemplo, se temos um grupo de clientes que pedem serviços a um servidor, quando o servidor pára por algum motivo os clientes podem ser terminados, porém quando um dos clientes pára, isso não deve interferir no trabalho do servidor nem dos outros clientes. Se um processo `A` executa:

```
Ref = erlang:monitor(process, B)
```

então se o processo `B` morre, o processo `A` recebe a seguinte mensagem:

```
{'DOWN', Ref, process, B, Motivo}
```

onde a variável `Motivo` descreve a razão do processo terminar.

4.6.6. Behaviours

Erlang define funções de alta-ordem (funções que recebem funções como argumentos) que encapsulam padrões recorrentes de computação concorrente. Essas funções, que são chamadas de *behaviours*, fornecem o esqueleto de um programa concorrente. O programador usa esse esqueleto configurando o mesmo através de seus argumentos. Todo o comportamento concorrente da aplicação já está definido dentro do esqueleto. O programador só define as partes específicas da aplicação. Por exemplo, o *behaviour* `gen_server` define um servidor genérico tolerante a falhas. A função `gen_server` fornece todas as funcionalidades comuns a um servidor, além de permitir modificação do código do servidor em tempo de execução da aplicação. Outros *behaviours* disponíveis em Erlang são: `gen_fsm`, que permite a implementação de máquinas de estado finitas, `supervisor` que permite a conexão de processos formando uma *árvore supervisonada*, entre outros.

4.7. O Jantar dos Filósofos

O *jantar dos filósofos* é um problema clássico de programação concorrente que foi proposto por Dijkstra em 1965. Cinco filósofos estão sentados em uma mesa redonda e cada um possui um prato de comida em sua frente. Para poder comer, um filósofo necessita de dois garfos, só que existe somente um garfo entre cada prato, assim como na Figura 1. No contexto do problema, um filósofo faz apenas duas ações: comer ou pensar. Toda a vez que sente fome, o filósofo tenta pegar dois garfos, um à direita e outro à esquerda para comer. Se conseguir pegar os dois garfos então o filósofo come um pouco, coloca os garfos novamente na mesa e volta a pensar.

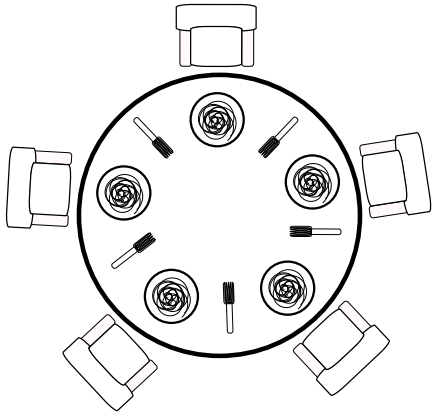

Figura 1: O Jantar dos Filósofos.

4.7.1. Implementação usando STM Haskell

Nesta seção usamos STM Haskell para implementar uma solução para o problema do *jantar dos filósofos*. O programa apresentado nesta seção é baseado na solução proposta em [Huch and Kupke, 2007].

Inicialmente, os garfos são representados por uma `TVar` que guarda um valor booleano indicando se o garfo está disponível (`True`) ou não (`False`).

```
type Garfo = TVar Bool
```

A palavra reservada `type` em Haskell serve para se criar *sinônimos de tipos* no sentido de que o tipo `Garfo` é equivalente ao tipo `TVar Bool`. Sinônimos de tipo servem para facilitar a leitura do código.

