Relatório do Simparticle

Aline Marcelo Garlet Millani

18 de junho de 2013

Resumo

Este trabalho implementa um simulador de partículas elétricas e de corpos massivos em OCaml.

1 Descrição do problema

O problema consiste em, dado um conjunto de partículas, similar as forças de atração e repulsão entre elas, desenhando-as conforme se movem. É feita a simulação somente das partículas elétricas usando um algoritmo de complexidade n^2, embora a estrutura para o algoritmo de Barnes-Hut ($n \log n$) esteja implementada. Essa estrutura acabou não sendo usada por falta de tempo.

Para desenhar numa janela, usamos as bibliotecas lablgl e lablglut, que são, respectivamente, implementações de OpenGl e GLUT para OCaml.

2 OCaml

OCaml é uma linguagem multi-paradigma que apresenta características imperativas, funcionais e de orientação a objetos. Atualmente, é a principal implementação de Caml.

Possui características de ML, como tipagem estática, que permite uma maior confiabilidade no código por sabermos em tempo de compilação que não haverão operações não definidas, e inferência de tipos, permitindo que o programador não se preocupe com o tipo de cada variável e melhorando a redigibilidade e diminuindo o acoplamento por tipagem.

Outra propriedade muito interessante de OCaml é pattern matching, pois aumenta consideravelmente a expressividade da linguagem ao descrever construções if-else encadeadas através de uma sintaxe muito mais clara, além de permitir uma decomposição estrutural de forma mais natural.

Por ser uma linguagem compilada, com tipagem estática e uso de referências a memória, programas em OCaml são bastante eficientes, fazendo com que a linguagem possa ser usada em diversos casos práticos, diferentemente de algumas linguagens como Python e Java, que apresentam um custo em processamento ou memória proibitivos para diversas aplicações.

A orientação a objetos de OCaml permite o encapsulamento de informações ao fornecer para o programador meios de definir métodos e atributos privados ou públicos. Python não possui mecanismos para isso, de forma que convenções devem ser adotados. Por um lado, se tudo for público há maior flexibilidade,

mas o uso de uma classe pode se tornar mais obscuros por não sabermos quais alterações manterão a consistência dos dados.

3 Implementação

3.1 Definição e uso de classes

Foram criadas 4 classes: Body, Massive, Particle e Electric, sendo Body e Particle abstratas. Como o problema em si não apresenta uma noção de hierarquia e a linguagem possui características funcionais, o uso de classes complicou a implementação já que foi necessário criar classes inúteis, além de ter impedido o uso de pattern matching e gerado tipos diferentes, complicando a tipagem das funções.

Se houvesse uma noção de hierarquia no problema, o uso de classes teria sido útil.

```
class virtual body:
  float ->
  float ->
  object
    val mutable position : float * float
    val mutable velocity : float * float
    val mutable acceleration : float * float
    method getPosition : float*float
    method setPosition : float*float -> unit
    method getAcceleration : float*float
    method loopBound : float*float -> float*float -> unit
    method setAcceleration : float*float-> unit
    method applyAcceleration : float*float -> unit
    method virtual draw : unit
    method move : unit
  end;;
```

Figura 1: Interface da classe body

3.2 Encapsulamento e proteção dos atributos

Todos os atributos das classes são privados. O encapsulamento de dados pode ser visto através do método move, por exemplo, que esconde o fato de se estar usando três vetores (posição, velocidade e aceleração) para se calcular o movimento da partícula.

```
method move =
  let (px,py) = position in
  let (vx,vy) = velocity in
  let (ax,ay) = acceleration in
  position <- (px +. vx,py +. vy);
  velocity <- (vx +. ax,vy +. ay);
  acceleration <- (0.0,0.0)
```

Figura 2: Implementação do método move da classe body

3.3 Organização do código em espaços de nome diferenciados

Em OCaml, cada arquivo conta como um espaço diferente, de forma que este deve ser incluído com open ou prefixado com seu nome antes de ser possível usar suas funções. Isso é útil quando temos funções com o mesmo nome em mais de um arquivo, evitando conflitos de nome. Se a linguagem não suportasse espaços de nome, seria necessário prefixar o nome de cada função com o seu módulo.

