Desfuncionalizar para Provar

Mário Pereira*

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

Resumo Este artigo explora a utilização da desfuncionalização como uma técnica de prova de programas de ordem superior. A desfuncionalização consiste em substituir valores funcionais por uma representação de primeira ordem. O seu interesse é portanto poder utilizar posteriormente uma ferramenta de prova de programas existente, sem ser necessário estender esta ferramenta com suporte para ordem superior. Este artigo ilustra e discute esta abordagem através de diversos exemplos práticos, construídos e validados na ferramenta de verificação dedutiva Why3.

1 Introdução

Um programa de ordem superior é um programa que utiliza funções como valores de primeira classe. Em tais programas, funções podem ser passadas como argumentos a outras funções ou devolvidas como o resultado de computações. O conceito de ordem superior é amplamente conhecido e utilizado em linguagens ditas *funcionais*, tais como OCaml, Haskell ou SML. Recentemente, linguagens como Java ou C++ introduziram suporte para funções de ordem superior.

A prova de correcção de programas de ordem superior coloca desafios complexos, em particular no contexto de programas com efeitos. Metodologias existentes [2, 10] empregam assistentes de prova interactivos e uma codificação dos efeitos directamente na lógica destes assistentes de prova. No contexto de prova automática de programas, Kanig e Filliâtre [8] propuseram um sistema de tipos e efeito, assim como um cálculo de pré-condições mais fracas, com o propósito de especificar e provar a correção funcional de programas de ordem superior. No entanto, tal sistema é difícil de utilizar de forma escalável.

Neste artigo, propomos uma nova metodologia para a verificação de programas de ordem superior com efeitos. Exploramos a ideia de empregar a técnica de desfuncionalização para verificar tais programas. A utilização da desfuncionalização permite-nos obter um programa de primeira ordem equivalente ao programa de ordem superior original. Assim, podemos recorrer a ferramentas de prova de primeira ordem existentes, sem que para tal seja necessário estender tais ferramentas com suporte para a ordem superior. Esta técnica apresenta como

* Este trabalho foi realizado enquanto o autor se encontrava afiliado ao LRI – Laboratoire de Recherche en Informatique, CNRS, Université Paris-Sud – na qualidade de aluno de doutoramento. Trabalho parcialmente suportado pela Fundação Tezos através do projeto FACTOR (`http://www-ctp.di.fct.unl.pt/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.

limitação o facto de ser necessário conhecer *à priori* todas as funções que serão utilizadas como valores de primeira classe. Mesmo se, teoricamente, tal nos impede de aplicar a técnica de desfuncionalização para provar qualquer programa, é possível identificar uma classe interessante e representativa de programas de ordem superior aos quais nos é possível aplicar a desfuncionalização para provar a sua correcção. Neste artigo, apresentamos a nossa abordagem através de diversos exemplos escritos e verificados no sistema de prova de programas Why3 [6].

O código Why3 apresentado neste artigo pode ser encontrado em `https://mariojppereira.github.io/defunc/`. Uma versão estendida deste documento pode ser encontrada em `http://arxiv.org/abs/1905.08368`. Este artigo estende e actualiza os resultados preliminares apresentados numa primeira versão não publicada deste trabalho [11].

2 Desfuncionalização

A desfuncionalização é uma técnica de transformação de programas de ordem superior para programas de primeira ordem. Esta técnica foi introduzida por Reynolds [15] como forma de obter um interpretador de ordem superior a partir de um interpretador de primeira ordem. Recentemente, esta técnica tem sido amplamente explorada por Danvy *et al.* [3,4]. Em particular, estes autores mostraram de que forma é possível derivar máquinas abstractas para diferentes estratégias de avaliação do cálculo-lambda a partir de interpretadores composicionais [1].

