Sintaxe Concreta de L1

Sumário

[Introdução 2](file:///C:\Users\Arthur\Dropbox\Documentos\UFRGS\Disciplinas\2016-2%20-%20Semântica%20Formal%20N\Workspace%20F%23\Interpretador%20L1\Interpretador%20L1\docs\Sintaxe.docx#_Toc466635311)

[Sintaxe Corrente 3](file:///C:\Users\Arthur\Dropbox\Documentos\UFRGS\Disciplinas\2016-2%20-%20Semântica%20Formal%20N\Workspace%20F%23\Interpretador%20L1\Interpretador%20L1\docs\Sintaxe.docx#_Toc466635312)

[Termos 3](file:///C:\Users\Arthur\Dropbox\Documentos\UFRGS\Disciplinas\2016-2%20-%20Semântica%20Formal%20N\Workspace%20F%23\Interpretador%20L1\Interpretador%20L1\docs\Sintaxe.docx#_Toc466635313)

[Termos Básicos 3](file:///C:\Users\Arthur\Dropbox\Documentos\UFRGS\Disciplinas\2016-2%20-%20Semântica%20Formal%20N\Workspace%20F%23\Interpretador%20L1\Interpretador%20L1\docs\Sintaxe.docx#_Toc466635314)

[Termos Adicionais 4](file:///C:\Users\Arthur\Dropbox\Documentos\UFRGS\Disciplinas\2016-2%20-%20Semântica%20Formal%20N\Workspace%20F%23\Interpretador%20L1\Interpretador%20L1\docs\Sintaxe.docx#_Toc466635315)

[Tipos 4](file:///C:\Users\Arthur\Dropbox\Documentos\UFRGS\Disciplinas\2016-2%20-%20Semântica%20Formal%20N\Workspace%20F%23\Interpretador%20L1\Interpretador%20L1\docs\Sintaxe.docx#_Toc466635316)

[Tipos Básicos 4](file:///C:\Users\Arthur\Dropbox\Documentos\UFRGS\Disciplinas\2016-2%20-%20Semântica%20Formal%20N\Workspace%20F%23\Interpretador%20L1\Interpretador%20L1\docs\Sintaxe.docx#_Toc466635317)

[Tipos Adicionais 4](file:///C:\Users\Arthur\Dropbox\Documentos\UFRGS\Disciplinas\2016-2%20-%20Semântica%20Formal%20N\Workspace%20F%23\Interpretador%20L1\Interpretador%20L1\docs\Sintaxe.docx#_Toc466635318)

[Traits 4](file:///C:\Users\Arthur\Dropbox\Documentos\UFRGS\Disciplinas\2016-2%20-%20Semântica%20Formal%20N\Workspace%20F%23\Interpretador%20L1\Interpretador%20L1\docs\Sintaxe.docx#_Toc466635319)

[Operadores 5](file:///C:\Users\Arthur\Dropbox\Documentos\UFRGS\Disciplinas\2016-2%20-%20Semântica%20Formal%20N\Workspace%20F%23\Interpretador%20L1\Interpretador%20L1\docs\Sintaxe.docx#_Toc466635320)

[Operadores Básicos 5](file:///C:\Users\Arthur\Dropbox\Documentos\UFRGS\Disciplinas\2016-2%20-%20Semântica%20Formal%20N\Workspace%20F%23\Interpretador%20L1\Interpretador%20L1\docs\Sintaxe.docx#_Toc466635321)

[Operadores Adicionais 6](file:///C:\Users\Arthur\Dropbox\Documentos\UFRGS\Disciplinas\2016-2%20-%20Semântica%20Formal%20N\Workspace%20F%23\Interpretador%20L1\Interpretador%20L1\docs\Sintaxe.docx#_Toc466635322)

[Prioridade e Associatividade 6](file:///C:\Users\Arthur\Dropbox\Documentos\UFRGS\Disciplinas\2016-2%20-%20Semântica%20Formal%20N\Workspace%20F%23\Interpretador%20L1\Interpretador%20L1\docs\Sintaxe.docx#_Toc466635323)

