Módulo V:

Sistemas de Tipos

MCC - 2ºano

José Bernardo Barros (jbb@di.uminho.pt)
José Carlos Bacelar Almeida (bacelar@di.uminho.pt)

Departamento de Informática
Universidade do Minho

Possibilidades de extensão para o sistema

O $\lambda^{\rightarrow}$ é um sistema com um poder expressivo muito limitado (quer quando visto sobre a perspectiva lógica ou computacional). Podemos extender o sistema[a] segundo diferentes objectivos mas, **tais extensões, não devem comprometer as boas propriedades do sistema, nomeadamente a propriedade de Church-Rosser e a normalização forte**.

Alguns *eixos*...

Lógica de Predicados: O sistema $\lambda^{\rightarrow}$ está associado a uma lógica proposicional. Uma extensão possível consiste em enriquecer o sistema por forma a corresponder à *lógica de predicados*.

Tipos de Dados: Quando entendemos o λ-calculus como uma linguagem de programação fica bem patente a sua pouca habilidade de representar naturalmente (i.e. facilmente) os dados dos problemas concretos — por outras palavras, *o sistema de tipos que lhe está subjacente é pouco expressivo*. É então natural procurarmos extender o sistema em:

Tipos Básicos - equipar o sistema com um conjunto de tipos básicos relevantes na programação (e.g. booleanos; naturais; etc.)

Construtores de Tipos - equipar o sistema com a habilidade de construir novos tipos de tipos existentes (e.g. pares; somas; listas; etc.)

Polimorfismo: A habilidade para definir funções que operam uniformemente sobre diferentes tipos de dados é um recurso poderoso em programação. A extensão do sistema-λ para incluir funções polimórficas foi um dos principais contributos desta teoria à "tecnologia de programação".

[a] Pelo que foi exposto, iremos adoptar a formulação de Church para conduzir as extensões ao sistema básico.

Tipos de dados Básicos

O sistema $\lambda^{\rightarrow}$ dispõe de um sistema de tipos que regula a definição/aplicação de funções. Mas, e numa perspectiva de *linguagem de programação*, carece do fundamental: a habilidade de representar naturalmente entidades sobre as quais pretendemos "calcular". Por outras palavras, falta um conjunto de **tipos básicos** sobre os quais possamos realizar as nossas funções (e.g. os booleanos; os naturais; etc.).

A teoria do λ-calculus pode facilmente ser extendida para considerar *tipos básicos*. É instrutivo antevermos quais os pontos da teoria que necessitamos ajustar:

1. Devemos reajustar a definição dos *tipos* do sistema (i.e. o conjunto $\mathcal{T}$). Nesse conjunto devemos incluir os tipos pretendidos.
2. Devemos descrever quais os *elementos* desses novos tipos. Em geral, e porque o conjunto desses elementos pode ser infinito, devemos definir a forma de "construir" elementos desses tipos (i.e. fornecer os **construtores** dos tipos).
3. Devemos disponibilizar uma forma de "fazer uso" desses tipos — por outras palavras, devemos ser capazes de condicionar a "execução do nosso programa" (i.e. a redução de uma função) com base no valor de um dado tipo. Este ponto divide-se tipicamente em;
   (a) definir "iteradores" (ou recursores[a]) sobre o tipo.
   (b) ajustar as regras de redução β para traduzirem a semântica pretendida.

É importante ter presente que **todas estes ajustes ao sistema-λ não devem nunca comprometer as suas propriedades fundamentais**, em particular a *normalização forte*.

[a] Serão introduziros adiante

Tipos e construtores

Comecemos por considerar uma extensão ao λ-calculus que inclua um tipo de **booleanos** e de **naturais**.

Extensão aos tipos

Na definição do conjunto de tipos é incluído um conjunto de tipos básicos. Assim,

$$\mathcal{T} ::= \mathbb{B} \mid \mathbb{N} \mid \mathcal{U} \mid \mathcal{T} \to \mathcal{T}$$

onde $\mathcal{U}$ é um conjunto de variáveis de tipos.

Construtores – regras de introdução

Os construtores dos tipos considerados são definidos como **constantes**. As regras de tipos associadas a estas constantes irão regular a sua aplicabilidade.