Passamos a explicar o princípio da desfuncionalização através de um exemplo escrito em OCaml[1] Consideremos o programa que calcula a altura de uma árvore binária, escrito em estilo CPS [12]:

```
type 'a tree = E | N of 'a tree * 'a * 'a tree

let rec height_tree_cps (t: 'a tree) (k: int → 'b) : 'b = match t with
  | E → k 0
  | N (l, x, r) →
     height_tree_cps l (fun hl →
     height_tree_cps r (fun hr → k (1 + max hl hr)))

let height_tree t = height_tree_cps t (fun x → x)
```

Este programa foi propositadamente escrito em estilo CPS para evitar quaisquer problemas de *stack overflow*, independentemente da forma da árvore. A transformação CPS pode ser usada como um meio automático para evitar problemas de *stack overflow*, desde que o compilador optimize chamadas recursivas terminais.

Chamamos a atenção para a presença, na função `height_tree_cps`, do argumento `k` do tipo `int → 'b`. É a utilização deste argumento que confere à função `height_tree_cps` a sua natureza de ordem superior. Este argumento actua como uma *continuação*, ou seja, ao invés de devolver um resultado, a função `height_tree_cps` passa-o como argumento a `k`. No caso da árvore vazia

[1] Este código pode ser facilmente adaptado a qualquer linguagem que suporte funções como valores de primeira classe.

(primeiro ramo da filtragem), por exemplo, aplicamos `k` a 0 em detrimento de devolver directamente tal resultado.

Três funções anónimas são utilizadas no código do programa que calcula a altura da árvore. Na chamada `height_tree_cps l` a continuação `(fun hl → ...)` é aplicada ao resultado do cálculo da altura da sub-árvore esquerda `l`. O argumento `hl` representa a altura de `l` já calculada. No interior desta continuação encontramos uma segunda chamada recursiva com uma outra continuação, desta vez sobre a sub-árvore direita `r`. Aquando da aplicação da função anónima `(fun hr → ...)`, conhecemos já todos os ingredientes necessários para calcular a altura da árvore inicial `t`. Também neste caso não devolvemos directamente o resultado da expressão `1 + max hl hr`, mas é sim passado a `k`. A terceira e última função anónima que encontramos neste programa é a função identidade `(fun x → x)`. A aplicação desta função garante que o resultado devolvido por `height_tree t` é de facto a altura da árvore `t`.

Para desfuncionalizar este programa, precisamos de uma representação de primeira ordem para as três abstracções utilizadas. Para isso, substituímos o tipo funcional por um tipo algébrico, através do qual capturamos os valores das variáveis livres utilizadas em cada função. Para o exemplo apresentado acima, introduzimos o tipo seguinte:

```
type 'a kont = Kid | Kleft of 'a tree * 'a kont | Kright of 'a kont * int
```

O construtor `Kid` representa a função identidade e por isso não contém variáveis livres. Os construtores `Kleft` e `Kright` representam, respectivamente, as funções `(fun hl → ...)` e `(fun hr → ...)`. Os seus argumentos capturam as variáveis livres utilizadas em cada uma das duas funções. No caso de `Kleft` guardamos a sub-árvore `r` e a continuação `k`; no caso de `Kright` o primeiro argumento é o valor de `k` e o segundo representa `hl`, a altura da sub-árvore esquerda.

Tendo entre mãos uma representação de primeira ordem, podemos agora proceder à substituição de todas as abstracções pelo construtor de `'a kont` correspondente:

```
let rec height_tree_cps t (k: 'a kont) = match t with
  | E → ??? (* aplicar k desfuncionalizado ao seu argumento *)
  | N (l, x, r) → height_tree_cps l (Kleft (r, k))

let height_tree t = height_tree_cps t Kid
```

A segunda etapa do processo de desfuncionalização consiste em introduzir uma função `apply` para substituir as aplicações no programa original. Esta função aceita como argumentos um valor do tipo `'a kont` sobre o qual realizamos uma análise por casos. Para cada construtor, a função `apply` executa o código da abstracção que corresponde a esse mesmo construtor. O segundo argumento de `apply` trata-se do argumento da aplicação no programa original. A função `apply` para o nosso exemplo é a seguinte:

```
let rec apply (k: 'a kont) (arg: int) = match k with
  | Kid → let x = arg in x
  | Kleft (r, k) → let hl = arg in height_tree_cps r (Kright (k, hl))
```

```
  | Kright (k, hl) → let hr = arg in apply k (1 + max hl hr)
```

Para obter o programa completamente desfuncionalizado, basta então substituir todas as aplicações da continuação `k` por chamadas à função `apply`. O código completo deste exemplo pode ser encontrado no Apêndice A.1.

