Guião para aula laboratorial de Verificação Formal (2018/19)

Why3 (1)

O Why3 é uma plataforma para verificação dedutiva de programas. O Why3 formece uma linguagem para especificação e programação, o WhyML, e utiliza (vários) sistemas externos de prova (automática ou assistida) para testar a validade das condicões de verificação geradas. O Why3 vem equipado com uma grande biblioteca de teorias lógicas e de estruturas de dados.

Podem-se escrever programas WhyML diretamente (e obter programas OCaml correctos por construção através de um mecanismo de extração automatizado), mas o WhyML também é usado como linguagem intermediária para a verificação de programas (C, Java e Ada).

Esta aula é dedicada à utilização do Why3 na sua vertente de ferramenta lógica.

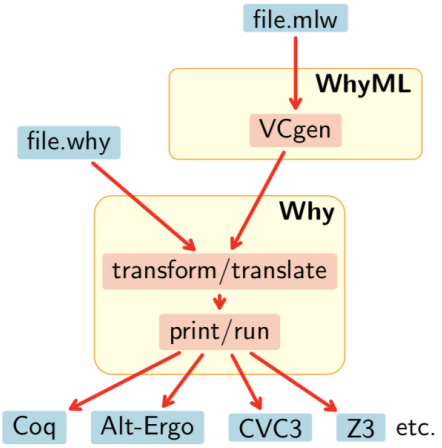
No website do Why3 encontratrá diversa documentação e muitos exemplos.

Este guião tem por base o tutorial *"Deductive Program Verification with Why3"*, de Jean-Christophe Filliâtre, assim como exemplos do manual do Why3.

1 Introdução

A plataforma Why3 fornece uma linguagem lógica (chamada Why) e uma linguagem de programação (chamada WhyML). A lógica é uma extensão da lógica de primeira ordem com polimorfismo, tipos de dados algébricos e predicados indutivos. A linguagem de programação é uma linguagem do estilo ML de primeira ordem, com características imperativas, concordância de padrões e excepções. As duas linguagens estão intimamente ligadas: qualquer símbolo lógico pode ser usado nos programas, e a lógica é usada para especificar programas (via pré e pós-condições, invariantes de ciclo, etc.).

A extracção de condições de verificação é feita com base no cálculo da *weakest precondition*. As fórmulas resultantes são então sujeitas a várias transformações, para depois serem enviadas para (vários) demonstradores de teoremas externos.



Há 3 maneiras básicas de usar o Why3:

1. **como ferramenta para lógica de primeira ordem**

   É possível usar o Why3 sem escrever um único programa. Podemos usar apenas a logica (fazendo declarações, definições, axiomatizações e lançando objectivos de prova) e ver o Why3 como um mero front-end para um vasto conjunto de theorem provers.

2. **para verificar algoritmos e estruturas de dados**

   Embora as caracteristicas imperativas do WhyML sejam limitadas (o aliasing não é permitido), é possível implementar muitos programas e estruturas de dados. Quando alguma estrutura de dados não pode ser implementada, é geralmente é fácil modelá-la e realizar o resto da verificação pode ser realizada partir daí. Os programas WhyML podem ser traduzidos para código OCaml executável.

3. **como uma linguagem interédia**

   O Why3 é muito adquado a ser usado como linguagem intermédia, na verificação de programas escritos em linguagem de programação mainstream (como o C, Java ou Ada). Neste caso, é desenhado um modelo de memória adquado usando a lógica Why3 e, em seguida, as construções do programa são compiladas em construções WhyML. A tarefa de extracção de condições de verificação (e sua prova) fica a cargo do Why3.

Ilustraremos a utilização do Why3 na sua vertente de ferramenta lógica (na aula de hoje) e na sua vertente de ferramenta para derificação deductiva de algoritmos (na próxima aula).

