Linguagens e Ambientes de Programação (2018/2019)

Teórica 16 (08/mai/2019)

Sistemas de tipo e sua utilidade.
Tipificação estática. Tipificação dinâmica.
Sistemas de tipos com e sem falhas de proteção.
Polimorfismo e variedades de polimorfismo.

Tipos

Um **tipo** representa uma coleção de elementos de dados e tem associados um conjunto de *literais* mais um conjunto de *operações*.

Por exemplo, em OCaml o tipo `int` representa o conjunto do inteiros, tem associados os literais ..., -2, -1, 0, 1, 2, etc., e as operações `+`, `-`, `*`, `div`, `mod`, `succ`, etc.

Numa linguagem sem tipos, e.g. Assembler, uma operação pode ser aplicada a quaisquer dados sem que qualquer validação seja feita. Os dados são vistos como simples sequências de bits e cada operação interpreta uma sequências de bits da maneira que lhe convém.

Numa linguagem com tipos, cada elemento de dados tem um tipo associado. Ao tentar aplicar uma operação a valores que não têm o tipo esperado obtém-se um **erro de tipo**.

Por outras palavras, ao associar um tipo a uma sequência de bits, estamos a atribuir um significado a essa sequência de bits. A linguagem de programação usa a informação de tipo para determinar que as utilizações da sequência de bits fazem sentido ou não. Quando não fizerem sentido, estamos perante um **erro de tipo**.

Géneros de tipos

Cada linguagem de programação oferece um conjunto de tipos simples e um conjunto de construtores de tipos. É assim possível distinguir dois géneros de tipos:

- **Tipos simples:** tipos predefinidos cujos valores são escalares, ou seja valores atómicos: inteiros, carateres, reais, booleanos, enumerados.
- **Tipos estruturados:** tipos compostos definidos pelo programador aplicando determinadas construções a tipos existentes: arrays, registos, listas, tuplos, classes, funções, apontadores.

Utilidade dos tipos

Para que servem os tipos?

No caso das linguagens com tipificação estática:

- Permitem a deteção antecipada de erros, em tempo de compilação. **O compilador deve garantir que um programa validado já não vai ter erros de tipo em tempo de execução.**
- Ajudam a documentar os programas, tornando mais claras as intenções do programador (exceto se for usada inferência de tipos).
- Suportam a definição de módulos, que ocultam a representação interna dos valores.
- São úteis na otimização de código gerado.

No caso das linguagens com tipificação dinâmica:

- Suportam um modelo de execução no qual se lança uma exceção logo que seja detetado que um erro de tipo. **O código errado já não chega a ser executado.**

Sistemas de tipos

Um **sistema de tipos** é um conjunto de regras que:

- Associa tipos aos literais e às expressões duma linguagem. Por exemplo o literal `2` e a expressão `2+5*3` têm tipo inteiro.
- Determina quais são as manipulações permitidas dos valores de cada tipo. Por exemplo, em OCaml os inteiros podem ser manipulados usando apenas as operações associadas ao tipo inteiro.
- Define como os valores dos vários tipos interagem. Por exemplo, em C a expressão `2 + 5.2` e considerada válida, mas o valor `2` é automaticamente convertido em inteiro.

Para exemplificar, eis três regras que fazem parte do sistema de tipos da linguagem OCaml:

```
                                        exp : bool    exp' : bool
-------------        -------------     ---------------------------
 false : bool         true : bool          exp && exp' : bool
```

Nestas regras, por cima da barra são colocadas pré-condições e por baixo da barra são colocadas conclusões. As duas primeiras regras são axiomas e dizem que `false` e `true` são valores válidos de tipo `bool`; a terceira regra diz que se `exp` e `exp'` forem expressões válidas de tipo `bool`, então `exp && exp'` é também uma expressão válida de tipo `bool`.

Não mostramos mais regras, pois a formalização de sistemas de tipo não é matéria de LAP. Mas, pelo menos, fica a ideia.

