Síntese
On nderstanding types, data abstraction, and polymorphism

Iyan Lucas Duarte Marques[1], **Samir do Amorim Cambraia**[1], **Guilherme Cosso**[1]

[1]Instituto de Ciências Exatas e Informática — Pontifícia Universidade Católica Minas Gerais (PUC-MG)

1. From Typed to Untyped Universes

1.1. Organizing Untyped Universes

"Por que os tipos são necessários para as LPs", para responder a essa pergunta, examinamos como os tipos surgem em vários domínios da ciência da computação e da matemática. Para isso, vamos repartir esse universo em 4 partes.

- Strings de bits na memória do computador:
- Expressões-S em LISP puro:
- Expressões λ em calculo λ:
- Conjunto da teoria de conjuntos:

1.1.1. *Bit strings*

O mais concreto dos tipos. Considerado "não tipado", porém representado como cadeias de bits: caracteres, números, ponteiros, dados estruturados, programas. O pedaço de memória bruta, geralmente não temos como dizer o que está sendo representado este significado é dado por um interpretador externo.

1.1.2. Puro LISP

As expressões S de LISP formam outro universo não tipado, que geralmente é construído em cima do universo de *string* de bits anterior. Programas e dados não são distinguidos e, por fim, tudo é uma expressão S de algum tipo. Novamente, temos apenas um tipo, embora seja um pouco mais estruturado e tenha melhores propriedades do que cadeias de bits.

1.1.3. Calculo λ

No cálculo λ, tudo é uma função. Números, estruturas de dados e até cadeias de bits podem ser representados por funções apropriadas.

1.1.4. Teoria de conjuntos

Na teoria dos conjuntos, tudo é um elemento ou um conjunto de elementos e/ou outros conjuntos. Por exemplo, os inteiros são geralmente representados por conjuntos de conjuntos cujo nível de aninhamento representa a cardinalidade do inteiro, enquanto as funções são representadas por conjuntos possivelmente infinitos de pares ordenados com primeiros componentes únicos.

1.2. Static and Strong Typing

Um tipo é um conjunto de regras que restringem a maneira como os objetos podem interagir com outros objetos, por exemplo, objetos de tipos diferentes não podem interagir entre sí. Para evitar violações de tipo, geralmente impomos uma estrutura de tipos estática aos programas. Linguagens de programação nas quais o tipo de cada expressão pode ser

determinado por análise estática de programa são ditas estaticamente tipadas. A tipagem estática é uma propriedade útil, mas, ha o requisito de que todas as variáveis e expressões sejam vinculadas a um tipo em tempo de compilação. O que é muito restritivo. As linguagens em que todas as expressões são consistentes em tipo são chamadas de linguagens fortemente tipadas. Se uma linguagem for fortemente tipada, seu compilador pode garantir que os programas que ela aceita serão executados sem erros de tipo. Em geral, devemos nos esforçar para uma tipagem forte e adotar a tipagem estática sempre que possível.

1.3. Kinds of Polymorphism

Técnicas de programação que, embora sólidas, são incompatíveis com a vinculação inicial de objetos de programa a um tipo específico. Por exemplo, eles excluem procedimentos genéricos, como classificação, que capturam a estrutura de um algoritmo uniformemente aplicável a uma variedade de tipos.

Linguagens tipadas convencionais, como: Pascal C Java tem a ideia que cada valor e variável pode ser interpretado como sendo de um e apenas um tipo. Esta categoria de classificação nas Linguagens de Programação tem o nome de Linguagens monomórficas. Entretanto, as funções polimórficas são funções que os operandos (parâmetros reais) podem ter mais de um tipo. O Cientista da computação: Christopher S. Strachey em 1967 distinguiu informalmente o Polimorfismo entre dois tipos principais.

- Polimorfismo Universal
  - Paramétrica
  - Inclusão
- Ad-Hoc
  - *Overloading*
  - Coerção

Os tipos de polimorfismo são basicamente tipos cuja operação é aplicável a mais de um típo. No caso do polimorfismo universal, pode-se afirmar que alguns valores têm muitos tipos, enquanto no polimorfismo ad-hoc isso é mais difícil de manter, pois, pode-se assumir a posição de que uma função polimórfica ad-hoc é realmente um pequeno conjunto de funções monomórficas. Ao implementar, uma função polimórfica universal executará o mesmo código para argumentos de qualquer tipo admissível, enquanto uma função polimórfica ad-hoc pode executar um código diferente para cada tipo de argumento.

1.4. The Evolution of Types in Programming Languages

As mais diversas Linguagens foram evoluindo ao longo dos anos como, por exemplo, FORTRAN trabalhava com a distinção de pontos flutuantes e inteiros. Algol 60 tornou esta distinção explícita introduzindo declarações de identificador redundante para variáveis reais e booleanas inteiras. Pascal fornece uma extensão mais limpa de tipos para registros e ponteiros de vetores, bem como tipos definidos pelo usuário. LISP, que descreve o interpretador de linguagem LISP em LISP, mas é muito mais complexa. Ele descreve uma coleção de mais de 75 tipos de objetos de sistema relacionados por uma hierarquia de herança.

1.5. Type Expression Sublanguages

As *Expression Sublanguages* tipadas geralmente incluem tipos básicos, como inteiros e booleanos, e tipos compostos, como matrizes, registros e procedimentos construídos a partir de tipos básicos. Elas devem ser suficientemente ricas para suportar tipos para todos os valores com os quais desejamos calcular, mas suficientemente tratável para permitir a verificação de tipo decidível e eficiente.

1.6. Preview of Fun

Fun é uma linguagem baseada em cálculo lambda que enriquece o cálculo X tipado de primeira ordem com recursos de segunda ordem projetados para modelar polimorfismo e linguagens orientadas a objetos. As extensões sintáticas da expressão de tipo sublinguagem determinadas por esses recursos podem ser resumidas da seguinte forma:

```
Type ::= ... | QuantifiedType
QuantifiedType ::=
    ∀A. Type |                          Universal Quantification
    ∃A. Type |                          Existential Quantification
    ∀A⊆Type. Type | ∃A⊆Type. Type       Bounded Quantification
```

Tipos universalmente quantificados são eles próprios tipos de primeira classe e podem ser parâmetros reais em tal substituição. A quantificação existencial enriquece os recursos de primeira ordem, permitindo tipos de dados abstratos com representação oculta.

2. The λ Calculus

2.1. The Untyped λ-Calculus

A evolução de universos não tipados para universos tipados pode ser ilustrada pelo cálculo λ, inicialmente desenvolvido como uma notação não tipada para capturar a essência da aplicação funcional de operadores a operandos. As expressões no cálculo λ têm a seguinte sintaxe[1]:

```
e ::= x          -- a variable is a λ-expression
e ::= fun(x)e    -- functional abstraction of e
e ::= e(e)       -- operator e applied to operand e
```

A função de identidade e a função sucessora podem ser especificadas no cálculo λ como a seguir. Usamos o valor da palavra-chave para introduzir um novo nome vinculado a um valor ou função:

```
value id = fun(x) x          -- identity function
value succ = fun(x) x+1      -- successor function (for integers)
```

2.2. The Typed λ-Calculus

O cálculo λ digitado é como o cálculo λ, exceto que cada variável deve ser explicitamente digitada quando introduzida como uma variável limitada. Assim, a função sucessora no cálculo λ digitado tem a seguinte forma:

[1]Usamos diversão em vez do λ tradicional para trazer à tona a correspondência com as notações da linguagem de programação

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

2.3. Basic Types, Structured Types, and Recursion

O cálculo λ digitado é geralmente acrescido de vários tipos básicos e estruturados, os quais usaremos:

Unit	the trivial type, with only element ()
Bool	with an if-then-else operation
Int	with arithmetic and comparison operations
Real	with arithmetic and comparison operations
String	with string concatenation (infix) ^

Tipos estruturados podem ser construídos a partir desses tipos básicos por meio de construtores de tipo. Os construtores de tipo em nossa linguagem incluem espaços de função ($\rightarrow$), produtos cartesianos (x), tipos de registro (também chamados de produtos cartesianos rotulados) e tipos de variantes (também chamados de somas disjuntas rotuladas).

3. Anexo: Samir Cambraia

1. FROM UNTYPED TO TYPED UNIVERSES

1.1 Organizing Untyped Universes

Para responder a essa pergunta, examinamos como os tipos surgem em vários domínios da ciência da computação e da matemática. A estrada dos universos não tipados para os tipificados foi seguida muitas vezes, em muitos campos diferentes, e principalmente pelas mesmas razões.

(1) cadeias de bits na memória do computador,
(2) expressões S em LISP puro,
(3) expressões λ no cálculo λ,
(4) conjuntos na teoria dos conjuntos.

O mais concreto deles é o universo das cadeias de bits na memória do computador. “Sem tipo” na verdade significa que há apenas um tipo, e aqui o único tipo é

As expressões S de LISP formam outro universo não tipado, que geralmente é construído em cima do universo de string de bits anterior. Programas e dados não são distinguidos e, em última análise, tudo é uma expressão S de algum tipo. Novamente, temos apenas um tipo , embora seja um pouco mais estruturado e tenha melhores propriedades do que cadeias de bits. Na teoria dos conjuntos, tudo é um elemento ou um conjunto de elementos e / ou outros conjuntos.

Para entender o quão não tipificado esse universo é, é preciso lembrar que a maior parte da matemática, que está repleta de estruturas extremamente ricas e complexas, é representada na teoria dos conjuntos por conjuntos cuja complexidade estrutural reflete a complexidade das estruturas que estão sendo representadas. Por exemplo, os inteiros são geralmente representados por conjuntos de conjuntos cujo nível de aninhamento representa a cardinalidade do inteiro, enquanto as funções são representadas por conjuntos possivelmente infinitos de pares ordenados com primeiros componentes únicos. Assim que começamos a trabalhar em um universo não tipificado, começamos a organizá-lo de diferentes maneiras para diferentes propósitos.

