O CONCEITO DE TEMPO NAS LINGUAGENS DE PROGRAMAÇÃO

Thiago Negri[1], Roque Cesar Possamai[1]
[1]Não Informado
evohunz@gmail.com, roquecp@gmail.com

Resumo
A arquitetura atual de computadores pede para que os aplicativos estejam preparados para utilizar mais de um núcleo de processamento. Este trabalho verificará se o modelo de programação atual, i.e. orientado a objetos, permite que os programas façam bom uso desta arquitetura, evidenciando possíveis motivos para procurar alternativas para acompanhar a evolução arquitetural. Trará uma breve explanação da história das linguagens de programação, contextualizando os motivos que levaram o mercado até o paradigma orientado a objetos. Através destes motivos e da evolução da arquitetura, será feita uma projeção sobre algumas razões que poderão fazer com que o estilo de programação atual seja substituído. Serão abordados os conceitos de concorrência e paralelismo que se tornam cada vez mais necessários com esta nova arquitetura. Esta abordagem será feita no modelo atual, em Java, e, trazendo como uma possível alternativa, o ponto de vista do modelo funcional puro e de avaliação preguiçosa, mais precisamente na linguagem Haskell.

Palavras-chave: Concorrência. Paralelismo. Linguagem de programação

Abstract
The current architecture of computers calls for applications to be prepared to use more than one processing core. This paper will verify if the current programming model, i.e. object oriented, allows programs to make good use of this architecture, showing possible reasons to seek alternatives to accompany the architectural evolution. It will bring a brief explanation of the history of programming languages, contextualizing the reasons that led the market to the object-oriented paradigm. Through these reasons and the evolution of architecture, a projection will be made on some reasons that may cause the actual style of programming to be replaced. It will examine the concepts of concurrency and parallelism that are becoming more necessary with this new architecture. This examination will be made in the current model, in Java, and bringing as a possible alternative, the point of view of the functional model of pure and lazy evaluation, specifically in the Haskell language.

Keywords: *Concurrency. Parallelism. Programming language.*

1. Introdução

Está ocorrendo uma transição de arquitetura nos computadores populares. Saindo de um mundo onde apenas a frequência determinava a capacidade computacional e chegando a um universo onde diversos núcleos de processamento se concentram na solução de um único problema. A arquitetura multi-núcleo torna-se cada vez mais presente no dia-a-dia da sociedade. Atualmente, os mais variados tipos de equipamentos já fazem uso desta tecnologia, *e.g.* celulares, *tablets*, notebooks e computadores de mesa.

Este artigo tem como objetivo verificar se os conceitos presentes nas linguagens de programação atuais são suficientes para esta nova arquitetura ou se outros conceitos podem ajudar para que os aplicativos façam bom uso destes equipamentos. A linguagem mais popular, hoje, é o Java (TIOBE SOFTWARE, 2011) e por este motivo os exemplos que dizem respeito ao cenário atual serão descritos utilizando esta linguagem.

Existem diversos modelos de programação que divergem do modelo orientado a objetos que a linguagem Java oferece. Além da orientação a objetos, existem: linguagens funcionais[1], lógicas[2], orientadas ao fluxo de dados[3] e outras. Este trabalho trará como contraponto ao modelo orientado a objetos, o ponto de vista de uma linguagem funcional mais antiga do que Java: Haskell.

O foco da comparação das perspectivas será na dificuldade para construir, manter e pensar sobre um código que utilize os equipamentos multi-núcleos atuais, priorizando a legibilidade do código e a possibilidade de abstrair conceitos de baixo nível.

2. História das linguagens

Na década de 40 surgiram os primeiros computadores (COMPUTER HOPE, 2011) e, devido à capacidade computacional limitada, era necessário fazer código *Assembly* à mão para programá-los. Não demorou muito para ser percebido que esta atividade requeria muito esforço mental e era extremamente sujeita à falhas. O mercado precisava de algo que pudesse gerar este código de forma automática, evitando erros e proporcionando um nível maior de abstração para que a atividade de programação se tornasse mais fácil e pudesse ser discutida sem tantos detalhes técnicos.

Em busca desta abstração, nasceram linguagens direcionadas a algumas áreas, como FORTRAN para a física e COBOL para o ramo de negócios. Na década de 70 nasceu a linguagem C que se tornou extremamente popular por atender aos mais variados segmentos de mercado e áreas acadêmicas (RITCHIE, 1993). Baseadas nesta, surgiram as linguagens mais utilizadas nos dias atuais: Java, C++ e C# (TIOBE SOFTWARE, 2011).

3. Por que saímos do C?

A linguagem C foi um *boom* na área. Captou adeptos ao redor de todo o mundo e tornou-se a linguagem mais utilizada muito rapidamente (RITCHIE, 1993).

