Sistemas de Tipos

Cristiano Damiani Vasconcellos

cristiano.vasconcellos@udesc.br

Departamento de Ciência da Computação
Universidade do Estado de Santa Catarina

Tipos

Tipos: Coleção de valores ou objetos que possuem alguma propriedade em comum.

Na matemática, tipos impõe restrições que evitam paradoxos. Universos não tipados apresentam inconsistências lógicas tais como o **paradoxo de Russell**.

Paradoxo de Russell

Paradoxo de Russell

Alguns conjuntos não são membros de si próprios, como por exemplo o conjunto de todas as cadeiras. Outros, como por exemplo o conjunto formado por tudo que não é cadeira, são membros de si mesmos. Definindo R como o conjunto de todos os conjuntos que não são membros de si próprio:

$$R = \{A \mid A \notin A\}$$

- Se R é membro dele mesmo então, por definição, R não é membro de R.
- Se R não é membro de R então, por definição, R é membro de R.

O próprio Russell respondeu seu paradoxo usando a teoria de tipos, definindo uma hierarquia para as proposições. Um dado predicado é valido para todos objetos que estiverem em um mesmo nível (ou forem do mesmo tipo).

Tipos

Linguagens que não definem um intervalo de valores que uma variável pode armazenar são classificadas como não tipadas. Essas linguagens suportam um único tipo que representa todos os valores. λ-cálculo é um exemplo extremo de linguagens não tipadas.

Uma linguagem de programação é considerada segura (*safe*) se todos os erros de tipos podem ser detectados, ou seja, os tipos não podem ser violados. Linguagens não tipadas podem ser consideradas seguras efetuando a verificação em tempo de execução.

Tipos

A definição de um sistema de tipos no projeto de linguagens de programação é útil para:

- **Estruturação dos programas e documentação:** os tipo representam abstrações dos dados manipulados pelo programa e podem ajudar na compreensão do código.
- **Detecção de erros:** uma grande variedades de erros podem ser detectados automaticamente quando dados e funções são usados de forma inconsistente.
- **Eficiência:** informações sobre tipos permitem ao computador executar otimizações no código gerado.

Sistemas de Tipos

Sistemas de tipos são conjuntos de regras de inferência que permitem atribuir tipos as variáveis e expressões de linguagens de programação. O principal objetivo de um sistema de tipos é determinar, em tempo de compilação, se um programa é bem comportado, garantindo a ausência de erros de tipos em tempo de execução. Um sistema capaz de fornecer essa garantia é dito consistente (*sound*).

Para que seja possível provar a consistência do sistema de tipos é necessária a sua formalização.

Prova Matemática: Verificação de uma proposição por encadeamento de deduções lógicas a partir de um conjunto de axiomas.

Sistemas de Tipos

A definição formal de um sistema de tipos é feita por um conjunto de enunciados (regras) denominadas sentenças (*judgments*). Sentenças são afirmações sobre objetos sintáticos de um determinado tipo. Uma sentença tem a forma:

$$\Gamma \vdash e : \sigma$$

Essa sentença é lida como: no contexto Γ a expressão e tem tipo σ ou Γ implica (deriva) em e ter tipo σ. Sendo Γ um contexto, possivelmente vazio, onde estão definidos os tipos das variáveis que ocorrem livres em e ($\Gamma = \{x_1 : \sigma_1, x_2 : \sigma_2, \ldots, x_n : \sigma_n\}$).

A forma geral das regras de inferência é:

$$\frac{\Gamma_1 \vdash e_1 : \sigma_1 \quad \Gamma_2 \vdash e_2 : \sigma_2 \quad \ldots \quad \Gamma_n \vdash e_n : \sigma_n}{\Gamma \vdash e : \sigma}$$

As sentenças acima da linha horizontal são as premissas e a sentença abaixo é a conclusão. Por exemplo:

$$\frac{\Gamma \vdash e_1 : Nat \quad \Gamma \vdash e_2 : Nat}{\Gamma \vdash e_1 + e_2 : Nat}$$

λ-Cálculo Simplesmente Tipado

Variaveis de Tipos	α, β
Variáveis de Expressões	x, y, z
Expressões e	$::= x \mid \lambda x.\, e \mid e\ e'$
Tipo Simples α	$::= \tau \mid \tau \to \tau'$

$$\Gamma \vdash x : \tau\ (VAR) \quad \{x : \tau\} \in \Gamma$$

$$\frac{\Gamma \vdash e : \tau \to \tau' \quad \Gamma \vdash e' : \tau}{\Gamma \vdash e\ e' : \tau'}\ (APP) \qquad \frac{\Gamma, x : \tau' \vdash e : \tau}{\Gamma \vdash \lambda x.e : \tau' \to \tau}\ (ABS)$$

$\Gamma, x : \tau$ representada $\Gamma \cup \{x : \tau\}$, sendo que Γ não apresenta qualquer suposição de tipo para x.

Unificação

Unificação é a ideia central do processo de inferência de tipos, um unificador para dois tipos é uma substituição S que: $S\tau_1 = S\tau_2$.

Uma **substituição** é uma função que mapeia variáveis de tipos em expressões de tipos. Uma substituição pode ser representada como: $S = \{\alpha_1 \mapsto \tau_1, \alpha_2 \mapsto \tau_2, \ldots, \alpha_n \mapsto \tau_n\}$. A aplicação de uma substituição S em um tipo τ ($S\tau$) resulta na troca de todas as variáveis de tipo que ocorrem em τ e pertencem ao domínio de S pelo tipo correspondente em S.

