# Tipos em Linguagens de Programação

Carlos Camarão
DCC/UFMG

Lucília Figueiredo
DECOM/UFOP

Elaine Pimentel
DMAT/UFMG

16 de abril de 1999

**Resumo**

Estudos sobre tipos têm influenciado, de forma significativa, o projeto e a definição de linguagens de programação. Esse tutorial apresenta uma visão geral introdutória de tipos e sistemas de tipos para linguagens de programação modernas. Começamos identificando porque tipos são úteis, e seguimos discutindo a formalização da sintaxe de linguagens de programação, mostrando quais propriedades devem ser satisfeitas por sistemas de tipos, em particular com relação a definições da semântica denotacional e operacional de uma linguagem. É apresentada uma visão geral de sistemas de tipos simples, sistemas de tipos polimórficos, inferência de tipos, polimorfismo restrito, subtipagem e tipos abstratos.

**Palavras-chave:** Tipos, Linguagens de Programação, Sistemas de Tipos, Polimorfismo, Inferência de Tipos, Polimorfismo Restrito, Subtipagem, Tipos Abstratos.

**Abstract**

Studies about types have influenced, in a significant way, the design and definition of programming languages. This survey presents an introductory overview of concepts related to types and type systems for modern programming languages. We introduce by identifying why types are useful, and go on to discuss the formalization of the syntax of programming languages by pointing out which properties should type systems satisfy, in particular with respect to denotational and operational semantic definitions. We provide an overview of simple type systems, polymorphic type systems, type inference, constrained polymorphism, subtyping and abstract types.

**Keywords:** Types, Programming Languages, Type Systems, Polymorphism, Type Inference, Constrained Polymorphism, Abstract Datatypes.

# 1 Introdução

Na teoria de conjuntos tradicional, o agrupamento de elementos em um conjunto independe da natureza desses elementos. Quando passamos a trabalhar em aplicações específicas, precisamos classificar os objetos em categorias, de acordo com o seu uso ou aplicação. A noção de tipos origina-se dessa classificação: um *tipo* é uma coleção de objetos ou valores que possuem alguma propriedade em comum.[1] Em geral, para cada tal conjunto de valores, existe uma classe sintática correspondente, qual seja, de termos que representam esses valores, que também é chamada de tipo, em abreviação a *expressão de tipo*.

Em matemática, tipos impõem restrições que evitam paradoxos. Universos não tipados, como o da teoria de conjuntos tradicional, apresentam inconsistências lógicas (tais como o paradoxo de Russell) que são evitadas, por exemplo, com o uso de uma teoria de conjuntos tipada [Qui69, Hat82].

Em computação, existem diversas linguagens não tipadas (ou seja, que possuem apenas um tipo, que contém todos os valores) como, por exemplo: LISP, $\lambda$-calculus, Self, Perl e Tcl. Essas linguagens não dispõem de nenhum mecanismo para a detecção de falhas devidas a operações aplicadas a argumentos impróprios. A ocorrência de um erro dessa natureza não interrompe a execução do programa, sendo possível que o erro seja detectado somente após uma seqüência bastante grande de operações subseqüentes à ocorrência do mesmo.

O agrupamento de valores (e portanto de termos) em tipos é considerado útil à tarefa de programação, devido aos seguintes motivos:

- Estruturação de programas: tipos representam abstrações existentes no domínio da aplicação ou problema, assim como abstrações usadas na implementação da sua solução. A organização dessas abstrações em tipos é de ajuda fundamental na estruturação da implementação.

  Isso deve ocorrer de forma a que os tipos definidos no programa, e o modo como são definidos, correspondam, de maneira natural, às abstrações inerentes ao problema e às relações existentes entre as abstrações. De fato, a existência de uma correspondência direta entre abstrações do problema e as abstrações definidas no programa constitui uma caracterização fundamental de "programa bem feito".

- Clareza de programas: o tipo de uma expressão indica o conjunto dos possíveis valores que essa expressão pode representar, o que constitui informação valiosa para o entendimento do significado de um trecho de programa no qual essa expressão ocorre.[2]

- Detecção de erros: uma grande variedade de erros pode ser detectada automaticamente, antes da execução do programa (veja comentários a seguir).

- Eficiência: o tipo de uma expressão fornece informação que pode ser usada para a geração de código mais eficiente para um programa.

O tipo de uma expressão determina em que contextos a ocorrência dessa expressão é válida ou não. Em outras palavras, o agrupamento de valores em tipos permite que se verifique se expressões que denotam tais valores não são usadas em contextos em que não fazem sentido.

Essa verificação, comumente chamada de "checagem" de tipo, pode ser feita em tempo de compilação ou em tempo de execução de um programa. Quando a verificação é feita em tempo de compilação, além dos erros de tipo serem detectados antecipadamente (um programa não é executado caso contenha erros de tipos), eles são sempre detectados, podendo ser então corrigidos. No caso de verificação em tempo de execução, um erro existente só será detectado se alguma execução do programa envolver, de fato, o ponto onde tal erro ocorre; em outras palavras, o erro

[1] Esta é a noção clássica de tipos. Em [RRW91], são apresentadas duas outras abordagens: 'tipos como álgebras' e 'tipos como teorias'.

[2] Para alguns autores, a noção de tipos pode ser vista como uma *especificação parcial* de comportamento. Por exemplo, dir-se-ia, "ao descrever o tipo de uma função, é feita uma especificação parcial do comportamento dessa função, uma vez que é definido que a função deve receber um valor de um determinado tipo (dos valores que compreendem o domínio da função) e retornar um valor também de um certo tipo (dos valores que determinam a imagem da função). Além disso, a existência de tipos garante a ausência de uma certa classe de erros." No entanto, observe que nada é especificado sobre o mapeamento entre um dado valor do domínio e a sua imagem. A noção de tipo, como normalmente existente em linguagens de programação, não define propriedades de termos, que especifiquem *o quê* sem necessariamente definir *como*.

só é detectado se a execução do programa constitui um teste para o caso correspondente ao erro de tipo. Como um exemplo simples, considere a expressão:

$$\texttt{if } E \texttt{ then 1 else 1 + } \text{“1”}$$

No caso de checagem em tempo de execução, o erro de tipo contido nesta expressão só é detectado se a avaliação da sub-expressão $E$ resulta no valor *falso*.

Podem então ocorrer casos em que um erro de tipo não é detectado, mesmo depois de um número grande de testes do programa. Esse erro poderá ocorrer, inesperadamente, em um passo de execução não testado anteriormente, com conseqüências potencialmente desastrosas. No caso de checagem em tempo de compilação, ao contrário, todos os erros de tipo serão sempre detectados.

É claro que existem formas de erro que não são erros de tipo. No entanto, os argumentos apresentados acima, são importantes devido à grande freqüência de erros de tipo usualmente cometidos durante uma tarefa de programação. Em face dos argumentos apresentados, o estudo de tipos em linguagens de programação tornou-se de grande importância, no sentido de sua influência sobre o projeto e a definição de linguagens de programação e, portanto, sobre o desenvolvimento de software em geral.

Apesar da similaridade entre as noções de tipo em matemática e em computação, existem algumas diferenças entre estes dois conceitos. Em primeiro lugar, a finalidade é diferente. Em computação, a noção de tipos é motivada pelos fatores apresentados acima: estruturação, clareza e eficiência de programas, e detecção de erros; em matemática, o propósito é o de evitar inconsistências lógicas. Outra diferença relevante é o fato de que tipos em linguagens de programação são definidos para objetos cuja avaliação pode não terminar, o que não ocorre em matemática; por exemplo, pode-se definir recursivamente uma variável *inteira* `v`, da seguinte forma:

$$\texttt{v} = \texttt{v} + 1$$

A variável `v` não pertenceria ao tipo (matemático) dos inteiros. A interpretação matemática para definições recursivas foi estabelecida a partir da *teoria de domínios* [Sco72, Sco76, Sto77].

# 2 Sistemas de Tipos

A definição de uma linguagem de programação serve a diferentes grupos de pessoas: projetistas, implementadores (de compiladores, interpretadores ou editores com recursos específicos para edição de programas) e programadores. Tal definição deve especificar tanto a sintaxe quanto a semântica da linguagem.

A definição da sintaxe de uma linguagem determina quais seqüências de símbolos são frases (válidas), e como frases podem ser combinadas de modo a formar outras frases. Aspectos dinâmicos, relativos à execução de programas, não são considerados. Exemplos de frases de programas são literais, variáveis, declarações, expressões, comandos, programas etc.

As condições que determinam se uma seqüência de símbolos é uma frase podem ser expressas através de regras *livres de contexto* e de regras *sensíveis ao contexto*. Como o nome indica, regras livres de contexto são aquelas em que as condições para a construção de novas frases podem ser especificadas sem considerar o contexto no qual essas frases ocorrem. Regras sensíveis ao contexto, ao contrário, levam em conta o contexto em que as frases ocorrem, na definição das condições para a construção de novas frases.

A maioria das condições que definem a sintaxe de uma linguagem (em geral, em número muito grande) pode ser especificada através de regras livres de contexto. Alguns exemplos simples são: uma *variável* é uma seqüência de símbolos que começa com uma letra e é seguida por uma seqüência de letras ou dígitos ou o símbolo “_”; uma *expressão* é uma variável, uma lambda-abstração ou uma aplicação; uma *aplicação* é uma expressão seguida de outra expressão (separadas por pelo menos um caractere delimitador); uma *lambda-abstração* é uma seqüência de símbolos iniciada pelo símbolo “$\lambda$”, seguido de uma variável, depois do símbolo “.” e, em seguida, de uma expressão.

São exemplos ilustrativos de condições sensíveis ao contexto: 1) toda variável deve ser declarada antes de ser usada; 2) uma variável que ocorre em uma expressão $e$ tem que ocorrer, anteriormente, após o símbolo $\lambda$ de alguma lambda-abstração que tem $e$ como sub-expressão; 3) em um comando de atribuição, o tipo da expressão (que ocorre no lado direito desse comando) tem que ser igual ao tipo da variável (que ocorre no lado esquerdo desse comando).

Regras sensíveis ao contexto podem ser *regras de escopo* e *regras de tipo*. Uma regra de escopo permite associar um nome a uma definição, especificando o escopo dessa definição. Os exemplos 1) e 2) acima são exemplos de regras de escopo. Regras de tipo determinam o tipo de cada expressão da linguagem, possibilitando garantir que cada operação tenha operandos de tipo apropriado. Por exemplo, supondo que `Integer` e `Bool` são tipos pré-definidos em uma dada linguagem, e que a operação de adição, denotada por `+`, é definida apenas sobre operandos de tipo `Integer`, uma expressão como `True + x`, que envolve a constante `True`, de tipo `Bool`, não é sintaticamente válida.

Existem diversas maneiras distintas de se definir a sintaxe de linguagens de programação (veja por exemplo [Ten81, Wat91, Mit96]). O uso de um formalismo baseado em lógica, constituído por axiomas e regras de inferência, possibilita definir, simultaneamente, tanto condições livres de contexto quanto condições sensíveis ao contexto. Tal formalismo tem sido cada vez mais usado, principalmente na definição e estudo de modelos/núcleos de linguagens de programação.

A idéia básica desses sistemas formais é estabelecer regras de formação das frases da linguagem, a partir de suas subfrases, levando em conta propriedades das construções envolvidas na formação dessas frases. Como as propriedades consideradas são, tipicamente, *tipos* de expressões, tais sistemas são denominados *sistemas de tipos*.

Sistemas de tipos de linguagens de programação são usualmente apresentados sob a forma de cálculo de seqüentes [Gal86], como descrito a seguir, podendo também ser apresentados sob a forma de um sistema de dedução natural [Pra65].

Os tipos de expressões atômicas são definidos por meio de axiomas de tipo da forma

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

Esse axioma define que a expressão atômica $e$ tem tipo $\sigma$ no *contexto de tipos* $\Gamma$.

Um contexto de tipos mantém informação sobre os tipos das variáveis que podem ser usadas (i.e. que foram declaradas), possibilitando detectar se uma expressão é bem tipada, com base nos tipos das variáveis livres que ocorrem nessa expressão. Por exemplo, verifica-se se a expressão `x + 1` é bem tipada, em um determinado contexto, examinando se a variável `x` tem tipo inteiro nesse contexto. É natural representar o contexto como um conjunto de pares $x : \sigma$, de variáveis e seus respectivos tipos:

$$\Gamma = \{x_1 : \sigma_1, \ldots, x_k : \sigma_k\}$$

A coleção das variáveis $x_1, \ldots, x_k$ em $\Gamma$ é indicada por $dom(\Gamma)$.

Em geral, exige-se que nenhuma variável ocorra mais de uma vez em um dado contexto. Isso significa que cada variável só pode ser ligada a uma única definição, em um dado escopo. Essa condição pode ser eliminada em sistemas de tipos que suportam *sobrecarga* (*overloading*) de um nome com diferentes definições, dadas por expressões com tipos distintos (veja [CF99a]). A forma geral de uma *regra de inferência* de um sistema de tipos é:

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

onde as *fórmulas* $\Gamma_i \vdash e_i : \sigma_i$, para $i = 1, \ldots, n$, são as hipóteses (ou premissas) da regra e $\Gamma \vdash e : \sigma$ é a sua conclusão. Intuitivamente, esta regra diz que, se cada expressão $e_i$ tem tipo $\sigma_i$ no contexto $\Gamma_i$, para $i = 1, \ldots, n$, então $e$ tem tipo $\sigma$ no contexto $\Gamma$.

Uma expressão $e$ é dita *bem tipada* em um dado contexto $\Gamma$, com relação a um sistema de tipos, se existe uma derivação de $\Gamma \vdash e : \sigma$, para algum $\sigma$, obtida mediante os axiomas e regras desse sistema de tipos.

Ao longo desse texto, apresentamos alguns exemplos de definições de sistemas de tipos, que suportam características de tipos usados em linguagens de programação modernas. Alguns tópicos de pesquisa mais recentes, como *tipos dependentes* [ML78, ML84, C$^+$86, CH88, Mac86], *tipos interseção* [Jim96], *polimorfismo impredicativo* [Gir71, Gir89, Rey74], e trabalhos relacionados com sistemas de módulos [HL94, Mac85, MTH90, Mac86, MMM91], não serão abordados.