Um filósofo pode ser representado pela seguinte função:

```
filosofo ::    Int            -- Nome do Filosofo
            -> Garfo          -- Garfo da Esquerda
            -> Garfo          -- Garfo da Direita
            -> IO ()
filosofo n e d = do
                 print ("Filósofo está com Fome!");
                 atomically ( do
                              pegaGarfo e
                              pegaGarfo d)
                 print ("O filósofo " ++ (show n)
                            ++ " esta comendo!")
                 atomically ( do
                              colocaGarfo e
                              colocaGarfo d)
                 print ("Filósofo está pensando!")
                 pensa
                 filosofo n e d
```

O `filosofo` recebe como argumentos um inteiro que o identifica e duas referências a garfos, um que está à sua esquerda e outro à direita. Primeiramente o filósofo realiza uma ação atômica que serve para adquirir os dois garfos. Uma vez adquiridos, o filósofo pode comer. Em seguida, este realiza outra ação atômica que serve para devolver os garfos para a mesa. Logo após, o filósofo `pensa` e então tenta novamente comer fazendo uma chamada recursiva à função `filosofo`.

A função `pegaGarfo` serve para ler a `TVar` que contém o garfo:

```
pegaGarfo :: Garfo -> STM ()
pegaGarfo g = do
              b <- readTVar g
              check b
              writeTVar g False
```

Se o garfo não se encontra disponível na `TVar` então o `retry` é chamado através da função `check` (Seção 4.4.3). A implementação da função `retry` faz com que esta ação seja tentada novamente no momento em que a `TVar` contendo o garfo seja modificada.

A função `colocaGarfo` possui uma implementação mais simples:

```
colocaGarfo :: Garfo -> STM ()
colocaGarfo g = writeTVar g True
```

Como no momento em que a função `colocaGarfo` é chamada os garfos estão com o filósofo, então sabe-se que o valor guardado na `TVar` é `False`, ou seja, o garfo não está disponível. Como o objetivo de `colocaGarfo` é devolver o garfo para a `TVar`, a função simplesmente escreve `True` na variável transacional.

Quando um filósofo `pensa` a sua thread fica sem fazer nada. Dessa maneira, a implementação de `pensa` usa a função `threadDelay` para que a thread do filósofo durma por um determinado número de milisegundos:

```
pensa :: IO ()
pensa =  threadDelay 1000000
```

Por último, usa-se um programa um pouco mais complicado, que serve para inicializar as threads que contêm os filósofos passando para elas as referências às variáveis transacionais que contêm o estado dos garfos:

```
iniciaFilosofos :: Int -> IO ()
iniciaFilosofos n =
       do
        garfosIO <- atomically (criaGarfos n)
        mapM_ (\(l,r,i)->forkIO (filosofo i l r))
              (zip3 garfosIO (tail garfosIO) [1..n-1])
        filosofo n (last garfosIO) (head garfosIO)
```

Além disso, uma função auxiliar `criaGarfos` é usada para criar a lista contendo todos os `Garfos`. Inicialmente todos os garfos estão disponíveis, ou seja, o valor guardado nas `TVars` é `True`:

```
criaGarfos :: Int -> STM [Garfo]
criaGarfos n = do
     garfos <- mapM (const (newTVar True)) [1..n]
     return garfos
```

4.7.2. O jantar dos filósofos em Erlang

Nesta seção apresentamos uma implementação do problema do *jantar dos filósofos* em Erlang. Existem várias maneiras de solucionar esse problema usando troca de mensagens e aqui apresentamos uma implementação usando o algoritmo apresentado em [Krishnaprasad, 2003]. A idéia da implementação é usar um processo que sincronize o acesso aos garfos. Os filósofos rodam em processos separados e sempre que um filósofo necessita comer, ele envia uma mensagem para o processo sincronizador para pedir acesso aos garfos à sua esquerda e direita.

