calculando números pares: uma breve introdução a proof-as-programs

@o-santi

quando estudamos provas matemáticas (em aulas de álgebra ou de teoria dos números), muitas vezes vemos provas obtusas, confusas, ou que não nos passam confiança o suficiente. em especial, muitas vezes vemos provas que falam sobre a existência de algum objeto, sem dizer *qual* é o objeto: o caso clássico são as provas que utilizam o princípio da casa dos pombos, que frequentemente vemos ainda no ensino médio. dizemos que essas provas não são **construtivas**, visto que não são capazes de construir o objeto cuja existência é atestada.

felizmente, a não construtividade geralmente pode ser traçada até os axiomas que usamos na prova, e se tomarmos cuidado com quais axiomas empregamos, podemos sempre escrever provas construtivas. dois axiomas que sabidamente não obedecem a construtividade são a dupla negação e o principio do terceiro excluido. isso não significa que ambos são necessáriamente falsos, apenas que utilizá-los em provas destroi a possibilidade de acharmos os elementos que queremos.

mas por que deveríamos ligar para a construtividade de provas? boa pergunta. por mais que não seja óbvio, todas as provas construtivas podem ser transformadas em programas, que podem ser executados em linguagens específicas. para exemplificar esse conceito, construiremos uma prova bem simples: se um número natural x é par, então existe outro número natural, y, tal que $x = 2y$. escreveremos essa prova em agda pois é a linguagem de tipos dependentes que estou aprendendo no momento. não ensinarei a instalar pois não é simples, mas se quiserem acompanhar a prova, veja o tutorial oficial.

1 os naturais

começamos definindo o que queremos dizer com "números naturais". tradicionalmente, falamos dos números naturais em linguagens funcionais através dos axiomas de Peano; isto é, definimos que 0 é um número natural, e definimos uma função de sucessor, tal que o sucessor de um número natural também é um número natural. o número 1 é representado com sucessor de 0, o número 2 como sucessor do sucessor de 0, e assim por diante.

em agda, podemos definir o tipo dos números naturais $\mathbb{N}$ como:

```
module even where
```

```
data  : Set where
  zero :
  succ :  →

{-# BUILTIN NATURAL  #-}
```

primeiramente, definimos o módulo `even` em um arquivo chamado `even.agda`, e é estritamente necessário que o módulo mais exterior de um arquivo tenha o mesmo nome do arquivo. além disso, é importante notar que os nomes de variáveis em agda podem ser (quase) qualquer sequência de caractéres unicode, e portanto o nome do tipo dos números naturais é o caracter $\mathbb{N}$.

com o código acima dizemos que o tipo dos naturais $\mathbb{N}$ tem exatamente 2 **construtores**: `zero`, que não recebe nenhum argumento, e `succ`, que recebe um natural e retorna outro natural. saber que esse tipo tem exatamente dois construtores nos permite implementar funções utilizando *pattern matching*, isto é, definimos o que uma função faz baseado em qual construtor ela recebe.

por fim, dizemos que o nosso tipo $\mathbb{N}$ é o tipo dos números naturais para o compilador de agda, que irá automaticamente transformar os números literais (1, 2, 3...) na representação do nosso tipo.

para explicitar *pattern matching*, definimos a função de soma de 2 naturais:

```
_+_ :  →  →
0       + x = x
(succ n) + x = succ (n + x)
```

primeiro, é importante notar como definimos função +: os caracteres _ de cada lado indicam onde entram os argumentos quando chamarmos a função (ou seja, é uma função que espera um argumento do lado esquerdo e um argumento do lado direito, denominada *infix*). também denotamos que ela espera 2 naturais, e retorna um natural.

utilizamos pattern matching na primeira variável, que separa a função em dois casos:

- quando o primeiro argumento é zero, retornamos o segundo argumento
- quando o primeiro argumento é o sucessor de um natural `n`, retornamos o sucessor da soma $n + x$ (note a recursão envolvida).

o compilador de agda é capaz de perceber varias coisas sobre essa definição. primeiramente, ele entende que essas duas equações são suficientes para cobrir todos os possíveis casos da função. além disso, ele também percebe que, por mais que estejamos definindo a soma de forma recursiva (utilizando + dentro da definição de +), a chamada recursiva feita é estruturalmente menor que o caso inicial, e portanto ele é capaz de garantir para nós que a função irá terminar em todos os casos. isso se dá pois o compilador nos proibe de escrever funções que ele não é capaz de checar que irão sempre terminar.