3 Prova por desfuncionalização

Nesta secção exploramos a ideia de utilizar a desfuncionalização como uma técnica de prova para programas de ordem superior com efeitos. A nossa proposta é a seguinte: (1) dado um programa de ordem superior (possivelmente contendo efeitos), instrumentamos tal programa com uma especificação lógica; (2) desfuncionalizamos este programa e ao mesmo tempo traduzimos a sua especificação, para que esta se torne uma especificação do programa desfuncionalizado; (3) utilizamos uma ferramenta de verificação de programas existente para construir uma prova de que o programa desfuncionalizado respeita a sua especificação. Passamos a ilustrar esta proposição através de diferentes exemplos: o programa que calcula a altura de uma árvore (Sec. 3.1); um programa que calcula o número de elementos distintos de uma árvore (Sec. 3.2); um interpretador *small-step* para uma pequena linguagem (Sec. 3.3). Todas as experiências descritas são realizadas com ajuda da ferramenta de verificação Why3. Como actualmente esta ferramenta não nos permite racionar um programa de ordem superior arbitrário, todos os exemplos de especificação de programas de ordem superior são escritos num sistema hipotético, uma possível extensão do Why3.

3.1 Altura de uma árvore

Recuperemos o exemplo da Sec. 2. A fim de especificar este programa, devemos instrumentar as funções `height_tree_cps` e `height_tree` com contratos. A função `height_tree` devolve a altura da árvore `t` passada como argumento, tal como especificamos na sua pós-condição:

```
let height_tree (t: tree 'a) : int
  ensures { result = height t }
= height_tree_cps t (fun x → x)
```

Aqui, `result` trata-se de uma palavra reservada do Why3 que representa o valor devolvido e `height` é uma função lógica, presente na biblioteca *standard* do Why3, que devolve a altura de uma árvore.

O valor devolvido pela função `height_tree_cps` é o resultado da aplicação da continuação `k` à altura da árvore `t`. Seria assim natural equipar `height_tree_cps` com a pós-condição `result = k (height t)`. No entanto, esta especificação coloca-nos o problema de como interpretar a utilização de funções de programa na lógica. Em geral, tal utilização pode introduzir incoerências lógicas, já que funções de programa podem conter efeitos, *e.g.*, divergência. Fazemos então a escolha de restringir a nossa linguagem de especificação: podemos utilizar nomes de funções, mas impomos uma barreira de abstracção entre o mundo lógico

e o programa. Para isso, adoptamos um sistema no qual as funções são abstraídas sob a forma de um par de predicados que representam as suas pré- e pós-condição [14]. Assim, no interior de uma fórmula lógica, uma função f do tipo $\tau_1 \to \tau_2$ é vista como o par de predicados

$$\begin{aligned} \texttt{pre}\ f &: \tau_1 \to \texttt{prop} \\ \texttt{post}\ f &: \tau_1 \to \tau_2 \to \texttt{prop} \end{aligned}$$

Utilizamos `pre` f e `post` f para nos referirmos à pré-condição e à pós-condição de f, respectivamente.