Resumidamente, aprender C é simples. Toda a complexidade adicionada à sintaxe para diferenciar uma estrutura dinâmica de uma estática é um simples asterisco colocado na declaração de seu tipo. Porém, utilizar corretamente estas estruturas, chamadas de ponteiros, é muito difícil (PROVENZA, 2011). A linguagem não oferece uma forma de garantir como esta estrutura deve ser utilizada. Cada programador pode adotar seus próprios padrões e tudo o que pode ser feito para tentar diminuir problemas futuros é documentar estes detalhes. O problema é que esta documentação tende a ficar desatualizada com o tempo e acaba perdendo sua confiabilidade (MARTIN, 2009).

Foi esta complexidade implícita na linguagem que fez muitas pessoas procurarem por algo mais fácil (HICKEY, 2009). É neste momento que Java surge como uma ótima alternativa e de fácil adoção para os usuários de C.[4]

4. Por que Java?

[1] e.g. Common Lisp, Scheme, Haskell, Clojure, Erlang.
[2] e.g. Prolog, Ciao, Oz, Mercury, ALF.
[3] e.g. Agilent VEE, AviSynth, LabVIEW, Microsoft Visual Programming Language.
[4] sintaxe e estilo de programação parecidas

As memórias dos computadores aumentaram e permitiram que os programas manipulassem mais informações ao mesmo tempo. O gerenciamento desta memória se tornou muito complexo para ser realizado na linguagem C e Java adicionou uma ilusão de uma memória infinita a este mundo (CLICK, 2011).

Toda a responsabilidade de controlar o ciclo de vida de um objeto na memória sai das mãos do programador e vai para o coletor de lixo da linguagem. Isto eliminou quase toda a complexidade implícita referente ao gerenciamento de memória que existia na linguagem C. A linguagem só conseguiu fazer isto porque impediu que os programadores manipulassem livremente endereços de memória. Ela diminuiu a liberdade dos programadores em troca da garantia de que a memória fosse controlada corretamente pela máquina virtual.

A migração para a linguagem Java ficou bem visível no mercado. Segundo o índice TIOBE Software (2011), em 1996 a linguagem C era a primeira colocada no ranking de linguagens mais utilizadas pelo globo enquanto que Java estava em sexto. Apenas dez anos mais tarde, em 2006, Java já estava em primeiro lugar.

5. Evolução dos computadores

A evolução do poder computacional está passando por uma quebra de paradigma. Até então a luta dos pesquisadores era conseguir transistores cada vez menores que pudessem garantir mais ciclos de processamento. Este aumento do número de ciclos é sustentado pela lei de Moore (MOORE, 1965), que indica que a cada dois anos a quantidade de transistores que pode ser colocada no mesmo circuito por baixo custo irá dobrar. Esta lei se mantém precisa por muito tempo e alguns afirmam que assim continuará por mais algumas décadas (KANELLOS, 2005).

Porém, segundo Sutter (2005), para dissipar melhor o uso da unidade de processamento e para que seja possível criar paralelismo de instruções, os fabricantes preferem dividir os transistores em mais núcleos de processamento do que investir todos eles em apenas um. Atualmente, é mais barato e eficiente ter mais de um processador para realizar a mesma tarefa do que continuar tentando conseguir espaço para mais transistores.

UDESC

6. Futuros aplicativos

Para que os aplicativos continuem ganhando desempenho gratuito com a melhoria de equipamentos, é necessário que eles sejam construídos preparados para os novos conceitos de multiprocessamento.

Garantindo que boa parte da computação possa ser efetuada em paralelo, também está assegurado o aumento de desempenho apenas com a troca do equipamento em que o código é executado. Esta melhoria é sustentada pela lei de Amdahl (equação 1), que é utilizada para encontrar o máximo esperado de melhoria no desempenho de um sistema em que apenas parte dele foi melhorada (AMDAHL, 1967). Em um sistema com uma fração P que pode ser executada em paralelo rodando em N núcleos, a fórmula para obter a melhoria esperada (M) é expressa com a seguinte equação:

$$M = \frac{1}{(1 - P) + (\frac{P}{N})}$$

Equação 1 – Lei de Amdahl

O problema atual, conforme será apresentado na seção 8 é que “garantir que boa parte da computação possa ser paralelizada” implica em grandes alterações durante a programação de um sistema. Estes conceitos poderiam ser abstraídos se o compilador conseguisse extrair

informações suficientes da própria linguagem para saber o que pode ser paralelizado e o que não pode. Pode parecer utopia repassar estas responsabilidades ao compilador, mas veremos no capítulo 16 que isto, além de ser possível, já está feito e pronto para uso na linguagem Haskell.

Por outro lado, nem tudo pode ser paralelizado. Existem computações e sistemas que são concorrentes por natureza e precisam ser assim. O trabalho para preparar um algoritmo para um ambiente concorrente no modelo atual é idêntico para preparar computações paralelas, pois, como será apresentado nas seções 7 e 10, no modelo atual não existe esta distinção. A linguagem Haskell oferece memória transacional de software para amenizar boa parte dos problemas de algoritmos concorrentes. A seção 15 apresenta este conceito e a seção 18 descreve como ele é aplicado em Haskell.