Seguindo essa linha de pensamento, o filósofo pode ser implementado da seguinte maneira:

```
filosofo(Id,Servidor) ->
   Servidor ! {self(), Id,pegaGarfos},
   receive
     ok  -> io:format("Filósofo ~w comendo!~n",[Id]),
            Servidor! {Id, devolveGarfos},
            pensa(),
            filosofo(Id,Servidor)
   end.
```

O `filosofo` é uma função que recebe como argumentos um identificador (`Id`), que é um número inteiro que serve para identificá-lo, e o nome do servidor que serve para sincronizar o acesso aos garfos (`Servidor`). O filósofo primeiramente envia uma mensagem ao servidor pedindo acesso aos garfos:

```
Servidor ! {self(), Id,pegaGarfos},
```

Nessa mensagem o filósofo envia o seu *pid*, para que possa receber uma resposta, seu identificador `Id`, e o átomo `pegaGarfos` que indica o objetivo da mensagem. Quando o filósofo recebe como resposta a mensagem `ok`, significa que os garfos estão disponíveis e que o filósofo pode comer. Quando termina de comer, este envia uma mensagem novamente ao servidor indicando que agora os garfos estão livres. Após liberar os garfos, o filósofo `pensa` e repete a operação.

O ato de pensar faz com que o processo corrente pare por um segundo:

```
pensar() -> sleep (1000).
```

O servidor de sincronização do uso dos garfos é implementado pela função `servidorDeGarfos`:

```
servidorDeGarfos(Garfos,Fila) ->
 receive
   {Filosofo, Id , pegaGarfos} ->
        Garfo1 = Id,
        Garfo2 = (Id+1) rem 5,
        Disponivel = (pega(Garfo1,Garfos))
                     and (pega(Garfo2,Garfos)),
        case Disponivel of
         true ->
           Tmp = muda(Garfo1,false,Garfos),
           GarfosN = muda(Garfo2,false,Tmp),
           Filosofo!ok,
           servidorDeGarfos(GarfosN,Fila);
         false ->
           servidorDeGarfos(Garfos,
                   adicionaFila({Filosofo,Id},Fila))
        end;

    {Id,devolveGarfos}->
        io:format("Recebi Garfo do filósofo ~w~n",[Id]),
        Garfo1 = Id,
        Garfo2 = (Id+1) rem 5,
        Tmp = muda(Garfo1,true,Garfos),
        GarfosN = muda(Garfo2,true,Tmp),
        {GarfosN2,Fila2} = acorda (GarfosN,Fila),
        servidorDeGarfos(GarfosN2,Fila2)
end.
```

Este servidor recebe dois argumentos. O primeiro, `Garfos`, é uma lista de booleanos, cada um representando um garfo. Se um booleando está como `true`, significa que o garfo está disponível, se está como `false` significa que o garfo está em uso. O segundo argumento é uma `Fila` de processos (`filosofo`s) bloqueados esperando por garfos. O servidor inicialmente faz um `receive` para receber mensagens enviadas por filósofos. As duas mensagens que o servidor trata são os dois tipos de mensagens que podem ser enviadas por filósofos: a mensagem para pedir os garfos (`pegaGarfos`) e a mensagem para liberar os garfos (`devolveGarfos`). A primeira mensagem que o servidor trata é a mensagem que serve para pedir os garfos:

```
    {Filosofo, Id , pegaGarfos} ->
```

Nesse caso o servidor, baseado no `Id` do filósofo, calcula as posições dos garfos que o filósofo está pedindo. O filósofo sempre tenta pegar o garfo que está na posição do seu `Id` na lista de garfos (variável `Garfo1`) e o próximo garfo (variável `Garfo2`). A expressão:

```
Disponivel = (pega(Garfo1,Garfos))
                     and (pega(Garfo2,Garfos)),
```

verifica se os dois garfos que o filósofo está pedindo estão disponíveis na lista de `Garfos`. A função `pega` recebe dois argumentos, uma posição e uma lista, retornando

o elemento que está nessa posição da lista. Se a variável `Disponível` possui `true`, significando que os dois garfos estão disponíveis, então os dois garfos são marcados como `false` na lista de `Garfos` usando a função `muda` e a mensagem `ok` é enviada para o filósofo. As funções `pega` e `muda` são funções simples de manipulação de listas e ficam como exercício de implementação para o leitor. Se os garfos não estão disponíveis, então o servidor adiciona o filósofo na fila de espera por garfos (`Fila`), usando a função `adicionaFila`.

