Lógica animada: conversão funcional correta para Forma Normal Conjuntiva*

Pedro Barroso, Mário Pereira e António Ravara

NOVA LINCS & DI-FCT – Universidade Nova de Lisboa, Portugal

Resumo Neste artigo apresenta-se uma abordagem à implementação formalmente verificada de algoritmos clássicos de Lógica Computacional. Escolhe-se como ferramenta de prova a plataforma Why3 que permite implementações próximas das definições matemáticas, assim como um elevado grau de automação no processo de verificação. Como prova de conceito, utiliza-se o algoritmo de conversão de fórmulas proposicionais para forma normal conjuntiva. Aplica-se a proposta sobre duas variantes deste algoritmo: uma em estilo direto e outra com uma estrutura de pilha explícita no código. Sendo ambas as versões de primeira ordem, o Why3 processa as provas naturalmente.

1 Introdução

Motivação. Unidades curriculares centrais em Engenharia Informática, como por exemplo Lógica Computacional, têm como objetivo apresentar conteúdo fundamental para a formação dos estudantes. Para fortalecer a ligação do conteúdo abordado nestas unidades curriculares com a prática de desenvolvimento de código correto, é relevante relacionar o conteúdo matemático a implantações corretas, claras e executáveis.

Este artigo insere-se no projeto `FACTOR` [2], que visa promover o uso do OCaml [10] e de práticas de desenvolvimento de código correto na comunidade académica de expressão portuguesa. Concretamente, o projeto tem como objetivos a implementação funcional de algoritmos clássicos de Lógica Computacional e Linguagens Formais, a realização de provas de correção das mesmas e a execução passo-a-passo para os ajudar a compreender a partir de exemplos.

O algoritmo de conversão de fórmulas proposicionais para Forma Normal Conjuntiva (FNC)[1] é frequentemente apresentado formalmente, com definições matemáticas rigorosas que, por vezes, são difíceis de ler [5,7,11], ou informalmente, destinados à Ciência da Computação, mas com definições textuais em pseudo-código não executável [4,9]. A implementação de algoritmos desta natureza é uma peça fundamental para a aprendizagem e compreensão dos mesmos.

---

* Este trabalho é financiado pela Fundação Tezos através do projeto FACTOR e, por fundos nacionais, através da FCT – Fundação para a Ciência e a Tecnologia, I.P., no âmbito do NOVA LINCS através do projeto UID/CEC/04516/2019.

[1] Uma fórmula está em FNC se é uma conjunção de cláusulas, onde uma cláusula é uma disjunção de literais, sendo um literal um símbolo proposicional ou a sua negação.

Linguagens como o OCaml permitem implementações muito próximas das definições matemáticas, ajudando o estudo por serem executáveis. Além disso as prova de correção são mais simples do que as das implementações imperativas.

Contribuições. Como prova de conceito, faz-se a implementação e prova de correção do algoritmo referido em Why3 [6,15], uma plataforma para verificação dedutiva de programas. Why3 fornece uma linguagem de primeira ordem com tipos polimórficos, *pattern matching* e predicados indutivos, chamada WhyML, oferecendo ainda um mecanismo de extração de código OCaml certificado e suporte para provadores de teoremas de terceiros.

Para no futuro suportar a execução passo-a-passo do algoritmo, uma importante funcionalidade para ajudar os estudantes a perceber as definições, implementa-se também uma versão em *Continuation-Passing Style* (CPS) [14] e via desfuncionalização obtém-se um avaliador, uma versão próxima a uma máquina abstrata de primeira ordem [3]. Devido ao limitado suporte do Why3 à ordem superior não foi possível fechar a prova de correção. Esta limitação levou ao desenvolvimento de uma implementação com estrutura de pilha explicita no código, mas desta vez em primeira ordem, implementação que resultou de uma transformação mecânica a partir da versão CPS. Esta versão foi naturalmente provada correta pelo Why3.

Em suma, este artigo apresenta material pedagógico de apoio ao ensino de algoritmos clássicos de Lógica Computacional, nomeadamente, duas implementações, formalmente verificadas em Why3, a partir de uma apresentação como função recursiva do algoritmo de conversão para FNC: a primeira em estilo direto e a segunda com estrutura de pilha explicita no código.