Para o caso dos booleanos basta-nos definir as suas duas constantes com as regras:

$$(\mathbb{B}\text{-Intro}) \quad \frac{}{\Gamma \vdash tt : \mathbb{B}} \qquad \frac{}{\Gamma \vdash ff : \mathbb{B}}$$

No caso dos naturais utilizamos os contrutores *zero* e *sucessor*.

$$(\mathbb{N}\text{-Intro}) \quad \frac{}{\Gamma \vdash 0 : \mathbb{N}} \qquad \frac{\Gamma \vdash n : \mathbb{N}}{\Gamma \vdash S\ n : \mathbb{N}}$$

Estas regras são designadas por **regras de introdução** (do tipo em questão) porque esse tipo surge na *conclusão* — assim, sempre que pretendermos estabelecer uma asserção de tipo onde esteja envolvido um desses tipos uma destas regras terá de ser utilizada.

Recursores

Utilização dos valores – regras de eliminação

Nesta extensão do sistema vamos considerar uma classe de operadores (ligeiramente) mais geral do que os *iteradores*. São designados por **recursores**[a] e só diferem dos iteradores para os tipos de dados recursivos. Para sermos mais precisos, devemos referir os iteradores/recursores como **uma família** de operadores indexada pelo tipo de resultado.

Façamos então a apresentação destes operadores relegando para mais tarde uma discussão da sua utilização e relação com o iterador. Para os booleanos temos.

$$(\mathbb{B}\text{-Elim}) \quad \frac{\Gamma \vdash b : \mathbb{B} \qquad \Gamma \vdash x : T \qquad \Gamma \vdash y : T}{\Gamma \vdash RecB_T\ xyb : T}$$

($RecB_T$ é, evidentemente, o eterno $if...then...else...$). Para os naturais

$$(\mathbb{N}\text{-Elim}) \quad \frac{\Gamma \vdash z : T \qquad \Gamma \vdash f : \mathbb{N} \to T \to T \qquad \Gamma \vdash n : \mathbb{N}}{\Gamma \vdash RecN_T\ z\ f\ n : T}$$

onde se observa que $RecN_T$ dispõe de mais um operador do que o iterador.

Por contraposição às regras de introdução, estas regras denominam-se por **regras de eliminação** já que o tipo em questão aparece nas premissas e "desaparece" da conclusão.

[a] Já referimos que os *iteradores* estão associados a uma classe importante de morfismos definidos sobre os tipos de dados indutivos: os *catamorfismos*. Também os *recursores* dispõe de uma classe de morfismos que está associada: os *paramorfismos* – estes últimos não foram referidos na disciplina de MP1

Relação de redução

A extensão ao sistema não fica completa sem atribuirmos a "semântica operacional" pretendida aos operadores intruduzidos. Para ilustrarmos esta necessidade consideremos o seguinte termo

$$RecB_N\ 0\ (S\ 0)\ tt$$

o termo é bem formado (pelas regras apresentadas) mas, até agora, nada existe no sistema que indique que este termo deva ser reduzido em 0 (traduzindo a semântica de $if...then...else...$ pretendida para $RecB_T$).

De facto, necessitamos de entender a noção de *redex* para incluir a interação dos recursores face aos construtores dos tipos considerados.

REDEXES ADICIONAIS:

$$\begin{array}{rcl} RecB_T\ x\ y\ tt & \rightsquigarrow & x \\ RecB_T\ x\ y\ ff & \rightsquigarrow & y \\ RecN_T\ z\ f\ 0 & \rightsquigarrow & z \\ RecN_T\ z\ f\ (S\ n) & \rightsquigarrow & f\ n\ (RecN_T\ z\ f\ n) \end{array}$$

Sistema T

Somando tudo obtemos o que é designado por **Sistema T**:

$$\text{(Ax)} \quad \frac{}{\Gamma, x : \tau \vdash x : \tau}$$

$$\text{(Abs)} \quad \frac{\Gamma, x : \tau_1 \vdash M : \tau_2}{\Gamma \vdash \lambda x : \tau_1 . M : \tau_1 \to \tau_2}$$

$$\text{(App)} \quad \frac{\Gamma \vdash M : \tau_1 \to \tau_2 \quad \Gamma \vdash N : \tau_1}{\Gamma \vdash M\ N : \tau_2}$$

$$(\mathbb{B}\text{-Intro}) \quad \frac{}{\Gamma \vdash \mathit{tt} : \mathbb{B}} \qquad \frac{}{\Gamma \vdash \mathit{ff} : \mathbb{B}}$$