Vale observar que sistemas formais podem definir não apenas tipos, mas também outras propriedades de construções de linguagens de programação.

## 2.1 Tipos e Significados

Programas bem tipados, assim como os tipos usados nesses programas, possuem significado. Nessa seção, abordamos métodos formais usados na definição da semântica de linguagens de programação. Duas diferentes abordagens são usualmente empregadas:

- Denotacional — associa a cada frase da linguagem um significado, como um elemento de um domínio semântico escolhido para a interpretação da linguagem [Sto77, Gor79, Mos90]. Formalmente, uma semântica denotacional pode ser definida como um homomorfismo, entre uma *álgebra de termos* (álgebra inicial sobre a assinatura de tipos e símbolos da linguagem) e uma *álgebra semântica* (sobre a mesma assinatura, cujos "carriers" são os domínios semânticos correspondentes aos valores dos termos da linguagem) [Ast91].

- Operacional — o significado de cada frase é definido a partir de uma relação de transição, apresentada sob a forma de um sistema de inferência [Plo81]. Para ser caracterizado como operacional, o sistema de inferência deve ser finitário, isto é, cada regra ou axioma deve ter um conjunto finito de hipóteses.

### 2.1.1 Semântica Denotacional

Em uma semântica denotacional, o significado de termos e frases de uma linguagem é definido por meio de uma função que associa, a cada termo, um elemento de um domínio matemático adequado para a interpretação da linguagem. Esse elemento é chamado *denotação* do termo. A cada tipo da linguagem é também associado um significado: a denotação de um tipo é um conjunto (possivelmente munido de alguma estrutura) de elementos do domínio de interpretação dos termos da linguagem.[3]

A denotação de um termo que possui variáveis livres depende do significado atribuído a essas variáveis. [4] Por isso, a função semântica é parametrizada por outra função, denominada *ambiente*, que associa a cada variável um elemento do domínio. A denotação de um termo $e$, em um ambiente $\rho$, é representada por $[\![e]\!]\rho$.

Da mesma forma, em sistemas de tipo que incluem variáveis de tipo,[5] o significado de uma expressão de tipo depende do significado atribuído a suas variáveis de tipo livres. Na apresentação que segue, supomos que o ambiente $\rho$ também atribui significado a variáveis de tipo, ou seja, $\rho$ é uma função que associa variáveis de expressão a valores do domínio, e associa subconjuntos de valores do domínio a variáveis de tipo. A denotação de um tipo $\sigma$, em um ambiente $\rho$, é representada por $[\![\sigma]\!]\rho$.

Dois termos $e_1$ e $e_2$ são *denotacionalmente equivalentes*, $e_1 \stackrel{den}{=} e_2$, se, para todo ambiente $\rho$, temos

$$[\![e_1]\!]\rho = [\![e_2]\!]\rho$$

O significado dos termos de uma linguagem é definido, de forma mais natural, por indução sobre a estrutura de provas de derivação de tipos, no sistema de tipos da linguagem. Isso garante que se atribua significado apenas a termos bem tipados.[6] A semântica de uma linguagem é então definida da seguinte forma:

1. Para cada termo atômico $e$, tal que $\Gamma \vdash e : \sigma$ é provável, o significado de $e$ é definido em função de um ambiente $\rho$ e do próprio termo $e$. Usualmente definimos:

   $$[\![\Gamma, x : \sigma \vdash x : \sigma]\!]\rho = \rho(x)$$

   isto é, o significado da variável livre $x$ é o valor que lhe é atribuído pelo ambiente $\rho$.

2. O significado de um termo composto $e$, tal que $\Gamma \vdash e : \sigma$ é provável, a partir das provas de $\Gamma_1 \vdash e_1 : \sigma_1, \ldots, \Gamma_n \vdash e_n : \sigma_n$, onde $e_1, \ldots, e_n$ são os subtermos de $e$, é obtido em função dos significados de $e_1, \ldots, e_n$.

Dizemos que um ambiente $\rho$ *satisfaz* um contexto $\Gamma$ se $\rho(x) \in [\![\sigma]\!]\rho$, para todo $x : \sigma \in \Gamma$, isto é, se para todo $x$ de tipo $\sigma$ em $\Gamma$, o valor atribuído a $x$ pelo ambiente $\rho$ é um elemento do conjunto denotado por $\sigma$.

[3] Em sistemas de tipos em que tipos podem ser usados como valores, os conjuntos denotados por tipos devem ser, eles próprios, elementos do domínio de interpretação da linguagem.

[4] Veja, na seção 3, a definição de ocorrências livres e ligadas de variáveis em uma expressão.

[5] Veja, nas seções 4 e 6, a descrição de sistemas de tipos polimórficos, que incluem variáveis de tipo e tipos quantificados sobre essas variáveis.

[6] No caso de linguagens dinamicamente tipadas, o domínio de interpretação dos termos deve incluir um elemento especial, como denotação de erro dinâmico de tipo.

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

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

Figura 1: Regras de Inferência

Como a semântica denotacional de uma linguagem é definida por indução sobre a estrutura de provas em seu sistema de tipos, ocorre o problema de *coerência*: diferentes derivações de tipo para um mesmo termo devem possuir o mesmo significado (ou um significado relacionado). Em outras palavras, o significado de um termo $e$ deve ser independente de uma escolha particular de derivação de tipo para $e$, dependendo apenas do próprio termo $e$ e dos valores atribuídos às suas variáveis livres pelo ambiente. Essa propriedade, denominada *coerência*, é formalmente enunciada a seguir:

**Propriedade 1 (Coerência)** Sejam $\Delta$ e $\Delta'$ derivações de tipo com conclusões $\Gamma \vdash e : \sigma$ e $\Gamma' \vdash e : \sigma'$, respectivamente, e suponha que toda variável $x$, livre em $e$, possui o mesmo tipo em $\Gamma$ e em $\Gamma'$. Se $\rho$ é um ambiente que satisfaz $\Gamma$ e $\Gamma'$, então

$$[\![\Gamma \vdash e : \sigma]\!]\rho = [\![\Gamma' \vdash e : \sigma']\!]\rho$$

onde os significados são definidos utilizando $\Delta$ e $\Delta'$ respectivamente.

Alguns sistemas de tipos não satisfazem esta propriedade, dificultando a definição da semântica de termos e tipos da linguagem. Por exemplo, considere um sistema de tipos que contenha as regras da Figura 1 e no qual contextos de tipos podem conter várias ocorrências de uma mesma variável, com diferentes tipos.[7] Suponha um contexto $\Gamma$ que contenha as seguintes atribuições de tipo: $\texttt{g} : \texttt{Int} \to \texttt{Int}$, $\texttt{g} : \texttt{Float} \to \texttt{Int}$, $\texttt{one} : \texttt{Int}$ e $\texttt{one} : \texttt{Float}$. A partir das regras da Figura 1, podemos obter as seguintes derivações para a expressão `g one`, nesse contexto $\Gamma$:

$$\frac{\Gamma \vdash \texttt{g}\colon \texttt{Float}\to\texttt{Int} \qquad \Gamma \vdash \texttt{one}\colon \texttt{Float}}{\Gamma \vdash \texttt{g one}\colon \texttt{Int}} \qquad (1)$$

$$\frac{\Gamma \vdash \texttt{g}\colon \texttt{Int}\to\texttt{Int} \qquad \Gamma \vdash \texttt{one}\colon \texttt{Int}}{\Gamma \vdash \texttt{g one}\colon \texttt{Int}} \qquad (2)$$

O significado de $\Gamma \vdash \texttt{g one}\colon \texttt{Int}$, em um ambiente $\rho$ que satisfaz $\Gamma$, é dado, indutivamente, em função de $[\![\Gamma \vdash \texttt{g} : \texttt{Float} \to \texttt{Int}]\!]\rho$ e de $[\![\Gamma \vdash \texttt{one} : \texttt{Float}]\!]\rho$, na derivação (1), e em função de $[\![\Gamma \vdash \texttt{g} : \texttt{Int} \to \texttt{Int}]\!]\rho$ e de $[\![\Gamma \vdash \texttt{one} : \texttt{Int}]\!]\rho$, na derivação (2). Como estes significados podem ser diferentes, é possível que $\texttt{g one} : \texttt{Int}$ tenha significados diferentes, dependendo da derivação escolhida.

A concordância entre o sistema de tipos e a semântica denotacional dos termos e tipos desse sistema é dada pela propriedade de *correção*, a seguir. Essa propriedade expressa que o significado de uma expressão bem tipada deve ser um elemento do conjunto denotado pelo seu tipo.

**Propriedade 2 (Correção)** Se $\Gamma \vdash e : \sigma$ é provável, e $\rho$ é um ambiente que satisfaz $\Gamma$, então $[\![\Gamma \vdash e : \sigma]\!]\rho \in [\![\sigma]\!]\rho$.

### 2.1.2 Semântica Operacional

Em uma semântica operacional, o significado de um termo é usualmente outro termo, em *forma normal*, isto é, que não pode ser simplificado pela relação de transição. A relação de transição

[7] Como dito anteriormente (página 3), isso pode ocorrer no caso de sistemas de tipos que suportam sobrecarga de um nome com diversas definições, de diferentes tipos.

(ou *redução*) captura a noção de *um passo de avaliação* de termos da linguagem. Uma regra de transição, escrita na forma $e \Rightarrow e'$, define que o termo $e$ avalia, ou *reduz*, para o termo $e'$, em um passo. Nesse caso, dizemos que $e$ é um *redex* e $e'$ é o seu *reduto*. Escrevemos $e \Rightarrow^* e'$, se $e$ avalia para $e'$ em zero ou mais passos. Formalmente, uma expressão $e$ é uma *forma normal* se não existe nenhum termo $e'$, tal que $e$ pode ser reduzido para $e'$. Se $e \Rightarrow^* e'$, e $e'$ é uma forma normal, dizemos que $e'$ é uma forma normal de $e$.

Note que podem existir diferentes seqüências de redução para uma dada expressão. Por exemplo, duas diferentes seqüências de redução para a expressão `square` (2+5), onde `square` é definido por `square x = x * x`, seriam:

`square` (2+5) $\Rightarrow$ (2+5) $*$ (2+5) $\Rightarrow$ 7 $*$ 7 $\Rightarrow$ 49

`square` (2+5) $\Rightarrow$ `square` 7 $\Rightarrow$ 7 $*$ 7 $\Rightarrow$ 49

Uma *estratégia de redução* é uma função parcial $F$, de termos em termos, tal que $F(e) = e'$ implica que $e \Rightarrow e'$. $F$ é uma "estratégia" porque é definida com base em uma estratégia particular para a escolha do redex a ser reduzido em cada passo da avaliação da expressão. Por exemplo, na primeira das reduções acima, a estratégia é escolher, em cada passo, o redex mais externo e que ocorre mais à esquerda na expressão.

Dada uma estratégia de redução $F$, o significado de um termo $e$ é dado pela *função parcial de avaliação* $eval_F$, definida indutivamente do seguinte modo:

$$eval_F(e) = \begin{cases} e & \text{se } e \text{ é uma forma normal} \\ e' & \text{se } F(e) = e_1 \text{ e } eval_F(e_1) = e' \end{cases}$$

De modo geral, diferentes escolhas da estratégia de redução podem resultar em funções de avaliação que atribuem significados distintos a uma mesma expressão (mesmo no caso de relações de redução que satisfazem a propriedade de Church-Rosser).[8]

Existem diferentes estratégias de avaliação : avaliação *estrita* (*chamada-por-valor*), avaliação *normal* (*chamada-por-nome*) e avaliação *preguiçosa* (*chamada-por-necessidade*). [9]

A função de avaliação, como definida acima, atribui significado, indistintamente, tanto a termos válidos quanto a termos que não são bem tipados. Por exemplo, o significado da expressão (2+5) é 7, uma vez que (2+5) $\Rightarrow$ 7, e 7 é uma forma normal. Do mesmo modo, o significado da expressão (2+True) é essa própria expressão, uma vez que ela não avalia para nenhuma outra expressão, sendo, portanto, uma forma normal.

É possível distinguir esses dois casos, definindo-se explicitamente a sintaxe livre de contexto do conjunto dos termos em *forma canônica* da linguagem. A motivação de tal distinção é possibilitar que se verifique se o sistema de tipos e a semântica operacional da linguagem estão definidos de forma concordante, a saber, satisfazem as propriedades 3 e 4 definidas a seguir. A função de avaliação parcial $eval_F$ é redefinida como:

$$eval_F(e) = \begin{cases} e & \text{se } e \text{ é uma forma canônica} \\ e' & \text{se } F(e) = e_1 \text{ e } eval_F(e_1) = e' \\ \texttt{erro} & \text{se } e \text{ não é uma forma canônica e} \\ & \text{não existe } e_1 \text{ tal que } F(e) = e_1 \end{cases}$$

Utilizando uma sintaxe adequada para os termos em forma canônica da linguagem, teríamos $eval_F(2+5) = 7$ (como anteriormente), mas $eval_F(2+\texttt{True}) = \texttt{erro}$, uma vez que $(2+\texttt{True})$ não seria uma forma canônica.

Dizemos que dois programas $p_1$ e $p_2$ são *operacionalmente equivalentes* se $eval_F(p_1) = eval_F(p_2)$. Isto significa que, ou $p_1$ e $p_2$ avaliam para uma mesma forma canônica, ou ambos resultam em `erro`, ou $eval_F$ é indefinida para ambos os casos (tanto $p_1$ quanto $p_2$ "entram em *loop*"). Escrevemos, nesse caso, $p_1 \stackrel{op}{=} p_2$.

Para estender a definição de equivalência operacional para termos (e não apenas para programas) é necessário introduzir a noção de contexto.[10] Essa noção é necessária, uma vez que termos

[8] Veja seção 3, página 9.