Estas regras são úteis por várias razões. Por exemplo, servem para o utilizador compreender melhor a linguagem, servem para provar que o sistema de tipos é consistente (ou seja, que a cada expressão é associado um tipo único), servem como base para a escrita duma parte importante do compilador.

Verificação de tipos (Type checking)

Chama-se **verificação de tipos** ao processo de validação dum programa face às regras dum sistema de tipos.

Tipificação estática e suas limitações

Diz-se que uma linguagem usa **tipificação estática** se a verificação de tipos for efetuada em tempo de compilação. Os erros de tipo são detetados antes do programa começar a correr. Nestas linguagens declaram-se os tipos das variáveis, dos argumentos e do resultado das funções para o compilador ter uma base para fazer as suas validações. Em algumas linguagens, como o OCaml, não se declaram tipos porque é feita inferência de tipos.

As seguintes linguagens usam tipificação estática: OCaml, C, C++ e Java.

Numa linguagem com tipificação estática, repare que os tipos são associados às expressões que aparecem no texto dos programas e é o próprio texto que é validado. Depois, durante a execução do programa já compilado, os tipos dos valores são ignorados porque o compilador já garantiu a ausência de erros de tipo.

Mas atenção: Num programa validado, ou seja num programa sem erros de tipo, podem mesmo assim ocorrer outro género de problemas durante a execução:

1. Certos **valores particulares** podem gerar exceções: **zero** usado como denominador duma divisão, ou **null** usado como recetor de mensagem. O sistema de tipos lida com classes de valores (os tipos) e não com valores individuais.
2. O programa pode entrar num **ciclo infinito**.
3. O programa pode estar **incorreto**, ou seja ter erros do lógica. Apesar de funcionar, não faz o que o programador pretendia.

Tipificação estática e conservadorismo

Uma característica importantíssima dos sistemas de tipos das linguagens com tipificação estática é serem **conservadores**. Repare que quando se usa tipificação estática, os tipos representam informação incerta, correspondendo a "aproximações" dos valores que realmente serão usados em tempo de execução (os valores exatos não se podem prever, em geral). Para garantir segurança, o compilador tem sempre de assumir o pior caso.

Por exemplo, no código Java que se segue, a variável `a`, de tipo `Animal` recebe um gato. Depois, mais adiante, tenta-se fazer miar o animal referido pela variável `a`. Mas o compilador tem considerar que, nessa altura, a variável `a` poderá já não referir um gato e, por isso, produz um erro de compilação; no entanto, em tempo de execução até podia não haver problema se estivesse um gato na variável...

```
Animal a = new Cat() ;
...
a.meou() ;   // ERRO DE TIPO
```

Tipificação dinâmica e suas limitações

Diz-se que uma linguagem usa **tipificação dinâmica** se a verificação de tipos for efetuada em tempo de execução. Os erros de tipo são detetados durante a execução dos programas. Nestas linguagens não de declaram os tipos das variáveis, dos argumentos e do resultado das funções. Nestas linguagens uma mesma variável pode conter valores de tipos diferentes, em diferentes momentos da execução.

As seguintes linguagens usam tipificação dinâmica: JavaScript, Prolog, Lisp, Perl, Python, Ruby, APL e Smalltalk.

Numa linguagem com tipificação dinâmica, os tipos são associados aos valores que são usados em tempo de execução e não às variáveis. Portanto, em tempo de execução tem de existir informação de tipo associada aos valores. Sempre que se aplica uma operação a alguns valores, o tipo desses valores é testado pelo sistema de execução.

A limitação característica dos sistemas de tipos dinâmicos é o facto dos erros de tipo só serem detetados durante a execução dos programas. Mesmo um erro de tipo básico pode ficar por descobrir durante muito tempo, por se se situar em código que é executado muito raramente. O erro só será apanhado quando esse código for executado pela primeira vez.

Repare também que numa linguagem com tipificação dinâmica se gasta mais memória com a informação de tipo que é guardada nos valores e se gasta mais tempo a fazer validação de tipos em tempo de execução.