A outra mensagem que o servidor de garfos pode receber é a mensagem que serve para o filósofo devolver os garfos que estava usando:

```
{Id,devolveGarfos}->
```

Quando recebe esta mensagem o servidor, usando o `Id` do filósofo, modifica a lista de garfos para torná-los disponíveis (usando a função `muda`) e percorre a fila de filósofo bloqueados para acordar um filósofo que esteja esperando pelos talheres devolvidos. A função `acorda` pode ser implementada da seguinte maneira:

```
acorda(G,F) ->
   case procuraFilosofo(G,F) of
      {Filosofo,Id,G2} -> Filosofo!ok,
                          {G2,retira({Filosofo,Id},F)};
      false -> {G,F}
   end.
```

A função `procuraFilosofo` procura na fila um filósofo que possa ser desbloqueado. Caso esse filósofo exista, a mensagem `ok` é enviada para o filósofo encontrado e este é retirado da fila de filósofos bloqueados.

Por último, usa-se uma função `start()` que serve para inicializar o processo servidor e também os filósofos:

```
start() ->
   Garfos = [true,true,true,true,true],
   Servidor = spawn(filosofos,servidorDeGarfos,[Garfos,[]]),
   spawn (filosofos, filosofo, [0, Servidor]),
   spawn (filosofos, filosofo, [1, Servidor]),
   spawn (filosofos, filosofo, [2, Servidor]),
   spawn (filosofos, filosofo, [3, Servidor]),
   spawn (filosofos, filosofo, [4, Servidor]).
```

Obviamente, quando o servidor de garfos é inicializado todos os garfos estão livres (a lista de booleanos contém apenas `true`) e a fila de processos bloqueados está vazia. Logo após inicializar o servidor a função `start()` cria cinco filósofos, cada um deles recebe como argumento um inteiro que o identifica e uma referência para o servidor de sincronização.

4.8. Outras Alternativas

Nesta seção são apresentadas outras alternativas interessantes para a programação de sistemas concorrentes. Essas alternativas também são de alguma maneira baseadas em linguagens funcionais. Obviamente não pretendemos esgotar o assunto neste texto.

4.8.1. Futures

Várias linguagens de programação apresentam uma abstração um pouco mais simples para a programação *multi-thread* usando a idéia de *futuros*. Futuros é uma abstração para

programação concorrente que foi primeiramente desenvolvida na linguagem Lisp, porém abstrações parecidas são fornecidas em várias outras linguagens funcionais e também em linguagens como Java. Por exemplo, a linguagem Alice [Rossberg et al., 2006] foi desenvolvida para a programação de sistemas distribuídos fortemente tipados. Concorrência é fornecida na linguagem através do conceito de futuros. Um futuro é um valor que pode ainda não ter sido computado. A expressão:

```
val x = spawn exp
```

cria em `x` um novo futuro que representa o valor ainda não computado de `exp`. Uma nova thread é criada automaticamente para avaliar `exp` e assim que a thread termina a avaliação, o resultado substitui o futuro criado. Uma thread *toca* um futuro quando tenta usar o valor que esse futuro representa. Uma thread que toca um futuro é suspensa até que o valor seja computado, fornecendo assim sicronização por fluxo de dados. Quando a avaliação de um futuro gera uma exceção, qualquer thread que tocar esse futuro vai receber essa exceção.

Java define uma interface `Future`, que serve para modelar a idéia de futuros. Qualquer objeto que implemente esta interface representa o futuro de uma computação.

4.8.2. C Omega

Em [Benton et al.,], é apresentada uma extensão da linguagem C# para programação concorrente baseada no cálculo Join [Fournet and Gonthier, 1996] e na linguagem Jocaml [Conchon and Fessant, 1999]. O cálculo Join fornece uma teoria de processos, semelhante ao cálculo pi [Milner, 1999], que serve para modelar tanto concorrência local quanto distribuída.