7. **Paralelismo e concorrência**

Para Marlow (2009, 2011), existe uma diferença importante entre paralelismo e concorrência. Estas palavras são normalmente utilizadas para se referir ao mesmo conceito, pois em uma linguagem orientada a objetos, onde não há uma distinção clara entre código puro e código com efeitos colaterais, sempre se está em um ambiente com concorrência. Em linguagens que fazem esta distinção, é possível obter, de fato, paralelismo determinístico.

Paralelismo é a utilização de processamentos independentes em paralelo. Estes processos não se comunicam e nem se interferem. O resultado final de cada processamento paralelo sempre será o mesmo, independente da ordem em que as operações forem executadas pelos processadores. Desta forma, o único objetivo de se paralelizar uma computação é chegar mais rápido ao resultado final.

Por outro lado, a concorrência se dá quando mais de um processo compete com outro para chegar a um resultado. Faz-se necessária a utilização de um mecanismo de sincronismo entre eles para que tenham uma comunicação saudável entre si. Neste tipo de processo, o resultado pode mudar dependendo da ordem em que as instruções forem executadas, resultando em uma computação não determinística. A concorrência é utilizada, por exemplo, para permitir maior responsividade em uma aplicação com interface gráfica, o processo que responde às ações do usuário compete com o processo que executa os cálculos em *background*, mantendo a interface gráfica responsiva mesmo em momentos de processamento intenso.

É possível criar processos concorrentes em um único núcleo de processamento, intercalando as instruções. Por outro lado, esta técnica não ajuda um processo a chegar mais rápido no seu resultado, portanto não serve para algoritmos paralelos, que precisam de uma arquitetura multi-núcleos para atingirem seu objetivo.

8. **Cenário atual**

Java suporta concorrência de forma que vários processos compartilhem a mesma área de memória. A preparação de um algoritmo para suportar este tipo de processo não é suave. É necessário muito código e atenção ao fazer um algoritmo para funcionar em um ambiente concorrente. Este processo é sensivelmente sujeito a erros (JONES, 2007).

Martin (2009) reserva um capítulo inteiro sobre concorrência em seu livro “Clean Code” e começa ele com a seguinte frase: “Escrever programas concorrentes é difícil -- muito difícil.”
As próximas sessões (9 e 10) irão demonstrar os conceitos de concorrência e paralelismo em Java.

9. **Concorrência em Java**

Para se construir um componente que funcione em ambiente concorrente, o tradicional é utilizar *locks* e variáveis de condição para garantir que um objeto não fique em um estado inconsistente. O problema deste tipo de solução é que não é possível utilizar estas construções em uma composição e o uso de *locks* não faz bom proveito dos equipamentos com mais de um núcleo.

Por exemplo, uma aplicação financeira que é utilizada de forma concorrente por vários clientes precisa garantir que o saldo total de duas contas se mantém constante mesmo durante uma transferência de dinheiro. Uma programação sem esta preocupação pode levar à solução mostrada no quadro 1. Porém, em um ambiente concorrente, entre a diminuição do saldo da conta de origem e a adição do saldo na conta de destino, o valor a ser transferido não está em conta nenhuma, gerando um ponto de inconsistência nos dados da aplicação. Dependendo da ordenação das *threads* de execução, situações ainda piores podem acontecer: a tabela 1 mostra um cenário em que um saldo sai de uma conta e não fica em nenhuma outra.

```
class Conta {
  private double saldo;
  public double getSaldo() { return saldo; }
  public void setSaldo(double novoSaldo) { saldo = novoSaldo; }
}
class Banco {
  public void transferir(double valor, Conta origem, Conta destino) {
    double novoSaldoOrigem = origem.getSaldo() - valor;
    double novoSaldoDestino = destino.getSaldo() + valor;
    origem.setSaldo(novoSaldoOrigem);
    destino.setSaldo(novoSaldoDestino);
  }
}
```

Quadro 1 – Transferência bancária insegura

Seq.	*Thread A*	*Thread B*	*Saldo A*	*Saldo B*	*Total*
0	transferir(100, a, b)	transferir(50, b, a)	**500**	**300**	**800**
1	a.getSaldo() → 500		500	300	800
2	novoSaldoOrigem = 400		500	300	800
3		b.getSaldo() → 300	500	300	800
4		novoSaldoOrigem = 250	500	300	800
5		a.getSaldo() → 500	500	300	800
6		novoSaldoDestino = 550	500	300	800
7		b.setSaldo(250)	500	**250**	**750**
8	b.getSaldo() → 250		500	250	750
9	novoSaldoDestino = 350		500	250	750
10		a.setSaldo(550);	**550**	250	**800**
11	a.setSaldo(400);		**400**	250	**650**
12	b.setSaldo(350);		400	**350**	**750**