Podemos então escrever, utilizando a notação apresentada acima, o seguinte contrato para a função `height_tree_cps`:

```
let rec height_tree_cps (t: tree 'a) (k: int → 'b) : 'b
  ensures { post k (height t) result }
```

Esta pós-condição impõe uma relação entre o valor passado a `k` (a altura de `t`) e o resultado devolvido (`result`). Seguindo a nossa metodologia, podemos igualmente fornecer contratos às funções anónimas utilizadas:

```
let rec height_tree_cps (t: tree 'a) (k: int → 'b) : 'b
  ensures { post k (height t) result }
= match t with
  | Empty → k 0
  | Node l x r →
     height_tree l (fun hl →
       ensures { post k (1 + max hl (height r)) result }
     height_tree r (fun hr →
       ensures { post k (1 + max hl hr) result }
     k (1 + max hl hr)))
  end
```

Para a primeira abstracção, especificamos que o resultado da sua aplicação se relaciona com a altura da árvore completa, através da altura `hl` da sub-árvore esquerda e da altura da sub-árvore direita, que ainda não conhecemos. Juntamos à segunda abstracção uma pós-condição semelhante, com a nuance de que no momento de aplicar esta função já conhecemos a altura `hr`.

Procedemos agora à desfuncionalização deste programa, assim como da sua especificação, para em seguida provar a sua correcção funcional com a ajuda da ferramenta Why3. O código completo obtido após desfuncionalização pode ser encontrado no Apêndice A.2.

A especificação do programa desfuncionalizado é a mesma do programa original. Em particular, mantemos a nossa utilização da projecção `post` para a pós-condição de `height_tree_cps`. Precisamos então de fornecer uma especificação à função `apply`, a nova função gerada pelo processo de desfuncionalização. Como a função `apply` simula a aplicação de uma função ao seu argumento, a única especificação que lhe podemos fornecer é a de que a sua pós-condição é a pós-condição da função `k`. Da mesma forma, a pré-condição de `apply` seria a pré-condição de `k`, à qual podemos aceder utilizando a projecção `pre`.

Finalmente, é necessário introduzir o predicado `post` na especificação lógica desfuncionalizada. Para tal, criamos um predicado `post` que reúne as pós-condições fornecidas no programa original. Tal como para a função `apply`, um tal predicado efectua uma filtragem sobre o tipo algébrico `kont 'a` e para cada construtor copiamos a pós-condição fornecida na abstracção correspondente. Para este exemplo, o predicado `post` é o seguinte:

```
predicate post (k: kont 'a) (arg result: int) = match k with
 | Kid → let x = arg in x = result
 | Kleft r k' → let hl = arg in post k' (1 + max hl (height r)) result
 | Kright hl k' → let hr = arg in post k' (1 + max hl hr) result
 end
```

Como pós-condição do construtor `Kid` utilizamos a pós-condição trivial. Esta fórmula representa a pós-condição mais forte desta função, podendo ser automaticamente inferida. Utilizando a ferramenta Why3 sobre este programa, todas as obrigações de prova geradas são automaticamente provadas por SMTs[2].

Um aspecto importante a ressalvar é de que a forma como desfuncionalizámos este programa e a sua especificação, em particular a forma de construir o predicado `post` e o contrato da função `apply`, pode ser mecanizada. Podemos assim conceber uma ferramenta que recebe um programa de ordem superior anotado, que o desfuncionaliza e finalmente o envia à ferramenta Why3.

3.2 Número de Elementos Distintos de uma Árvore

O próximo exemplo apresenta um programa com o propósito de calcular o número de elementos distintos de uma árvore binária, com complexidade linear no número de nós da árvore. Como anteriormente, adoptamos um estilo CPS para evitar qualquer problema de *stack overflow*. Este programa difere do da secção anterior pela presença de *efeitos*. Utilizamos um conjunto mutável (uma referência para um conjunto finito) a fim de guardar os elementos já encontrados. No final do programa devolvemos o cardinal deste conjunto, obtendo assim o número de elementos distintos na árvore. Segue-se a implementação em Why3:

```
let n_distinct_elements (t: tree 'a) : int =
  let h = ref empty in
  let rec distinct_elements_loop (t: tree 'a) (k: unit → 'b) : 'b =
    match t with
    | Empty → k ()
    | Node l x r →
       h := add x !h;
       distinct_elements_loop l (fun () →
       distinct_elements_loop r (fun () → k ()))
    end in
  distinct_elements_loop t (fun x → x);
  cardinal !h
```

[2] Neste artigo omitimos a prova de terminação dos diferentes exemplos. A versão estendida deste artigo, assim como código disponibilizado online, apresentam provas de correção total.