[Comentários 7](file:///C:\Users\Arthur\Dropbox\Documentos\UFRGS\Disciplinas\2016-2%20-%20Semântica%20Formal%20N\Workspace%20F%23\Interpretador%20L1\Interpretador%20L1\docs\Sintaxe.docx#_Toc466635324)

[Identificadores 7](file:///C:\Users\Arthur\Dropbox\Documentos\UFRGS\Disciplinas\2016-2%20-%20Semântica%20Formal%20N\Workspace%20F%23\Interpretador%20L1\Interpretador%20L1\docs\Sintaxe.docx#_Toc466635325)

[Indentação e Espaçamento 8](file:///C:\Users\Arthur\Dropbox\Documentos\UFRGS\Disciplinas\2016-2%20-%20Semântica%20Formal%20N\Workspace%20F%23\Interpretador%20L1\Interpretador%20L1\docs\Sintaxe.docx#_Toc466635326)

[Caracteres e Strings 8](file:///C:\Users\Arthur\Dropbox\Documentos\UFRGS\Disciplinas\2016-2%20-%20Semântica%20Formal%20N\Workspace%20F%23\Interpretador%20L1\Interpretador%20L1\docs\Sintaxe.docx#_Toc466635327)

[Input e Output 8](file:///C:\Users\Arthur\Dropbox\Documentos\UFRGS\Disciplinas\2016-2%20-%20Semântica%20Formal%20N\Workspace%20F%23\Interpretador%20L1\Interpretador%20L1\docs\Sintaxe.docx#_Toc466635328)

[Açúcar Sintático 8](file:///C:\Users\Arthur\Dropbox\Documentos\UFRGS\Disciplinas\2016-2%20-%20Semântica%20Formal%20N\Workspace%20F%23\Interpretador%20L1\Interpretador%20L1\docs\Sintaxe.docx#_Toc466635329)

[Extensões Práticas 9](file:///C:\Users\Arthur\Dropbox\Documentos\UFRGS\Disciplinas\2016-2%20-%20Semântica%20Formal%20N\Workspace%20F%23\Interpretador%20L1\Interpretador%20L1\docs\Sintaxe.docx#_Toc466635330)

[Comentários 9](file:///C:\Users\Arthur\Dropbox\Documentos\UFRGS\Disciplinas\2016-2%20-%20Semântica%20Formal%20N\Workspace%20F%23\Interpretador%20L1\Interpretador%20L1\docs\Sintaxe.docx#_Toc466635331)

[Bibliotecas 10](file:///C:\Users\Arthur\Dropbox\Documentos\UFRGS\Disciplinas\2016-2%20-%20Semântica%20Formal%20N\Workspace%20F%23\Interpretador%20L1\Interpretador%20L1\docs\Sintaxe.docx#_Toc466635332)

[stdlib 10](file:///C:\Users\Arthur\Dropbox\Documentos\UFRGS\Disciplinas\2016-2%20-%20Semântica%20Formal%20N\Workspace%20F%23\Interpretador%20L1\Interpretador%20L1\docs\Sintaxe.docx#_Toc466635333)

[Extensões de L1 15](file:///C:\Users\Arthur\Dropbox\Documentos\UFRGS\Disciplinas\2016-2%20-%20Semântica%20Formal%20N\Workspace%20F%23\Interpretador%20L1\Interpretador%20L1\docs\Sintaxe.docx#_Toc466635334)

[Função Recursiva 15](file:///C:\Users\Arthur\Dropbox\Documentos\UFRGS\Disciplinas\2016-2%20-%20Semântica%20Formal%20N\Workspace%20F%23\Interpretador%20L1\Interpretador%20L1\docs\Sintaxe.docx#_Toc466635335)

[Igualdade Genérica 16](file:///C:\Users\Arthur\Dropbox\Documentos\UFRGS\Disciplinas\2016-2%20-%20Semântica%20Formal%20N\Workspace%20F%23\Interpretador%20L1\Interpretador%20L1\docs\Sintaxe.docx#_Toc466635336)