2 Logica

Pode trabalhar com o Why3 ao nível da linha de comando ou usar um ambiente gráfico de desenvolvimento integrado (IDE). O IDE permite navegar num ficheiro ou conjunto de ficheiros, e verificar a validade dos objetivos de prova com diversos theorem provers, de forma amigável. Caso não tenha feito ainda, deverá executar a detecção automática dos provers com comando

```
$ why3 config --detect-provers
```

2.1 hello_proof.why

Este primeiro ficheiro contém objectivos de prova muito simples e ilustra também a utilização da teoria de inteiros do Why3.

Analise o ficheiro `hello_proof.why` carregando-o no IDE com o comando

```
$ why3 ide hello_proof.why
```

Qualquer declaração deve ocorrer dentro de uma teoria, neste caso a teoria `HelloProof`. Ela contém três objectivos de prova (`G1, G2, G3`). `G1` e `G2` são fórmulas proposicionais básicas, enquanto `G3` envolve alguma aritmética de inteiros e, portanto, requer a importação da teoria da aritmética de inteiros da biblioteca do Why3.

1. Tente provar todos os objectivos com um qualquer solver. A tarefa de prova corrente pode ser visualizada a qualquer momento no tab *"Task"*.

2. É evidente que um deles não será nunca provado porque não é válido. No entanto, trata-se da conjunção de duas fórmulas, uma das quais é válida. Isole e prove a parte válida do objectivo utilizando a transformação *"split"*.

2.2 demo_first.why

Este ficheiro ilustra a utilização de predicados, funções não interpretadas, e quantificadores.

1. Tente provar ambos os objectivos com um SMT solver.
2. Um dos objectivos contém um quantificador existencial e poderá não ser provado pelo solver. Caso isso aconteça, será sempre possível efectuar a prova utilizando um *proof assistant*. Faça a prova em Coq desse objectivo. Para isso:
   (a) Depois de primir o botão Coq prima *"Edit"*. Será lançado o CoqIde com este objectivo de prova.
   (b) Conclua a prova em Coq.
   (c) Grave o script e feche o CoqIde.
   (d) Na janela do Why3, prima *"Replay"*. O objectivo deverá aparecer como provado.

2.3 demo_list.why

Neste ficheiro, começa-se por definir o tipo algébrico das listas polimórficas. Para isso, introduz-se uma teoria `List` que contém a definição do tipo `list 'a` (as variáveis de tipo começam por `'`).

```
theory List
  type list 'a = Nil | Cons 'a (list 'a)
end
```

Este ficheiro pode ser carregado no `why3 ide`, mas também pode ser processado em na linha de comando, fazendo

```
$ why3 execute demo_list.why
```

Vamos agora acrescentar à teoria o seguinte predicado recursivo.

```
  predicate mem (x: 'a) (l: list 'a) =
    match l with
    | Nil -> false
    | Cons y r -> x = y \/ mem x r
    end
```

Ao processar de novo o ficheiro o sistema automaticamente testa a terminação de `mem`. Esperimente alterar a chamada recursiva para `men x l` e veja como a sistema reage.

Acrescente agora à teoria o seguinte objectivo de prova:

```
  goal G1: mem 2 (Cons 1 (Cons 2 (Cons 3 Nil)))
```

Pode agora tentar resolver este objectivo usando um SMT solver a que o Why3 tenha acesso.

```
$ why3 prove -P alt-ergo demo_list.why
```

Verá que o objectivo `G1` foi validado. É claro que pode usar `why3 ide` para este fim, e será o mais adquado.

Vamos agora definir a função que calcula o comprimento de uma lista, mas criando uma segunada teoria `Length` que vai importar a teoria `List` e a teoria da aritmética de inteiros da biblioteca do Why3, `int.Int`.

```
theory Length
  use List
  use int.Int

  function length (l: list 'a) : int =
    match l with
    | Nil -> 0
    | Cons _ r -> length r + 1
    end
end
```

Ao fazer *"Save all and Refresh session"* no IDE o sistema, mais uma vez, verifica automaticamente para a terminação.

