PONTIFÍCIA UNIVERSIDADE CATÓLICA DE MINAS GERAIS

Bacharelado em Ciência da Computação - Linguagens de Programação

Danniel Henrique Correa Vieira, Letícia Americano, Marcos Ani Cury Vinagre Silva

**RESUMO:**

**On understanding types, data abstraction, and polymorphism, by Luca Cardelli and Peter Wegner**

Belo Horizonte

2020

O grupo resumiu o artigo em sessões de forma que fique mais simples pra quem lê o resumo.

1. De universos não-tipificados para universos tipificados.

1.1

Não-tipificados: Significa na verdade que existe apenas um tipo nesse universo, como na memória de computador tudo é representando por string bits, ou, words. Porém existem linguagens como por exemplo Lips que utilizam outro tipo com sua propriedades únicas para ser seu único tipo.

Tipos surgem informalmente em qualquer domínio para categorizar objetos de acordo com o seu uso e comportamento. Essa categorização de objetos em termos do propósito para qual são usados, são usados para definir em um sistema de tipos mais ou menos bem definido.

Tipos surgem naturalmente

A-Calculus: Em Calculus tudo são funções e existe apenas um tipo, o tipo dos valores para valores. Aonde todos os valores são funções do mesmo tipo.

1.2

Tipos podem ser vistos como um conjunto de roupas ou de armadura que protege uma representação não-tipificada de uso arbitrário ou não-intencional.

Objetos de um determinado tipo têm uma representação que respeita as propriedades esperadas do tipo de dados. A representação é escolhida para facilitar a realização de operações esperadas em objetos de dados.

1.3

Linguagens Monomorficas: “Cada valor e variavel pode ser interpretado como pertencente de um e apenas um tipo”.

Linguagens Polimorficas: “Cada valor e variavel pode ser interpretado como pertencente por mais de um tipo”.

Funções Polimorficas são funções

1. O Cálculo Lambda

2.1 Cálculo Lambda não-tipificado

Cálculo Lambda inicialmente desenvolvido como uma notação sem tipo para capturar a essênciada aplicação funcional de operadores a operandos.

O simbolo Lambda vai ser substituido por fun na sintaxe para assemelhar mais a linguagens de programação.

Sintaxe:

- e::= x -- uma variavel é uma expressão Lambda

- e::= fun(x)e -- abstração funcional de e

- e::= e(e) -- operador e aplicado no operando e

Função identidade e Função Sucessora: (palavra-chave value usada para introduzir um novo nome amarrado a um valor ou função)

- value id = fun(x) x -- função identidade

- value succ = fun(x) x+1 -- função sucessora (para inteiros)

A exatidão das operações de soma de inteiros em calculo lambda não requer suposições sobre o que acontece quando fazemos operações com expressões que não representam inteiros. Para tal, é desejável que se defina o efeito de tentar somar argumentos não inteiros como um erro. Isso é feito em linguagens tipificadas por checagem de tipos em tempo de compilação, que elimina a possibilidade de operações em objetos de um tipo incorreto.

- value twice = fun(f) fun(y) f( f(y) ) -- twice function

- twice (succ) >>> fun(y) succ( succ(y) )

2.2 Calculo Lambda Tipificado

Sintaxe e ideia semelhante a não-tipificada, exceto que cada variavél deve ser explicitamente tipificada quando introduzida como uma variavél amarrada.

- value succ = fun (x: Int) x+1

- value twice = fun(f: Int -> Int) fun(y: Int) f( f(y) )

Essa notação se aproxima mais de uma escificação funcional de linguagens tipificadas mas otimitindo a tipificação do resultado. Podemos denotar o tipo do resultado dessa forma:

- value succ = fun(x: Int) (returns Int) x+1

Mas não é necessário pois tem como inferir pelo corpo da função: x+1.

Declaração de tipo usa a palavra chave type.

Convenção de Fun: Nome de tipo começa com letra maiuscula, e valores e funções começam com letra minuscula:

type IntPair = Int x Int

type intFun = Int -> Int

ATENÇÃO: Declaração de tipo introduz um nome (abreviação) para expressão de tipos, e não criam novos tipos.