[Trait Orderable 18](file:///C:\Users\Arthur\Dropbox\Documentos\UFRGS\Disciplinas\2016-2%20-%20Semântica%20Formal%20N\Workspace%20F%23\Interpretador%20L1\Interpretador%20L1\docs\Sintaxe.docx#_Toc466635337)

[Operações Booleanas Curto-Circuito 19](file:///C:\Users\Arthur\Dropbox\Documentos\UFRGS\Disciplinas\2016-2%20-%20Semântica%20Formal%20N\Workspace%20F%23\Interpretador%20L1\Interpretador%20L1\docs\Sintaxe.docx#_Toc466635338)

[Input e Output 20](file:///C:\Users\Arthur\Dropbox\Documentos\UFRGS\Disciplinas\2016-2%20-%20Semântica%20Formal%20N\Workspace%20F%23\Interpretador%20L1\Interpretador%20L1\docs\Sintaxe.docx#_Toc466635339)

[Conversão de e para Strings 22](file:///C:\Users\Arthur\Dropbox\Documentos\UFRGS\Disciplinas\2016-2%20-%20Semântica%20Formal%20N\Workspace%20F%23\Interpretador%20L1\Interpretador%20L1\docs\Sintaxe.docx#_Toc466635340)

# Introdução

Este documento tem como objetivo especificar a sintaxe concreta feita da linguagem L1. É presumido conhecimento da gramática, semântica e sistema de tipos de L1, focando apenas nos detalhes específicos a essa implementação da linguagem.

Originalmente, a implementação possuía apenas as construções básicas de L1, porém foram sendo adicionados tanto açúcares sintáticos quanto extensões práticas, que não modificam a definição da linguagem (como comentários, bibliotecas, etc). Mais tarde, a própria linguagem L1 foi expandida, adicionando novos termos e tipos.

A seção *Sintaxe Corrente* mostra todas as construções básicas presentes na linguagem, separando cada tipo de construção em básico (ou seja, presente na definição padrão de L1) e adicional (ou seja, criada nessa implementação específica de L1). Além disso, ela possui informações sobre restrições impostas para cada construção (identificadores, por exemplo).

Embora a seção *Sintaxe Corrente* mostre todas as construções gramaticais da linguagem, é necessário ler a seção *Extensões Práticas* para saber a sintaxe completa dessa implementação de L1. Essas construções não adicionam nenhuma funcionalidade à linguagem, mas apenas facilitam o trabalho de programação com comentários, reuso de código, etc. A única exceção é a adição de argumentos de linha de comando, permitindo alterar o resultado de um programa sem mexer em seu código fonte. Porém, como essa é uma construção externa ao ambiente de codificação, decidi categorizá-la como uma extensão pratica.

# Sintaxe Corrente

## Termos

### Termos Básicos

|  |  |
| --- | --- |
| Termo | Sintaxe |
|  |  |
|  |  |
| op | op |
| if then else | if then else |
|  |  |
|  |  |
| fn | fn |
| fn | fn |
|  | rec |
|  | rec |
| let in | let |
| let in | let |
|  |  |
|  |  |
| isempty | empty? |
| hd | head |
| tl | tail |
| raise | raise |
| try with | try except |

### Termos Adicionais

|  |  |
| --- | --- |
| Termo | Sintaxe |
|  | ‘’ |
| skip | skip |
| input | input |
| output | output |
|  |  |

## Tipos

### Tipos Básicos

|  |  |
| --- | --- |
| Tipo | Sintaxe |
| int | Int |
| bool | Bool |
|  | -> |
| list |  |

Na especificação de funções, o tipo é associativo à direita. Dessa forma, é o mesmo que . É possível utilizar parênteses para obter a associatividade desejada.

### Tipos Adicionais

|  |  |
| --- | --- |
| Tipo | Sintaxe |
| char | Char |
| string\*\* | String |
| unit | Unit |

\*\* O tipo string é, na realidade, apenas um pseudônimo para o tipo char list.

### Traits

Traits são características que certos tipos podem ter. Essas características servem para criar restrições de tipos em funções e operadores, permitindo-as a serem polimórficas com certas restrições.

#### Equatable

Termos de tipos que possuem a trait *Equatable* podem ser testados igualdade. A regra para a construção de tipos *Equatable* é a seguinte.