Tipificação dinâmica e ausência de conservadorismo

Os sistemas de tipos das linguagens com tipificação dinâmica **não precisam de ser conservadores** pois exercem a sua influência durante a execução dos programas, quando os argumentos exatos das operações são conhecidos e estão disponíveis para teste.

Por exemplo, no código JavaScript que se segue, a variável `a` recebe um gato. Depois, mais adiante, tenta-se fazer miar o animal referido por essa variável `a`. O compilador aceita o código, mas depois em tempo de execução, poderá ser detetado ou não um erro de tipo. Se a variável `a` referir mesmo um gato, tudo correrá bem. Se referir um valor de outro tipo, então ocorrerá um erro de tipo e será lançada uma exceção.

```
var a = NEW(Cat) ;
...
a.meou() ;
```

Elementos de tipificação dinâmica dentro de linguagens com tipificação estática

Há linguagens de programação com tipificação estática que incluem alguns elementos de tipificação dinâmica.

Um exemplo em Java: a operação **instanceof** permite ao programador testar dinamicamente o tipo de qualquer objeto; a operação de cast aplicada a um objeto também executa um teste de tipo implícito em tempo de execução. Para isto funcionar, os objetos em Java precisam de registar a classe a que pertencem.

Duas escolas de programação

Nos dias de hoje, entre as linguagens de programação mais usadas encontram-se tanto linguagens com tipificação estática como linguagens com tipificação dinâmica. Existem alguns adeptos enquadrados em cada uma das duas escolas de programação, os quais defendem renhidamente a sua "dama" (veja por exemplo esta discussão).

Por exemplo, os adeptos da tipificação estática acreditam que os seus programas são mais seguros depois de verificados, mas os adeptos da tipificação dinâmica argumentam que conseguem programar melhor as suas ideias sem constrangimentos artificiais e que apesar de tudo os seus programas têm provado ser robustos e conter poucos erros.

Na verdade é perfeitamente possível ser bons resultados dentro de cada uma das escolas de programação desde que se usem boas técnicas de desenvolvimento de software. Uma das técnica mais importante é certamente a técnica de escrever muitos testes unitários para validar sistematicamente todos os aspetos do software em desenvolvimento.

Só uma curiosidade: O CLIP, o sistema da informação da FCT, é um exemplo de software da escola da tipificação dinâmica - está escrito em Prolog.

Sistemas de tipos com e sem falhas de proteção

Um sistema de tipos diz-se **sem falhas** (em Inglês, ***safe***) se conseguir garantir que todos os erros de tipo são detetados e que código com erros de tipo nunca chegará a ser executado.

Os sistemas de tipos do C e C++ têm falhas porque:

- Alguns casts permitem violar o sistema de tipos de forma muito básica. Por exemplo é possível converter um inteiro num apontador para função.
- Mesmo sem usar um cast, usando aritmética de apontadores é fácil pôr um apontador inteiro a apontar para uma variável de tipo double e depois escrever nela (ou em parte dela) um inteiro.

- A gestão explícita de memória também é um problema. Suponhamos que temos um apontador para uma variável dinâmica inteira e que libertamos essa variável. Suponhamos que logo a seguir criarmos uma variável dinâmica de tipo double. Suponhamos ainda que, por acaso, a variável é criada na zona de memória anteriormente libertada, mas ainda apontada pelo apontador inteiro...

Exemplos de linguagens com sistemas de tipos sem falhas:

- OCaml
- Java
- Smalltalk
- JavaScript

Repare que na lista de linguagens com sistemas de tipos sem falhas aparecem linguagens com tipificação estática e linguagens com tipificação dinâmica.

Polimorfismo

Os programadores gostam de escrever código geral que possa ser aplicado a vários tipos de dados. É penoso, e causador de erros, ter de reescrever um algoritmo com ligeiras variações só porque surgiu a necessidade de o aplicar a um tipo de dados diferente.