Tabela 1 – Ordenação de Threads causando problemas de concorrência

O uso de *locks* evita este tipo de problema, pois apenas uma *thread* poderá alterar o saldo de uma conta. Entretanto, um estudo detalhado do modelo de execução dos processos concorrentes é necessário para evitar situações de *deadlock*. O quadro 2 mostra um código que usa *lock* e tenta

evitar *deadlocks* através da ordenação dos *locks*, a função *ordenacaoLock* é responsável por ditar esta ordem, o código adicionado em relação ao quadro 1 está com enfâse.

```
public void transferir(double valor, Conta origem, Conta destino) {
  Conta a, b;
  if (ordenacaoLock(origem, destino)) {
    a = origem;   b = destino;
  } else {
    a = destino;  b = origem;
  }
  synchronized(a) { synchronized(b) {
    double novoSaldoOrigem = origem.getSaldo() - valor;
    double novoSaldoDestino = destino.getSaldo() + valor;
    origem.setSaldo(novoSaldoOrigem);
    destino.setSaldo(novoSaldoDestino);
  } }
}
```

Quadro 2 – Transferência bancária utilizando locks

A composição de algoritmos que usam *locks* se torna difícil, pois o cliente do código precisa saber quais *locks* o código chamado irá adquirir para evitar que eles sejam adquiridos em uma ordem incorreta, quebrando o encapsulamento do código. O quadro 3 mostra um código que tenta reutilizar a função do quadro 2 em uma composição.

```
public void transferirContaConjunta(double valor, Conta origemA, Conta origemB,
                                    Conta destino) {
  Conta a, b, c;
  if (ordenacaoLock(origemA, origemB)) {
    if (ordenacaoLock(origemB, destino)) {
      a = origemA;   b = origemB;   c = destino;
    } else if (ordenacaoLock(origemA, destino)) {
      a = origemA;   b = destino;   c = origemB;
    } else {
      a = destino;   b = origemA;   c = origemB;
    }
  } else {
    if (ordenacaoLock(origemA, destino)) {
      a = origemB;   b = origemA;   c = destino;
    } else if (ordenacaoLock(origemB, destino)) {
      a = origemB;   b = destino;   c = origemA;
    } else {
      a = destino;   b = origemB;   c = origemA;
    }
  }
  synchronized(a) { synchronized(b) { synchronized(c) {
    if (origemA.getSaldo() < saldo) {
      transferir(saldo, origemB, destino);
    } else {
      transferir(saldo, origemA, destino);
    }
  } } }
```

```
}
```

Quadro 3 – Transferência bancária utilizando locks

Jones (2007) afirma: "a tecnologia hoje dominante para programação concorrente – *locks* e variáveis de condição – é fundamentalmente falha." Ele detalha alguns dos problemas relacionados a este tipo de solução:

> **Pegando poucos locks.** É fácil esquecer-se de adquirir um lock e acabar com duas threads
> alterando a mesma memória simultaneamente.
> **Pegando muitos locks.** É fácil adquirir muitos locks e inibir a concorrência (no melhor dos casos) ou causar um deadlock (no pior dos casos).
> **Pegando locks errados.** Na programação baseada em locks a conexão entre os locks e a informação protegida por eles normalmente existem na mente do programador, e não explicitamente no programa. Como resultado, é muito fácil adquirir os locks errados.
> **Pegando locks na ordem errada.** Na programação baseada em locks, o programador deve ser cuidadoso para pegar os locks na ordem "correta". Evitando o deadlock que de outra forma pode ocorrer é sempre cansativo e suscetível a erros, e algumas vezes extremamente difícil.
> **Recuperações de falhas** podem ser muito difíceis, porque o programador deve garantir que nenhum erro possa deixar o sistema em um estado inconsistente, ou em um estado que locks são segurados indefinidamente.
> **Sinais perdidos e tentativas desperdiçadas.** É fácil esquecer-se de avisar uma variável de condição que uma thread está esperando, ou de re-testar a condição após receber o sinal.
> (JONES, 2007)

10. Paralelismo em Java

A partir da versão 7 do Java, é oferecido o *framework* Fork/Join para atender computações paralelas.

A documentação da linguagem (ORACLE, 2011) fornece um exemplo de utilização deste *framework* para ordenar uma lista que faz uso de memória compartilhada, mostrado no quadro 4. Ainda é possível que um processo interfira no outro através desta área compartilhada, sendo assim, conforme explicado na seção 7, o conceito de paralelismo utilizado pela linguagem Java é essencialmente concorrente, pois não é possível assegurar completo isolamento entre dois processos de uma mesma aplicação em Java.

```
class SortTask extends RecursiveAction {
  final long[] array; final int lo; final int hi;
  SortTask(long[] array, int lo, int hi) {
    this.array = array; this.lo = lo; this.hi = hi;
  }
  protected void compute() {
    if (hi - lo < THRESHOLD)
      sequentiallySort(array, lo, hi);
    else {
      int mid = (lo + hi) >>> 1;
      invokeAll(new SortTask(array, lo, mid),
                new SortTask(array, mid, hi));
      merge(array, lo, hi);
    }
  }
}
```

Quadro 4 – Utilização do framework Fork/Join em Java 7, enfâse na área de memória compartilhada

11. Por que sair do Java?

A máquina virtual Java deixou a linguagem C para trás por causa da ilusão de memória infinita que ela oferecia (CLICK, 2011). Da mesma maneira é possível supor que a linguagem Java poderá ser deixada para trás quando algo se tornar muito complexo para ser feito nesta linguagem e uma linguagem melhor[5] conceder a ilusão de um mundo perfeito.