[9] Uma discussão mais detalhada sobre estratégias de redução, e implicações do uso de cada uma em uma linguagem de programação, pode ser encontrada em [Ten81, Mit96].

[10] Observe que o termo contexto definido na seção anterior refere-se a um *contexto de tipos*, diferentemente do definido aqui, que pode ser visto como um termo incompleto, com um buraco em que outro termo pode ser inserido.

podem conter variáveis livres, cujo significado não é observável, isto é, não é definido pela relação de redução. Um *contexto* $\mathcal{C}[\,]$ é um termo que contém um "buraco" (representado por um par de colchetes), onde outro termo pode ser inserido. Por exemplo:

$$\mathcal{C}_0[\,] \stackrel{def}{=} \lambda x{:}\texttt{Integer}.\, x + [\,]$$

é um contexto tal que, se inserimos nele um termo $e$ qualquer, ele passa a ter a forma[11]

$$\mathcal{C}_0[e] \stackrel{def}{=} \lambda x{:}\texttt{Integer}.\, x + e$$

Na inserção de um termo $e$ em um contexto $\mathcal{C}[\,]$, resultando em $\mathcal{C}_0[e]$, ocorrências livres de variáveis em $e$ podem ser capturadas em $\mathcal{C}_0[e]$, uma vez que a inserção de um termo em um contexto $\mathcal{C}[\,]$ é efetuada sem que as variáveis ligadas nesse contexto sejam renomeadas.

Dois termos $e_1$ e $e_2$ são operacionalmente equivalentes ($e_1 \stackrel{op}{=} e_2$) se, para todo contexto $\mathcal{C}[\,]$ tal que $\mathcal{C}[e_1]$ e $\mathcal{C}[e_2]$ são programas, tem-se que $eval_F(\mathcal{C}[e_1]) = eval_F(\mathcal{C}[e_2])$.

A concordância entre o sistema de tipos e a semântica operacional de uma linguagem é expressa pelas seguintes propriedades:

**Propriedade 3 (Redução preserva tipo)** Se $\Gamma \vdash e : \sigma$ é provável, e $e \Rightarrow e'$ então $\Gamma \vdash e' : \sigma$ é provável.

**Propriedade 4 (Programas bem tipados não contêm erros)** Se $\Gamma \vdash p : \sigma$ é provável, então $eval_F(p) \neq \texttt{erro}$

A primeira propriedade expressa que a relação de transição deve preservar o tipo das expressões. A segunda propriedade expressa que termos válidos (bem tipados) avaliam para formas canônicas da linguagem. A verificação dessa propriedade possibilita a detectar eventuais erros na definição do sistema de tipos (a atribuição de um tipo a uma expressão que não deveria ser tipada), assim como na definição da semântica operacional (a avaliação de um programa bem tipado resultando em `erro`). Uma linguagem que satisfaz essa segunda propriedade é dita *fortemente tipada*.

Deve-se notar também que as semânticas operacional e denotacional de uma linguagem devem ser definidas de forma concordante, isto é, devem satisfazer a seguinte propriedade:

**Propriedade 5 (Adequação computacional)** Seja $p$ um programa e $v$ uma forma canônica. Temos que $p \stackrel{den}{=} v$ se, e somente se, $eval_F(p) = v$.

Para termos arbitrários $e$ e $e'$, o que se espera, em geral, é que, se $e$ e $e'$ são denotacionalmente equivalentes, então $e$ e $e'$ são operacionalmente equivalentes ($\stackrel{den}{=}$ implica $\stackrel{op}{=}$). Se a recíproca é também verdadeira (ou seja, se $\stackrel{op}{=}$ implica $\stackrel{den}{=}$), então a semântica denotacional é dita *totalmente abstrata*. Entretanto, a definição de uma semântica denotacional totalmente abstrata pode ser muito difícil (veja, por exemplo, [Sto88]).

Se as semânticas operacional e denotacional de uma linguagem estão relacionadas pela propriedade de adequação computacional, então a propriedade 4 torna-se um corolário da propriedade 2. De fato, suponha que $\Gamma \vdash p : \sigma$ é provável e que $eval_F(p) = \texttt{erro}$. Pela propriedade de *adequação computacional*, temos $p \stackrel{den}{=} \texttt{erro}$. Mas como `erro` não possui significado denotacional, o mesmo deve acontecer com $p$. Absurdo, pois $[\![\Gamma \vdash p : \sigma]\!]\rho \in [\![\sigma]\!]\rho$, para qualquer $\rho$ que satisfaz $\Gamma$, pela propriedade 2.

## 2.2 Outras propriedades de sistemas de tipos

Além das propriedades mencionadas anteriormente, existem outras propriedades desejáveis para sistemas de tipos.

- Decidibilidade — O sistema de tipos da maioria das linguagens de programação é decidível, isto é, existe um algoritmo tal que, dado um programa $p$, determina se esse programa é bem tipado ou não. Em termos do sistema de tipos, o algoritmo determina se existe uma derivação

[11] A notação de $\lambda$-expressões é descrita na seção 3. Intuitivamente, uma expressão $\lambda x{:}\sigma.\, e$ representa uma função que, ao receber um argumento denotado por $x$, de tipo $\sigma$, fornece como resultado o valor definido pela expressão $e$.

para a fórmula $\Gamma \vdash p : \sigma$, para algum $\sigma$, de acordo com as regras do sistema de tipos, onde $\Gamma$ é um contexto contendo tipagens para os símbolos pré-definidos da linguagem.

Alguns sistemas de tipos de linguagens de programação são indecidíveis, como os das linguagens Quest [CL91] e Cayenne [Aug99]. O sistema de tipos de Cayenne usa *tipos dependentes* [ML78, ML84, Mac86, C$^+$86, CH88], isto é, o tipo de uma expressão pode depender não apenas dos tipos de outras expressões, mas também dessas próprias expressões. Por exemplo, o tipo do resultado de uma função pode depender tanto do tipo do seu argumento como do *valor* desse argumento; o tipo de um componente de um registro pode depender do valor de outros componentes do registro. A indecidibilidade desse sistema de tipos provém da necessidade de testar se dois tipos são (estruturalmente) iguais, o que é indecidível no caso de linguagens com tipos dependentes e sem a propriedade de normalização forte (veja seção 3). O algoritmo de checagem de tipos de Cayenne poderia não terminar, para determinados programas, caso não adotasse um limite para o número de reduções a serem realizadas no processo de verificação de tipos. Com essa restrição, o algoritmo de fato sempre termina, podendo emitir, como resultado da compilação de um determinado programa, uma mensagem indicando que não foi possível realizar com sucesso a checagem de tipos para esse programa.

A argumentação em favor do uso de sistemas de tipos indecidíveis, em geral mais complexos, é baseada em dois aspectos. O primeiro é o de que, em geral, a linguagem compreende um conjunto de frases bem tipadas maior do que o de uma linguagem com um sistema de tipos decidível. O segundo é o de que o número de casos em que, de fato, o algoritmo pára, sem uma verificação de tipo bem sucedida (por ter alcançado o limite estabelecido para o número de reduções) seria muito pequeno. Entretanto, é ainda necessária uma maior experiência com o uso de *tipos dependentes* para que se possa avaliar como se comportam, na prática, sistemas de tipos indecidíveis.

- Unicidade — Um sistema possui a propriedade de unicidade de tipos se, para cada frase $e$ bem formada e cada contexto $\Gamma$, existe um único $\sigma$ tal que $\Gamma \vdash e : \sigma$ é provável. Para sistemas de tipos com subtipagem (polimorfismo), a propriedade de unicidade de tipos é, em geral, substituída pelas propriedades de tipo e tipagem mínima (principal) - veja seções 4 e 5.

- Tipo e tipagem mínima, tipo e tipagem principal — veja seções 4 e 5.

# 3 Tipos Simples

Estudos sobre tipos em linguagens de programação são usualmente desenvolvidos sob o arcabouço do $\lambda$-calculus tipado. O $\lambda$-calculus tipado surgiu a partir do $\lambda$-calculus não tipado, ambos definidos por Church, na década de 1930 [Chu32, Chu36, Chu41]. O $\lambda$-calculus (não tipado) provê um modelo muito simples de avaliação de expressões. Apesar dessa simplicidade, o $\lambda$-calculus é um modelo "universal" de computabilidade, no sentido de que qualquer função recursiva [Kle36], assim como qualquer função computável por uma máquina de Turing [Tur37], pode ser expressa como um termo do $\lambda$-calculus.

Termos do $\lambda$-calculus são expressões, que podem ser: uma *variável*, uma $\lambda$-*abstração*, usada para descrever funções, ou uma *aplicação*, que especifica a aplicação de uma expressão (que denota uma função) a outra expressão (parâmetro real, que denota o argumento).

Uma $\lambda$-abstração é escrita na forma $\lambda x.\, e$, onde $x$ é uma variável e $e$ uma expressão. A variável $x$ é chamada parâmetro (formal) da função, e o *escopo* de $\lambda x$ é definido como sendo $e$, com exceção de possíveis $\lambda$-abstrações em $e$ que também tenham $x$ como parâmetro. Uma ocorrência de um variável $x$ é dita *ligada* em $e$ se ocorre no escopo de $\lambda x$. Caso contrário a ocorrência de $x$ é dita *livre*.

A avaliação da aplicação da função $\lambda x.\, e$ a um argumento $e'$ é definida por uma regra baseada na substituição textual das ocorrências do parâmetro formal $x$ pelo parâmetro real $e'$ na expressão $e$. Essa regra, denominada axioma de ($\beta$)-redução, é representada da seguinte forma:

$$(\lambda x.\, e)\; e' \Rightarrow [e'/x]e$$

A notação $[e'/x]e$ representa a expressão obtida substituindo-se todas as ocorrências livres de $x$ em $e$ pela expressão $e'$.

Como dito anteriormente, pode-se expressar, em $\lambda$-calculus, qualquer função recursiva ou, equivalentemente, qualquer função computável por uma máquina de Turing. Para se ter uma idéia de como o $\lambda$-calculus consegue tal poder de expressão, é interessante considerar como se pode expressar definições recursivas em $\lambda$-calculus, ou seja, utilizando um *operador de ponto fixo*. Um operador de ponto fixo é qualquer expressão `fix` tal que, para toda expressão $e$, tem-se que $\texttt{fix}\ e = e\ (\texttt{fix}\ e)$, ou seja, $\texttt{fix}\ e$ é um ponto fixo da expressão $e$. O operador de ponto fixo pode ser definido em $\lambda$-calculus, por exemplo, pela seguinte expressão:

$$\texttt{fix} \equiv \lambda f.\,(\lambda x.f(xx))(\lambda x.f(xx)) \tag{3}$$

De fato, aplicada a uma expressão qualquer $e$, $\texttt{fix}\ e$ retorna um ponto fixo de $e$:

$$\begin{aligned} \texttt{fix}\, e &= (\lambda f.\,(\lambda x.f(xx))(\lambda x.f(xx)))\, e \\ &\Rightarrow (\lambda x.e(xx))(\lambda x.e(xx)) \\ &\Rightarrow e((\lambda x.e(xx))(\lambda x.e(xx))) \\ &= e\ (\texttt{fix}\ e) \end{aligned}$$

Expressões como `fix` acima introduzem dificuldades na definição do siginificado de expressões do $\lambda$-calculus. Por exemplo: se uma expressão $e$ tem mais de um ponto fixo, qual deles é dado por $\texttt{fix}\, e$? O que significa $\texttt{fix}\, e$, se $e$ não tem ponto fixo (por exemplo, $e \equiv \lambda x.\, x + 1$)? Certamente, um modelo de interpretação para o $\lambda$-calculus deve ser capaz de responder a essas questões.[12] A complexidade de interpretação do operador de ponto fixo é evidenciada pelo fato de sua definição envolver a aplicação de uma variável ($x$) a ela própria. Essa possibilidade pode levar a paradoxos. Considere, por exemplo, a seguinte expressão:

$$\lambda y.\, \texttt{if}\ y\ y = a\ \texttt{then}\ b\ \texttt{else}\ a$$

Se $x$ representa essa expressão, então $xx$ resulta, contraditoriamente, em:

$$\texttt{if}\ x\ x = a\ \texttt{then}\ b\ \texttt{else}\ a$$

A possibilidade de qualquer expressão poder ser aplicada a ela própria requer que o espaço de valores representados por essas expressões seja isomorfo ao espaço de funções sobre esses valores [Sto77, Sco76]. Dessa forma, embora o modelo de execução do $\lambda$-calculus não tipado seja muito simples e poderoso, a complexidade de seus modelos de interpretação, assim como o fato de que linguagens de programação são, em sua maior parte, linguagens tipadas, motivaram o estudo de cálculos baseados no $\lambda$-calculus tipado.

O $\lambda$-calculus tipado simples, proposto por Church [Chu40], considera apenas tipos básicos (por exemplo, `Int`, o tipo dos inteiros, e `Bool`, o tipo dos valores booleanos) e tipos funcionais. O sistema de redução do $\lambda$-calculus tipado simples herda as propriedades fundamentais do sistema de redução do $\lambda$-calculus não tipado: confluência e normalização. A propriedade de *confluência*, também chamada de *propriedade de Church-Rosser*, garante que, se uma expressão $e$ reduz para $e_1$ ou para $e_2$, então existe uma expressão $e'$, tal que $e_1$ reduz para $e'$ e $e_2$ reduz para $e'$. Como conseqüência, se um termo tem uma forma normal, então ela é única. A propriedade de *normalização* garante a existência de uma determinada estratégia de redução para a qual a seqüência de reduções de qualquer expressão resulta em sua forma normal, caso a expressão possua uma forma normal. [13]

A Figura 2 apresenta a sintaxe livre de contexto das expressões do $\lambda$-calculus tipado simples. No $\lambda$-calculus tipado simples, ao contrário do $\lambda$-calculus não tipado, uma expressão não pode ser aplicada a si própria. Assim, não se pode definir o operador de ponto fixo na própria linguagem. Além disso, o $\lambda$-calculus tipado simples possui a propriedade de *terminação* (também chamada de *normalização forte*): toda seqüência de redução de qualquer expressão termina, ou seja, toda expressão possui uma (única) forma normal. [14]

A Figura 3 apresenta o sistema de tipos do $\lambda$-calculus tipado simples.