Dos tipos básicos, apenas funções não são *Equatable*. Além disso, para uma lista ser *Equatable*, os elementos da mesma precisam ser *Equatable*.

#### Orderable

Essa trait permite que termos sejam comparados por ordem (menor, maior). Para um tipo ser *Orderable*, ele também precisa ser *Equatable*.

## Operadores

### Operadores Básicos

|  |  |  |
| --- | --- | --- |
| Operador | Sintaxe | Tipos Aceitos |
|  |  | Int |
|  |  | Int |
|  |  | Int |
| div |  | Int |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

### Operadores Adicionais

Além dos operadores definidos na gramática de L1, essa implementação possui os seguintes operadores:

|  |  |  |
| --- | --- | --- |
| Operador | Uso | Significado |
| (prefixo) |  | Oposto numérico de \*\* |
|  |  | Resto da divisão inteira de por \*\* |
|  |  | Concatena a lista com a lista \*\* |
|  |  | Operação logica “e” em e |
|  |  | Operação logica “ou” em e |
| ; |  | Operação de sequência |
| $ |  | Aplica em |
| . |  | Composição de funções |
| !! |  | Acessa o “-ésimo” elemento de \*\* |

\*\* Requer a *stdlib* para funcionar

### Prioridade e Associatividade

Abaixo está a prioridade de cada operador, além de sua associatividade. É importante ressaltar que é possível mudar tanto a prioridade quando a associatividade com o uso de parênteses.

|  |  |  |
| --- | --- | --- |
| Prioridade | Operador | Associatividade |
| 0 | (aplicação de função) | Esquerda |
| 1 | . (composição) | Direita |
|  | !! | Esquerda |
| 2 | ,, | Esquerda |
| 3 | , (infixo e prefixo) | Esquerda |
| 4 |  | Direita |
| 5 |  | Esquerda |
| 6 |  | Nenhuma |
| 7 |  | Direita |
| 8 |  | Direita |
| 9 | **;** | Esquerda |
| 10 | $ | Direita |

### 

### Comentários

* Os operadores booleanos (&& e ||) possuem avaliação em curto-circuito. Isso significa que, caso seja possível determinar o resultado da operação com apenas o primeiro valor, o segundo não é avaliado. Isso permite fazer testes para impedir a avaliação de algum termo que levaria a uma exceção (por exemplo, pegar o primeiro elemento de uma lista vazia).

## Identificadores

Identificadores são strings de tamanho arbitrário, sem espaços em brancos, compostos de quaisquer caracteres com exceção dos seguintes:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| . | , | ; | : | \ | $ | % | + |
| - | / | \* | % | > | < | = | ! |
| & | | | @ | ‘ | “ | ( | ) | [ |
| ] | { | } | \n | \t | \r |  |  |

* Numerais não são permitidos no começo de um identificador, porém podem ser utilizados depois.
* A interrogação é permitida em identificadores, e o uso mais indicado para a mesma é no final de funções que retornem um booleano.

Existem termos reservados na linguagem L1, e esses termos não podem ser utilizados como identificadores:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| fn | let | rec | for | in | head | skip |
| tail | empty? | nil | try | except | raise | output |
| true | false | if | then | else | import | input |

## Indentação e Espaçamento

Indentação não é considerada na hora de avaliar termos. Dessa forma, é possível colocar a indentação que o programador desejar.

Existem algumas construções que requerem espaços entre seus termos (, por exemplo), e esses “espaços” podem ser quaisquer um dos seguintes caracteres (além do próprio espaço):

\n, \t, \r

# Caracteres e Strings

Literais de caracteres (*chars*) são delimitados por aspas simples (‘), e literais de strings, por aspas duplas (“). O caractere de barra invertida (\) é usado para escapar certos caracteres especiais. Esses são:

*\b, \n, \r, \t, \\, \”, \’*

Literais de caractere possuem tamanho exatamente igual a 1, enquanto strings não possuem restrições.

# Input e Output

Os termos *input* e *output* permitem o programa se tornar interativo, recebendo entradas do usuário e mostrando saídas durante a sua execução. Ambos lidam apenas com strings (ou seja, listas de caracteres), com *input* retornando uma nova string recebida do usuário e *output* imprimindo para o console a string recebida como parâmetro.