No cálculo λ , algumas funções são escolhidas para representar valores booleanos, outras para representar números inteiros. Na teoria dos conjuntos, alguns conjuntos

são escolhidos para denotar pares ordenados, e alguns conjuntos de pares ordenados são então chamados de funções. Universos não tipificados de objetos computacionais se decompõem naturalmente em subconjuntos com comportamento uniforme. Conjuntos de objetos com comportamento uniforme podem ser nomeados e referidos como tipos.

Por exemplo, todos os inteiros exibem comportamento uniforme por terem o mesmo conjunto de operações aplicáveis. Funções de inteiros para inteiros se comportam uniformemente, pois se aplicam a objetos de um determinado tipo e produzem valores de um determinado tipo. Depois de um grande esforço de organização, então, podemos começar a pensar em universos não tipados como se fossem tipificados.

1.2 Static and Strong Typing

Um tipo pode ser visto como um conjunto de roupas que protege uma representação não tipificada subjacente do uso arbitrário ou não intencional. Ele fornece uma cobertura protetora que oculta a representação subjacente e restringe a maneira como os objetos podem interagir com outros objetos. Em um sistema não tipado, os objetos não tipificados estão nus de modo que a representação subjacente é exposta para que todos possam ver. Violar o sistema de tipos envolve remover o conjunto de roupas de proteção e operar diretamente na representação nua.

Os objetos de um determinado tipo têm uma representação que respeita as propriedades esperadas do tipo de dados. A representação é escolhida para facilitar a execução das operações esperadas em objetos de dados. Quebrar o sistema de tipos permite que uma representação de dados seja manipulada de maneiras que não foram pretendidas, com resultados potencialmente desastrosos. Por exemplo, o uso de um inteiro como um ponteiro pode causar modificações arbitrárias em programas e dados.

Para evitar violações de tipo, geralmente impomos uma estrutura de tipo estática aos programas. Em linguagens como Pascal e Ada, o tipo de variáveis e símbolos de função é definido por declarações redundantes, e o compilador pode verificar a consistência de definição e uso. Em linguagens como ML, declarações explícitas são evitadas sempre que possível, e o sistema pode inferir o tipo de expressões do contexto local, enquanto ainda estabelece um uso consistente. Linguagens de programação nas quais o tipo de cada expressão pode ser determinado por análise estática de programa são ditas estaticamente tipadas.

A tipagem estática é uma propriedade útil, mas o requisito de que todas as variáveis e expressões sejam vinculadas a um tipo em tempo de compilação às vezes é muito

restritivo. As linguagens em que todas as expressões são consistentes em tipo são chamadas de linguagens fortemente tipadas. Se uma linguagem for fortemente tipada, seu compilador pode garantir que os programas que ela aceita serão executados sem erros de tipo. Em geral, devemos nos esforçar para uma tipagem forte e adotar a tipagem estática sempre que possível.

A tipagem estática permite que inconsistências de tipo sejam descobertas em tempo de compilação e garante que os programas executados sejam consistentes com o tipo.

técnicas de programação que, embora sólidas, são incompatíveis com a vinculação inicial de objetos de programa a um tipo específico. Por exemplo, eles excluem procedimentos genéricos, como classificação, que capturam a estrutura de um algoritmo uniformemente aplicável a uma variedade de tipos.

Linguagens tipadas convencionais, como Pascal, baseiam-se na ideia de que funções e procedimentos e, portanto, seus operandos, têm um tipo único. Essas linguagens são ditas monomórficas, no sentido de que cada valor e variável pode ser interpretado como sendo de um e apenas um tipo. Linguagens de programação monomórficas podem ser contrastadas com linguagens polimórficas nas quais alguns valores e variáveis podem ter mais de um tipo. Funções polimórficas são funções cujos operandos podem ter mais de um tipo.

Os tipos polimórficos podem ser definidos como tipos cujas operações são aplicáveis a operandos de mais de um tipo. Strachey fez uma distinção, informalmente, entre dois tipos principais de polimorfismo. No caso do polimorfismo universal, pode-se afirmar com confiança que alguns valores têm muitos tipos, enquanto no polimorfismo ad-hoc isso é mais difícil de manter, pois pode-se assumir a posição de que uma função polimórfica ad-hoc é realmente um pequeno conjunto de funções monomórficas. Em termos de implementação, uma função polimórfica universal executará o mesmo código para argumentos de qualquer tipo admissível, enquanto uma função polimórfica ad-hoc pode executar um código diferente para cada tipo de argumento.

Existem dois tipos principais de polimorfismo universal, ou seja, duas maneiras principais pelas quais um valor pode ter muitos tipos. No polimorfismo paramétrico, uma função polimórfica tem um parâmetro de tipo implícito ou explícito que determina o tipo do argumento para cada aplicação dessa função. As funções que exibem polimorfismo paramétrico também são chamadas de funções genéricas. Por

exemplo, a função de comprimento de listas de tipo arbitrário para inteiros é chamada de função de comprimento genérica.

Uma função genérica é aquela para a qual pode funcionar.

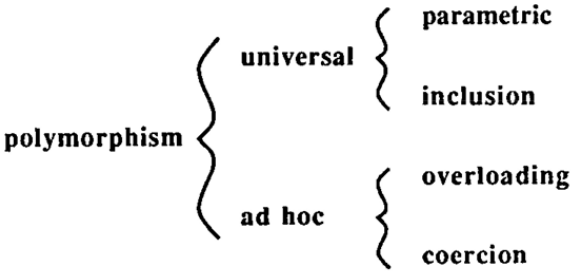

Figure 1. Varieties of polymorphism.

Argumentos de muitos tipos, geralmente fazendo o mesmo tipo de trabalho, independentemente do tipo de argumento. Se considerarmos uma função genérica como um valor único, ela tem muitos tipos funcionais e, portanto, é polimórfica. As funções genéricas Ada são um caso especial desse conceito de genérico. Na sobrecarga, o mesmo nome de variável é usado para denotar funções diferentes e o contexto é usado para decidir qual função é denotada por uma instância particular do nome.

As coerções podem ser fornecidas estaticamente, inserindo-as automaticamente entre argumentos e funções em tempo de compilação, ou podem ter que ser determinadas dinamicamente por testes de tempo de execução nos argumentos. Isso é particularmente verdadeiro quando se considera linguagens não digitadas e linguagens interpretadas.

```
3 + 4
3.0 + 4
3 + 4.0
3.0 + 4.0
```

Aqui, o polimorfismo ad-hoc de + pode ser explicado de uma das seguintes maneiras:

- O operador + tem quatro significados sobrecarregados, um para cada uma das quatro combinações de tipos de argumento.

- O operador + tem dois significados sobrecarregados, correspondendo a inteiro e adição real. Quando um dos argumentos é do tipo inteiro e o outro é do tipo real, o argumento inteiro é forçado para o tipo real.
- O operador + é definido apenas para adição real, e argumentos inteiros são sempre forçados para reais correspondentes.

Neste exemplo, podemos considerar a mesma expressão para exibir sobrecarga ou coerção, ou ambos (e também para alterar o significado), dependendo de uma decisão de implementação. Nossa definição de polimorfismo é aplicável apenas a linguagens com uma noção muito clara de tipo e valor. Em particular, deve haver uma distinção clara entre o tipo inerente de um objeto e o tipo aparente de suas representações sintáticas em linguagens que permitem sobrecarga e coerção. Essas questões são discutidas mais detalhadamente abaixo.

Se vermos um tipo como especificando parcialmente o comportamento, ou uso pretendido, de valores associados, então os sistemas de tipo monomórfico restringem os objetos a ter apenas um comportamento, enquanto os sistemas de tipo polimórfico permitem que os valores sejam associados a mais de um comportamento. Linguagens estritamente monomórficas são muito restritivas em seu poder expressivo porque não permitem que valores, ou mesmo símbolos sintáticos que denotam valores, exibam comportamentos diferentes em contextos de uso diferentes. Línguas como Pascal e Ada têm maneiras de relaxar o monomorfismo estrito, mas o polimorfismo é a exceção e não a regra e podemos dizer que eles são principalmente monomórficos. As exceções reais e aparentes à regra de tipagem monomórfica em linguagens convencionais incluem
(1) Sobrecarga: constantes inteiras podem ter tipo inteiro e real. Operadores como + são aplicáveis a argumentos inteiros e reais.
(2) Coerção: Um valor inteiro pode ser usado onde um real é esperado e vice-versa.
(3) Subtipagem: Elementos de um tipo de subfaixa também pertencem a tipos de superfaixa.
(4) Compartilhamento de valor: nil em Pascal é uma constante que é compartilhada por todos os tipos de ponteiro.
Vamos ver como eles se encaixam na descrição anterior de diferentes tipos de polimorfismo. As conversões de tipo necessárias devem ser determinadas pelo sistema, inseridas no programa e usadas pelo compilador para gerar o código de

conversão de tipo necessário. As coerções são essencialmente uma forma de abreviatura que pode reduzir o tamanho do programa e melhorar a legibilidade do programa, mas também pode causar erros de sistema sutis e às vezes perigosos. Subtipagem é uma instância de polimorfismo de inclusão.