As operações sobre conjuntos utilizadas neste programa provêm da biblioteca standard do Why3. As continuações presentes neste programa têm uma utilização semelhante às que podemos encontrar no programa da Sec. 3.1. A sub-árvore esquerda é tratada pela chamada `distinct_elements_loop l (fun () → ...)`; a chamada `distinct_elements_loop r (fun () → ...)` trata a sub-árvore direita; para assegurar que de facto devolvemos o resultado correcto, a primeira chamada a `distinct_elements_loop` no interior de `distinct_elements` é feita com a função identidade.

Estando escrito em estilo CPS, este programa combina a utilização de ordem superior com efeitos. Devemos, por isso, considerar a noção de estado do programa na sua especificação. Para tal, modificamos a representação das funções ao nível da lógica, segundo a tese de J. Kanig [7]. Introduzimos, em primeiro lugar, um novo tipo `state` e estendemos o tipo das projecções como se segue:

$$\begin{aligned} \texttt{pre}\ f &: \tau_1 \to \texttt{state} \to \texttt{prop} \\ \texttt{post}\ f &: \tau_1 \to \texttt{state} \to \texttt{state} \to \tau_2 \to \texttt{prop} \end{aligned}$$

O argumento extra da pré-condição corresponde ao estado no momento da chamada da função. Os dois argumentos extra da pós-condição representam, respectivamente, o estado *antes* e *após* a execução da função. A natureza do tipo `state` dependerá das funções consideradas. Para este exemplo, o estado é simplesmente `set 'a`.

Para especificar `n_distinct_elements` e `distinct_elements_loop` utilizamos uma função lógica `set_of_tree`, que representa o conjunto lógico de todos os elementos de uma árvore. Damos à função `n_distinct_elements` o seguinte contrato:

```
let n_distinct_elements (t: tree 'a) : int
  ensures { result = cardinal (set_of_tree t) }
```

Especificamos `distinct_elements_loop` e as continuações utilizadas de forma semelhante:

```
let rec distinct_elements_loop (t: tree 'a) (k: unit → unit) : unit
  ensures { post k () (set_of_tree_acc t (old !h)) !h () }
= match t with
  | Empty → k ()
  | Node l x r →
     h := add x !h;
     distinct_elements_loop l (fun () →
       ensures { post k () (set_of_tree_acc r (old !h)) !h () }
     distinct_elements_loop r (fun () →
       ensures { post k () (old !h) !h () } k ()))
  end
```

A função lógica `set_of_tree_acc` representa a reunião do conjunto de elementos de uma árvore (primeiro argumento) com um conjunto acumulador (segundo argumento). A pós-condição de `distinct_elements_loop` carece de uma explicação detalhada. Dado que utilizamos a continução `k` no seu interior, a pós-

condição desta função depende da pós-condição de `k`. É assim necessário caracterizar o estado do programa quando chamamos `k` e o estado após a sua execução. Este último é o estado obtido após a execução, por inteiro, da função `distinct_elements_loop`, representado pelo valor contido na referência `h`. O estado inicial é mais subtil. Recordemos que no momento de aplicar `k` teremos já percorrido toda a árvore `t`. Assim, o estado a partir do qual chamamos a função `k` é o conjunto de todos os elementos de `t` reunidos com o valor contido em `h` no momento da chamada a `distinct_elements_loop`. Podemos recuperar este valor através da etiqueta `old` do Why3, que nos permite aceder ao valor contido numa referência no momento de chamada a uma função.

