The Potatoes Project

A Types Analyzer Project

Inês Justo, Luís Moura, Maria Lavoura, Pedro Teixeira

Versão Final
Universidade de Aveiro

The Potatoes Project

Departamento de Eletrónica, Telecomunicações e Informática
Universidade de Aveiro

Inês Justo, Luís Moura, Maria Lavoura, Pedro Teixeira
inesjusto@ua.pt (84804), luispedromoura@ua.pt (83808)
mjlavoura@ua.pt (84681), pedro.teix@ua.pt (84715)

28 Junho 2018

Resumo

Este relatório começa por descrever como se utiliza o projecto desenvolvido. De seguida, descreve as linguagens desenvolvidas (nomeadamente as instruções que suportam) para a criação de um compilador para a linguagem Java de uma linguagem general-purpose cujas operações estão garantidas por um sistema de tipos. Por último são apresentadas algumas considerações sobre a implementação.

Agradecimentos

Agradecemos ao prof. Miguel Oliveira e Silva pela enorme ajuda dada e pela paciência a esclarecer as muitas dúvidas que foram surgindo.

"x quê? Batatas?"

Conteúdo

Listings

Capítulo 1

Introdução

1.1 Notas sobre o nome do projecto

O nome `Potatoes` é o nome dado quer ao projecto quer à linguagem general purpose desenvolvida e advém da pergunta feita por muitos professores, sobretudo durante os primeiros anos de ensino, quando um aluno não indica as unidades de uma dada operação : *"x quê? Batatas?"*.

1.2 Conteúdos do Repositório

O repositório contém, para além da pasta do relatório, a pasta que contém o código fonte do projecto (*src*):

- *compiler*: Contém as classes associadas à compilação e à análise semântica de código fonte da linguagem general purpose (*Potatoes*) para a linguagem Java;
- *potatoesGrammar*: Contém a gramática da linguagem general purpose *Potatoes*;
- *tests*: Contém ficheiros de teste/exemplo de código fonte para ambas as linguagens desenvolvidas;
- *typesGrammar*: Contém a gramática as classes associadas à intepretação de código fonte da linguagem `Types`;
- *utils*: Contém classes auxiliares.

1.3 Como Instalar

- O repositório não contém as classes resultantes à compilação das gramáticas *ANTLR*. Se for a primeira vez que o projecto é utilizado, ou após correr `antlr4-clean`, é necessário compilar o projecto estando na pasta *src* e utilizando o *script* fornecido:

  `./build`.

- Assume-se que estão instalados e adicionados ao *classpath* os JARs associados ao *ANTLR* e *String Template* (disponíveis na pasta *antlr*). Assume-se também que estão instalados os *scripts* disponibilizados (`antlr4-build`, etc).

1.4 Como Utilizar

- Para compilar um programa válido, são necessários 2 ficheiros, um contendo código fonte à linguagem de definição de tipos (por exemplo *tests/global/CompleteExample_ Types.txt*) e um contendo código fonte da linguagem general purpose (por exemplo *tests/global/CompleteExample.txt*).
- Por exemplo, para compilar um ficheiro *test.txt*, com um ficheiro de tipos *types.txt* basta:
  - Garantir que a primeira linha de código do ficheiro *test.txt* contém a instrução
    `using "types.txt";`
  - Compilar executando
    `./compile text.txt`
  - Caso não existam erros nos ficheiros a compilar, será gerado um ficheiro chamado *test.txt.java*.

Capítulo 2

Documentação

2.1 Linguagem de Definição de Tipos

A linguagem de definição de tipos, definida pelas gramáticas do ficheiro *Types.g4*, define as instruções necessárias para a definição de tipos e de prefixos[1].

2.1.1 Palavras Reservadas

As palavras reservadas da linguagem são: *types* e *prefixes*.

2.1.2 Instruções básicas e comentários

Todos os ficheiros de código fonte desta linguagem têm obrigatoriamente de conter a instrução:

```
types {
    ...
}
```

Listing 2.1: Linguagem de Definição de Tipos - Instrução `types {...}`

podendo ou não conter a instrução:

```
prefixes {
    ...
}
```

Listing 2.2: Linguagem de Definição de Tipos - Instrução `prefixes {...}`

antes ou depois da instrução `types`.

Terminação de instruções

Nenhuma instrução necessita de um terminador.

Comentários

São suportados comentários *in-line*, iniciados pela sequência `//` (equivalente à linguagem Java). Não são suportados comentários *multiline*.

[1] Deprecated. Não há análise semântica para o uso de prefixos na linguagem general purpose e nenhum exemplo fornecido para a linguagem general purpose utiliza prefixos.

