Primeiras Especificações

Aula para disciplina de Métodos Formais

Gabriela Moreira

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

04 de setembro de 2024

Conteúdo

Correção exercícios

REPL

Ações em Quint

Modos de Quint

Jarros de Água

Runs

Outline

Correção exercícios

REPL

Ações em Quint

Modos de Quint

Jarros de Água

Runs

Correção dos exercícios

Vamos corrigir juntos usando a REPL.

Outline

Correção exercícios

REPL

Ações em Quint

Modos de Quint

Jarros de Água

Runs

Usando a REPL

x' = x + 1 na REPL

Usando a REPL

x' = x + 1 na REPL

```
>>> var x: int

>>> x' = 0
true
>>> x
0
>>> action step = x' = x + 1

>>> step
true
>>> step
true
>>> x
2
```

PS: Definições aninhadas

Sempre podemos definir operadores, valores e ações dentro do próprio corpo de outra definição. Semelhante ao `let` do Haskell.

- Para isso, basta escrever a(s) definição(ões) seguidas por fim da expressão final a ser retornada.
  - Sem separar com vírgula!

```
1 pure def f(x) = {
2   pure def quadrado(y) = y * y
3   pure def dobro(y) = 2 * y
4   quadrado(x) + dobro(x)
5 }
```

PS: Definições aninhadas

Sempre podemos definir operadores, valores e ações dentro do próprio corpo de outra definição. Semelhante ao `let` do Haskell.

- Para isso, basta escrever a(s) definição(ões) seguidas por fim da expressão final a ser retornada.
  - Sem separar com vírgula!

```
1 pure def f(x) = {
2   pure def quadrado(y) = y * y
3   pure def dobro(y) = 2 * y
4   quadrado(x) + dobro(x)
5 }
```

As chaves `{}` são opcionais, e você pode usar elas quando quiser pra deixar mais claro o escopo, por exemplo:

```
1 val foo = pure def f(x) = 2 * x { f(5) }
```

Outline

Correção exercícios

REPL

Ações em Quint

Modos de Quint

Jarros de Água

Runs

Ações

Ações são expressões booleanas que podem incluir o operador *primed* (').

Ações

Ações são expressões booleanas que podem incluir o operador *primed* (').

Contudo, não podemos usar os operadores booleanos normais sobre essa expressão. Quint não permite isso para evitar possíveis confusões. Por exemplo, as seguintes operações **não** são permitidas:

- `not(x' = 1)`
- `x' = 1 or x' = 2`
- `x' > 1 and x' < 2`
- `Set(1, 2, 3).exists(i ==> x' = i)`

Ações

Ações são expressões booleanas que podem incluir o operador *primed* (').

Contudo, não podemos usar os operadores booleanos normais sobre essa expressão. Quint não permite isso para evitar possíveis confusões. Por exemplo, as seguintes operações **não** são permitidas:

- `not(x' = 1)`
- `x' = 1 or x' = 2`
- `x' > 1 and x' < 2`
- `Set(1, 2, 3).exists(i ==> x' = i)`

Os equivalentes em TLA+ são permitidos. Em Quint, somos forçados a escrever as ações de um jeito específico, de uma forma que elas não possam ser confundidas com não-ações.

all e any

- Em vez de `and`, usamos `all`

```
1  action incrementa_x_e_y = all {
2    x' = x + 1,
3    y' = y + 1,
4  }
```

all e any

- Em vez de `and`, usamos `all`

```
1  action incrementa_x_e_y = all {
2    x' = x + 1,
3    y' = y + 1,
4  }
```

- Em vez de `or`, usamos `any`

```
1  action incrementa_ou_decrementa_x = any {
2    x' = x + 1,
3    x' = x - 1,
4  }
```

nondet e oneOf

- Em vez de `exists`, usamos `nondet` e `oneOf`

```
1  action x_recebe_algum_valor = {
2    nondet valor = Set(1, 2, 3).oneOf()
3    x' = valor
4  }
```

nondet e oneOf

- Em vez de `exists`, usamos `nondet` e `oneOf`

```
1  action x_recebe_algum_valor = {
2    nondet valor = Set(1, 2, 3).oneOf()
3    x' = valor
4  }
```

- Importante: O `oneOf` só pode ser usado nesse cenário (dentro de uma ação, em uma definição do tipo `nondet`, sem operações adicionais sobre ele).

nondet e oneOf

- Em vez de `exists`, usamos `nondet` e `oneOf`

```
1  action x_recebe_algum_valor = {
2    nondet valor = Set(1, 2, 3).oneOf()
3    x' = valor
4  }
```

- Importante: O `oneOf` só pode ser usado nesse cenário (dentro de uma ação, em uma definição do tipo `nondet`, sem operações adicionais sobre ele).
  - A syntaxe em Quint é dessa forma para deixar o não-determinismo explícito, mas na lógica (TLA), isso ainda é um `exists`.

nondet e oneOf

- Em vez de `exists`, usamos `nondet` e `oneOf`

```
1  action x_recebe_algum_valor = {
2    nondet valor = Set(1, 2, 3).oneOf()
3    x' = valor
4  }
```

