Why3: Desenvolvimento baseado no estado - Documento original

Simão Cunha

5 de dezembro de 2022

Uma ferramenta de verificação dedutiva como o Why3 pode ser usado para verificar o comportamento da coleção de funções que trabalham com um estado partilhado, tal como as classes em programação orientada aos objetos (partilhando variáveis de instância), contratos inteligentes em desenvolvimento blockchain, ou programação concorrente multithreading (baseada em memória partilhada).

Vamos ilustrar isto através de um exemplo onde modelamos funções que representam operações de uma conta bancária.

O estado global partilhado armazena o saldo de cada conta identificado por um número.

```
1  module Accounts_MapImp
2    use int.Int
3    type accNumber = int
4    type amount    = int
5
6    val open (n :accNumber) : ()
7    val deposit (n :accNumber) (x :amount) : ()
8    val withdraw (n :accNumber) (x :amount) : ()
9    val transfer (from :accNumber) (to_ :accNumber) (x :amount) : ()
```

Precisámos de uma forma de associar saldo com números de contas. Usaremos o tipo*finite map* do Why3 para este fim.

Map types implementam dicionários, a biblioteca Why3 `https://why3.lri.fr/stdlib/fmap.html` dispõe:

- um map type ao nível lógico (módulo `Fmap`) que não pode ser extraído para código;
- e dois tipos de map de programação, que podem ser extraídas (tipicamente implementadas em hashtables): tipo funcional - módulo *MapApp*; tipo imperativo - módulo *MapImp*;

Uma vez que queremos programar com *maps*, usaremos *types* de programação.

Começaremos por usar o tipo imperativo. A leitura e escrita de funções deste tipo são especificadas na biblioteca da forma abaixo. Note-se que as funções `find` e `add` usadas nos contratos são funções sinónimas ao nível lógico: o comportamento dos maps ao nível programacional são especificados usando o tipo lógico `Fmap`.

```
    val find (k: key) (m: t 'v) : 'v
      requires { mem k m }
      ensures  { result = m[k] }
      ensures  { result = find k m }

    val add (k: key) (v: 'v) (m: t 'v) : unit
      writes  { m }
      ensures { m = add k v (old m) }
```

Os tipos de programação são usado ao *clonar* o módulo respetivo à nossa maneira. Neste caso concreto, instanciaremos o tipo de *map keys*:

```
clone fmap.MapImp with type key = accNumber
```

O tipo de *maps* com `accNumber` como chaves está agora disponível com o nome `t`. O seguinte bloco declara a variável de estado `accounts` como um map com valores do tipo *amount*:

```
val accounts : t amount
```

Quereremos escrever funções que façam as seguintes tarefas:

- abrir (`open`) uma nova conta com um dado número de conta;
- depositar (`deposit`) fundos numa dada conta;
- levantar (`withdraw`) fundos de uma dada conta;
- transferir (`transfer`) fundos internamente de uma conta para outra

Como um primeiro passo, consideremos uma linguagem de especificação naturarl do seu comportamento. Por exemplo:

- dado um número de conta `n`, a chamada `open n` vai inserir o par ("n", 0) no dicionário *accounts*;
- dado um número de conta `n` e um montante `x`, a chamada `deposit n x` vai adicionar `x` ao saldo atual da conta `n`

O próximo passo é discutir o que as funções devem fazer caso recebam valores de argumento inesperados. O que deve acontecer se:

- na chamada `open n`, o número de conta `n` já existir, i.e. já está presente no domínio de `accounts`?
- na chamada `deposit n x`, o número `n` não for um número válido de conta (i.e. não estiver no domínio de `accounts`) ou `x` for um número negativo?

Iremos escolher "não programar defensivamente": as funções vão tomar como garantido que as situações acima não acontecem.

Então as definições são realmente simples:

```
1       let open (n :accNumber) : ()
2       = add n 0 accounts
3
4       let deposit (n :accNumber) (x :amount) : ()
5       = let bal = find n accounts in
6         add n (bal+x) accounts
```

Agora, na ausência da programação defensiva, é da responsabilidade do "chamador"ter a certeza de que todas as chamadas satisfazem as condições desejadas, o que devem ser incluídas nos contratos das funções do "chamador"como pré-condições:

```
1       let open (n :accNumber) : ()
2         requires { not mem n accounts }
3       = add n 0 accounts
4
5       let deposit (n :accNumber) (x :amount) : ()
6         requires { mem n accounts /\ x > 0 }
7       = let bal = find n accounts in
8         add n (bal+x) accounts
```