A ideia de um tipo ser um subtipo de outro tipo é útil não apenas para subfaixas de tipos ordenados, como inteiros, mas também para estruturas mais complexas, como um tipo que representa Toyotas, que é um subtipo de um tipo mais geral, como Veículos. Cada objeto de um subtipo pode ser usado em um contexto de supertipo, no sentido de que todo Toyota é um veículo e pode ser operado por todas as operações aplicáveis aos veículos. O compartilhamento de valor é um caso especial de polimorfismo paramétrico. O fato de um objeto com muitos tipos ser uniformemente representado para todos os tipos é característico do polimorfismo paramétrico.

Como está implícito na escolha dos nomes, o polimorfismo universal é considerado polimorfismo verdadeiro, enquanto o polimorfismo ad-hoc é algum tipo de polimorfismo aparente.

Na implementação, as representações são escolhidas com muito cuidado, de forma que nenhuma coerção seja necessária ao usar um objeto de um subtipo no lugar de um objeto do supertipo. Da mesma forma, as operações têm o cuidado de interpretar as representações uniformemente para que possam funcionar uniformemente em elementos de subtipos e supertipos. O símbolo + pode ser sobrecarregado para denotar ao mesmo tempo soma inteira, soma real e concatenação de string. O uso do mesmo símbolo para essas três operações reflete uma semelhança aproximada da estrutura algébrica, mas viola os requisitos do monomorfismo.

A ambigüidade geralmente pode ser resolvida pelo tipo de operandos imediatos de um operador sobrecarregado, mas isso pode não ser suficiente. ALGOL 68 é bem conhecido por seu esquema de coerção barroco. Os problemas a serem resolvidos aqui são muito semelhantes à sobrecarga, mas, além disso, as coerções têm efeitos de tempo de execução. As classes de Simula são tipos definidos pelo usuário, organizados em uma hierarquia de inclusão simples em que cada classe tem uma superclasse imediata única.

Os objetos e procedimentos de Simula são polimórficos porque um objeto de uma subclasse pode aparecer sempre que um objeto de uma de suas superclasses é necessário. Smalltalk [Goldberg e Robson 19831, embora uma linguagem não tipada, também popularizou essa visão de polimorfismo. Mais recentemente, LISP Flavors [Weinreb e Moon 19811 estendeu este estilo de polimorfismo para múltiplas superclasses imediatas, e Amber [Cardelli 19851 o estende ainda mais para funções de ordem superior.

Finalmente, devemos mencionar procedimentos genéricos do tipo encontrado em Ada, que são modelos parametrizados que devem ser instanciados com valores de parâmetros reais antes de serem usados. O polimorfismo dos procedimentos genéricos de Ada é semelhante ao polimorfismo paramétrico de linguagens como ML, mas é especializado em tipos específicos de parâmetros. Os parâmetros podem ser parâmetros de tipo, parâmetros de procedimento ou parâmetros de valor. Os procedimentos genéricos com parâmetros de tipo são polimórficos no sentido de que os parâmetros de tipo formal podem assumir diferentes tipos reais para diferentes instanciações.

O polimorfismo de tipo genérico em Ada, no entanto, é sintático, pois a instanciação genérica é realizada em tempo de compilação com valores de tipo reais que devem ser determináveis em tempo de compilação. A semântica dos procedimentos genéricos é a macroexpansão conduzida pelo tipo dos argumentos. Assim, procedimentos genéricos podem ser considerados abreviações para conjuntos de procedimentos monomórficos.

1.4 The Evolution of Types in Programming Languages

Nas primeiras linguagens de programação, a computação era identificada com a computação numérica e os valores podiam ser vistos como tendo um único tipo aritmético. Já em 1954, no entanto, em FORTRAN foi considerado conveniente distinguir entre números inteiros e números de ponto flutuante, em parte porque as diferenças na representação de hardware tornavam a computação de inteiros mais econômica e em parte porque o uso de inteiros para iteração e computação de matriz era logicamente diferente do uso de números de ponto flutuante para computação numérica. FORTRAN distingue entre variáveis inteiras e de ponto flutuante pela primeira letra de seus nomes. O ALGOL 60 tornou esta distinção explícita introduzindo declarações de identificador redundantes para variáveis reais e booleanas inteiras.

ALGOL 60 foi a primeira linguagem significativa a ter uma noção explícita de tipo e requisitos associados para verificação de tipo em tempo de compilação. Seus requisitos de estrutura de bloco permitiam não apenas o tipo, mas também o escopo das variáveis a serem verificadas em tempo de compilação. A noção ALGOL 60 de tipo foi estendida a classes de valores mais ricas na década de 1960. Das inúmeras linguagens tipadas desenvolvidas durante este período, PL / I, Pascal, ALGOL 68 e Simula, são notáveis por suas contribuições para a evolução do conceito de tipo.

PL / I tenta combinar os recursos de FORTRAN, ALGOL 60, COBOL e LISP. Pascal fornece uma extensão mais limpa de tipos para registros e ponteiros de arrays, bem como tipos definidos pelo usuário. No entanto, Pascal não define equivalência de tipo, de forma que a questão de quando duas expressões de tipo denotam o mesmo

tipo depende da implementação. Pascal deixa lacunas na especificação de tipo forte ao não exigir que o tipo completo de procedimentos passados como parâmetros seja especificado e ao permitir que o campo de tag de registros variantes seja manipulado de forma independente.

As ambigüidades e inseguranças do sistema do tipo Pascal são discutidas em Welsh et al. ALGOL 68 tem uma noção de tipo mais rigorosa do que Pascal, com uma noção bem definida de equivalência de tipo. A noção de tipo é estendida para incluir procedimentos como valores de primeira classe. ALGOL 68 definiu cuidadosamente regras para coerção, usando desreferenciamento, desprocedimento, alargamento, remo, união e anulação para transformar valores para o tipo necessário para computação posterior.

Sua noção de tipo inclui classes cujas instâncias podem ser atribuídas como valores de variáveis com valor de classe e podem persistir entre a execução dos procedimentos que contêm. ML introduziu a noção de polimorfismo paramétrico em linguagens. Os tipos de ML podem conter variáveis de tipo instanciadas para diferentes tipos em diferentes contextos.

A estrutura histórica acima fornece uma base para uma discussão mais profunda das relações entre tipos, abstração de dados e polimorfismo em linguagens de programação reais. Consideramos as abstrações de dados não digitados (pacotes) de Ada, indicamos o impacto na metodologia de exigir que as abstrações de dados tenham tipo e herança, discutimos a interpretação da herança como polimorfismo de subtipo e examinamos a relação entre o polimorfismo de subtipo de Smalltalk e o polimorfismo paramétrico de ML. Ada tem uma rica variedade de módulos, incluindo subprogramas para suportar programação orientada a procedimentos, pacotes para suportar abstrações de dados e tarefas para suportar programação simultânea. Mas tem uma noção relativamente fraca de tipo, excluindo procedimentos e pacotes do domínio de objetos tipados e incluindo tipos de tarefas relativamente tarde no processo de design como uma reflexão tardia. Sua escolha de equivalência de nome como equivalência de tipo é mais fraca do que a noção de equivalência estrutural usada em ALGOL 68. Sua severa restrição contra coerção implícita enfraquece sua capacidade de fornecer operações polimórficas aplicáveis a operandos de muitos tipos. Os pacotes em Ada têm uma especificação de interface de componentes nomeados que podem ser variáveis simples, procedimentos, exceções e até mesmo tipos. Eles podem ocultar um estado local por um tipo de dados privado ou no corpo do pacote. Os pacotes são como instâncias de registro em ter uma interface de usuário de componentes nomeados. Os pacotes Ada diferem dos registros porque os componentes do registro devem ser valores digitados, enquanto os componentes do pacote podem ser procedimentos, exceções, tipos e outras entidades nomeadas. Uma vez que os pacotes não são eles próprios tipos, eles não podem ser parâmetros, componentes de estruturas ou valores de variáveis de ponteiro [Wegner 19831. Pacotes em Ada são objetos de segunda classe, enquanto instâncias de classe em Simula ou objetos em linguagens orientadas a objetos são os primeiros objetos -class. As diferenças de comportamento entre pacotes e registros em Ada são evitadas em linguagens

orientadas a objetos, estendendo a noção de tipo para procedimentos e abstrações de dados. No contexto desta discussão, é útil definir as linguagens orientadas a objetos como extensões das linguagens orientadas a procedimentos que suportam abstrações de dados digitados com herança. Assim, dizemos que uma linguagem é orientada a objetos se e somente se satisfizer os seguintes requisitos:

- Ele suporta objetos que são abstrações de dados com uma interface de operações nomeadas e um estado local oculto.
- Os objetos têm um tipo de objeto associado.
- Os tipos podem herdar atributos de supertipos.

Esses requisitos podem ser resumidos como:

orientado a objetos = abstrações de dados + tipos de objetos + herança de tipo.