Talvez Java não esteja preparado para as mudanças que a arquitetura de computadores está sofrendo e não seja bom o suficiente para manter a produtividade dos programadores neste novo universo. Se isto se confirmar, a linguagem que apresentar uma solução simples para estes problemas é uma ótima candidata a ser a próxima no topo do índice TIOBE.

12. Estruturas funcionais

Estruturas funcionais são estruturas de dados persistentes, i.e. depois de criadas, não sofrem alterações.[6] Cada "atualização" de uma estrutura persistente cria uma nova versão dela, mantendo a versão antiga disponível para consulta. Estas estruturas não permitem alterações destrutivas e por isto podem ser utilizadas em processos concorrentes sem nenhuma modificação, mesmo se forem compartilhadas entre processos.

Em uma linguagem em que este tipo de estrutura é utilizado por padrão, é garantido que nada será alterado, nem a estrutura e nem as estruturas que estiverem dentro dela. Segundo Okasaki (1999), isto permite o compartilhamento de referências em todos os lugares, torna desnecessária a realização de cópias inteiras da estrutura e garante um bom desempenho para a aplicação. Por serem seguras em ambiente concorrente por natureza, é possível paralelizar as cópias e "atualizações" que são necessárias. Mesmo com um custo assintótico maior, este paralelismo faz com que fiquem tão rápidas quanto às estruturas imperativas em equipamentos multi-núcleos.

13. Controlando o tempo

Para controlar o avanço de uma estrutura no tempo, é possível guardar uma referência para a última versão desta estrutura e conforme forem geradas novas versões dela, a referência é alterada para apontar para a nova versão. Hickey (2009) dá o nome de "identidade" a este conceito.

É possível avançar esta estrutura no tempo aplicando funções a ela. Estas funções recebem como parâmetro a estrutura atual e retornam a nova versão dela. Estas atualizações podem ser enfileiradas para criar uma linha do tempo da estrutura, conforme demonstrado na figura 1.

A identidade pode ser construída como um ponto mutável na memória encapsulado em operações c*ompare-and-swap* para garantir atomicidade das alterações.

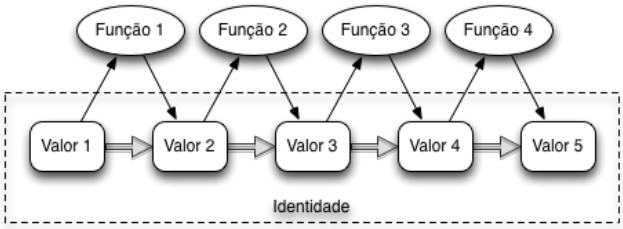

Figura 1 – Conceito de *identidade*

14. Observando o tempo

[5] a definição de "uma linguagem melhor" segue a linha de raciocínio de Graham (2003)
[6] diferente das estruturas imperativas que são efêmeras

Para observar mais de uma identidade no tempo, é possível fazê-lo de duas formas: realizar um "scan" ou retirar um "snapshot" do que queremos analisar (HICKEY, 2009).

Realizar um "scan" é olhar uma estrutura de cada vez. Como olhar um carro no inicio da estrada, olhar para o céu, e depois olhar para o final da estrada e ver o mesmo carro. É possível reproduzir este efeito apenas utilizando estruturas funcionais e identidades. Cada "olhar" é traduzido como uma leitura do valor contido em uma identidade.

O conceito de "snapshot" é o mesmo que tirar uma fotografia. Como realizada a captura de um quadro contendo a estrada e o horizonte, ao mesmo tempo em que se pode ver o carro na estrada, também é possível ver o céu. Mesmo que estes objetos tenham mudado após a fotografia, a análise é feita com base em todos os valores no mesmo momento temporal. É possível utilizar memória transacional de software para conseguir este efeito.

15. Memória transacional

Memória transacional adiciona à memória do computador alguns dos conceitos existentes nos bancos de dados. A idéia principal é que seja possível encapsular um bloco de código, incluindo chamadas aninhadas, em um bloco atômico, com a garantia de que serão respeitados todos os outros blocos atômicos no programa (HARRIS ET AL., 2008).

A utilização deste conceito durante a programação adiciona tanto valor a esta atividade que Bryan O'Sullivan afirma que trocar um ambiente de programação com memória transacional para um sem este conceito é como voltar à idade das trevas (O'SULLIVAN, 2011).

Para utilizar com segurança uma solução com memória transacional, os efeitos colaterais devem ser proibidos em uma transação, pois mudanças na memória podem ser revertidas, mas retroceder efeitos colaterais[7] é muito difícil e às vezes impossível. Isto gera problemas em linguagens que não permitem a separação destes conceitos, como na linguagem Java.

