Universidade do Minho
Escola de Engenharia
Departamento de Informática

Bárbara Andreia Cardoso Ferreira

Conversão para Why3 de Formalizações em Coq

December 2021

Universidade do Minho
Escola de Engenharia
Departamento de Informática

Bárbara Andreia Cardoso Ferreira

Conversão para Why3 de Formalizações em Coq

Master dissertation
Integrated Master's in Informatics Engineering

Dissertation supervised by
Jorge Miguel Matos Sousa Pinto
Maria João Gomes Frade

December 2021

AGRADECIMENTOS

A concretização da presente dissertação não teria sido possível sem a colaboração, estímulo e empenho de diversas pessoas, às quais quero expressar toda a minha gratidão e apreço. Em particular:

Ao meu orientador, Professor Jorge Sousa Pinto, agradeço pelo acompanhamento e disponibilidade, pela prontidão sempre demonstrada, pela sua forma dinâmica de esclarecer as minhas questões e de me ajudar a resolver os problemas que foram surgindo.

À minha co-orientadora, Professora Maria João Frade, agradeço pelos esclarecimentos e pelo apoio prestado durante a realização deste trabalho.

A toda a minha família, em especial aos meus pais e irmãos, por me apoiarem incondicionalmente a todos os níveis, por depositarem total confiança nas minhas capacidades e dando-me a força necessária para ultrapassar os momentos mais complicados e prosseguir os meus objetivos.

Ao meu namorado, agradeço pelo amor, pela compreensão, pelo companheirismo e inesgotável paciência, pelo apoio e constante encorajamento a fim de prosseguir os meus sonhos.

Resta agradecer aos meus amigos pela partilha de conhecimentos, pelo espírito de amizade e entreajuda demonstrados. Obrigada por estarem sempre disponíveis, por me motivarem e me acompanharem diariamente. Com certeza, amigos para à vida.

A todos quero manifestar os meus sinceros agradecimentos.

DECLARAÇÃO DE INTEGRIDADE

Declaro ter atuado com integridade na elaboração do presente trabalho académico e confirmo que não recorri à prática de plágio nem a qualquer forma de utilização indevida ou falsificação de informações ou resultados em nenhuma das etapas conducente à sua elaboração.

Mais declaro que conheço e que respeitei o Código de Conduta Ética da Universidade do Minho

ABSTRACT

The present document is the dissertation report that describes all the work developed in the scope of the project "Conversion of Coq Formalizations to Why3".

This work has as main objective the conversion of the definitions of some functional algorithms, as well as their proofs, developed in Coq, to Why3. That is, to use the two languages of Why3 to realize to what extent it is possible to formalize algorithms defined in Coq. These formalizations belong to the "Software Foundations"book on functional algorithms. This demonstrates how a variety of fundamental algorithms can be specified and verified mechanically.

Through the conversion of three different algorithms it was possible to see that Why3 is a very versatile language. It shows that it is possible to convert Coq formalizations to its language without too much difficulty, especially using its program language, **WhyML**.

The intention, beyond the definition, is also to explore the kind of proofs that the two languages (logical and program) of Why3 allow one to perform. In the program language, proofs are extremely simple, achieved, for the most part, through just the automatic *solvers*. In the Why3 logic language it is possible to perform some inductive proofs using proof transformations. However, these are restricted only to proofs that use structural induction. It is more natural to use the program language, because in this language the inductive proof is automatic and follows the structure of the function definition, without the need to define induction principles.

Comparing the two Why3 languages, the program language is actually more interesting than the logical language.

KEYWORDS Coq, Induction, Logic, Formal Proofs, Sorting Algorithms, Why3, WhyML.

RESUMO

O presente documento consiste no relatório da dissertação que descreve todo o trabalho desenvolvido no âmbito do projeto "Conversão para Why3 de Formalizações em Coq".

Este trabalho tem como objetivo principal a conversão das definições de alguns algoritmos funcionais, bem como as suas provas, desenvolvidas em Coq, para Why3. Ou seja, utilizar as duas linguagens do Why3 para perceber até que ponto é possível formalizar algoritmos definidos em Coq. Estas formalizações pertencem ao livro da "Software Foundations" sobre algoritmos funcionais. Este demonstra como uma variedade de algoritmos fundamentais podem ser especificados e verificados mecanicamente.

Através da conversão de três algoritmos diferentes foi possível perceber que o Why3 apresenta uma linguagem bastante versátil. Este revela ser possível sem grandes dificuldades a conversão das formalizações Coq para a sua linguagem, principalmente utilizando a sua linguagem de programas, **WhyML**.

A intenção, para além da definição, é também explorar o tipo de provas que as duas linguagens (lógica e de programas) do Why3 permitem realizar. Na linguagem de programas, as provas são extremamente simples, conseguidas, na sua grande maioria, através apenas dos *solvers* automáticos. Na linguagem lógica do Why3 é possível realizar algumas provas indutivas recorrendo às transformações de prova. No entanto, estas ficam restritas apenas às provas que utilizem a indução estrutural. O mais natural é utilizar a linguagem de programas, pois nesta a prova indutiva é automática e segue a estrutura da definição da função, não sendo necessário a definição de princípios de indução.

Comparando as duas linguagens do Why3, a linguagem de programas é efetivamente mais interessante que a linguagem lógica.

PALAVRAS-CHAVE Algoritmos de Ordenação, Coq, Indução, Lógica, Provas Formais, Why3, WhyML.

CONTEÚDO

Anexos

LISTA DE ACRÓNIMOS

CIC *Calculation of Inductive Constructions*. 5, 6

CV Condiç$\widetilde{o}$es de Verificaç$\widetilde{a}$o. vi, 7, 9, 10, 12, 17, 18, 31, 32, 42–44, 52, 53

IDE *Integrated Development Environment*. vi, 7, 8, 10, 13, 25–27, 31, 32

SMT *Satisfiability Modulo Theories*. vi, 3, 4, 7–9, 20, 22, 26, 35, 40, 44

LISTA DE FIGURAS

LISTA DE EXCERTOS DE CÓDIGO

1

INTRODUÇÃO

1.1 CONTEXTUALIZAÇÃO

A ferramenta Why3 é uma ferramenta para verificação dedutiva de programas. Esta concede uma linguagem lógica rica, capaz de lidar com polimorfismo, tipos indutivos e construções de predicados indutivos. Esta ferramenta admite codificar muitas construções e permite que uma parte substancial das obrigações de prova possa ser descartada através da utilização de *Satisfiability Modulo Theories* (SMT) *solvers*.

A ferramenta Coq é um assistente de prova dotado de uma linguagem formal para escrever definições matemáticas, teoremas ou algoritmos executáveis. Existe um projeto chamado "Software Foundations" (Pierce et al., 2012) que tem providenciado a produção de textos que introduzem conceitos e resultados fundamentais, na área de linguagens de programação, totalmente formalizados e *machine-checked* com Coq.

1.2 MOTIVAÇÃO

As provas *machine-checked* têm um fator confiança muito superior às provas geradas por ação humana, pois estão menos sujeitas a erros, além de permitirem uma melhor gestão de provas muito longas. Sendo as ferramentas Coq e Why3, dois dos principais pilares das provas *machine-checked*, a ideia de tentar converter algumas formalizações já existentes em Coq para Why3 deve-se, sobretudo, ao facto do Why3 ser mais *user friendly* e mais fácil de utilizar devido aos SMT *solvers*.

Mesmo tendo conhecimento das limitações da linguagem do Why3, o objetivo deste trabalho consiste em perceber até onde é que esta consegue converter formalizações já existentes em Coq. Sabendo também para que o Why3 foi criado, isto é, para fazer a verificação de programas, foi escolhido, por essa razão, para uma fase inicial, o livro da "Software Foundations" que aborda exemplos de formalizações de programas em Coq (Appel, 2020).

1.3 OBJETIVOS

O principal objetivo desta dissertação é perceber até que ponto é possível utilizar a ferramenta Why3, cuja linguagem lógica é mais limitada do que a ferramenta Coq e cujos princípios são muito diferentes, para formalizar um conjunto de conceitos e resultados dos volumes do projeto "Software Foundations" (Pierce et al., 2012). É

importante referir que neste trabalho o que se pretende é tentar usar uma ferramenta menos expressiva, como o Why3, para que se possa aproveitar toda a automação que este proporciona, através do uso dos SMT *solvers*, ou seja, tentar automatizar em Why3 as formalizações que estão já desenvolvidas em Coq.

Assim, o resultado final inclui:

- um repositório que contém:
  - um conjunto de módulos Why3 correspondentes a formalizações previamente existentes em Coq;
  - provas *"replayable"* de todos os resultados sobre cada formalização, recorrendo, tanto quanto possível, a ferramentas de prova automática.
- um conjunto de orientações gerais para ajudar na conversão de formalizações em Coq para Why3.

2

ESTADO DA ARTE

Esta dissertação tem como principal foco o estudo da possibilidade da formalização de um conjunto de conceitos já existentes em Coq, cuja linguagem é extremamente potente, para Why3. Sendo estas duas ferramentas o ponto mais importante desta dissertação, a exploração mais pormenorizada sobre o funcionamento, as características e as principais limitações destas, é uma tarefa imprescindível. Ambas têm princípios muito diferentes, e, por isso é fundamental conhecer bem cada uma destas.

2.1 COQ

O Coq é uma ferramenta interativa, baseada em teoria de tipos, que funciona como assistente de prova de teoremas e programas. Permite formalizar conceitos matemáticos e ajuda a gerar, interativamente, provas de teoremas. Esta ferramenta é muito valorizada, dado que a "verificação por máquina" dá uma maior confiança comparando com as provas geradas e verificadas por ação humana (Barras et al., 2020).

Apresenta uma linguagem de programação extremamente potente, baseada em tipos dependente que conjuga uma lógica de ordem superior e uma linguagem funcional ricamente tipada, o *Calculation of Inductive Constructions* (CIC). Esta, é uma linguagem formal que tem baseada em si, uma outra linguagem matemática de nível superior utilizada no Coq denominada *Gallina*. Este sistema de prova assenta na analogia proposições-como-tipos (ou isomorfismo de Curry-Howard), onde uma **proposição A** pode ser vista como um **tipo** e uma **prova da proposição A** como um **termo de tipo A**.

Assim, **A** é demonstrável se o **tipo A** for habitado e a tarefa de verificar a prova resume-se à verificação de tipos. Ao longo do desenvolvimento da prova de uma **proposição A**, o utilizador vai conduzindo a prova aplicando táticas que correspondem à construção de um termo de prova. Quando a prova fica concluída, é realizada a verificação do tipo do termo que corresponde à **prova de A**. No Coq faz-se uma codificação direta entre a lógica e a teoria de tipos. Cada construção lógica tem a respectiva codificação na teoria de tipos em que o Coq se baseia que é o CIC.

O Coq fornece uma linguagem formal para escrever definições matemáticas, programas funcionais e teoremas, e tem um ambiente de desenvolvimento interativo. O sistema Coq tem estado em continuo desenvolvimento desde há 30 anos. É um sistema *open source*, suportado por uma biblioteca substancial e tem uma grande e ativa comunidade de utilizadores.

Conta com um sistema, *Kernel*, que garante uma verificação mecânica à prova gerada que assegura a sua validade. Este sistema garante que, caso uma tática tenha algum problema, o Coq não introduz uma prova que não esteja correta no seu sistema. As provas geradas por máquina são, assim, garantidamente corretas.

Existem três linguagens caraterísticas no Coq:

- **Gallina -** linguagem de especificação, que permite desenvolver teoremas matemáticos e provar especificações de programas;
- **Vernacular -** linguagem de comandos, que inclui todos os tipos de consultas e pedidos úteis para o sistema;
- **Itac -** linguagem específica do domínio, que permite escrever provas e táticas.

No CIC todos os objetos têm um tipo. Existem tipos para funções (ou programas), tipos atómicos (especialmente tipos de dados) e tipos para provas. Os tipos são classificados pelos três *kinds* básicos que são tipos abstratos atómicos:

- **Prop -** Proposições lógicas;
- **Set -** Coleções matemáticas;
- **Type -**Tipos abstratos.

Este assistente de prova foi já utilizado em inúmeros projetos emblemáticos de verificação. Nomeadamente, no projeto CompCert, em que se procedeu à verificação integral de um compilador para a linguagem C, e também na prova matemática do teorema das quatro cores (Gonthier, 2005), entre muitas outras formalizações matemáticas.

A forma mais fácil de demonstrar como funciona uma determinada ferramenta é através de um exemplo. Vejamos como se pode demonstrar em Coq um resultado da lógica de primeira ordem. O teorema refere uma **proposição Q** e um **predicado P** sobre o **tipo A**.

Começa-se, então, por declarar primeiro estas entidades, antes de formular como *lemma* o resultado que se pretende provar.

```
Variables (A:Set) (P : A->Prop).
Variable Q : Prop.

Lemma example : forall x:A, (Q -> Q -> P x) -> Q -> P x.
 Proof.
    intros x H H0.
    apply H.
    assumption.
    assumption.
 Qed.
```

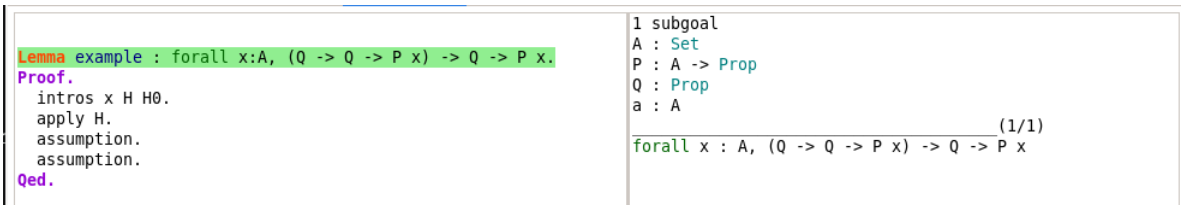

Figura 1: Apresentação inicial no IDE do Coq da prova anterior.

As provas podem ser desenvolvidas num editor de texto, ou alternativamente utilizando um *Integrated Development Environment* (IDE) específico, como ilustrado na Figura 1.

Quando uma prova está finalizada, o Coq permite a extração do programa verificado para uma linguagem de programação. Esta funcionalidade é uma mais valia, uma vez que permite usar o código Coq para criar bibliotecas de software verificadas e executá-lo com eficiência.

2.2 WHY3

O Why3 é uma ferramenta de prova lógica, que permite a verificação dedutiva de programas. Esta utiliza como linguagem lógica, uma extensão da lógica de primeira ordem, recorrendo a tipos e definições indutivas, predicativos indutivos e ao polimorfismo. Interage com variados SMT *solvers* automatizados, nomeadamente, o Z3, o Alt-Ergo, o CVC4, entre outros, além de outros demonstradores de teoremas que não encaixam nesta categoria, como o próprio Coq ou o assistente de prova Isabelle.

Conta com um conjunto de transformações de prova, que permitem fazer provas indutivas, o que seria impossível recorrendo a um SMT *solver*. Com a existência destas transformações, torna-se, por vezes, prescindível o recurso a uma ferramenta interativa (Bobot et al., 2020).

O Why3 é também uma ferramenta de verificação de programas. Esta, dispõe de uma linguagem de programação e especificação interna, **WhyML**, que engloba características funcionais e imperativas (Filliâtre and Paskevich, 2013).

A verificação de programas é baseada na Lógica de Hoare, e por isso, nos conceitos pré e pós condição, variante e invariante de ciclo. Esta ferramenta caracteriza-se por, dado um programa anotado com uma especificação, incluir um gerador de Condições de Verificação (CV), que produzem um conjunto de fórmulas lógicas (CV) cuja validade implicará que o programa é correto face à sua especificação. Estas condições são provadas recorrendo às múltiplas ferramentas com que o Why3 interage.

Um programa **WhyML** depois de provado corretamente poderá ser extraído para uma linguagem de programação real, como Ocaml ou C.

Existe a possibilidade de compatibilizar estes dois aspetos da ferramenta, pois a linguagem lógica pode ser usada no raciocínio sobre propriedades dos programas. Assim, é possível utilizar o Why3 de duas formas diferentes, que permitem efetuar provas de duas naturezas diferente:

- como uma linguagem lógica;

- como uma linguagem de programação para provar algoritmos.

Para ajudar a perceber esta ferramenta, nada melhor do que apresentar alguns exemplos. Nesta primeira abordagem, o mais adequado é um programa mais básico com um pequeno conjunto de objetivos de prova.