A especificação das continuações utilizadas no interior das chamadas recursivas a `distinct_elements_loop` segue o raciocínio que acabamos de descrever. As pós-condições destas duas abstracções dependem, igualmente, da pós-condição de `k`. Na chamada a `distinct_elements_loop l`, especificamos na pós-condição da sua continuação que o estado com o qual aplicaremos a continuação é `set_of_tree r (old !h)`. Aqui, `old !h` representa o estado antes de se aplicar a continuação, isto é, após se ter percorrido a sub-árvore `l`. Como esta continuação é utilizada para percorrer toda a sub-árvore direita, no momento de se aplicar `k` a referência `h` contém, assim, o conjunto dos elementos distintos de `l` e `x`, que juntámos inicialmente. Finalmente, a continuação para a aplicação `distinct_elements_loop r` apenas aplica `k` e por isso esta aplicação é feita com o estado inicial `old !h`, exactamente o estado antes da chamada a `k`.

Para desfuncionalizar este programa e traduzir a sua especificação, começamos por introduzir o tipo algébrico que representa as continuações:

```
type kont 'a = Kid | Kleft (tree 'a) (kont 'a) | Kright (kont 'a)
```

Os únicos efeitos produzidos neste exemplo são o acesso (para escrita e leitura) à referência `h`. Assim, podemos definir o tipo `state` como sendo o tipo dos valores guardados em `h`:

```
type state 'a = set 'a
```

Finalmente, introduzimos o predicado `post`:

```
predicate post (k: kont 'a) (arg: unit) (old cur: state 'a) (result: unit)
= match k with
  | Kid → let () = arg in old = cur
  | Kleft r k → let () = arg in post k () (set_of_tree_acc r old) cur result
  | Kright k → let () = arg in post k () old cur result
  end
```

Para o caso da função identidade, este predicado especifica que o estado à saída da função é o mesmo que à entrada. Utilizando esta definição de `post`, podemos gerar a função `apply` com a sua especificação, como se segue:

```
let rec apply (k: kont 'a) (arg: unit) : unit
  ensures { post k arg (old !h) !h () }
= match k with
  | Kid → let x = arg in x
```

```
| Kleft r k → let () = arg in distinct_elements_loop r (Kright k)
| Kright k → let () = arg in apply k arg
end
```

Uma vez processado pelo Why3, todas as obrigações de prova geradas para este programa são automaticamente descartadas. A implementação e especificação desfuncionalizadas para este exemplo podem ser encontradas no Apêndice B.

3.3 Interpretador *small-step*

O último exemplo que apresentamos é o de um interpretador *small-step* para uma mini linguagem de expressões aritméticas. A implementação e especificação de ordem superior para este exemplo podem ser encontradas no Apêndice C. Trata-se de uma linguagem limitada a constantes literais e subtracções:

```
type exp = Const int | Sub exp exp
```

Para definir uma relação de redução, começamos por definir a relação $\xrightarrow{\epsilon}$, que corresponde a uma redução à cabeça da expressão. Para esta linguagem existe uma só regra de redução à cabeça:

$$\texttt{Sub}\ (\texttt{Const}\ \texttt{v}_1)\ (\texttt{Const}\ \texttt{v}_2) \quad \xrightarrow{\epsilon} \quad \texttt{Const}\ (\texttt{v}_1 - \texttt{v}_2)$$

Para reduzir em profundidade, introduzimos agora a regra de inferência

$$\frac{\texttt{e} \xrightarrow{\epsilon} \texttt{e}'}{C[\texttt{e}] \to C[\texttt{e}']}$$

onde C representa um contexto de redução, definido pela seguinte gramática:

$$\begin{aligned} C ::= {} & \square \\ & \mid C[\texttt{Sub}\ \square\ \texttt{e}] \\ & \mid C[\texttt{Sub}\ (\texttt{Const}\ \texttt{v})\ \square] \end{aligned}$$

O símbolo $\square$ representa o buraco. Definimos $C[\texttt{e}]$ como sendo o contexto C no qual o buraco foi completamente substituído pela expressão `e`. A esta operação damos o nome de *composition*. O resultado desta operação é uma nova expressão.