O axioma (VAR) indica que uma variável tem o tipo especificado na sua "declaração" (isto é, o tipo $\tau$ que ocorre em $\lambda x : \tau.\, e$).

[12] Veja, por exemplo, [Sco72, Sco76, Sto77]. Na interpretação definida por Scott, $\texttt{fix}e$ denota o ponto fixo minimal ("menos definido") da função (contínua) denotada por $e$, no domínio semântico.

[13] Para uma descrição formal e provas dessas propriedades veja, por exemplo, [Bar84].

[14] Para uma descrição formal e prova dessa propriedade veja, por exemplo, [FLO83, Tho91].

| | | | |
|---|---|---|---|
| Tipos | $\tau ::=$ | $b$ | tipo básico (ou primitivo) |
| | | $\tau_1 \to \tau_2$ | tipo funcional |
| Termos | $e ::=$ | $x$ | variável |
| | | $\lambda\, x : \tau . e$ | $\lambda$-abstração |
| | | $e\; e'$ | aplicação |

Figura 2: Sintaxe livre de contexto do $\lambda$-calculus tipado simples

$$x : \tau \vdash x : \tau \tag{VAR}$$

$$\frac{\Gamma \vdash e : \tau}{\Gamma, x : \tau' \vdash e : \tau} \tag{AD VAR}$$

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

$$\frac{\Gamma \vdash e : \tau' \to \tau,\;\; \Gamma \vdash e' : \tau'}{\Gamma \vdash e\; e' : \tau} \tag{APL}$$

Figura 3: Sistema de tipos do $\lambda$-calculus tipado simples

A regra (AD VAR) permite que uma hipótese seja adicionada ao contexto. A notação $\Gamma, x : \tau$ significa $\Gamma \cup \{x : \tau\}$, com a condição de que $x$ não ocorra em $\Gamma$. Isto indica que, se um termo $e$ é bem tipado em um dado contexto, então todas as variáveis livres de $e$ devem ocorrer nesse contexto. Segundo a regra (AD VAR), se um termo $e$ possui tipo $\tau$ em um contexto $\Gamma$, então, para qualquer variável $x$, de tipo $\tau'$, que não ocorra livre em $e$ e não pertença ao domínio de $\Gamma$, podemos formar um novo contexto $\Gamma' = \Gamma, x : \tau'$ tal que $e$ possui tipo $\tau$ em $\Gamma'$.

A regra (ABS) estabelece que se $e : \tau$ é derivável em um contexto $\Gamma'$ onde $x$ tem tipo $\tau'$, então a expressão $\lambda x : \tau' . e$ define uma função de tipo $\tau' \to \tau$ no contexto $\Gamma = \Gamma' - \{x : \tau'\}$.

A regra (APL) permite a aplicação de qualquer função $e$, de tipo $\tau' \to \tau$, a um argumento $e'$ de tipo $\tau'$, produzindo um resultado, $e\; e'$, de tipo $\tau$.

O $\lambda$-calculus tipado simples pode ser estendido de diversas maneiras, introduzindo-se novos tipos básicos ou novos construtores de tipos. Construtores de tipos comumente adicionados são construtores para os tipos produto e soma (união disjunta) e tipos recursivos (veja seção 3.1).

Para servir como modelo para linguagens de programação, o $\lambda$-calculus tipado simples tem que ser estendido com a introdução de operadores de ponto fixo (um para cada tipo funcional $\tau \to \tau'$), de modo a prover definições de funções recursivas. Desta forma, a linguagem torna-se "universal" (toda função recursiva definida sobre os naturais pode ser expressa na linguagem).

O seguinte axioma de tipo é usado na formação de expressões bem tipadas usando operadores de ponto fixo:

$$\Gamma \vdash \mathtt{fix}_\tau : (\tau \to \tau) \to \tau \tag{FIX}$$

A semântica (operacional) de operadores de ponto fixo é definida pelo seguinte axioma de redução:

$$\mathtt{fix}_\tau \Rightarrow \lambda f : \tau \to \tau . f(\mathtt{fix}_\tau\, f) \tag{fix}$$

A introdução de operadores de ponto fixo possibilita definir, como no $\lambda$-calculus não tipado, termos que não possuem forma normal. Também da mesma forma, mesmo que um termo possua uma forma normal, a estratégia de redução adotada é importante para que esse termo seja reduzido para a sua forma normal. Por exemplo, para qualquer expressão $e : \tau \to \tau$, pode-se aplicar repetidas vezes o axioma de redução (fix):

$$\texttt{fix}_\tau\, e \Rightarrow e(\texttt{fix}_\tau e) \Rightarrow e(e(\texttt{fix}_\tau\, e)) \Rightarrow \ldots$$

e, portanto, para qualquer termo $\texttt{fix}_\tau e : \tau$ (que possua ou não forma normal) existe uma seqüência infinita de reduções, determinada por uma estratégia de redução que escolha o redex mais interno em cada passo de redução. Ao contrário do que ocorre com o $\lambda$-calculus não tipado, termos da forma $x\,x$ não podem ser definidos.

## 3.1 Tipo produto, tipo soma e tipos recursivos

Extensões do $\lambda$-calculus tipado simples com tipos produto e soma, e com tipos recursivos, são usadas para modelar diversas construções em linguagens de programação.

### 3.1.1 Produtos cartesianos

Um tipo produto $\tau_1 \times \tau_2$ é um tipo de pares de valores, onde o primeiro componente tem tipo $\tau_1$ e o segundo tem tipo $\tau_2$. As regras de formação e acesso a componentes de valores de tipo produto são apresentadas na Figura 4.

$$\frac{\Gamma \vdash e : \tau_1 \quad \Gamma \vdash e' : \tau_2}{\Gamma \vdash \langle e,\, e' \rangle : \tau_1 \times \tau_2} \qquad \text{(PROD)}$$

$$\frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \texttt{proj}^1_{\tau_1,\tau_2}\, e : \tau_1} \qquad \text{(PROJ1)}$$

$$\frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \texttt{proj}^2_{\tau_1,\tau_2}\, e : \tau_2} \qquad \text{(PROJ2)}$$

Figura 4: Regras de inferência para tipo produto

As regras (PROJ1) e (PROJ2) introduzem as funções de projeção $\texttt{proj}^1_{\tau_1,\tau_2}$ e $\texttt{proj}^2_{\tau_1,\tau_2}$, utilizadas para obter os componentes de um par. A semântica desses operadores é definida pelos seguintes axiomas de redução:

$$\texttt{proj}^1_{\tau_1,\tau_2}\, \langle e, e' \rangle \Rightarrow e$$
$$\texttt{proj}^2_{\tau_1,\tau_2}\, \langle e, e' \rangle \Rightarrow e'$$

Um *registro* é uma tupla cujos componentes (chamados campos) possuem rótulos. A diferença entre tuplas e registros é que, enquanto cada componente de uma $n$-tupla é identificado pela sua posição (de 1 a $n$), os campos de um registro podem ocorrer em qualquer ordem, sendo identificados apenas pelo rótulo associado a cada um deles. O tipo de um registro $(r_1 = e_1, \ldots, r_n = e_n)$ pode ser escrito como $(r_1 : \tau_1, \ldots, r_n : \tau_n)$, onde $r_1, \ldots, r_n$ são rótulos sintaticamente distintos. Escolhendo uma ordenação para os componentes, podemos transformar registros em produtos cartesianos.

### 3.1.2 Somas

A notação $\tau_1 + \tau_2$ é usada para denotar união disjunta dos conjuntos denotados pelos tipos $\tau_1$ e $\tau_2$. Isto significa que um elemento $v$ de tipo $\tau_1 + \tau_2$ pode ser um elemento de tipo $\tau_1$ ($\tau_2$) juntamente com uma indicação de que $v$ é um elemento de tipo $\tau_1$ ($\tau_2$) (veja Figura 5).

A função de injeção $\texttt{inEsq}_{\tau_1,\tau_2}$ recebe um elemento de tipo $\tau_1$ e constrói um elemento de tipo $\tau_1 + \tau_2$. Similarmente, a função de injeção $\texttt{inDir}_{\tau_1,\tau_2}$ recebe um elemento de tipo $\tau_2$ e constrói um elemento de tipo $\tau_1 + \tau_2$.

$$\frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash \texttt{inEsq}_{\tau_1,\tau_2}\ e : \tau_1 + \tau_2} \quad \text{(IN1)}$$

$$\frac{\Gamma \vdash e : \tau_2}{\Gamma \vdash \texttt{inDir}_{\tau_1,\tau_2}\ e : \tau_1 + \tau_2} \quad \text{(IN2)}$$

$$\frac{\Gamma \vdash e : \tau_1 + \tau_2 \quad \Gamma \vdash f : \tau_1 \to \tau_3 \quad \Gamma \vdash g : \tau_2 \to \tau_3}{\Gamma \vdash \texttt{case}\ e\ f\ g : \tau_3} \quad \text{(ELIM)}$$

Figura 5: Regras de inferência para tipo soma

A regra de eliminação (ELIM) caracteriza o uso correto de expressões cujo tipo é uma soma. Intuitivamente, `case` $e\,f\,g$ inspeciona o tipo de $e$ e aplica $f$ se $e$ possui tipo $\tau_1$, ou $g$ se $e$ possui tipo $\tau_2$.

A semântica desses operadores é definida pelos seguintes axiomas de redução:

$$\texttt{case}\ (\texttt{inEsq}_{\tau_1,\tau_2}\ e)\ f\ g \Rightarrow f$$
$$\texttt{case}\ (\texttt{inDir}_{\tau_1,\tau_2}\ e)\ f\ g \Rightarrow g$$

### 3.1.3 Tipos Variantes

Um *tipo variante* é uma forma de soma rotulada, possuindo a mesma relação com tipos soma que registros possuem com tipos produto. Um tipo variante que define uma soma de tipos $\tau_1, \ldots, \tau_n$ pode ser escrito como $[r_1 : \tau_1, \ldots, r_n : \tau_n]$, onde $r_1, \ldots, r_n$ são rótulos sintaticamente distintos. Como para registros, a ordem não é importante.

Intuitivamente, um elemento de um tipo variante $[r_1 : \tau_1, \ldots, r_n : \tau_n]$ é um elemento de um dos tipos $\tau_i$, para $1 \leq i \leq n$, rotulado por $r_i$.

Em SML [Har93], um tipo variante é definido através da declaração de um *datatype*, como a seguir:

$$\textbf{datatype}\ \texttt{dt} = k_1\ \textbf{of}\ \tau_1 \mid \ldots \mid k_n\ \textbf{of}\ \tau_n$$

$k_1, \ldots, k_n$ são chamados de *construtores de valores* do tipo `dt`. Dado um valor $v_i$ de tipo $\tau_i$, $k_i\,v_i$ constrói um valor de tipo `dt`. A parte "**of** $\tau_i$" pode estar ausente (caso em que, informalmente, $k_i$ é um construtor que não requer parâmetro). Por exemplo, a seguinte declaração define um tipo de árvores binárias com valores inteiros nos nodos (ramos):

```
datatype arvore = Folha | Nodo of arvore * int * arvore
```

Uma função sobre um datatype é declarada através de *casamento de padrões*. Por exemplo, a função `tamanho` definida sobre o tipo `arvore`:

```
fun tamanho Folha = 0
  | tamanho (Nodo t1 n t2) = 1 + tamanho t1 + tamanho t2;
```

determina o número de nodos de uma árvore. Se a árvore é da forma `Folha`, a primeira cláusula da definição é utilizada. Caso contrário, utiliza-se a segunda.

Datatypes e definições de função por casamento de padrão provêem uma forma elegante de uso de tipos variantes. De fato, definições de funções por casamento de padrão podem ser vistas como formas abreviadas de definições por análise de casos. Em um sistema de tipos, a construção e o acesso a componentes de valores de datatypes são baseados nas regras básicas de formação e acesso a valores de tipo soma (Figura 5). Essas regras de tipo não permitem acesso incorreto a componentes de valores de tipos variantes, como é possível, por exemplo, no caso de acesso a componentes de registros variantes em Pascal [WSH77].

### 3.1.4 Tipos Recursivos

A definição do tipo de dado `arvore` apresentada anteriormente é recursiva, uma vez que `arvore` ocorre no corpo de sua definição. Em geral, um tipo de dado em ML

$$\textbf{datatype}\ \texttt{dt} = k_1\ \textbf{of}\ \tau_1 \mid \ldots \mid k_n\ \textbf{of}\ \tau_n$$

é recursivo se o tipo declarado `dt` ocorre em $\tau_i$, para algum $1 \leq i \leq n$.

Assim como ML, várias linguagens de programação admitem definição de tipos recursivos. Podemos adicionar tipos definidos recursivamente em uma linguagem baseada em $\lambda$-calculus tipado simples pela adição de variáveis de tipo ($\alpha$) à sintaxe de expressões de tipo, e pela inclusão de uma nova forma de expressão de tipo, $\mu\alpha.\tau$. Intuitivamente, $\mu\alpha.\tau$ denota o menor tipo que satisfaz a equação

$$\alpha = \tau$$

onde $\alpha$ pode ocorrer em $\tau$.