O Why3, tal como o Coq, dispõe de um IDE específico, que foi utilizado ao longo desta dissertação.

Task hello_proof.why

```
1 theory HelloProof
2
3   goal G1 : true
4
5   goal G2 : (true -> false) /\ (true \/ false)
6
7   use import int.Int
8
9   goal G3: forall x:int. x*x >= 0
10
11 end
12
```

Figura 2: Ficheiro **hello_proof.why**.

Este ficheiro, **hello_proof.why**, contém três *goals* diferentes, G1, G2 e G3. O intuito é provar cada um destes através de um *solver*.

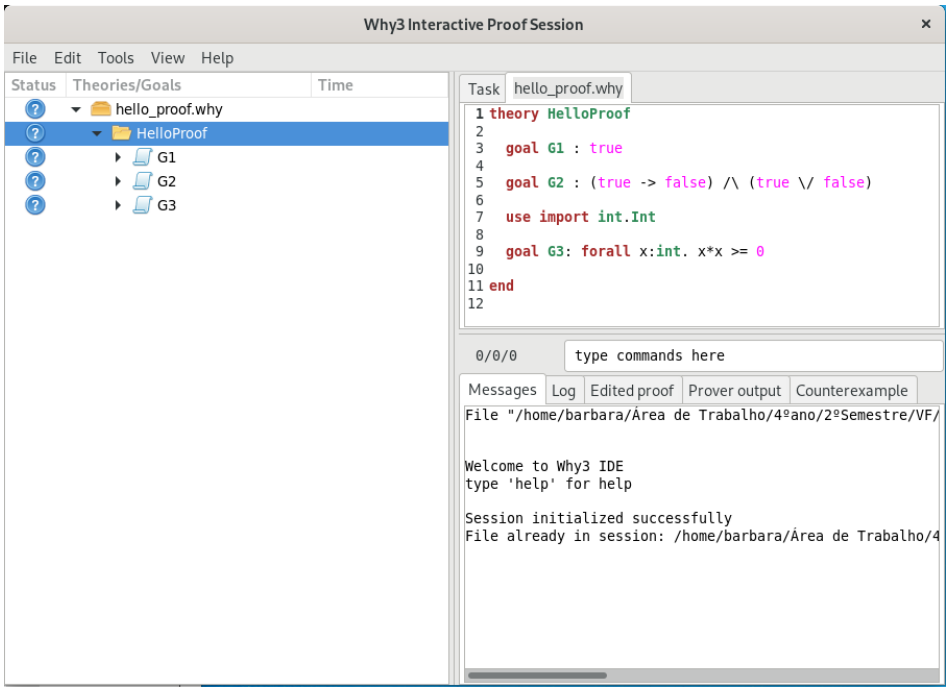

Figura 3: Apresentação no IDE do Why3 do ficheiro **hello_proof.why**.

Os *goals* G1 e G3 são provados recorrendo ao SMT *solver* **Alt-Ergo**. A prova de G2 não é concluída com sucesso, pois a primeira parte da prova é falsa.

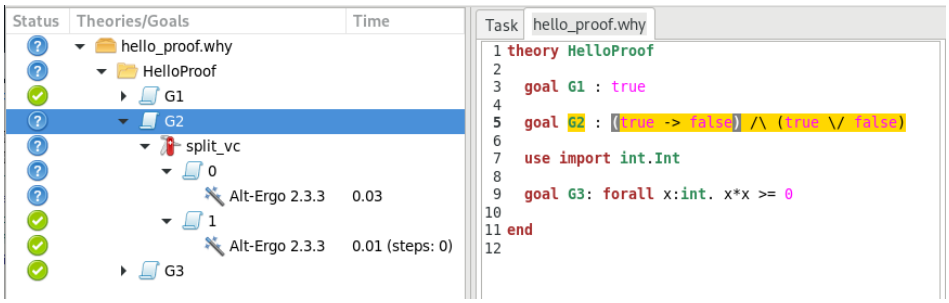

Figura 4: Resultado após correr o SMT *solver* **Alt-Ergo** em todas as provas.

Nesta prova, G2, foi escolhida a estratégia **split_vc**, que divide a prova em duas partes diferentes. Com a ajuda do **Alt-Ergo** é possível provar, apenas, a segunda parte da prova. Assim, aplicar estas estratégias, permite ver com maior facilidade que esta prova trata-se de uma conjunção, na qual, somente, uma das partes pode ser provada.

Como já foi referido anteriormente, o Why3 permite também escrever programas. Nestes programas, além do código é necessário escrever algumas anotações no programa. As pré e pós são a especificação do programa, e os invariantes são anotações fornecidas pelos utilizadores para permitir a sua prova.

As CV são fórmulas geradas pelo Why3 a partir das pré e pós condições e dos invariantes, tais que, se todas elas forem válidas, então os programas serão corretos face às suas especificações.

Para uma melhor consolidação destes conceitos, nada melhor do que um exemplo. A função ***select***, que calcula o **valor mínimo num segmento de *array***, isto é, o valor mínimo desde o índice **i** de um *array* até ao final do mesmo. Este é um programa imperativo típico do Why3 e a sua prova irá gerar as tais CV.

```
module SelectionSort

  use int.Int
  use ref.Ref
  use array.Array
  use array.IntArraySorted
  use array.ArrayPermut
  use array.ArrayEq

  let select (a: array int) (i: int)
    requires { 0 <= i < length a }
    ensures  { i <= result < length a }
    ensures  { forall k: int. i <= k < length a -> a[result] <= a[k] }
    ensures  { permut_all (old a) a }
    =
      let min = ref i in
      for j = i + 1 to length a - 1 do
        invariant { forall k: int. i <= k < j -> a[!min] <= a[k] }
        invariant { i <= !min < j }
        if a[j] < a[!min] then min := j
      done;
    !min
```

Figura 5: Programa que calcula o valor mínimo de um segmento de *array*.

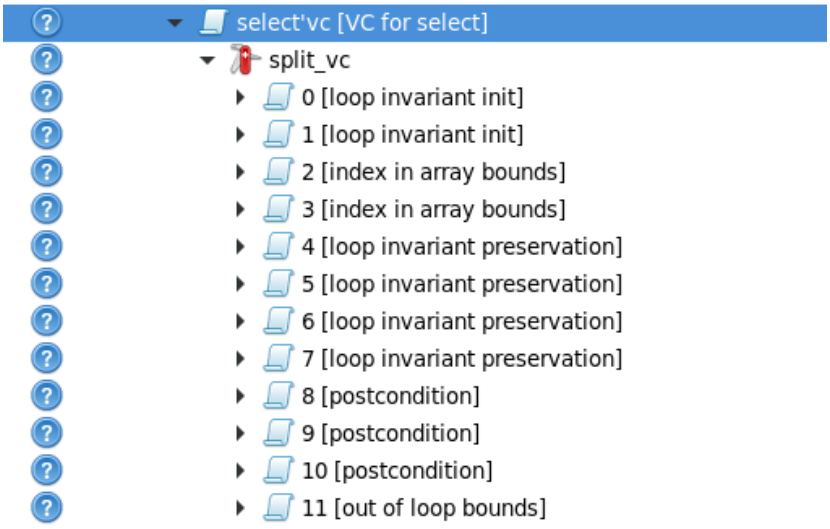

Figura 6: CV geradas após correr a estratégia **split_vc**.

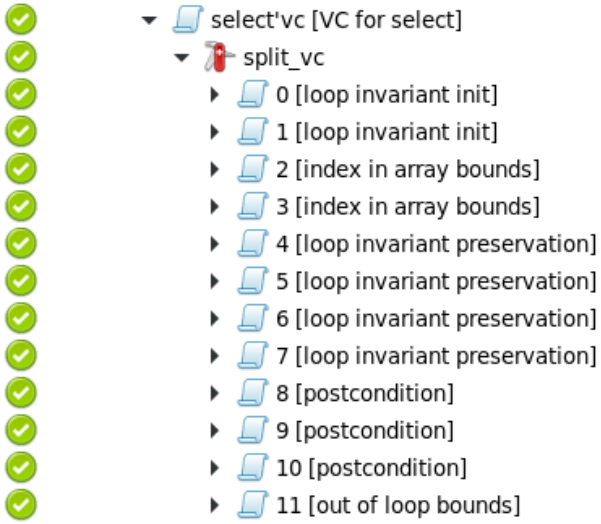

Figura 7: Resultado após correr o **Alt-Ergo** em todas as CV.

Neste exemplo, após utilizar a estratégia **split_vc** são geradas as CV necessárias para executar a prova deste algoritmo, como pode ser observado na Figura 6. Através do **Alt-Ergo** foi possível provar cada uma das CV e por consequência a prova do programa fica completa. Na figura 7 está representado o estado do IDE, neste ponto, sendo indicada para cada uma das condições, a cor verde, remetendo para o sucesso da sua prova.

Esta ferramenta torna-se mais versátil e vantajosa devido às diferentes transformações que é possível fazer. A capacidade de lidar com indução equipara o Why3 com ferramentas mais interativas. Duas das transfor-

mações mais importantes e que permitem realizar inúmeras provas baseadas na indução são **induction_pr** e **induction_ty_lex** (Bobot et al., 2020).

Esta última recorre à indução estrutural, ou seja, gera hipóteses de indução sobre variáveis universalmente quantificadas de tipos de dados algébricos, tais como, listas e árvores. Por exemplo, a prova apresentada abaixo ficaria transformada da seguinte forma:

```
theory Lists

  use import list.List
  use import int.Int

  type list 'a = Nil | Cons 'a (list 'a)

goal app_nil : forall l:list 'a. (l ++ (Nil: list 'a)) = l

goal app_nil :
  forall l:list 'a.
    match l with
    | Nil -> (l ++ (Nil: list 'a)) = l
    | Cons a l1 -> (l1 ++ Nil: list 'a))
                = l1 -> (l ++ (Nil: list 'a)) = l
    end
```

Listing 2.1: Código Why3 do *goal* **app_nil** onde é utilizada a transformação **induction_ty_lex**.

Quando a indução pode ser aplicada a várias variáveis, a transformação escolhe uma delas heuristicamente.

A transformação **induction_pr** funciona de forma parecida à anterior com a diferença que, esta indução aplica-se sobre predicativos indutivos e não sobre tipos algébricos, quando um predicado indutivo ocorre no antecedente da fórmula que se pretende provar. Para uma melhor consolidação, abaixo mostra-se um exemplo do resultado de aplicar esta transformação a um *goal*:

```
theory Lists

    use import list.List
    use import int.Int

    type list 'a = Nil | Cons 'a (list 'a)

inductive last 'a (list 'a) =
    | Last_one : forall x :'a. last x (Cons x Nil)
    | Last_two : forall x y :'a, l :list 'a. last x l -> last x (Cons y l)

function lastf (l :list int) : int =
  match l with
  | Nil        -> 0
```

```
  | Cons x Nil -> x
  | Cons x t   -> lastf t
  end

goal lastf2 : forall x:int, l:list int. last x l -> x = lastf l

goal lastf2 :
  forall x:int.
   forall x1:int.
    forall l:list int.
        l = Cons x1 (Nil: list int) -> x = x1 -> x = lastf l

goal lastf2 :
  forall x:int.
   forall x1:int, y:int, l:list int.
    last x1 l /\ (forall l1:list int.
        l1 = l -> x = x1 -> x = lastf l1) ->
            (forall l1:list int.
                l1 = Cons y l -> x = x1 -> x = lastf l1)
```

Listing 2.2: Código Why3 do *goal* **lastf2** onde é utilizada a transformação **induction_pr**.

Neste exemplo, o *goal* a ser provado é o **lastf2**. Ao aplicar a transformação **induction_pr** resultam dois *goals* diferentes. Cada um deles é provado através do ***solver* Alt-Ergo**.

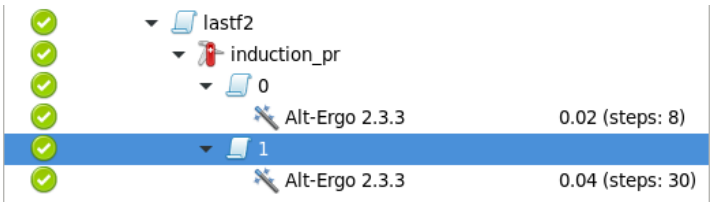

Figura 8: Resultado após executar a transformação **induction_pr** e, seguidamente, correr o **Alt-Ergo**.

Quando uma prova está completa, o Why3 permite fazer a reprodução dessa prova através da funcionalidade *replay*, isto é, permite "exportar" todas as técnicas usadas ao efetuar a prova de um determinado *goal* (ver Figura 9). Reprodução essa que é do género de um *script*, o que permite até partilhar a "solução", ou seja, quando alguém partilha uma formalização já provada, esta possui as transformações que foram usadas dentro da ferramenta, e, também, a identificação do *prover* com que se provou cada CV, o que evita a repetição de todo esse esforço. Esta caraterística torna a ferramenta Why3 bastante mais útil e dinâmica, quando comparada com outras ferramentas orientadas para a prova automática, e mais equiparada com os assistentes de provas mais interativos, como, por exemplo, o Coq.

```
[barbara@localhost theoryLists]$ why3 replay Lists/
File "/home/barbara/Área de Trabalho/Tese/Exercicios/theoryLists/Lists.why", line 75, characters 11-12: unused variable x
File "/home/barbara/Área de Trabalho/Tese/Exercicios/theoryLists/Lists.why", line 82, characters 11-12: unused variable x
 12/12 (replay OK)
```

Figura 9: Exemplo do uso do *replay* numa prova.

O IDE é uma ferramenta imprescindível no desenvolvimento das provas, uma vez que torna o Why3 bastante mais intuitivo e ajuda a ter uma visão geral, à medida que a prova vai avançando do que vai sendo provado, além de permitir gravar o estado de todo o desenvolvimento.

No entanto, o IDE nem sempre é fundamental, por exemplo, uma funcionalidade muito útil do Why3 é o *smoke detector* que permite fazer a verificação do código e detetar se o contexto apresenta algum tipo de contradição. Neste caso, tal como no *replay*, o IDE não é particularmente essencial, visto que se trata de uma operação efetuada depois de provadas todas as condições (Bobot et al., 2020).

```
[barbara@localhost theoryLists]$ why3 replay --smoke-detector top Lists/
File "/home/barbara/Área de Trabalho/Tese/Exercicios/theoryLists/Lists.why", line 75, characters 11-12: unused variable x
File "/home/barbara/Área de Trabalho/Tese/Exercicios/theoryLists/Lists.why", line 82, characters 11-12: unused variable x
 12/12 No smoke detected
```

Figura 10: Exemplo do uso do *smoke detector* numa prova.

Existem três configurações distintas do *smoke detector*:

- **none**: serve para não correr o *smoke detector*, ou seja, é útil quando é obrigatório correr o *smoke detector* mas não é necessário;
- **top**: acrescenta a negação ao *goal* original e tenta fazer a prova. Caso seja "acionado" significa que existe um erro na formalização;

  Por exemplo, aplicando o *smoke detector* ao seguinte *goal*:

  ```
  Goal G : forall x:int. q x -> (p1 x \/ p2 x)
  ```

  O que a opção top executa é:

  ```
  Goal G : ~ (forall x:int. q x -> (p1 x \/ p2 x))
  ```

  Posto isto, se este mecanismo não correr com sucesso significa que existem contradições no contexto da prova.

- ***deep***: esta opção acrescenta também a negação, mas mantém a quantificação e faz a negação internamente. Verifica se existem contradições entre o antecedente e o contexto. Ou seja, em termos práticos, se o *smoke detector* for "acionado" significa que o erro está no objetivo de prova, o que significa que o *goal* que se pretende provar não é interessante, pois nunca se realiza.

Aplicando ao exemplo referido anteriormente, o que o *deep* executa é:

```
Goal G : forall x:int. q x /\ ~ (p1 x \/ p2 x)
```

Apesar destas inúmeras capacidades e de estar cada vez mais desenvolvido, o Why3 continua, mesmo assim, a ser mais limitado do que o Coq em variados aspetos. Nomeadamente, o Coq permite quantificar sobre funções lógicas ou predicados, admite formalizações envolvendo tipos dependentes e inúmeros predicados que não conseguimos definir em Why3. Mas, é, também, importante realçar que as limitações do Why3 são ao nível da linguagem e não da ferramenta em si. Faz parte do *design* do Why3 já que, este tenta estar num *sweet spot*, isto é, ter uma linguagem suficientemente expressiva mas sem deixar de ser focado na prova automática.