2.1.3 Definição de Tipos

Dentro da instrução `types {...}`, podem ser definidos 0 ou mais tipos. Cada tipo pode ser definido através de três instruções distintas:

Tipo Básico (Numérico)

```
1 <ID> "<STRING>"
```

Listing 2.3: Linguagem de Definição de Tipos - Instrução de definição de tipo básico (numérico)

em que:

- `ID` representa o nome do tipo.
- `STRING` representa o nome de impressão do tipo.

Por exemplo:

```
1 meter "m"
```

representa um tipo chamado *meter*, com nome de impressão *m*.

Tipo Derivado (baseado em aritmética entre tipos já-existentes)

```
1 <ID> "<STRING>" : <OPERATION BETWEEN PRE-EXISTING TYPES>
```

Listing 2.4: Linguagem de Definição de Tipos - Instrução de definição de tipo derivado (baseado em aritmética entre tipos já-existentes)

em que:

- `OPERATION BETWEEN PRE-EXISTING TYPES` representa 1 ou mais multiplicações (`typeA * typeB`), divisões (`typeA / typeB`) ou potências (`typeA ^ <INT>`) entre tipos pré-existentes.

Por exemplo:

```
1 velocity "v" : distance / time
```

representa um tipo chamado *velocity*, com nome de impressão *v*, definida como sendo a divisão dos tipos *distance* e *time*.

```
1 velocitySquare "v2" : velocity^2
```

representa um tipo chamado *velocitySquare*, com nome de impressão *v2*, definida como sendo o quadrado do tipo *velocity*.

Tipo Derivado (baseado em conversões entre tipos já-existentes)

```
1 <ID> "<STRING>" : (<REAL NUMBER>) <TYPE ALTERNATIVE ID> | (<REAL NUMBER>) <TYPE
      ALTERNATIVE ID> ...
```

Listing 2.5: Linguagem de Definição de Tipos - Instrução de definição de tipo derivado (baseado em conversões entre tipos já-existentes)

em que:

- o padrão `| (<REAL NUMBER>) <TYPE>` pode repetir-se 0 ou mais vezes.
- `<REAL VALUE>` representa um número real ou uma operação entre números reais. As operações suportadas são potência, multiplicação, divisão, soma e subtração, existindo também associatividade através do uso de parêntesis. Esse número real **representa o factor de conversão do tipo `ID` para o tipo `TYPE ALTERNATIVE ID`.**
- `<TYPE ALTERNATIVE ID>` representa o nome do tipo alternativa.

Por exemplo:

```
1 distance "d" : (-1+ 49725)meter    | (0.025)inch | (0.9)yard
```

representa um tipo chamado *distance*, com nome de impressão *d*, definida como sendo $(-1 + 49725)$ `meter` ou (0.025) `inch` ou (0.9) `yard`.[2]. Em linguagem natural, tal é equivalente a dizer que *1 distance é igual a* $(-1 + 49725)$ ***meters*** *ou* (0.025) ***inches*** *ou* (0.9) ***yards***.

[2]Factores de conversão não correspondem à realidade.

2.1.4 Definição de Prefixos

Dentro da instrução `prefixes {...}`, podem ser definidos 0 ou mais prefixos. Cada prefixo pode ser definido através da seguinte instrução:

```
<ID> "<STRING>" : <REAL VALUE>
```

Listing 2.6: Linguagem de Definição de Tipos - Instrução de definição de prefixo

em que:

- `ID` representa o nome do prefixo.
- `STRING` representa o nome de impressão do prefixo.
- `REAL VALUE` representa um número real ou uma operação entre números reais. As operações suportadas são as mesmas das referidas na sub-subsecção 2.1.3.

Por exemplo:

```
DECA  "da"  : 10^1
```

representa um prefixo chamado *DECA*, com nome de impressão *da*, definida como sendo o valor 10^1.

2.2 Linguagem General-Purpose de Utilização de Tipos

A linguagem general purpose, definida pelas gramáticas do ficheiro *Potatoes.g4*, define as instruções necessárias para operações básicas com variáveis de tipos distintos, cujo universo é definido pelos ficheiros de código-fonte da linguagem de tipos descrita anteriormente.

A maioria das instruções suportadas seguem a sintaxe típica de linguagens como *C* ou Java, pelo que optamos por detalhar apenas as instruções longe dessa sintaxe típica.

2.2.1 Palavras Reservadas

As palavras reservadas da linguagem são: *using*, *main*, *fun*, *return*, *if*, *else*, *for*, *while*, *number*, *boolean*, *string*, *void*, *false*, *true*, *print*, *println*.

