Revisão de programação funcional em Quint

Aula para disciplina de Métodos Formais

Gabriela Moreira

Departamento de Ciência da Computação - DCC
Universidade do Estado de Santa Catarina - UDESC

25 de março de 2024

Conteúdo

Outline

Instalando as ferramentas - Dependências

- NodeJS >= 18
- Java Development Kit >= 17

Quint

1. Com acesso de administrador
   - Instalar: `npm i @informalsystems/quint -g`
   - Executar: `quint --help`
2. Sem acesso de administrador
   - Instalar: `npm i @informalsystems/quint --user`
   - Executar: `npx quint --help`
   - Essa instalação é local, então você vai precisar instalar de novo se trocar de pasta

TLC

- TLC para linha de comando
  - Opcional, se quiser usar pela linha de comando
- Extensão no VSCode
  - Após instalar, abra um arquivo `.tla`, aperte F1 e procure o comando "TLA+: Check model with TLC"

Apalache

- Apalache para linha de comando
- É usado internamente pelo Quint quando invocamos `quint verify`. Só precisa baixar separadamente se quiser utilizar com TLA+.
- Usuários de **Windows**: Baixar a versão disponível no moodle, que inclui um `.bat`
  - Também precisam executar o servidor do Apalache manualmente: `.\apalache-mc.bat server`
  - Se não fizerem isso, `quint verify` não vai funcionar

Restrições de Quint e TLA+

- Não há recursão*
  - *Existe recursão em TLA+, mas foi adicionado posteriormente. Não suportado pelo Apalache.
- Não há laços de repetição (`for`, `while`)
- Não há manipulação de `string`

Forma dos operadores em Quint

Todos os operadores (exceto os com símbolos, como +) podem ser aplicados de duas formas em Quint:

1. `operador(arg0, ..., argn)`
2. `arg0.operador(arg1, ..., argn)`

Escolha a forma que você acha mais fácil de ler!

Outline

Conjuntos!

Conjuntos, ou *Sets*, são a principal estrutura de dados em Quint em TLA+.

- A não ser que a **ordem** dos elementos seja realmente importante, use conjuntos em vez de listas.
- Importante! Isso é um ponto que vou avaliar no trabalho de vocês.

Conjuntos!

Conjuntos, ou *Sets*, são a principal estrutura de dados em Quint em TLA+.

- A não ser que a **ordem** dos elementos seja realmente importante, use conjuntos em vez de listas.
- Importante! Isso é um ponto que vou avaliar no trabalho de vocês.

O tipo de um conjunto é dado por `Set[<elemento>]`. Ou seja, um conjunto de inteiros tem tipo `Set[int]`.
Criando conjuntos:

```
Set(1, 2, 3) // Set(1, 2, 3)
1.to(3) // Set(1, 2, 3)
```

`map`, seu novo melhor amigo

Em linguagens funcionais, usamos muito a função `map`, que permite a aplicação deu uma função a cada elemento de um conjunto.

```
Set(1, 2, 3).map(x => x * 2) // Set(2, 4, 6)
```

map, seu novo melhor amigo

Em linguagens funcionais, usamos muito a função `map`, que permite a aplicação deu uma função a cada elemento de um conjunto.

```
Set(1, 2, 3).map(x => x * 2) // Set(2, 4, 6)
```

Pode ser usado com lambdas (operadores anônimos), como acima, ou com operadores nomeados:

```
pure def quadrado(x: int): int = x * x

Set(1, 2, 3).map(quadrado) // Set(1, 4, 9)
```

map com operadores de múltiplos argumentos

Dado um conjunto de duplas, podemos aplicar um operador em cada uma das duplas. Mas cuidado! Se o operador espera dois argumentos, temos que fazer o unpacking das duplas, utilizando parênteses duplos.

```
pure def soma(x: int, y: int): int = x + y

Set((1, 1), (2, 3)).map(soma)
// static analysis error: error: [QNT000] Expected 1 arguments,

Set((1, 1), (2, 3)).map(((a, b)) => soma(a, b))
// Set(2, 5)
```

map com operadores que esperam uma dupla

```
pure def somaDupla(t: (int, int)): int = t._1 + t._2

Set((1, 1), (2, 3)).map(somaDupla)
// Set(2, 5)
```

`map` não resolve tudo!

O `map` só nos ajuda quando queremos um conjunto como retorno.