A utilidade desta definição pode ser ilustrada considerando o impacto de cada um desses requisitos na metodologia. A abstração de dados por si só fornece uma maneira de organizar dados com operações associadas que difere consideravelmente da metodologia tradicional de programação orientada a procedimentos. A realização da metodologia de abstração de dados foi um dos objetivos primários da Ada, e essa metodologia é descrita detalhadamente na literatura da Ada em publicações como Booth [1983]. No entanto, Ada satisfaz apenas o primeiro de nossos três requisitos para a programação orientada a objetos, e é interessante examinar o impacto dos tipos de objetos e da herança na metodologia de abstração de dados [Hendler e Wegner 19861. O requisito de que todos os objetos tenham um tipo permite que os objetos sejam valores de primeira classe, para que possam ser gerenciados como estruturas de dados dentro da linguagem, conforme A utilidade desta definição pode ser ilustrada considerando o impacto de cada um desses requisitos na metodologia. A abstração de dados por si só fornece uma maneira de organizar dados com operações associadas que difere consideravelmente da metodologia tradicional de programação orientada a procedimentos. A realização da metodologia de abstração de dados foi um dos objetivos primários da Ada, e essa metodologia é descrita detalhadamente na literatura da Ada em publicações como Booth [1983]. No entanto, Ada satisfaz apenas o primeiro de nossos três requisitos para a programação orientada a objetos, e é interessante examinar o impacto dos tipos de objeto e da herança na

metodologia de abstração de dados [Hendler e Wegner 19861. O requisito de que todos os objetos tenham um tipo permite que os objetos sejam valores de primeira classe para que possam ser gerenciados como estruturas de dados dentro da linguagem como

Quando dizemos que uma função paramétrica é aplicável a listas de tipo arbitrário, realmente queremos dizer que ela pode ser especializada, fornecendo um parâmetro de tipo T, e que a função especializada pode então ser aplicada aos operandos especializados. Essa distinção entre uma função paramétrica e suas versões especializadas é borrada em linguagens como ML porque os parâmetros de tipo omitidos pelo usuário são reintroduzidos automaticamente pelo mecanismo de inferência de tipo. Supertipos em linguagens orientadas a objetos podem ser vistos como tipos paramétricos cujo parâmetro é omitido pelo usuário. Veremos abaixo que Fun tem parâmetros de tipo explícitos para tipos e supertipos paramétricos, a fim de fornecer um modelo uniforme para polimorfismo paramétrico e de subtipo.

1.5 Type Expression Sublanguages

As sublinguagens da expressão de tipo geralmente incluem tipos básicos, como inteiros e booleanos, e tipos compostos, como matrizes, registros e procedimentos construídos a partir de tipos básicos:

```
Type ::= BasicType | ConstructedType
BasicType ::= Int | Bool | ...
ConstructedType ::= Array(Type) | Type → Type | ...
```

A expressão de tipo sublinguagem deve ser suficientemente rica para suportar tipos para todos os valores com os quais desejamos calcular, mas suficientemente tratável para permitir a verificação de tipo decidível e eficiente. Um dos objetivos deste artigo é examinar os trade-offs entre riqueza e tratabilidade para sublinguagens de expressão de tipo de linguagens fortemente tipadas. A expressão de tipo sublinguagem geralmente pode ser especificada por uma gramática livre de contexto. No entanto, estamos interessados não apenas na sintaxe da expressão de tipo sub-linguagem, mas também em sua semântica.

Isto é, estamos interessados em quais tipos denotam e nas relações entre as expressões de tipo. Relações de similaridade entre expressões de tipo que permitem que uma expressão de tipo denote mais de um tipo, ou seja compatível com muitos tipos, são chamadas de polimorfismo. A capacidade de expressar relações entre tipos envolve alguma capacidade de realizar cálculos em tipos para determinar se eles satisfazem a relação desejada.

Fun é uma linguagem baseada em cálculo λ que enriquece o cálculo λ tipado de primeira ordem com recursos de segunda ordem projetados para modelar polimorfismo e linguagens orientadas a objetos.

A seção 2 revisa o cálculo λ não tipado e digitado e desenvolve recursos de primeira ordem da sub-linguagem de expressão do tipo Fun. Fun tem os tipos básicos Bool, Int, Real, String e tipos construídos para registro, variante, função e tipos recursivos. A seção 3 faz uma breve revisão de modelos teóricos de tipos relacionados a recursos de Diversão, especialmente modelos que visualizam tipos como conjuntos de valores. As seções 4, 5 e 6, respectivamente, aumentam o cálculo λ de primeira ordem com quantificação universal para realizar tipos parametrizados, quantificação existencial para realizar abstração de dados e quantificação limitada para realizar herança de tipo.

```
Type ::= ... | QuantifiedType
QuantifiedType ::=
    ∀A. Type |                          Universal Quantification
    ∃A. Type |                          Existential Quantification
    ∀A⊆Type. Type | ∃A⊆Type. Type       Bounded Quantification
```

A quantificação universal enriquece o cálculo λ de primeira ordem com tipos parametrizados que podem ser especializados ao substituir parâmetros de tipo reais por parâmetros quantificados universalmente. Tipos universalmente quantificados são eles próprios tipos de primeira classe e podem ser parâmetros reais em tal substituição. A interação da quantificação universal e existencial é ilustrada na Seção 5.3 para o caso de pilhas com um tipo de elemento quantificado universalmente e uma representação de dados ocultos quantificada existencialmente.

A quantificação universal enriquece o cálculo λ de primeira ordem com tipos parametrizados que podem ser especializados ao substituir parâmetros de tipo reais por parâmetros quantificados universalmente. Tipos universalmente quantificados são eles próprios tipos de primeira classe e podem ser parâmetros reais em tal substituição. A interação da quantificação universal e existencial é ilustrada na Seção 5.3 para o caso de pilhas com um tipo de elemento quantificado universalmente e uma representação de dados ocultos quantificada existencialmente.

A evolução de universos não tipados para universos tipados pode ser ilustrada pelo cálculo λ, inicialmente desenvolvido como uma notação não tipada para capturar a essência da aplicação funcional de operadores a operandos. As expressões no cálculo λ têm a seguinte sintaxe (usamos diversão em vez do λ tradicional para trazer à tona a correspondência com as notações da linguagem de programação):

```
e ::= x          -- a variable is a λ-expression
e ::= fun(x)e    -- functional abstraction of e
e ::= e(e)       -- operator e applied to operand e
```

A função de identidade e a função sucessora podem ser especificadas no cálculo λ como segue (com algum açúcar sintático explicado mais tarde). Usamos o valor da palavra-chave para introduzir um novo nome vinculado a um valor ou função:

```
value id = fun(x) x          -- identity function
value succ = fun(x) x+1      -- successor function (for integers)
```

A função de identidade pode ser aplicada a uma expressão λ arbitrária e sempre produz a própria expressão λ. Para definir a adição em inteiros no cálculo λ puro, escolhemos uma representação para inteiros e definimos a operação de adição de forma que seu efeito nas expressões λ que representam os inteiros n e m produza a expressão λ que representa n + m . A função sucessora deve ser aplicada apenas a expressões λ que representam números inteiros e sugerem uma noção de tipagem.

A ideia de expressões λ operando em funções para produzir outras funções pode ser ilustrada pela função duas vezes, que tem a seguinte forma:

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

A aplicação de duas vezes à função sucessora produz uma expressão λ que calcula o sucessor do sucessor:

```
twice(succ)          ⇒  fun(y) succ(succ(y))
twice (fun(x)x+1)    ⇒  fun(y) (fun(x)x+1) ((fun(x)x+1) (y))
```

A discussão acima ilustra como os tipos surgem quando especializamos uma notação não tipada, como o cálculo λ, para realizar determinados tipos de computação, como a aritmética de inteiros. Na próxima seção, introduzimos tipos explícitos no cálculo λ. A notação resultante é semelhante à notação funcional em linguagens de programação digitadas tradicionais.

2.2 The Typed λ-Calculus

O cálculo λ digitado é como o cálculo λ, exceto que cada variável deve ser explicitamente digitada quando introduzida como uma variável limitada. Assim, a função sucessora no cálculo λ digitado tem a seguinte forma:

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

A função duas vezes de inteiros para inteiros tem um parâmetro f cujo tipo é Int + Int (o tipo de funções de inteiros para inteiros) e pode ser escrita da seguinte forma:

```
value twice = fun(f: Int + Int) fun (y:Int) f(f(y))
```

Essa notação se aproxima da especificação funcional em linguagens de programação tipadas, mas omite a especificação do tipo de resultado. Podemos denotar o tipo de resultado com uma palavra-chave de retorno da seguinte maneira:

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

No entanto, o tipo de resultado pode ser determinado a partir da forma do corpo da função x + 1. Omitimos as especificações do tipo de resultado por motivos de

brevidade. Os mecanismos de inferência de tipo que permitem que essas informações sejam recuperadas durante a compilação são discutidos em uma seção posterior. As declarações de tipo são introduzidas pelo tipo de palavra-chave. Ao longo deste documento, os nomes dos tipos começam com letras maiúsculas, enquanto os nomes dos valores e funções começam com letras minúsculas:

type IntPair = Int x Int
type IntFun = Int + Int

As declarações de tipo apresentam nomes (abreviações) para expressões de tipo; eles não criam novos tipos em nenhum sentido. Isso às vezes é expresso dizendo que usamos equivalência estrutural em tipos em vez de equivalência de nomes: dois tipos são equivalentes quando têm a mesma estrutura, independentemente dos nomes que usamos como abreviações. O fato de um valor v ter um tipo T é indicado por v: T:

(3,4): IntPair
succ: IntFun

Sobre tipos de compreensão, abstração de dados e polimorfismo

Não precisamos introduzir variáveis por declarações de tipo da forma var: T porque o tipo de uma variável pode ser determinado a partir da forma do valor atribuído. Por exemplo, o fato de que intPair abaixo tem o tipo IntPair pode ser determinado pelo fato de que (3, 4) tem o tipo Int X Int, que foi declarado equivalente a IntPair:

value intPair = (3,4)

No entanto, se quisermos indicar o tipo de uma variável como parte de sua inicialização, podemos fazer isso pelo valor de notação var: T = valor:

value intPair: IntPair = (3,4)
value succ: Int --f Int = fun(x: Int) x + 1

Variáveis locais podem ser declaradas pela construção let-in, que introduz uma nova variável inicializada (após let) em um escopo local (uma expressão após). O valor da construção é o valor dessa expressão:

let a = 3 in a + 1 yields 4

Se quisermos especificar tipos, também podemos escrever

let a : Int = 3 in a + 1

A construção let-in pode ser definida em termos de expressões divertidas básicas:

let a : T = M in N I (fun(a:T) N)(M)