2.2.2 Instruções básicas e comentários

Terminação de instruções

Todas as instruções, à excepção de instruções condicionais e de repetição, são necessariamente terminadas por `;`.

Comentários

São permitidos comentários de linha e de bloco (*multiline*) no formato equivalente à linguagem Java:

```
// commented line
/*
   commented block;
*/
```

Listing 2.7: Linguagem General-Purpose - Comentários

Importar um ficheiro de tipos

O código fonte da linguagem de definição de tipos e o código fonte da linguagem *Potatoes* têm de estar separados em ficheiros diferentes. Existe, com o propósito de importar um ficheiro com diferentes definições de tipos, o comando `using` seguido de uma String com o *path* para o ficheiro pretendido. Por exemplo:

```
using "src/TypesDefinitionFiles.txt";
```

Listing 2.8: Linguagem General-Purpose - Instrução `using`

Esta instrução é **obrigatória** e tem de constituir a **primeira instrução** no ficheiro.

2.2.3 Instruções condicionais e de repetição

São permitidas instruções de repetição `for` e `while` e condicionais `if`, `else if` e `else` em formato semelhante à linguagem Java.

2.2.4 Output

As funções de impressão estão incluídas na linguagem *Potatoes* com as instruções `print` e com funcionamento semelhante à linguagem Java, com a limitação de não permitirem operações.

No caso específico da impressão de variáveis, é impresso o seu valor seguido do seu nome de impressão definido no código fonte da linguagem de definição de tipos.

```
meter m1 = (meter) 5;
println("This code is " + m1 + "long!");     // output: This code is 5 m long!
```

Listing 2.9: Linguagem General-Purpose - Impressão

2.2.5 Funções

A implementação de funções, apesar de definida deste o início do projeto, não pôde ser implementada. Ainda assim, manteve-se o formato de definição de uma função *"main"*. Assim, após a declaração do ficheiro de definição de tipos é possível fazer declarações de variáveis globais e atribuição de valores e declaração de funções.

Para criar a função *main* deverá ser usada a assinatura:

```
fun main{
    ...
}
```

Listing 2.10: Linguagem General-Purpose - Assinatura da Função *Main*

2.2.6 Declaração de variáveis

A linguagem *Potatoes* disponibiliza 3 tipos base além daqueles definidos na Linguagem de Definição de Tipos: *number*, um tipo numérico real adimensional, *boolean* e *string*.

A declaração de variáveis, apesar de apresentar uma sintaxe em tudo semelhante à linguagem Java, tem de ter em conta a compatibilidade entre tipos. Em particular, há 2 situações distintas:

- `Inicialização de variável com atribuição dinâmica de valor.`

  ```
  meter m = (meter) 3;
  ```

  Uma vez que a gramática de tipos implementa operações com dimensões, a atribuição de um valor numérico adimensional deixa de fazer sentido. Pelo que é sempre necessário fazer o *cast* desse valor para o tipo a inicializar.

- `Atribuição de valor usando uma variável de tipo compatível.`

  ```
  meter m = yardVar;
  ```

  A compatibilidade entre tipos implica que podem ser considerados como unidades de uma mesma dimensão, pelo que essa atribuição de valor é direta sem *cast* havendo apenas a necessidade de se fazer a conversão do valor numérico com o factor correspondente.

2.2.7 Operações

A gramática Potatoes suporta operações numéricas (%,;*,/,+,-) com todos os tipos numéricos, e lógicas com tipos numéricos e booleanos (<,>,<=,>=,==,!=,!) com uso semelhante à linguagem Java mas condicionadas pelo sistema de tipos.

Por exemplo, para as seguintes declarações:

```
meter m = (meter) 1;
yard y = (yard) 2;
```

- `Adição e Subtração` são permitidas entre tipos numéricos iguais ou compatíveis.

  ```
  meter m2 = m + m;  // correto
  meter m3 = m + y;  // correto
  meter m4 = m + 1;  // errado (metro e adimensional incompatíveis)
  ```

- `Multiplicação e Divisão` são permitidas entre qualquer tipo numérico, sendo que a operação poderá criar unidades novas que estejam ou não definidas inicialmente.

  ```
  meter m2 = m * m;   // operação correta, atribuição errada
  meter m3 = m * y;   // operação correta, atribuição errada
  number n1 = m / m;  // operação correta e atribuição correta
  ```

- `Módulo e Potência` são apenas realizáveis entre um tipo numérico e um valor numérico (ou o tipo *number*).

  ```
  meter m2 = m^2;      // operação correta, atribuição errada
  meter m3 = m \% 2    // operação correta, atribuição correta
  number n1 = m \% m;  // operação errada
  ```

- `Comparações "<, >, <=, >="` são apenas permitidas entre tipos numéricos compatíveis.

```
1 boolean b1 = m < m2; // operação permitida
2 boolean b2 = m >= y; // operação permitida
3 boolean b3 = m > b1; // operandos não compatíveis
```

- `Comparações "==, !="` são apenas permitidas exclusivamente entre tipos booleanos ou entre tipos numéricos compatíveis.
- `Operações lógicas "&&, ||, !"` são permitidas entre tipos booleanos e resultados booleanos de comparações.