Satisfazer a equação $\alpha = \tau$ significa encontrar um tipo $\mu\alpha.\tau$ *isomórfico* ao tipo $\tau$. Como $\alpha = \tau$ implica que $\alpha = \tau[\tau/\alpha]$ (onde $\tau[\tau/\alpha]$ denota a substituição por $\tau$ de todas as ocorrências livres de $\alpha$ em $\tau$), a solução procurada é tal que

$$\mu\alpha.\tau \cong \tau[\mu\alpha.\tau/\alpha]$$

Este isomorfismo é estabelecido associando-se a tipos recursivos as seguintes regras, que permitem a conversão de um termo de tipo $\mu\alpha.\tau$ para o tipo $\tau[\mu\alpha.\tau/\alpha]$ e vice-versa:

$$\frac{\Gamma \vdash e : \tau[\mu\alpha.\tau/\alpha]}{\Gamma \vdash \texttt{fold}\ e : \mu\alpha.\tau} \qquad \text{(FOLD)}$$

$$\frac{\Gamma \vdash e : \mu\alpha.\tau}{\Gamma \vdash \texttt{unfold}\ e : \tau[\mu\alpha.\tau/\alpha]} \qquad \text{(UNFOLD)}$$

Figura 6: Regras para o $\lambda$-calculus com tipos recursivos

As funções `fold` e `unfold` são inversas uma da outra, isto é:

$$\texttt{unfold}(\texttt{fold}(e)) = e \quad \text{e} \quad \texttt{fold}(\texttt{unfold}(e')) = e'$$

Quando utilizados juntamente com tipos funcionais, tipos recursivos são muito expressivos. Em particular, podemos traduzir termos do $\lambda$-calculus não tipado como termos do $\lambda$-calculus tipado com tipos recursivos, utilizando o tipo recursivo $\mu\alpha.\alpha \to \alpha$.

# 4 Polimorfismo

O $\lambda$-calculus tipado simples possibilita definir apenas funções *monomórficas*, ou seja, funções que podem ser aplicadas a argumentos de um único tipo. Isso requer, por exemplo, que se tenha que definir uma função identidade, de tipo $\tau \to \tau$, para cada tipo $\tau$, embora a definição dessa função seja a mesma para valores de qualquer tipo.

Em algumas linguagens, tais como Standard ML [MHT87] (ou simplesmente ML), podem ser definidas funções *polimórficas*, isto é, funções que operam uniformemente sobre argumentos de vários tipos. O mecanismo utilizado para a definição de funções polimórficas em ML, constitui, em sua essência, uma extensão simples e elegante do $\lambda$-calculus tipado simples.

## 4.1 core-ML

No sistema de tipos de ML, tipos são divididos em dois universos: *monomórficos* e *polimórficos* (Figura 7). Tipos monomórficos são exatamente os tipos do $\lambda$-calculus tipado simples, exceto que

| | | | |
|---|---|---|---|
| Tipos monomórficos | $\tau ::=$ | $\alpha$ | variável de tipos |
| | | $\tau_1 \to \tau_2$ | tipos funcionais |
| Tipos polimórficos | $\sigma ::=$ | $\tau$ | |
| | | $\forall\alpha.\,\sigma$ | tipo quantificado |

Figura 7: Tipos de core-ML

variáveis de tipo são também incluídas. Tipos polimórficos incluem os tipos monomórficos e *tipos quantificados* $\forall\alpha.\,\sigma$.

O símbolo $\forall$ é denominado *quantificador universal*. A variável de tipo $\alpha$ é ligada na expressão $\forall\alpha.\,\sigma$. Se uma expressão $e$ possui o tipo $\forall\alpha.\sigma$, então, $e$ pode ser usada como uma expressão de tipo $\sigma[\tau/\alpha]$, para qualquer tipo simples $\tau$ (veja regra (INST) na Figura 9).

A sintaxe livre de contexto de expressões de core-ML (usada para abstrair os conceitos fundamentais da linguagem [Mil84]) é apresentada na Figura 8 e um sistema de tipos para core-ML é apresentado na Figura 9.

| | | | |
|---|---|---|---|
| Termos | $e ::=$ | $x$ | variável |
| | | $\lambda\, x.\, e$ | função |
| | | $e\ e'$ | aplicação |
| | | $\texttt{let}\ x = e'\ \texttt{in}\ e$ | declaração polimórfica |

Figura 8: Sintaxe livre de contexto de core-ML

As expressões de core-ML são *implicitamente tipadas*, isto é, ML é uma linguagem tipada, embora não requeira anotações explícitas de tipos.[15] O tipo de cada expressão válida é determinado por um algoritmo de *inferência de tipos* (veja seção 4.2).

As regras do sistema de tipos de core-ML incluem as regras usuais do sistema de tipos do $\lambda$-calculus tipado simples, além de três outras regras. A regra (LET) define a formação correta de expressões que utilizam a declaração polimórfica `let`. Uma expressão $\texttt{let}\ x = e'\ \texttt{in}\ e$ liga $x$ à expressão $e'$, de tipo polimórfico, em $e$. Se $e'$ tem tipo monomórfico, então $\texttt{let}\ x = e'\ \texttt{in}\ e$ corresponde à expressão $(\lambda x.\, e)\, e'$. Essa correspondência não é válida, entretanto, caso $e'$ tenha tipo polimórfico, uma vez que, em ML, o argumento de uma $\lambda$-abstração apenas pode ter tipo monomórfico (veja regra (ABS)).

Note que sistemas de tipos polimórficos que incluem a regra (INST) não possuem a propriedade de unicidade de tipos: a regra (INST) possibilita a derivação de diferentes tipos para uma mesma expressão $e$, tal como ilustra o exemplo a seguir. Considere a seguinte expressão, em que a identidade polimórfica é definida:

$$\texttt{let}\ \ id = \lambda x.\, x\ \ \texttt{in}\ \ id\, id$$

No sistema de tipos de core-ML podemos deduzir $\emptyset \vdash \lambda x.x : \forall\alpha.\alpha \to \alpha$ e, portanto, deduzimos tanto $id : \forall\alpha.\alpha \to \alpha \vdash id : \alpha \to \alpha$ como, pela regra (INST), $id : \forall\alpha.\alpha \to \alpha \vdash id : (\alpha \to \alpha) \to (\alpha \to \alpha)$.

Como sistemas de tipos polimórficos possibilitam derivar diferentes tipos para uma mesma expressão bem tipada, em um dado contexto, é necessário caracterizar o tipo que deve ser inferido para essa expressão, nesse contexto, pelo algoritmo de inferência de tipos da linguagem. Esse tipo é denominado *tipo principal*.

[15] É também possível definir um sistema de tipos para uma versão explicitamente tipada de core-ML, denominada core-XML, que é equivalente ao sistema de tipos de core-ML [MH93].

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

$$\frac{\Gamma \vdash e : \sigma}{\Gamma \vdash e : \forall\alpha.\,\sigma} \quad (\alpha \text{ não ocorre livre em } \Gamma) \qquad \text{(GEN)}$$

$$\frac{\Gamma \vdash e : \forall\alpha.\sigma}{\Gamma \vdash e : \sigma[\tau/\alpha]} \qquad \text{(INST)}$$

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

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

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

Figura 9: Sistema de tipos de core-ML

### Tipo principal e tipagem principal

Intuitivamente, o *tipo principal* $\sigma$ para uma expressão $e$, em um dado contexto $\Gamma$, é aquele que representa todos os possíveis tipos $\sigma'$, tais que $\Gamma \vdash e : \sigma'$ é provável. Esse conceito é formalmente definido em termos de uma relação de instanciação entre tipos, $\sqsubseteq$, definida a seguir.

Uma substituição de tipos é uma função (finita) de variáveis de tipo em tipos simples. $S\sigma$ representa o tipo obtido substituindo-se toda variável de tipo $\alpha$, que ocorre livre em $\sigma$, por $S(\alpha)$. Definimos:

$$\begin{array}{ll} \tau \sqsubseteq \tau' & \text{se } S\tau = \tau' \text{ para alguma substituição } S \\ \forall\alpha.\,\sigma \sqsubseteq \tau' & \text{se } \sigma \sqsubseteq \tau', \\ \forall\alpha.\,\sigma \sqsubseteq \forall\alpha'.\,\sigma' & \text{se } \sigma \sqsubseteq \sigma' \end{array}$$

Se $\sigma \sqsubseteq \sigma'$ dizemos que $\sigma'$ é uma *instância* de $\sigma$, ou que $\sigma$ é *mais geral* do que $\sigma'$.

Note que, se $\Gamma \vdash e : \sigma$ é provável então $\Gamma \vdash e : \sigma'$ é provável, para todo $\sigma'$ que é uma instância de $\sigma$ (pela regra INST). Portanto, uma definição adequada para tipo principal de uma expressão $e$, em um contexto $\Gamma$, é o tipo *mais geral* que pode ser derivado para $e$ nesse contexto.

Um conceito mais geral é o de *tipagem principal* para uma expressão $e$, que constitui um par $(\Gamma, \sigma)$, tal que $\Gamma \vdash e :\ \sigma$ é provável, e $\Gamma \vdash e :\ \sigma$ representa todas as possíveis fórmulas prováveis $\Gamma' \vdash e :\ \sigma'$, para a expressão $e$. Essa noção é importante quando se considera a compilação em separado de módulos de programas, tal como discutido em [Jim96]. Intuitivamente, uma tipagem principal $(\Gamma, \sigma)$, para uma expressão $e$, é tal que $\Gamma$ *requer menos* das variáveis $x \in dom(\Gamma)$ e o tipo $\sigma$ *provê mais* do que qualquer outro possível tipo para $e$, onde as noções de *provê mais* e *requer menos* são definidas em termos da relação de instanciação de tipos $\sqsubseteq$.

## 4.2 Inferência de tipos

Anotações de tipo podem ser muito úteis, pelas razões expostas na seção 1, mas não constituem parte necessária de programas. Para permitir a omissão de anotações de tipo, caso se julgue conveniente, linguagens de programação utilizam mecanismos de *inferência de tipos*, isto é, mecanismos para determinar automaticamente o tipo de expressões que ocorrem em um programa, em tempo de compilação.

Toda linguagem tipada faz uso de inferência de tipos, uma vez que, em geral, utiliza anotações de tipos apenas em determinados pontos do programa (nas declarações de variáveis ou funções). Os tipos das demais expressões são inferidos, de acordo com as informações pré-definidas ou providas por declarações. A linguagem ML provê um mecanismo de inferência de tipos que possibilita a total omissão de anotações de tipo em programas.

Para dar uma noção do mecanismo de inferência de tipos de ML, considere, por exemplo, a seguinte expressão:

$$\texttt{fn}\ x\ \Rightarrow\ x+1$$

Ao analisar a expressão $x+1$, o compilador atribui tipo $\alpha$ à variável $x$ (onde $\alpha$ é uma variável de tipo). A ocorrência do operador "+" não determina, ainda, o tipo de $x$, já que "+" é definido tanto para argumentos inteiros quanto para argumentos reais. Com a ocorrência da constante inteira 1, o compilador reconhece que trata-se de uma soma de argumentos inteiros e infere, portanto, que $x$ tem tipo `int`. Conseqüentemente, o tipo inferido para a expressão acima é $\texttt{int} \to \texttt{int}$. Uma apresentação bastante didática do algoritmo de inferência de tipos de ML pode ser encontrada em [DB96].

# 5 Subtipagem

Subtipagem é toda relação de preordem (i.e. uma relação reflexiva e transitiva) sobre tipos (estritamente, sobre expressões de tipo) para a qual vale a seguinte regra de substituição entre expressões:

> Se $\sigma' <: \sigma$ então expressões de tipo $\sigma'$
> podem ser usadas no lugar de expressões de tipo $\sigma$

"No lugar de expressões de tipo $\sigma$" significa, em termos mais precisos, o mesmo que "em qualquer contexto em que expressões de tipo $\sigma$ podem ser usadas, de modo a gerar frases bem tipadas." A notação $\sigma' <: \sigma$ expressa que $\sigma'$ é um *subtipo* de $\sigma$. Nesse caso, $\sigma$ é dito um *supertipo* de $\sigma'$.

Na maioria das linguagens de programação tipadas (notadamente em linguagens mais antigas), a checagem de tipos é baseada em uma relação de compatibilidade entre tipos que é, na maioria dos casos, uma relação de *equivalência*. Nessas linguagens, a regra de substituição entre expressões é freqüentemente bidirecional: uma expressão de um tipo $\sigma'$ pode ser usada no lugar de uma expressão de um tipo $\sigma$ equivalente, e vice-versa. A relação de subtipagem retira o "vice-versa", tornando a checagem de tipos baseada em uma regra de substituição de expressões unidirecional.

Em termos de sistemas de tipos, uma relação de subtipagem caracteriza-se como sendo qualquer relação de preordem entre tipos, tal que, para quaisquer tipos $\sigma' <: \sigma$, e qualquer expressão $e$ de tipo $\sigma'$, pode-se deduzir que $e$ tem tipo $\sigma$. Isso é expresso pela seguinte regra de derivação:

$$\frac{\sigma' <: \sigma \qquad e : \sigma'}{e : \sigma} \tag{SUB}$$

Esta regra estabelece, de fato, uma relação (unidirecional) de substituição entre expressões, que caracteriza toda relação de subtipagem: note que, pela regra acima, se $\sigma'$ é um subtipo de $\sigma$, em todo lugar em que uma expressão de tipo $\sigma$ ocorre podemos usar uma expressão de tipo $\sigma'$, uma vez que essa expressão de tipo $\sigma'$ pode ser transformada em uma expressão de tipo $\sigma$.

Em particular, um tipo polimórfico quantificado (seção 4) pode ser visto como um (caso especial de) subtipo de qualquer instância desse tipo. O uso e a interrelação entre construções de linguagens de programação associadas com subtipagem, como formas de polimorfismo, classes, objetos e registros, são tópicos atuais de pesquisa (veja, por exemplo, [GM94, AC96, Mit96]).

Relações de subtipagem existem na grande maioria das linguagens de programação, mas não de forma generalizada. Por exemplo, o tratamento de expressões aritméticas em Fortran, cuja idéia básica foi adotada em muitas linguagens definidas posteriormente, é caracterizado por permitir combinações de expressões inteiras com reais (ponto flutuante). Nessas expressões, valores inteiros são convertidos, quando necessário, para valores reais correspondentes. Um outro exemplo de

subtipagem ocorre no caso de intervalos de tipos discretos, como em Pascal e Ada (veja, por exemplo, [JW74] e [Ich79]). O intervalo de um tipo discreto $\sigma$ é um subtipo de $\sigma$.

O interesse em linguagens orientadas por objetos (LOOs) incentivou ainda mais o estudo de aspectos relativos a subtipagem. Em LOOs tipadas (assim como em linguagens tipadas que suportam alguma forma de polimorfismo), a checagem de tipos baseia-se fundamentalmente em uma relação de subtipagem, e não de equivalência entre tipos.