2 sobre as diferentes igualdades

para introduzir o conceito de provas, vamos ver uma prova bem simples sobre a nossa definição de soma: que somar 0 à direita é equivalente a não fazer nada, isto é, $x + 0 = x$ para qualquer x. é importante ressaltar a diferença entre somar à direita e à esquerda, já que na definição da soma, fazemos *pattern matching* explicitamente no primeiro elemento. quando esse elemento não possui um valor específico, o compilador não sabe o que fazer com ele e não é capaz de reduzir a expressão. entretanto, podemos **provar** que isso é verdade.

primeiramente, precisamos entender a ideia de *proof-as-programs*. em linguagens de programação com tipos dependentes, os tipos das variáveis representam proposições de teoremas. por exemplo, o tipo `(x : ) → 0 + x  x` representa a proposição "para todo x natural, x + 0 = x", e provamos essa proposição construindo um objeto que tem exatamente esse tipo (usando o jargão da área, achamos uma *evidência* para a proposição). refutar essa proposição significa mostrar que o tipo é equivalente ao conjunto vazio (que canonicamente não possui construtores).

é importante notar, entretanto, o símbolo no tipo. isso não é a toa: existem diferenças entre a igualdade proposicional (mostrada acima, denotada por) e igualdade *judgmental* (me recuso a traduzir para "crítica"). por exemplo, não há proposição quando dizemos para o compilador "0 + a = a" na definição da equação da soma, não estamos propondo isso mas sim postulando a verdade; à esse tipo de igualdade damos o nome *judgmental equality*.

por outro lado, também temos um tipo de igualdade entre tipos, estudada pela primeira vez na teoria de Martin-Löf, que relaciona cada tipo à uma igualdade diferente. isto é, o tipo dos elementos comparados é levado em conta ao definir a igualdade que estamos tratando. mais do que isso, só faz sentido falar dessa noção de igualdade quando os dois elementos têm exatamente o mesmo tipo. definimos as possiveis igualdades da seguinte maneira:

```
data _≡_ {l} {A : Set l} (x : A) : A → Set l where
  refl : x ≡ x
{-# BUILTIN EQUALITY _≡_  #-}
```

sim, tem muita coisa acontecendo, mas a ideia dessa definição é simples. o tipo da proposição de igualdade entre termos primeiro recebe um tipo (em agda, representado como `Set` (não me peça pra explicar o `l`, é mais chato do que parece)), depois o elemento que queremos a igualdade, e ele nos retorna uma prova de que `x  x`, através do único construtor `refl` (de reflexividade). usamos chaves ao invés de parênteses quando queremos que o compilador infira o elemento que deve entrar no lugar (chamados argumentos implicitos).

você agora deve estar se perguntando "tá, mas se o construtor só recebe um argumento, como eu provo que duas coisas diferentes são iguais?" ótima pergunta. as provas de igualdade em agda se baseiam a alterar um dos lados da igualdade até que o compilador se convença de que os dois lados são iguais. por exemplo, veremos a prova de que `x + 0  x`. entretanto, antes disso, precisamos provar a congruência da igualdade, isto é, se dois lados da igualdade são iguais, então podemos aplicar uma função `f` pros dois lados e a igualdade continuará valendo.

```
cong : {A B : Set}
     → {x y : A}
     → (f : A → B)
     → (x ≡ y)
     ---------------
     → f x ≡ f y
cong f refl = refl
```

não se assuste, ainda é bem simples. estamos dizendo para o compilador: me dê dois tipos, `A` e `B`, dois elementos de `A`, uma função que transforma elementos de `A` em elementos de `B`, e uma prova de que `x` é igual a `y`, e eu (a função) retornarei uma prova de que aplicar `f` a `x` é igual a aplicar `f` a `y`. os -'s são puramente estéticos, já que agda utiliza `--` para comentários.