A composição de substituições é representada por $S \circ S'$. Um unificador S_g é chamado de **unificador mais geral** se, para qualquer outro unificador S, existe uma substituição S' tal que $S' \circ S_g = S$.

Inferência de Tipos

```
data SimpleType = TVar Id
                |TArr SimpleType SimpleType
                deriving (Eq, Show)

data Expr       = Var Id
                |App Expr Expr
                |Lam Id Expr
                deriving (Eq, Show)
```

Inferência de Tipos

```
tiExpr g (Var i)   = return (tiContext g i, [])
tiExpr g (App e e')=
         do  (t, s1) <- tiExpr g e
             (t', s2) <- tiExpr g e'
             b <- freshVar
             let s3 = unify (apply s2 t) (t' --> b)
             return (apply s3 b, s3 @@ s2 @@ s1)
tiExpr g (Lam i e) =
         do  b <- freshVar
             (t, s) <- tiExpr (g /+/ [i:>:b]) e
             return (apply s (b --> t), s)
```

Tipo Produto (*Product Type, Record*)

O produto de dois tipos consistem em um par ordenado de valores, sendo cada valor do tipo especificado. Podemos generalizar o tipo produto como o produto de um conjunto finito de n tipos, sendo $n \geq 0$. O tipo **unit** (tipo unitário) é representado pelo produto nulo.

$$\Gamma \vdash \langle\rangle : \text{unit}$$

$$\frac{\Gamma \vdash e : \tau \quad \Gamma \vdash e' : \tau'}{\Gamma \vdash e \times e' : \tau \times \tau'} \ (PAIR)$$

$$\frac{\Gamma \vdash e \times e : \tau \times \tau'}{\Gamma \vdash \pi_1(e \times e') : \tau} \ (PROJ) \quad \frac{\Gamma \vdash \ e \times e' : \tau \times \tau'}{\Gamma \vdash \pi_2(e \times e') : \tau'}$$

Tipo União Disjunta (*Disjoint Union, Sum Type, Tagged Union*)

A união disjunta de dois tipos oferece uma escolha entre dois elementos de tipos possivelmente distintos. Cada um dos tipos é marcado com uma etiqueta (construtor do tipo) que permite a seleção por casamento de padrões (*pattern match*).

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash e + e' : \tau + \tau'}\ (SUM)\ \frac{\Gamma \vdash e' : \tau'}{\Gamma \vdash e + e' : \tau + \tau'}$$

$$\frac{\Gamma, x_1 : \tau_1 \vdash e_1 : \tau \quad \Gamma, x_2 : \tau_2 \vdash e_2 : \tau \quad \Gamma \vdash e : \tau_1 + \tau_2}{\Gamma \vdash \text{case } e \text{ of } \{x_1 \Rightarrow e_1 \mid x_2 \Rightarrow e_2\} : \tau}\ (CASE)$$

Booleanos

Um exemplo simples do tipo união disjunta é o tipo booleano:

$$\Gamma \vdash \mathit{True} : \mathit{Bool} \qquad \Gamma \vdash \mathit{False} : \mathit{Bool}$$

$$\frac{\Gamma \vdash e : \mathit{Bool} \quad \Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \text{if } e \text{ then } e_1 \text{ else } e_2 : \tau} \ (\mathit{IF})$$

Tipos Recursivos

Muitas linguagens permitem a definição de tipos recursivos, para podermos representar tipos recursivos em usamos um **operador de ponto fixo de tipos**. A expressão de tipo $\mu\alpha.\tau$ denota o isomorfismo dos tipos que satisfazem a equação $\mu\alpha.\tau \cong \{\alpha \mapsto \mu\alpha.\tau\}\tau$.

Por exemplo, o tipo Lista de inteiros pode ser definido como:

$$\mu\alpha.\langle\rangle + (\mathsf{Int} \times \alpha)$$

Que admite infinitas substituições de α por $\langle\rangle + (\mathsf{Int} \times \alpha)$:

$$\begin{array}{l}
\langle\rangle + (\mathtt{Int} \times \alpha) \\
\langle\rangle + (\mathtt{Int} \times \langle\rangle + (\mathtt{Int} \times \alpha)) \\
\langle\rangle + (\mathtt{Int} \times \langle\rangle + (\mathtt{Int} \times \langle\rangle + (\mathtt{Int} \times \alpha))) \\
\langle\rangle + (\mathtt{Int} \times \langle\rangle + (\mathtt{Int} \times \langle\rangle + (\mathtt{Int} \times \langle\rangle + (\mathtt{Int} \times \alpha)))) \ \ldots
\end{array}$$

Correspondência Curry-Howard

Demonstra uma correspondência direta entre tipos e teoremas. Uma função é uma prova, e o tipo de uma função é a fórmula provada.

Essa correspondência é demostrada com a **lógica intuicionista**.