Em C Omega todos os objetos podem possuir métodos *síncronos* e *assíncronos*. Os métodos síncronos funcionam como os métodos normais existentes em qualquer linguagem orientada a objetos: o objeto que chamou o método fica bloqueado esperando a resposta da chamada. Nos métodos assíncronos, a chamada de método retorna automaticamente sem retornar um resultado (void). Uma classe também pode definir *chords*, ou *padrões de sincronização*, que são ações que são executadas somente quando um certo conjunto de métodos for executado. No momento em que todos os métodos descritos no *chord* são chamados, então o corpo da definição do *chord* roda. Em uma classe podem existir vários *chords* e o mesmo método pode participar da definição de *chords* diferentes. Na medida em que os vários métodos são chamados isso é registrado internamente até que todos os métodos necessários para rodar um *chord* tenham sido chamados. Se os métodos chamados batem com a definição de mais de um *chord* então um é escolhido aleatoriamente para rodar. Se o padrão de um *chord* define apenas métodos assíncronos, então uma nova thread é criada para rodar o seu corpo. Por exemplo, pode-se definir um *buffer* em C Omega da seguinte maneira:

```
class Buffer {
  String get() & async put(String s) {
     return s;
  }
}
```

Nesta definição temos um *chord* cujo padrão é composto pela chamada ao método síncrono `get()` (que retorna uma `String`) e o método assíncrono `put()`. A chamada ao método `put()` retorna automaticamente por ser assíncrona, mas é colocada em espera caso não houver nenhuma chamada ao `get()`. Da mesma maneira, uma chamada ao

`get()` fica bloqueada até que exista uma chamada ao método `put()`. Se ocorrer as duas chamadas então o argumento passado para o `put()` é retornado como resposta para chamada ao `get()`. No padrão de cada *chord* só pode existir um método síncrono que é o método que retornará a resposta.

Através dos *chords* a linguagem C Omega oferece uma alternativa simples para a sincronização de threads tanto locais quanto distribuídas. Esta abordagem possui uma teoria formal consolidada e através dessas construções simples podemos implementar primitivas mais complexas e com um alto poder de expressão.

4.8.3. pFun

Com o surgimento de máquinas multi-core domésticas, a comunidade de linguagens de programação começa novamente a direcionar sua atenção para as linguagens funcionais paralelas [Harris and Singh, 2007, Harris et al., 2005a]. Nesses trabalhos recentes, geralmente tenta-se dois extremos: ou trabalha-se com uma linguagem funcional totalmente explícita, com threads e primitivas de sincronização [Harris et al., 2005a], ou com linguagens totalmente implrícitas, onde o paralelismo é automaticamente deduzido pelo compilador da linguagem [Harris and Singh, 2007].

Para atingir o objetivo de programar máquinas multi-core de forma simples, em [Du Bois et al., 2007], foi proposta uma linguagem funcional com paralismo *semi-explícito*. A linguagem pFun possui uma sintaxe parecida com a de Haskell estendida com duas primitivas: `par` e `sync`. A primitiva `par` serve para anotar no código expressões que poderiam ser avaliadas em paralelo:

```
par :: a -> Par a
```

A primitiva recebe uma expressão qualquer (de um tipo qualquer `a`) e retorna um objeto `Par a`, indicando que a expressão pode ser avaliada em paralelo. As expressões anotadas com `par` não serão obrigatoriamente avaliadas em paralelo. A criação, escalonamento, sincronização e comunicação entre as *threads* é tarefa da máquina virtual que implementa a linguagem e não é preocupação do programador.

A primitva `sync` recebe como argumento um objeto do tipo `Par a` e retorna o resultado da avaliação de `a`:

```
sync :: Par a -> a
```

O comportamento de `sync` é ficar bloqueado se o seu argumento estiver sendo avaliado ou começar a avaliação da expressão caso contrário. A função `sync` só retorna quando a expressão de tipo `a` estiver completamente avaliada. Note que as funções `par` e `sync` fornecem uma abstração parecida com a de futuros.