16. Conhecendo Haskell

Haskell é uma linguagem funcional pura com tipagem estática polimórfica inferida e de avaliação preguiçosa, bem diferente da maioria das outras linguagens (HASKELL WIKI, 2011). Ser uma linguagem funcional pura significa que todas as construções da linguagem giram em torno de funções que não possuem efeitos colaterais, i.e. que todas as entradas estão definidas no tipo da função e que possuem apenas um valor como saída.

Ter tipagem estática significa que os tipos de todos os valores são conhecidos durante a compilação, mas não significa que o programador precisa informar estes tipos durante a programação. A linguagem Haskell não obriga a declaração explícita dos tipos de cada valor, ela permite que estas informações sejam omitidas. O compilador se encarrega de inferir o tipo de cada valor. Programadores Haskell apenas fazem a declaração dos tipos nas funções como forma de documentação, mas ela é desnecessária na maioria dos casos. (LENTCZNER, 2011).

Tipagem polimórfica permite que as funções operem de maneira diferente dependendo dos tipos reais que são fornecidos a elas durante a execução do programa.

A linguagem é preguiçosa para realizar a avaliação de uma expressão, i.e. ela só irá realizar as computações necessárias para obter o valor de uma expressão quando o valor em questão for necessário para tomar alguma decisão sobre o programa.

17. Produzindo efeitos colaterais em Haskell

Haskell faz uso de um conceito chamado Monad para que seja possível simular efeitos colaterais dentro de um contexto computacional (HUDAK ET AL., 2000). A linguagem fornece um

[7] e.g. escrita em disco, impressão de documento, lançamento de um foguete

Monad mágico[8] chamado "IO" que permite efeitos colaterais no mundo real. Para que uma função possa interferir com o mundo exterior, o seu tipo precisa declarar isto. O quadro 5 mostra a diferença entre uma função com efeitos colaterais e uma sem.

```
somar :: Int -> Int -> Int
somar a b = a + b

imprimeESoma :: Int -> Int -> IO Int
imprimeESoma a b =
  do putStrLn ("O resultado da soma é: " ++ show c)
     return c
  where c = a + b
```

Quadro 5 – Diferença entre função pura e impura em Haskell

Uma vez dentro do Monad IO, não é possível escapar dele (JONES, 2001). Funções que não são do tipo IO não podem gerar efeitos colaterais e também não podem acessar funções que estejam neste contexto. Esta distinção no tipo das funções permite que a pureza do código seja verificada em tempo de compilação.

Chamar qualquer função pura é seguro de qualquer parte do programa e em qualquer ordem. Independentes de quantas vezes forem chamadas, estas funções sempre terão os mesmos valores para os mesmos argumentos e não corromperão algum estado do programa. Esta propriedade implica que todas estas funções são seguras para utilização em ambiente concorrente ou paralelo.

18. Concorrência em Haskell com memória transacional

O suporte à memória transacional vem por padrão no Haskell e para fazer uso dela basta utilizar o Monad STM[9] ao invés do Monad IO. Computações em um contexto STM são avaliadas quando aplicadas à função "atomically", que cria uma transação sobre o código no Monad STM e tenta aplicá-lo no Monad IO. Caso a transação falhe, ela será re-computada.[10] Isto permite composição, pois mesmo se um contexto STM estiver dentro de outro, todo o código será agrupado em um mesmo bloco atômico.

O quadro 6 demonstra a programação em Haskell utilizando STM para controle de conta corrente, contrastando com os exemplos anteriores escritos em Java.

```
data Conta = Conta { saldo :: TVar Double }

transferir :: Double -> Conta -> Conta -> STM ()
transferir saldo origem destino = do
  saldoOrigem <- readTVar (saldo origem)
  saldoDestino <- readTVar (saldo destino)
  writeTVar (saldo origem) (saldoOrigem – saldo)
  writeTVar (saldo destino) (saldoDestino + saldo)

transferirContaConjunta :: Double -> Conta -> Conta -> Conta -> STM ()
transferirContaConjunta saldo origemA origemB destino = do
  saldoOrigemA <- readTVar (saldo origemA)
```

[8] o único que pode fugir ao contexto computacional

[9] "STM" significa "Software Transactional Memory"

[10] a programação de memória transacional em Haskell sabe quais estruturas foram lidas durante a transação e só fará a computação novamente quando um destes valores for alterado, garantindo que não seja desperdiçado processamento

```
if saldoOrigemA < saldo
  then transferir origemB destino
  else transferir origemA destino
```

Quadro 6 – Programação Haskell com STM

19. Tolerância a falhas

O contexto STM fornece algumas funções para lidar com possíveis falhas durante a execução de uma transação. Permitindo que, por exemplo, seja controlado o saldo de uma conta corrente através deste conceito: marcando a transação como falha caso a conta corrente não possua saldo suficiente durante a retirada do valor. Indicar que a transação falhou significa que *todas* as alterações geradas pelo bloco atômico serão descartadas.