2.3 Basic Types, Structured Types, and Recursion

O cálculo λ digitado é geralmente acrescido de vários tipos de tipos básicos e estruturados. Para os tipos básicos, devemos usar

Unit	the trivial type, with only element ()
Bool	with an if-then-else operation
Int	with arithmetic and comparison operations
Real	with arithmetic and comparison operations
String	with string concatenation (infix) ^

Tipos estruturados podem ser construídos a partir desses tipos básicos por meio de construtores de tipo. Os construtores de tipo em nossa linguagem incluem espaços de função (-->), produtos cartesianos (x), tipos de registro (também chamados de produtos cartesianos rotulados) e tipos de variantes (também chamados de somas disjuntas rotuladas). Um par é um elemento de um tipo de produto cartesiano, por exemplo,

```
value p = 3,true : Int × Bool
```

Operações em pares são seletores para o primeiro e segundo componentes:

```
fst(p)    yields  3
snd(p)    yields  true
```

Um registro é um conjunto não ordenado de valores rotulados. Seu tipo pode ser especificado indicando o tipo associado a cada um de seus rótulos. Um tipo de registro é denotado por uma sequência de tipos rotulados, separados por vírgulas e entre chaves:

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

Um registro desse tipo pode ser criado inicializando cada uma das gravadoras com um valor do tipo requerido. Ele é escrito como uma sequência de valores rotulados separados por vírgulas e entre chaves:

```
value r: ARecordType = {a = 3, b = true, c = "abcd"}
```

Os rótulos devem ser exclusivos em qualquer registro ou tipo de registro. A única operação em registros é a seleção de campo, denotada pela notação de ponto usual:

```
r.b  yields  true
```

Uma vez que as funções são valores de primeira classe, os registros podem, em geral, ter componentes de função:

```
type FunctionRecordType = {f1: Int → Int, f2: Real → Real}
value functionRecord = {f1 = succ, f2 = sin}
```

Um tipo de registro pode ser definido em termos de tipos de registro existentes por um operador &, que concatena dois tipos de registro:

```
type NewFunctionRecordType = FunctionRecordType & {f3: Bool → Bool}
```

Isso é uma abreviatura, em vez de escrever os três campos f1, f2 e f3 explicitamente. É válido apenas quando usado em tipos de registro e quando nenhum rótulo duplicado está envolvido. Uma estrutura de dados pode se tornar local e privada para uma coleção de funções por meio de declarações let-in. Registros com componentes de função são uma forma particularmente conveniente de conseguir isso; aqui está uma variável de contador privada compartilhada por um incremento e uma função total:

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

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

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

Um tipo variante também é formado a partir de um conjunto não ordenado de tipos rotulados, que agora estão entre colchetes:

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

Um elemento desse tipo pode ser um inteiro rotulado como a, um booleano rotulado como b ou uma string rotulada como C:

```
value v1 = [a = 3]
value v2 = [b = true]
value v3 = [c = "abcd"]
```

A única operação nas variantes é a seleção de caso. Uma declaração de caso para uma variante do tipo AVariantType tem o seguinte formato:

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

onde em cada caso uma nova variável é introduzida e ligada aos respectivos conteúdos da variante. Essa variável pode então ser usada na respectiva ação.

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

onde o conteúdo do objeto variante x é vinculado aos identificadores anInt, aBool ou astring, dependendo do caso. No entanto, todos os cálculos expressos no cálculo λ digitado devem terminar. A função fatorial pode ser expressa como

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

Para simplificar, assumimos que os únicos valores que podem ser definidos recursivamente são funções. Finalmente, apresentamos definições de tipo recursivo. Isso nos permite, por exemplo, definir o tipo de listas de inteiros fora do registro e tipos de variantes:

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

Uma lista de inteiros é nula (representada como [nil = ()]) ou os contras de um inteiro e uma lista de inteiros (representada como, por exemplo, [cons = {cabeça = 3, cauda = nulo)]).

4. Anexo: Guilherme Cosso

1. FROM UNTYPED TO TYPED UNIVERSES

1.1. Organizing Untyped Universes

Para compreendermos os diferentes tipos que uma Linguagem de Programação contém, primeiramente devemos nos perguntar: “Por que os tipos são necessários para as LPs”. Conseguimos responder está pergunta através da evolução da matemática e da computação do universo não tipado para o tipado, vamos analisar e dividir este universo em 4 partes:

a) *Strings de bits na memória do computador:*

O mais concreto dos tipos. Considerado “não tipado” porém representado como cadeias de bits: caracteres, números, ponteiros, dados estruturados, programas.

O pedaço de memória bruta, geralmente não temos como dizer o que está sendo representado este significado é dado por um interpretador externo.

b) $Expressões - S\ em\ LISP\ puro:$

Baseado nas cadeias de bits as Expressões-S o LISP é um pouco mais estruturado (átomos e células).

c) $Expressões - \lambda$ *em calculo* λ *:*

No cálculo λ, tudo é uma função. Números, estruturas de dados e até cadeias de bits podem ser representados por funções apropriadas

d) *Conjunto da teoria de conjuntos:*

Neste tópico , tudo é um elemento ou um conjunto de elementos e / ou outros conjuntos.

A partir da classificação do universo não tipado podemos inferir tipos para integrarem gerando um universo tipado por meio da categorização objetos de acordo com seu uso e comportamento.

1.2. Static and Strong Typing

Um dos principais objetivos de uma LP ao implementar um universo tipado é eliminar inconsistências. O compilador conferir se o tipo representado está de acordo para evitar incoerências e erros intencionais ou não no sistema como se fosse uma camada protetora.

A técnica descrita acima e realizada para facilitar a execução das operações esperadas em objetos de dados além de erros no programa em funcionamento serem todos lógicos, uma vez que erros de tipos não são passados desapercebido pelo compilador. No entretanto o programa impõe ao programador uma disciplina de programação que torna os programas mais estruturados e fáceis de ler.

1.3. Kinds of Polymorphism:

Linguagens tipadas convencionais, como: Pascal C Java tem a ideia que cada valor e variável pode ser interpretado como sendo de um e apenas um

tipo. Esse tipo de classificação nas Linguagens de Programação tem o nome de Linguagens monomórficas.

Entretanto as funções polimórficas são funções que os operandos (parâmetros reais) podem ter mais de um tipo. O Cientista da computação **Christopher S. Strachey** em 1967 distinguiu informalmente o Polimorfismo entre dois tipos principais.

- Polimorfismo universal
  - Parametric.
  - Inclusion
- Ad-Hoc.
  - Overloading
  - Coercion

1.4. The Evolution of Types in Programming Languages

As mais diversas Lps foram evoluindo ao longo dos anos FORTRAN trabalhava com a distinção de pontos flutuantes e inteiros. ALGOL 60 tornou esta distinção explícita introduzindo declarações de identificador redundantes para variáveis reais e booleanas inteiras.

Pascal fornece uma extensão mais limpa de tipos para registros e ponteiros de arrays, bem como tipos definidos pelo usuário.

LISP, que descreve o interpretador de linguagem LISP em LISP, mas é muito mais complexa. Ele descreve uma coleção de mais de 75 tipos de objetos de sistema relacionados por uma hierarquia de herança.

1.5. Type Expression Sublanguages

À medida que o conjunto de tipos de uma linguagem de programação se torna mais rico e seu conjunto de tipos definíveis torna-se infinito, torna-se útil definir o conjunto de tipos por uma sublinguagem de expressão de tipo.

A capacidade de expressar relações entre tipos envolve alguma capacidade de realizar cálculos em tipos para determinar se eles satisfazem a relação desejada. Tais cálculos poderiam, em princípio, ser tão poderosos quanto os cálculos executados em valores.

1.6. Preview of Fun

Fun é uma linguagem baseada em cálculo lambda que enriquece o cálculo X tipado de primeira ordem com recursos de segunda ordem projetados para modelar polimorfismo e linguagens orientadas a objetos.

As extensões sintáticas da expressão de tipo sublinguagem determinadas por esses recursos podem ser resumidas da seguinte forma:

Type ::= . . . (QuantifiedType QuantifiedType ::= V

A. Type 1 Universal Quantification 3

A. Type 1 Existential Quantification V

AEType. Type 1 3Adype. Type Bounded Quantification

Tipos universalmente quantificados são eles próprios tipos de primeira classe e podem ser parâmetros reais em tal substituição. A quantificação existencial enriquece os recursos de primeira ordem, permitindo tipos de dados abstratos com representação oculta.

2. THE X-CALCULUS

2.1 The Untyped X-Calculus

A evolução de universos não tipados para universos tipados pode ser ilustrada pelo cálculo X, inicialmente desenvolvido como uma notação não tipada para capturar a essência da aplicação funcional de operadores a operandos. As expressões no cálculo X têm a seguinte sintaxe (usamos diversão em vez do X tradicional para trazer à tona a correspondência com as notações da linguagem de programação):

2.2 The Typed X-Calculus

O cálculo X digitado é como o cálculo X, exceto que cada variável deve ser explicitamente digitada quando introduzida como uma variável limitada. Assim, a função sucessora no cálculo X digitado tem a seguinte forma:

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

2.3 Basic Types, Structured Types, and Recursion

O cálculo X digitado é geralmente acrescido de vários tipos de tipos básicos e estruturados.Formados por tipos basicos Unit Bool Int Real

5. Anexo: Iyan Lucas

1. DE UNIVERSOS NÃO TIPADOS A UNIVERSOS TIPADOS

1.1 Organizando Universos

O caminho dos universos não tipados para os tipificados foi seguida muitas vezes, em muitos campos diferentes, e principalmente pelas mesmas razões.