Lógica Clássica

Três princípios fundamentais:

- **Reflexividade** – toda proposição deriva de si mesma: $\varphi \vdash \varphi$
- **Terceiro-excluído** – toda proposição é verdadeira ou falsa: $\vdash \varphi \vee \neg\varphi$
- **Não-Contradição** – não é possível que uma proposição seja simultaneamente verdadeira e falsa: $\vdash \neg(\varphi \wedge \neg\varphi)$

Lógica Intuicionista

Um sistema lógico mais fraco que a lógica clássica, possui um número menor de teoremas que podem ser demonstrados. Algo somente é verdade caso exista uma **prova construtiva**, portanto provas por absurdo não são permitidas.

Nesse sistema não existe **princípio do terceiro excluído**. Para uma prova construtiva do terceiro excluído, seria necessária uma prova da validade ou da falsidade de cada possível fórmula proposicional, o que é impossível.

A negação de uma proposição $\neg\varphi$ é definida como a obtenção do falso caso φ for vista como verdade, simbolicamente:

$$\neg\varphi \equiv \varphi \rightarrow \bot$$

Dedução Natural Intuicionista

$$\Gamma, \varphi \vdash \varphi \ (Ax)$$

$$\frac{\Gamma \vdash \varphi \quad \Gamma \vdash \psi}{\Gamma \vdash \varphi \wedge \psi} \ (\wedge I) \qquad \frac{\Gamma \vdash \varphi \wedge \psi}{\Gamma \vdash \varphi} \ (\wedge E) \frac{\Gamma \vdash \varphi \wedge \psi}{\Gamma \vdash \psi}$$

$$\frac{\Gamma \vdash \varphi}{\Gamma \vdash \varphi \vee \psi} \ (\vee I) \frac{\Gamma \vdash \psi}{\Gamma \vdash \varphi \vee \psi} \qquad \frac{\Gamma, \varphi \vdash \rho \quad \Gamma, \psi \vdash \rho \quad \Gamma \vdash \varphi \vee \psi}{\Gamma \vdash \rho} \ (\vee E)$$

$$\frac{\Gamma, \varphi \vdash \psi}{\Gamma \vdash \varphi \to \psi} \ (\to I) \qquad \frac{\Gamma \vdash \varphi \to \psi \quad \Gamma \vdash \varphi}{\Gamma \vdash \psi} \ (\to E)$$

$$\frac{\Gamma \vdash \bot}{\Gamma \vdash \varphi} \ (\bot E)$$

Fórmulas **Não** Deriváveis

As seguintes fórmulas **NÃO** são deriváveis na lógica intuicionista

$$\varphi \vee \neg\varphi$$

$$\neg\neg\varphi \to \varphi$$

$$(\neg\psi \to \neg\varphi) \to (\varphi \to \psi)$$

$$((\varphi \to \psi) \to \varphi) \to \varphi$$

$$\neg(\varphi \wedge \psi) \to (\neg\varphi \vee \neg\psi)$$

Correspondência Curry–Howard

$$\Gamma, \varphi \vdash \varphi \ (Ax) \qquad \Gamma, x : \tau \vdash x : \tau \ (VAR)$$

$$\frac{\Gamma \vdash \varphi \to \psi \quad \Gamma \vdash \varphi}{\Gamma \vdash \psi} \ (\to E) \qquad \frac{\Gamma \vdash e : \tau \to \tau' \quad \Gamma \vdash e' : \tau}{\Gamma \vdash e\ e' : \tau'} \ (APP)$$

$$\frac{\Gamma, \varphi \vdash \psi}{\Gamma \vdash \varphi \to \psi} \ (\to I) \qquad \frac{\Gamma, x : \tau' \vdash e : \tau}{\Gamma \vdash \lambda x.e : \tau' \to \tau} \ (ABS)$$

Correspondência Curry–Howard

$$\frac{\Gamma \vdash \varphi \quad \Gamma \vdash \psi}{\Gamma \vdash \varphi \wedge \psi}\ (\wedge I) \qquad \frac{\Gamma \vdash e : \tau \quad \Gamma \vdash e' : \tau'}{\Gamma \vdash e \times e' : \tau \times \tau'}\ (PROD)$$

$$\frac{\Gamma \vdash \varphi \wedge \psi}{\Gamma \vdash \varphi}\ (\wedge E) \qquad \frac{\Gamma \vdash \varphi \wedge \psi}{\Gamma \vdash \psi}$$

$$\frac{\Gamma \vdash e \times e : \tau \times \tau'}{\Gamma \vdash \pi_1(e \times e') : \tau}\ (PROJ) \qquad \frac{\Gamma \vdash \ e \times e' : \tau \times \tau'}{\Gamma \vdash \pi_2(e \times e') : \tau'}$$

Correspondência Curry–Howard

$$\frac{\Gamma \vdash \varphi}{\Gamma \vdash \varphi \vee \psi}\ (\vee I)\ \frac{\Gamma \vdash \psi}{\Gamma \vdash \varphi \vee \psi}$$

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash e + e' : \tau + \tau'}\ (SUM)\ \frac{\Gamma \vdash e' : \tau'}{\Gamma \vdash e + e' : \tau + \tau'}$$

$$\frac{\Gamma, \varphi \vdash \rho \quad \Gamma, \psi \vdash \rho \quad \Gamma \vdash \varphi \vee \psi}{\Gamma \vdash \rho}\ (\vee E)$$

$$\frac{\Gamma, x_1 : \tau_1 \vdash e_1 : \tau \quad \Gamma, x_2 : \tau_2 \vdash e_2 : \tau \quad \Gamma \vdash e : \tau_1 + \tau_2}{\Gamma \vdash \mathsf{case}\ e\ \mathsf{of}\ \{x_1 \Rightarrow e_1 \mid x_2 \Rightarrow e_2\} : \tau}\ (CASE)$$