Iremos também incluir pós-condições nos contratos das funções. ISto não vai só permitir que nós conseguiremos provar que as implementações respeitam as especificações (expressas abaixo em linguagem natural), mas também que nenhuma chamada que foi feita desrespeita as pré-condições.

```
1   let open (n :accNumber) : ()
2     requires { not mem n accounts }
3     ensures  { mem n accounts /\ find n accounts = 0 }
4     ensures  { forall a :accNumber. mem a accounts <-> mem a (old accounts) \/ a = n }
5     writes   { accounts }
6   = add n 0 accounts
7
8   let deposit (n :accNumber) (x :amount) : ()
9     requires { mem n accounts /\ x > 0 }
10    ensures  { find n accounts = find n (old accounts) + x }
11    ensures  { forall a :accNumber. mem a accounts  /\ a <> n
12                   -> find a accounts = find a (old accounts) }
13    ensures  { forall a :accNumber. mem a accounts <-> mem a (old accounts) }
14    writes   { accounts }
15  = let bal = find n accounts in
16    add n (bal+x) accounts
```

Note-se que:

- A pós-condição destacada em `deposit` expressa um facto óbvio que é implícito na especificação em linguagem natural, mas deve ser explícito no contrato: todos os saldos são preservados com exceção da conta `n`;
- Os contratos também incluem pós-condições em relação aos conjuntos de chaves (domínios) do mapping antes e depois da execução de cada função;
- As *frame conditions* `writes ...` tornam explícito os *efeitos* das funções, i.e. as partes do estado global que são modificados por eles

1 Invariantes de estado

Nós poderemos querer provar que certas propriedades do estado global sempre "seguram" (hold), i.e. são invariantes de todas as funções que alteram o estado.

Por exemplo:

- O saldo de todas as contas é não-negativo

Tais propriedades devem ser tratadas por simplesmente incluí-las simultaneamente pré e pós-condições em todas as funções. Devemos, depois, adicionar o seguinte bloco aos contratos das funções:

```
1  requires { forall a :accNumber. mem a accounts -> find a accounts >= 0 }
2  ensures  { forall a :accNumber. mem a accounts -> find a accounts >= 0 }
```

2 Record types e invariantes de tipo

Podámos alternativamente usar tipos declarativos/funcionais para *maps*. Iremos ilustrar o seu uso com uma implementação alternativa do módulo abaixo:

```
1  module Accounts_MapApp_Record
2    use int.Int
3    type accNumber = int
4    type amount    = int
5    clone fmap.MapApp with type key = accNumber
6
7    type state = { mutable bal: t amount }
8      invariant  { forall a :accNumber. mem a bal -> find a bal >= 0 }
9      by { bal = create() }
10
11   val accounts :state
```

Declarar o estado usando um *record type* permite-nos incluir um invariante de estado diretamente no tipo da definição, o que torna desnecessário incluí-lo explicitamente como pré e pós condições nas definições da função (condições de verificação vão sendo criadas automaticamente para todas as funções que alteram estados, assegurando a preservação do invariante de tipo).

Note-se que:

- A cláusula `by` é obrigatória. O seu *role* é dar uma testemunha que satisfaça o invariante de tipo. Simplesmente damos um mapeamento vazio como exemplo (que é devolvido pela função `create`);
- Os campos do *record* são por defeito imutáveis (tal como na programação funcional. Campos mutáveis devem ser explicitamente identificados como se efetua abaixo.

Tanto no código como na especificação, o valor do saldo vai ter de ser referenciado usando a notação de campo do *record*. Note-se que o uso do operador de atribuição `<-`: uma vez que estamos a usar o tipo de mapeamento aplicativo, a função `add` pode não ter efeitos colaterais no estilo funcional, toma o *map* e devolve um novo *map*.

```
1  let open (n :accNumber) : ()
2    requires { not mem n accounts.bal }
3    ensures  { mem n accounts.bal /\ find n accounts.bal = 0 }
4    ensures  { forall a :accNumber. mem a accounts.bal <-> mem a (old accounts.bal) \/ a = n }
5    writes   { accounts.bal }
6  = accounts.bal <- add n 0 accounts.bal
7
8  let deposit (n :accNumber) (x :amount) : ()
9    requires { mem n accounts.bal /\ x > 0 }
10   ensures  { find n accounts.bal = find n (old accounts.bal) + x }
11   ensures  { forall a :accNumber. mem a accounts.bal  /\ a <> n
12               -> find a accounts.bal = find a (old accounts.bal) }
13   ensures  { forall a :accNumber. mem a accounts.bal <-> mem a (old accounts.bal) }
14   writes   { accounts.bal }
15 = let baln = find n accounts.bal in
16   accounts.bal <- add n (baln+x) accounts.bal
```