- Importante: O `oneOf` só pode ser usado nesse cenário (dentro de uma ação, em uma definição do tipo `nondet`, sem operações adicionais sobre ele).
  - A syntaxe em Quint é dessa forma para deixar o não-determinismo explícito, mas na lógica (TLA), isso ainda é um `exists`.
  - Por exemplo, não é possível usar o `oneOf` na implementação de uma função, i.e. para encontrar o máximo em um conjunto.

if também pode ser usado em ações

- O `if` não tem uma versão especial pra ações. Usamos ele normalmente.

```
1  action incrementa_x_se_par = {
2    if (x % 2 == 0) {
3      x' = x + 1
4    } else {
5      x' = x
6    }
7  }
```

Balanceamento de atualizações

Todas as ações em Quint devem ser devidamente balanceadas, e uma variável nunca pode ser atualizada mais de uma vez em uma mesma ação.

Balanceamento de atualizações

Todas as ações em Quint devem ser devidamente balanceadas, e uma variável nunca pode ser atualizada mais de uma vez em uma mesma ação.

Isso significa que:

- Todas as ações em um `any` devem atualizar as mesmas variáveis

Balanceamento de atualizações

Todas as ações em Quint devem ser devidamente balanceadas, e uma variável nunca pode ser atualizada mais de uma vez em uma mesma ação.

Isso significa que:

- Todas as ações em um `any` devem atualizar as mesmas variáveis
- Em um `if`, os blocos `then` e `else` devem atualizar as mesmas variáveis

Balanceamento de atualizações

Todas as ações em Quint devem ser devidamente balanceadas, e uma variável nunca pode ser atualizada mais de uma vez em uma mesma ação.

Isso significa que:

- Todas as ações em um `any` devem atualizar as mesmas variáveis
- Em um `if`, os blocos `then` e `else` devem atualizar as mesmas variáveis
- Em um `all`, as variáveis atualizadas por cada ação não podem se repetir

Balanceamento de atualizações

Todas as ações em Quint devem ser devidamente balanceadas, e uma variável nunca pode ser atualizada mais de uma vez em uma mesma ação.

Isso significa que:

- Todas as ações em um `any` devem atualizar as mesmas variáveis
- Em um `if`, os blocos `then` e `else` devem atualizar as mesmas variáveis
- Em um `all`, as variáveis atualizadas por cada ação não podem se repetir

Essas restrições valem pra TLA+ também, mas em TLA+ isso só será detectado na hora de rodar o *model checker*. Em Quint, vocês vão ver sublinhados vermelhos no editor.

Exemplos desbalanceados

Os exemplos a seguir **não** são permitidos no Quint. PS: Para vê-los no editor, você precisa declarar as variáveis (`var x: int` e `var y: int`).

```
action any_desbalanceado = any {
  x' = 1,
  y' = 2,
}

action if_desbalanceado = {
  if (x > 0) {
    x' = 1
  } else {
    y' = 2
  }
}
```

Exemplos com múltiplas atualizações da mesma variável

Os exemplos a seguir **não** são permitidos no Quint.

```
action all_multiplas_atualizacoes = all {
  x' = 1,
  x' = 2,
}

action a1 = x' = 1
action a2 = x' = 2
action all_multiplas_atualizacoes = all { a1, a2 }
```

Outline

Correção exercícios

REPL

Ações em Quint

Modos de Quint

Jarros de Água

Runs

Modos de Quint

Percebam como nas últimas aulas conversamos sobre coisas um tanto diferentes:

- Aula passada, definimos
  - **operadores** com `def` e `pure def`
  - **valores** com `val` e `pure val`
- Nessa aula, usaremos ações com `action`

Definição dos modos

Primeiramente, temos a diferença entre `val` e `def`

- `val` (ou `pure val`): Valores, onde não há nenhum parâmetro.
- `def` (ou `pure def`): Operadores, onde há pelo menos um parâmetro.

Definição dos modos

Primeiramente, temos a diferença entre `val` e `def`

- `val` (ou `pure val`): Valores, onde não há nenhum parâmetro.
- `def` (ou `pure def`): Operadores, onde há pelo menos um parâmetro.

Esses são os **modos** das definições. Eles definem o tanto de acesso que as definições tem às variáveis.

- `pure def` e `pure val`: Nenhum acesso. Como funções puras, onde o mesmo input vai sempre gerar o mesmo output.
- `def` e `val`: Leitura.
- `action`: Escrita e Leitura.

Definição dos modos

Primeiramente, temos a diferença entre `val` e `def`

- `val` (ou `pure val`): Valores, onde não há nenhum parâmetro.
- `def` (ou `pure def`): Operadores, onde há pelo menos um parâmetro.

Esses são os **modos** das definições. Eles definem o tanto de acesso que as definições tem às variáveis.

- `pure def` e `pure val`: Nenhum acesso. Como funções puras, onde o mesmo input vai sempre gerar o mesmo output.
- `def` e `val`: Leitura.
- `action`: Escrita e Leitura.