Se tentarmos, por exemplo, definir em Why3 um tipo para *arrays* com um determinado tamanho **x**, o Why3 não consegue, pois o tamanho do *array* vai depender sempre de um valor e este não permite a formalização de tipos dependentes.

Outro exemplo de mais uma limitação do Why3 pode ser uma formalização de um predicado envolvendo lógica de segunda ordem, por exemplo:

$$\forall P \, \forall x (Px \vee \neg Px)$$

Em suma, o Why3 é, ainda, uma ferramenta bastante mais limitada do que o Coq, apesar de já ser capaz de realizar até provas indutivas. Assim, o principal objetivo é, então, perceber até que ponto consegue esta ferramenta acompanhar o Coq, e que formalizações é possível converter para a sua linguagem.

2.3 EXEMPLO DE UMA FORMALIZAÇÃO EM COQ E WHY3

O principal objetivo desta dissertação é selecionar algumas formalizações já existentes em Coq e tentar converter as mesmas para Why3. Nomeadamente, algumas do projeto “Software Foundations”. Este é constituído por cinco livros: (Pierce et al., 2018b); (Pierce et al., 2018a); (Appel, 2020); (Lampropoulos and Pierce, 2018) e (Andrew W. Appel, 2020) cujo conteúdo incide numa introdução aos fundamentos matemáticos da Ciência da Computação. Destacam-se pela particularidade de serem 100% formalizados e *machine-checked*. Cada um dos volumes inclui *scripts* de prova para o assistente Coq, exercícios e textos explicativos.

No entanto, existem já alguns exemplos de algoritmos formalizados nos dois sistemas de prova. Nesta secção foi analisado um destes exemplos, antes de nos dedicarmos, no capitulo seguinte, ao desenvolvimento das nossas formalizações em Why3.

2.3.1 *Provas formais do Algoritmo de Tarjan*

O propósito do algoritmo de *Tarjan* é possível encontrar os componentes fortemente ligados de um grafo orientado (Chen et al., 2018). Fazendo uma prova formal deste algoritmo em Why3 e em Coq, é possível

ter uma ideia das principais diferenças, vantagens e desenvolvimentos destes dois assistentes de prova. O algoritmo é definido, em ambos os sistemas, no mesmo nível de abstração e a prova baseia-se nos mesmos argumentos.

2.3.2 *Algoritmo*

O algoritmo de *Tarjan* executa uma procura em profundidade num conjunto de vértices de um grafo. Cada um dos vértices é visitado apenas uma vez, sendo-lhe atribuído um número de série da sua visita. O algoritmo guarda um ambiente **e** constituído por quatro campos:

- uma stack que guarda os vértices visitados que não fazem parte dos componentes fortemente ligados, **e.stack**;
- um conjunto composto pelos componentes fortemente ligados, **e.sccs**;
- um novo número de série, **e.sn**;
- uma função que faz o registo dos números de série atribuídos a cada vértice, **e.num**.

A organização da pesquisa em profundidade é baseada em duas funções recursivas, **dfs1** e **dfs**. Esta segunda recebe como argumentos, um conjunto **r** de raízes e um ambiente **e**, e retorna um par composto por um número inteiro e o ambiente modificado.

Quando o **r** é vazio, o inteiro retornado é $+\infty$. Caso contrário, o inteiro retornado é o mínimo dos resultados das chamadas à função **dfs1** em vértices não visitados em **r** e dos números de série dos que já foram visitados. O resultado é o conjunto de componentes retornados pela função **dfs** chamada em todos os vértices do grafo. A principal função deste algoritmo é a **dfs1** que "visita" um novo vértice **x**.

```
let rec dfs1 x e =
    let n0 = e.sn in
    let (n1, e1) = dfs (successors x)
        (add_stack_incr x e) in
    if n1 < n0 then (n1, e1) else
        let (s2, s3) = split x e1.stack in
        (+∞, {stack = s3;
            sccs = add (elements s2) e1.sccs;
            sn = e1.sn; num = set infty s2 e1.num})
with dfs r e = if is_empty r then (+∞, e) else
    let x = choose r in
    let r' = remove x r in
    let (n1, e1) = if e.num[x] 6 = -1
        then (e.num[x], e) else dfs1 x e in
```

```
        let (n2, e2) = dfs r' e1 in (min n1 n2, e2)

    let tarjan () =
        let e = {stack = Nil; sccs = empty;
            sn = 0; num = const (-1)} in
        let (_, e') = dfs vertices e in e'.sccs
```

A função auxiliar **add_stack_incr** atualiza os valores do ambiente acrescentando o valor **x** na stack, atribuindo-lhe o novo número de série atual e incrementando esse número para chamadas no futuro. A função **dfs1** executa uma chamada recursiva para **dsf** para os vértices sucessores que recebem como parâmetros o valor **x** como raízes e o ambiente atualizado.

Se o valor retornado do inteiro **n1** for menor do que o valor atribuído a **ax**, logo a função retorna **n1** e o ambiente atual. Caso contrário, a função declara que foi encontrado um novo componente fortemente ligado, consistindo em todos os vértices contidos no topo de **x** na *stack* atual. Assim, a *stack* é esvaziada *(popped)* até **x** e os vértices extraídos são armazenados como um novo **e.sccs** e os seus números estão configurados para $+\infty$, para que, assim, eles não interfiram com cálculos futuros de valores mínimos.

As funções auxiliares **set infty** e **split** são utilizadas para realizar estas atualizações.

```
    let add_stack_incr x e = let n = e.sn in
        {stack = Cons x e.stack; sccs = e.sccs;
            sn = n+1; num = e.num[x ← n]}

    let rec set infty s f = match s with Nil → f
        | Cons x s' → (set infty s' f)[x ← +∞] end

    let rec split x s = match s with Nil → (Nil, Nil)
        | Cons y s' → if x = y then (Cons x Nil, s')
            else let (s1', s2) = split x s' in
                (Cons y s1', s2) end
```

2.3.3 *Formalização em Why3*

Em Why3, a formalização do algoritmo de *Tarjan* é apresentada recorrendo às estruturas de dados disponíveis no mesmo. Assim, para listas, existem os construtores **Nil** e **Cons**. Para conjuntos finitos, existe o conjunto vazio *empty* e a função **add**, para adicionar um elemento a um conjunto, a função **remove**, para remover um elemento e a função **choose** para escolher um número arbitrário num conjunto (não vazio) e **is_empty** para testar se um conjunto é vazio (Chen et al., 2018).

Também foi utilizado *maps* com funções *const* para a função constante, *[ ]* para aceder o valor de um elemento e *[<-]* para criar um *map* obtido de um mapa existente. Foi definido um tipo abstrato *vertex* para vértices e a

constante *vertices* para o conjunto finito de todos os vértices do grafo. O ambiente foi definido recorrendo ao tipo *env* que se resume ao registo dos quatros campos, *stack, sccs, sn* e *num*.

Assim, a prova deste algoritmo, como é comum em algoritmos de procura em profundidade, vai envolver três conjuntos de vértices:

- brancos - vértices não visitados;
- pretos - vértices totalmente visitados;
- cinza - vértices que estão a ser visitados.

Algumas propriedades invariantes a serem mantidas podem ser expressas, para os vértices na *stack*, da seguinte forma:

- nenhum vértice é branco;
- qualquer vértice pode atingir todos os vértices mais altos da *stack*;
- qualquer vértice pode alcançar algum vértice cinza inferior na *stack*.

As funções **dfs1** e **dfs** aumentam o conjunto dos vértices totalmente visitados (pretos), mantêm invariante os vértices que estão a ser visitados (cinza), aumentam a *stack* com novos vértices pretos, descobrem novos componentes fortemente ligados e mantêm invariante o número de série dos vértices na *stack*.

O Why3, como referido anteriormente, gera CV. A função abaixo é anotada com contratos que especificam o seu comportamento. Após expressos os invariantes, é necessário adicionar algumas afirmações no corpo da função **dfs1** para ajudar a provar o programa.

```
let rec dfs1 x e =
    (* pre-condition *)
    requires {mem x vertices}
    requires {∀ y. mem y e.gray → reachable y x}
    requires {not mem x (union e.black e.gray)}
    requires {wf_env e} (* I *)
    (* post-condition *)
    returns {(_, e') → wf_env e' ∧ subenv e e'}
    returns {(_, e') → mem x e'.black}
    returns {(n, e') → n ≤ e'.num[x]}
    returns {(n, e') → n = +∞ ∨ num_of_reachable n x e'}
    returns {(n, e') → ∀ y. xedge_to e'.stack e.stack y
                    → n ≤ e'.num[y]}
```

```
let n0 = e.sn in
let (n1, e1) =
    dfs (successors x) (add_stack_incr x e) in
if n1 < n0 then begin
    assert {∃ y. y 6 = x ∧ precedes x y e1.stack ∧
        reachable x y};
    assert {∃ y. y 6 = x ∧ mem y e1.gray ∧
        e1.num[y] < e1.num[x] ∧ in_same_scc x y};
    (n1, add_black x e1) end
else
    let (s2, s3) = split x e1.stack in
        assert {is_last x s2 ∧ s3 = e.stack ∧
            subset (elements s2) (add x e1.black)};
        assert {is_subscc (elements s2)};
        assert {∀ y. in_same_scc y x → lmem y s2};
        assert {is_scc (elements s2)};
        assert {inter e.gray (elements s2) == empty};
        (+∞, {black = add x e1.black; gray = e.gray;
            stack = s3; sccs = add (elements s2) e1.sccs;
            sn = e1.sn; num = set infty s2 e1.num})
```

De acordo com os autores do artigo, todas as CV foram provadas pelos *solvers* automáticos, exceto duas. A primeira prova é a da extensão preta da pilha no caso **n1 < n0**, uma vez que os *solvers* não conseguiram lidar com o quantificador existencial, já o Coq permite e é até bastante curta a sua prova.

A segunda prova é a afirmação no corpo que **dfs1**, que afirma que qualquer **y** no componente ligado de **x** pertence a **s2**. A prova desta afirmação é feita por contradição, ou seja, se **y** não estiver em **s2**, deve haver uma aresta de **x'** em **s2** para algum **y'** que não esteja em **s2**, de modo a que **x** alcance **x'** e **y'** alcance **y**. Esta prova também é conseguida apenas usando o Coq. Todos os *lemmas* utilizados na prova são provados através dos *solvers* automáticos, usando estratégias de indução, *splitting* e *inlining*.

2.3.4 *Formalização em Coq*

Em Coq a formalização do algoritmo de *Tarjan* segue o mesmo raciocínio que em Why3, sendo a principal diferença que este é parametrizado por um tipo finito **V** para os vértices e uma função *successors* que representa o grafo (Chen et al., 2018). O ambiente referenciado no algoritmo é definido com o registo de cinco campos:

- black;
- stack;

- esccs;
- sn;
- num.

Nesta definição não existe o mecanismo para o campo preto e não existem os vértices cinza. Aqui, eles são definidos globalmente como os elementos da *stack* que não são pretos.

A principal diferença é a forma como a recursividade é tratada. A função **dfs1** trata um vértice **xe** e a função **dfs** trata um conjunto de "vértices-raiz" num ambiente **e**. Então, estas duas funções são utilizadas na função recursiva **tarjan_rec**, que recebe o parâmetro **n**, que controla a altura recursiva máxima. Os invariantes são os mesmos que da prova em Why3, apenas são "agrupados" de forma diferente. O principal invariante que reúne todas as propriedades:

```
Record invariants (e : env) := Invariants {
    inv_wf_color : wf_color e;
      inv_wf_num : wf_num e;
    inv_wf_graph : wf_graph e;
wf_noblack_towhite : noblack_to_white e;
        inv_sccs : sccs e = black_gsccs e;
}.
```

As pré condições são expressas de forma idêntica às que foram definidas em Why3: todos os vértices cinza de **e** estão ligados a todos os elementos das **roots** e todos os invariantes se mantêm.

Como no Coq é possível utilizar a expressividade dos grandes operadores (Bertot et al., 2008), as pós condições são expressas de forma um pouco diferente. Assim, estas afirmam que: os invariantes se mantêm, o próximo ambiente é uma extensão do anterior, os novos vértices brancos foram decrementados pelos vértices que são acessíveis a partir das raízes por vértices brancos e o valor retornado **m** é o menor número de todos os vértices que perderam a sua cor branca.

```
Record post_dfs (roots : {set V}) (e e' : env) (m : nat)
  := PostDfs {
    post_invariants : invariants e';
    post_subenv     : subenv e e';
    post_whites     :
      whites e' = white e :\: \bigcup_(x in roots) wreach e x;
    post_num        :
      m = \min_(y in \bigcup_(x in roots) wreach e x)
         @inord ∞ (num e' y);
}.
```

Os dois teoremas para provar são, então:

```
Lemma dfs_is_correct dfs1 dfsrec (roots : {set V}) e :
    (∀ x, x ∈ roots -> dfs1_correct dfs1 x e) ->
    (∀ x, x ∈ roots -> ∀ e1, white e1 \subset white e ->
        dfs_correct dfsrec (roots :\ x) e1) ->
    dfs_correct (dfs dfs1 dfsrec) roots e.

Lemma dfs1_is_correct dfs (x : V) e :
    (dfs_correct dfs [set y | edge x y] (add_stack x e)) ->
    dfs1_correct (dfs1 dfs) x e.
```

A prova do primeiro *lemma* é direta, uma vez que o **dfs** simplesmente itera numa lista. Já a prova deste segundo *lemma* é um pouco mais complexa. Caos se junte os dois teoremas e tentar provar esta junção obtém-se uma prova menos complexa.

```
Theorem tarjan_rec_terminates n roots e :
    n ≥ #|white e| * #|V|.+1 + #|roots| ->
    dfs_correct (tarjan_rec n) roots e.
```

Deste último teorema, a prova é também direta e simples. A maioria das provas são de uma linha e a prova mais complicada é a que corresponde ao *lemma* **dfs1_is_correct**.

2.3.5 *Conclusão*

A grande vantagem do Why3 é a separação entre programas e a lógica com as asserções da lógica de *Hoare*, pré e pós condições, o que torna a prova bastante mais legível. Além disso, o facto de a linguagem lógica estar restrita à lógica de primeira ordem ajuda, pois é fácil de entender. Outra característica importante do Why3 é a sua interface com os *provers* automáticos de teoremas (*provers* SMT), já que torna as provas mais curtas e mais fáceis de realizar e entender.

A prova em Coq foi inspirada na prova em Why3, e por isso o seu desenvolvimento necessitou de menos tempo, embora o seu texto seja mais longo. Na prova Coq foram utilizadas inúmeras bibliotecas de componentes matemáticos e recorreu-se também à ordem superior, o que torna a prova mais abstrata e mais próxima da prova original de *Tarjan*.

Em suma, cada um dos dois sistemas tem as suas próprias vantagens. A parte mais complicada, em qualquer um dos dois sistemas é encontrar invariantes que sejam fortes o suficiente e compreensíveis, embora os assistentes de prova forneçam alguma ajuda.

Este artigo Chen et al. (2018), além da formalização do programa, foca-se, também, no tempo de execução e no número de linhas da prova, deste algoritmo, em cada um dos sistemas. Apesar de, neste trabalho, não serem estes os fatores a que seja dada particular atenção, continua a ser interessante fazer referência a este

artigo. Nesta dissertação, o foco é a conversão ou a tentativa de conversão das formalizações já existentes em Coq para Why3.

No entanto, este artigo é mencionado, uma vez que faz referência à formalização do algoritmo em diferentes sistemas de prova, apesar de, apenas ser importante dois deles para este estudo. Além disso, este também compara algumas das características e limitações de cada um dos diferentes sistemas de prova.

3

FORMALIZAÇÕES EM WHY3

Como destacado anteriormente, o principal objetivo desta dissertação consiste na conversão direta das formalizações existentes no livro da "Software Foundations" em Coq para um versão Why3, bem como a prova da sua correção. Queremos com isto dizer que as definições dos diferentes algoritmos serão as que resultam de converter as versões do livro para uma das linguagens do Why3. Pretende-se averiguar até que ponto pode a estrutura de cada uma das provas utilizada no livro (i.e. a sequência de *lemmas* necessários) ser mantida em Why3. Numa primeira fase, utilizando apenas a linguagem lógica que este disponibiliza, numa fase seguinte usando a particular linguagem de programas, **WhyML**. Posteriormente, estas duas versões também são comparadas.