O código Why3 apresentado neste artigo pode ser encontrado no repositório público https://bitbucket.org/laforetbarroso/cnfwhy3.

2 Apresentação funcional do algoritmo

Descrição. Designa-se T ao algoritmo que converte qualquer fórmula de Lógica Proposicional para FNC. Uma fórmula proposicional ϕ é um elemento do conjunto G_p, definido como: $G_p \triangleq\ \phi ::= T \mid p \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \phi \rightarrow \phi$.

A função T produz uma formula sem o conectivo de implicação, portanto define-se um conjunto H_p como um sub-conjunto de G_p sem implicações. Tem-se então que T: $G_p \rightarrow H_p$, onde:

$$\mathrm{T}(\phi) = \mathrm{CNFC}(\mathrm{NNFC}(\mathrm{Impl\_Free}(\phi)))$$

O algoritmo compõe três funções: `Impl_Free` responsável por eliminar as implicações; `NNFC` responsável pela conversão para Forma Normal de Negação (FNN)[2]; `CNFC` responsável pela conversão de FNN para FNC.

Implementação dos conjuntos. Para representar o conjunto G_p define-se o tipo `formula` que declara variáveis (`FVar`), constantes (`FConst`), conjunções (`FAnd`), disjunções (`FOr`), implicações (`FImpl`) e negações (`FNeg`):

[2] Uma fórmula está na FNN, se só as suas sub-fórmulas que são literais estão negadas.

```
type formula =
  | FVar ident
  | FConst bool
  | FAnd formula formula
  | FOr formula formula
  | FImpl formula formula
  | FNeg formula
```

Para representar o conjunto H_p define-se o tipo `formula_wi`, semelhante ao anterior mas sem o construtor de implicação.

Implementação das funções. A função `Impl_Free` elimina todas as implicações. É definida recursivamente nos casos do tipo `formula` e homomórfica, excepto no caso da implicação, onde utiliza a lei de Lógica Proposicional $\mathtt{A} \rightarrow \mathtt{B} \equiv \neg\mathtt{A} \vee \mathtt{B}$. A implementação da função converte os construtores do tipo `formula` para os do tipo `formula_wi` e efetua chamadas recursivas sobre os argumentos:

```
let rec impl_free (phi: formula) : formula_wi
= match phi with
  | FNeg phi1 → FNeg_wi (impl_free phi1)
  | FOr phi1 phi2 → FOr_wi (impl_free phi1) (impl_free phi2)
  | FAnd phi1 phi2 → FAnd_wi (impl_free phi1) (impl_free phi2)
  | FImpl phi1 phi2 → FOr_wi (FNeg_wi (impl_free phi1)) (impl_free phi2)
  | FConst phi → FConst_wi phi
  | FVar phi → FVar_wi phi
end
```

A função `NNFC` converte a fórmula para a FNN. É definida recursivamente em combinações de construtores: as duplas negações são eliminadas aplicando a lei de Lógica Proposicional $\neg\neg\mathtt{A} \equiv \mathtt{A}$ e, usando as leis de De Morgan, as negações de conjunções passam a disjunções de negações e as negações de disjunções passam a conjunções de negações. O código da função é o seguinte:

```
let rec nnfc (phi: formula_wi) : formula_wi
= match phi with
  | FNeg_wi (FNeg_wi phi1) → nnfc phi1
  | FNeg_wi (FAnd_wi phi1 phi2) → FOr_wi (nnfc (FNeg_wi phi1))
    (nnfc (FNeg_wi phi2))
  | FNeg_wi (FOr_wi phi1 phi2) → FAnd_wi (nnfc (FNeg_wi phi1))
    (nnfc (FNeg_wi phi2))
  | FOr_wi phi1 phi2 → FOr_wi (nnfc phi1) (nnfc phi2)
  | FAnd_wi phi1 phi2 → FAnd_wi (nnfc phi1) (nnfc phi2)
  | phi → phi
end
```