$$(\mathbb{B}\text{-Elim}) \quad \frac{\Gamma \vdash b : \mathbb{B} \qquad \Gamma \vdash x : T \qquad \Gamma \vdash y : T}{\Gamma \vdash RecB_T\ x\ y\ b : T}$$

$$(\mathbb{N}\text{-Intro}) \quad \frac{}{\Gamma \vdash 0 : \mathbb{N}} \qquad \frac{\Gamma \vdash n : \mathbb{N}}{\Gamma \vdash S\ n : \mathbb{N}}$$

$$(\mathbb{N}\text{-Elim}) \quad \frac{\Gamma \vdash z : T \qquad \Gamma \vdash f : \mathbb{N} \to T \to T \qquad \Gamma \vdash n : \mathbb{N}}{\Gamma \vdash RecN_T\ z\ f\ n : T}$$

Redexes:

$$\begin{array}{rcl}
(\lambda x : \tau . M)\ N & \rightsquigarrow & M[x \leftarrow N] \\
RecB_T\ x\ y\ \mathit{tt} & \rightsquigarrow & x \\
RecB_T\ x\ y\ \mathit{ff} & \rightsquigarrow & y \\
RecN_T\ z\ f\ 0 & \rightsquigarrow & z \\
RecN_T\ z\ f\ (S\ n) & \rightsquigarrow & f\ n\ (RecN_T\ z\ f\ n)
\end{array}$$

Comentário: Vimos que os *redexes* adicionais considerados assumem sempre a forma da "interação" entre um operador introduzido por uma regra de eliminação com outro de uma regra de introdução. É curioso notar que também a definição "standard" de *redex* é afinal uma mesma instância deste fenómeno já que a regra de *Aplicação* pode ser identificada como *eliminação de* $\to$ e a de *Abstração* com a de *introdução*.

Recursividade estrutural: *iteradores/recursores*

Na presença de tipos recursivos, é importante a capacidade de definir funções recursivamente. Mas devemos ser criteriosos por forma a admitirmos um padrão de recursividade que garantidamente termine (admitirmos uma forma de recursividade arbitrária permitinos-ia definir facilmente funções que não terminavam comprometendo a normalização do sistema).

A solução consiste em controlar a recursividade com base nos próprios valores dos tipos de dados recursivos. Considere-se, a título de exemplo, o tipo dos naturais.

```
data Nat = Zero | Succ Nat
```

Um **padrão de recursividade** sobre os naturais que garantidamente termina (e que foi largamente estudado noutros cursos) corresponde ao que nós designamos por **Iteradores**. Esse padrão corresponde a uma função definida como

```
f Zero = ...
f (Succ n) = ... (f n) ...
```

Assim, definíamos o operador *iterNat* que permitia definir funções com este padrão de recursividade.

```
iterNat :: a -> (a->a) -> Nat -> a
iterNat z f Zero = z
iterNat z f (Succ n) = f (iterNat z f n)
```

iteradores e recursores (cont.)

No caso dos **Recursores** consideramos um *padrão de recursividade* menos restritivo, na medida em que permitimos à função definida utilizar o argumento da invocação recursiva. Este padrão de recursividade é designado por **recursividade primitiva**.

```
f Zero = ...
f (Succ n) = ... n ... (f n) ...
```

Assim o novo operador ajusta o tipo do segundo argumento.

```
recNat :: a -> (Nat->a->a) -> Nat -> a
recNat z f Zero = z
recNat z f (Succ n) = f n (recNat z f n)
```

EXEMPLOS:

```
soma Zero y = y
soma (Succ x) y = Succ (soma x y)
```

Daqui vemos que a função soma pode ser expressa como:

```
soma x y = iterNat y (\r->Succ r) x
soma' x y = recNat y (\p r->Succ r) x
```

Tratando-se a *recursividade primitiva* de um padrão mais liberal do que o imposto pelo iterador, é evidente que tudo expresso com o iterador pode ser expresso com o recursor.

De forma análoga podemos chegar à definição das funções...

```
mult x y = iterNat Zero (\r->soma y r) x
factorial x = recNat (Succ Zero) (\p r->mult (Succ p) r) x
```

OBS: a função `factorial` já faz uso do argumento do construtor pelo que o seu padrão de recursividade "já obriga" ao uso do recursor. Outro exemplo emblemático é a função `pred` (predecessor)[a].