A relação de subtipagem não é, necessariamante, uma relação anti-simétrica, uma vez que dois tipos distintos podem ser subtipos um do outro. Por exemplo, na linguagem Modula-3 [CDJ+89, Nel91], um tipo "empacotado" ("*packed*") e outro "não-empacotado" ("*unpacked*") podem ser subtipos um do outro. Dados empacotados e não-empacotados têm representações diferentes, e podem ser convertidos de um para o outro.

A relação de subtipagem entre expressões de tipos depende, em geral, de uma *relação de conversão entre o conjunto de valores* desses tipos. Valores de um subtipo de $\sigma$ devem poder ser transformados em valores de tipo $\sigma$. Teoricamente, domínios semânticos podem ser definidos, por exemplo, de forma a considerar o conjunto de valores de um subtipo $\sigma'$, de um dado tipo $\sigma$, como um subconjunto do conjunto de valores de $\sigma$. Nesse caso, a relação de conversão é uma *relação de inclusão*.

Na prática, a conversão de valores de um subtipo de $\sigma$ para valores de $\sigma$ pode constituir-se em uma mudança na representação desses valores. Por exemplo, objetos (em LOOs) e registros são em geral implementados de forma a não necessitar de uma mudança de representação (na transformação de valores do subtipo para o supertipo), ao passo que inteiros são convertidos para reais (números de ponto flutuante) mediante uma mudança de representação. No caso de intervalos de tipos discretos, a mudança ou não na representação é mais dependente da implementação. Em geral, no caso em que o custo requerido para conversão é grande e a freqüência de conversões é grande, o uso de uma relação de inclusão é mais adequado. Por outro lado, se a freqüência de operações (sobre valores de um tipo ou de seu subtipo) é grande e há uma grande diferença entre a eficiência dessas operações (com relação a operações realizadas sobre sua contraparte) o uso de representações diferentes é mais adequado.

Uma interpretação semântica de subtipagem pode ser feita tanto em termos de uma relação de inclusão, estabelecendo que o conjunto de valores do subtipo é um subconjunto do valores do supertipo, quanto em termos de uma relação de conversão, que transforma valores de um subtipo em valores do supertipo. Em geral, ambas são úteis, no sentido de prover suporte a uma implementação da linguagem e de desenvolver um melhor entendimento sobre uma linguagem. [16]

Um sistema de tipos com suporte a subtipagem contém, tipicamente, um axioma ou uma regra de inferência para cada expressão-de-tipo, que determina as propriedades da relação de subtipagem para esse tipo. O princípio básico para se definir que um tipo $\sigma'$ é subtipo de um outro tipo $\sigma$ é o de que todas as operações definidas sobre expressões de tipo $\sigma$ devem também ser bem definidas para expressões de tipo $\sigma'$. Nas seções 5.1 e 5.2 veremos alguns exemplos, para tipos específicos.

## Tipagem mínima e tipo mínimo

Um sistema de tipos com subtipagem, que inclui a regra (SUB), não possui a propriedade de unicidade de tipos, segundo a qual, em um dado contexto, existe um único tipo para cada expressão. Usualmente, verificam-se as propriedades de *tipo mínimo* e *tipagem mínima*, análogas às propriedades de tipo principal e tipagem principal, descritas na seção 4.1.

Em um dado contexto, para toda expressão $e$ existe um *tipo mínimo* $\sigma$, tal que $\Gamma \vdash e : \sigma$ é provável e $\Gamma \vdash e : \sigma'$ é provável, se, e somente se, $\sigma <: \sigma'$.

Uma tipagem mínima $(\Gamma, \sigma)$, para uma expressão $e$, é tal que $\Gamma$ *requer menos* das variáveis $x \in dom(\Gamma)$ e o tipo $\sigma$ *provê mais* do que qualquer outro possível tipo para $e$, onde as noções de requer menos e provê mais são definidas com base na relação de subtipagem $<:$.

## 5.1 Funções

Para tipos funcionais, tem-se a seguinte regra de subtipagem:

$$\frac{\sigma_a <: \sigma'_a \qquad \sigma'_r <: \sigma_r}{\sigma'_a \to \sigma'_r <: \sigma_a \to \sigma_r}$$

[16] Para tratamentos sobre a interpretação semântica de subtipagem veja, por exemplo, [Mit96].

Se a relação de subtipagem ($<:$) é vista como uma relação de ordem, a regra acima indica que o construtor de tipo $\rightarrow$ é *monotônico* (também chamado de *covariante*) em seu segundo argumento, e *antimonotônico* (também chamado de *contravariante*) no primeiro argumento. Essa regra reflete o fato de que, se $f$ tem tipo $\sigma'_a \rightarrow \sigma'_r$, então, pela regra (SUBS):

- $f$ aceita argumentos de tipo $\sigma_a$, para qualquer $\sigma_a <: \sigma'_a$ (uma vez que esses argumentos podem ser transformados em argumentos de tipo $\sigma'_a$);
- $f$ fornece resultados de tipo $\sigma_r$, para qualquer $\sigma_r$ tal que $\sigma'_r <: \sigma_r$ (analogamente).

Portanto, toda função $\sigma'_a \rightarrow \sigma'_r$ também tem tipo $\sigma_a \rightarrow \sigma_r$, ou, em outras palavras, $\sigma'_a \rightarrow \sigma'_r$ é um subtipo de $\sigma_a \rightarrow \sigma_r$, se $\sigma_a <: \sigma'_a$ e $\sigma'_r <: \sigma_r$.

Como exemplo, considere `Int` $<:$ `Real`. Se $f$ é uma função de tipo `Real` $\rightarrow$ `Int`, então $f$ também tem tipo `Int` $\rightarrow$ `Int`, assim como `Int` $\rightarrow$ `Real` e `Real` $\rightarrow$ `Real`. A função $f$ pode ser aplicada a um inteiro ou real, fornecendo um resultado inteiro, o qual pode ser usado no lugar de valores reais. Note que, em contraposição, se $f$ tivesse tipo `Int` $\rightarrow$ `Real`, não poderia receber argumentos de tipo `Real`, e o seu resultado não poderia ser usado no lugar de valores inteiros.

A interpretação de subtipagem como uma relação de inclusão não parece a princípio natural, no caso de tipos funcionais: e.g. não parece a princípio natural considerar uma função $f$ de tipo `Real` $\rightarrow$ `Int` como um elemento do conjunto de funções de tipo `Int` $\rightarrow$ `Int`. O domínio de definição de $f$ (dos números reais) é (nessa interpretação) um superconjunto do domínio de definição das funções em `Int` $\rightarrow$ `Int`. Entretanto, a interpretação é válida, se considerarmos a interpretação abaixo, sutil, para tipos funcionais. Essa interpretação define, por exemplo, `Int` $\rightarrow$ `Int` como o conjunto das funções cujo domínio de definição *contém* o conjunto dos inteiros e, para qualquer argumento inteiro, fornece um inteiro como resultado:

$$[\![\sigma_a \rightarrow \sigma_r]\!]\rho = \{f \in V \mid \text{ se } x \in [\![\sigma_a]\!]\rho \text{ então } f(x) \in [\![\sigma_r]\!]\rho\}$$

A fim de garantir que "o conjunto de todas as funções que contenha um dado conjunto" seja, de fato, um conjunto bem definido (não possibilitando a introdução de paradoxos), é suficiente definir um "domínio universal" $V$, dos valores de todos os tipos, construído a partir de valores computáveis, definindo interpretações de tipos como subconjuntos desse domínio universal.

Como vimos anteriormente, uma função $f$ de tipo $\sigma_a \rightarrow \sigma_r$ também tem tipo $\sigma'_a \rightarrow \sigma_r$, se $\sigma'_a <: \sigma_a$. Esta função pode, portanto, quando considerada como um elemento do conjunto denotado por $\sigma'_a \rightarrow \sigma_r$, ser igual a uma outra função (digamos) $g$, de tipo $\sigma'_a \rightarrow \sigma_r$. No entanto, quando considerada como elemento do conjunto denotado por $\sigma_a \rightarrow \sigma_r$ (um subconjunto do conjunto denotado por $\sigma'_a \rightarrow \sigma_r$), $g$ e $f$ podem ser funções distintas. Isso não permite que se interprete a relação de subtipagem como uma relação de inclusão simples, no sentido de que dois elementos $a$ e $b$ de um conjunto $X$ são iguais se e somente se são iguais como elementos de qualquer subconjunto de $X$. Entretanto, existem interpretações que permitem considerar que a igualdade varie com o tipo em relações de inclusão (sem requerer uma função de conversão entre valores de um tipo para outro). Estas interpretações são baseadas em *relações parciais de equivalência*. O leitor interessado é estimulado a consultar, por exemplo, [Mit96].

## 5.2 Registros

A operação de seleção de um componente (ou campo) de um registro, de um determinado tipo $\sigma$, é a única operação a ser preservada para valores de subtipos de $\sigma$. Usando a notação $(r_1 : \sigma_1, \ldots, r_n : \sigma_n)$ para um tipo registro que tem $r_1, \ldots, r_n$ como nomes (ou rótulos) dos componentes do tipo registro, e $\sigma_1, \ldots, \sigma_n$ como tipos desses componentes, respectivamente, a relação de subtipagem é caracterizada pela seguinte regra:

$$\frac{\sigma'_1 <: \sigma_1 \ \ldots \ \sigma'_n <: \sigma_n}{(r_1 : \sigma'_1, \ldots, r_n : \sigma'_n, r_{n+1} : \sigma_{n+1}, \ldots, r_{n+m} : \sigma_{n+m}) \ <: \ (r_1 : \sigma_1, \ldots, r_n : \sigma_n)}$$

Segundo essa regra, subtipos de um tipo registro $\sigma$ são obtidos adicionando-se componentes ($r_{n+1}, \ldots, r_{n+m}$ na regra acima) ou substituindo o tipo $\sigma_i$ de um componente $r_i$ de $\sigma$ por um subtipo de $\sigma_i$. O primeiro exemplo seguinte ilustra o primeiro caso, e o segundo exemplo ilustra ambos os casos acima:

```
classe C
  método m (x:  A) : B ...

subclasse C′ de C
  método m (x:  A′) :  B′ ...
```

Figura 10: Redefinição de métodos

$$(\mathtt{a} : \mathtt{Int}, \mathtt{b} : \mathtt{Int}) <: (\mathtt{a} : \mathtt{Int})$$
$$(\mathtt{r} : (\mathtt{a} : \mathtt{Int}, \mathtt{b} : \mathtt{Int}), \mathtt{c} : \mathtt{Int}) <: (\mathtt{r} : (\mathtt{a} : \mathtt{Int}), \mathtt{c} : \mathtt{Int})$$

Para uma interpretação da subtipagem de registros como uma relação de inclusão, é natural interpretar um tipo registro $(r_i : \sigma_i)$, para $i = 1, \ldots, n$, como o conjunto de todos os valores (subconjuntos de um domínio universal) que contêm $n$ pares ordenados $(r_1, v_1), \ldots, (r_n, v_n)$, onde $v_i$ é um valor do conjunto denotado por $\sigma_i$, para $i = 1, \ldots, n$. Dessa forma, $(\mathtt{a} : \mathtt{Int}, \mathtt{b} : \mathtt{Int})$, por exemplo, representa um subconjunto do conjunto denotado por $(\mathtt{a} : \mathtt{Int})$.

### 5.2.1 Objetos

Para não nos estendermos muito em questões menos relevantes ao escopo desse artigo, essa seção pressupõe um conhecimento de conceitos básicos de orientação por objetos.

Denotemos por $t(\mathtt{C})$ o tipo de uma classe `C`. A relação de subtipagem tem, na maioria das linguagens orientadas por objetos, a seguinte propriedade.[17]

**Propriedade 6 (Subclasse $\Leftrightarrow$ Subtipo)** $t(\mathtt{C}') <: t(\mathtt{C})$ se e somente se $\mathtt{C}'$ é uma subclasse de `C`.

A fim de que erros de tipo possam ser detectados estaticamente, essa propriedade requer, em geral, que, em uma subclasse $\mathtt{C}'$ da classe `C` (Figura 10), tenhamos:

- os tipos dos parâmetros de métodos redefinidos em $\mathtt{C}'$ variem antimonotonicamente, e os tipos dos resultados desses métodos variem monotonicamente (cf. seção 5.1);
- objetos da classe $\mathtt{C}'$ devem ter pelo menos (possivelmente mais) os campos dos objetos da classe `C` (cf. seção 5.2).

## 5.3 Covariância versus Contravariância

As regras acima são conseqüência da observação de que, em uma invocação `o.m(a)`, de um método `m` de uma classe `C` que aceita parâmetros de um tipo `A` e fornece um resultado de um tipo `B`, devemos ter:

- uma redefinição de `m`, com parâmetro de tipo $\mathtt{A}'$, deve ser tal que $\mathtt{A} <: \mathtt{A}'$ (de modo a que a redefinição possa aceitar um argumento de tipo `A`, o qual pode ser convertido para o tipo $\mathtt{A}'$);
- a redefinição de `m` com resultado de tipo $\mathtt{B}'$ deve ser tal que $\mathtt{B}' <: \mathtt{B}$ (de modo a que o resultado fornecido possa ser convertido para o tipo `B`).

Supondo covariância de tipos de parâmetros na redefinição de métodos em subclasses, após a invocação $\mathtt{o}'.\mathtt{m}(\mathtt{a})$, quando $\mathtt{o}'$ é um objeto da classe $\mathtt{C}'$, e `a` tem tipo `A`, a execução do método `m` da classe $\mathtt{C}'$ poderia fazer acesso a um campo de $\mathtt{A}'$ que não existe em `A`.