Como indicar que um valor tem um tipo:

v : T

(3,4): IntPair

succ: IntFun

Não é necessário introduzir uma variavel com o tipo, porque ele pode ser determinado pela forma do valor dado:

value intPair = (3,4)

Mas se quiser pode ser feito assim:

value intPair : IntPair = (3,4)

value succ: Int -> Int = fun(x: Int) x + 1

Variaveis locais podem ser declaradas pelo let-in:

let a = 3 in a + 1 -- yields 4

let a : Int = 3 in a + 1

let a : T = M in N <---> (fun(a:T) N) (M)

2.3 Tipos básicos, Tipos Estruturados e Recursão

* Tipos básicos:
  + Unit , o tipo trivial, de um elemento ()
  + Bool , de operações if-then-else
  + Int , de operações aritméticas e de comparações
  + Real , de operações aritméticas e de comparações
  + String , de concatenação de strings (infixo) ^
* Tipos estruturados:

São construídos a partir dos básicos pelos construtores de tipos, tais como mapeamentos (->), produtos cartesianos (x), registros (também chamados produtos cartesianos) e tipos variantes (chamados de União discriminada).

Um par por exemplo é um tipo de Produto Cartesiano:

value p = 3, true : Int x Bool

Operadores em pares:

Primeiro componente -> fst(p) yields 3

Segundo componentet -> snd(p) yields true

Exemplo de registro:

type ARecordType = {a: Int, b: Bool, c: String}

value r: ARecordType = { a = 3, b = true, c = “abcd” }

Lembrando que cada elemento do registro tem quer ser de nome único e não repetido.

Operador em registro:

Seleção de campo ( . ) -> r.c yields “abcd”

Exemplo de registro com funções:

type FunctionRecordType = {f1: Int -> Int, f2: Real -> Real}

value functionRecord = {f1 = succ, f2 = sin}

Isso é possível pois funções são valores de primeira classe.

Um registro pode ser definido pode outro registro pelo operador &, que concactena os tipos de registro:

type NewFunctionRecordType = FunctionRecordType & {f3: Bool -> Bool}

Isso funciona como uma abreviação. É valido apenas quando usados em tipos de registro.

É possivel fazer uma estrutura ser privada e local para apenas uma coleção de funções por meio do operador let-in. Usar registros com funções é a forma mais eficiente. Por exemplo, aqui a variavel counter é privada e dividida pelas funções increment e total.

Value counter = let count = ref(0) in {increment = fun(n:Int) count := count + n, total = fun() count

}

counter.increment(3) counter.total() yields 3

Este exemplo envolve efeitos colaterais, uma vez que o principal uso de variáveis privadas é atualizá-las privadamente. O primitivo ref retorna uma referência atualizavel a um objeto, e as atribuições são restritas para trabalhar em tais referências. Esta é uma forma comum de ocultação de informações que permite a atualização do estado local usando escopo estático para restringir a visibilidade.

Exemplode de Tipo variante:

type AVariantType = [a: Int, b:Bool, c: String]

Esse tipo é formado por um conjunto de tipos rotulados não ordenados, dentro de colchetes.

Um elemento desse tipo pode ser um dos tipos rotulados:

value v1 = [a = 3] value v2 = [b = true] value v3 = [c = “abcd”]

A única operação com variantes é o case select:

case *variant* of [a = *variable of type* Int] *action for case* a [b = *variable of type* Bool] *action for case* b [c = *variable of type* String] *action for case* c otherwise *action for otherwise*

onde em cada case, uma nova variavél é introduzida e amarrada a conteudo respectivo do variante. Aquela variavel é então usada na ação respectiva. Por exemplo, aqui tem uma função que dado o elemento do variante acima, retorna uma string:

value f = fun(x: AVariantType) case x of [a = anInt] “it is an integer” [b = aBool] “it is a boolean” [c = aString] “it is the string: “ ^ aString otherwise “error”

aqui, o conteudo de x é amarrado a anInt, aBool ou aString dependendo do caso.

No calculo lambda não-tipificado é possível expressar operadores de recursividade usa-los para definir funções recursivas, poŕem, em calculo lambda tipificado todas as expressões devem terminar. Por isso, definições recursivas são introduzidas como um conceito primitivo. Por exemplo, a função fatorial:

rec value fact = fun(n: Int) if n=0 then 1 else n\*fact(n-1)

Apenas valores de função podem ser definidas recursivamente.

Recursividade pode ser usada também em definição de tipos. Isso permite, por exemplo, definir o tipo de uma de lista de inteiros a partir de registros e variantes:

rec type IntList = [nil: Unit, cons: {head: Int, tail: IntList } ]

nesse tipo, uma lista ou é nil (nula) (representada como [nil = ()]) ou é cons que é representada por uma head e um tail.

**REFERÊNCIAS**

* Cardelli, Luca; Wegner, Peter. On understanding types, data abstraction, and polymorphism. Computing Surveys. Vol. 17. Nº 4. December 1985. Pgs. 471-489.