O ponto interessante deste exemplo é a maneira como escolhemos representar os contextos. Em detrimento de uma representação baseada num tipo algébrico com três construtores, escolhemos aqui representar um contexto como uma função. Um contexto `c` é assim uma função que recebe como argumento uma expressão `e` e devolve a expressão obtida por substituição do buraco de `c` para uma expressão `e`:

```
type context = exp → exp
```

O contexto vazio, o buraco, é representado pela função identidade:

```
let hole = fun x → x
```

Os contextos para a redução sobre o primeiro ou sobre o segundo argumento de `Sub` são representados, respectivamente, como se segue:

```
let sub_left e2 c = fun e1 → c (Sub e1 e2)
let sub_right v c = fun e1 → c (Sub (Const v) e1)
```

Esta maneira de representar os contextos conduz-nos a um código de ordem superior elegante, mais compacto que um código utilizando um tipo concreto para os contextos, tal como observaremos em seguida.

Tendo definida a noção de contexto, a próxima etapa consiste na implementação de uma função que decompõe uma expressão `e` em um *redex* `e'` e um contexto C, tais que $C[\mathtt{e}'] = \mathtt{e}$. Para a nossa linguagem esta decomposição é única. Tal operação pode ser implementada com base na seguinte função auxiliar `decompose_term`:

```
let rec decompose_term (e: exp) (c: context) : (context, exp)
  requires { not (is_value e) }
  returns  { (c', e') → is_redex e' ∧
             forall res. post c e res → post c' e' res }
= match e with
  | Const _ → absurd
  | Sub (Const v1, Const v2) → (c, e)
  | Sub (Const v, e) → decompose_term e (fun x →
      ensures { post c (Sub (Const v) x) result } c (Sub (Const v) x))
  | Sub (e1, e2) → decompose_term e1 (fun x →
      ensures { post c (Sub x e) result } c (Sub x e2))
  end
```

Esta função recebe como argumento um contexto `c` e uma expressão `e` que desejamos decompor. Para isso, procedemos a uma análise por casos sobre a forma da expressão `e` [3].

De notar que este programa não apresenta efeitos. Por isso, utilizamos uma projecção `post` de dois argumentos, semelhante ao que fizemos na Sec. 3.1. A função `decompose_term` deve verificar a propriedade $C'[\mathtt{e}'] = C[\mathtt{e}]$, C' e $\mathtt{e}'$ sendo o contexto e a expressão devolvida e C et `e` os argumentos. Ora, dado que escolhemos representar os contextos como funções, esta operação de composição corresponde precisamente à aplicação de um contexto ao seu argumento. A fim de especificar esta aplicação, utilizamos a projecção `post` sobre um contexto e a quantificação universal para referir que para qualquer expressão `res` que verifica `post c e res`, então `post c' e' res` é igualmente verificada. A pós-condição de `decompose_term` assegura ainda que a expressão `e'` devolvida é um *redex*.

Introduzimos agora a função `decompose` e a sua especificação. Esta função toma como argumento uma expressão que aplicamos a `decompose_term`, dando-lhe ainda como argumento o contexto vazio, aqui representado pela função identidade:

```
let decompose (e: exp) : (context, exp)
```

[3] O leitor notará que o código das funções `hole`, `sub_left` e `sub_right` se encontra expandido na construção dos contextos.

```
  requires { not (is_value e) }
  returns  { (c', e') → is_redex e' ∧ post c' e' e  }
= decompose_term e (fun x → x)
```

Para avaliar uma expressão `e` em um valor, introduzimos uma função de iteração que se comporta como o fecho transitivo da relação →. O seu código Why3 é o seguinte:

```
let rec red (e: exp) : int = match e with
  | Const v → v
  | _ → let (c, r) = decompose e in
      let r' = head_reduction r in
      red (c r')
  end
```