As linguagens Eiffel [Mey92] e Beta [MMPN93] de fato adotam a covariância de tipos de parâmetros de métodos (veja discussões sobre o assunto em [Mey92, Coo89, CM97]). Em Beta,

[17] Essa propriedade é comumente chamada de *herança-é-subtipagem*. Seguindo Abadi e Cardelli [AC96], diferenciamos as relações de herança e subclasse. Herança é uma relação de compartilhamento de atributos entre uma classe e uma sua subclasse. Uma subclasse pode, por exemplo, redefinir todos os métodos da superclasse, e portanto não herdar nada.

erros são detectados em tempo de execução. Eiffel adiciona restrições, de modo a possibilitar a realização de testes em tempo de ligação. Existe muita controvérsia na comunidade de orientação por objetos a respeito do assunto (veja também [Cas95]).

Métodos com argumentos que têm o tipo (`Self`) do objeto corrente (denotados por `self`, em C++ [Str91], `this`, em Java [AG96] e `Current`, em Eiffel [Mey92]), aparecem naturalmente em exemplos, de forma covariante (cf. construção `like Current` em Eiffel).

#### Parametrização por Tipos com Restrições

Uma solução para o problema da covariância de tipos de argumentos de métodos pode ser obtida através do uso de *restrições de subtipagem* (veja seção 6), um recurso disponível, por exemplo, em Eiffel. Considere as classes definidas a seguir, parametrizadas por tipos com restrições de subtipagem:

classe $\mathtt{C_1}$ (X <: A)
  método m (x: X) : B ...

subclasse $\mathtt{C'_1}$ (X <: A') de $\mathtt{C_1}$[X]
  método m (x: X) : $\mathtt{B'}$ ...

Figura 11: Redefinição de métodos

A restrição de subtipagem `X <: A` indica que, para qualquer objeto da classe $\mathtt{C_1}$, o tipo `X` tem que ser um subtipo de `A`. `C`[`X`] representa a classe obtida pela instanciação de `C` com o tipo dado por `X`.

Dado $\sigma <: \mathtt{A'}$, seja $\sigma_1 = t(\mathtt{C_1}[\sigma])$ o tipo de objetos criados através da instanciação do parâmetro `X` de $\mathtt{C_1}$ por $\sigma$ e, similarmente, seja $\sigma'_1 = t(\mathtt{C'_1}[\sigma])$. Temos que $\sigma'_1 <: \sigma_1$. Em particular, $t(\mathtt{C'_1}[\mathtt{A'}]) <: t(\mathtt{C_1}[\mathtt{A'}])$.

### Subclasse $\not\Rightarrow$ Subtipo

Algumas linguagens de programação orientadas por objetos mais recentes desvinculam herança de subtipagem, com base na visão de que a separação entre interfaces e implementações facilita a estruturação de programas. A implicação bidirecional da propriedade 6 é transformada em uma implicação unidirecional:

**Propriedade 7 (Subclasse $\Rightarrow$ Subtipo)** Se $\mathtt{C'}$ é uma subclasse de `C` então $t(\mathtt{C'}) <: t(\mathtt{C})$.

Com a propriedade 7, pode-se ter subtipos não provenientes de subclasses. A relação de subtipagem requer uma definição independente, baseada tipicamente na estrutura dos tipos (cf. exemplos nas seções 5.1 e 5.2).

Convém ainda notar que uma separação entre interfaces e implementações torna o mecanismo de parametrização por subtipos mais elegante.

Com o intuito de permitir a herança de métodos com parâmetros que tenham tipo `Self`, uma outra proposta abandona também a propriedade (7) (veja [CHC90]). O preço pago pela flexibilidade adicional proporcionada ao mecanismo de herança é exatamente a perda de flexibilidade na relação de subtipagem, uma vez que subclasses podem não dar origem a subtipos, quando se usa o tipo `Self` como tipo de parâmetros redefinidos em subclasses.

## 6 Polimorfismo Restrito

Tipos polimórficos $\forall\alpha.\sigma$ representam a coleção de todos os tipos $\sigma[\tau/\alpha]$ (onde $\tau$ and $\sigma$ são tomados de universos especificados) obtidos pela substituição de $\alpha$ por $\tau$ em $\sigma$ (modulo renomeação de variáveis de tipo quantificadas).

É frequentemente útil (ou requerido) que se tenha um controle mais fino sobre a formação de tipos polimórficos. Precisamos de uma forma de restringir o conjunto de tipos que podem ser usados para formar um tipo polimórfico.

Seja $\kappa$ um conjunto de pares $v : \tau$ (onde $v$ é um nome e $\tau$ é um tipo). O *tipo polimórfico restrito* $\forall\alpha.\,\kappa.\,\tau$ representa a coleção de todos os tipos $\tau[\tau'/\alpha]$ (onde $\tau$ and $\tau'$ são tomados de universos especificados) obtidos pela substituição de $\alpha$ por $\tau'$ em $\tau$ (modulo renomeação de variáveis de tipo quantificadas) tais que valores em $\kappa[\tau'/\alpha]$ são definidos. Por exemplo:

$$\forall\alpha.\,\{(<) : \alpha \to \alpha \to \texttt{Bool}\}.\,[\alpha] \to [\alpha]$$

representa a coleção de todos os tipos $[\tau] \to [\tau]$ tais que existe uma definição para uma função (<), de tipo $\tau \to \tau \to \texttt{Bool}$.

Devido a restrições de espaço, não apresentaremos aqui regras de sistemas de tipos com tipos polimórficos restritos. O leitor é encorajado a consultar [CF99a, FM96, Car96, Pie97].

Tipos polimórficos restritos são adequados para expressar tipos de símbolos sobrecarregados [CF99a], como ilustrado pelos seguintes exemplos:

$$\begin{aligned}
(<) &: \{(<) : \alpha \to \alpha \to \texttt{Bool}\}.\,\alpha \to \alpha \to \texttt{Bool} \\
(+) &: \{(+) : \alpha \to \alpha \to \alpha\}.\,\alpha \to \alpha \to \alpha \\
\texttt{sort} &: \{(<) : \alpha \to \alpha \to \texttt{Bool}\}.\,[\alpha] \to [\alpha]
\end{aligned}$$

Uma outra forma de polimorfismo restrito, denominada *polimorfismo limitado-por-subtipagem* [CW85, Mit88, CG92, FM96, FM98], usa restrições de subtipagem.[18] Suponha, por exemplo, um tipo registro (definido recursivamente) Ord:

$$\texttt{Ord} = ((<) : \texttt{Ord} \to \texttt{Ord} \to \texttt{Bool})$$

Uma função de ordenação pode ser definida e, em um sistema de tipos com polimorfismo limitado-por-subtipagem, ter o tipo:

$$\forall\alpha <: \texttt{Ord}.\,[\alpha] \to [\alpha]$$

Entretanto, esta função não seria nada útil, uma vez que o construtor de tipos functionais ($\to$) é contravariante no seu primeiro argumento (veja seção 5.1). Portanto, esta função (polimórfica restrita) não poderia ser aplicada a listas de elementos cujos tipos são subtipos de Ord. Ela não poderia ser aplicada a inteiros, strings, ou qualquer registro R, com um campo (<) de tipo $\texttt{R} \to \texttt{R} \to \texttt{Bool}$ (de fato, o único elemento desse tipo é a função identidade).

Uma extensão do polimorfismo limitado-por-subtipagem, chamada de *polimorfismo $F$-limitado* [HP95, Mit96], resolve o problema acima utilizando funções de tipos em tipos. No exemplo acima, o tipo da função de ordenação seria dado, em um sistema de tipos com polimorfismo $F$-limitado, por:

$$\forall\alpha <: F(\alpha).\,[\alpha] \to [\alpha]$$

onde $F(\alpha) = ((<) : \alpha \to \alpha \to \texttt{Bool})$. Note a similaridade desse tipo com aquele dado pela nossa primeira forma de polimorfismo restrito.

Uma alternativa para polimorfismo $F$-bounded, chamda de *polimorfismo com subtipagem de ordem superior*, usa uma ordenação (<:) sobre construtores de tipos. Com essa abordagem, o tipo de nossa função de ordenação seria reescrito como a seguir:

$$\forall F <: \texttt{Ord}.\,[\mu\alpha.\,F(\alpha)] \to [\mu\alpha.\,F(\alpha)]$$

onde $\mu\alpha.\,F(\alpha)$ representa um tipo definido recursivamente (veja seção 3.1).

Um estudo comparativo entre essas abordagens, assim como as conexões com outras áreas relacionadas, tais como sistemas de módulos [HL94, Mac85, MTH90, Mac86, MMM91], será abordado em trabalhos futuros.

[18] Existe um número grande de publicações na literatura recente sobre o assunto. Linguagens que incluem parâmetros de tipo limitado-por-subtipagem incluem Eiffel, Trellis/Owl, Sather, PolyTOIL, Rapide, Fun, Quest e Abel. Veja referências em e.g. [Pie97, Car96].

# 7 Tipos abstratos

Tipos *concretos* são definidos por meio de construtores de valores que denotam os *elementos* do tipo. Tipos *abstratos*, por outro lado, são definidos através de uma descrição do conjunto de *operações* definidas sobre os elementos do tipo. A representação dos elementos de um tipo abstrato, assim como a implementação das operações sobre esse tipo, são abstraídas: tanto a representação dos elementos quanto a implementação das operações pode ser alterada, sem que isso altere o significado de programas que envolvam valores do tipo abstrato.

A maioria das linguagens de programação provê algum mecanismo para suporte à implementação de tipos abstratos. Em operações definidas sobre um tipo abstrato, o tipo representação (i. e. o tipo da representação dos valores do tipo abstrato) é, em geral, um sinônimo desse tipo abstrato (nesse caso, o tipo abstrato é dito *transparente*). Em outras partes do programa, que usam o tipo abstrato definido, o tipo abstrato é distinto de quaisquer outros tipos (nesse caso, o tipo abstrato é *opaco*). Em algumas linguagens, o tipo abstrato é distinguido de outros tipos pelo uso de uma construção especial de definição de tipos, que determina o uso de equivalência *por nome* para o teste de equivalência com tipos abstratos.

Em linguagens funcionais modernas, tais como Haskell [Tho96, J+99], Miranda [Tur85] e SML [Mit90, Pau96], utiliza-se um mecanismo diferente para distinguir o tipo abstrato de outros tipos. O tipo representação é usualmente definido como um *datatype* (veja seção 3.1). Os construtores do tipo da representação são visíveis apenas nas definições das operações sobre o tipo abstrato, não podendo ser usados nas demais partes do programa, que usam o tipo abstrato.

Uma visão geral das abordagens propostas é apresentada a seguir.

A construção *abstype* de SML e Miranda utiliza um *datatype* como tipo da representação. Por exemplo, a definição de um tipo abstrato `set` é esquematizada a seguir:

```
abstype ''a set = set of ''a list with
   ...
   fun union (set [ ]) s = s
     | union (set (a::x)) (s@set y) =
       if a mem y then union (set x) s
       else let val set x'= union (set x) s
            in set (a::x')
```

Uma desvantagem do uso de um datatype como o tipo da representação é o incômodo "empacotamento" e "desempacotameto" de construtores, necessário na definição de funções sobre o tipo abstrato, como exemplificado pela definição da função `union` acima.

Uma construção semelhante, adotada em Hugs [JP97], evita esse problema. O exemplo acima pode ser reescrito em Hugs da seguinte forma:

```
type Set = [a] in ..., union

union [ ] s = s
union (a:x) s
   | a `elem` s = union x s
   | otherwise  = a:  (union x s)
```

Em Hugs, o fato de o tipo abstrato e o tipo da sua representação serem sinônimos, na definição de funções do tipo abstrato, tem conseqüências indesejáveis quando se considera a definição de uma operação sobrecarregada tanto sobre elementos do tipo abstrato quanto sobre elementos do tipo da sua representação: na aplicação dessa operação a argumentos do tipo abstrato, nunca é usada a instância dessa operação definida sobre o tipo abstrato (nem mesmo se indicado explicitamente por uma anotação de tipo), ao contrário, é aplicada a instância dessa operação definida sobre o tipo da sua representação.

Em Haskell e SML, tipos abstratos podem ser definidos através do sistema de módulos. Em Haskell, um módulo consiste de uma interface, que especifica uma lista de nomes exportados e importados, e de uma implementação. A definição de um tipo abstrato requer o uso de um *datatype* como o tipo da sua representação e requer a não exportação dos construtores desse tipo,

como forma de ocultar a representação. Assim, apenas as funções exportadas pelo módulo podem ser usadas para operar sobre valores do tipo abstrato. Por exemplo, o tipo abstrato `Queue a`, de pilhas de elementos de um tipo genérico `a`, pode ser definido em Haskell do seguinte modo:

```
module Queue (Queue a, emptyQueue, ins, rem)
    data Queue a = MakeQueue [a]
    emptyQueue = MakeQueue [ ]
    ins a (MakeQueue q) = MakeQueue (q ++ [a])
    rem (MaqueQueue [ ]) = error ...
    rem (MakeQueue (a:q)) = (a, MakeQueue q)
```

O exemplo acima ilustra, mais uma vez, o inconveniente "empacotamento" de "desempacotamento" de construtores decorrente do uso de um *datatype* como tipo da representação.

As abordagens anteriormente apresentadas admitem a definição de uma única implementação para cada tipo abstrato. SML possui um sistema de módulos mais flexível, que possibilita a definição de *assinaturas* e *estruturas* como entidades independentes. Uma estrutura é uma coleção de declarações de tipos, valores e estruturas. Uma assinatura constitui o "tipo" ou interface de uma estrutura.

No sistema de módulos de SML, várias estruturas podem compartilhar uma mesma assinatura e uma única estrutura pode satisfazer a diversas assinaturas. Além disso, SML provê estruturas parametrizadas denominadas funtores.[19]

As primeiras propostas de sistemas de tipos para tipos abstratos consideram tipos abstratos como valores de tipos existenciais [CW85, MP88]. Um valor de tipo existencial $\exists\alpha.\sigma$ representa um valor de tipo $\sigma[\tau/\alpha]$, para *algum* $\tau$ (visto como o tipo representação do tipo $\sigma[\tau/\alpha]$).[20]

Tipos abstratos, assim como objetos em LOOs, são também modelados por tipos existenciais com restrições, análogas às restrições sobre tipos polimórficos descritas na seção 6 (veja, por exemplo, [FM96, FM98]).