Tipos Recursivos

Esse isomorfismo é definido por regras de conversão de um tipo $\mu\alpha.\tau$ em $[\mu\alpha.\tau/\alpha]\tau$ e vice-versa.

$$\frac{\Gamma \vdash e : \{\alpha \mapsto \mu\alpha.\tau\}\tau}{\Gamma \vdash \text{fold } e : \mu\alpha.\tau}\ (\textit{FOLD}) \qquad \frac{\Gamma \vdash e : \mu\alpha.\tau}{\Gamma \vdash \text{unfold } e : \{\alpha \mapsto \mu\alpha.\tau\}\tau}\ (\textit{UNFOLD})$$

Sistema Hindley-Milner

Em λ-cálculo simplesmente tipado os tipos são monomórficos as variáveis de tipos representam um único tipo em um contexto. No sistema *Hindley-Milner* são introduzidos tipos quantificados para o suporte ao **polimorfismo paramétrico**.

Variaveis de Tipos	α, β, γ		
Tipo Simples	τ	$::=$	$\alpha \mid \tau \to \tau'$
Tipo Polimórfico	σ	$::=$	$\tau \mid \forall \alpha.\sigma$

Figura: Expressões de Tipos

$$
\begin{aligned}
\mathit{ftv}(\alpha) &= \{\alpha\} \\
\mathit{ftv}(\tau \to \tau') &= \mathit{ftv}(\tau) \cup \mathit{ftv}(\tau') \\
\mathit{ftv}(\forall \alpha.\sigma) &= \mathit{ftv}(\sigma) - \{\alpha\}
\end{aligned}
$$

$\mathit{ftv}(\Gamma)$ denota a união $\mathit{ftv}(\sigma)$ para todo tipo σ que ocorre em Γ.

$$\frac{\beta_i \notin \mathit{ftv}(\forall\overline{\alpha}.\tau) \quad \tau' = \{\overline{\alpha} \mapsto \overline{\tau}\}\tau}{\forall\overline{\alpha}.\tau \leqslant \forall\overline{\beta}.\tau'}$$

Informalmente podemos dizer que o tipo $\forall\overline{\beta}.\tau$ é mais especifico ou o mesmo que $\forall\overline{\alpha}.\tau'$.

Variáveis		x, y, z
Expressões e	$::=$	x
		$\mid \lambda x.\, e$
		$\mid e\ e'$
		$\mid$ let $x = e$ in e'

Figura: Sintaxe de Expressões

Sistema Hindley-Milner

$$\Gamma \vdash x : \sigma \qquad \{x : \sigma\} \in \Gamma \qquad \text{(VAR)}$$

$$\frac{\Gamma \vdash e : \sigma}{\Gamma \vdash e : \sigma'} \qquad (\sigma \leqslant \sigma') \qquad \text{(INST)}$$

$$\frac{\Gamma \vdash e : \sigma}{\Gamma \vdash e : \forall \alpha.\sigma} \qquad (\alpha \notin \mathit{ftv}(\Gamma)) \qquad \text{(GEN)}$$

Figura: Sistema de tipos *Hindley-Milner*

Sistema Hindley-Milner

$$\frac{\Gamma \vdash e : \tau' \to \tau \qquad \Gamma \vdash e' : \tau'}{\Gamma \vdash e\ e' : \tau} \quad \text{(APP)}$$

$$\frac{\Gamma, x : \tau' \vdash e : \tau}{\Gamma \vdash \lambda x.e : \tau' \to \tau} \quad \text{(ABS)}$$

$$\frac{\Gamma \vdash e : \sigma \qquad \Gamma, x : \sigma \vdash e' : \tau}{\Gamma \vdash \text{let } x = e \text{ in } e' : \tau} \quad \text{(LET)}$$

Figura: Sistema de tipos *Hindley-Milner*

Algoritmo W

$W(\Gamma, x)$ =

Se $\Gamma(x) = \forall\alpha_1 ... \alpha_2.\tau$ então $(\{\alpha_i \mapsto \beta_i\}\tau, Id)$
senão *Falha*, sendo β_i *fresh*

$W(\Gamma, e\ e')$ =

let $(\tau, S_1) = W(\Gamma, e)$
$(\tau', S_2) = W(S_1\Gamma, e')$
S = *unificar* $(S_2\tau, \tau' \to \beta)$, sendo β *fresh*
in $(S\beta, S \circ S_2 \circ S_1)$

Algoritmo W

$$
\begin{aligned}
&W(\Gamma, \lambda x.e) \ = \\
&\qquad \texttt{let } (\tau, S) \ = \ W(\Gamma, x : \beta, e) \\
&\qquad \texttt{in } (S(\beta \to \tau), S)
\end{aligned}
$$

$$
\begin{aligned}
&W(\Gamma, \mathit{let}\ \ x = e\ \ \mathit{in}\ \ e') \ = \\
&\qquad \texttt{let } (\tau, S_1) \ = \ W(\Gamma, e) \\
&\qquad\qquad (\tau', S_2) \ = \ W(S_1\Gamma, x : \mathit{fechamento}\ \ (S_1\Gamma, \tau), e') \\
&\qquad \texttt{in } (\tau', S_1 \circ S_2)
\end{aligned}
$$

Sendo $\mathit{fechamento}(\Gamma, \tau) = \forall\overline{\alpha}.\tau$ e $\overline{\alpha} = \mathit{ftv}(\tau) - \mathit{ftv}(\Gamma)$.