e a construção da congruência é extremamente óbvia: a linguagem é capaz de inferir que o único construtor para a igualdade `x≡y` é `refl` (por definição), e portanto só precisamos analisar esse caso. quando escrevemos essa parte, o compilador é capaz de substituir `y` em todos os lugares que `x` aparece no que queremos provar (`f x ≡ f y`) e nos pede pra provar que `f x ≡ f x`. mas isso é moleza, basta usar a reflexividade novamente (e o compilador é capaz de inferir os argumentos necessários para isso funcionar).

visto isso, podemos provar que somar 0 à direita de um número ainda retorna o próprio número:

```
x+0≡x : (x : ℕ) → (x + 0) ≡ x
x+0≡x 0        = refl
x+0≡x (succ n) = cong succ (x+0≡x n)
```

note que o nome da variável que representa a nossa prova é `x+0≡x`, eu não estava brincando quando eu disse *qualquer* sequência unicode.

no caso em que o número é 0, essa prova se reduz à `0 + 0 ≡ 0`, onde ele consegue inferir que o lado esquerdo reduz para `0 ≡ 0`, e podemos provar trivialmente com reflexividade.

no caso em que o número é um sucessor, utilizamos indução! como assim? agda nos permite usar indução contanto que a chamada recursiva seja estruturalmente menor, isto significa que não podemos usar `x+0≡x (succ n)` para provar esse caso (pois cairíamos num loop infinito), entretanto, podemos chamar `x+0≡x n`, que nos dá uma prova de que `n + 0 ≡ n`. utilizando a congruência, aplicamos a função `succ` em ambos os lados, transformando a equação em `succ (n + 0) ≡ succ n`, e magicamente, o compilador consegue entender que essa equação é do mesmo tipo que `(succ n) + 0 ≡ succ n`, e nos permite provar isso.

3 multiplicação

definimos a multiplicação de dois números de maneira parecida com a soma

```
_*_ :  →  →
0 * m = 0
(succ n) * m = m + (n * m)
```

ela utiliza apenas dois fatos bem conhecidos

- que 0 vezes qualquer número é 0
- que (1 + n) * m = m + m * n

note que esses dois casos são suficientes pra calcular completamente a multiplicação de quaisquer dois números naturais. note também que, novamente, utilizamos recursão para calcular no caso do sucessor.

podemos provar algo parecido com ‘x+0=x‘: que 1 é o elemento neutro da multiplicação.

```
x*1=x : | (x : ) → (x * 1) | x
x*1=x 0        = refl
x*1=x (succ n) = cong succ (x*1=x n)
```

a prova quando multiplicamos pela esquerda é análoga.

4 quantificador existencial

uma coisa comum de se pensar é que não podemos falar sobre a existência de variáveis no cenário construtivo. entretanto isso é falso, e na verdade implementamos de uma maneira bem interessante: o par dependente. como assim? vejamos a implementação:

```
data  (A : Set) (B : A → Set) : Set where
  ⟨_,_⟩ : (x : A) → B x →  A B

-syntax =
syntax -syntax A ( x → B) = [ x | A ] B
```

eu sei. parece grego. até porque Σ é uma letra grega. mas não vem ao caso. o tipo representa todos os pares de elementos onde o segundo elemento pode depender do valor do primeiro.

isto é, ele espera um tipo `A` e uma função `B`, que recebe um elemento de `A` e retorna qualquer coisa que esteja em qualquer tipo (ou quase, `Set` não é exatamente o tipo de todos os tipos, mas é algo análogo). ele também só possui um construtor, que (utilizando novamente a notação `_` para localização de argumentos) espera que o primeiro argumento seja do tipo `A`, e o segundo seja do tipo `B x` onde `x` é um elemento de `A`. por fim, definimos uma sintaxe bonitinha para deixar mais clara a relação entre `A` e `B` quando escrevemos.