(1) cadeias de bits na memória do computador,
(2) expressões S em LISP puro
(3) λ-expressões no λ-calculus
(4) define a teoria dos conjuntos.

1 - O mais concreto deles é o universo das cadeias de bits na memória do computador. “Untype” na verdade significa que existe apenas um tipo, e aqui o único tipo é a palavra de memória, que é uma sequência de bits de tamanho fixo. O significado de um pedaço de memória é criticamente determinado por uma interpretação externa de seu conteúdo. As expressões S do LISP formam outro universo não tipado, que geralmente é construído sobre o universo de string de bits anterior.

Programas e dados não são distinguidos e, em última análise, tudo é uma expressão λ de algum tipo. No cálculo-λ, tudo é uma função. Números, estruturas de dados e até cadeias de bits podem ser representados por funções apropriadas. Assim que começamos a trabalhar em um universo não tipificado, começamos a organizá-lo de diferentes maneiras para diferentes propósitos.

2 - No LISP, algumas “ expressões S” são chamadas de listas, enquanto outras formam programas legais.
3 - No cálculo-λ, algumas funções são escolhidas para representar valores booleanos, outras para representar inteiros. Na teoria dos conjuntos, alguns conjuntos são escolhidos para denotar pares ordenados, e alguns conjuntos de pares ordenados são então chamados de funções. Universos não tipificados de objetos computacionais se decompõem naturalmente em subconjuntos com comportamento uniforme.

4 - Conjuntos de objetos com comportamento uniforme podem ser nomeados e são chamados de tipos. Por exemplo, todos os inteiros exibem comportamento uniforme por terem o mesmo conjunto de operações aplicáveis. Funções de inteiros para inteiros se comportam uniformemente, pois se aplicam a objetos de um determinado tipo e produzem valores de um determinado tipo.
Depois de um grande esforço de organização, então, podemos começar a pensar em universos não tipados como se fossem tipificados.

Digitação estática e forte

Na matemática, assim como na programação, os tipos impõem restrições que ajudam a impor a correção. Um tipo pode ser visto como um conjunto de roupas que protege uma representação não tipificada subjacente do uso arbitrário ou não intencional. Ele fornece uma cobertura protetora que oculta a representação

subjacente e restringe a maneira como os objetos podem interagir com outros objetos. Em um sistema não tipado, os objetos não tipificados estão nus de modo que a representação subjacente é exposta para que todos possam ver.

Violar o sistema de tipos envolve remover o conjunto de roupas de proteção e operar diretamente na representação nua. Os objetos de um determinado tipo têm uma representação que respeita as propriedades esperadas do tipo de dados. A representação é escolhida para facilitar a execução das operações esperadas em objetos de dados. Quebrar o sistema de tipos permite que uma representação de dados seja manipulada de maneiras que não foram pretendidas, com resultados potencialmente desastrosos.

Por exemplo, o uso de um inteiro como um ponteiro pode causar modificações arbitrárias em programas e dados. Para evitar violações de tipo, geralmente impomos uma estrutura de tipo estática aos programas. Em linguagens como Pascal e Ada, o tipo de variáveis e símbolos de função é definido por declarações redundantes, e o compilador pode verificar a consistência de definição e uso. Em linguagens como ML, declarações explícitas são evitadas sempre que possível, e o sistema pode inferir o tipo de expressões do contexto local, enquanto ainda estabelece um uso consistente.

Linguagens de programação nas quais o tipo de cada expressão pode ser determinado por análise estática de programa são ditas estaticamente tipadas. A tipagem estática é uma propriedade útil, mas o requisito de que todas as variáveis e expressões sejam vinculadas a um tipo em tempo de compilação às vezes é muito restritivo. As linguagens em que todas as expressões são consistentes em tipo são chamadas de linguagens fortemente tipadas. Se uma linguagem for fortemente tipada, seu compilador pode garantir que os programas que ela aceita serão executados sem erros de tipo.

Em geral, devemos nos esforçar para uma tipagem forte e adotar a tipagem estática sempre que possível. A tipagem estática permite que inconsistências de tipo sejam descobertas em tempo de compilação e garante que os programas executados sejam consistentes com o tipo. Os sistemas tradicionais estaticamente tipados excluem técnicas de programação que, embora sólidas, são incompatíveis com a vinculação inicial de objetos de programa a um tipo específico.

Tipos de polimorfismo

As linguagens tipicamente convencionais baseiam-se na ideia de que funções e procedimentos, e portanto seus operandos, têm um tipo único. Essas linguagens são ditas monomórficas, no sentido de que cada valor é variável pode ser interpretado como sendo de um e apenas um tipo. As linguagens de programação monomórficas podem ser contrastadas com as linguagens polimórficas nas quais alguns valores e variáveis podem ter mais de um tipo. Funções polimórficas são funções cujos operandos podem ter mais de um tipo.

Os tipos polimórficos podem ser definidos como tipos cujas operações são aplicáveis a operandos de mais de um tipo. Strachey fez uma distinção, informalmente, entre dois tipos principais de polimorfismo. No caso do polimorfismo

universal, pode-se afirmar com confiança que alguns valores têm muitos tipos, enquanto no polimorfismo ad-hoc isso é mais difícil de manter, pois pode-se assumir a posição de que uma função polimórfica ad-hoc é realmente um pequeno conjunto de funções monomórficas. Em termos de implementação, uma função polimórfica universal executará o mesmo código para argumentos de qualquer tipo admissível, enquanto uma função polimórfica ad-hoc pode executar um código diferente para cada tipo de argumento.

Existem dois tipos principais de polimorfismo universal, ou seja, duas maneiras principais pelas quais um valor pode ter muitos tipos.

Aqui era bom fazer tipo uma lista:

- polimorfismo paramétrico
- sobrecarregando
- ad-hoc

Sem lista, e pra ficar geral mesmo pra não abordar ponto e o texto ficar maior

No Polimorfismo Paramétrico, uma função polimórfica possui um parâmetro de tipo implícito ou explícito que determina o tipo do argumento para cada aplicação daquela função. As funções que exibem polimorfismo paramétrico também são chamadas de funções genéricas. Por exemplo, a função de comprimento de listas de tipos arbitrários para inteiros é chamada de função de comprimento genérica.

Uma função genérica é aquela que pode funcionar para argumentos de muitos tipos, geralmente fazendo o mesmo tipo de trabalho independentemente do tipo de argumento. Se considerarmos uma função genérica como um valor único, ela tem muitos tipos funcionais e, portanto, é polimórfica. Existem também dois tipos principais de polimorfismo ad-hoc.
Na sobrecarga, o mesmo nome de variável é usado para denotar funções diferentes e o contexto é usado para decidir qual função é denotada por uma instância particular do nome. As coerções podem ser fornecidas estaticamente, inserindo-as automaticamente entre argumentos e funções em tempo de compilação, ou podem ter que ser determinadas dinamicamente por testes de tempo de execução nos argumentos. Isso é particularmente verdadeiro quando se considera linguagens não digitadas e linguagens interpretadas. Mas mesmo em linguagens compiladas estáticas, pode haver confusão entre as duas formas de polimorfismo ad-hoc. Nossa definição de polimorfismo é aplicável apenas a linguagens com uma noção muito clara de tipo e valor.
Se vemos um tipo como especificando parcialmente o comportamento, ou uso pretendido, de valores associados, então os sistemas de tipo monomórfico restringem os objetos a ter apenas um comportamento, enquanto os sistemas de tipo polimórfico permitem que os valores sejam associados a mais de um comportamento. Linguagens estritamente monomórficas são muito restritivas em seu poder expressivo porque não permitem que valores, ou mesmo símbolos sintáticos que denotam valores, exibam comportamentos diferentes em contextos de uso diferentes. Línguas como Pascal e Ada têm maneiras de relaxar o monomorfismo estrito, mas o polimorfismo é a exceção e não a regra e podemos dizer que eles são principalmente monomórficos.

(1) Sobrecarga: constantes inteiras podem ser do tipo inteiro e real. Operadores como + são aplicáveis a argumentos inteiros e reais.

(2) Coerção: Um valor inteiro pode ser usado onde um real é esperado e vice-versa.
(3) Subtipagem: Elementos de um tipo de subfaixa também pertencem a tipos de super-faixa.
(4) Compartilhamento de valor: nil em Pascal é uma constante que é compartilhada por todos os tipos de ponteiros.

Esses quatro exemplos, que podem ser encontrados na mesma linguagem, são exemplos de quatro maneiras radicalmente diferentes de estender um sistema de tipo monomórfico. Vamos ver como eles se encaixam na descrição anterior de diferentes tipos de polimorfismo. A coerção permite que o usuário omita as conversões de tipo semanticamente necessárias. As conversões de tipo necessárias devem ser determinadas pelo sistema, inseridas no programa e usadas pelo compilador para gerar o código de conversão de tipo necessário.

As coerções são essencialmente uma forma de abreviatura que pode reduzir o tamanho do programa e melhorar a legibilidade do programa, mas também pode causar erros de sistema sutis e às vezes perigosos. A necessidade de coerções em tempo de execução geralmente é detectada em tempo de compilação, mas linguagens como LISP têm muitas coerções que só são detectadas e executadas em tempo de execução <- RODAPÉ (eu coloco no latex). Subtipagem é uma instância de polimorfismo de inclusão. A ideia de um tipo ser um subtipo de outro tipo é útil não apenas para subfaixas de tipos ordenados, como inteiros, mas também para estruturas mais complexas, como um tipo que representa Toyotas, que é um subtipo de um tipo mais geral, como Veículos.