Uma **função polimórfica** é uma função que pode ser aplicada a argumentos de vários tipos. A nossa conhecida função `len` em OCaml é polimórfica pois aplica-se a listas de qualquer tipo:

```
len : 'a list -> int
```

Um **tipo polimórfico** é um tipo cujas operações se aplicam a valores de mais do que um tipo. Em OCaml o tipo da listas `'a list` é polimórfico. Em Java o tipo `Vector<E>` também é.

Uma **variável polimórfica** é uma variável mutável que pode conter valores de tipos diferentes. Em Java, uma variável de tipo `Animal` pode referir qualquer objeto cujo tipo seja subtipo de `Animal`. Em C uma variável de tipo `void *` pode guardar qualquer apontador.

Entidades que não sejam polimórficas dizem-se **monomórficas**.

Muitas das linguagens com tipificação estática modernas suportam polimorfismo. Todas as linguagens com tipificação dinâmica suportam polimorfismo de forma inerente.

Variedades de polimorfismo

O seguinte diagrama identifica as variedades e subvariedades de polimorfismo de funções em linguagens com tipificação estática, de acordo com Cardelli:

- Polimorfismo
  - Universal
    - Paramétrico
    - Inclusão (ou subtipo)
  - Ad hoc
    - Overloading
    - Coerção

Polimorfismo universal - A função trabalha de forma uniforme sobre uma diversidade infinita de tipos que partilham a mesma estrutura. A implementação é única e o mesmo código consegue lidar com todos os tipos considerados.

Polimorfismo paramétrico - É uma forma de polimorfismo universal onde a função polimórfica tem um parâmetro de tipo implícito ou explicito. Na chamada da função o parâmetro de tipo pode ser ou não inferido. A função `len` em OCaml é polimórfica paramétrica, sendo `'a` o nome do parâmetro de tipo:

```
len : 'a list -> int

let rec len l =
    match l with
        [] -> 0
      | x::xs -> 1 + len xs
;;
```

A seguinte função em Java é polimórfica paramétrica, sendo `T` o nome do parâmetro de tipo:

```
<T> void fromArrayToCollection(T[] a, Collection<T> c) {
  for (T o : a) {
   c.add(o) ;
}}
```

Polimorfismo de inclusão - É uma forma de polimorfismo universal que resulta da noção de subtipo. Uma função que declare aceita argumentos dum dado tipo, digamos `Animal`, também aceita argumentos de subtipos desse tipo, digamos `Cat` ou `Lion`. Qualquer linguagens com subtipos suporta polimorfismo de inclusão.

A seguinte função em Java é polimórfica de inclusão:

```
int weight(Animal a) { ... }
```

Polimorfismo ad hoc - A função trabalha de forma não uniforme sobre uma diversidade finita de tipos que não partilham a mesma estrutura. Existem múltiplas implementações, uma para cada tipo considerado.

Overloading - O mesmo nome de função é usado para denotar diferentes implementações monomórficas. No ponto da chamada usa-se o contexto para descobrir qual das implementações deve ser usada. Portanto esta forma de polimorfismo não é mais do que uma conveniência sintática. Um exemplo: o operador "+" em Java denota três operações monomórficas distintas, não relacionadas entre si: (1) soma de inteiros; (2) soma de reais; (3) concatenação de strings.

Coerção - Uma coerção é uma conversão automática de tipo. As coerções fazem com que funções essencialmente monomórficas se tornem polimórficas, pois passam a poder ser chamadas com argumentos de diferentes tipos. A seguinte função em C foi escrita para ser monomórfica

```
double inc(double d) { return d + 1 ; }
```

mas devido ao facto de em C existir coerção de inteiros para doubles, a função passa a poder ser aplicada tanto a reais como inteiros.

Lendo as descrição anteriores, percebe-se porque razão o "polimorfismo ad hoc" também se chama "polimorfismo aparente". É apenas "açúcar sintático" que está em jogo.

#80