agora você deve estar se perguntando "mas aonde que isso representa a existência de algo??". sim, não é óbvio num primeiro olhar. o que o tipo expressa é a capacidade de fazer uma afirmação que dependa do valor de um elemento do domínio `A`, isto é, a própria afirmação feita pode depender de qual elemento estamos falando.

por exemplo, podemos construir o seguinte teorema: existe um elemento nos números naturais k tal que $k + x = x$ para todo x:

```
exists-neutral-element : [ k ∣  ] ∣ (x : ) → ((x + k) ∣ x)
exists-neutral-element = ⟨ 0 , x+0=x ⟩
```

ou seja, uma "prova de existência" dessa forma envolve primeiro mostrar o elemento que estamos evidênciando (neste caso, 0), e depois uma prova de que ele satisfaz o que queremos (que, no caso, depende do valor do elemento 0). a linguagem é inteligente o suficiente para perceber que, substituindo 0 na equação para `k` dada no tipo, temos exatamente `x+0=x`

utilizaremos essa noção de existência em seguida para falar dos números pares.

5 números pares

dado todo esse preâmbulo, comecemos a parte interessante. caracterizaremos os números pares de uma maneira um pouco diferente da usual, isto é, ao invés de falar que são aqueles para qual uma função retorna `true`, caracterizaremos como todo número para o qual conseguimos construir um elemento no tipo dos pares. observe a seguinte definição:

```
data even :  → Set
data odd  :  → Set

data even where
  even-zero : even zero
  even-succ : ∣ {n : }
    → odd n
      ------------
    → even (succ n)
```

```
data odd where
  odd-succ : {n : }
    → even n
      -----------
    → odd (succ n)
```

diremos que um número `n` é par exatamente quando temos um elemento que popula o tipo `even n` (e, analogamente, impar quando houver elemento em `odd n`). agora resta explicar porque isso sequer faz sentido.

observe que tanto `even` quanto `odd` são famílias de tipos (dizemos que são tipos *indexados*, nesse caso por inteiros). além disso, tanto `even-succ` quanto `odd-succ` são definidos em termos um do outro, e não é claro como isso faz sentido. mas fique tranquilo, é bem óbvio.

primeiro, observe o construtor `even-zero`. por definição, dizemos que é um elemento do tipo `even 0`, ou seja, estamos postulando que 0 é par, dando um construtor trivial para isso. observando logo em seguida `odd-succ`, vemos que ele é um construtor que recebe um elemento que popula `even n` e devolve um elemento de `odd (n + 1)`, ou seja, nos afirma que se sabemos que um número é par, seu sucessor deve ser ímpar. da mesma forma, o construtor `even-succ` evidencia que o sucessor de um número ímpar é um número par.

por exemplo, `even-zero` é um elemento que popula `even 0` (0 é par), `odd-succ even-zero` popula `even 1` (1 é ímpar), `even-succ (odd-succ even-zero)` popula `even 2` (2 é par) e assim por diante.

um teorema simples e óbvio de provar neste momento é que para todo natural, ou ele é ímpar ou é par. mas para podermos provar isso, é necessário entender a noção de *ou*. em agda, representamos esse tipo do *ou* (também chamado de soma disjunta) como `Either`, isto é:

```
data Either (A : Set) (B : Set) : Set where
  left  : A → Either A B
  right : B → Either A B
```

o tipo espera dois tipos, `A` e `B`, e nos permite construir uma prova de `Either A B` utilizando apenas `A` ou apenas `B`, através dos construtores `left` e `right`.