Este controle também permite composição, dentro de um mesmo bloco atômico podemos tentar um caminho de execução e caso a transação falhe durante este percurso, adotamos outro. Neste caso, somente as alterações causadas pelo caminho transacional falho serão descartadas, permitindo que os programas tentem determinada situação sem a preocupação de corrupção dos dados. O quadro 7 demonstra o código para transferência de saldo de conta corrente que falha ao encontrar uma conta sem saldo suficiente. A função *orElse* recebe duas transações STM e faz com que a primeira transação seja tentada e caso ela falhe (através da função *retry*), a segunda transação é executada, no caso de ambas falharem, o resultado como um todo é uma falha.

```
transferir saldo origem destino = do
  saldoOrigem <- readTVar (saldo origem)
  if saldoOrigem > saldo
    then do saldoDestino <- readTVar (saldo destino)
            writeTVar (saldo origem) (saldoOrigem - saldo)
            writeTVar (saldo destino) (saldoDestino + saldo)
    else retry

transferirContaConjunta saldo origemA origemB destino =
  transferir origemA destino `orElse` transferir origemB destino
```

Quadro 7 – Composição de transações em memória com tolerância a falhas em Haskell

20. Paralelismo em Haskell

Como Haskell faz distinção entre funções puras e funções que não são puras, é possível obter paralelismo determinístico. Para usufruir deste paralelismo, basta anotar os valores como candidatos a serem calculados em paralelo através da função *par*. Estas anotações adicionam a computação em um *pool* de candidatos a processamento paralelo. Em tempo de execução, os processos que estão parados furtam[11] estas computações e calculam o valor em paralelo.

A linguagem poderia fazer a computação de todos os valores puros em paralelo, mas isto geraria uma sobrecarga de processamento apenas para criar estas notificações paralelas. Também pelo fato de Haskell ter avaliação preguiçosa, o programa não sabe o que será necessário ou não de antemão.

Como a função *par* é pura e ela só pode ser aplicada sobre valores puros, é garantido que o funcionamento do programa não irá alterar em nada ao adicionar estas anotações, é possível experimentar em código em produção para ver quanto o desempenho da aplicação muda ao colocar valores para serem calculados em paralelo. O quadro 7 mostra a diferença entre uma

11 do inglês “work stealing”

função que calcula o valor de fibonacci em um único núcleo e uma que se aproveita de paralelismo em uma arquitetura multi-núcleos, com ênfase na diferença de código entre as duas funções.

```
fib :: Int -> Int
fib 0 = 0
fib 1 = 1
fib n = a + b
  where a = parFib (n - 1)
        b = parFib (n - 2)

parFib :: Int -> Int
parFib 0 = 0
parFib 1 = 1
parFib n = a `par` b `pseq` a + b
  where a = parFib (n - 1)
        b = parFib (n - 2)
```

Quadro 7 – Aproveitando paralelismo em arquiteturas multi-nucleos em Haskell

21. Considerações finais

As seções 5 e 6 constataram que a arquitetura de computadores está mudando e que é preciso adequar a maneira de produzir software para garantir que estas melhorias reflitam em melhor desempenho para as aplicações.

Na seção 8 foram apresentados possíveis problemas que o modelo atual (i.e. orientado a objetos) possui em relação a processamentos paralelos e concorrentes. A programação para usufruir dos equipamentos atuais através do uso de *locks* e variáveis de condição é difícil de programar de maneira correta e não permitem composição e reutilização dos componentes, pois eles carregam muitas informações sobre o ambiente em que executam (e.g. ordenação de *threads* e sincronismo).

Foi proposto como alternativa, na seção 16, a programação funcional em Haskell. Esta oferece algumas vantagens, como a utilização de memória transacional de software e paralelismo determinístico. A programação em Haskell permite uma maior abstração sobre os conceitos de multiprocessamento, permitindo uma expressividade maior sobre o que o programa se propõe a resolver, ao invés sobrecarregar o código com os detalhes técnicos envolvidos. Além de ser mais fácil de desenvolver uma solução desta forma, também auxilia na comunicação entre os programadores permitindo a troca de ideias em conceitos de alto nível.

O problema do modelo atual é conhecido e a programação funcional é uma das alternativas para tentar solucionar este problema. Pode ser que este se torne o próximo paradigma utilizado pelo mercado. Depende de quão bem estes conceitos forem recebidos pelos desenvolvedores e da adoção deste paradigma de uma forma mais abrangente. Todo o ecosistema atual que suporta a programação Java, como ferramentas de monitoramento e um vasto número de bibliotecas exercem um grande peso sobre decisões deste tipo. Os recentes avanços da linguagem Java para suportar modelagem funcional através de *lambdas* demonstra a preocupação de manter a linguagem aderente ao mercado e aos novos problemas que surgem.

Outras respostas para estes problemas podem aparecer ao longo do tempo. Esta ainda é uma área nova na computação, ela trouxe conceitos e preocupações novas para os programadores. Alguns conceitos já foram difundidos no mercado[12] e outros ainda precisam amadurecer.

[12] e.g. utilizar concorrência para melhorar a resposta da tela ao invés de um loop principal que captura e processa eventos

Independente da resposta que o mercado adotar como correta, aprender uma linguagem funcional e os conceitos que ela possui[13] trazem novas alternativas para soluções de problemas atuais. Estes conceitos podem ser utilizados para repensar uma solução que utilize o modelo atual de programação e que faça bom uso dos novos equipamentos.