Consistência (*Soundness*): Se $W(\Gamma, e)$ retorna (τ, S) então $\Gamma \vdash e : \tau$.

Completude (*Completeness*): Se $\Gamma \vdash e : \tau'$ então $W(\Gamma, e) = (\tau, S)$ tal que para qualquer substituição S': $\tau \leqslant S'\tau'$.

Sistema F

Sistema F ou **Cálculo Lambda Polimórfico** é uma linguagem mínima que ilustra os conceitos de tipos polimórficos. O Sistema F representa o polimorfismo de maneira explicita, o tipo é um parâmetro. Diferente do Sistema *Hindley-Milner* no qual o polimorfismo é explícito.

Foi proposto pelo lógico Jean-Yves Girard (*System F*) em 1972 e pelo cientista da computação John C. Reynolds (*Polymorphic Lambda Calculus*) em 1974.

Sistema F

Tipo	τ	$::=$	α	(Variável de tipo)
			$\mid \tau \to \tau'$	(Função)
			$\mid \forall \alpha.\tau$	(Tipo polimórfico)
Expressão	e	$::=$	x	(Variável)
			$\mid \lambda x : \tau.e$	(Abstração)
			$\mid e_1\ e_2$	(Aplicação)
			$\mid \Lambda \alpha.e$	(Abstração de tipo)
			$\mid e\ [\tau]$	(Aplicação de tipo)

Figura: Tipos e Expressões (Sistema F)

Sistema F

Intuitivamente $\Lambda\alpha.e$ denota um termo e que tem como parâmetro um tipo polimórfico α. Uma aplicação de tipo corresponde ao chamado de uma função polimórfica passando o tipo como parâmetro real (*actual parameter*).

Um termo no Sistema F é fortemente normalizável, uma vez que não é possível definir um operador de ponto fixo por meio de termos fechados (combinadores).

Obs: Em linguagens como *Haskell* e *ML* o operador de ponto fixo pode ser definido como:

```
fix f = let x = f x in x
```

Sistema F

$$\Gamma \vdash x : \tau \ (VAR) \quad \{x : \tau\} \in \Gamma$$

$$\frac{\Gamma \vdash e : \tau \to \tau' \quad \Gamma \vdash e' : \tau}{\Gamma \vdash e\ e' : \tau'} \ (APP) \qquad \frac{\Gamma, x : \tau' \vdash e : \tau}{\Gamma \vdash \lambda x : \tau'.e : \tau' \to \tau} \ (ABS)$$

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \Lambda\alpha.e : \forall\alpha.\tau} \ (TABS) \qquad \frac{\Gamma \vdash e : \forall\alpha.\tau}{\Gamma \vdash e\ [\tau'] : \{\alpha \mapsto \tau'\}\tau} \ (TAPP)$$

Exemplos:

- $\Lambda\alpha.\lambda x : \alpha.x$
- $\Lambda\alpha.\Lambda\beta.\lambda f : \alpha \to \beta.\lambda x : \alpha.f\ x$
- $\Lambda\alpha.\Lambda\beta.\lambda f : \alpha \to \beta.\Lambda\gamma.\lambda g : \gamma \to \alpha.\lambda x : \gamma.f\,(g\ x)$

Exemplo em Haskell:

```
foo ::  (forall a.  a -> a) -> (Int, Bool)
foo = \f -> (f 1, f True)

foo' ::  (forall a.  a -> a) -> b -> c -> (b, c)
foo' = \f x y -> (f x, f y)
```

Sistema F (expressão `foo'`):
$\lambda f : \forall\alpha.\alpha \to \alpha.\, \Lambda\beta.\, \lambda x : \beta.\, \Lambda\gamma.\, \lambda y : \gamma.\, f\,[\beta]\,x \times f\,[\gamma]\,y$

Tipos de Dados Abstratos

Um **tipo de dado abstrato** fornece um contrato entre o programador que implementa e o que usa. Esse contrato é definido por meio de uma interface, ou seja, operações que podem ser aplicadas ao dado. O usuário pode manipular o dado apenas por meio dessa interface, dessa forma uma posterior alteração na representação do dado ou alterações no código que o manipula não ocasionarão alterações no código que faz uso do tipo abstrato.