O primeiro exemplo apresentado corresponde ao algoritmo **insertion sort**.

3.1 INSERTION SORT

Este algoritmo **insertion sort** é, habitualmente, conhecido como um programa imperativo operando sobre *arrays*. Contudo, pode ser apresentado como um programa funcional operando sobre listas. É sobre esta última versão que incidirá o trabalho.

A definição deste programa **insertion sort** escrito na linguagem formal do Coq inclui a função **insert** e a função **sort**, onde a primeira é uma função auxiliar da segunda. Para fazer a tradução destas funções, da linguagem utilizada em Coq para Why3, não é necessário alterar grande parte do corpo das funções. As principais diferenças entre estas definições são:

- a palavra que introduz a definição de uma função;
- a sintaxe utilizada para a representação de listas;
- os tipos da função, dado que, no Why3 não existe o tipo indutivo **nat**, e por isso utiliza-se o tipo **int**.

Contrariamente ao que acontece em Coq, em Why3 não existe uma teoria de naturais. Era possível, através da indução, definir os naturais, porém não traria grande vantagem, uma vez que o objetivo é trabalhar com os SMT *solvers* e a biblioteca SMT **lib** também não conta com esta teoria.

Existem também outros pormenores característicos do Why3, como por exemplo: a necessidade de indicar o tipo do *output* da função, ao contrário do que acontece em Coq. E a necessidade de se colocar no final de uma definição em Coq um "**.**", algo que em Why3 não se verifica.

Nesta primeira conversão, recorrendo apenas ao nível lógico (nível do Why3 mais próximo do Coq), pode verificar-se todos estes aspetos referidos, no código abaixo apresentado:

```
Fixpoint insert (i : nat) (l : list nat) :=
  match l with
  | [] => [i]
  | h :: t => if i <=? h then i :: h :: t else h :: insert i t
  end.
```

```
function insert (i :int) (l :list int) : list int =
  match l with
  | Nil -> Cons i Nil
  | Cons h t -> if i <= h then Cons i (Cons h t)
                          else Cons h (insert i t)
  end
```

Listing 3.1: Código Why3 da função **insert**.

```
Fixpoint sort (l : list nat) : list nat :=
  match l with
  | [] ⇒ []
  | h :: t ⇒ insert h (sort t)
  end.
```

```
function sort (l :list int) : list int =
  match l with
  | Nil  -> Nil
  | Cons h t -> insert h (sort t)
  end
```

Listing 3.2: Código Why3 da função **sort**.

A função **insert** é uma função que, dado um número **i** e uma lista **l**, insere o número ordenadamente na lista. A função **sort** utiliza a anterior recursivamente, por forma a ordenar a lista que recebe como argumento. Estas são as definições do algoritmo propriamente dito.

O predicado **sorted** verifica se os elementos de uma determinada lista se encontram de forma ordenada. Na formalização em Coq, este é escrito de três formas distintas. Não seria necessário definir todas as opções, mas por uma questão de completude optou-se por apresentar cada uma delas.

3.1.1 *Formalização da função Sorted*

A primeira definição é recorrendo ao seguinte predicado indutivo:

```
Inductive sorted : list nat → Prop :=
  | sorted_nil  : sorted []
  | sorted_1    : ∀ x, sorted [x]
  | sorted_cons : ∀ x y l, x ≤ y → sorted (y :: l)
                → sorted (x :: y :: l).
```

Ou seja:

- a lista vazia está ordenada;
- qualquer lista composta apenas por um elemento está ordenada;
- para quaisquer dois elementos adjacentes, estes devem estar na ordem correta.

A tradução desta definição para Why3 apresenta-se abaixo.

```
inductive sorted (list int) =
  | sorted_nil  : sorted Nil
  | sorted_1    : forall x :int. sorted (Cons x Nil)
  | sorted_cons : forall x y :int, l :list int.
      x <= y -> sorted (Cons y l) -> sorted (Cons x (Cons y l))
```

Listing 3.3: Código Why3 do predicado indutivo **sorted**.

As principais diferenças nesta definição, entre as duas linguagens, são, de uma forma geral, da mesma natureza que as que foram referidas nas funções anteriores.

Recorrendo à função **sorted**, especificamos o que significa ser um algoritmo de ordenação correto, em Coq:

```
Definition is_a_sorting_algorithm (f: list nat → list nat) :=
    ∀ al, Permutation al (f al) ∧ sorted (f al).
```

Numa primeira instância, o predicado **is_a_sorting_algorithm** não fora definido, uma vez que não era conhecida esta funcionalidade do Why3 admitir a ordem superior. Esta foi uma das descobertas mais interessantes sobre a linguagem a nível lógico do Why3, visto que revela ser uma linguagem bastante mais versátil e completa do que era expectável. Nesta ferramenta, o predicado define-se, portanto, da seguinte forma:

```
predicate is_a_sorting_algorithm (f: list int -> list int) =
  forall al :list int. permut al (f al) /\ sorted (f al)
```

Listing 3.4: Código Why3 do predicado **is_a_sorting_algorithm**.

O predicado **permut** aplicado na definição anterior pertence ao módulo **Permut** existente na biblioteca *standard* do Why3. Este predicado também poderia ser definido de raiz como um predicado indutivo.

Concluída a definição do programa **insertion sort**, a tarefa seguinte consiste na realização da prova da correção do algoritmo.

Os resultados lógicos são *lemmas* ou objetivos de prova (*goasl*). Os *lemmas* são provados e, posteriormente, são introduzidos no contexto para a prova dos resultados seguintes (outros *lemmas* ou objetivos de prova). Do ponto de vista do IDE, tanto os *lemmas* como os *goals* vão aparecer como obrigações de prova.

É necessário escrever então os seguintes *lemmas*:

- **insert_sorted** - serve para garantir que a lista retornada como resultado da função **select** continua ordenada;
- **sort_sorted** - prova que o resultado produzido pela função **sort** é também uma lista ordenada;
- **insert_perm** - serve para garantir que a lista retornada como resultado da função **select** é uma permutação da lista inicial;
- **sort_perm** - prova que o *output* da função **sort** é uma permutação do seu *input*;
- **insertion_sort_correct** - finaliza a prova da correção.

Foi também definido o *lemma* **insertion_sort_correct2**, que corresponde à mesma definição do anterior, porém sem recorrer à ordem superior.

Em Coq definem-se desta forma:

```
Lemma insert_sorted:
  ∀ a l, sorted l → sorted (insert a l).

Theorem sort_sorted: ∀ l, sorted (sort l).

Lemma insert_perm: ∀ x l,
  Permutation (x :: l) (insert x l).

Theorem sort_perm: ∀ l, Permutation l (sort l).

Theorem insertion_sort_correct:
  is_a_sorting_algorithm sort.
```

Fazendo a tradução dos mesmos para Why3:

```
lemma insert_sorted: forall a :int, l :list int.
                       sorted l -> sorted (insert a l)

lemma sort_sorted: forall l :list int.
                     sorted (sort l)

lemma insert_perm: forall x :int, l :list int.
                     permut (Cons x l) (insert x l)

lemma sort_perm: forall l :list int.
                   permut l (sort l)

goal insertion_sort_correct: is_a_sorting_algorithm sort
```

Listing 3.5: Código Why3 dos *lemmas* das funções **insert** e **sort**.

Para fazer a prova do *lemma* **insert_sorted** foi necessário recorrer primeiro à transformação **induction_ty_lex**. Em seguida, os SMT *solvers* já foram capazes de resolver a restante prova.

Na prova do *lemma* **sort_sorted** foi também utilizada a transformação **induction_ty_lex** e com a ajuda do **CVC4** foi possível provar o resultado pós transformação.

Foi preciso aplicar a transformação **induction_ty_lex** ao *lemma* **insert_perm** antes da utilização do SMT *solver* **CVC4** para a finalização da prova.

Para provar o *lemma* **sort_perm** foi necessário recorrer, em primeiro, à transformação **induction_ty_lex**. Posteriormente, os SMT *solvers* foram capazes de resolver a restante prova.

Na prova do *lemma* **insertion_sort_correct** apenas foi necessário utilizar o *solver* **CVC4** para efetuar a prova.

Concluídas as provas de todos os *lemmas*, o programa fica automaticamente provado. No IDE do Why3, quando uma prova está totalmente concluída este é o cenário com que nos deparamos:

Figura 11: Apresentação no IDE do Why3 da prova completa do algoritmo **insertion sort** na lógica.

3.1.2 *Formalização da função Sorted'*

Uma outra forma de escrever a função **sorted** é utilizando uma definição um pouco mais familiar: para quaisquer dois elementos da lista, estes devem estar na ordem correta. Assim sendo, para quaisquer índices válidos **i** e **j** numa lista **lst**, se **i < j**, então **lst i <= lst j**, onde o índice **lst n** significa o elemento de **lst** no índice **n**.

A formalização desta ideia em Coq revela-se uma tarefa complexa, pois mesmo quando o índice está fora do intervalo da lista, algum resultado deve ser retornado. Deste modo, a formalização da função **sorted** em Coq utiliza uma função existente na sua biblioteca *standard*:

```
Check nth_error : ∀ A : Type, list A → nat → option A.
```

Esta função garante que um resultado é sempre retornado. **Nth_error** retorna um argumento do tipo **option A**, ou seja, retorna **Some v** caso o índice esteja dentro do intervalo e **None** caso contrário. Em Coq quando se utiliza a função **nth_error** para definir a *sorted* é necessário adicionar antecedentes:

```
Definition sorted' (al : list nat) := ∀ i j iv jv,
    i < j →
    nth_error al i = Some iv →
    nth_error al j = Some jv →
    iv ≤ jv.
```

Para a formalização desta função **nth_error** em Why3, existem duas alternativas: define-se de raiz ou opta-se pela opção já existente na biblioteca *standard*. Em Why3 esta função **nth_error** já existe na biblioteca *standard* como **nth** pertencente ao módulo **Nth**. Após alguma reflexão, optou-se pela seguinte definição da biblioteca do Why3:

```
module Nth

let rec function nth (n: int) (l: list 'a) : option 'a =
    match l with
    | Nil      -> None
    | Cons x r -> if n = 0 then Some x else nth (n - 1) r
    end

end
```

Listing 3.6: Código Why3 da função **nth** definida na biblioteca.

Esta função ***nth*** tem a particularidade interessante de ser definida como ***let rec function***, isto é, pertence ao nível de programas do Why3, no entanto, como é uma função pura, pode ser utilizada nos dois níveis, lógico e de programas.

A função **nth** é equivalente à função **nth_error** existente em Coq, na medida em que esta também recebe uma lista e um inteiro, como argumentos, e retorna um argumento do tipo **option**. Posto isto, a função **sorted'** é definida recorrendo à função **nth** do módulo **Nth** existente na biblioteca do Why3:

```
predicate sorted' (al: list int) =
    forall i j iv jv :int.
    i < j ->
    nth i al = Some iv ->
    nth j al = Some jv ->
    iv <= jv
```

Listing 3.7: Código Why3 do predicado **sorted'**.

Analisando as principais diferenças entre as duas linguagens, é importante destacar que em Why3 não existe a *keyword* **Definition**. Nesta linguagem existem as definições **predicate**, que são funções cujo resultado é do tipo *Booleano* e são, por isso, adequadas para a tradução das **Definition** apresentadas em Coq.

Como não se prova a especificação do programa, apenas a implementação, a melhor forma de validar a sua especificação, passa por escrever duas especificações diferentes e provar que estas são equivalentes. Estes são os dois *lemmas* que garantem esta propriedade:

```
lemma sorted_sorted': forall al :list int.
                      sorted al -> sorted' al
```

```
lemma sorted'_sorted: forall al :list int.
                      sorted' al -> sorted al
```

Listing 3.8: Código Why3 dos *lemmas* **sorted_sorted'** e **sorted'_sorted**.

A prova dos *lemmas* **sorted_sorted'** e **sorted'_sorted** não foram provadas automaticamente. É provável que a razão para o insucesso destas provas esteja na utilização da função **nth**.

Tal como em Coq, dependendo da forma como se escreve a especificação de um programa, pode ser mais fácil ou mais complicado provar que este está correto. Posto isto, concluída a prova de que os predicados **sorted** e **sorted'** são equivalentes, é um bom desafio fazer a prova da correção do programa **insertion sort**, mas utilizando a função **sorted'**. Para isto, foram definidos os *lemmas* **nth_error_insert**, **insert_sorted'** e **sort_sorted'**.

Estes três *lemmas* também não foram provados automaticamente, o que vem confirmar a ideia transmitida pelo livro da "Software Foundation", que dependendo da forma como se faz a especificação de um programa, pode ser mais ou menos difícil fazer a sua verificação. Ou seja, apesar de **sorted** ser equivalente a **sorted'**, a prova de **insert_sorted** é substancialmente mais fácil do que a prova de **insert_sorted'**. De um modo geral, caso fosse necessário provar **sort_sorted'**, seria muito mais fácil definir o predicado **sorted** e, posteriormente, provar **sort_sorted** e **sort_sorted'**.

Podemos concluir então que formulações diferentes da mesma especificação funcional podem levar a grandes desigualdades na complexidade das provas de correção.

3.1.3 *Formalização da função Sorted"*

A terceira opção para a definição da função **sorted** é baseada no mesmo princípio da alternativa anterior: para quaisquer dois elementos da lista, estes devem estar na ordem correta. Isto é, para quaisquer índices válidos **i** e **j** numa lista **lst**, se **i < j**, então **lst i <= lst j**, onde o índice **lst n** significa o elemento de **lst** no índice **n**. Como foi referido anteriormente, formalizar esta ideia em Coq é complexo, dado que mesmo quando o índice está fora do intervalo da lista, tem que ser retornado algum resultado. Então, nesta terceira formalização da função **sorted**, em Coq, recorre-se à função que existe na sua biblioteca *standard*:

```
Check nth : ∀ A : Type, nat → list A → A → A.
```

A função **nth** recebe um argumento extra do tipo **A** - um valor *default* - a ser retornado, caso o índice esteja fora do intervalo. Desta forma, garante que é sempre retornado um resultado.

Quando é utilizada a função **nth** é necessário garantir que os índices **i** e **j** estão dentro do intervalo. Em Coq define-se a função **sorted** à custa de **nth**, da seguinte forma:

```
Definition sorted'' (al : list nat) := ∀ i j,
    i < j < length al →
    nth i al 0 ≤ nth j al 0.
```

A escolha do valor *default* (neste caso 0) não é importante, já que este nunca será retornado para os valores **i** e **j** passados. Em Why3, para formalizar esta função **nth**, existem, novamente, duas opções distintas: define-se do início ou utiliza-se a função da biblioteca *standard*. Esta função do Coq, já existe em Why3 como **nth** pertencente ao módulo **NthNoOpt**. No entanto, o tipo da função não é exatamente igual, sendo que a função do módulo **NthNoOpt** recebe como argumentos, apenas um inteiro e uma lista, enquanto que a função do Coq recebe um inteiro (no caso, um **nat**), uma lista e um argumento extra do tipo **A**. Apesar disso, foi decidido utilizar-se a seguinte definição da biblioteca do Why3:

```
module NthNoOpt

function nth (n: int) (l: list 'a) : 'a

axiom nth_cons_0: forall x:'a, r:list 'a. nth 0 (Cons x r) = x
axiom nth_cons_n: forall x:'a, r:list 'a, n:int.
  n > 0 -> nth n (Cons x r) = nth (n-1) r

end
```

Listing 3.9: Código Why3 da função **nth**.

A função **nth** presente na biblioteca do Why3 está definida por axiomas, ou seja, não está estabelecida para todos os *inputs* possíveis. No entanto, como os índices passados à função são sempre válidos é possível utilizar esta função a par com a função **nth**, definida em Coq. Portanto, a função **sorted"** é escrita à custa da função **nth** do módulo **NthNoOpt**:

```
predicate sorted'' (al: list int) =
    forall i j :int.
    i < j < length al -> nth i al <= nth j al
```

Listing 3.10: Código Why3 do predicado **sorted"**.