A função `CNFC` converte uma fórmula em FNN para FNC, sendo homomórfica excepto no caso da disjunção, onde efetua a distribuição da disjunção pela conjunção chamando a função auxiliar `distr`.

```
let rec cnfc (phi: formula_wi) : formula_wi
= match phi with
  | FOr_wi phi1 phi2 → distr (cnfc phi1) (cnfc phi2)
  | FAnd_wi phi1 phi2 → FAnd_wi (cnfc phi1) (cnfc phi2)
```

```
  | phi → phi
end
```

A função `distr` por sua vez tira partido da lei de Lógica Proposicional:

$$\mathtt{A} \vee (\mathtt{B} \wedge \mathtt{C}) \equiv (\mathtt{A} \vee \mathtt{B}) \wedge (\mathtt{A} \vee \mathtt{C})$$

O código da função `distr` é o seguinte:

```
let rec distr (phi1 phi2: formula_wi) : formula_wi
= match phi1, phi2 with
  | FAnd_wi phi11 phi12, phi2 → FAnd_wi (distr phi11 phi2)
    (distr phi12 phi2)
  | phi1, FAnd_wi phi21 phi22 → FAnd_wi (distr phi1 phi21)
    (distr phi1 phi22)
  | phi1,phi2 → FOr_wi phi1 phi2
end
```

Finalmente, o código da função T que compõe todas estas funções é o seguinte:

```
let t (phi: formula) : formula_wi
= cnfc(nnfc(impl_free phi))
```

A partir desta implementação é possível a extração de código OCaml. Em ambas as implementações ressalta a semelhança com as definições matemáticas, demonstrando assim que o OCaml é uma linguagem adequada para a apresentação destes algoritmos, providenciando definições executáveis sem sacrifício de rigor ou clareza.

3 Como obter a correção

O algoritmo T, como apresentado anteriormente, é uma composição de três funções, sendo a correção do algoritmo resultado dos critérios de correção de cada uma dessas três funções.

Critérios. Para todas as funções o critério de correção básico é que o seu resultado deve ser uma fórmula equivalente à formula argumento. Além disso requer-se que: o resultado da função `Impl_Free` não deve conter conetivos de implicação; a fórmula de entrada e resultado da função `NNFC` não devem conter conetivos de implicação e o resultado tem que estar em FNN; a fórmula de entrada e resultado da função `CNFC` não devem conter conetivos de implicação e tem que estar em FNN e o resultado tem que estar em FNC.

De notar que os critérios de correção de uma função são propagados para as funções seguintes, garantindo que uma função não viola as pós-condições já assegurados pelas funções previamente executadas.

Semântica das fórmulas. Como o critério básico de correção é a equivalência de fórmulas, é preciso uma função para as avaliar (ou seja, de valoração):

```
type valuation = ident → bool

function eval (v: valuation) (f: formula) : bool
= match f with
```

```
  | FVar x      → v x
  | FConst b    → b
  | FAnd f1 f2  → eval v f1 ∧ eval v f2
  | FOr f1 f2   → eval v f1 || eval v f2
  | FImpl f1 f2 → eval v f1 → eval v f2
  | FNeg f      → not (eval v f)
end
```

Esta função recebe um argumento de tipo `valuation` que atribui um valor do tipo `bool`[3] a cada variável da fórmula, recebe a `fórmula` a ser valorada e retorna um valor do tipo `bool`. Para os construtores base `FVar` e `FConst`, apenas é retornado o valor booleano da variável e o valor da constante, respetivamente. Para os restantes casos construtores são valoradas recursivamente as fórmulas associadas e o resultado traduzido para a operação booleana correspondente do WhyML. A função de valoração para o tipo de fórmulas `formula_wi` é semelhante.

4 Prova de correção

A prova da correção da implementação consiste em mostrar que cada função respeita os critérios de correção definidos na secção anterior.

Palavras-chave. Em WhyML as palavras-chaves `ensures` correspondem à indicação de pós-condições, consequentemente, à prova de correção parcial. A terminação e prova de correção total são asseguradas com a palavra-chave `variant`.