Exemplo em Haskell (Fila de Prioridade):

$$
\begin{aligned}
&empty &&:: (Ord\ a) \Rightarrow PQueue\ a \\
&insert &&:: (Ord\ a) \Rightarrow a \rightarrow PQueue\ a \rightarrow PQueue\ a \\
&remove &&:: (Ord\ a) \Rightarrow PQueue\ a \rightarrow (a, Queuea) \\
&isEmpty &&:: (Ord\ a) \Rightarrow PQueue\ a \rightarrow Bool
\end{aligned}
$$

$$\text{type } PQueue\ a = [a]$$

$$empty = [\,]$$

$$
\begin{aligned}
&insert\ e\ [\,] &&= [e] \\
&insert\ e\ xxs@(x : xs) &&= \text{if } x < e \text{ then } x : insert\ e\ xs \text{ else } e : xxs
\end{aligned}
$$

$$remove(x : xs) = (x, xs)$$

$$
\begin{aligned}
&isEmpty\ [\,] &&= True \\
&isEmpty\ \_ &&= False
\end{aligned}
$$

Exemplo em Haskell (Fila de Prioridade)[1]:

class $PQueue\ f$ where

$$
\begin{aligned}
&empty &&:: (Ord\ a) \Rightarrow f\ a \\
&insert &&:: (Ord\ a) \Rightarrow a \rightarrow f\ a \rightarrow f\ a \\
&remove &&:: (Ord\ a) \Rightarrow f\ a \rightarrow (a, f\ a) \\
&isEmpty &&:: (Ord\ a) \Rightarrow f\ a \rightarrow Bool
\end{aligned}
$$

instance $PQueue$ [ ] where

$$
\begin{aligned}
&empty = [\,] \\
&insert\ e\ [\,] = [e] \\
&insert\ e\ xxs@(x : xs) = \text{if } x < e \text{ then } x : insert\ e\ xs \text{ else } e : xxs \\
&remove(x : xs) = (x, xs) \\
&isEmpty\ [\,] = True \\
&isEmpty\ \_ = False
\end{aligned}
$$

[1] Apenas um exemplo ilustrativo, tratamento de sobrecarga não será abordado.

Tipo	τ	$::=$	α	(Variável de tipo)
			$\mid \tau \to \tau'$	(Função)
			$\mid \forall\alpha.\tau$	(Tipo polimórfico)
			$\mid \exists\alpha.\tau$	(Interface)
Expressão	e	$::=$	x	(Variável)
			$\mid \lambda x : \tau.e$	(Abstração)
			$\mid e_1\ e_2$	(Aplicação)
			$\mid \Lambda\alpha.e$	(Abstração de tipo)
			$\mid e\ \tau$	(Aplicação de tipo)
			$\mid$ pack τ' with e as $\exists\alpha.\tau$	(Implementação)
			$\mid$ open e as $x : \tau$ in e'	(*uso*)

Figura: Sistema F estendido com Quantificador Existencial

Tipos de Dados Abstratos

$$\frac{\Gamma \vdash e : \{\alpha \mapsto \tau'\}\tau}{\Gamma \vdash \text{pack } \tau' \text{ with } e \text{ as } \exists\alpha.\tau : \exists\alpha.\tau} \ (\textit{PACK})$$

$$\frac{\Gamma \vdash e : \exists\alpha.\tau \qquad \Gamma, x : \tau \vdash e' : \tau'}{\Gamma \vdash \text{open } e \text{ as } x : \tau \text{ in } e' : \tau'} \ (\textit{OPEN})$$

Exemplo em Haskell (Fila):

$$\begin{aligned}
&\mathit{empty} :: \mathit{Queue}\ a \\
&\mathit{insert} :: a \to \mathit{Queue}\ a \to \mathit{Queue}\ a \\
&\mathit{remove} :: \mathit{Queue}\ a \to (a, \mathit{Queue}\ a)
\end{aligned}$$

type $\mathit{Queue}\ a = [a]$

$$\frac{\Gamma \vdash \mathit{insert} : \{\alpha \mapsto \mathit{List}\}\ \forall\beta.\beta \to \alpha\ \beta \to \alpha\ \beta}{\begin{array}{c}\Gamma \vdash \text{pack } \mathit{List} \text{ with } \mathit{insert} \text{ as } \exists\alpha.\forall\beta.\beta \to \alpha\ \beta \to \alpha\ \beta : \\ \exists\alpha.\forall\beta.\beta \to \alpha\ \beta \to \alpha\ \beta\end{array}}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathit{insert} : \exists\alpha.\forall\beta.\beta \to \alpha\ \beta \to \alpha\ \beta \\ \Gamma_{,\mathit{insert}:\exists\alpha.\forall\beta.\beta\to\alpha\ \beta\to\alpha\ \beta} \vdash \lambda x.\mathit{insert}\ x\ \mathit{empty} : \forall\beta.\beta \to \alpha\ \beta\end{array}}{\begin{array}{c}\Gamma \vdash \text{open } \mathit{insert} \text{ as } \mathit{insert} : \forall\beta.\beta \to \alpha\ \beta \to \alpha\ \beta \text{ in} \\ \lambda x.\mathit{insert}\ x\ \mathit{empty} : \forall\beta.\beta \to \alpha\ \beta\end{array}}$$

Sistema F: Codificação de Church

Dados primitivos, como números naturais e valores booleanos, podem ser representados em *cálculo lambda puro*. Essas representações podem ser transportadas para o *Sistema F* adicionando as anotações de tipos correspondentes:

$$\begin{aligned} \mathit{true} &= \lambda v.\lambda f.v \\ \mathit{false} &= \lambda v.\lambda f.f \end{aligned}$$

$$\begin{aligned} \mathit{true} &= \Lambda\alpha.\lambda v : \alpha.\lambda f : \alpha.v \\ \mathit{false} &= \Lambda\alpha.\lambda v : \alpha.\lambda f : \alpha.f \end{aligned}$$

O tipo **booleano** é então representado pelo tipo das funções que codificam os valores *true* e *false*.