Referências

AMDAHL, G. M. **Validity of the single processor approach to achieving large scale computing capabilities**. In Proceedings of the April 18-20, 1967, spring joint computer conference, AFIPS'67 (Spring), p. 483–485, New York, NY, USA, 1967. ACM.

CLICK, C. Apresentação: **A JVM does that?** Google Tech Talk. Março 2011. Disponível em: <http://www.youtube.com/watch?v=uL2D3qzHtqY> Acesso em: 3 out. 2011.

Computer Hope. **Computer history - 1940 - 1960**. 2011. Disponível em: <http://www.computerhope.com/history/194060.htm> Acesso em: 10 out. 2011.

GRAHAM, P. **Beating the averages**. Symposium A Quarterly Journal In Modern Foreign Literatures, v. 2003, p. 1–9, Abril 2003. Disponível em: <http://www.paulgraham.com/avg.html> Acesso em: 3 out. 2011.

HARRIS, T.; MARLOW, S.; JONES, S. P.; HERLIHY, M. **Composable memory transactions**. In Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming, PPoPP '05, p. 48–60, New York, NY, USA, 2008. ACM. ISBN 1-59593-080-9.

Haskell Wiki. **Wiki: Introduction**. 2011. Edição em: 29 out. 2011 22:36. Disponível em: <http://www.haskell.org/haskellwiki/Introduction> Acesso em: 24 nov. 2011.

HICKEY, R. Apresentação: **Are we there yet?** Sun 2009 JVM Language Summit. Novembro 2009. Disponível em: <http://www.infoq.com/presentations/Are-We-There-Yet-Rich-Hickey> Acesso em: 3 nov. 2011

HUDAK, P.; PETERSON, J.; FASEL, J. **A gentle introduction to Haskell - version 98**. Junho 2000. Disponível em: <http://www.haskell.org/tutorial/index.html> Acesso em: 6 out. 2011

JONES, S. P. **Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell**. In Engineering theories of software construction, p. 47–96. IOS Press, 2001.

JONES, S. P. **Beautiful code:** Leading programmers explain how they think, cap. Beautiful Concurrency. Beijing: O'Reilly Media, Junho 2007. ISBN 978-0-596-51004-6.

KANELLOS, M. **New life for Moore's law**. Abril 2005. Disponível em: <http://news.cnet.com/New-life-for-Moores-Law/2009-1006_3-5672485.html> Acesso em: 4 nov. 2011.

LENTCZNER, M. Apresentação: **Haskell amuse-bouche**. Google Tech Talk. Outubro 2011. Disponível em: <http://www.youtube.com/watch?v=b9FagOVqxmI> Acesso em: 2 nov. 2011.

[13] e.g. valores persistentes, composição de funções, funções de ordem superior, estruturas funcionais

MARLOW, S. **Parallelism /= concurrency**. Outubro 2009. Disponível em: <http://ghcmutterings.wordpress.com/2009/10/06/parallelism-concurrency> Acesso em: 3 out. 2011.

MARLOW, S. **Parallel and concurrent programming in Haskell**. Cambridge, U.K.: Microsoft Research Ltd., Setembro 2011.

MARTIN, R. C. **Clean code:** A handbook of agile software craftsmanship. Upper Saddle River, NJ: Prentice Hall. Person Education, Inc., Dezembro 2009. ISBN 0132350882.

MOORE, G. E. **Cramming more components onto integrated circuits**. Electronics, v. 38, Abril 1965. Disponível em: <ftp://download.intel.com/museum/Moores_Law/Articles-Press_Releases/Gordon_Moore_1965_Article.pdf> Acesso em: 24 nov. 2011.

OKASAKI, C. **Purely functional data structures**. Cambridge University Press, 1999. ISBN 0521663504.

Oracle. **Java™ Platform Standard**: Class RecursiveAction. 2011. Disponível em: <http://download.oracle.com/javase/7/docs/api/java/util/concurrent/RecursiveAction.html> Acesso em: 3 out. 2011.

O'SULLIVAN, B. Apresentação: **Running a startup on Haskell**. Strange Loop. Outubro 2011. Disponível em: <http://www.infoq.com/presentations/Running-a-Startup-on-Haskell> Acesso em: 1 nov. 2011.

PROVENZA, P. **Why are pointers so hard?** Julho 2011. Disponível em: <http://tenaciousc.com/?p=2922> Acesso em: 24 nov. 2011.

RITCHIE, D. M. **The development of the C language**. ACM SIGPLAN Notices, v. 28, p. 201–208, Março 1993. ISSN 0362-1340.

SUTTER, H. **The free lunch is over:** A fundamental turn toward concurrency in software. Dr. Dobb's Journal, v. 30, p. 202–210, 2005.

TIOBE Software. **TIOBE programming community index**. 2011. Disponível em: <http://www.tiobe.com/index.php/tiobe_index> Acesso em: 3 out. 2011.

UDESC