A definição **predicate** é o que corresponde à *keyword* **Definition** utilizada em Coq, como já foi visto anteriormente. Esta é a definição da função **sorted"**, no entanto não foi explorada a sua prova. Foi definida apenas por uma questão de completude, para ser o mais parecida possível com o Coq e também para se conhecer a função existente na biblioteca *standard* do Why3 que permite escrever esta definição.

Apesar do objetivo final deste trabalho consistir na conversão de formalizações em Coq para Why3, o foco passa também por conhecer melhor a ferramenta e explorar as suas diferentes funcionalidades. Assim, existirá duas versões alternativas no Why3 para a mesma formalização Coq:

- utilizando a linguagem lógica;
- utilizando a linguagem de programas.

3.1.4 *Formalização com a linguagem de programas*

Uma vez concluída a versão lógica, a próxima tarefa consiste na conversão do mesmo algoritmo, mas agora para uma alternativa que utiliza a linguagem **WhyML**. Esta versão é substancialmente mais pequena, visto que o programa é provado através das CV. Estas CV, do ponto de vista do IDE, vão aparecer como obrigações de prova, tal como acontece na lógica com os *lemmas* e os *goals*.

Assim, apenas, é necessário definir as duas funções principais do algoritmo, **insert** e **sort**, e as respetivas anotações necessárias para garantir a prova de cada uma destas. De um modo geral, estas anotações representam o papel dos *lemmas* na versão apresentada anteriormente. A função **insert** define-se:

```
let rec function insert (i: int) (l: list int) : list int
    requires { sorted l }
    ensures { sorted result } (* insert_sorted *)
    ensures { permut result (Cons i l) } (* insert_perm *)
    =
    match l with
    | Nil -> Cons i Nil
    | Cons h t -> if i <= h then Cons i l
                            else Cons h (insert i t)
    end
```

Listing 3.11: Código Why3 da função **insert**.

Na linguagem de programas, uma função recursiva introduz-se pela construção **let rec**. Quando é utilizada esta definição, é necessário a inclusão de uma anotação que prove a terminação da função, ou seja, o variante.

Uma alternativa a esta definição é utilizar a *keyword* **function**, que também permite definir uma função, com a particularidade da definição total e a terminação. A recursividade aplicada neste função é estrutural, isto é, opera sobre a cauda da lista, o que torna possível utilizar esta definição. Assim sendo, a anotação referente ao variante torna-se prescindível.

A função **insert** necessita de uma pré condição, que assegure que a lista recebida como argumento esteja ordenada, dado que o propósito desta função é inserir um determinado inteiro numa lista ordenada. As restantes anotações são pós condições resultantes dos *lemmas* **insert_sorted** e **insert_perm**.

A função **sort** escreve-se da seguinte maneira:

```
let rec function sort (l: list int) : list int
    ensures { sorted result } (* sort_sorted *)
    ensures { permut result l } (* sort_perm *)
    (* variant { length l } *)
    =
    match l with
    | Nil -> Nil
    | Cons h t -> insert h ( sort t )
```

```
        end
```

Listing 3.12: Código Why3 da função **sort**.

A função **sort** é definida seguindo os mesmos princípios da anterior. Também nesta a ausência da *keyword* **function** obriga a existência de uma anotação variante. As pós condições resultam dos *lemmas* **sort_sorted** e **sort_perm** definidos na formalização em Coq.

Após o programa anotado, são geradas as CV, posteriormente provadas. Nestas provas com contrato, as estratégias escolhem-se automaticamente. Por esta razão, demonstra-se mais uma vez, que a linguagem de programas é muito versátil, dado que na lógica, quando é utilizada a indução numa prova, esta tem que ser escolhida manualmente.

Nesta versão também foi definido o predicado **is_a_sorting_algorithm** e o *lemma* **insertion_sort_correct** que servem provar que o **insertion_sort** é um algoritmo de ordenação correto. Após concluir a prova da correção do algoritmo, o cenário com que nos deparamos é o seguinte:

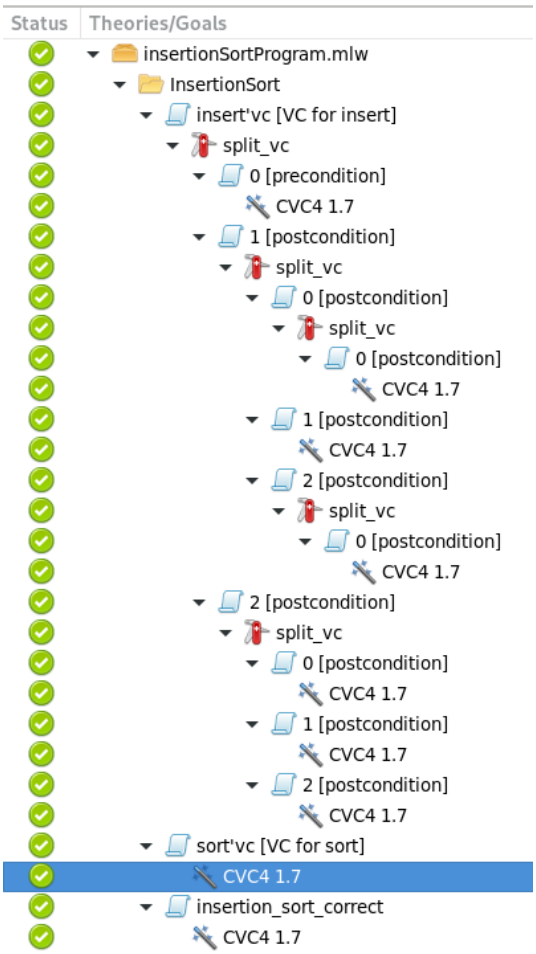

Figura 12: Apresentação no IDE do Why3 da prova completa do algoritmo **insertion sort** em **WhyML**.

3.2 SELECTION SORT

O algoritmo **selection sort**, tal como o **insertion sort** é, também, um algoritmo geralmente apresentado como um programa operando sobre *arrays*. Porém, neste caso, o objetivo centra-se na análise do algoritmo operando sobre listas.

Este algoritmo, em termos de código Coq, é mais interessante do que o algoritmo anterior, no sentido em que este explora uma nova técnica para mostrar o término de uma função. Em termos de eficiência, ambos os algoritmos têm um tempo de execução quadrático, no entanto o programa **selection sort** faz mais comparações que o **insertion sort**. No melhor caso, este último algoritmo executa em tempo linear, porém no caso médio e no pior caso, o tempo de execução é quadrático. Posto isto, acaba por torna-se mais vantajoso utilizar o algoritmo **insertion sort** em *inputs* de pequenas dimensões.

O **selection sort** é um algoritmo de ordenação que consiste na procura do menor elemento de uma lista, repetidamente, e, posteriormente, na construção da lista de resultados.

Para uma melhor organização deste exemplo, foi decidido separar-se o código em três módulos diferentes. O primeiro corresponde à função **select** e a todos os *lemmas* necessários à prova das suas propriedades. O segundo e o terceiro representam a função **selsort** e **selsort'**, respetivamente, bem como os *lemmas* para a sua prova.

São ambas a mesma função recursiva, embora a segunda seja uma definição alternativa da primeira. A principal diferença está na recursividade da função, que na primeira é estrutural ao contrário do que acontece na segunda opção. Estes dois últimos módulos importam o primeiro, já que a função **select** é utilizada como auxiliar nas duas definições de **selsort**.

3.2.1 *Formalização da função Select*

A função **select** é uma das funções auxiliares deste algoritmo, composta por duas ações distintas: a primeira corresponde à procura do menor elemento e a segunda é onde ocorre a sua transição para a posição inicial da lista. A função recebe como argumentos, um inteiro **x** e uma lista **l** e retorna como resultado o par **(y, l')**, onde **y** é o menor elemento e **l'** os restantes elementos da lista **x :: l**, na sua ordem original.

```
Fixpoint select (x: nat) (l: list nat) : nat × list nat :=
  match l with
  | [] ⇒ (x, [])
  | h :: t ⇒ if x <=? h
             then let (j, l') := select x t
                  in (j, h :: l')
             else let (j, l') := select h t
                  in (j, x :: l')
  end.
```

A função **select** em Why3, abaixo, é definida através da definição **function**, visto que a função é recursiva e apresenta um padrão de recursividade simples (estrutural).

```
function select (x :int) (l :list int) : (int, list int) =
  match l with
  | Nil -> (x, Nil)
  | Cons h t -> if x <= h
                then let (j, l') = select x t
                     in (j, Cons h l')
                else let (j, l') = select h t
                     in (j, Cons x l')
  end
```

Listing 3.13: Código Why3 da função **select**.

A conversão desta função é simples e direta. As diferenças existentes entre as duas versões (Coq e Why3) já foram referidas em definições anteriores.

Focando na prova da correção, é necessário definir alguns *lemmas*:

- **select_perm** - prova que o resultado da função **select** é uma permutação em relação ao seu *input*;
- **select_rest_length** - verifica que o tamanho da lista que a função **select** recebe como argumento é igual ao tamanho da lista que esta retorna como resultado;
- **select_fst_leq** - afirma que a primeira componente do resultado da função **select** é menor ou igual do que o valor de **x** que é passado como *input*, ou seja, se **select x al = (y, bl)**, então **y <= x**;
- **select_smallest** - garante que a primeira componente do resultado da função **select** não é superior do que todos os elementos da segunda componente, isto é, se **select x al = (y, bl)**, então **y <= bl**;
- **select_in** - prova que o resultado da função **select** tem que ser um dos elementos da lista que recebe como argumento.

```
Lemma select_perm: ∀ x l y r,
  (y, r) = select x l → Permutation (x :: l) (y :: r).

Lemma select_rest_length : ∀ x l y r,
  select x l = (y, r) → length l = length r.

Lemma select_fst_leq: ∀ al bl x y,
  select x al = (y, bl) → y ≤ x.

Definition le_all x xs := Forall (fun y ⇒ x ≤ y) xs.
```

```
Infix "<=*" := le_all (at level 70, no associativity).

Lemma select_smallest: ∀ al bl x y,
  select x al = (y, bl) → y <=* bl.

Lemma select_in : ∀ al bl x y,
  select x al = (y, bl) → In y (x :: al).
```

Convertendo para Why3 as definições apresentadas acima:

```
lemma select_perm: forall l r:list int, x y:int.
                    let (y, r) = select x l in permut (Cons x l) (Cons y r)

lemma select_rest_length: forall l r :list int, x :int.
                           let (_, r) = select x l in length l = length r

lemma select_fst_leq: forall al :list int, x :int.
                       let (y, bl) = select x al in y <= x

predicate le_all (x :int) (xs :list int) =
  forall y :int. mem y xs -> x <= y

lemma select_smallest: forall al :list int, x :int.
                        let (y, bl) = select x al in le_all y bl

lemma select_in: forall al :list int, x :int.
                  let (y, bl) = select x al in mem y (Cons x al)
```

Listing 3.14: Código Why3 dos *lemmas* da função **select**.

As provas destes *lemmas* são relativamente simples, procedendo com a ajuda da indução e dos *solvers* automáticos. O *lemma* **select_perm** foi provado através da tranformação **induction_ty_lex**, seguido do SMT **Alt-Ergo**. Esta prova apenas foi conseguida utilizando uma outra máquina, contendo a versão **2.4.0** do respetivo *solver*. A função **permut** utilizada na definição deste *lemma* pertence à biblioteca do Why3.

Na prova dos *lemmas* **select_rest_length**, **select_fst_leq**, **select_smallest** e **select_in** foi utilizada a transformação **induction_ty_lex** e, posteriormente, com a ajuda do **Alt-Ergo** foi possível provar os resultados pós transformações.

Uma vez que a função **select** é definida por recursividade estrutural, seria de esperar que as suas provas fossem realizáveis com a transformação **induction_ty_lex**, o que se verificou na prática.

O predicado **le_all** utilizado na definição anterior serve para verificar, dado um inteiro **x** e uma lista **l**, se o valor de **x** é menor ou igual do que todos os elementos de **l**. Este predicado utiliza o predicado **mem** definido no

módulo **Mem** da biblioteca do Why3. Este verifica se um determinado elemento pertence a uma determinada lista.

Para o sucesso das provas, foi fundamental a utilização do **let** e de *pattern-matching*, visto que permitem evitar a utilização de mais variáveis lógicas quantificadas. No Why3 como as provas são automáticas, pode ser necessário formular os resultados de forma ligeiramente diferente do Coq, tal como trocar a ordem das variáveis quantificadas ou introduzir **let**. Estas propriedades podem ser verificadas no exemplo abaixo.

```
lemma select_fst_leq: forall al bl :list int, x y :int.
                        select x al = (y, bl) -> y <= x

lemma select_fst_leq2: forall x :int, al :list int.
                        let (y, bl) = select x al in y <= x

lemma select_fst_leq3: forall al :list int, x :int.
                        let (y, bl) = select x al in y <= x
```

Figura 13: Definições do *lemma* **select_fst_leq**.

Estão demonstradas três formas diferentes de definir o *lemma* **select_fst_leq**. Na primeira, uma versão exatamente igual à do Coq, na segunda é já introduzido um **let** na definição, mas onde primeiro é declarada a variável **x** do tipo **int** e só depois a lista **al** e na terceira o *lemma* é definido também à custa de um **let**, porém declarando em primeiro a variável **al** e só depois o **x**.

A prova destes três *lemmas* confirma precisamente o que foi dito anteriormente, dado que na prova do primeiro *lemma* é aplicada a transformação **induction_ty_lex**, porém os *solvers* não são capazes de resolver a restante prova. Na prova do *lemma* **select_fst_leq2** é necessário primeiramente aplicar um **intros al** para que esta variável seja adicionada ao contexto e para que depois se possa realizar a indução sobre esta. Na prova do último *lemma* a prova é feita utilizando apenas a transformação **induction_ty_lex** e, posteriormente, os *solvers* concluem a restante prova.

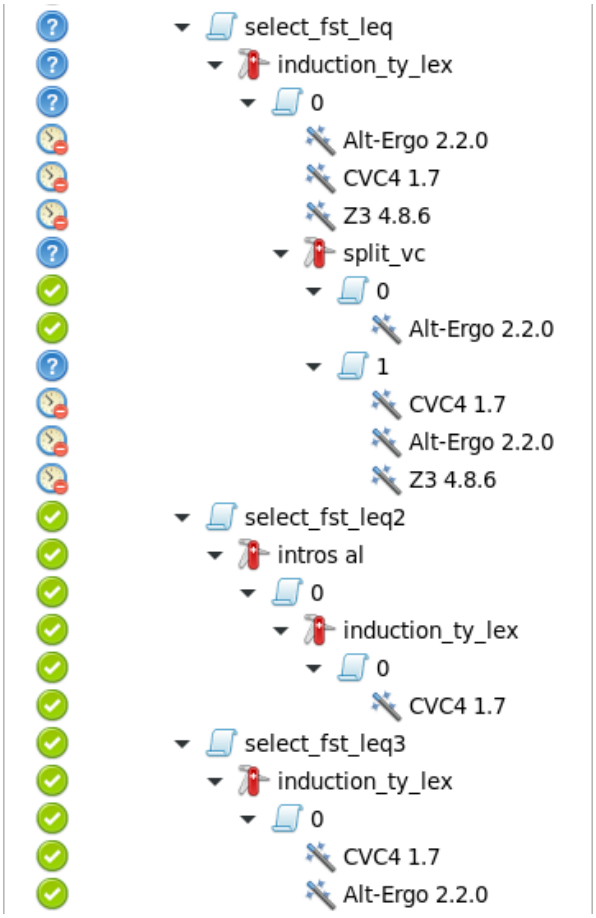

Figura 14: Prova das três versões do *lemma* **select_fst_leq**.