Vamos agora declarar o seguinte lema:

```
  lemma length_nonnegative: forall l:list 'a. length(l) >= 0
```

Tente agora validar este lema com os SMT solvers que tem disponíveis. Provavelmente nenhum consegue validar o lema. `Unknown` ou `Timeout` devem ser a as respostas obtidas...

De facto, provar o este lema requer indução, e isso está fora do alcance dos SMT solvers. Prove então este lema usando o CoqIde.

No entanto, a maioria dos SMT solvers é capaz de provar, usando o lema `length_nonnegative` como hipótese, o seguinte:

```
  goal G2: forall x: int, l: list int. length (Cons x l) > 0
```

Isto é o que distigue `goal` de `lemma`: o `lemma` introduz um objectivo de prova que pode ser usado a seguir. O IDE (na janela superior direita) ilustra graficamente o facto do lema `length_nonnegative` ser de facto uma hipótese do objetivo `G2`.

Vamos agora definir a noção de lista ordenada, começando por trabalhar com uma lista de inteiros. Introduzimos uma nova teoria `SortedList` com esse propósito, contendo aa declaração de um predicado indutivo `sorted`.

```
theory SortedList
  use List
  use int.Int

  inductive sorted (list int) =
    | sorted_nil: sorted Nil
```

```
    | sorted_one: forall x: int. sorted (Cons x Nil)
    | sorted_two: forall x y: int, l: list int.
                  x <= y -> sorted (Cons y l) -> sorted (Cons x (Cons y l))
end
```

Esta declaração define `sorted` como o menor predicado que satisfaz os três "axiomas" `sorted_nil`, `sorted_one` e `sorted_two`.

Podemos agora colocar o seguinte objectivo de prova

```
goal sorted123: sorted (Cons 1 (Cons 2 (Cons 3 Nil)))
```

que é facilmente resolvido por qualquer SMT solver.

2.4 demo_sortedlist.why

No secção anterior definimos o perdicado `sorted` apenas para listas de inteiros. Seria mais útil defini-lo de forma mais genérica para que possa ser reaproveitado para outros tipos de listas e para outras relações de ordem entre os seus elementos.

A lógica do Why3 é uma lógica de primeira ordem, pelo que não é possível passar a relação de ordem como um argumento do predicado `sorted`. A alternativa é modificar a teoria `SortedList` para usar um tipo de dados abstrato t em vez de inteiros e um predicado binário não interpretado $\leq$ sobre t.

```
theory SortedList
  use List
  type t
  predicate (<=) t t

  axiom le_refl: forall x: t. x <= x
  axiom le_asym: forall x y: t. x <= y -> y <= x -> x = y
  axiom le_trans: forall x y z: t. x <= y -> y <= z -> x <= z
...
```

Adapte a definição do perdicado `sorted`.

Para lidar com o caso particular de listas de inteiros:

1. Crie agora uma nova teoria na qual importe listas e inteiros:

   ```
   theory SortedIntList
     use int.Int
     use List
   ```

2. Instâncie a teoria genérica `SortedList` com o tipo `int` e com a relação de ordem `<=` sobre inteiros. Para fazer isso, use o comando `clone` (em vez de `use`) da seguinte forma:

   ```
   clone SortedList with type t = int, predicate (<=) = (<=),
                         lemma le_refl, lemma le_asym, lemma le_trans
   ```

Este comando faz uma cópia da teoria `SortedList`, enquanto substitui o tipo `int` pelo tipo `t` e a relação de ordem `<=` sobre inteiros para o predicado não interpretado `<=` (neste caso os dois predicados têm o mesmo nome, mas isso não é um requisito). Este comando introduz a declaração de um novo predicado indutivo `sorted`, com um argumento do tipo `list int`. Podemos usá-lo para o seguinte objetivo:

```
goal sorted123: sorted (Cons 1 (Cons 2 (Cons 3 Nil)))
```

Note que o comando `clone` ao incorporar os axiomas relativos à relação de ordem `<=` como lemas, faz com que eles tenham que ser provados.