3.4 Mecanismo de Herança

Foram usados três níveis de hierarquia, sendo que body é a superclasse de particle, que por sua vez é superclasse de electric.

```
class particle x y =
  object (self)
    inherit body x y
    method draw =
      GlDraw.begins `points;
      GlDraw.vertex2 position;
      GlDraw.ends ()
  end;;
```

Figura 3: Implementação da classe particle, mostrando a herança

3.5 Especificação de uma classe abstrata

A classe body é abstrata por possuir o método draw como virtual (ver Figura 1). O uso de métodos abstratos serve para melhor organizar a estrutura do código e permite um comportamento mais uniforme, isto é, garantir que operações iguais terão o mesmo nome. Se não usássemos métodos abstratos, poderíamos ter definido draw para partículas elétricas e show para massivas, o que complicaria o uso das classes.

3.6 Polimorfismo por inclusão

Usamos polimorfismo por inclusão para desenhar os corpos. Partículas elétricas possuem uma função de desenho diferente de partículas massivas. Essa técnica é útil para facilitar estender o código. Dessa forma, se quisermos desenhar outro tipo de corpo, podemos reutilizar essa função.

```
let rec drawBodies points = match points with
  | h::r -> h#draw;
            drawBodies r;
  | [] -> ()

let rec convertToBody lst = match lst with
  | h::r -> (h :> Body.body)::(convertToBody r)
  | [] -> []

(*
  atualiza o desenho
*)
let display dots circles drawTree ()=
  GlClear.color (0.0, 0.0, 0.0);
  GlClear.clear [ `color ];
  drawBodies (List.append (convertToBody !dots)
    (convertToBody !circles));
  if !drawTree then
    let dTree = Quadtree.buildQuadtree !dots in
    let cTree = Quadtree.buildQuadtree !circles in
    Quadtree.draw dTree;
    Quadtree.draw cTree;
  else
    ();
  Glut.swapBuffers ()
```

Figura 4: Funções para desenho de partículas

3.7 Polimorfismo paramétrico

Foi usada uma quadtree para a simulação das interações entre as partículas. A definição da estrutura é válida para qualquer tipo de dado, mas apenas um de cada vez, isto é, podemos ter uma árvore de inteiros, outra de partículas, etc... mas não podemos misturar floats e ints na mesma árvore.

Como OCaml usa inferência de tipos, o tipo da árvore não precisa estar explícito no código.

```
type 'a quadtree =
  | Empty
  | Leaf of float*(float*float)*'a
  | Node of float*(float*float)*
    ('a quadtree * 'a quadtree * 'a quadtree * 'a quadtree);;
```

Figura 5: Descrição da estrutura de uma quadtree. 'a indica o tipo parametrizado

Sem a possibilidade de usarmos polimorfismo paramétrico, teríamos ou que fazer uma árvore para cada tipo que será usado ou usar um tipo genérico (como, por exemplo, *void ** em C) e fazer cast para o tipo adequado. Por exemplo, em C, poderíamos usar a seguinte estrutura:

```c
struct quadtree
{
  char type;
  union data
  {
    struct quadtree *sections[4];
    void *leaf;
  }
  float value;
  float position[2];
};
```

Figura 6: Estrutura similar em C

Pode-se ver que a estrutura em C requer maior controle por parte do programador e é mais suscetível a erros.

3.8 Polimorfismo por sobrecarga

OCaml não suporta polimorfismo por sobrecarga. Isso está relacionado com o mecanismo de inferência de tipos, que seria muito mais complicado de se implementar (talvez até impossível para qualquer caso). No entanto, a linguagem permite a criação de funções que recebem parâmetros de tipos arbitrários (como, por exemplo, map), mas a implementação é a mesma para todos os tipos.