[a] Adiante voltaremos a este assunto.

Sobre os Iteradores e os Recursores

Na presença de *pares* o poder expressivo de ambos operadores é equivalente.

```
data Nat = Zero | Succ Nat

iterNat :: a -> (a->a) -> Nat -> a
iterNat x f Zero = x
iterNat x f (Succ n) = f (iterNat x f n)

recNat :: a -> (Nat->a->a) -> Nat -> a
recNat x f Zero = x
recNat x f (Succ n) = recNat n (recNat x f n)
```

As seguintes definições mostram-nos que podemos definir um à custa do outro...

```
iterNat' x f n = recNat x (\ p r -> f r) n

recNat' x f n =
  snd (iterNat (Zero,x) (\(p,r)->(Succ p,f p r)) n)
```

Obs.: As definições apresentadas mostram-nos que (na presença de pares) "o poder expressivo" dos iteradores é equivalente ao dos recursores. Mas em termos *operacionais* a tradução de um recursor mediante a utilização do iterador tem um peso importante: o que com o recursor era conseguido com um passo de redução é agora obtido com uma sequência de reduções (com magnitude proprocionais ao argumento recursivo).

Construtores de tipos

Um outro eixo onde é possível extender o sistema-λ é *disponibilizarmos novos construtores de tipos.* De entre estes podemos salientar

produtos: dotar o sistema da capacidade de manipular "pares" de valores;

somas: valores que alternativamente pertencem a um tipo ou a outro;

listas: sequências finitas de valores de um dado tipo.

A metedologia que seguiremos é precisamente a mesma adoptada aquando da introdução dos tipos primitivos:

1. extender o conjunto de tipos
2. regras de introdução
3. regras de eliminação
4. ajuste da relação de redução

Como conjunto de tipos vamos considerar

$$\mathcal{T} ::= \mathcal{U} \mid \mathbb{B} \mid \mathbb{N} \mid \mathcal{T} \times \mathcal{T} \mid \mathcal{T} + \mathcal{T} \mid \text{List } \mathcal{T} \mid \mathcal{T} \to \mathcal{T}$$

Extensão ao sistema T – (versão c/ iteradores)

$$(\text{Ax})\ \frac{}{\Gamma, x : \tau \vdash x : \tau}$$

$$(\rightarrow\text{-intro})\ \frac{\Gamma, x : \tau_1 \vdash M : \tau_2}{\Gamma \vdash \lambda x : \tau_1 . M : \tau_1 \rightarrow \tau_2} \qquad (\rightarrow\text{-elim})\ \frac{\Gamma \vdash M : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash N : \tau_1}{\Gamma \vdash M\ N : \tau_2}$$

$$(\mathbb{B}\text{-intro})\ \frac{}{\Gamma \vdash \mathit{tt} : \mathbb{B}} \quad \frac{}{\Gamma \vdash \mathit{ff} : \mathbb{B}} \qquad (\mathbb{B}\text{-elim})\ \frac{\Gamma \vdash x : T \qquad \Gamma \vdash y : T \qquad \Gamma \vdash b : \mathbb{B}}{\Gamma \vdash iterB_T\ x\ y\ b : T}$$

$$(\mathbb{N}\text{-intro})\ \frac{}{\Gamma \vdash 0 : \mathbb{N}} \qquad \frac{\Gamma \vdash n : \mathbb{N}}{\Gamma \vdash S\ n : \mathbb{N}}$$

$$(\mathbb{N}\text{-elim})\ \frac{\Gamma \vdash z : T \qquad \Gamma \vdash f : T \rightarrow T \qquad \Gamma \vdash n : \mathbb{N}}{\Gamma \vdash iterN_T\ z\ f\ n : T}$$

$$(\times\text{-intro})\ \frac{\Gamma \vdash M : \tau_1 \qquad \Gamma \vdash N : \tau_2}{\Gamma \vdash (M, N) : \tau_1 \times \tau_2}$$

$$(\times\text{-elim})\ \frac{\Gamma \vdash M : \tau_1 \times \tau_2}{\Gamma \vdash \pi_1\ M : \tau_1} \quad \frac{\Gamma \vdash M : \tau_1 \times \tau_2}{\Gamma \vdash \pi_2\ M : \tau_2}$$