Tipos abstratos modelados como valores de tipos existenciais são objetos de primeira classe, no sentido de que valores desses tipos podem ser associados a variáveis, assim como podem ser passados como argumentos ou retornados como resultado de funções. Estruturas, ao contrário, não são tratadas como valores de primeira classe (veja discussões de problemas, por exemplo, em [MR86, MH88]). Essa distinção tem levado a discussões quanto a adequabilidade de se definir tipos abstratos por meio de um sistema de módulos [Mac85, MTH90], originando também propostas para o tratamento de estruturas como valores de primeira classe (veja, por exemplo, [MMM91, HL94, Ler94]).

Outra questão importante é o conflito existente entre tipos abstratos e a definição de funções por casamento de padrão, essencialmente baseada na representação de dados. Nenhuma das abordagens anteriormente discutidas possibilita o uso de valores de tipos abstratos na definição de funções por casamento de padrão, o que limita fortemente o uso de tipos abstratos em programas. Esse problema é discutido nos trabalhos apresentados em [Wad87, BC93, CF99b], que utilizam mecanismos baseados no conceito de *visões*, para possibilitar a definição de funções por casamento de padrão sobre valores de tipos abstratos.

# Referências


[AC96] M. Abadi and L. Cardelli. *A theory of objects*. Springer, 1996.

[AG96] K. Arnold and J. Gosling. *The Java Programming Language*. Addison-Wesley, 1996.

[Ast91] E. Astesiano. Inductive and Operational Semantics. In E.J. Neuhold and M. Paul, editors, *Formal Description of Programming Concepts*, pages 51–136. Springer-Verlag, 1991.


[19] Uma descrição mais detalhada do sistema de módulos de SML pode ser obtida, por exemplo, em [Mac85, MTH90, MT91].

[20] Tipos existenciais foram anteriormente tratados no sistema de $\lambda$-calculus polimórfico impredicativo, desenvolvido, independentemente, por Girard [Gir71, Gir89] e Reynolds [Rey74, Rey83], mas não associados, no trabalho de Girard, ao conceito de tipos abstratos. Uma descrição mais didática da conexão entre tipos existenciais e tipos abstratos pode ser encontrada em [Mit96].

[Aug99] L. Augustsson. Cayenne – a language with dependent types. *ACM SIGPLAN Notices*, 34(1):239–250, January 1999.

[Bar84] H.P. Barendregt. *The Lambda Calculus: its Syntax and Semantics*. North Holland, 1984.

[BC93] F. W. Burton and R. D. Cameron. Pattern matching with abstract types. *Journal of Functional Programming*, 1993.

[C+86] R. L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice Hall, 1986.

[Car96] L. Cardelli. Type Systems. *Draft: DEC/SRC*, pages 1–42, Mar 1996.

[Cas95] G. Castagna. Covariance and contravariance: Conflict without a cause. *ACM Transactions on Programming Languages and Systems*, 17(3):431–447, 1995.

[CDJ+89] L. Cardelli, J. Donahue, M. Jordan, B. Kalsow, and G. Nelson. The Modula-3 type system. In *Proceedings of the 16th ACM Symp. Principles of Programming Languages*, pages 202–121, 1989.

[CF99a] C. Camarão and L. Figueiredo. Type inference for overloading: without restrictions, declarations or annotations. Submmited to The International Conf. on Principles and Practice of Declarative Programming (PPDP'99), Paris, France, September 29-October 1st 1999.

[CF99b] C. Camarão and L. Figueiredo. A type with a view. In *Anais do III Simpósio Brasileiro de Linguagens de Programação (SBLP'99)*, 1999.

[CG92] P.-L. Curien and G. Ghelli. Coherence of subsumption: Minimum typing and type-checking in $F_{\leq}$. *Mathematical Structures in Computer Science*, 2:55–91, 1992. Also in Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design* (MIT Press, 1994).

[CH88] T. Coquant and G. Huet. The calculus of constructions. *Information and Computation*, 73(213):95–120, 1988.

[CHC90] W. Cook, W. Hill, and P. Canning. Inheritance is not Subtyping. In *Proc. of the 17th Annual ACM Symp. on Principles of Programming languages (POPL'90)*, page 125, 1990.

[Chu32] A. Church. A set of postulates for the foundation of logic. *Ann. of Math.*, 33:346–366, 1932.

[Chu36] A. Church. Un unsolvable problem of elementary number theory. *Amer. J. Math.*, 58:345–363, 1936.

[Chu40] A. Church. A formulation of the simple theory of types. *J. Symbolic Logic*, 5:56–58, 1940.

[Chu41] A. Church. *The Calculi of Lambda Conversrion*. Princeton Univ. Press, 1941.

[CL91] L. Cardelli and G. Longo. A semantics basis for Quest. *Journal of Functional Programming*, 1(4):417–458, 1991.

[CM97] Y. Crespo and M. K. Mora. More about system-level validity in Eiffel. In *Anais do II Simpósio Brasileiro de Linguagens de Programação (SBLP'97)*, pages 299–314, 1997.

[Coo89] W. R. Cook. A proposal for making Eiffel type-safe . In *Proceedings of the 3rd European Conference on Object-Oriented Programming*, pages 57–70, 1989.

[CW85] L. Cardelli and P. Wegner. On understanding types, data abstraction and polymorphism. *Computing Surveys*, 17(4), December 1985.

[DB96] D. Duggan and F. Bent. Explaining type inference. *Science of Computer Programming*, 27:37–83, 1996.

[FLO83] S. Fortune, D. Leivant, and M. O'Donnel. The expressiveness of simple and second-order type sructures. *Journal of the ACM*, 30(1):151–185, 1983.

[FM96] K. Fischer and J. Mitchell. The Development of Type Systems for Object-Oriented Languages. *Theory and Practice of Object Systems*, 1(3):189–200, 1996.

[FM98] K. Fischer and J. Mitchell. On the relationship between classes, objects and data abstraction. *Theory and Practice of Object Systems*, 4(1):3–32, 1998.

[Gal86] J. H. Gallier. *Logic for Computer Science*. Harper & Row, 1986.

[Gir71] J.-Y. Girard. Une extension de l'interpretation de Gödel à l'analyse, et son application à l'elimination des coupures dans l'analyse et la théorie des types. In *Proc. of the 2nd Scandinavian logic symposium*, pages 63–92. North-Holland, Amsterdam, 1971.

[Gir89] J. Girard. *Proofs and Types*. Cambridge University Press, 1989.

[GM94] C. A. Gunter and J. C. Mitchell, editors. *Theoretical Aspects of Object Oriented Programming Languages: Types, Semantics, and Language Design*. MIT Press Foundations of Computing Series, 1994.

[Gor79] M. J. C. Gordon. *The denotational description of programming languages*. Springer-Verlag, 1979.

[Har93] R. Harper. Introduction to Standard ML. Technical report, School of Computer Science, Carnegie Mellon University, 1993.

[Hat82] W. S. Hatcher. *The Logical Foundations of Mathematics*. Pergamon, 1982.

[HL94] R. Harper and M. Lillibridge. A type-theoretic approach to higher-order modules with sharing. In ACM, editor, *Proc. of 21st Annual ACM SIGACT-SIGPLAN Symp. on Principles of Programming Languages*, pages 123–137, New York, NY, USA, 1994. ACM Press.

[HP95] M. Hofmann and B. Pierce. Positive subtyping. In *Proc. of 22nd Annual ACM Symp. on Principles of Programming Languages*, pages 186–197. ACM, January 1995.

[Ich79] J. Ichbiah. Rationale for the design of the Ada programming language. *ACM SigPlan Notices*, 14(6B), 1979.

[J+99] S. Peyton Jones et al. *The Haskell 98 Report*. http://haskell.systemsz.cs.yale.edu/onlinereport/, 1999.

[Jim96] T. Jim. What are principal typings and what are they good for? *ACM*, 1996.

[JP97] M. P. Jones and J. C. Peterson. *The Hugs User Manual*. http://www.haskell/systemsz.cs.yale.edu/hugs/docs, 1997.

[JW74] K. Jensen and N. Wirth. *PASCAL User Manual and Report*. Springer-Verlag, 1974. Second edition.

[Kle36] S.C. Kleene. Lambda definability and recursiveness. *Duke Math. J.*, 2:340–353, 1936.

[Ler94] X. Leroy. Manifest types, modules and separate compilation. In *Proc. 21th ACM Symp. on Principles of Programming Languages*, pages 109–123, January 1994.

[Mac85] D. MacQueen. Modules for Standard ML. *Polymorphism*, 2(2):35, 1985.

[Mac86] D. MacQueen. Using Dependent Types to Express Modular Structure. In *Proc. of the 13th ACM Symp. on Principles of Programming Languages*, pages 277–286, 1986.

[Mey92] B. Meyer. *Eiffel: the Language*. Prentice Hall, 1992.

[MH88] J.C. Mitchell and R. Harper. The essence of ML. *Proc. 5th ACM Symp. on Principles of Programming Languages*, pages 28–46, 1988.

[MH93] J. Mitchell and R. Harper. On the Type Structure of Standard ML. *ACM Transactions on Programming Languages and Systems*, 15(2):211–252, Apr 1993.

[MHT87] J. Mitchell, R. Harper, and M. Tofte. A type Discipline for Program Modules. *Lecture Notes in Computer Science*, 250, 1987.

[Mil84] R. Milner. The Standard ML Core Language. Technical Report CSR-168-84, Computer Science Department, Edinburgh University, 1984.

[Mit88] J.C. Mitchell. Polymorphic type inference and containment. *Information and Computation*, 76(2/3):211–249, 1988.

[Mit90] J.C. Mitchell. *Handbook of Theoretical Computer Science, Vol. B*, chapter Type Systems for Programming Languages, pages 365–458. MIT Press, 1990.

[Mit96] J.C. Mitchell. *Foundations for programming languages*. MIT Press, 1996.

[ML78] Per Martin-Löf. Constructive Mathematics and Computer Programming. In *Proc. of the 6th International Congress for Logic, Methodology, and Phylosophy of Science*, pages 153–175. North-Holland, Amsterdam, 1978.

[ML84] Per Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, Naples, 1984.

[MMM91] J. Mitchel, S. Meldal, and N. Madhav. An extension of standard ML modules with subtyping and inheritance. Technical Report CSL-TR-91-472, Computer Systems Laboratory, Departments of Electrical Engineering and Computer Science Standford University, May 1991.

[MMPN93] O. Madsen, B. Möler-Pedersen, and K. Nygaard. *Object-oriented programming in the Beta programming language*. Addison-Wesley, 1993.

[Mos90] P. D. Mosses. Denotational semantics. In J. van Leewen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, chapter 11, pages 577–631. The MIT Press, New York, N.Y., 1990.

[MP88] J. Mitchell and G. Plotkin. Abstract types have existential types. *ACM Trans. on Programming Languages and Systems*, 10(3):470–502, 1988.

[MR86] A. Meyer and M. Reinhold. Type is not a type. In *Proc. 13th ACM Symp. on Principles of Programming Languages*, January 1986.

[MT91] R. Milner and M. Tofte. *Commentary on Standard ML*. MIT Press, 1991.

[MTH90] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, 1990.

[Nel91] G. Nelson. *Systems Programming with Modula-3*. Prentice Hall, 1991.

[Pau96] L.C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 1996. 2nd edition.

[Pie97] B. C. Pierce. Bounded quantification with bottom. Technical Report 492, Computer Science Department, Indiana University, 1997.

[Plo81] G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University, Department of Computer Science, 1981.

[Pra65] D. Prawitz. *Natural Deduction*. AlmQvist & Wiksell, 1965.

[Qui69] W. V. O. Quine. *Set Theory and its Logic*. Harvard University Press, 1969.

[Rey74] J. C. Reynolds. Towards a theory of type structure. In B. Robinet, editor, *Colloque sur la programmation*, pages 408–425. Lecture Notes in Computer Science, vol.19, Springer-Verlag, 1974.

[Rey83] J. C. Reynolds. Types, abstraction and parametric polymorphism. In R. E. A. Mason, editor, *Information Processing 83*, pages 513–523, Amsterdam, 1983. Elsevier Science Publishers B. V. (North-Holland).

[RRW91] G. M. Reed, A. W. Roscoe, and R. F. Wachter, editors. *Topology and Category Theory in Computer Science*, chapter 14. Oxford University Press, 1991.

[Sco72] D. S. Scott. Lattice Theory, Data types and Semantics. *NYU Symposium on Formal Semantics*, pages 64–106, 1972.

[Sco76] D. S. Scott. Data types as lattices. *SIAM J. Computing*, 5(3):522–587, 1976.

[Sto77] J. E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* MIT Press Series in Computer Science. MIT Press, Cambridge, Mass., 1977.

[Sto88] A. Stoughton. *Fully Abstract Models of Programming Languages.* John Wiley and Sons, 1988.

[Str91] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley, 2nd edition, 1991.

[Ten81] R.D. Tennent. *Principles of Programming Languages.* Prentice-Hall, 1981.

[Tho91] S. Thompson. *Type Theory and Functional Programming.* Addison-Wesley, 1991.

[Tho96] S. Thompson. *Haskell: The Craft of Functional Programming.* Addison-Wesley, 1996.

[Tur37] A. M. Turing. Computability and lambda definability. *J. Symbolic Logic*, 2:153–163, 1937.

[Tur85] D. Turner. A non-strict functional language with polymorphic types. In *Proceedings of the 2nd International Conference on Functional Programming and Computer Architecture.* IEEE Computer Society Press, 1985.

[Wad87] P. Wadler. Views: a way for pattern matching to cohabit with data abstraction. *POPL'87*, 14:307–313, 1987.

[Wat91] D. A. Watt. *Programming language syntax and semantics.* Prentice Hall, 1991.

[WSH77] J. Welsh, W. Sneeringer, and C.A.R. Hoare. Ambiguities and insecurities in pascal. *Software-Practice and Experience*, 7(6):685–696, 1977.