3.9 Especificação e uso de funções como elementos de primeira ordem

Para carregar as partículas de um arquivo, foi feita uma função para ler as linhas de um canal, que foi passada como argumento para a função parseChannel.

```
let readLine chan =
  let rec readLineAux acc =
    let c =
      try
        input_char chan
      with
        End_of_file ->
        if String.length acc = 0 then
          raise End_of_file
        else '\n'
    in
    match c with
     '\n' -> acc
    |'\r' -> acc
    |' ' ->
      if String.length acc = 0 then readLineAux acc else acc
    |'\t' ->
      if String.length acc = 0 then readLineAux acc else acc
    | _ -> (readLineAux (acc ^ (String.make 1 c)) )
  in
    readLineAux ""

let loadConfig filename categoryFunc =
  let chan = open_in_bin filename in
  let lines = parseChannel chan readLine in
  parseList categoryFunc lines
```

Figura 7: Função que lê uma linha de um arquivo e outra que carrega o arquivo de configuração

Sem funções de ordem maior, poderíamos obter o mesmo comportamento usando delegates ou usando classes e fazendo com que readLine fosse um método que pudesse ser sobrescrito. No entanto, a abordagem funcional é muito mais simples do que essas alternativas.

3.10 Especificação e uso de funções de ordem maior

Foi feita uma função que lê o conteúdo de um arquivo, sendo que um de seus parâmetros é uma função que recebe um canal e retorna um elemento.

```
let rec parseChannel chan readElement =
  try
    let el = readElement chan in
    el::(parseChannel chan readElement)
  with
    End_of_file -> []
```

Figura 8: Função que lê o conteúdo de um arquivo

As vantagens dessa técnica é descrita na Subseção 3.9

3.10.1 Currying

Foi usado currying para as funções de callback da glut. Como estas recebem somente informações da glut, sem ser possível passar dados do programa, é necessário usar variáveis globais para que elas sejam úteis. O uso de currying permitiu que as variáveis se tornassem locais, pois podíamos definir a função como quiséssemos, contanto que ao passarmos para a glut ela fosse do tipo esperado.

```
let rec timerF mili dots ~value =
  Physics.move !dots;
  Physics.simulateElectricNaive !dots;
  Glut.timerFunc ~ms:mili ~cb:(timerF mili dots) ~value:1;
  Glut.postRedisplay ()

let _ =

  let dots = ref [] in
  let circles = ref [] in
  let drawTree = ref true in
  let (elec,massive) =  Loader.loadBodies "config" in
  dots := elec;
  circles := massive;

  ignore (Glut.init Sys.argv);
  Glut.initDisplayMode ~double_buffer:true ();
  ignore (Glut.createWindow ~title:"Simparticle");

  Glut.displayFunc ~cb:(display dots circles drawTree);
  Glut.reshapeFunc ~cb:reshape;
  Glut.mouseFunc ~cb:(mouseHandler dots);
  Glut.keyboardFunc ~cb:(keyhandler drawTree);
  (*Glut.idleFunc ~cb:(Some idle);*)
  Glut.timerFunc ~ms:mili ~cb:(timerF mili dots) ~value:1;
  Glut.mainLoop()
```

Figura 9: Funções display e main do programa

Sem currying, só é possível usar as funções de callback da glut com variáveis globais.

3.11 Pattern Matching

Diversas funções feitas usaram pattern matching. A versão sem pattern matching das mesmas seria mais longa e menos legível. No exemplo abaixo, seria necessário fazer ifs possivelmente longos ou criar uma máquina de estados para se obter o mesmo comportamento.

```
let readElectricParticle lst =
  let rec readAux x y charge lst = match lst with
    |"x"::"="::value::rest ->
      readAux (fun k -> float_of_string value) y charge rest
    |"y"::"="::value::rest ->
      readAux x (fun k -> float_of_string value) charge rest
    |"charge"::"="::value::rest ->
      readAux x y (fun k -> float_of_string value) rest
    |"end"::r -> (Electric ((x 0),(y 0),(charge 0)) , r)
    | _ -> (Electric ((x 0),(y 0),(charge 0)) , lst)
  in
    let none = (fun x -> raise Invalid_format) in
    readAux none none none lst
```

Figura 10: Função que usa pattern matching

Para uma análise mais detalhada desta função, ver Figura 15.

