Programação Funcional — Apontamentos

(Versão 1.16)

Pedro Quaresma de Almeida[1]

2 de Outubro de 2008

[1]Departamento de Matemática da Universidade de Coimbra.

Conteúdo

Capítulo 1

Introdução

1.1 Diferentes Metodologias

No prosseguimento do estudo da construção de programas correctos que sejam soluções de problemas típicos das ciências da computação, um factor, que frequentemente é apontado como causa das dificuldades nesta construção, é o próprio paradigma da programação sequencial imperativa.

As características da programação sequencial estão, conceptualmente, muito distantes das noções normalmente usadas na descrição dos problemas. Estas requerem abordagens formais para poderem ser descritas de um modo preciso e, normalmente têm uma natureza estática: isto é, o problema é o mesmo independentemente do instante de tempo em que é observado (existem, obviamente, excepções).

Por seu lado, a programação imperativa não é nada formal, muito pouco precisa e é dinâmica no sentido em que a dimensão tempo procura substituir a dimensão da complexidade: um programa grande é decomposto em problemas simples que são resolvidos um de cada vez, sequencialmente no tempo.

Daqui resulta que a resolução de problemas seria bastante mais simples se a metodologia de construção de programas seguisse de perto a metodologia de descrição dos problemas. Por isso será conveniente programar de um modo: formal, preciso, e em que a estrutura e complexidade do problema se manifeste directamente na estrutura e complexidade do programa.

Algumas metodologias de descrição de problemas que têm vindo a ganhar importância crescente são:

- Modelos, baseados na *teoria dos conjuntos*, para as estruturas de informação e *descrição funcional das operações* sobre essas estruturas.
- *Lógica de primeira ordem* sob várias formas particulares.
- *Regras de simplificação ou substituição*.
- *Teorias algébricas axiomáticas* (a chamada teoria dos tipos abstractos de dados);
- Teoria dos *objectos* e das *comunicações entre objectos* [10].

Existem outras metodologias, designadamente ligadas à noção de processo e comunicação entre processos, mas, como envolvem directamente o tempo como componente intrínseca do problema, não serão tratadas neste curso.

A cada uma das metodologias atrás referenciadas está associada uma (ou mais) metodologias de programação que, em maior ou menor grau, satisfazem as condições já referidas.

A primeira destas metodologias deu origem à metodologia formal mais antiga: a chamada *programação funcional*. Existem imensas abordagens distintas a esta metodologia, muitas delas, porém, muito longe de formais e precisas. Duas das que melhor satisfazem as nossas condições, centram-se nas linguagens *ML* [11, 14] e *Haskell* [3, 13, 7, 6].

A lógica de primeira ordem foi sempre um veículo para a criação de metodologias de descrição de problemas: uma das primeiras áreas abordadas pelas metodologias formais foi a criação de demonstradores de teoremas.

No entanto a sua complexidade e a indecidibilidade da lógica de primeira ordem não permitia a criação de sistemas computacionais que reproduzissem toda a teoria. Assim procurou-se encontrar um fragmento da lógica de primeira ordem que fosse decidível e bem adaptado à construção de linguagens de programação, a lógica das cláusulas de Horn é um desses fragmentos sendo a base da linguagem de programação em lógica *Prolog* [8].

A terceira metodologia está ligada à demonstração de teoremas por simplificação sistemática de formulas lógicas mas também, à descrição de problemas através de regras de transformação.

Este tipo de descrição de problemas deu origem a sistemas computacionais que têm o nome genérico de *sistemas de reescrita*. Um desses sistemas chama-se *ERIL*.

Os tipos abstractos de dados são uma metodologia básica em ciências da computação. A família de linguagens descendentes da linguagem OBJ constituem uma família de linguagens que implementam estes conceitos, temos nomeadamente as linguagens *OBJ3* [5], *CafeOBJ* [4], e *Maude*.

Em relação às metodologias que usam objectos como conceito base tem-se que, infelizmente não existem sistemas computacionais que assentem sobre uma metodologia formal e precisa. Inversamente as (poucas) descrições precisas da noção de objecto não têm suporte em qualquer sistema computacional que seja acessível. Metodologias de objectos com descrição formal mas sem suporte computacional são, por exemplo, *OBLOG* e *FOOPS*. A situação inversa tem muitos mais representantes: *Smalltalk*, *Actor*, *C++*, *Java*, etc.