- Dado um conjunto de números, retorne um conjunto do quadrado desses números.
- Dado um conjunto de pessoas, retorne um conjunto com as idades das pessoas.

map não resolve tudo!

O map só nos ajuda quando queremos um conjunto como retorno.

- Dado um conjunto de números, retorne um conjunto do quadrado desses números.
- Dado um conjunto de pessoas, retorne um conjunto com as idades das pessoas.

map não pode ser usado para as seguintes operações:

- Dado um conjunto de números, retorne a soma de todos esses números.
- Dado um conjunto de números, retorne um conjunto apenas com os números pares.

`map` não resolve tudo!

O `map` só nos ajuda quando queremos um conjunto como retorno.

- Dado um conjunto de números, retorne um conjunto do quadrado desses números.
- Dado um conjunto de pessoas, retorne um conjunto com as idades das pessoas.

`map` não pode ser usado para as seguintes operações:

- Dado um conjunto de números, retorne a soma de todos esses números.
- Dado um conjunto de números, retorne um conjunto apenas com os números pares.

Lembram quais funções podem ajudar com esses casos?

`filter` permite filtrar o conjunto

Exemplo: Dado um conjunto de números, retorne um conjunto apenas com os números pares.

```
Set(1, 2, 3, 4).filter(x => x % 2 == 0)
// Set(2, 4)
```

`fold` permite acumular um valor ao percorrer o conjunto

Argumentos do `fold`

1. O conjunto sobre o qual iterar
2. Um valor inicial para o acumulador
3. Um operador que recebe dois argumentos (o acumulador, e o elemento iterado), e retorna o novo valor para o acumulador

`fold` permite acumular um valor ao percorrer o conjunto

Argumentos do `fold`

1. O conjunto sobre o qual iterar
2. Um valor inicial para o acumulador
3. Um operador que recebe dois argumentos (o acumulador, e o elemento iterado), e retorna o novo valor para o acumulador

Exemplo: Dado um conjunto de números, retorne a soma de todos esses números.

```
Set(1, 2, 3, 4).fold(0, (acc, i) => acc + i)
// 10
```

`fold` permite acumular um valor ao percorrer o conjunto

Argumentos do `fold`

1. O conjunto sobre o qual iterar
2. Um valor inicial para o acumulador
3. Um operador que recebe dois argumentos (o acumulador, e o elemento iterado), e retorna o novo valor para o acumulador

Exemplo: Dado um conjunto de números, retorne a soma de todos esses números.

```
Set(1, 2, 3, 4).fold(0, (acc, i) => acc + i)
// 10
```

Atenção: Não assumir nada sobre a ordem em que os elementos são iterados.

Exercício: `map` e `filter` com `fold`

Exercício: Re-escreva nossos exemplos anteriores usando `fold` ao invés de `map` e `filter`:

1. Dado um conjunto de números, retorne um conjunto do quadrado desses números.
2. Dado um conjunto de números, retorne um conjunto apenas com os números pares.

Operações de conjuntos

1. União: `union`
2. Intersecção: `intersect`
3. Diferença: `exclude`

Operadores booleanos para conjuntos

1. Pertence, $\in$: `in`, `contains`
   `e.in(S)` é equivalente a `S.contains(e)`
2. Contido, $\subseteq$: `subseteq`
3. Para todo, $\forall$: `forall`
4. Existe, $\exists$: `exists`

Powerset - Conjunto das partes

```
Set(1, 2).powerset()
// Set(Set(), Set(1), Set(2), Set(1, 2))
```

Útil quando queremos gerar várias possibilidades para escolher dentre elas.

`flatten`, para conjuntos de conjuntos

Um conjunto de conjuntos de elementos to tipo `t` pode ser convertido em um conjunto de elementos do tipo `t` com o operador `flatten`.

```
Set(Set(1, 2), Set(1, 3)).flatten()
// Set(1, 2, 3)
```

Outline

Maps

`Map` é a estrutura de dicionário em Quint. Em TLA+, essa mesma estrutura tem nome de função.

Maps

`Map` é a estrutura de dicionário em Quint. Em TLA+, essa mesma estrutura tem nome de função.

O tipo de um mapa é dado por `<chave> -> <valor>`. Ou seja, um mapa de inteiros para strings tem tipo `int -> str`.

Maps

`Map` é a estrutura de dicionário em Quint. Em TLA+, essa mesma estrutura tem nome de função.