Capítulo 3

Notas sobre a Implementação

Embora não seja parte integrante das especificações das linguagens, é relevante apresentar, em traços muitos gerais, a implementação quer do do interpretador da linguagem de tipos quer do analisador semântico da linguagem general purpose.

3.1 Interpretador da linguagem de definição de tipos

Os ficheiros com código fonte da linguagem de definição de tipos, após a criação com sucesso da árvore sintática associada a estes, são processados através do seu interpretador, responsável quer pela análise semântica, quer pela interpretação do código.

A interpretação do código fonte, em ficheiros desta linguagem, implica a criação de estruturas de dados utilizadas depois na análise de tipos feita posteriormente na análise semântica de código fonte da linguagem general purpose (secção 3.2):

- Tabela de Tipos (`typesTable`) : mapa que corresponde o nome do tipo à sua instância da classe `Type`;
- Grafo de Tipos (`typesGraph`) : grafo em que cada vértice é uma instância da classe `Type` e cada aresta é uma instância da classe `Factor`. Por outras palavras, um grafo que representa a ligação entre os tipos e as suas alternativas.

3.2 Análise semântica da linguagem general purpose

O ponto mais difícil da operação foi o desafio de implementar uma estrutura de métodos capaz de lidar com os tipos compatíveis e todas as suas implicações. Ao permitir operações que são normalmente fechadas à mesma unidade passa a ser necessário nas multiplicações e divisões além de aceitar qualquer tipo, decidir quando é necessário fazer conversões e quais fazer. Por exemplo para tipos de volume definidos de formas diferentes:

```
1 metricVolume    "m^3"    : meter * meter * meter
2 yardVolume      "yd^3"   : yard * yard * yard
3 weirdVolume     "wV"     : meter * meter * yard
```

qualquer operação que envolva 3 variáveis de tipo compatível com os anteriores é válida, no entanto fazendo as conversões corretamente, o valor numérico de volume será diferente na atribuição a cada um dos tipos anteriores.

Deste modo, há duas implementações fundamentais para a resolução correta de operações:

- `Códigos de Tipos`

  Cada tipo simples ou derivado é identificado com um número primo. Os tipos obtidos através de composição de outros tipos são a composição dos seus códigos únicos. Desta forma qualquer que seja a operação realizada com tipos, dois códigos iguais serão sempre obrigatoriamente identificadores de tipos iguais.

- `Algoritmo` ***`Greedy`*** `de Cálculo`

  O algoritmo de cálculo está enraizado na utilização de duas estruturas auxiliares específicas:

  - A classe *Type* que contêm especificamente um *checklist* de tipos pertencentes à sua composição.
  - Um grafo de tipos compatíveis (secção 3.2) que liga duplamente tipos (os vértices) e contém os fatores de conversão (as arestas) entre eles e a sua relação de herança (não implícita no grafo em si).

  Por exemplo, para a operação seguinte:

```
1 weirdVolume = (meter * meter * meter * gram) / ounce;
```

  Os primeiros passos seriam:

  1. `meter * meter`: são marcados na checkList de *weirdvolume* os dois tipos *meter*. Fica por marcar um tipo *yard*;
  2. `area * meter` :
     (a) `area`
        i. é feita uma tentativa sem sucesso de marcar *area* na checklist de *weirdVolume*;
        ii. é feita uma tentativa sem sucesso de converter *area* para o primeiro tipo já marcado;
        iii. é feita uma tentativa sem sucesso de converter *area* para o seu parente hierárquico mais alto;
        iv. a variável é aceite com o seu valor sem conversão e sem alterar o seu tipo;
     (b) `meter` é feita tentativa com sucesso de marcar meter na checklist de *weirdVolume*.

  Sendo que o algoritmo continua de igual forma e resolve com sucesso qualquer operação com tipos compatíveis com herança simples e vários tipos iguais com herança múltipla.

  No caso de haver tipos diferentes com herança múltipla, para operações e atribuições invulgares, o algoritmo pretende alcançar o resultado correto a maioria das vezes, não sendo garantido o resultado.

  Prevemos que a única outra solução envolveria calcular todas as combinações possíveis, que pode facilmente tornar-se impraticável mesmo com exemplos relativamente simples.