Como está implícito na escolha dos nomes, o polimorfismo universal é considerado polimorfismo verdadeiro, enquanto o polimorfismo ad-hoc é algum tipo de polimorfismo aparente cujo caráter polimórfico desaparece de perto. Na implementação, as representações são escolhidas com muito cuidado, de forma que nenhuma coerção seja necessária ao usar um objeto de um subtipo no lugar de um objeto do supertipo. Da mesma forma, as operações têm o cuidado de interpretar as representações uniformemente para que possam funcionar uniformemente em elementos de subtipos e super-tipos.
O uso do mesmo símbolo para essas três operações reflete uma semelhança aproximada da estrutura algébrica, mas viola os requisitos do monomorfismo. A ambigüidade geralmente pode ser resolvida pelo tipo de operandos imediatos de um operador sobrecarregado, mas isso pode não ser suficiente. O ALGOL 68 é bem conhecido por seu esquema de coerção barroco. Os problemas a serem resolvidos aqui são muito semelhantes à sobrecarga, mas, além disso, as correções têm efeitos de tempo de execução.

Finalmente, devemos mencionar procedimentos genéricos do tipo encontrado em Ada, que são modelos parametrizados que devem ser instanciados com valores de parâmetros reais antes de serem usados. O polimorfismo dos procedimentos genéricos de Ada é semelhante ao polimorfismo paramétrico de linguagens como ML, mas é especializado em tipos específicos de parâmetros. Os parâmetros podem ser parâmetros de tipo, parâmetros de procedimento ou parâmetros de valor. Os procedimentos genéricos com parâmetros de tipo são polimórficos no sentido de que os parâmetros de tipo formal podem assumir diferentes tipos reais para diferentes instâncias.

O polimorfismo de tipo genérico em Ada, no entanto, é sintático, uma vez que a instanciação genérica é realizada em tempo de compilação com valores de tipo reais que devem ser determináveis em tempo de compilação. A semântica dos procedimentos genéricos é a expansão da macro impulsionada pelo tipo dos argumentos. Assim, procedimentos genéricos podem ser considerados abreviações para conjuntos de procedimentos monomórficos.

A evolução dos tipos nas linguagens de programação

Nas primeiras linguagens de programação, a computação era identificada com a computação numérica e os valores podiam ser vistos como tendo um único tipo aritmético. Já em 1954, no entanto, em FORTRAN foi considerado conveniente distinguir entre números inteiros e números de ponto flutuante, em parte porque as diferenças na representação de hardware tornavam a computação de inteiros mais econômica e em parte porque o uso de inteiros para iteração e computação de matriz era logicamente diferente do uso de números de ponto flutuante para computação numérica. FORTRAN distingue entre variáveis inteiras e de ponto flutuante pela primeira letra de seus nomes. O ALGOL 60 tornou esta distinção explícita introduzindo declarações de identificador redundantes para variáveis reais e booleanas inteiras.

ALGOL 60 foi a primeira linguagem significativa a ter uma noção explícita de tipo e requisitos associados para verificação de tipo em tempo de compilação. Seus requisitos de estrutura de bloco permitiam não apenas o tipo, mas também o escopo das variáveis a serem verificadas em tempo de compilação. A noção ALGOL 60 de tipo foi estendida a classes de valores mais ricas na década de 1960. Das inúmeras linguagens tipadas desenvolvidas durante este período, PL / I, Pascal, ALGOL 68 e Simula, são notáveis por suas contribuições para a evolução do conceito de tipo.

PL / I tenta combinar os recursos de FORTRAN, ALGOL 60, COBOL e LISP. Pascal fornece uma extensão mais limpa de tipos para registros de array e ponteiros, bem como tipos definidos pelo usuário. No entanto, Pascal não define equivalência de tipo, de forma que a questão de quando duas expressões de tipo denotam o mesmo tipo depende da implementação. Pascal deixa lacunas na especificação de tipo forte ao não exigir que o tipo completo de procedimentos passados como parâmetros seja especificado e ao permitir que o campo de tag de registros variantes seja manipulado de forma independente.

As ambiguidades e inseguranças do sistema do tipo Pascal são discutidas em Welsh et al. ALGOL 68 tem uma noção de tipo mais rigorosa do que Pascal, com uma noção bem definida de equivalência de tipo. A noção de tipo é estendida para incluir procedimentos como valores de primeira classe. ALGOL 68 definiu cuidadosamente regras para coerção, usando desreferenciamento, de procedimento, alargamento, remo, união e anulação para transformar valores para o tipo necessário para computação posterior.

Sua noção de tipo inclui classes cujas instâncias podem ser atribuídas como valores de variáveis com valor de classe e podem persistir entre a execução dos procedimentos que contêm. ML introduziu a noção de polimorfismo paramétrico em

linguagens. Os tipos de ML podem conter variáveis de tipo instanciadas para diferentes tipos em diferentes contextos.

Consideramos as abstrações de dados não digitados de Ada, indicamos o impacto na metodologia de exigir que as abstrações de dados tenham tipo e herança, discutimos a interpretação da herança como polimorfismo de subtipo e examinamos a relação entre o polimorfismo de subtipo de Smalltalk e o polimorfismo paramétrico de ML. Ada tem uma rica variedade de módulos, incluindo subprogramas para suportar programação orientada a procedimentos, pacotes para suportar abstrações de dados e tarefas para suportar programação simultânea. Eles podem ocultar um estado local por um tipo de dados privado ou no corpo do pacote. Pacotes em Ada são objetos de segunda classe, enquanto instâncias de classe em Simula ou objetos em linguagens orientadas a objetos são objetos de primeira classe.

As diferenças de comportamento entre pacotes e registros em Ada são evitadas em linguagens orientadas a objetos, estendendo a noção de tipo para procedimentos e abstrações de dados. No contexto desta discussão, é útil definir as linguagens orientadas a objetos como extensões das linguagens orientadas a procedimentos que suportam abstrações de dados digitados com herança.

- Ele suporta objetos que são abstrações de dados com uma interface de operações nomeadas e um estado local oculto.
- Os objetos têm um tipo de objeto associado.
- Os tipos podem herdar atributos de super-tipos.

Esses requisitos podem ser resumidos como;

orientado a objetos = abstrações de dados + tipos de objetos + herança de tipo.

A utilidade desta definição pode ser ilustrada considerando o impacto de cada um desses requisitos na metodologia. A abstração de dados por si só fornece uma maneira de organizar dados com operações associadas que difere consideravelmente da metodologia tradicional de programação orientada a procedimentos. A realização da metodologia de abstração de dados foi um dos objetivos primários da Ada, e essa metodologia é descrita detalhadamente na literatura da Ada em publicações como Booth [1983]. No entanto, Ada satisfaz apenas o primeiro de nossos três requisitos para a programação orientada a objetos, e é interessante examinar o impacto dos tipos de objeto e da herança na metodologia de abstração de dados [Hendler e Wegner 19861.

O requisito de que todos os objetos tenham um tipo permite que os objetos sejam valores de primeira classe para que possam ser gerenciados como estruturas de dados dentro da linguagem, bem como usados para computação. O requisito de herança de tipo permite que as relações entre os tipos sejam especificadas. A herança pode ser vista como um mecanismo de composição de tipo que permite que as propriedades de um ou mais tipos sejam reutilizadas na definição de um novo tipo. Isso é ilustrado pela hierarquia de objetos Smalltalk em Goldberg e Robson [1983].

A hierarquia de objetos Smalltalk é uma descrição do ambiente de programação Smalltalk em Smalltalk. Ele descreve uma coleção de mais de 75 tipos de objetos de sistema relacionados por uma hierarquia de herança. Ao fazer isso, a coleção de mais de 75 tipos de objetos que compõem o ambiente Smalltalk é descrita como uma hierarquia estruturada relativamente simples de tipos de objetos. A abreviatura fornecida pela hierarquia de objetos na reutilização de superclasses cujos atributos são compartilhados por subclasses é claramente incidental para a parcimônia conceitual alcançada pela imposição de uma estrutura coerente na coleção de tipos de objetos.

A hierarquia de objetos Smalltalk também é significativa como uma ilustração do poder do polimorfismo. Podemos caracterizar uma função polimórfica como uma função aplicável a valores de mais de um tipo e o polimorfismo de inclusão como uma relação entre tipos que permite que operações sejam aplicadas a objetos de diferentes tipos relacionados por inclusão.
Quando dizemos que uma função paramétrica é aplicável a listas de tipo arbitrário, realmente queremos dizer que ela pode ser especializada, fornecendo um parâmetro de tipo T, e que a função especializada pode então ser aplicada aos operandos especializados. Essa distinção entre uma função paramétrica e suas versões especializadas é borrada em linguagens como ML porque os parâmetros de tipo omitidos pelo usuário são reintroduzidos automaticamente pelo mecanismo de inferência de tipo. Supertipos em linguagens orientadas a objetos podem ser vistos como tipos paramétricos cujo parâmetro é omitido pelo usuário. Veremos abaixo que Fun tem parâmetros de tipo explícitos para tipos e supertipos paramétricos, a fim de fornecer um modelo uniforme para polimorfismo paramétrico e de sub-tipo, expressão de tipo geralmente incluem tipos básicos como inteiro e booleano e tipos compostos como matrizes, registros e procedimentos construídos a partir de tipos básicos:

Type :: = BasicType | ConstructedType
BasicType :: = Int | Bool | ...
ConstructedType :: = Array (Type) | Tipo -> Tipo I…

A expressão de tipo sub linguagem deve ser suficientemente rica para suportar tipos para todos os valores com os quais desejamos calcular, mas suficientemente tratável para permitir a verificação de tipo decidível e eficiente. Um dos objetivos deste artigo é examinar os trade-offs entre riqueza e tratabilidade para sublinguagens de expressão de tipo de linguagens fortemente tipadas. A expressão de tipo sublinguagem geralmente pode ser especificada por uma gramática livre de contexto. No entanto, estamos interessados não apenas na sintaxe da expressão de tipo sub linguagem, mas também em sua semântica.