1.2 Metodologia de Programação Formal

Programas e problemas são, normalmente, sistemas grandes e complexos. A capacidade humana para entender tais sistemas é limitada e só consegue ter sucesso quando consegue aplicar uma de duas estratégias: a abstracção e a modularidade.

A abstracção é a capacidade para agrupar um número elevado de casos concretos diferentes (mas semelhantes) numa única entidade a que podemos dar o nome de *classe*, isto é, as propriedades comuns a todos os casos concretos de modo a que, face a essas propriedades, seja possível:

- prever os efeitos das transformações a que, eventualmente, estejam sujeitos os diferentes casos dentro da classe;
- validar frases lógicas que se apliquem universalmente a todos os casos da classe.

A estratégia de abstracção baseia-se em três técnicas fundamentais:

Instanciação Dada uma classe criar um dos seus casos (uma instância da classe);

Abstracção Dados dois ou mais casos concretos, caracterizar a "menor classe" que os contém como instâncias;

Validação Dada uma classe e um caso concreto verificar se o caso é uma instância da classe
ou
Prever os efeitos de transformações em instâncias de classe
ou
Validar frases lógicas quantificadas universalmente às instâncias da classe.

A modularidade é a capacidade para decompor casos complexos numa colecção de casos mais simples (chamados módulos) e em regras de composição de tal modo que:

- A análise (ou síntese) de qualquer um dos módulos seja independente da análise (ou síntese) dos restantes. De preferência cada módulo deve ser uma instância de uma classe de módulos bem conhecida.
- É possível construir o significado do caso complexo por aplicação das regras de composição ao significado dos módulos. Analogamente, se o problema for de síntese, as regras de composição devem permitir construir o sistema global a partir dos módulos.

Uma das características importantes em qualquer metodologia é o seu suporte à abstracção e à modularidade.

O modo mais directo para uma metodologia de programação poder suportar abstracção é o facto de possuir uma teoria de tipos bem desenvolvida.

As linguagens *ML* e *Haskell* (ao contrário do seu ilustre antecessor, *Lisp* [1]) possuêm tal teoria de tipos. O mesmo se passa com o *CafeOBJ* e, em certo grau, com o sistema de reescrita *ERIL*. Apenas o *Prolog* (tal como o *Lisp*) não suporta qualquer teoria de tipos sendo que, dificilmente suporta as técnicas de abstracção.

A mesma observação pode ser feita em relação à modularidade; dada a íntima relação entre estes dois conceitos, não é coincidência que uma teoria de tipos bem desenvolvida esteja ligada a uma boa gestão de módulos.

As linguagens *ML*, *Haskell* e *CafeOBJ* suportam ambas modularidade de programas. Porém nem Prolog nem Eril suportam qualquer tipo de modularidade.

A apresentação das várias metodologias terá em vista sempre o modo como estas técnicas podem (ou não) ter um suporte adequado.

Se todas estas metodologias oferecem tantas vantagens quando comparadas com a programação imperativa, qual a razão porque a grande maioria dos programas que correm na generalidade dos sistemas informáticos continuam a ser imperativos?

A resposta é simples: eficiência na execução. A programação imperativa está ligada intimamente à arquitectura da máquina que os executa e, como dissemos, substitui a dimensão complexidade do problema pela dimensão tempo. De certo modo podemos dizer que um problema grande divide a ocupação da máquina, pelos seus módulos ou componentes, através de fatias de tempo.

Quando a complexidade total do problema é reflectida na estrutura do programa e não partilhada no tempo, é natural que tal programa seja, à partida, mais exigente em termos de recurso da máquina.

Por outro lado, as metodologias de programação formal, fazem automaticamente para qualquer programa um conjunto de serviços genéricos que não existem nas metodologias de programação imperativas; em particular, toda a gestão de memória, é usualmente transparente ao programa que não lida com problemas de detalhe de implementação.

Na programação imperativa os custos da implementação estão sempre presentes e, portanto, é mais simples melhorar a eficiência do programa.

A programação formal, porém, oferece um tipo mais importante de eficiência: a eficiência no desenvolvimento e a garantia de correcção.