O termo *output* é o primeiro termo da linguagem a ser usado apenas para seus efeitos colaterais e, como consequência, não retorna nenhum valor útil (representado como o termo *skip*). Assim, seu uso geralmente vem acompanhado do operador de sequência “;”, permitindo avaliação sucessiva de termos sem valor (do tipo *Unit*).

# Açúcar Sintático

|  |  |
| --- | --- |
| Construção | Equivalência |
|  |  |
| let | let |
| let | let |
|  | **\*** |
|  | **\*** |
|  | \* |
| fn | fn |
| fn | fn |
|  |  |
|  |  |
| rec | rec |
| rec | rec |
| let rec | let |
| let rec | let |
| let rec | let rec |
| let rec | let rec |
| let | let |
| let | let |
|  |  |

\* Requer a *stdlib*

# Extensões Práticas

## Comentários

São permitidos comentários de linha única. Tudo que vem depois da string é ignorado pelo interpretador até o final da linha.

## Bibliotecas

Foi implementada uma versão simples de bibliotecas externas para permitir o reuso de código e (principalmente) para permitir a criação de um conjunto de funções básicas para o funcionamento da linguagem (a *stdlib*).

Para importar uma biblioteca, é usada a expressão

onde é o caminho do arquivo a ser importado, relativo ao caminho do executável do interpretador. Vale ressaltar que é obrigatório o uso de aspas duplas na especificação do caminho. Por fim, a extensão do arquivo pode ser omitida, porém ela precisa ser .

Em L1, bibliotecas são arquivos normais de código (*nome.l1*) que possuem uma sequência de *let*s e *let rec*s, sendo que o último desses está incompleto (isto é, não possui ). Ao importar a biblioteca, o interpretador irá substituir a expressão pelo conteúdo da biblioteca, adicionando assim seus identificadores ao escopo atual.

### stdlib

Como em toda linguagem real, existem certas funcionalidades que faz mais sentido implementar através de funções do que construções da linguagem. Para isso, existe uma biblioteca de funções, automaticamente incluída em todos os programas, que fornece uma base mais robusta de funcionalidades. Abaixo estão todas as funções da *stdlib*, organizadas pelo tipo de manipulação que fazem.

#### Operações aritméticas

|  |
| --- |
|  |
| Resto da divisão inteira |

|  |
| --- |
|  |
| Negação unária (oposto numérico) |

|  |
| --- |
|  |
| Valor absoluto |

#### Operações lógicas

|  |
| --- |
|  |
| Negação lógica |

|  |
| --- |
|  |
| “ou exclusivo” lógico |

#### Listas

##### Funções básicas

|  |
| --- |
|  |
| Adiciona um elemento no final de uma lista |

|  |
| --- |
|  |
| Concatena duas listas, colocando todos os elementos da segunda após os elementos da primeira |

|  |
| --- |
|  |
| Retorna o último elemento de uma lista não vazia |

|  |
| --- |
|  |
| Retorna todos os elementos de uma lista não vazia exceto o último |

|  |
| --- |
|  |
| Retorna o número de elementos de uma lista |

##### Geração de listas

|  |
| --- |
|  |
| retorna uma lista de inteiros da forma  onde caso e caso |

##### Transformações

|  |
| --- |
|  |
| Inverte a ordem dos elementos de uma lista |

|  |
| --- |
|  |
| Retorna uma lista obtida ao aplicar a função a cada argumento da lista. Ou seja: |