$$(+\text{-intro})\ \frac{\Gamma \vdash M : \tau_1}{\Gamma \vdash i_1\ M : \tau_1 + \tau_2} \quad \frac{\Gamma \vdash M : \tau_2}{\Gamma \vdash i_2\ M : \tau_1 + \tau_2}$$

$$(+\text{-elim})\ \frac{\Gamma \vdash M : \tau_1 + \tau 2 \qquad \Gamma \vdash f : \tau_1 \rightarrow T \qquad \Gamma \vdash g : \tau_2 \rightarrow T}{\Gamma \vdash case_T\ M\ f\ g : T}$$

$$(\text{List-intro})\ \frac{}{\Gamma \vdash Nil : List\tau} \qquad \frac{\Gamma \vdash M : \tau \qquad \Gamma \vdash L : List\ \tau}{\Gamma \vdash Cons\ M\ L : List\ \tau}$$

$$(\text{List-elim})\ \frac{\Gamma \vdash z : T \qquad \Gamma \vdash f : \tau \rightarrow T \rightarrow T \qquad \Gamma \vdash l : List\ \tau}{\Gamma \vdash iterList_T\ z\ f\ l : T}$$

Extensão ao sistema T (cont.)

REDEXES:

$$
\begin{array}{rcl}
(\lambda x : \tau . M)\ N & \rightsquigarrow & M[x \leftarrow N] \\
iterB_T\ x\ y\ tt & \rightsquigarrow & x \\
iterB_T\ x\ y\ ff & \rightsquigarrow & y \\
iterN_T\ z\ f\ 0 & \rightsquigarrow & z \\
iterN_T\ z\ f\ (S\ n) & \rightsquigarrow & f\ (iterN_T\ z\ f\ n) \\
\pi_1\ (x, y) & \rightsquigarrow & x \\
\pi_2\ (x, y) & \rightsquigarrow & y \\
case_T\ (i_1\ x)\ f\ g & \rightsquigarrow & f\ x \\
case_T\ (i_2\ x)\ f\ g & \rightsquigarrow & g\ x \\
iterList_T\ z\ f\ Nil & \rightsquigarrow & z \\
iterList_T\ z\ f\ (Cons\ x\ l) & \rightsquigarrow & f\ x\ (iterList_T\ z\ f\ l)
\end{array}
$$

OBSERVAÇÃO 1: Os contrutores de tipos *produto* e *soma* podem ser identificados (via analogia de Curry-Howard) pelas conectivas proposicionais *conjunção* e *disjunção* respectivamente.

OBSERVAÇÃO 2: O iterador do tipo (`List T`) é essencialmente o `foldr` do *Haskell*.

OBSERVAÇÃO 3: (Para quem se recorda de MP1...) O destrutor do coproduto $case_T$ está intimamente ligado ao "*either*". Mais precisamente,

$$case_T\ M\ f\ g = [f, g]\ M$$

Tipos de dados Indutivos

Dos exemplos apresentados é evidente o *padrão* que podemos generalizar para qualquer tipo de dados indutivo (ou construtores de tipos).

Os tipos indutivos são completamente caracterizados pela assinatura dos seus **construtores**. Considere-se um tipo indutivo **I** com construtores

$$\begin{array}{lcl} C_1 & : & \alpha_{11} \to \cdots \to \alpha_{1i} \to I \\ & \cdots & \\ C_n & : & \alpha_{n1} \to \cdots \to \alpha_{nj} \to I \end{array}$$

Comentário: Nos tipos α_{kl} pode surgir ainda o tipo I (nesse caso dizemos que I é um tipo recursivo). De facto, existem algumas restrições sobre a forma como esse tipo pode aparecer nos argumentos dos construtores.