Correção da função Impl_Free. A ausência de conetivos de implicação é assegurado pelo tipo de retorno da função (`formula_wi`); a equivalência das fórmulas é assegurada usando as funções de valoração de fórmulas e usa-se a fórmula de entrada como medida para garantir a terminação.

```
let rec function impl_free (phi: formula) : formula_wi
  ensures { forall v. eval v phi = eval_wi v result }
  variant { phi }
= ...
```

Correção da função NNFC. A ausência de conetivos de implicação nas fórmulas de entrada e saída é assegurada pelo tipo `formula_wi`. Para provar que o resultado está na FNN, submete-se à prova o predicado de boa formação `wf_negations_of_literals`. Este estabelece que as sub-fórmulas do construtor `FNeg_wi` não podem conter construtores `FOr_wi`, `FAnd_wi` ou `FNeg_wi`:

```
predicate wf_negations_of_literals (f: formula_wi)
= match f with
  | FNeg_wi f → (forall f1 f2. f ≠ FOr_wi f1 f2 ∧ f ≠ FAnd_wi f1 f2 ∧
    f ≠ FNeg_wi f1) ∧ wf_negations_of_literals f
  | FOr_wi f1 f2 | FAnd_wi f1 f2 → wf_negations_of_literals f1 ∧
    wf_negations_of_literals f2
  | FVar_wi _ → true
  | FConst_wi _ → true
end
```

[3] bool é o tipo booleano do WhyML

Nesta prova não é possível usar a própria fórmula como medida de terminação visto que no caso da distribuição da negação pela conjunção ou disjunção são adicionados construtores à cabeça, impossibilitando o critério indutivo estrutural. Criou-se então, uma função que conta o número de construtores de uma fórmula. No entanto, para ser possível usá-la como medida de terminação é preciso assegurar, com um lema, que o número de construtores nunca é negativo.

Com o predicado e medida de terminação definidos é possível fechar a prova de correção da função `NNFC`, sendo o código submetido à prova o seguinte:

```
let rec nnfc (phi: formula_wi) : formula_wi
  ensures { forall v. eval_wi v phi = eval_wi v result }
  ensures { wf_negations_of_literals result }
  variant { size phi }
= ...
```

Prova da função CNFC. O critério básico de equivalência é mais uma vez assegurado usando a função de valoração de fórmulas. Para assegurar que uma determinada fórmula está na FNC introduzem-se os predicados de boa formação `wf_conjunctions_of_disjunctions` e `wf_disjunctions`. Estes garantem que após uma disjunção não há nenhuma conjunção:

```
predicate wf_conjunctions_of_disjunctions (f: formula_wi)
= match f with
  | FAnd_wi f1 f2 → wf_conjunctions_of_disjunctions f1 ∧
    wf_conjunctions_of_disjunctions f2
  | FOr_wi f1 f2 → wf_disjunctions f1 ∧ wf_disjunctions f2
  | FConst_wi _ → true
  | FVar_wi _ → true
  | FNeg_wi f1 → wf_conjunctions_of_disjunctions f1
end

predicate wf_disjunctions (f: formula_wi)
= match f with
  | FAnd_wi _ _ → false
  | FOr_wi f1 f2 → wf_disjunctions f1 ∧ wf_disjunctions f2
  | FConst_wi _ → true
  | FVar_wi _ → true
  | FNeg_wi f1 → wf_disjunctions f1
end
```

Finalmente, adicionam-se os predicados `wf_conjunctions_of_disjunctions` e `wf_negations_of_literals` às pós-condições para assegurar que o resultado está na FNN e FNC, respetivamente; para assegurar que a fórmula de entrada está na FNN, adiciona-se também o predicado `wf_negations_of_literals` às pré-condições. O código submetido à prova de correção é o seguinte:

```
let rec cnfc (phi: formula_wi)
  requires{ wf_negations_of_literals phi }
  ensures{ forall v. eval_wi v phi = eval_wi v result }
  ensures{ wf_negations_of_literals result }
  ensures{ wf_conjunctions_of_disjunctions result }
```

```
  variant { phi }
= ...
```