##### Reduções

|  |
| --- |
|  |
| Aplica uma função a um *acumulador* e cada elemento de uma lista em ordem, retornando sempre o novo valor do *acumulador* |

|  |
| --- |
| **reduce** |
| Mesma funcionalidade de *fold*, porém utilizando o primeiro elemento da lista como *acumulador.* Por causa disso, requer uma lista não vazia. |

|  |
| --- |
|  |
| termina se qualquer elemento de uma lista satisfaz um predicado |

|  |
| --- |
|  |
| Determina se todos os elementos de uma lista satisfazem um predicado |

|  |
| --- |
|  |
| O maior elemento de uma lista não vazia |

|  |
| --- |
|  |
| O menor elemento de uma lista não vazia |

##### Sublistas

|  |
| --- |
|  |
| retorna os primeiros elementos de , ou a lista inteira caso |

|  |
| --- |
|  |
| retorna o final de após pular elementos, ou uma lista vazia caso |

|  |
| --- |
|  |
| Retorna todos os elementos de uma lista até o primeiro que não satisfaz o predicado |

|  |
| --- |
|  |
| Retorna uma lista, pulando todos os elementos até o primeiro que não satisfaz o predicado. |

|  |
| --- |
|  |
| retorna a sublista de que começa no índice e termina no índice . Caso , aciona uma exceção. |

##### Buscas

|  |
| --- |
|  |
| Verifica se um determinado elemento ocorre na lista |

|  |
| --- |
|  |
| Retorna todos os elementos de uma lista que satisfazem um predicado |

##### Indexações

|  |
| --- |
|  |
| Acessa uma lista de forma indexada, começando a contagem em 0 |

|  |
| --- |
|  |
| Retorna o índice da primeira ocorrência de um elemento em uma lista, ou -1 caso ele não ocorra. |

##### Ordenação

|  |
| --- |
|  |
| Ordena os elementos de uma lista em ordem crescente. Usa algoritmo de bubble sort (só que ao contrário). |

##### Conversão de Strings

|  |
| --- |
|  |
| Interpreta a string de entrada como um inteiro. Caso a string não possa ser interpretada, aciona uma exceção. |

|  |
| --- |
|  |
| Transforma um inteiro na sua representação em string. |

|  |
| --- |
|  |
| Interpreta a string de entrada como um booleano. Apenas dois valores são aceitos: “true” e “false”, interpretados como True e False, respectivamente. |

|  |
| --- |
|  |
| Transforma o booleano de entrada em sua representação em string. |

# Extensões de L1

## Função Recursiva

Na definição original de L1, funções recursivas podem ser definidas apenas com declaração de identificadores, sendo essa uma parte não ortogonal da linguagem. Com essa modificação, funções recursivas passarão a ser definidas como funções normais, e existirá apenas uma maneira de declarar identificadores, a expressão *let*.

Dois termos foram adicionados, e dois outros foram removidos

**~~let rec fn in~~**

**~~let rec fn in~~**

As regras de tipos para os novos termos são:

As regras de semântica operacional são as seguintes:

Foram feitas algumas mudanças na sintaxe de L1, também:

|  |  |
| --- | --- |
|  | rec |
|  | rec |

As construções originalmente usadas para os termos *let rec* agora são açúcar sintático, e traduzem para:

|  |  |
| --- | --- |
| let rec | let |
| let rec | let |

## Igualdade Genérica

Inicialmente, L1 permitia apenas a comparação de igualdade (e desigualdade) para inteiros. Com essa extensão, é possível comparar termos de todos os tipos, exceto funções.

Para permitir isso, foi criada a ideia de *Traits* para o sistema de tipos. Para implementar igualdade, foi necessário criar apenas uma trait (*Equatable*), mas o sistema permite expansão para outras traits no futuro (*Iterable, Orderable, etc*).

A nova regra de tipos para a operação de igualdade e desigualdade fica:

Foi criado um novo tipo de constraint para o algoritmo de inferência de tipos (“Tipo possui trait ”). Agora, a função *collectEqs* retorna uma lista de constraints, que podem ser:

A função unify foi modificada para identificar qual o tipo de constraint na cabeça da fila e, caso ela seja uma contraint de Trait, ela vê se a constraint é para uma variável de tipo ou não. Caso positivo, ela chama o método de expansão de constraint, que procura por ocorrências da variável de tipo em outras constraints e gera novas constraints de trait. Caso contrário, ela tenta resolver a constraint, gerando novas constraints de trait caso não seja possível (ou seja, caso o tipo possua variáveis de tipo internas.