3.12 Recursão como mecanismo de iteração

```
let rec drawBodies points = match points with
  | h::r -> h#draw;
            drawBodies r;
  | [] -> ()
```

Figura 11: Função para desenhar uma lista de pontos na tela

Em Python, a solução seria:

```
def drawBodies(bodies):
  for b in bodies:
    b.draw()
```

Figura 12: Equivalente para python

3.13 Delegates

Como OCaml suporta funções como elementos de primeira ordem, delegates se tornam completamente desnecessários, uma vez que são apenas uma forma de se passar métodos como argumento para alguma classe em paradigmas orientados a objeto. Como é possível passar a função diretamente, OCaml não possui sintaxe para delegates.

3.14 Uso de lista para manipulação de estruturas em funções de ordem maior

```
let rec changeValue f lst = match lst with
  | Electric (x,y,charge)::rest ->
    (Electric (x,y,f charge))::(changeValue f rest)
  | Massive (x,y,mass)::rest ->
    (Massive (x,y,f mass))::(changeValue f rest)
  | _ -> []

let loadBodies filename =
  let data = loadConfig filename readBodyCategory in
  constructBodies (changeValue (fun x -> x*.1.1) data)
```

Figura 13: Função para carregar os corpos e outra para alterar os valores carregados

4 Análise da Linguagem

característica	nota
simplicidade	8/10
ortogonalidade	5/10
expressividade	8/10
estruturas de controle	10/10
mecanismos de definição de tipos	10/10
abstração de dados e processos	10/10
modelo de tipos	8/10
portabilidade	7/10
reusabilidade	8/10
suporte e documentação	4/10
tamanho do código	9/10
generalidade	9/10
eficiência e custo	10/10

Tabela 1: Notas das características da linguagem segundo a opinião do grupo

4.1 Simplicidade

Graças ao sistema de inferência de tipos e ao pattern matching, a declaração de funções e o uso de tipos de dados se torna bastante simples para o programador. Além disso, estruturas sintaxes como let podem ser usadas tanto para declarar variáveis como para funções e podem ser usadas em qualquer ponto do código.

No entanto, a necessidade de se usar operadores aritméticos diferentes para tipos diferentes, o uso de let rec para funções recursivas e a ausência de coerção implícita torna a linguagem mais complicada de se usar em alguns casos como, por exemplo, ao misturarmos números inteiros e flutuantes na mesma expressão.

4.2 Ortogonalidade

O pattern matching permite tratar diversos tipos de dados de uma forma similar e bastante expressiva, o que aumenta a ortogonalidade da linguagem. Por outro lado, a necessidade de usarmos operadores diferentes para soma de inteiros e de floats baixa consideravelmente essa característica.

No trecho abaixo, foi possível tratar diversos construtores diferentes da mesma maneira:

```
let treeCenter a b c d =
  let rec avgAux lst amount accx accy = match lst with
    | Empty::rest -> avgAux rest amount accx accy
    | Leaf(_,(x,y),_)::rest ->
      avgAux rest (amount + 1) (accx +. x) (accy +. y)
    | Node(_,(x,y),_)::rest ->
      avgAux rest (amount + 1) (accx +. x) (accy +. y)
    | _ ->
      let fAmount = float_of_int amount in
      (accx /. fAmount, accy /. fAmount)
  in
  avgAux (a::b::c::[d]) 0 0.0 0.0
```

Figura 14: Função que calcula o centro de massa de 4 corpos usada para gerar a Quadtree

4.3 Expressividade

Novamente por causa de pattern matching e inferência de tipos, a expressividade da linguagem é bastante boa.