Sendo `distr` uma função auxiliar da função `CNFC`, é preciso também provar a correção da mesma. Nesta função é necessário garantir os mesmos critérios da função `CNFC`, mas por se tratar da distribuição das disjunções pelas conjunções, é preciso adicionalmente assegurar que as fórmulas de entrada estão na FNC, o que se obtém adicionando os predicados `wf_conjunctions_of_disjunctions` e `wf_negations_of_literals` às pré-condições:

```
let rec distr (phi1 phi2: formula_wi)
  requires{ wf_negations_of_literals phi1 }
  requires{ wf_negations_of_literals phi2 }
  requires{ wf_conjunctions_of_disjunctions phi1 }
  requires{ wf_conjunctions_of_disjunctions phi2 }
  ensures { forall v. eval_wi v (FOr_wi phi1 phi2) = eval_wi v result }
  ensures { wf_negations_of_literals result }
  ensures { wf_conjunctions_of_disjunctions result }
  variant { size phi1 + size phi2 }
= ...
```

No entanto, não se consegue provar que uma disjunção de duas fórmulas na FNC é efetivamente uma fórmula na FNC, isto porque é necessário assegurar que numa disjunção de duas fórmulas na FNC, as fórmulas não contêm o construtor `FAnd_wi`. Para tal, reforça-se a prova com um lema auxiliar:

```
lemma aux: forall x. wf_conjunctions_of_disjunctions x ∧
wf_negations_of_literals x ∧ not (exists f1 f2. x = FAnd_wi f1 f2) →
wf_disjunctions x
```

Prova da função T. Com as provas de correção de cada uma das três funções efetuadas, pode-se agora obter a prova de correção da função T. Esta garante todos os critérios assegurados pelas três funções:

```
let t (phi: formula) : formula_wi
  ensures { wf_negations_of_literals result }
  ensures { wf_conjunctions_of_disjunctions result }
  ensures { forall v. eval v phi = eval_wi v result  }
= cnfc (nnfc (impl_free phi))
```

A prova em estilo direto da implementação e especificação – próxima das definições matemáticas clássicas – é imediata em Why3, tornando este exercício numa bem sucedida prova de conceito.

5 Continuation-Passing Style

Continuation-Passing Style (CPS) é um estilo de programação onde o controlo é passado explicitamente na forma de continuação, evitando assim o *overflow* da pilha se o compilador subjacente optimizar as chamadas recursivas terminais. Com uma estrutura de pilha explicita no código é possível, no futuro, introduzir um mecanismo que permita a execução passo-a-passo das funções.

Processo de transformação para CPS.

A transformação é efetuada de forma mecânica e segue os seguintes passos: dada uma função `t' → t`, adiciona-se um argumento que representará a continuação (uma função do tipo `t → 'a`) e é alterado o tipo de retorno da função para `'a`; para os casos bases em vez de se retornar os valores desejados, aplicam-se estes valores à função de continuação; para os restantes casos, começa-se por efetuar uma chamada recursiva à função, sendo as continuações criadas com o resto da computação. Por fim, é criada uma função `main` que chama a função CPS com a função identidade como continuação.

Aplicando este processo à função `Impl_Free`: adiciona-se um argumento à função do tipo `formula_wi → 'a` e altera-se o tipo de retorno para `'a`:

```
let rec impl_free_cps (phi: formula) (k: formula_wi → 'a ) : 'a
```

Os casos bases são então aplicados à função de continuação:

```
| FConst phi → k (FConst_wi phi)
| FVar phi → k (FVar_wi phi)
```

Para os restantes casos começa-se com uma chamada recursiva e define-se as continuações:

```
| FNeg phi1 → impl_free_cps phi1 (fun con → k (FNeg_wi con))
| FImpl phi1 phi2 → impl_free_cps phi1 (fun con → impl_free_cps phi2
    (fun con1 → k (FOr_wi (FNeg_wi con) con1)))
...
```

Por fim, cria-se uma função `main` que chama a função CPS com a função identidade como continuação:

```
let impl_free_main (phi: formula) : formula_wi
= impl_free_cps phi (fun x → x)
```

A transformação em CPS das restantes funções é obtida de forma semelhante.