Faça o *"Refresh"* do ficheiro e observe os objectivos de prova gerados. Use os solvers para fazer as provas.

Como ilustrado neste exemplo, a clonagem permite que as teorias genéricas sejam instanciadas posteriormente de várias maneiras. Isso é análogo a classes genéricas em Java ou functores em ML. Mas é um pouco mais flexível, pois os parâmetros não precisam ser definidos de uma vez por todas. Uma teoria é assim parametrizada de várias maneiras simultaneamente.

No exemplo acima, podemos introduzir ainda outra teoria genérica: a da relação de ordem. Ela contém o tipo t, a relação de ordem $\leq$ e os três axiomas.

1. Declare a teoria `Order`
2. Pode agora clonar esta teoria para dentro da teoria `SortedIntList` e obter uma teoria exactamente igual à anterior.

```
theory SortedList
  use List
  clone export Order
...
```

Assim tem a possibilidade de reutilizar a teoria `Order` em outros contextos.

Note que, neste caso, estamos a fazer a clonagem com `export`. Assim, cada símbolo s de `Order` acessível simplesmente como s. Caso contrário o nome do símbolo seria `Order.`s

A biblioteca do Why3 é construída dessa maneira. Além do benefício óbvio da fatorização, dividir as declarações em pequenas teorias também permite controlar de forma fina o contexto de cada objectivo de prova. Limitar o tamanho do contexto lógico pode melhorar significativamente os desempenhos dos SMT solvers.

2.5 Em resumo

A lógica do Why3 é uma extensão da lógica de primeira ordem com polimorfismo, tipos de dados algébricos mutuamente recursivos, símbolos de função/predicado mutuamente recursivos, predicados mutuamente indutivos e construções `let-in`, `match-with` e `if-then-else`. As declarações lógicas são de quatro tipos diferentes:

- declarações de tipo;

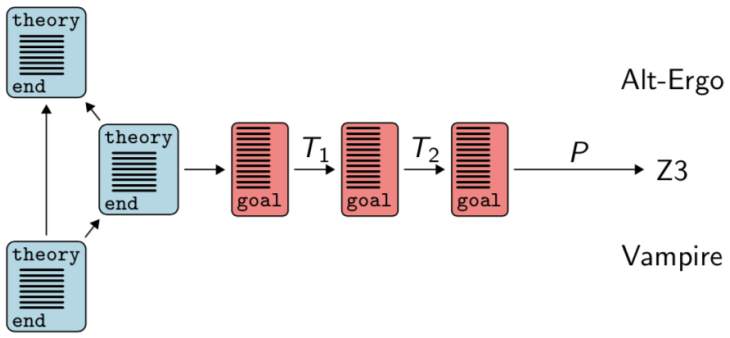
- declarações de funções ou predicados;
- declarações de predicados indutivos;
- declarações de axioma, lema ou de objectivo de prova.

As declações lógicas estão organizadas em *teorias*. Um teoria T_1 pode ser

- usada (`use`) em outra teoria T_2. Nesse caso, símbolos de T_1 são *compartilhados*, axiomas de T_1 permanecem axiomas, lemas de T_1 tornam-se axiomas e os objectivos de prova de T_1 são descartados.
- clonada (`clone`) em outra teoria T_2. Nesse caso, declarações de T_1 são copiadas ou *substituídas*, axiomas de T_1 permanecem como axiomas ou tornam-se lema/objectivo, lemas de T_1 tornam-se axiomas e os objectivos de prova de T_1 são descartados.

Uma das vantagens do Why3 é fornecer uma tecnologia para falar com os *solvers*. Existem vários theorem solvers disponíveis e eles têm distintas linguagens lógicas, teorias pré-definidas ou sistemas de tipos. O Why3 fornece uma linguagem comum e uma ferramenta comum para usá-los a todos.