Na seguinte função, podemos notar diversos aspectos da linguagem:

```
let readElectricParticle lst =
  let rec readAux x y charge lst = match lst with
     "x"::"="::value::rest ->
      readAux (fun k -> float_of_string value) y charge rest
    |"y"::"="::value::rest ->
      readAux x (fun k -> float_of_string value) charge rest
    |"charge"::"="::value::rest ->
      readAux x y (fun k -> float_of_string value) rest
    | h::r -> (new electric (x 0) (y 0) (charge 0) , r)
    | _ -> (new electric (x 0) (y 0) (charge 0) , lst)
  in
    let none = (fun x -> raise Invalid_format) in
    readAux none none none lst
```

Figura 15: Função que lê os dados de uma partícula elétrica

Sem pattern matching, seria necessário fazer diversos if-elses, várias comparações de strings e percorrer a lista manualmente. A outra opção seria fazer um autômato, que também requereria mais código.

O uso de funções lambda permitiu que uma exceção fosse gerada somente nos casos em que um dos atributos não fosse inicializado e sem ser necessário usar variáveis auxiliares nem testar o valor dos atributos até então.

Podemos ver um caso de recursão como método de iteração, pois estamos percorrendo a lista de palavras do início até acharmos um "end". Graças ao pattern matching, a versão recursiva é mais expressiva que a iterativa por poder consumir diversos elementos de uma só vez sem explicitamente alterar o índice.

Se a linguagem usasse coerção implícita de string para float, o código ficaria mais simples pois os casts explícitos desapareceriam.

4.4 Estruturas de Controle

OCaml conta com os mecanismos de controle de linguagens imperativas, map de linguagens funcionais, um if-then-else que retorna um valor e pattern matching. Praticamente qualquer método de controle de fluxo que conhecemos pode ser usado em OCaml.

4.5 Mecanismos de Definição de Tipos

A linguagem permite a declaração de classes no estilo orientado a objetos, além de um meio de definir estruturas que possuem diversos construtores e registros com campos nomeados.

Eis um código onde definidos os três tipos de dados:

```
type 'a quadtree =
  | Empty
  | Leaf of float*(float*float)*'a
  | Node of float*(float*float)*
    ('a quadtree * 'a quadtree * 'a quadtree * 'a quadtree);;

type 'a registro = {codigo : int ; valor : 'a};;

class virtual particle x y =
  object (self)
    inherit body x y
    val mutable dotColor = (1.0, 0.7, 0.2)
    method draw =
      GlDraw.begins `points;
      GlDraw.color dotColor;
      GlDraw.vertex2 position;
      GlDraw.ends ()
  end;;
```

Figura 16: Exemplos de classes, estruturas e registros

4.6 Abstração de Dados e Processos

A expressão let da linguagem permite nomear qualquer expressão válida. Junto com funções lambda e currying, isso permite uma alta abstração de processos, pois podemos encapsular uma função dentro de uma expressão lambda para gerar uma nova função com comportamento específico, por exemplo.

4.7 Modelo de Tipos

OCaml usa tipagem forte, estática e inferida durante a compilação. Isso dá segurança e confiabilidade pois sabemos que as operações estarão definidas sempre e não haverá uma troca de tipo sem que o programador perceba. Linguagem com esse modelo de tipo são conhecidas por serem mais confiáveis e, se o programa compila, há uma boa chance de ele funcionar.

Linguagens com tipagem forte e estática geralmente sofrem o problema de o programador precisar anotar os tipos em muitos pontos. No entanto, a inferência de tipos faz com que isso não seja mais um problema.

4.8 Portabilidade

Embora a linguagem em si seja portável, é necessário usar o cygwin para compilar algum programa OCaml em plataformas Windows. No entanto, isso é mais uma restrição pela existência de ferramentas e não pela linguagem em si, isto é, um código OCaml feito para Linux deve funcionar perfeitamente bem em qualquer outra plataforma desde que existe um compilador para esta.