3.2.2 *Formalização da função Selsort*

A segunda função auxiliar do algoritmo **selection sort** é a função **selsort** que corresponde ao processo de correr a função **select**, repetidamente, por forma a ordenar uma lista. Na formalização Coq, na primeira tentativa da definição desta função foi escrita a seguinte opção:

```
Fail Fixpoint selsort (l : list nat) : list nat :=
  match l with
  | [] ⇒ []
  | x :: r ⇒ let (y, r') := select x r
              in y :: selsort r'
  end.
```

Porém, esta definição não funciona, já que a chamada recursiva não é estrutural. Por outras palavras, para o sistema não é evidente que **r'** é uma parte menor da lista original **l** e por isso não satisfaz os requisitos

para a terminação da função. Para resolver este problema, basta adicionar um argumento extra, cuja função é, apenas, limitar o número de vezes que se aplica a recursividade. A recursividade passa, então, a ser estrutural neste argumento **n**: a chamada recursiva é feita sobre o predecessor (**n**) do argumento (**S n**), com um caso de paragem para o construtor 0, o que garante a terminação. Assim, quando o argumento **n**, representado abaixo, atinge o valor 0, a recursividade termina.

```
Fixpoint selsort (l : list nat) (n : nat) : list nat :=
  match l, n with
  | _, O ⇒ [] (* ran out of fuel *)
  | [], _ ⇒ []
  | x :: r, S n' ⇒ let (y, r') := select x r
                   in y :: selsort r' n'
  end.
```

O valor necessário para **n** é o tamanho da lista **l**. A função **selection sort** define-se, então, da seguinte forma:

```
Definition selection_sort (l : list nat) : list nat :=
  selsort l (length l).
```

Fazendo a conversão para Why3 utilizando, apenas, a linguagem lógica, também aqui, se encontram mais dificuldades do que no exemplo realizado anteriormente, o **insertion sort**.

A maior dificuldade nesta formalização recai sobre o facto de não existir teoria de naturais em Why3 e por isso não é possível fazer *pattern-matching*. A melhor forma de resolver este problema é criando um tipo **nat** e definindo uma função **natLength** que, não é nada mais que a função **length** mas com o tipo de retorno **nat**. Com este tipo, é possível definir a função **selsort** e a **selection_sort**, tal como em Coq:

```
type nat = Z | S nat

function natLength (l :list int) : nat =
  match l with
  | Nil -> Z
  | Cons _ t -> S (natLength t)
  end

function selsort (l :list int) (n :nat) : list int =
  match l,n with
  | _, Z   -> Nil
  | Nil, _ -> Nil
  | Cons x r, S n' -> let (y, r') = select x r
                      in Cons y (selsort r' n')
  end
```

```
function selection_sort (l : list int) : list int =
  let n = natLength l in selsort l n
```

Listing 3.15: Código Why3 da função **selsort**.

Concluída a definição do programa **selection sort**, segue-se a prova da correção. Ordenar os elementos de uma lista por forma a torná-la ordenada é a especificação de um algoritmo de ordenação correto. Em Coq é definido o predicado indutivo **sorted**, no entanto em Why3 optamos por utilizar, mais uma vez a definição da biblioteca *standard*. Novamente, é definido também o predicado **is_a_sorting_algorithm**.

Para a prova da correção do algoritmo, temos que definir os seguintes *lemmas*:

- **selsort_perm** - prova que o *output* da função **selsort** é uma permutação em relação ao seu *input*;
- **selection_sort_perm** - serve para verificar que a função **selection_sort** retorna uma permutação da lista que recebe como argumento;
- **cons_of_small_maintains_sort** - prova que a adição de um elemento ao início de uma lista ordenada, mantém a ordenação, desde que o elemento seja suficientemente pequeno e que seja fornecido combustível suficiente;
- **selsort_sorted** e **selection_sort_sorted** - provam que a lista retornada das funções **selsort** e **selection_sort**, respetivamente, está ordenada;
- **selection_sort_is_correct** - finaliza a prova de correção, provando que **selection_sort** é um algoritmo de ordenação correto.

```
Lemma selsort_perm: ∀ n l,
  length l = n → Permutation l (selsort l n).

Lemma selection_sort_perm: ∀ l,
  Permutation l (selection_sort l).

Lemma cons_of_small_maintains_sort: ∀ bl y n,
  n = length bl → y <=* bl →
  sorted (selsort bl n) →
  sorted (y :: selsort bl n).

Lemma selsort_sorted : ∀ n al,
  length al = n → sorted (selsort al n).

Lemma selection_sort_sorted : ∀ al,
  sorted (selection_sort al).
```

```
Theorem selection_sort_is_correct :
  is_a_sorting_algorithm selection_sort.
```

Escrevendo em Why3:

```
lemma selection_sort_perm: forall l :list int.
                           permut l (selection_sort l)

lemma selection_sort_perm: forall l :list int.
                           permut l (selection_sort l)

lemma cons_of_small_maintains_sort: forall n :nat, y :int, bl :list int.
                                    n = natLength bl -> le_all y bl ->
                                    sorted (selsort bl n) ->
                                    sorted (Cons y (selsort bl n))

lemma selsort_sorted: forall n :nat, al :list int.
                      n = natLength al -> sorted (selsort al n)

lemma selection_sort_sorted: forall al : list int.
                             sorted (selection_sort al)

lemma selection_sort_is_correct: is_a_sorting_algorithm selection_sort
```

Listing 3.16: Código Why3 dos *lemmas* das funções **selsort** e **selection_sort**.

As provas destes *lemmas* são simples e diretas, sendo apenas necessário recorrer aos SMT *solvers*. Para provar os *lemmas* **selection_sort_perm** e **cons_of_small_maintains_sort** foi apenas necessária a ajuda do SMT *solver* **CVC4**.

As provas dos *lemmas* **selsort_perm** e **selsort_sorted** não foram concluídas com sucesso. Seria de esperar que, através da indução, estas fossem provadas, já que esta versão da definição de **selsort** é recursivamente estrutural em **n**. É muito provável que a causa deste problema seja o facto de a função **natLength** ter sido definida por nós, e por isso, não ter quaisquer *lemmas* definidos sobre as suas propriedades. O ***solver* CVC4** também foi capaz de fazer a prova dos *lemmas* **selection_sort_sorted** e **selection_sort_is_correct**.

3.2.3 *Formalização da função Selsort'*

Anteriormente, definimos a função **selsort** como sendo uma função estruturalmente recursiva. Foi necessário adicionar um valor **n**, que diminui a cada chamada recursiva, até atingir o valor 0. Ou seja, apesar de numa primeira análise, esta função não ser estruturalmente recursiva, tanto o Coq como o Why3, aceitaram esta versão em que o **n** é estruturalmente decrescente. No entanto, esta definição dificulta tanto a implementação

da função **selsort** como as suas provas. Então, a função **selsort'** é uma definição alternativa à **selsort** definida acima.

Em Coq existe a *keyword* **Function** que implementa uma definição similar à anterior, no entanto permite lidar com a recursividade não estrutural. Em vez disso, é anotada com uma **measure** que é decrescente a cada chamada recursiva.

```
Function selsort' l {measure length l} :=
  match l with
  | [] ⇒ []
  | x :: r ⇒ let (y, r') := select x r
              in y :: selsort' r'
  end.
```

Esta definição é anotada com uma **measure** que serve para informar o Coq que em cada chamada recursiva diminui o comprimento de **l**. A anotação **measure** receber dois parâmetros, uma função e um argumento. Neste caso, a função é **length** e o argumento é **l**. O resultado desta função **length** é do tipo **nat**.

Em Why3 a anotação **variant** é o equivalente à **measure** utilizada em Coq. Esta anotação é o que indica qual é a variável decrementada.

Apesar de estarmos na lógica, e este tipo de definições serem utilizadas na linguagem **WhyML**, existe, em Why3, a *keyword* **ghost** que permite fazer a ligação entre a linguagem lógica e a de programas. Esta admite anotar uma função, mas no nível lógico, o que permite garantir que uma variável é decrementada e, assim assegurar que a mesma termina. Posto isto, a função **selsort'** pode definir-se alternativamente, em Why3, da seguinte forma:

```
let rec ghost function selsort' (l :list int) : list int
  variant {length l}
=
  match l with
  | Nil -> Nil
  | Cons x r -> let (y, r') = select x r
                in Cons y (selsort' r')
  end
```

Listing 3.17: Código Why3 da função **selsort'**.

Na formalização em Coq, não foi provada a correção do algoritmo, dado que ficou a cargo do leitor. Para esta prova não é possível utilizar a indução, visto que a recursividade da função não é estrutural. Por esta razão, para tentar fazer a prova em Why3, utilizando apenas a lógica, seríamos levados a estudar detalhadamente e utilizar a linguagem de transformações, o que nos aproximaria da forma de efectuar provas em Coq, que não é o que procuramos no Why3. Por isso, optou-se por parar neste ponto e passar à linguagem de programas, que é a forma mais natural de realizar estas provas em Why3.

3.2.4 *Formalização com a linguagem de programas*

Concluída a versão lógica, a próxima fase consiste na conversão, do mesmo algoritmo, para uma opção que utiliza, apenas, a linguagem de programas.

Esta versão é, novamente, bastante mais pequena, dado que o programa é provado através das CV. Ao analisar esta questão percebe-se que, de facto, esta possibilidade de escrever programas dispondo da linguagem **WhyML**, torna muito mais intuitivo o Why3.

As funções do algoritmo, propriamente dito, são definidas, bem como, as anotações necessárias para as suas provas. A função **select** é definida abaixo:

```
predicate le_all (x:int) (xs:list int) = forall y :int . mem y xs -> x <= y


let rec function select (x : int) (l : list int) : (int, list int)
  ensures { let (y,l') = result in permut (Cons y l') (Cons x l) } (*
select_perm *)
  ensures { let (_,r) = result in length l = length r } (* select_rest_length
 *)
  ensures { let (y,_) = result in y <= x } (* select_fst_leq *)
  ensures { let (y,bl) = result in le_all y bl } (* select_smallest *)
  ensures { let (y,_) = result in mem y (Cons x l)  } (* select_in *)
=
  match l with
  | Nil -> (x, Nil)
  | Cons h t ->
    if x <= h
    then let (j, l') = select x t
         in (j, Cons h l')
    else let (j, l') = select h t
         in (j, Cons x l')
  end
```

Listing 3.18: Código Why3 da função **select**.

Cada uma das anotações apresentadas corresponde a um dos *lemmas* definidos anteriormente, tanto na versão Coq como na versão lógica do Why3. Quanto ao código, todas as diferenças existentes já foram referidas noutras funções.

Na formalização em Why3, optou-se por definir a função **selsort** como a segunda versão do Coq, ou seja, sem recorrer à definição dos naturais para garantir a recursividade estrutural. Isto porque na linguagem de programas a definição de funções pode utilizar variantes que desempenham a mesma função que a anotação **measure** nas **Function** do Coq. A conversão da função **selsort'** resulta no seguinte:

```
let rec function selsort (l : list int) : list int
```

```
    variant { length l }
    ensures { permut result l} (* selsort_perm *)
    ensures { forall y: int. le_all y l -> sorted result -> sorted (Cons y
  result) } (* cons_of_small_maintains_sort *)
    ensures { sorted result } (* selsort_sorted *)
  =
    match l with
    | Nil -> Nil
    | Cons x r -> let (y, r') = select x r
          in Cons y (selsort r')
    end
```

Listing 3.19: Código Why3 da função **selsort**.

Nesta função, as anotações resultam todas dos *lemmas* anteriores, à exceção do **variant**. Esta anotação é essencial para garantir que a função termina, devido ao facto da recursividade desta não ser estrutural.

Por uma questão de completude, define-se a função **selsort** e a função **selection_sort** presentes no Coq, apesar de esta segunda apenas chamar a primeira e retornar o seu resultado. Posto isto, esta é definida assim:

```
let rec function selection_sort (l : list int) : list int
    ensures { permut result l} (* selection_sort_perm *)
    ensures { sorted result } (* selection_sort_sorted *)
  =
    selsort l
```

Listing 3.20: Código Why3 da função **selection_sort**.

Mais uma vez, estas anotações **ensures** são resultantes dos *lemmas* apresentados na formalização Coq. Após o programa totalmente anotado, as CV são geradas e posteriormente provadas. A prova é muito simples, uma vez que não é preciso utilizar qualquer estratégia, porque a prova indutiva é feita seguindo a estrutura não das listas mas da própria definição da função. A prova das CV geradas pelas anotações da função **selsort** encontra-se abaixo totalmente verificada.

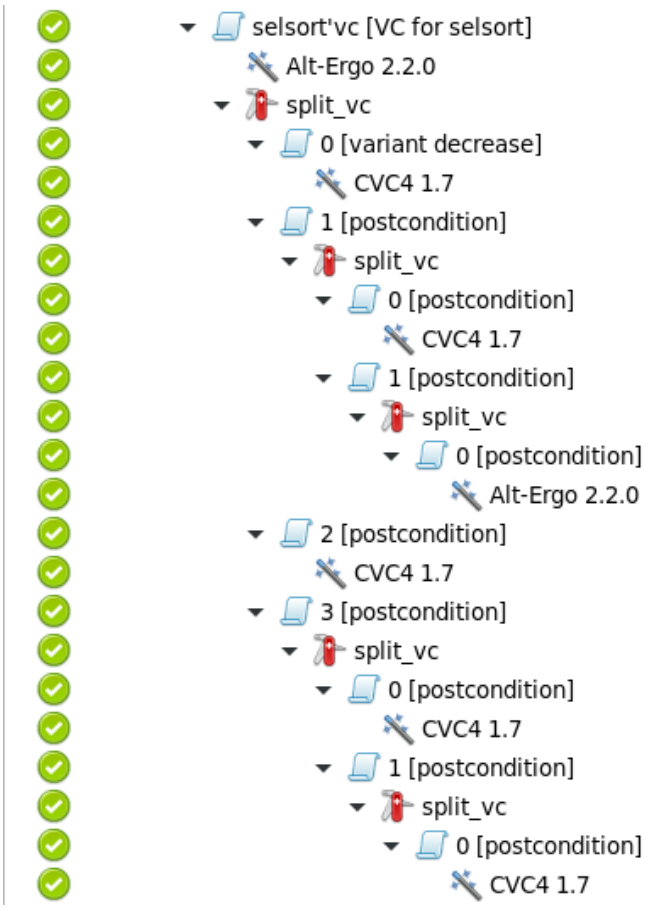

Figura 15: Prova das CV da função **selsort**.

Foi criado um total de quatro CV para esta prova e foram todas provadas com os SMT *solvers*. E, posteriormente, para dar mais segurança foi executado o *smoke detector* sobre este desenvolvimento. As CV do *smoke detector* foram também todas provadas e por isso pode concluir-se que não existem inconsistências no código.

```
[barbara@localhost theorySelectionSort]$ why3 replay --smoke-detector=top
selectionSortProgram
 4/4 No smoke detected
```

Figura 16: Resultado de correr o *smoke detector* na verificação do algoritmo **Selection Sort**.

3.3 MERGE SORT

O algoritmo de ordenação **merge sort** é normalmente apresentado como um algoritmo imperativo que opera sobre *arrays*. Mais uma vez, a versão utilizada opera sobre listas.

A ideia básica deste algoritmo consiste em dividir (**split**) a lista a meio, ordenar recursivamente cada uma das partes e depois fundir (**merge**) os resultados das duas sublistas:

```
mergesort xs =
  split xs into ys,zs;
  ys' = mergesort ys;
  zs' = mergesort zs;
  return (merge ys' zs')
```

Neste exemplo, a formalização em Coq apresenta a definição das funções juntamente com os *lemmas* essenciais à prova da sua correção.

3.3.1 *Definição e verificação da função Split em Coq*

O primeiro passo é a escrita da função **split**.

```
Fixpoint split {X:Type} (l:list X) : (list X × list X) :=
  match l with
  | [] ⇒ ([],[])
  | [x] ⇒ ([x],[])
  | x1::x2::l' ⇒
    let (l1,l2) := split l' in
    (x1::l1,x2::l2)
  end.
```

O método utilizado para a definir é simplesmente alternar a atribuição de elementos em sublistas à esquerda e à direita. Esta é uma função polimórfica, visto que o tipo dos valores do *input* é irrelevante para o processo de divisão.

Embora esta função seja simples de definir, pode ser desafiante trabalhar com ela. Na formalização em Coq, os autores do livro tentam provar o seguinte *lemma*, que é obviamente válido:

```
Lemma split_len_first_try: ∀ {X} (l:list X) (l1 l2: list X),
  split l = (l1,l2) →
  length l1 ≤ length l ∧
  length l2 ≤ length l.
```

O *lemma* **split_len_first_try** prova que o comprimento das duas listas que a função **split** retorna como resultado, têm comprimento não superior ao da lista que esta recebe como argumento.

Para fazer esta prova, não é possível utilizar a indução, já que esta função não é recursivamente estrutural e o princípio de indução em listas exige que mostremos que a propriedade que está a ser provada segue para qualquer lista não vazia se esta se mantiver para a cauda dessa lista. Logo, é necessário fazer esta prova recorrendo a um princípio de indução de **dois passos**, que requer que mostremos que a propriedade a ser provada se segue para uma lista de pelo menos dois elementos, se ela se mantiver para a cauda da cauda dessa lista.