O tipo de um mapa é dado por `<chave> -> <valor>`. Ou seja, um mapa de inteiros para strings tem tipo `int -> str`.

Criando `Maps`:

```
Map(1 -> "a", 2 -> "b")
// Map(1 -> "a", 2 -> "b")

Set((1, "a"), (2, "b")).setToMap()
// Map(1 -> "a", 2 -> "b")

Set(1, 2).mapBy(x => if (x < 2) "a" else "b")
// Map(1 -> "a", 2 -> "b")
```

Chaves e valores

Para obter todas as chaves:

```
Map(1 -> "a", 2 -> "b").keys()
// Set(1, 2)
```

E os valores?

```
val m = Map(1 -> "a", 2 -> "b")
m.keys().map(k => m.get(k))
// Set("a", "b")
```

Acessando e atualizando

`set` atualiza um elemento existente, e `put` pode criar um novo par chave-valor.

```
val m = Map(1 -> "a", 2 -> "b")

m.get(1)
// "a"

m.set(1, "c")
// Map(1 -> "c", 2 -> "b")

m.set(3, "c")
// runtime error: error: [QNT507] Called 'set' with a non-existi

m.put(3, "c")
// Map(1 -> "a", 2 -> "b", 3 -> "c")
```

Atualizando com setBy

setBy é uma utilidade para quando queremos fazer uma operação sobre um valor existente no mapa.

```
val m = Map("a" -> 1, "b" -> 2)

m.set("a", m.get("a") + 1)
// Map("a" -> 2, "b" -> 2)

m.setBy("a", x => x + 1)
// Map("a" -> 2, "b" -> 2)
```

Criando todos os Maps possíveis

Para criar todos os Maps possíveis dado um domínio e um co-domínio, podemos usar o setOfMaps:

```
Set(1, 2).setOfMaps(Set("a", "b"))
// Set(Map(1 -> "a", 2 -> "a"), Map(1 -> "b", 2 -> "a"),
//     Map(1 -> "a", 2 -> "b"), Map(1 -> "b", 2 -> "b"))
```

Outline

Tuplas

Tuplas são combinações de tipos diferentes em um mesmo valor, onde a ordem dos elementos é o que define o tipo esperado.

Tuplas

Tuplas são combinações de tipos diferentes em um mesmo valor, onde a ordem dos elementos é o que define o tipo esperado.

O tipo de uma tupla é dado por `(t0, ..., tn)`. Uma tupla com tipo `(int, str, bool)` permite valores como `(1, "a", true)`.

Tuplas

Tuplas são combinações de tipos diferentes em um mesmo valor, onde a ordem dos elementos é o que define o tipo esperado.

O tipo de uma tupla é dado por `(t0, ..., tn)`. Uma tupla com tipo `(int, str, bool)` permite valores como `(1, "a", true)`.

Existe um único jeito de criar uma tupla:

```
(1, "a", true)
```

Acessando itens

Itens de tuplas são acessados com `._1`, `._2`, `._3`, ...

Não existe `._0`, a contagem inicia do 1.

```
val t = (1, "a", true)

t._1
// 1

t._3
// true
```

Criando todas as tuplas possíveis

Para criar um conjunto com todas as tuplas possíveis com elementos em dados conjuntos, usamos o `tuples`:

```
tuples(Set(1, 2), Set("a", "b"))
// Set((1, "a"), (2, "a"), (1, "b"), (2, "b"))

tuples(Set(1), Set("a", "b"), Set(false))
// Set((1, "a", false), (1, "b", false))
```

Outline

Records

`Records` são combinações de tipos diferentes em um mesmo valor, onde os elementos são nomeados.

Records

`Records` são combinações de tipos diferentes em um mesmo valor, onde os elementos são nomeados.

O tipo de um *record* é dado por `{ field0: t0, ..., fieldn: tn }`. Um *record* com tipo `{ nome: str, idade: int }` permite valores como `{ nome: "Gabriela", idade: 25 }`.

Acessando e atualizando

```
val r = { nome: "Gabriela", idade: 25 }

r.nome
// "Gabriela"

r.with(idade, 26)
// { nome: "Gabriela", idade: 26 }

{ ...r, idade: 26 }
// { nome: "Gabriela", idade: 26 }

r
// { nome: "Gabriela", idade: 25 }
```

Outline

Listas

Listas são como conjuntos, porém com uma ordem definida e, possivelmente, com elementos repetidos. Em TLA+, essa mesma estrutura tem nome de sequência.