Especificação dos critérios de correção. Um aspecto interessante na prova de correção das funções em CPS é o uso da correspondente função em estilo direto, visto estas serem puras e totais, como própria especificação, ou seja, assegura-se que o resultado é igual ao resultado das funções em estilo direto aplicado à continuação.

Para a função `Impl_Free` em CPS basta então assegurar que o resultado é igual ao resultado da função `Impl_Free` em estilo direto aplicado à continuação:

```
let rec impl_free_cps (phi: formula) (k: formula_wi → 'a ) : 'a
  variant { phi }
  ensures { result = k(impl_free phi) }
= ...
```

A especificação da função em estilo direto é depois aplicada à função `main`:

```
let impl_free_main (phi: formula) : formula_wi
  ensures { forall v. eval v phi = eval_wi v result }
= ...
```

As especificações das funções `NNFC` e `CNFC` em CPS são semelhantes à especificação da função `Impl_Free`; no entanto, é necessário provar as pré-condições da função `CNFC`, ou seja, provar que a fórmula de entrada está na FNN.

Sempre que é efectuada uma chamada recursiva no interior de uma continuação, uma obrigação de prova é gerada respeitante à validade da pré-condição desta chamada. Para provar tal obrigação de prova, é necessário especificar a natureza dos argumentos das continuações. Assim, encapsula-se o predicado `wf_negations_of_literals` dentro de um novo tipo (tipo invariante):

```
type nnfc_type = {
    nnfc_formula : formula_wi
  } invariant { wf_negations_of_literals nnfc_formula }
    by{ nnfc_formula = FConst_wi true }
```

Visto que o tipo de retorno da função é alterado, a prova das pós-condições implica agora a comparação de dois tipos invariante, o que levantou dificuldades.

Dificuldades da prova. Comparar dois tipos invariante implica dar-lhes uma testemunha, ou seja, valores com o tipo em causa; só assim é possível provar que dois valores do mesmo tipo respeitam o invariante; no entanto como o tipo invariante em Why3 é um tipo opaco, tendo apenas acesso às suas projeções, não é possível a construção de um habitante deste tipo na lógica, impossibilitando assim a sua comparação. Esta dificuldade é facilmente traduzida para um lema:

```
lemma types: forall x y. x.cnfc_formula = y.cnfc_formula → x = y
```

Não é possível provar este lema porque tendo apenas acesso às projeções do *record* não é possível assegurar que, neste caso, o campo `cnfc_formula` é o único campo do *record*. Tendo em conta esta limitação do Why3 [1], o que neste caso impossibilita a prova da pós-condição, tentou-se apenas comparar a fórmula de cada tipo com um predicado de igualdade extensional (`==`) e usar este predicado como pós-condição em vez da igualdade estrutural polimórfica (`=`).

```
predicate (==) (t1 t2: cnfc_type) = t1.cnfc_formula = t2.cnfc_formula
```

Mesmo com a igualdade extensional, não foi possível concluir a prova. Isto porque, nos casos bases, devido à aplicação à continuação, acaba-se sempre por deparar com uma comparação de records e nos restantes casos não é possível especificar as funções de continuação nas chamadas recursivas. Esta falta de sucesso levou à procura de outras abordagens que permitissem obter as mesmas vantagens que a transformação CPS.

Qual o problema com CPS?. A transformação em CPS adiciona sempre uma função como argumento, passando assim a uma função de ordem superior. Sendo o Why3 uma plataforma que por razões de decidibilidade opera sobre uma linguagem de primeira ordem, a solução passa por "voltar"para a primeira ordem, surgindo então a desfuncionalização como uma possível abordagem.

6 Desfuncionalização

A desfuncionalização é uma técnica de transformação de programas de ordem superior para programas de primeira ordem [13].