O **iterador** para o tipo **I** pode também ser inferido com base na informação relativa aos tipos dos construtores

$$\begin{array}{lcl} iterI_T & : & (\overline{\alpha_{11}} \to \cdots \to \overline{\alpha_{11}} \to T) \to \\ & & (\overline{\alpha_{11}} \to \cdots \to \overline{\alpha_{11}} \to T) \to \\ & & I \to a \end{array}$$

onde $\overline{\alpha_{kl}} = \alpha_{kl}[I := T]$. A definição desta operação surge como

$$\begin{array}{rcl} iterI_T\ f_1\ \cdots\ f_n\ (C_1\ x_1\ \cdots\ x_i) & = & f_1\ \overline{x_1}\ \cdots\ \overline{x_i} \\ & \cdots & \\ iterI_T\ f_1\ \cdots\ f_n\ (C_n\ x_1\ \cdots\ x_j) & = & f_n\ \overline{x_1}\ \cdots\ \overline{x_j} \end{array}$$

em que

$$\overline{x_k} = \begin{cases} x_k & \text{,se tipo de } x_k \text{ diferente de } I \\ (iterI_T\ f_1\ \cdots\ f_n\ x_k) & \text{,se tipo de } x_k \text{ é } I \end{cases}$$

O polimorfismo

Como capturar num sistema-λ o polimorfismo[a]?

Esta questão é particularmente pertinente numa formulação de Church porque ai (por via das anotações de tipo nas abstrações) temos que um termo possui necessariamente um único tipo. Um exemplo paradigmático do polimorfismo é a função identidade: uma função que retorne directamente o seu argumento. É intuitivo que esta função pode operar sobre qualquer tipo mas, quando escrevemos esta na versão de Church de $\lambda^{\rightarrow}$ vemo-nos na necessidade de explicitar qual o tipo do argumento[b].

Antevisão do sistema:

Para capturar a noção de polimorfismo introduz-se o quantificador universal sobre as variáveis de tipo. É então possível exprimir um tipo como

$$\forall_\alpha \ \alpha \rightarrow \alpha$$

que pretende representar precisamente o tipo da função *identidade*.

A correspondencia da quantificação universal ao nível dos termos faz-se por intermédio de um novo operador de abstração que opera ao nível das variáveis de tipos (denotado por Λ). A referida função identidade polimórfica será dada então pelo termo

$$\Lambda_\alpha \ \lambda x : \alpha . x$$

Podemos ler este termo como: "recebe primeiramente *um tipo* como argumento; um segundo argumento é um valor do tipo recebido e retorna como resultado esse valor"

[a] i.e. a capacidade de definir funções que podem ser aplicadas a diferentes tipos de dados

[b] Aqui podemos argumentar que na formulação de Curry dispomos de "alguma forma de polimorfismo", na medida que o termo $\lambda x.x$ possui realmente muitos tipos (sendo o tipo principal $\tau \rightarrow \tau$). Iremos ter oportunidade que verificar que este polimorfismo é na realidade um caso particular do que será aqui estudado

Sistema F

O λ-calculus polimórfico é normalmente designado por **Sistema F** e foi introduzido por *J. Y. Girard (1972)*.

FORMALIZAÇÃO:

No que respeita aos tipos devemos considerar um novo operador: *quantificação universal*. Podemos defini-lo pela gramática,

$$\mathcal{T} ::= \mathcal{U} \mid \mathcal{T} \rightarrow \mathcal{T} \mid \forall_{\mathcal{U}} \mathcal{T}$$

onde $\mathcal{U}$ é um conjunto de variáveis de tipo. Considera-se que o operador $\forall$ dispõe de menor precedência do que o operador $\rightarrow$ e que é também associativo à direita.

Quanto ao conjunto de termos, devemos considerar um novo operador de abstração (Λ). Existindo dois tipos de abstração (de variáveis de termos e de tipos) iremos dispôr de dois tipos de aplicação (termos aplicados a termos ou termos aplicados a tipos) – tal como a aplicação de termos a termos omitiremos qualquer símbolo na aplicação de termos a tipos.

O conjunto dos *psedo-termos*[a] é então dado pela gramática,

$$\mathcal{L} ::= \mathcal{V} \mid \lambda \mathcal{V} : \mathcal{T}.\mathcal{L} \mid \Lambda_{\mathcal{U}}.\mathcal{T} \mid \mathcal{L}\ \mathcal{L} \mid \mathcal{L}\ \mathcal{T}$$

onde $\mathcal{V}$ é o conjunto das vasriáveis de termos. Iremos considerar mesmas regras de precedência e associatividade nas operações de abstração e aplicação.

[a] As regras de tipos é que irão determinar a sua boa formação...