Isto é, estamos interessados em quais tipos denotam e nas relações entre as expressões de tipo. Relações de similaridade entre expressões de tipo que permitem que uma expressão de tipo denote mais de um tipo, ou seja compatível com muitos tipos, são chamadas de polimorfismo. A capacidade de expressar relações entre tipos envolve alguma capacidade de realizar cálculos em tipos para determinar se eles satisfazem a relação desejada. O leitor interessado em uma

discussão sobre linguagens de expressão de tipo e algoritmos de verificação de tipo para linguagens como Pascal e C deve consultar o Capítulo 6 de Aho et al.

Fun

Fun é uma linguagem baseada em cálculo-λ que enriquece a " first-order typed" cálculo-λ com recursos de segunda ordem projetados para modelar polimorfismo e linguagens orientadas a objetos.

```
Type ::= ... | QuantifiedType
QuantifiedType ::=
    ∀A. Type |                          Universal Quantification
    ∃A. Type |                          Existential Quantification
    ∀A⊆Type. Type | ∃A⊆Type. Type       Bounded Quantification
```

A quantificação universal enriquece "first-order typed" cálculo-λ com tipos parametrizados que podem ser especializados ao substituir parâmetros de tipo reais por parâmetros quantificados universalmente. Tipos universalmente quantificados são eles próprios tipos de primeira classe e podem ser parâmetros reais em tal substituição. A quantificação existencial enriquece os recursos de primeira ordem, permitindo tipos de dados abstratos com representação oculta.

Fun apóia a ocultação de informações não apenas por meio da quantificação existencial, mas também por meio de sua construção let, que facilita a ocultação de variáveis locais do corpo de um módulo. A ocultação por meio de let é referida como ocultação de primeira ordem porque envolve a ocultação de identificadores locais e valores associados, enquanto a ocultação por meio de quantificadores existenciais é referida como ocultação de segunda ordem porque envolve a ocultação de representações de tipo. A quantificação limitada enriquece o "first-order typed", fornecendo parâmetros de subtipo explícitos. A quantificação limitada fornece um mecanismo explicativo para o polimorfismo orientado a objetos que é complicado de usar explicitamente, mas útil para iluminar a relação entre o polimorfismo paramétrico e o herdado.

A evolução da untyped a universos digitados podem ser ilustradas pela λ-calculus, desenvolvida inicialmente como uma notação untyped para capturar a essência da aplicação funcional dos operadores e operandos.

```
e ::= x          -- a variable is a λ-expression
e ::= fun(x)e    -- functional abstraction of e
e ::= e(e)       -- operator e applied to operand e
```

A função de identidade e a função sucessora podem ser especificadas no λcálculocomo segue.

```
value id = fun(x) x          -- identity function
value succ = fun(x) x+1      -- successor function (for integers)
```

A função de identidade pode ser aplicada a uma arbitrária λexpressãoe sempre produz a própria expressão X. A fim de definir a adição em inteiros no puro λcálculo, escolhemos uma representação para inteiros e definimos a operação de adição de modo que seu efeito nas λ que expressões representam os inteiros n e m produza a λexpressãoque representa n + m . A função sucessora deve ser aplicada apenas a λ-expressões que representam inteiros e sugerem uma noção de tipagem.

valor duas vezes = fun (f) fun (y) f (f (y)) - função duas vezes

A aplicação de duas vezes à função sucessora produz uma λexpressãoque calcula o sucessor do sucessor:

duas vezes (succ) => fun (y) succ (succ (y))
duas vezes (fun (x) x + 1) => fun (y) (fun (x) x + 1) ((fun (x) x + 1) (y))

O a discussão acima ilustra como os tipos surgem quando especializamos uma notação não, como o cálculo-λ tipado, para realizar tipos particulares de computação, como a aritmética de inteiros.

A forma de escrita é como o λ-calculus, exceto que todas as variáveis devem ser explicitamente digitado quando introduzido como uma variável ligada. Assim, a função sucessora no cálculo digitado λ-tem a seguinte forma:

valor succ = fun (x: Int) x + 1

A função duas vezes de inteiros para inteiros tem um parâmetro f cujo tipo é Int => Int (o tipo de funções de inteiros para inteiros) e pode ser escrita da seguinte forma:

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

Esta notação se aproxima da especificação funcional em linguagens de programação digitadas, mas omite a especificação do tipo de resultado. Podemos denotar o tipo de resultado com uma palavra-chave de retorno da seguinte maneira:

valor succ = fun (x: Int) (retorna Int) x + 1

No entanto, o tipo de resultado pode ser determinado a partir da forma do corpo da função x + 1. Os mecanismos de inferência de tipo que permitem que essas informações sejam recuperadas durante a compilação são discutidos em uma seção posterior. Ao longo deste documento, os nomes dos tipos começam com letras maiúsculas, enquanto os nomes dos valores e funções começam com letras minúsculas:

símbolo igual na vdd é outra coisa, confere lá no documento

tipo IntPair = Intx Int
tipoIntFun = Int + Int

Declarações de tipo introduzem nomes para expressões de tipo; eles não criam novos tipos em nenhum sentido. Isso às vezes é expresso pelo uso de equivalência estrutural em tipos em vez de nome equivalentes: dois tipos são equivalentes se tiverem a mesma estrutura, independentemente dos nomes que usamos como abreviações.

(3,4): IntPair
succ: IntFun

Não precisamos introduzir variáveis por declarações de tipo da forma var: T porque o tipo de uma variável pode ser determinado a partir da forma do valor atribuído.

valor intPair = (3,4)

No entanto, se quisermos indicar o tipo de uma variável como parte de sua inicialização, podemos fazê-lo pela notação valor var: T = valor:

```
value intPair: IntPair = (3,4)
value succ: Int → Int = fun(x: Int) x + 1
```

Variáveis locais podem ser declaradas pelo let-in construção, que introduz uma nova variável inicializada em um escopo local.

deixe a = 3 em a + 1 resulta em 4

Se quisermos especificar tipos, também podemos escrever

let a: Int = 3 em a + 1

A construção let-in pode ser definida em termos de expressões divertidas básicas:

deixe a: T = M em NI (fun (a: T) N) (M)

Tipos básicos, tipos estruturados e recursão

O cálculo-λ tipado é geralmente aumentado com vários tipos de tipos básicos e estruturados. Para tipos básicos, devemos usar.

Unit	the trivial type, with only element ()
Bool	with an if-then-else operation
Int	with arithmetic and comparison operations
Real	with arithmetic and comparison operations
String	with string concatenation (infix) ^

Tipos estruturados podem ser construídos a partir desses tipos básicos por meio de construtores de tipo. Os construtores de tipo em nossa linguagem incluem espaços de função, produtos cartesianos, tipos de registro e tipos de variantes.
valor p = 3, verdadeiro: Int x Bool As

operações em pares são seletores para o primeiro e o segundo componentes:

fst (p) resulta em 3

snd (p) resulta em verdadeiro

Um registro é um conjunto não ordenado de valores rotulados. Seu tipo pode ser especificado indicando o tipo associado a cada um de seus rótulos

tipo ARecordType = (a: Int, b: Bool, c: String}

Um registro deste tipo pode ser criado inicializando cada um dos rótulos de registro com um valor de o tipo necessário.

valor r: ARecordType = (a = 3, b = true, c = “abed”)

Os rótulos devem ser exclusivos em qualquer registro ou tipo de registro.

rb produz true

Como as funções são valores de primeira classe, os registros podem, em geral, ter componentes de função:

```
type FunctionRecordType = {f1: Int → Int, f2: Real → Real}
value functionRecord = {f1 = succ, f2 = sin}
```

Um tipo de registro pode ser definido em termos de tipos de registro existentes por um operador &, que concatena dois tipos de registro:

tipo NewFunctionRecordType = FunctionRecordType & (f3: BooI => B00l)

Pretende-se que seja uma abreviatura, em vez de escrever os três campos f 1, f 2 e f3 explicitamente. Uma estrutura de dados pode se tornar local e privada para uma coleção de funções por meio de declarações let-in.

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

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

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

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

Um elemento desse tipo pode ser um inteiro rotulado a, um Booleano rotulado b ou uma string rotulada C:

```
value v1 = [a = 3]
value v2 = [b = true]
value v3 = [c = "abcd"]
```

A única operação nas variantes é a seleção de caso. Uma declaração de caso para uma variante do tipo AVariantType tem a seguinte forma:

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

onde em cada caso uma nova variável é introduzida e ligada ao respectivo conteúdo da variante. Essa variável pode então ser usada na respectiva ação.

Aqui está uma função que, dado um elemento do tipo AVariantType acima, retorna uma string:

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

onde o conteúdo do objeto variante x é vinculado aos identificadores anInt, aBool ou aString, dependendo do caso. No entanto, todos os cálculos expressos no cálculo digitado λdevem terminar. A função fatorial pode ser expressa como

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

Para simplificar, assumimos que os únicos valores que podem ser recursivamente definidos são funções. Finalmente, apresentamos definições de tipo recursivo. Isso nos permite, por exemplo, definir o tipo de listas de inteiros fora do registro e tipos de variantes:

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

Uma lista de inteiros é nula (representada como [nil = ()]) ou os contras de um inteiro e uma lista de inteiros (representada como , por exemplo, [cons = {cabeça = 3, cauda = nulo)]).