Processo de transformação. A desfuncionalização consiste numa transformação "mecânica" em dois passos: representação de primeira ordem das continuações da função e substituição das continuações por esta nova representação;

introdução de uma função `apply` que substitui as aplicações à continuação no programa original. Aplicando este processo à função `Impl_Free` em CPS, representam-se em primeira ordem as continuações da função:

```
type impl_kont =
  | KImpl_Id
  | KImpl_Neg impl_kont formula
  | KImpl_OrLeft formula impl_kont
  | KImpl_OrRight impl_kont formula_wi
  ...
```

O construtor `KImpl_id` representa a identidade e o construtor `KImpl_Neg` representa a continuação do caso do construtor FNeg_wi. Como os restantes construtores contêm duas funções de continuação criam-se dois construtores, um `left` e um `right`, representado assim a ordem da formula na árvore sintaxe abstrata.

Depois substituí-se as continuações pela nova representação das continuações, introduz-se uma função `apply` e substitui-se as aplicações à continuação:

```
let rec impl_free_desf_cps (phi: formula) (k: impl_kont) : formula_wi
= match phi with
  | FNeg phi1 → impl_free_desf_cps phi1 (KImpl_Neg k phi1)
  | FOr phi1 phi2 → impl_free_desf_cps phi1 (KImpl_OrLeft phi2 k)
  | FVar phi → impl_apply (FVar_wi phi) k
  ...
end

with impl_apply (phi: formula_wi) (k: impl_kont) : formula_wi
= match k with
  | KImpl_Id → phi
  | KImpl_Neg k phi1 → impl_apply (FNeg_wi phi) k
  | KImpl_OrLeft phi1 k → impl_free_desf_cps phi1 (KImpl_OrRight k phi)
  | KImpl_OrRight k phi2 → impl_apply (FOr_wi phi2 phi) k
  ...
end
```

As transformações das restantes funções são obtidas de forma semelhante.

Prova de Correção. A especificação do programa desfuncionalizado é a mesma do programa original; no entanto, dada a existência de uma função adicional (a função `apply` gerada pelo processo de desfuncionalização), é preciso fornecer uma especificação a esta. Sendo a função `apply` uma simulação da aplicação de uma função ao seu argumento, a única especificação que se pode fornecer é a de que a sua pós-condição é a pós-condição da função `k` [12].

Para ser possível o uso das funções em estilo direto como especificação, cria-se um predicado `post` que reúne as pós-condições da função em estilo direto. Tal como para a função apply, um tal predicado efectua uma filtragem sobre o tipo da continuação e para cada construtor copia-se a pós-condição presente na abstracção correspondente [12]. Por exemplo para a função `Impl_Free`:

```
let rec impl_free_desf_cps (phi: formula) (k: impl_kont) : formula_wi
  ensures{impl_post k (impl_free phi) result}
 = ...
```

```
with impl_apply (phi: formula_wi) (k: impl_kont) : formula_wi
  ensures{impl_post k phi result}
 = ...
```

Sendo o predicado impl_post o seguinte:

```
predicate impl_post (k: impl_kont) (phi result: formula_wi)
= match k with
  | KImpl_Id → let x = phi in x = result
  | KImpl_Neg k phi1 → let neg = phi in impl_post k (FNeg_wi phi) result
  | KImpl_OrLeft phi1 k → let hl = phi in impl_post k (FOr_wi phi
    (impl_free phi1)) result
  | KImpl_OrRight k phi2 → let hr = phi in impl_post k (FOr_wi phi2 hr)
    result
  ...
end
```

A prova das pós-condições das restantes funções desfuncionalizadas é semelhante à da função `Impl_Free`. No entanto, à semelhança da prova em CPS, na função `CNFC` é preciso provar as suas pré-condições. Para tal, cria-se o tipo invariante `wf_cnfc_kont` com o predicado de boa formação `wf_cnfc_kont` como invariante:

```
type wf_cnfc_kont = {
    cnfc_k: cnfc_kont;
  } invariant { wf_cnfc_kont cnfc_k }
    by { cnfc_k = KCnfc_Id }
```

De notar que no predicado de boa formação apenas se quer assegurar a FNC para as fórmulas que já estão convertidas. Tendo em conta que as fórmulas só são convertidas na continuação da direita, apenas estas e só estas contêm o predicado `wf_conjunctions_of_disjunctions`:

```
predicate wf_cnfc_kont (phi: cnfc_kont)
= match phi with
  | KCnfc_Id → true
  | KCnfc_OrLeft phi k → wf_negations_of_literals phi ∧ wf_cnfc_kont k
  | KCnfc_OrRight k phi → wf_negations_of_literals phi ∧
    wf_conjunctions_of_disjunctions phi ∧ wf_cnfc_kont k
  ...
end
```

Finalmente, a prova da função T acaba por ser semelhante à prova em estilo direto referida na Página 7.