$$\mathit{CBool} = \forall\alpha.\alpha \to \alpha \to \alpha$$

$$\begin{aligned} &\mathit{not} :: \mathit{CBool} \to \mathit{CBool} \\ &\mathit{not} = \lambda b : \mathit{CBool}.\Lambda\alpha.\lambda v : \alpha.\lambda f : \alpha.b\ [\alpha]\ f\ v \end{aligned}$$

Sistema F: Codificação de Church

Números naturais:

$$\mathit{zero} = \Lambda\alpha\lambda s : \alpha \to \alpha\lambda s : \alpha\lambda z : \alpha.z$$
$$\mathit{one} = \Lambda\alpha\lambda s : \alpha \to \alpha\lambda s : \alpha\lambda z : \alpha.s(z)$$
$$\mathit{two} = \Lambda\alpha\lambda s : \alpha \to \alpha\lambda s : \alpha\lambda z : \alpha.s(s(z))$$

$$\mathit{CNat} = \forall\alpha.(\alpha \to \alpha) \to \alpha$$

$$\mathit{succ} :: \mathit{CNat} \to \mathit{CNat}$$
$$\mathit{succ} = \lambda n : \mathit{CNat}.\Lambda\alpha.\lambda s : \alpha \to \alpha.\lambda z : \alpha.s(n\ [\alpha]\ s\ z)$$

É importante observar que é possível representar dados primitivos no Sistema F puro, pois isso sinaliza que é possível, no projeto de uma linguagem, substituir a representação desses dados sem que isso influa nas propriedades da linguagem.

$$PairNat = \forall\alpha.(CNat \to CNat \to \alpha) \to \alpha$$

$$pairNat = \lambda x : CNat.\lambda y : CNat.\Lambda\alpha.\lambda p : CNat \to CNAt \to \alpha.p\ x\ y$$

$$fstNat = \lambda p : PairNat.p\ [CNat]\ (\lambda x : CNat.\lambda y : CNat.x)$$
$$sndNat = \lambda p : PairNat.p\ [CNat]\ (\lambda x : CNat.\lambda y : CNat.y)$$

Generalizando os tipos do par:

$$Pair = \Lambda\beta.\Lambda\gamma.\forall\alpha.(\beta \to \gamma \to \alpha) \to \alpha$$

Obs: Note que não é possível representar o **construtor de tipos** *Pair* no Sistema F.

Sistema F_ω

Contexto	Γ	$::=$	$\emptyset$	(Vazio)
			$\mid \Gamma, x : \tau$	(Variável)
			$\mid \Gamma, \alpha :: k$	(Variável de Tipo)
Kind	k	$::=$	$\star$	(Tipo Apropriado)
			$\mid k \Rightarrow k$	(Operador)
Tipo	τ	$::=$	α	(Variável de tipo)
			$\mid \tau \rightarrow \tau'$	(Função)
			$\mid \forall \alpha :: k.\tau$	(Tipo polimórfico)
			$\mid \Lambda \alpha :: k.\tau$	(Operador de tipo)
			$\mid \tau\ \tau'$	(Aplicação de Tipo)
Expressão	e	$::=$	x	(Variável)
			$\mid \lambda x : \tau.e$	(Abstração)
			$\mid e_1\ e_2$	(Aplicação)
			$\mid \Lambda \alpha :: k.e$	(Abstração de tipo)
			$\mid e\ [\tau]$	(Aplicação de tipo)

Figura: Kind, Tipos e Expressões (Sistema F_ω)

A introdução de abstração e aplicação sobre tipos possibilita escrever o mesmo tipo de várias formas, o que torna necessário a definição de equivalência entre tipos.

$$
\begin{aligned}
& \mathit{CNat} \to \mathit{CBool} \\
& \equiv ((\Lambda\alpha.\alpha)\mathit{CNat}) \to \mathit{CBool} \\
& \equiv \mathit{CNat} \to ((\Lambda\alpha.\alpha)\mathit{CBool}) \\
& \equiv (\Lambda\alpha.\alpha)(\mathit{CNat} \to \mathit{CBool}) \\
& \equiv (\Lambda\alpha.\alpha)(((\Lambda\alpha.\alpha)\mathit{CNat}) \to \mathit{CBool})
\end{aligned}
$$

Sistema F_ω - Equivalência Estrutural entre Tipos

$$\tau \equiv \tau \ (\textit{E-REFL})$$

$$\frac{\tau \equiv \tau'}{\tau' \equiv \tau} \ (\textit{E-REFL}) \qquad \frac{\tau_1 \equiv \tau_2 \quad \tau_2 \equiv \tau_3}{\tau_1 \equiv \tau_3} \ (\textit{E-TRANS})$$

$$\frac{\tau_1 \equiv \tau_1' \quad \tau_2 \equiv \tau_2'}{\tau_1 \to \tau_2 \equiv \tau_1' \to \tau_2'} \ (\textit{E-ARROW}) \qquad \frac{\tau \equiv \tau'}{\forall \alpha :: k.\tau \equiv \forall \alpha :: k.\tau'} \ (\textit{E-ALL})$$

$$\frac{\tau \equiv \tau'}{\Lambda \alpha :: k.\tau \equiv \Lambda \alpha :: k.\tau'} \ (\textit{E-ABS}) \qquad \frac{\tau_1 \equiv \tau_1' \quad \tau_2 \equiv \tau_2'}{\tau_1 \ \tau_2 \equiv \tau_1' \ \tau_2'} \ (\textit{E-APP})$$

$$(\Lambda \alpha :: k.\tau) \ \tau' \equiv \{\alpha \mapsto \tau'\} \tau \ (\textit{E-APPABS})$$