Listas

Listas são como conjuntos, porém com uma ordem definida e, possivelmente, com elementos repetidos. Em TLA+, essa mesma estrutura tem nome de sequência.

O tipo de uma lista é dado por `List[<elemento>]`. Ou seja, uma lista de inteiros tem tipo `List[int]`.

Listas

Listas são como conjuntos, porém com uma ordem definida e, possivelmente, com elementos repetidos. Em TLA+, essa mesma estrutura tem nome de sequência.

O tipo de uma lista é dado por `List[<elemento>]`. Ou seja, uma lista de inteiros tem tipo `List[int]`.

Criando listas:

```
[1, 2, 3]
// [1, 2, 3]

List(1, 2, 3)
// [1, 2, 3]

range(1, 4)
// [1, 2, 3]
```

Acessando

```
val l = [1, 2, 3]

l[1]
// 2

l.head()
// 1

l.tail()
// [2, 3]
```

Atualizando

```
val l = [1, 2, 3]

l.replaceAt(0, 5)
// [5, 2, 3]

l.concat([4, 5])
// [1, 2, 3, 4, 5]

l.append(4)
// [1, 2, 3, 4]

l
// [1, 2, 3]
```

Filtrando listas

`slice` retorna uma nova lista com um intervalo de elementos da lista original.

```
[1, 2, 3].slice(0, 1)
// [1]
```

`select` é semelhante ao `filter` (de conjuntos).

```
[1, 2, 3, 4, 5].select(x => x > 3)
// [4, 5]
```

`foldl` e `foldr`

Diferente do `fold` pra conjuntos, a operação de *fold* sobre listas respeita uma ordem específica. `foldl` (*fold left*) vai iterar da esquerda pra direita, enquanto `foldl` (*fold right*) vai iterar da direita pra esquerda.

foldl e foldr

Diferente do `fold` pra conjuntos, a operação de *fold* sobre listas respeita uma ordem específica. `foldl` (*fold left*) vai iterar da esquerda pra direita, enquanto `foldl` (*fold right*) vai iterar da direita pra esquerda.

Atenção também para a ordem dos argumentos do operador dado como último argumento.

```
[1, 2, 3].foldl([], (acc, i) => acc.append(i))
// [1, 2, 3]

[1, 2, 3].foldr([], (i, acc) => acc.append(i))
// [3, 2, 1]
```

Use `indices` para fazer um `map`

O operador `map` não funciona pra listas. Conseguimos reproduzir essa funcionalidade usando o operador `indices`, que returna o índices de uma lista (isso é, 0 até $length(l) - 1$).

```
val l = [1, 2, 3]
def f(x) = x + 1

l.indices().map(i => f(l[i]))
// Set(2, 3, 4)
```

Perceba que o resultado aqui é um conjunto. Para que o resultado seja uma lista, temos que usar `foldl` ou `foldr`.

Outline

Definindo tipos (*aliases*)

Nomes de tipos devem sempre iniciar com letra maiúscula.

```
type Idade = int

val a: Idade = 1
```

Tipos soma

```
type Periodo = Manha | Tarde | Noite

type EstadoLogin = Logado(str) | Deslogado

type Opcional[a] = Algum(a) | Nenhum
```

Recursos

- Cheatsheet Quint
- Documentação dos builtins
  - em Quint
  - em Markdown
- Spells - bibliotecas auxiliares
  - PS: Quer contribuir pra opensource? Esse é um ótimo local pra começar
- Manual do Quint

Exercícios

1. Escreva um operador que recebe um conjunto e retorna a média dos valores.
2. Dado um conjunto de `records` do tipo `{ nome: str, idade: int }`, escreva um operador que recebe esse conjunto e retorna a diferença de idade entre o mais velho e o mais novo.
3. Defina um valor que contenha todos os conjuntos possíveis com valores inteiros de 1 a 10, com tamanho maior que 2 e menor que 5.
4. Escreva um operador que calcule o fatorial de um número. Lembre-se que recursão não é permitida.
5. Escreva um operador que recebe uma lista e retorna um mapa onde as chaves são os elementos da lista, e os valores são inteiros representando a quantidade de ocorrências daquele elemento na lista.