Resultados. A prova de correção da versão desfuncionalizada do algoritmo T é processada naturalmente pelo Why3, sendo a prova de cada objetivo de prova realizada em menos de um segundo. O resultado da prova pode ser observado no repositório do projeto.

7 Conclusão

Linguagens como o OCaml, permitem implementações próximas das definições matemáticas, sem sacrificar clareza e rigor, sendo adequadas ao uso pedagógico de auxílio ao estudo e compreensão de algoritmos.

Neste artigo apresenta-se uma prova de conceito: a implementação e prova de correção da conversão de fórmulas proposicionais para a FNC. A prova das duas vertentes do algoritmo – estilo direto e desfuncionalizada – foi conseguida naturalmente pelo Why3, tornando bem sucedida a prova de conceito de implementações formalmente verificadas de algoritmos de Lógica Computacional.

Futuramente, pretende-se efetuar implementações suportando a execução passo-a-passo, através de uma estrutura de pilha explicita no código, visto que cada chamada da função retorna uma função (continuação) que pode ser usada como bloqueio, permitindo assim a paragem e retorno da execução. Pretende-se também aplicar esta abordagem a outros algoritmos de Lógica Computacional, como por exemplo o algoritmo de Horn [8].

Referências

1. Add injectivity for type invariant (#287) · Why3 Issues, https://gitlab.inria.fr/why3/why3/issues/287
2. FACTOR: Functional ApproaCh Teaching pOrtuguese couRses, http://ctp.di.fct.unl.pt/FACTOR/
3. Ager, M.S., Biernacki, D., Danvy, O., Midtgaard, J.: A functional correspondence between evaluators and abstract machines. In: Proceedings of the International Conference on Principles and Practice of Declarative Programming (2003)
4. Ben-Ari, M.: Mathematical Logic for Computer Science, 3rd Edition. Springer (2012), https://doi.org/10.1007/978-1-4471-4129-7
5. Enderton, H.B.: A mathematical introduction to logic. Academic Press (1972)
6. Filliâtre, J., Paskevich, A.: Why3 - Where Programs Meet Provers. In: Programming Languages and Systems. Lecture Notes in Computer Science, Springer, https://doi.org/10.1007/978-3-642-37036-6_8
7. Hamilton, A.G.: Logic for mathematicians. Cambridge University Press (1988)
8. Horn, A.: On Sentences Which are True of Direct Unions of Algebras, vol. 16 (1951), https://doi.org/10.2307/2268661
9. Huth, M., Ryan, M.D.: Logic in computer science - modelling and reasoning about systems (2. ed.). Cambridge University Press (2004)
10. Leroy, X., Doligez, D., Frisch, A., Garrigue, J., Rémy, D., Vouillon, J.: The OCaml system release 4.07: Documentation and user's manual. Intern report, Inria (2018), https://hal.inria.fr/hal-00930213
11. Mendelson, E.: Introduction to mathematical logic (3. ed.). Chapman and Hall (1987)
12. Pereira, M.: Desfuncionalizar para Provar. CoRR **abs/1905.08368** (2019), http://arxiv.org/abs/1905.08368
13. Reynolds, J.C.: Definitional Interpreters for Higher-Order Programming Languages. vol. 11, pp. 363–397 (1998), https://doi.org/10.1023/A:1010027404223
14. Sabry, A., Felleisen, M.: Reasoning about Programs in Continuation-Passing Style. Lisp and Symbolic Computation **6**(3-4), 289–360 (1993)
15. de Sousa, S.M.: Verificação Deductiva de Programas em Why3, https://www.di.ubi.pt/~desousa/why3/aula_why3-pp.pdf