Sistema F

Regras de tipos

$$(\text{Ax})\ \frac{}{\Gamma, x : \tau \vdash x : \tau}$$

$$(\rightarrow\text{-intro})\ \frac{\Gamma, x : \tau_1 \vdash M : \tau_2}{\Gamma \vdash \lambda x : \tau_1 . M : \tau_1 \rightarrow \tau_2}$$

$$(\rightarrow\text{-elim})\ \frac{\Gamma \vdash M : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash N : \tau_1}{\Gamma \vdash M\ N : \tau_2}$$

$$(\forall\text{-intro})\ \frac{\Gamma \vdash M : \tau}{\Gamma \vdash \Lambda\alpha . M : \forall_\alpha\ \tau}\ \text{se } \alpha \text{ não ocorre em } \Gamma$$

$$(\forall\text{-elim})\ \frac{\Gamma \vdash M : \forall_\alpha \tau}{\Gamma \vdash M\ \theta : \tau[\alpha := \theta]}$$

Redexes:

A relação de redução β deverá então ser ajustada convenientemente por forma a considerar as novas formas de abstração/aplicação:

$$\begin{array}{rcl} (\lambda x : \tau . M)\ N & \rightsquigarrow & M[x \leftarrow N] \\ (\Lambda\alpha . M)\ \theta & \rightsquigarrow & M[\alpha \leftarrow \theta] \end{array}$$

Obs.: numa hipotética formulação de Curry do sistema **F** o problema de inferência de tipos é indecidível.

Obs.: o *Haskell* admite polimorfismo mas numa forma restrita – todos os quantificadores são ao nível mais exterior. De resto, e tratando-se de um sistema de Curry, o resultado anterior torna impossível o processo de inferência de tipos.

Expressividade do Sistema F

Ao contrário do sistema $\lambda^{\rightarrow}$ em que foi necessário extender o sistema por forma a contemplar tipos indutivos, o sistema **F** tem a extraordinária capacidade de ser possível definir esses tipos no sistema puro (sem tipos primitivos).

A técnica é essencialmente a apresentada aquando da codificação de tipos indutivos no sistema λ sem tipos: *o tipo de dados é o tipo dos seus iteradores* — agora o sistema de tipos permite-nos capturar esse tipo introduzindo uma quantificação universal no tipo de resultado do iterador.

O exemplo dos *booleanos* é então definido como:

$$\begin{aligned} Bool &\doteq \forall_\alpha\ \alpha \rightarrow \alpha \rightarrow \alpha \\ True &\doteq \Lambda_\alpha \lambda x, y : \alpha . x \\ False &\doteq \Lambda_\alpha \lambda x, y : \alpha . y \end{aligned}$$

Os naturais podem ser definidos como:

$$\begin{aligned} Nat &\doteq \forall_\alpha\ \alpha \rightarrow (\alpha \rightarrow \alpha) \rightarrow \alpha \\ Zero &\doteq \Lambda_\alpha \lambda z : \alpha; f : \alpha \rightarrow \alpha . z \\ One &\doteq \Lambda_\alpha \lambda z : \alpha; g : \alpha \rightarrow \alpha . f\ z \\ &\ldots \end{aligned}$$

...e assim por diante...

Expressividade (cont.)

A utilização da técnica apresentada atrás permite-nos codificar toda a *lógica proposicional (intuicionista)* no sistema **F**.

$$\begin{aligned} A \Rightarrow B &\doteq A \rightarrow B \\ \bot &\doteq \forall_\alpha.\alpha \\ A \wedge B &\doteq \forall_\alpha.(A \rightarrow B \rightarrow \alpha) \rightarrow \alpha \\ A \vee B &\doteq \forall_\alpha.(A \rightarrow \alpha) \rightarrow (B \rightarrow \alpha) \rightarrow \alpha \end{aligned}$$

A negação é evidentemente definida como

$$\neg A \doteq A \Rightarrow \bot$$

O tipo $\bot$ corresponde ao tipo indutivo sem construtores (o tipo `void` do *Haskell*). Trata-se de um tipo sem nenhum habitante. Note-se que podemos provar *Ex Falsum Quod Libet*

$$\bot \Rightarrow A$$

o que é testemunhado pelo termo

$$\lambda x : \forall_\alpha \alpha \ . \ x \ A$$

Exemplo: Considere-se a formula $A \Rightarrow \neg\neg A$. Uma prova dessa formula corresponde a um termo do tipo $A \rightarrow (A \rightarrow \bot) \rightarrow \bot$. O candidato evidente é o termo $\lambda x : A.\lambda f : A \rightarrow \bot.f \ x$.