Usando as primitivas simples da linguagem pFun, podemos implementar abstrações de mais alto nível, como por exemplo *algorithmic skeletons* [Cole, 1989], que são funções de alta ordem que encapsulam padrões de paralelismo. Um exemplo desse tipo de abstração é a função `map` paralela, que recebe como argumentos uma função e uma lista, e aplica essa função a todos os elementos de uma lista em paralelo.

Para implementar essa função em pFun, primeiramente precisamos de uma função que avalie a aplicação de uma função a todos os elementos de uma lista em paralelo:

```
parList:: (a->b) -> [a] -> [Par b]
parList f l = case l of
        [] -> [];
        x:xs -> (par (f x)) : (parList f xs);
```

A função `parList` cria uma lista de objetos do tipo `Par`. Precisamos então uma função que pegue os valores computados por esses objetos:

```
syncList :: [Par a] -> [a]
syncList list = map sync list;
```

Finalmente, o função `parMap` pode ser implementada usando `parList` e `syncList`:

```
parMap :: (a->b) -> [a] -> [b]
parMap f l = syncList (parList f l)
```

A função `parMap` primeiramente usa `parList` para criar todas as tarefas paralelas e em seguida usa `syncList` para coletar os resultados.

4.9. Considerações Finais

A popularização de máquinas multi-core está causando uma revolução no desenvolvimento de software. Para se programar esse tipo de processador geralmente usa-se um modelo em que threads se comunicam através de uma área de memória compartilhada e bloqueios são usados para a sincronização de tarefas. Esse modelo de programação possui vários problemas: é difícil de programar, dificulta o reuso de código e erros no software são de difícil correção.

Neste texto foram apresentadas algumas alternativas para a programação de máquinas multi-core que são um pouco diferentes da tradicional programação usando threads, memória compartilhada e bloqueios. Em especial foi apresentada a programação usando memórias transacionais na linguagem STM Haskell e programação usando troca de mensagens em Erlang.

Memórias transacionais permitem programação concorrente sem os vários problemas relacionados com o uso de bloqueios. Transações não conseguem ver o resultado intermediário de outras transações: dessa maneira, o resultado de várias transações concorrentes é o mesmo resultado da execução dessas threads em alguma ordem seqüencial, o que permite a eliminação do uso bloqueios. Essas características permitem o desenvolvimento de código mais limpo e fácil de entender, como pode ser analisado no exemplo do *jantar dos filósofos* apresentado na Seção 4.7.

A linguagem Erlang é baseada na idéia de processos completamente independentes, sem memória compartilhada, como se cada processo rodasse em sua própria máquina. Processos comunicam-se apenas através de troca de mensagens. Todos os dados são copiados para as mensagens pois não existem ponteiros. Quando elimina-se totalmente a memória compartilhada entre processos, mecanismos de sincronização como *mutexes* não são necessários. Erlang é uma linguagem estável desenvolvida pela Ericsson e usada em vários produtos comerciais que apresentam alto nível de concorrência e tolerância a falhas.

4.10. Agradecimentos

Agradeço ao Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq) pelo apoio. Agradeço também ao Mateus Rauber Du Bois e ao prof. Gerson Cavalheiro pela leitura de versões preliminares deste texto.

Referências

Adl-Tabatabai, A.-R., Kozyrakis, C., and Saha, B. (2006). Unlocking concurrency. *ACM Queue*, 4(10):24–36.

Armstrong, J. (2003). *Making reliable distributed systems in the presence of errors*. PhD thesis, Royal Institute of Technology, Stockholm.

Barendregt, H. (1984). *The Lambda Calculus, Its syntax and Semantics*. North Holland.

Benton, N., Cardelli, L., and Cédric, F. Modern concurrency abstractions for C#. *ACM Transactions on Programming Languages and Systems*, 26(5):269–304.

Blauth, P. and Haeusler, E. H. (2002). *Teoria das Catogorias para Ciência da Computação*. Sagra Luzzatto.