Sistema F_ω - *Kind*

Podemos considerar tipos polimórficos como construtores de tipos que necessitam receber outros tipos como parâmetros. Um tipo monomórfico é um construtor de tipo com aridade 0, denotado por $\star$ e chamado de tipo apropriado (*proper type*).
Kind pode ser definido como a aridade do tipo.

$$\texttt{data}\ \underbrace{\mathit{Either}}_{\substack{\text{Constutor}\\ \text{de tipos}}}\ a\ b\ =\ \underbrace{\mathit{Left}}_{\substack{\text{Constutor}\\ \text{de dados}}}\ a\ \mid\ \underbrace{\mathit{Right}}_{\substack{\text{Constutor}\\ \text{de dados}}}\ b$$

$\text{data}\ \mathit{Bool} = \mathit{True} \mid \mathit{False}$ - $\mathit{Bool} :: \star$
$\text{data}\ \mathit{List}\ a = \mathit{Cons}\ a\ (\mathit{List}\ a) \mid \mathit{Nil}$ - $\mathit{List} :: \star \Rightarrow \star$
$\text{data}\ \mathit{Either}\ a\ b = \mathit{Left}\ a \mid \mathit{Right}\ b$ - $\mathit{Either} :: \star \Rightarrow \star \Rightarrow \star$

$\mathit{List}\ \mathit{Bool} :: \star$
$\mathit{Either}\ \mathit{Bool} :: \star \Rightarrow \star$

Sistema F_ω - *Kind*

$$\Gamma \vdash \alpha :: k \quad \{\alpha :: k\} \in \Gamma \ (K\text{-}VAR)$$

$$\frac{\Gamma, \alpha :: k \vdash \tau :: k'}{\Gamma \vdash \Lambda\alpha :: k.\tau :: k \Rightarrow k'} \ (K\text{-}ABS) \qquad \frac{\Gamma \vdash \tau :: k \Rightarrow k' \quad \Gamma \vdash \tau' :: k}{\Gamma \vdash \tau\ \tau' :: k'} \ (K\text{-}APP)$$

$$\frac{\Gamma \vdash \tau :: \star \quad \Gamma \vdash \tau' :: \star}{\Gamma \vdash \tau \to \tau' :: \star} \ (K\text{-}ARROW) \qquad \frac{\Gamma, \alpha :: k \vdash \tau :: \star}{\Gamma \vdash \forall\alpha :: k.\tau :: \star} \ (K\text{-}ALL)$$

Sistema F_ω

$$\Gamma \vdash x : \tau \quad \{x : \tau\} \in \Gamma \ (VAR)$$

$$\frac{\Gamma \vdash \tau :: \star \quad \Gamma, x : \tau \vdash e :: \tau'}{\Gamma \vdash \lambda x : \tau . e : \tau' \to \tau} \ (ABS) \qquad \frac{\Gamma \vdash e : \tau \to \tau' \quad \Gamma \vdash e' : \tau}{\Gamma \vdash e \ e' : \tau'} \ (APP)$$

$$\frac{\Gamma, \alpha :: k \vdash e : \tau}{\Gamma \vdash \Lambda \alpha :: k . e : \forall \alpha :: k . \tau} \ (TABS) \qquad \frac{\Gamma \vdash e : \forall \alpha :: k . \tau \quad \Gamma \vdash \tau' :: k}{\Gamma \vdash e \ [\tau'] : \{\alpha \mapsto \tau'\} \tau} \ (TAPP)$$

$$\frac{\Gamma \vdash e : \tau \quad \tau \equiv \tau' \quad \Gamma \vdash \tau' :: \star}{\Gamma \vdash e : \tau'} \ (EQ)$$

Sistema F_ω

$$\mathit{Pair} = \Lambda\beta :: \star \,.\, \Lambda\gamma :: \star \,.\, \forall\alpha \,.\, (\beta \to \gamma \to \alpha) \to \alpha$$

$$\mathit{pair} = \Lambda\beta :: \star \,.\, \lambda x : \beta \,.\, \Lambda\gamma :: \star \,.\, \lambda y : \gamma \,.\, \Lambda\alpha :: \star \,.\, \lambda f : \beta \to \gamma \to \alpha \,.\, p\ x\ y$$

$$\begin{aligned}\mathit{fst} \quad = \Lambda\beta :: \star \,.\, \Lambda\gamma :: \star \,.\, \lambda p : \forall\alpha \,.\, (\beta \to \gamma \to \alpha) \to \alpha \,.\\ p\ [\beta]\,(\lambda x : \beta \,.\, \lambda y : \gamma \,.\, x)\end{aligned}$$

$$\begin{aligned}\mathit{snd} \quad = \Lambda\beta :: \star \,.\, \Lambda\gamma :: \star \,.\, \lambda p : \forall\alpha \,.\, (\beta \to \gamma \to \alpha) \to \alpha \,.\\ p\ [\gamma]\,(\lambda x : \beta \,.\, \lambda y : \gamma \,.\, y)\end{aligned}$$