Em Coq é possível definir este princípio, dado que a manipulação da indução é uma das propriedades desta ferramenta, assim:

```
Definition list_ind2_principle:=
  ∀ (A : Type) (P : list A → Prop),
    P [] →
    (∀ (a:A), P [a]) →
    (∀ (a b : A) (l : list A), P l → P (a :: b :: l)) →
    ∀ l : list A, P l.
```

Se se assumir a correção deste princípio de indução, a prova de **spit_len** é fácil, uma vez que, o Coq, é capaz de especificar o princípio de indução a utilizar:

```
Lemma split_len': list_ind2_principle →
  ∀ {X} (l:list X) (l1 l2: list X),
  split l = (l1,l2) →
  length l1 ≤ length l ∧
  length l2 ≤ length l.
```

No entanto, ainda é necessário provar **list_ind2_principle**. Para isso, escreve-se um termo de prova explícito. Portanto:

```
Definition list_ind2 :
  ∀ (A : Type) (P : list A → Prop),
      P [] →
      (∀ (a:A), P [a]) →
      (∀ (a b : A) (l : list A), P l → P (a :: b :: l)) →
      ∀ l : list A, P l :=
  fun (A : Type)
      (P : list A → Prop)
      (H : P [])
      (H0 : ∀ a : A, P [a])
      (H1 : ∀ (a b : A) (l : list A), P l → P (a :: b :: l))
    ⇒ fix IH (l : list A) : P l :=
    match l with
    | [] ⇒ H
    | [x] ⇒ H0 x
    | x::y::l' ⇒ H1 x y l' (IH l')
    end.
```

A *keyword* **fix** define uma função local recursiva **IH**, que é devolvida como o valor global do **list_ind2**. Com o princípio de indução já definido, podemos finalmente provar o *lemma* **split_len** inicial, através das definições anteriores:

```
Lemma split_len: ∀ {X} (l:list X) (l1 l2: list X),
    split l = (l1,l2) →
    length l1 ≤ length l ∧
    length l2 ≤ length l.
Proof.
    apply (@split_len' list_ind2).
Qed.
```

Também é necessário para a prova da correção da função **mergesort** incluir o *lemma* **split_perm**, que prova que a concatenação das listas retornadas como resultado da função **split** é uma permutação da lista inicial:

```
Lemma split_perm : ∀ {X:Type} (l l1 l2: list X),
    split l = (l1,l2) → Permutation l (l1 ++ l2).
```

Este exemplo consegue demonstrar uma clara limitação do Why3 face ao Coq. Sendo a linguagem deste de primeira ordem, não é possível de todo definir *lemmas* correspondentes a princípios de indução. Estes princípios correspondem, no Why3, a transformações de prova, que não são editáveis pelo utilizador. Contudo, esta limitação é bastante natural, visto que não faz parte das funcionalidades do Why3 a definição de princípios de indução nem de transformações para provas de indução. Por esta razão, a prova da formalização deste algoritmo utilizando apenas a linguagem lógica do Why3, seria pouco intuitiva e extremamente complexa. Como na linguagem de programas as provas indutivas são muito naturais e simples, optou-se por escrever o algoritmo, apenas recorrendo à linguagem de programas.

3.3.2 *Definição e verificação da função Split em Why3*

Fazendo então a conversão da função **split** e dos respetivos *lemmas* para a sua prova, fica:

```
let rec function split (l :list int) : (list int, list int)
  ensures { let (l1,l2) = result in length l1 <= length l /\ length l2 <=
length l } (* split_len *)
  ensures { let (l1,l2) = result in length l >= 2 -> length l1 < length l /\
length l2 < length l } (* split_len *)
  ensures { let (l1,l2) = result in permut l (l1 ++ l2) } (* split_perm *)
=
  match l with
  | Nil -> (Nil, Nil)
  | Cons x Nil -> (Cons x Nil, Nil)
  | Cons x1 (Cons x2 l') -> let (l1, l2) = split l'
```

```
                                in (Cons x1 l1, Cons x2 l2)
    end
```

Listing 3.21: Código Why3 da função **split**.

As anotações **ensures** exibidas são as pós condições resultantes dos dois *lemmas* apresentados anteriormente para a prova da correção. O **split_len** está dividido em dois, já que o tamanho das listas retornadas só têm o mesmo tamanho da lista recebida como argumento, quando esta é vazia ou tem apenas um elemento. Posto isto, quando a lista recebida tem tamanho **>=2**, as listas retornadas têm sempre tamanho inferior.

Esta separação é fundamental para a prova do algoritmo, caso contrário não se consegue concluir a prova da função **mergesort** com sucesso, pois é o que está a garantir que o resultado da função **split** retorna sempre duas lista menores que a lista original, confirmando assim a terminação desta função. Quanto à escrita da função em si e dos *lemmas* como pós condições, a conversão é direta e simples.

3.3.3 *Definição e verificação da função Merge em Coq*

A segunda função auxiliar do algoritmo **merge sort** corresponde à função **merge**, que recebe duas listas ordenadas e retorna como resultado a junção numa única e totalmente ordenada. A primeira tentativa desta definição foi a seguinte:

```
Fixpoint merge l1 l2 :=
  match l1, l2 with
  | [], _ ⇒ l2
  | _, [] ⇒ l1
  | a1::l1', a2::l2' ⇒
      if a1 <=? a2 then a1 :: merge l1' l2
                   else a2 :: merge l1 l2'
  end.
```

Acontece que a definição **Fixpoint** exige que a função seja estruturalmente recursiva em algum dos parâmetros e por isso esta definição falha. Apesar de em cada chamada recursiva ser passado ou **l1'** ou **l2'**, isto é, a cauda do seu valor original, as chamadas recursivas na definição **Fixpoint** devem sempre diminuir num único argumento fixo. A função **merge** foi então definida recorrendo a outra alternativa:

```
Fixpoint merge l1 l2 {struct l1} :=
  let fix merge_aux l2 :=
  match l1, l2 with
  | [], _ ⇒ l2
  | _, [] ⇒ l1
  | a1::l1', a2::l2' ⇒
```

```
        if a1 <=? a2 then a1 :: merge l1' l2
                     else a2 :: merge_aux l2'
    end
    in merge_aux l2.
```

Esta definição externa é aceite pelo Coq, dado que a função é estruturalmente decrescente em **l1** (especificado com a anotação **struct l1**), tal como a definição interna é também aceite porque é estruturalmente recursiva no seu único argumento.

Sobre os *lemmas* acerca da função **merge**, foram definidos dois, o **merge2** e o **merge_nil_l**. O *lemma* **merge2** prova que para um dado **x1 <= x2** e para uma lista **r1** e **r2** , correndo a função **merge**, tal que, **merge (x1::r1) (x2::r2)**, o elemento **x1** fica à cabeça da lista e a recursividade é aplicada sobre a restante lista.

O *lemma* **merge_nil_l** garante que fazendo o **merge** de uma lista vazia e um lista **l**, o resultado é a própria da lista **l**. Estes dois *lemmas* são definidos, em Coq, da seguinte maneira:

```
Lemma merge2 : ∀ (x1 x2:nat) r1 r2,
    x1 ≤ x2 →
    merge (x1::r1) (x2::r2) =
    x1::merge r1 (x2::r2).

Lemma merge_nil_l : ∀ l, merge [] l = l.
```

3.3.4 *Definição e verificação da função Merge em Why3*

Novamente, fazendo a tradução, para Why3, da função **merge** e dos respetivos *lemmas* definidos:

```
let rec function merge (l1 l2 :list int) : list int
  variant { length (l1 ++ l2) }
  ensures { l1 = Nil -> result = l2 } (* merge_nil_l *)
  ensures { forall x x1 x2 :int, l1' l2' :list int. l1 = (Cons x1   l1') ->
l2 = (Cons x2 l2') -> x <= x1 -> x <= x2 -> sorted result -> sorted (Cons x
result) } (* sorted_merge1 *)
  ensures { sorted l1 -> sorted l2 -> sorted result } (* sorted_merge *)
  ensures { permut (l1 ++ l2) result } (* merge_perm *)
=
  match l1, l2 with
  | Nil, _ -> l2
  | _, Nil -> l1
  | (Cons a1 l1'), (Cons a2 l2') -> if a1 <= a2
                                   then (Cons a1 (merge l1' l2))
                                   else (Cons a2 (merge l1 l2'))
end
```

Listing 3.22: Código Why3 da função **merge**.

Em Why3, podemos utilizar a primeira definição da função **merge** do Coq, uma vez que na linguagem de programas do Why3, quando uma função não é estruturalmente recursiva, existe a possibilidade de a anotar com um **variant**, que garante a terminação da mesma.

Portanto, a função **merge** tem a necessidade de incluir uma anotação **variant**. As anotações **ensures** resultam dos *lemmas* definidos para a correção da função.

Mais à frente na formalização em Coq, para provar a correção do programa **merge sort** é necessário mostrar que este retorna como resultado final uma lista ordenada e que este resultado é uma permutação da lista inicial. Para isso, a função **merge** também precisa de alguns *lemmas* para provar estas mesmas propriedades sobre si mesma. Daí, surgem as restantes pós condições, acima, associadas aos *lemmas* **sorted_merge1**, **sorted_merge** e **merge_perm**.

O *lemma* **sorted_merge1** prova que para um determinado **x** não superior ao primeiro elemento das duas lista *input* da função **merge**, se o **merge** destas retorna uma lista ordenada então pondo **x** à cabeça deste resultado, também esta lista **(Cons x (merge l1 l2))** é ordenada. O **sorted_merge** prova que dada uma lista **l1** ordenada e uma lista **l2** também ordenada, o resultado do **merge** das duas é uma lista ordenada. O *lemma* **merge_perm** garante que o resultado da função **merge** é uma permutação das duas listas do seu *input* concatenadas.

Uma anotação do tipo pós condição não permite a invocação da própria função. No caso do *lemma* **merge2**, este exige a invocação da função **merge** na segunda chamada. Por este motivo, é definido como um *lemma* e não como uma pós condição, da seguinte forma:

```
lemma merge2: forall x1 x2 :int, r1 r2 :list int.
                x1 <= x2 -> merge (Cons x1 r1) (Cons x2 r2) =
                            Cons x1 (merge r1 (Cons x2 r2))
```

Listing 3.23: Código Why3 do *lemma* **merge2**.

A prova deste *lemma* foi extremamente simples, tendo sido apenas necessário recorrer ao ***solver* CVC4**.

3.3.5 *Definição e verificação da função Mergesort*

Finalmente, fica a faltar a definição da função principal **mergesort** propriamente dita. Esta recebe uma lista, cujo objetivo é retornar a mesma mas totalmente ordenada. Para isso, é dividida em duas e, posteriormente, estas são fundidas numa só, de forma ordenada. Esta definição parece simples, porém esta primeira tentativa (abaixo) foi rejeitada:

```
Fixpoint mergesort (l: list nat) :  list nat :=
   let (l1,l2) := split l in
   merge (mergesort l1) (mergesort l2).
```

Como referido anteriormente, a *keyword* **Fixpoint** exige que a função seja recursivamente estrutural num argumento específico. Mais uma vez, o problema é que o Coq não tem maneira de saber que **l1** e **l2** são

menores do que **l**. Considerando o comportamento da divisão em listas vazias ou de um só elemento, acontece que esta propriedade não se verifica. Assim sendo, foi escrita a seguinte definição:

```
Fixpoint mergesort (l: list nat) :  list nat :=
   match l with
   | []    ⇒ []
   | [x]   ⇒ [x]
   | _     ⇒ let (l1,l2) := split l
             in merge (mergesort l1) (mergesort l2).
```

Ainda assim, não foi a definição final, visto que apesar de a função efetivamente terminar, o Coq ainda não permite escrevê-la através da definição **Fixpoint**. A melhor forma de resolver esta questão é utilizando a definição **measure** já utilizada, para demonstrar de forma explicita que, de facto, a função chama-se a si própria com argumentos mais pequenos. Posto isto, utiliza-se a *keyword* **Function**, semelhante a **Fixpoint**, no entanto permite especificar uma anotação **measure** sobre o argumento da função:

```
Function mergesort (l: list nat) {measure length l} : list nat
:=
  match l with
  | [] ⇒ []
  | [x] ⇒ [x]
  | _ ⇒ let (l1,l2) := split l
        in merge (mergesort l1) (mergesort l2)
  end.
```

Esta anotação **measure length l** declara que o comprimento da função **l** serve como uma medida decrescente. Finalizada a definição da função principal **mergesort**, resta apenas escrevê-la com a linguagem de programas do Why3:

```
let rec function mergesort (l :list int) : list int
  variant { length l }
  ensures { sorted result } (* mergesort_sorts *)
  ensures { permut l result } (* mergesort_perm *)
=
  match l with
  | Nil -> Nil
  | Cons x Nil -> Cons x Nil
  | _ ->   let (l1,l2) = split l
           in merge (mergesort l1) (mergesort l2)
  end
```

Listing 3.24: Código Why3 da função **mergesort**.

A função **mergesort** exige a presença de uma anotação do tipo **variant**, neste caso **length l**, para garantir que existe um argumento que é decrementado a cada chamada recursiva, e portanto que a função termina. As restante anotações resultam dos *lemmas* necessários para a prova de que este algoritmo é um algoritmo de ordenação correto. Ou seja, que o resultado retornado seja uma lista ordenada e que é uma permutação da lista original.

Para isso, é necessário definir o *lemma* **mergesort_sorts** (não demonstrado), que afirma que o resultado de **mergesort** é uma lista ordenada e o *lemma* **mergesort_perm** (não demonstrado) que prova que a lista retornada resulta de uma permutação da original. Estas anotações, em Why3, geram então CV que foram provadas automaticamente e sem problemas.

Utilizando o predicado **is_a_sorting_algorithm** definido nos exemplos anteriores e o *lemma* que prova a sua correção:

```
predicate is_a_sorting_algorithm (f: list int -> list int) =
  forall al :list int. permut al (f al) /\ sorted (f al)

lemma mergesort_correct: is_a_sorting_algorithm mergesort
```

Listing 3.25: Código Why3 do predicado **is_a_sorting_algorithm** e respetivo *lemma*.

O *lemma* **mergesort_correct** totalmente provado conclui assim a prova de que este algoritmo é um algoritmo de ordenação correto.

4

CONCLUSÃO

O principal objetivo desta dissertação foi perceber até que ponto é possível o Why3 formalizar alguns algoritmos funcionais e respetivas provas, definidas em Coq.

Convertemos três algoritmos diferentes de onde é possível retirar diferentes conclusões de cada um deles. Analisando estes exemplos, de forma genérica, é possível perceber que o Why3 é afinal bastante capaz de converter as formalizações Coq para a sua linguagem, principalmente para a sua linguagem de programas, **WhyML**, onde é também notório que as provas são extremamente mais simples, conseguidas, na sua grande maioria, através apenas dos *Solvers* automáticos. Comparando as duas linguagens do Why3, a linguagem de programas é efetivamente mais interessante que a linguagem lógica.

Na lógica, recorrendo às transformações, é possível efetuar provas indutivas, no entanto apenas se estas provas utilizarem a indução estrutural. Para lidar com recursividade não estrutural, o Why3 exigiria novas transformações de prova que só os *developers* conseguem alterar. Posto isto, é mais natural, no Why3, realizar as provas na linguagem de programas, dado que nesta linguagem a recursividade segue a estrutura da definição das funções, evitando assim a necessidade da definição de princípios de indução. Isto só se aplica a *lemmas* que podem ser escritos como contratos, caso contrário não são anotações, ou seja, não geram CV na prova da respetiva função e por isso a sua prova apenas pode ser verificada recorrendo às conhecidas transformações.

O Why3 tem também funcionalidades muito interessantes, tanto na lógica como em programas, tais como o *smoke detector* que permitem às provas ter uma maior fator confiança, garantindo que não existem quaisquer inconsistências no código.

Um facto interessante, embora não muito explorado nesta dissertação, é que, como vimos na definição da função **selsort'** (**selection sort**), é possível fazer a ligação entre as duas linguagens do Why3, sendo que utilizando a definição **ghost**, que pertence ao nível lógico, é permitido anotar funções, funcionalidade essa que pertence ao nível de programas (ver página 41).