Cavalheiro, G. G. H. (2006). Princípios da programação concorrente. In *Caderno dos Cursos Permanentes, Escola Regional de Alto Desempenho*.

Cole, M. (1989). *Algorithmic Skeletons: Structured Management of Parallel Computation*. Pitman.

Conchon, S. and Fessant, F. L. (1999). Jocaml: Mobile agents for Objective-Caml. In *ASA'99/MA'99*, Palm Springs, CA, USA.

Du Bois, A. R., Cavalheiro, G., Yamin, A., and Pilla, M. (2007). pFun: A semi-explicit parallel purely functional language. In *Electronic Proceedings of the First Workshop On Languages and Tools for Parallel and Distributed Programming*.

Fournet, C. and Gonthier, G. (1996). The reflexive CHAM and the join-calculus. In *Proceedings of the 23rd ACM Symposium on Principles of Programming Languages*, pages 372–385. ACM Press.

Geist, A., Beguelin, A., Dongerra, J., Jiang, W., Manchek, R., and Sunderam, V. (1994). *PVM: Parallel Virtual Machine*. MIT.

Gropp, W., Lusk, E., and Skjellum, A. (1999). *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. MIT, second edition.

Hammond, K. and Michaelson, G., editors (1999). *Research Directions in Parallel Functional Programming*. Springer-Verlag, UK.

Harper, R., MacQueen, D., and Milner, R. (1986). Standard ML. Technical report, LFCS, University of Edinburgh.

Harris, T., Marlow, S., and Peyton Jones, S. (2005a). Haskell on a shared-memory multiprocessor. In *Haskell '05: Proceedings of the 2005 ACM SIGPLAN workshop on Haskell*, pages 49–61. ACM Press.

Harris, T., Marlow, S., Peyton Jones, S., and Herlihy, M. (2005b). Composable memory transactions. In *PPoPP'05*. ACM Press.

Harris, T. and Singh, S. (2007). Feedback directed implicit parallelism. In *ACM ICFP 2007*. ACM Press.

Heller, M. (2003). Erlang. *Byte Magazine*.

Herlihy, M. and Moss, E. (1993). Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*.

Huch, F. and Kupke, F. (2007). Composable Memory Transactions in Concurrent Haskell. In *Implementation and Applications of Functional Programming Languages 2007*. Springer-Verlag.

Jones, S. L. P. (2003). Haskell 98 language and libraries: the revised report. *Journal of Functional Programming*, 1(13).

Krishnaprasad, S. (2003). Concurrent/distributed programming illustrated using the dining philosophers problem. *Consortium for Computing in Small Colleges*, 18(4):104–110.

Lee, E. A. (2006). The problem with threads. *IEEE Computer*, 39(5):33–42.

Milner, R. (1999). *Communicating and Mobile Systems: The π-Calculus*. Cambridge University Press.

OCaml (2002). OCaml. WWW page, http://caml.inria.fr/.

Peyton Jones, S. (2007). *Beautiful Concurrency*. O'Reilly.

Rajwar, R. and James, G. (2003). Transactional execution: Toward reliable, high-performance multithreading. *IEEE Micro*, 23(6):117–125.

Rigo, S., Centoducatte, P., and Baldassin, A. (2007). Memórias transacionais: Uma nova alternativa para programação concorrente. In *Anais Eletrônicos do WSCAD 2007*.

Rossberg, A., Botlan, L. D., Tack, G., Brunsklaus, T., and Smolka, G. (2006). Alice through the looking glass. In Loidl, H.-W., editor, *Trends in Functional Programming*, volume 5.

Tanenbaum, A. S. and Woodhull, A. S. (1997). *Operating Systems, Design and Implementation, 2nd Edition*. Prentice Hall.

Wadler, P. (1990). Comprehending monads. In *Proceedings of the 1990 ACM conference on LISP and functional programming*, pages 61–78. ACM Press.

Winston, P. and Horn, B. (1989). *Lisp*. Addison Wesley, 3rd edition.