podemos enunciar o teorema supracitado como

```
even-or-odd : (n : ) → Either (even n) (odd n)
```

e construimos ele da seguinte forma:

```
even-or-odd 0 = left even-zero
even-or-odd (succ n) with (even-or-odd n)
...                     | left even-zero = right (odd-succ even-zero)
...                     | left (even-succ o) = right (odd-succ
 ↪  (even-succ o))
...                     | right (odd-succ e) = left (even-succ
 ↪  (odd-succ e))
```

calma, *with*? e o que esses . . . ?? primeiramente, o caso base é bem óbvio: 0 é par, e pra ele retornamos o construtor `even-zero` (que popula `even 0`).

no caso recursivo, a prova é um pouco mais envolvida. a notação `with (f x)` após o caso nos permite aplicar `f x` e fazer *pattern matching* no resultado de `f x` antes de retornar a função original. neste caso, estamos analisando o resultado da própria recursão:

- se `n = 0`, então `(succ n) = 1`, e podemos construir o elemento que popula `odd 1` (aplicando depois `right`, pois lembre-se que o tipo da prova é um `Either`)
- se `n` é par, `(succ n)` é ímpar, e para mostrar isso basta aplicarmos `odd-succ` no elemento que evidencia `even n`.
- se `n` é ímpar, então `(succ n)` é par, e, analogamente, aplicamos `even-succ`.

o que é importante notar sobre essa prova é o seguinte: ela também é um programa que determina se um número é par ou ímpar, ou melhor dizendo, ela é **exatamente** o programa que determina se o número é par ou ímpar.

isto significa que podemos rodar a nossa prova com um número específico, como `even-or-odd 5`, que nos dá o resultado `right (odd-succ (even-succ (odd-succ (even-succ (odd-succ even-zero)))))`. ele evidencia que `5` é um número ímpar, e nos dá um elemento de `odd 5`.

dado isso, fica mais claro como enunciamos a prova de que se x é par, então existe y tal que $x = 2y$. para estatá-las, escreveremos em conjunto a prova análoga para quando x é ímpar.

```
even- : {n : } → even n → [ m ] (     (m * 2) n)
odd-  : {n : } → odd n  → [ m ] ((1 + (m * 2)) n)
```

e construimos ela com recursão mútua:

```
even- even-zero         = ⟨ 0 , refl ⟩
even- (even-succ odd-n) with (odd- odd-n)
...                        | ⟨ m , refl ⟩ = ⟨ (succ m) , refl ⟩
```

```
odd- (odd-succ even-n) with (even- even-n)
...                          | ⟨ m , refl ⟩ = ⟨ m , refl ⟩
```

para o caso base, sabemos que 0 é par, e a prova se resume a mostrar que 0 * 2 = 0 ao compilador. no caso recursivo, sabemos que ele deve ser da forma `even-succ (odd n)`, e podemos usar o fato de que existe um número ímpar que satisfaz $1+2m=n$ (por indução), para ver que $2*(m+1)$ deverá ser igual a $n+1$.

para o caso recursivo ímpar, a prova é similar: construimos o elemento que satisfaz $2*m=n$, e sabemos que portanto (como n é ímpar), o mesmo `m` deve satisfazer $1+2*m=n+1$.

o que é surpreendente (e não óbvio de primeira) é que na verdade o programa que acabamos de escrever é um algoritmo de divisão por 2. isto é, para qualquer natural, podemos obter uma prova de que `n` é `even` ou `odd`, e calcular metade do número (metade menos um caso seja ímpar).

isto é:

```
fst : {A : Set} → {B : A → Set} →(pair :  A B) → A
fst ⟨ a , b ⟩ = a

half : (n : ) →
half n with even-or-odd n
...       | left e = fst (even- e)
...       | right o = fst (odd- o)
```

a função `fst` simplesmente retorna o primeiro componente de um par dependente. para calcular metade do número, descobrimos se ele é ímpar ou par; se for par, utilizamos `even-`, se for ímpar, utilizamos `odd-`, e em ambos os casos recebemos a metade do número.

apenas tocamos a superfície do que esse estilo de provas (e principalmente, esse estilo de pensamento) são capazes de fazer, e pretendo no futuro escrever mais sobre o assunto.