| Trait (typ, trait') as c ->

match typ with

| Type.X \_ ->

unify <| rest @ (expandTraitConstraint c rest)

| \_ ->

unify <| rest @ (getTraitRequirements typ trait')

A função *expandTraintConstraint* recebe uma constraint de Trait e uma lista de constraints. Para cada constraint de igualdade da lista que contenha a variável de tipo original, cria uma nova constraint de trait para adicionar no final da lista. Isso permite a verificação posterior das constraints de trait para variáveis de tipo que não tenham sido unificadas.

let rec expandTraitConstraint (Trait (typ, trait') as constraint') list =

match list with

| [] -> []

| first::rest ->

match first with

| Equals (s, t) | Equals (t, s) when s = typ ->

expandTraitConstraint constraint' <| rest @ [Trait (t, trait')]

| \_ ->

expandTraitConstraint constraint' rest

A função *getTraitRequirements* verifica se um tipo satisfaz a trait . Para isso, ela analisa recursivamente até encontrar a resposta (positiva ou negativa) ou encontrar uma nova variável de tipo. Caso encontre uma variável de tipo, ela cria um novo constraint de trait, que é então adicionado no final da lista de constraints do unify.

É importante ressaltar que cada trait vai possuir uma regra de construção diferente, então é impossível criar um método genérico para isso. A cada nova trait que for adicionado, será necessário aumentar esse método com mais um *match* e novas regras de recursão.

let rec getTraitRequirements typ trait' =

match trait' with

| Equatable ->

match typ with

| Int | Bool -> []

| Type.X x as typ' -> [Trait (typ', Equatable)]

| List typ' -> getTraitRequirements typ' trait'

| Function (\_, \_) ->

raise <| InvalidType "Did not meet equatable trait requirement"

As regras de semântica operacional são as seguintes (regras de propagação de raise não são mostradas):

As regras para a operação “=”, usada acima, são as seguintes:

## Trait *Orderable*

Seguindo das ideias da igualdade, foi criada uma nova trait *Orderable* para tipos que podem ser ordenáveis. Para ser *Orderable*, um tipo também deve ser .

A regra de tipo para todos os operadores de ordenação é a seguinte:

As regras da semântica operacional, com exceção das de propagação de raise, são as seguintes:

As regras para a operação *op’*, usada acima, são as seguintes:

(comparações numéricas normais)

## 

## Operações Booleanas Curto-Circuito

Com a definição padrão de L1, as operações booleanas “e” e “ou” precisam ser definidas na biblioteca padrão. Como elas são definidas como funções, ambos os parâmetros precisam ser avaliados antes de obter o resultado da operação. Isso é diferente do que ocorre em qualquer outra linguagem de programação, nas quais, se o primeiro parâmetro já definir o resultado da operação (verdadeiro para “ou” e falso para “e”), o segundo parâmetro não é avaliado. Para permitir essa funcionalidade em L1, foi criada essa extensão.

Foram criados dois novos termos:

As regras para inferência de tipos são as seguintes:

As regras de semântica operacional são:

## Input e Output

Para permitir entrada e saída, foram necessárias diversas modificações na linguagem. Primeiramente, foram introduzidos 2 novos tipos (e um pseudônimo de tipo).

O tipo *String* é, na realidade, apenas um açúcar sintático na definição, e é tratado exatamente como uma lista de *Char*.

Foram introduzidos novos termos gramaticais:

Dos termos introduzidos, dois deles são valores:

Abaixo estão as regras do sistema de tipos para essa extensão:

Igualmente, as regras de avaliação:

Para lidar com entrada e saída, foi necessário criar duas funções para transformar entre strings nativas de F# e as strings definidas nessa extensão. São duas funções recursivas simples, tratando os dois tipos de string como listas de caracteres e fazendo a conversão entre ambos.

## Polimorfismo e Traits (16/11/2016)

Para unir o sistema de traits com polimorfismo, foi necessário fazer grandes modificações no sistema de tipos. Primeiramente, uma variável de tipo passou a ter, além de uma string para ser identificada, uma lista de traits. Além disso, foi removido o constraint de trait e o resultado de Unify foi modificado para, além de associar nomes de variáveis de tipo a tipos, associar nomes de variáveis de tipos a listas de traits.