Essa tecnologia é organizada em torno da noção de *tarefa*. Uma tarefa é um contexto lógico, que é uma lista de declarações, seguida por um único objetivo de prova. As tarefas são extraídas das várias teorias. O Why3 pega numa dada tarefa e um solver alvo, e realiza uma série de transformações na tarefa, de modo que ela se encaixe na lógica do solver.



Este processo é conduzido por um ficheiro (*driver* na terminologia do Why3) que indica as transformações a serem aplicadas, como ler o output do solver, etc. Esse driver pode ser configurado pelo utilizador, por exemplo, para adicionar suporte para um novo solver.

2.6 genealogy.why

Esta teoria ilustra a utilização de predicados definidos e axiomas envolvendo funções; usa também o predicado de igualdade.

1. Estude atentamente a teoria e prove os objectivos contidos no ficheiro.
2. Observe que a teoria define funções `father` e `mother`, e um predicado `parent` (pai ou mãe). Acrescente agora à teoria a definição de 3 novos predicado `grandfather`, `grandmother`, e `grandparent`, utilizando as funções e predicados referidos.

3. Prove depois o seguinte:

   (a) ser `grandparent` de alguém é equivalente a ser seu avô ou avó;

   (b) qualquer avô (resp. avó) é do sexo masculino (resp. feminino);

   (c) ninguém tem mais do que dois avôs (`grandparent`).

2.7 Einstein's logic problem

Vamos usar o Why3 para resolver um pequeno quebra-cabeça conhecido como *Einstein's logic problem*. O problema é o seguinte:

> *Five persons, of five different nationalities, live in five houses in a row, all painted with different colors. These five persons own different pets, drink different beverages, and smoke different brands of cigars. We are given the following information:*
>
> *- The Englishman lives in a red house;*
> *- The Swede has dogs;*
> *- The Dane drinks tea;*
> *- The green house is on the left of the white one;*
> *- The green house's owner drinks coffee;*
> *- The person who smokes Pall Mall has birds;*
> *- The yellow house's owner smokes Dunhill;*
> *- In the house in the center lives someone who drinks milk;*
> *- The Norwegian lives in the first house;*
> *- The man who smokes Blends lives next to the one who has cats;*
> *- The man who owns a horse lives next to the one who smokes Dunhills;*
> *- The man who smokes Blue Masters drinks beer;*
> *- The German smokes Prince;*
> *- The Norwegian lives next to the blue house;*
> *- The man who smokes Blends has a neighbour who drinks water.*
>
> *The question is: what is the nationality of the fish's owner?*

Analise o ficheiro `einstein.why` onde parte do problema já está codificado. Repare que se declarou uma teoria genérica que define a noção de *bijecção*, com dois tipos abstractos, t e u, juntamente com duas funções de um para o outro, e dois axiomas afirmando que essas funções são inversas uma da outra.

```
theory Bijection
  type t
  type u

  function of t : u
  function to_ u : t

  axiom To_of : forall x: t. to_ (of x) = x
  axiom Of_to : forall y: u. of (to_ y) = y
end
```

Esta teoria será de grande utilidade na formalização do problema dada a sua natureza.

Declara-se então uma nova teoria, `Einstein`, que contem todos os indivíduos do problema, establece a relação de vizinhança, e faz as associações bijectivas adquadas entre indivíduos, clonando apropriadamente a teoria da `Bijection`. Repare na utilização de `as` para renomear a teoria ao ser clonada. Por exemplo,

```
clone Bijection as Color with type t = house, type u = color, axiom.
```

permitirá chamar `Color.of` à função `to` clonada.

A seguir, axiomatizam-se as pistas do puzzle lógico, e, finalmente, declara-se o objetivo de prova na teoria `Problem`.

Complete a codificação do puzzle, excute-o no Why3 e dê resposta à questão:

What is the nationality of the fish's owner?