Concluiu-se também, a partir do exemplo **merge sort**, que, ao contrário do que acontece em Coq, não é possível em Why3 definir os próprios princípios de indução. Ainda assim, esta limitação é natural, uma vez que o objetivo deste é tornar as provas cada vez mais automáticas e menos manuais (ver página 47).

Apesar de o Coq não ser o foco deste trabalho, foi também necessário aprender um pouco mais sobre este e constatou-se que este tem formas distintas de definir funções que, do ponto de vista da terminação, correspondem ao nível lógico e ao nível de programas do Why3.

Em suma, o Why3 revelou ser uma ferramenta com uma linguagem verdadeiramente capaz, com uma facilidade evidente nas provas, sendo que estas são na sua grande maioria totalmente automáticas, sem qualquer tipo de intervenção manual.

Como trabalho futuro seria interessante explorar o algoritmo seguinte do livro da "Software Foundations". Este é sobre a formalização de árvores binárias de procura, onde cada folha é representada pelo par **(chave, valor)**. Seria também pertinente explorar a utilização de *lemma functions*.

BIBLIOGRAFIA

Qinxiang Cao Andrew W. Appel. *Verifiable C*. Software Foundations series, volume 5. Electronic textbook, 2020. Version 0.9.7. http://www.cis.upenn.edu/ bcpierce/sf.

Andrew W. Appel. *Verified Functional Algorithms*. Software Foundations series, volume 3. Electronic textbook, 2020. Version 1.4. http://www.cis.upenn.edu/ bcpierce/sf.

Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Jean-Christophe Filliâtre, Eduardo Gimenez, Hugo Herbelin, Gerard Huet, Cesar Munoz, Chetan Murthy, et al. *The Coq Proof Assistant Reference Manual*, 2020. Version 8.13.0. https://coq.github.io/doc/v8.13/refman.

Yves Bertot, Georges Gonthier, Sidi Ould Biha, and Ioana Pasca. Canonical big operators. In *TPHOLs, Montreal, Canada*, volume 5170 of LNCS. Springer, 2008.

François Bobot, Jean-Christophe Filliâtre, Claude Marché, Guillaume Melquiond, and Andrei Paskevich. *Why3 Documentation Release 1.3.3*, 2020. Version 1.3.3. https://why3.lri.fr/manual.pdf.

Ran Chen, Cyril Cohen, Jean Jacques Lévy, Stephan Merz, and Laurent Théry. Formal proofs of tarjan's algorithm in why3, COQ, and isabelle. *arXiv*, pages 1–23, 2018. ISSN 23318422.

Jean-Christophe Filliâtre and Andrei Paskevich. Why3 - where programs meet provers. In *22nd European Symposium on Programming, ESOP 2013*, pages 125–128, 2013. doi: 10.1007/978-3-642-37036-6_8. URL http://dx.doi.org/10.1007/978-3-642-37036-6_8.

Georges Gonthier. A computer-checked proof of the four colour theorem, 2005.

Leonidas Lampropoulos and Benjamin C. Pierce. *QuickChick: Property-Based Testing in Coq*. Software Foundations series, volume 4. Electronic textbook, 2018. Version 1.0. http://www.cis.upenn.edu/ bcpierce/sf.

Benjamin C Pierce, Chris Casinghino, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, and Brent Yorgey. *Software Foundations*. 2012.

Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, Andrew Tolmach, and Brent Yorgey. *Programming Language Foundations*. Software Foundations series, volume 2. Electronic textbook, May 2018a. Version 5.5. http://www.cis.upenn.edu/ bcpierce/sf.

Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, and Brent Yorgey. *Logical Foundations*. Software Foundations series, volume 1. Electronic textbook, May 2018b. Version 5.5. http://www.cis.upenn.edu/ bcpierce/sf.

ANEXOS

A

FORMALIZAÇÃO DO ALGORITMO INSERTION SORT

```
theory InsertionSortSorted

use int.Int
use list.List
use list.Permut

function insert (i :int) (l :list int) : list int =
  match l with
  | Nil -> Cons i Nil
  | Cons h t -> if i <= h then Cons i (Cons h t)
                         else Cons h (insert i t)
 end

function sort (l :list int) : list int =
  match l with
  | Nil  -> Nil
  | Cons h t -> insert h (sort t)
  end

inductive sorted (list int) =
  | sorted_nil  : sorted Nil
  | sorted_1    : forall x :int. sorted (Cons x Nil)
  | sorted_cons : forall x y :int, l :list int.
    x <= y -> sorted (Cons y l) -> sorted (Cons x (Cons y l))

predicate is_a_sorting_algorithm (f: list int -> list int) =
  forall al :list int. permut al (f al) /\ sorted (f al)

lemma insert_sorted: forall a :int, l :list int.
                       sorted l -> sorted (insert a l)

lemma sort_sorted: forall l :list int.
                     sorted (sort l)
```

```
lemma insert_perm: forall x :int, l :list int.
                     permut (Cons x l) (insert x l)

lemma sort_perm: forall l :list int.
                   permut l (sort l)

lemma insertion_sort_correct: is_a_sorting_algorithm sort

lemma insertion_sort_correct2: forall al :list int.
                                 permut al ( sort al ) /\ sorted ( sort al )

end



theory InsertionSortSorted'

use int.Int
use list.List
use list.Nth
use option.Option

function insert (i :int) (l :list int) : list int =
  match l with
  | Nil -> Cons i Nil
  | Cons h t -> if i <= h then Cons i (Cons h t)
                          else Cons h (insert i t)
  end

function sort (l :list int) : list int =
  match l with
 | Nil  -> Nil
  | Cons h t -> insert h (sort t)
  end

inductive sorted (list int) =
  | sorted_nil  : sorted Nil
  | sorted_1    : forall x :int. sorted (Cons x Nil)
  | sorted_cons : forall x y :int, l :list int.
    x <= y -> sorted (Cons y l) -> sorted (Cons x (Cons y l))

predicate sorted' (al: list int) =
  forall i j iv jv :int.
  i < j ->
  nth i al = Some iv ->
  nth j al = Some jv ->
  iv <= jv
```

```
lemma sorted_sorted': forall al :list int.
                        sorted al -> sorted' al

lemma sorted'_sorted: forall al :list int.
                        sorted' al -> sorted al

lemma nth_error_insert: forall a i iv :int, l :list int.
                          nth i (insert a l) = Some iv ->
                          a = iv \/ (exists i': int. nth i' l = Some iv)

lemma insert_sorted': forall a :int, l :list int.
                        sorted' l -> sorted' (insert a l)

lemma sort_sorted': forall l :list int.
                      sorted' (sort l)

end


theory InsertionSortSorted''

use int.Int
use list.List
use list.Length
use list.NthNoOpt

predicate sorted'' (al: list int) =
  forall i j :int.
  i < j < length al -> nth i al <= nth j al

end
```

Listing A.1: Formalização do algoritmo **insertion sort** na linguagem lógica do Why3.

```
module InsertionSort

  use  int.Int
  use  list.List
  use  list.Mem
  use  list.SortedInt
  use  list.Permut
  use  list.Length

let rec function insert (i :int) (l :list int) : list int
    requires { sorted l }
```

```
    ensures { sorted result } (* insert_sorted *)
    ensures { permut result (Cons i l) } (* insert_perm *)
    (* variant { length l } *)
    =
    match l with
    | Nil -> Cons i Nil
    | Cons h t -> if i <= h then Cons i l
                            else Cons h (insert i t)
    end

let rec function sort (l :list int) : list int
    ensures { sorted result } (* sort_sorted *)
    ensures { permut result l } (* sort_perm *)
    (* variant { length l } *)
    =
    match l with
    | Nil -> Nil
    | Cons h t -> insert h ( sort t )
    end

predicate is_a_sorting_algorithm (f: list int -> list int) =
    forall al :list int. permut al (f al) /\ sorted (f al)

goal insertion_sort_correct: is_a_sorting_algorithm sort

end
```

Listing A.2: Formalização do algoritmo **insertion sort** na linguagem de programas do Why3.

B

FORMALIZAÇÃO DO ALGORITMO SELECTION SORT

```
theory SelectionSortSelect

use int.Int
use list.List
use list.Length
use list.Permut
use list.Mem

(* select x l is (y, l'), where y is the smallest element
   of x :: l, and l' is all the remaining elements of x :: l
   in their original order. *)

function select (x :int) (l :list int) : (int, list int)
=
  match l with
  | Nil -> (x, Nil)
  | Cons h t -> if x <= h
                then let (j, l') = select x t
                     in (j, Cons h l')
                else let (j, l') = select h t
                     in (j, Cons x l')
  end

lemma select_perm: forall l r :list int, x y :int.
                     let (y, r) = select x l in permut (Cons x l) (Cons y r)

lemma select_rest_length: forall l :list int, x :int.
                            let (_, r) = select x l in length l = length r

lemma select_fst_leq: forall al :list int, x :int.
                        let (y, bl) = select x al in y <= x

predicate le_all (x :int) (xs :list int) =
  forall y :int. mem y xs -> x <= y
```

```
lemma select_smallest: forall al :list int, x :int.
                        let (y, bl) = select x al in le_all y bl

lemma select_in: forall al :list int, x :int.
                   let (y, bl) = select x al in mem y (Cons x al)

end



theory SelectionSortSelSort

use int.Int
use list.List
use list.Length
use list.SortedInt
use list.Permut
use list.Mem

use SelectionSortSelect

type nat = Z | S nat

function natLength (l :list int) : nat =
  match l with
  | Nil -> Z
  | Cons _ t -> S (natLength t)
  end

function selsort (l :list int) (n :nat) : list int =
      match l,n with
      | _, Z   -> Nil
      | Nil, _ -> Nil
      | Cons x r, S n' -> let (y, r') = select x r
                          in Cons y (selsort r' n')
      end

(* function selsort (l :list int) (n :nat) : list int =
  match n with
  | Z -> Nil
  | S n' ->  match l with
             | Nil -> Nil
             | Cons x r -> let (y, r') = select x r
                           in Cons y (selsort r' n')
             end
  end *)
```

```
function selection_sort (l :list int) : list int =
  let n = natLength l in selsort l n

predicate is_a_sorting_algorithm (f :list int -> list int) =
  forall al :list int. permut al (f al) /\ sorted (f al)

lemma selsort_perm: forall l :list int, n :nat.
                      n = natLength l -> permut l (selsort l n)

lemma selection_sort_perm: forall l :list int.
                             permut l (selection_sort l)

(* predicate le_all (x :int) (xs :list int) =
  forall y :int. mem y xs -> x <= y *)

lemma cons_of_small_maintains_sort: forall n :nat, y :int, bl :list int.
                                      n = natLength bl ->
                                      le_all y bl ->
                                      sorted (selsort bl n) ->
                                      sorted (Cons y (selsort    bl n))

lemma selsort_sorted: forall n :nat, al :list int.
                        n = natLength al -> sorted (selsort al n)

lemma selection_sort_sorted: forall al :list int.
                               sorted (selection_sort al)

lemma selection_sort_is_correct: is_a_sorting_algorithm selection_sort


end



theory SelectionSortSelSort'

use int.Int
use list.List
use list.Length
use list.Permut

use SelectionSortSelect

let rec ghost function selsort' (l :list int) : list int
  variant {length l}
=
```

```
  match l with
  | Nil -> Nil
  | Cons x r -> let (y, r') = select x r
                in Cons y (selsort' r')
  end

end
```

Listing B.1: Formalização do algoritmo **selection sort** na linguagem lógica do Why3.

```
module SelectionSort

use int.Int
use list.List
use list.Length
use list.SortedInt
use list.Permut
use list.Mem

predicate le_all (x :int) (xs :list int) = forall y :int . mem y xs -> x <= y

let rec function select (x :int) (l :list int) : (int, list int)
  (* variant { length l } *)
  ensures { let (y,l') = result in permut (Cons y l') (Cons x l) } (*
select_perm *)
  ensures { let (_,r) = result in length l = length r } (* select_rest_length
 *)
  ensures { let (y,_) = result in y <= x } (* select_fst_leq *)
  ensures { let (y,bl) = result in le_all y bl } (* select_smallest *)
  ensures { let (y,_) = result in mem y (Cons x l)  } (* select_in *)
=
  match l with
  | Nil -> (x, Nil)
  | Cons h t ->
    if x <= h
    then let (j, l') = select x t
         in (j, Cons h l')
    else let (j, l') = select h t
         in (j, Cons x l')
  end


let rec function selsort (l :list int) : list int
  variant { length l }
```

```
  ensures { permut result l} (* selsort_perm *)
  ensures { forall y :int. le_all y l -> sorted result -> sorted (Cons y
result) } (* cons_of_small_maintains_sort *)
ensures { sorted result } (* selsort_sorted *)
=
  match l with
  | Nil -> Nil
  | Cons x r -> let (y, r') = select x r
           in Cons y (selsort r')
  end

let rec function selection_sort (l :list int) : list int
  ensures { permut result l} (* selection_sort_perm *)
  ensures { sorted result } (* selection_sort_sorted *)
=
  selsort l

predicate is_a_sorting_algorithm (f :list int -> list int) =
    forall al :list int. permut al (f al) /\ sorted (f al)

goal selection_sort_is_correct: is_a_sorting_algorithm selection_sort

(* let rec ghost function selsort2 (l :list int) (n :int) : list int
  requires { n >= 0 } (* Porque [U+FFFD] um inteiro e n[U+FFFD]o um natural *)

  variant { n }
=
  match l with
  | Nil -> Nil
  | Cons x r -> if n = 0 then Nil (* ran out of fuel *)
                else let (y, r') = select x r
                     in Cons y (selsort2 r' (n-1))
  end *)

(* let rec function selection_sort2 (l :list int) : list int =
  selsort2 l (length l) *)

end
```

Listing B.2: Formalização do algoritmo **selection sort** na linguagem de programas do Why3.

C

FORMALIZAÇÃO DO ALGORITMO MERGE SORT

```
module MergeSort

use int.Int
use list.List
use list.Length
use list.Permut
use list.Append
use list.SortedInt

let rec function split (l :list int) : (list int, list int)
  ensures { let (l1,l2) = result
            in length l1 <= length l /\ length l2 <= length l } (*split_len*)
  ensures { let (l1,l2) = result
            in length l >= 2
            -> length l1 < length l /\ length l2 < length l } (* split_len *)
  ensures { let (l1,l2) = result in permut l (l1 ++ l2) } (* split_perm *)
=
  match l with
  | Nil -> (Nil, Nil)
  | Cons x Nil -> (Cons x Nil, Nil)
  | Cons x1 (Cons x2 l') -> let (l1, l2) = split l'
                           in (Cons x1 l1, Cons x2 l2)
  end

let rec function merge (l1 l2 :list int) : list int
  variant { length (l1 ++ l2) }
  ensures { l1 = Nil -> result = l2 } (* merge_nil_l *)
  ensures { forall x x1 x2 :int, l1' l2' :list int. l1 = (Cons x1 l1')
            -> l2 = (Cons x2 l2') -> x <= x1 -> x <= x2 -> sorted result
            -> sorted (Cons x result) } (* sorted_merge1 *)
  ensures { sorted l1 -> sorted l2 -> sorted result } (* sorted_merge *)
  ensures { permut (l1 ++ l2) result } (* merge_perm *)
=
  match l1, l2 with
```

```
  | Nil, _ -> l2
  | _, Nil -> l1
  | (Cons a1 l1'), (Cons a2 l2') -> if a1 <= a2
                                    then (Cons a1 (merge l1' l2))
                                    else (Cons a2 (merge l1 l2'))
  end


lemma merge2: forall x1 x2 :int, r1 r2 :list int.
                x1 <= x2 -> merge (Cons x1 r1) (Cons x2 r2) =
                Cons x1 (merge r1 (Cons x2 r2))


let rec function mergesort (l :list int) : list int
  variant { length l }
  ensures { sorted result } (* mergesort_sorts *)
  ensures { permut l result } (* mergesort_perm *)
=
  match l with
  | Nil         -> Nil
  | Cons x Nil  -> Cons x Nil
  | _           -> let (l1,l2) = split l
                   in merge (mergesort l1) (mergesort l2)
  end

predicate is_a_sorting_algorithm (f: list int -> list int) =
  forall al :list int. permut al (f al) /\ sorted (f al)

lemma mergesort_correct: is_a_sorting_algorithm mergesort

end
```

Listing C.1: Formalização do algoritmo **merge sort** na linguagem de programas do Why3.