4.9 Reusabilidade

A linguagem permite dividir o código em módulos e estes podem ser usados facilmente a partir de outros arquivos. Como as funções são naturalmente polimórficas (usando tipos genéricos), o potencial de reúso é bastante grande. Também possui orientação a objetos, sendo possível estender classes caso a biblioteca não forneça algo completo o suficiente para a aplicação.

No entanto, a solução para dependências circulares não é simples, o que dificulta a criação de bibliotecas maiores e, consequentemente, o reúso das mesmas.

4.10 Suporte e Documentação

Existe pouca documentação sobre a linguagem e suas bibliotecas. Em particular, foi necessário olhar as headers da biblioteca lablgl para se saber como usá-la e quais funções existiam pois não havia algum lugar listando-as e explicando-as.

Nós encontramos a referência para as funções da biblioteca padrão de OCaml, mas para questões de sintaxe como o uso de classes e a declaração de tipos para funções e para aspectos como compilação a informação era mais escassa, sendo necessário fazer tentativa e erro em alguns casos.

4.11 Tamanho do Código

Graças a inferência de tipos, não é necessário anotar quais são os tipos das variáveis, reduzindo o tamanho do código. Como a linguagem é fortemente tipada, não é necessário fazer verificação de tipos em execução.

O pattern matching consegue reduzir consideravelmente a quantidade necessária de código para se fazer algo.

Finalmente, características funcionais como funções lambda, currying e a facilidade de se definir funções locais fazem com que o código não precise de linhas "inúteis" com atribuições, declarações e variáveis temporárias. Um exemplo de

código pequeno em OCaml e que seria maior em outras linguagens encontra-se na Figura 15.

4.12 Generalidade

Por causa de sua eficiência, OCaml pode ser usada para muitas aplicações, mesmo as mais pesadas ou críticas. E como é bastante expressiva, pode também ser usada para escrever aplicações pequenas rapidamente, ainda mais por possuir um interpretador além do compilador, de forma que não é necessário gerar o binário e os objetos intermediários para se executar o programa.

Como a linguagem não permite um controle tão preciso de ponteiros como C, não seria muito adequada para um sistema operacional. Por outro lado, é possível usar bibliotecas escritas e C junto com OCaml, viabilizando a criação de projetos híbridos.

4.13 Eficiência e Custo

OCaml é uma linguagem compilada com tipagem forte e estática. Além disso, permite o uso de referências a memória. O compilador é bom o suficiente para produzir código com eficiência próxima de linguagens como C++.

5 Conclusões

A primeira dificuldade que encontramos foi aprender a linguagem, pois nunca havíamos usado OCaml antes e a documentação dela não é muito abundante. Felizmente a linguagem é bastante parecida com ML e Haskell, fazendo com que o estilo da sintaxe em si não parecesse estranho, mas detalhes foram um pouco complicados de se encontrar.

O fato de a linguagem ser funcional facilitou bastante o uso da Quadtree por ser um estrutura naturalmente recursiva. Junto com pattern matching, cálculo lambda e a existência de múltiplos construtores para a mesma estrutura, essa função se tornou bem mais simples do que o equivalente em outras linguagens.

A leitura do arquivo para carregar os corpos também foi bastante simples graças ao pattern matching e aos tipos estruturados, pois foi possível separar em casos simples e concisos algo que seria transformado em um autômato um tanto grande para fazer o parsing.

Sobre a linguagem em si, achamos que é bastante boa por conseguir uma sintaxe expressiva, segurança de tipos e uma boa eficiência. No entanto, a falta de documentação unida com algumas dificuldades no processo de compilação nos fazem preferir Haskell a OCaml.

Referências

[1] http://en.wikipedia.org/wiki/Barnes-Hut_simulation

[2] http://caml.inria.fr/pub/docs/manual-ocaml/libref/