Na definição da linguagem, foi modificado o caso de VarType, colocando uma lista de traits além da string (que representa o nome).

type Type =

| VarType of string \* Trait list

Além disso, a função que avalia se um tipo possui uma *trait* ou não foi transferida para o mesmo arquivo da definição. A função, *validadeTrait*, retorna um *Type option*, retornando *None* caso o tipo não possua a *trait* pedidae *Some tipo* quando o tipo possui a *trait*. Caso sejam encontrados *VarTypes* no tipo, a *trait* é adicionada à sua lista de *traits*, retornando um tipo modificado como resultado.

let rec validateTrait trt typ =

match typ with

| Int ->

match trt with

| Orderable | Equatable -> Some Int

| Bool ->

match trt with

| Equatable -> Some Bool

| Orderable -> None

| Char ->

match trt with

| Orderable | Equatable -> Some Char

| Unit ->

match trt with

| Orderable | Equatable -> None

| Function (typ1, typ2) ->

match trt with

| Orderable | Equatable -> None

| List typ1 ->

match trt with

| Orderable | Equatable ->

match validateTrait trt typ1 with

| None -> None

| Some typ1 -> Some <| List typ1

| VarType (x, traits) ->

if List.exists ((=) trt) traits then

Some typ

else

Some <| VarType (x, trt::traits)

O sistema de inferência de tipos foi bastante simplificado, voltando a ter apenas um tipo de constraint (*Tipo1* é igual a *Tipo2*). Ao invés de anotar necessidades de *traits* em contraints separados, esses requerimentos agora são inclusos direto na igualdade, utilizando uma variável de tipo com a *trait* adequada.

O resultado da função *unify* não é mais um mapa associando nomes a tipos. Agora, o resultado é um objeto composto de dois atributos: um mapa associando nomes a tipos e um mapa associando nomes a traits.

A função *unify* foi modificada apenas na parte que diz respeito à igualdade entre uma variável de tipo e um outro tipo qualquer. De forma simplificada, a modificação feita no algoritmo foi a seguinte:

1. Foi encontrado uma constraint do tipo *VarType1* = *Type2*
2. *Type2* é validado para todas as traits existentes em *VarType1*
   1. Caso *Type2* não seja válido, é retornado um erro de validação de trait
   2. Caso *Type2* seja válido, é retornado um novo tipo *Type2’* (que pode ser igual ou diferente de *Type2)*
3. É feita a substituição de todas as ocorrências de *VarType1* por *Type2’* no restante das constraints
4. Caso *Type2* seja diferente de *Type2’*:
   1. São encontradas todas as variáveis de tipo em *Type2’*
   2. As variáveis de tipo encontradas são substituídas no restante das constraints (ou seja, se for encontrada uma *VarType2* em *Type2’*, todas as ocorrências de *VarType2* serão atualizadas com as suas novas traits no restante das constraints).
   3. Para cada variável de tipo encontrada, associar seu nome à sua lista de contraints no dicionário de traits do resultado
5. Associar *VarType1* a *Type2’* no dicionário de substituições do resultado

Também foram feitas leves modificações em outras partes do código (a expressão agora apenas gera dois contraints, foi criado um novo tipo para auxiliar a comunicação de dados, etc).

## Registros e Tuplas (26/11/2016)

Essa extensão traz dois novos tipos de dados e algumas mudanças no sistema de coletas de tipo. Primeiramente, foram adicionados 4 novos termos a linguagem:

Dos termos introduzidos, dois deles são valores:

Para esses novos valores, foi necessário criar dois novos tipos também:

Abaixo estão as regras de avaliação para os termos criados:

Como é possível ver pela regra de avaliação do termo , é possível acessar tanto registros quanto tuplas através do índice. Isso é devido ao fato de registros serem, na realidade, apenas tuplas nomeadas, como é possível fazer na linguagem de programação Swift. Dessa forma, registros são apenas versões mais especializadas de tuplas, sendo possível usá-los em situações que normalmente aceitariam tuplas.

Isso nos leva à outra modificação que foi feita na linguagem para permitir tuplas e registros: subtipos.