Além destes, temos alguns modos adicionais:

- `nondet`: Para declarações com não determinismo (que usam `oneOf`).
- `temporal`: Para fórmulas temporais.
- `run`: Para execuções mais específicas, permitindo operadores que ajudam a definir o passo a passo esperado.

Outline

Correção exercícios

REPL

Ações em Quint

Modos de Quint

Jarros de Água

Runs

Exercício - Jarros de Água

- Você tem dois jarros:
  1. um grande, com capacidade de 5 litros
  2. um pequeno, com capacidade de 3 litros
- Você tem uma torneira de água com capacidade infinita
- Você pode descartar água a qualquer momento
- É possível, com precisão, ter uma medida de 4 litros de água?

Variáveis

```
1 module jarros {
2   var grande: int
3   var pequeno: int
4
5   ...
6 }
```

Ações

Tente escrever as ações abaixo, definindo os valores para `grande` e `pequeno` em cada uma delas. Nenhuma dessas ações precisa de parâmetros.

```
1  action enche_grande
2  action enche_pequeno
3  action esvazia_grande
4  action esvazia_pequeno
5  action grande_pro_pequeno
6  action pequeno_pro_grande
```

Estado inicial

```
1  action init = all {
2    grande' = 0,
3    pequeno' = 0,
4  }
```

Tentando resolver na REPL

```
quint -r jarros.qnt::jarros
```

Comece com `init`, e verifique os valores de `grande` e `pequeno`. Depois, tente invocar as outras ações, lembrando que o objetivo é chegar em um estado onde um dos jarros tem 4 litros.

Ação de próximo estado e invariante

Agora, vamos usar o *model checker* para encontrar a solução. Para isso, vamos definir:

- `step`, a ação de próximo estado. A cada passo, podemos tomar qualquer uma das ações definidas.
- `inv`, nossa invariante. Nesse caso, esperamos que a invariante seja quebrada, para obter nossa solução como contraexemplo.

```
1  action step = any {
2    enche_grande,
3    enche_pequeno,
4    esvazia_grande,
5    esvazia_pequeno,
6    grande_pro_pequeno,
7    pequeno_pro_grande,
8  }
9
10 val inv = grande != 4
```

Encontrando um contraexemplo

```
$ quint verify jarros.qnt --invariant=inv
An example execution:

[State 0] { grande: 0, pequeno: 0 }
[State 1] { grande: 5, pequeno: 0 }
[State 2] { grande: 2, pequeno: 3 }
[State 3] { grande: 2, pequeno: 0 }
[State 4] { grande: 0, pequeno: 2 }
[State 5] { grande: 5, pequeno: 2 }
[State 6] { grande: 4, pequeno: 3 }

[violation] Found an issue (156ms).
error: found a counterexample
```

Outline

Correção exercícios

REPL

Ações em Quint

Modos de Quint

Jarros de Água

Runs

Runs

- Representação de uma **execução** finita.
  - Pode ser uma execução concreta, ou
  - Pode ter não determinismo, representando mais de uma execução
- Descreve como reproduzir uma ou mais execuções, se possível

Runs

- Representação de uma **execução** finita.
  - Pode ser uma execução concreta, ou
  - Pode ter não determinismo, representando mais de uma execução
- Descreve como reproduzir uma ou mais execuções, se possível

Essa é uma feature exclusiva do Quint, e não há uma representação equivalente em TLA+.

- Em TLA+, só podemos usar o estado em si para determinar cada passo a ser dado.
- Em Quint, as runs permitem definir isso externamente, sem necessidade de manipular o estado.

Runs

- Representação de uma **execução** finita.
  - Pode ser uma execução concreta, ou
  - Pode ter não determinismo, representando mais de uma execução
- Descreve como reproduzir uma ou mais execuções, se possível

Essa é uma feature exclusiva do Quint, e não há uma representação equivalente em TLA+.

- Em TLA+, só podemos usar o estado em si para determinar cada passo a ser dado.
- Em Quint, as runs permitem definir isso externamente, sem necessidade de manipular o estado.

O propósito de runs está relacionado a testes, e não tem função alguma para o *model checker*.

Definindo uma run para a solução dos jarros

```
run solution =
  init
    .then(enche_grande)
    .then(grande_pro_pequeno)
    .then(esvazia_pequeno)
    .then(grande_pro_pequeno)
    .then(enche_grande)
    .then(grande_pro_pequeno)
    .expect(grande == 4)
```

Definindo uma run para a solução dos jarros

```
1  run solution =
2    init
3      .then(enche_grande)
4      .then(grande_pro_pequeno)
5      .then(esvazia_pequeno)
6      .then(grande_pro_pequeno)
7      .then(enche_grande)
8      .then(grande_pro_pequeno)
9      .expect(grande == 4)
```

Adicionando o `expect` no final, essa run também funciona como um teste

Rodando runs como testes

```
quint test jarros.qnt --match solution

  jarros
    ok solution passed 1 test(s)

  1 passing (12ms)
```

Invocando runs na REPL

```
$ quint -r jarros.qnt::jarros
>>> solution
true
>>> grande
4
>>> pequeno
3
```

FIM

Primeiras Especificações

Aula para disciplina de Métodos Formais

Gabriela Moreira

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

04 de setembro de 2024