Se a expressão `e` é da forma `Const v`, esta encontra-se já em forma normal e por isso não pode ser reduzida. Se, por outro lado, `e` não é ainda uma constante temos então de (1) decompor esta expressão no *redex* `r` e no contexto `c`; (2) reduzir `r` à cabeça e obter a expressão `r'`; (3) continuar a iteração com a expressão obtida pela composição de `c` e `r'`. A composição materializa-se, de uma forma elegante, como a aplicação de `c` a `r'`. O valor devolvido pela chamada `red e` deve ser o mesmo que aquele que é devolvido por uma avaliação *big-step*. Equipamos assim `red` com o seguinte contrato:

```
let rec red (e: exp) : int
  ensures { result = eval e }
```

onde `eval e` representa a semântica natural da expressão `e`. Procedemos agora à desfuncionalização deste exemplo para mostrar a sua correcção em Why3. Começamos por introduzir o tipo `context` desfuncionalizado:

```
type context = CHole | CApp_left context exp | CApp_right int context
```

A partir das abstracções presentes no programa original, podemos gerar a função `apply` conjuntamente com a sua especificação, como procedemos para os exemplos anteriores. O predicado `post` pode ser deduzido a partir das pós-condições presentes no programa original:

```
predicate post (c: context) (arg result: exp) = match c with
  | CHole → let x = arg in x = result
  | CApp_left c e → let x = arg in post c (Sub x e) result
  | CApp_right v c → let x = arg in post c (Sub (Const v) x) result
  end
```

Para obter o programa desfuncionalizado final basta, assim como demonstrado para os exemplos anteriores, substituir todas as aplicações das abstracções por chamadas a `apply` e todas as ocorrências dessas mesmas abstracções pelo construtor respectivo do tipo `context` desfuncionalizado.

Todas as obrigações de prova geradas pelo Why3 para o programa desfuncionalizado e a respectiva especificação seriam provadas, excepto a validade da pós-condição da função `red`. Para melhor guiar os demonstradores automáticos e descartar esta pós-condição, introduzimos o seguinte lema auxiliar:

```
lemma post_eval: forall c arg1 arg2 r1 r2.
  eval arg1 = eval arg2 → post c arg1 r1 → post c arg2 r2 →
  eval r1 = eval r2
```

Este lema exprime que para qualquer expressão `arg1` e `arg2` cuja a avaliação produz o mesmo valor, então as expressões obtidas pela composição de `arg1` e `arg2` com o mesmo contexto `c` devem-se avaliar no mesmo valor. Provamos este resultado por indução sobre `c`. Com este lema auxiliar, a pós-condição de `red` é automaticamente descartada. O enunciado deste lema quantifica universalmente sobre valores do tipo `context`. Escrever tal enunciado directamente sobre o programa de ordem superior poderia conduzir a uma contradição, visto que não nos é possível afirmar tal resultado para qualquer função do tipo `exp` → `exp`. Ainda assim, seria interessante ter um mecanismo que nos permitisse escrever este género de enunciados no código original (restringindo as funções abstractas aceites), traduzindo-os automaticamente para o código desfuncionalizado.

4 Discussão e perspectivas

Neste artigo explorámos a utilização da desfuncionalização como uma técnica de prova para programas de ordem superior contendo potencialmente efeitos. O que propomos é que um programa de ordem superior seja directamente anotado, e em seguida desfuncionalizar este programa e a sua especificação para um programa de primeira ordem. Uma ferramenta de verificação dedutiva existente pode ser então utilizada para demonstrar a correcção deste programa. Ilustrámos a nossa proposta através de três exemplos desfuncionalizados (manualmente). Os programas de primeira ordem obtidos foram automaticamente verificados no sistema Why3. A utilização da desfuncionalização num contexto de prova é, tanto quanto sabemos, nova.

O trabalho apresentado neste artigo mantém-se, por agora, exploratório. Utilizámos a desfuncionalização para provar diversos exemplos de programas de ordem superior e os resultados encorajam-nos a continuar a exploração desta metodologia. O nosso objectivo é melhorar e formalizar a nossa utilização da desfuncionalização sobre programas de ordem superior. O primeiro passo será definir formalmente a classe de programas sobre os quais esta técnica pode ser aplicada, a fim de compreender se a desfuncionalização pode ser utilizada como uma técnica robusta de prova de programas de ordem superior com efeitos.

A função `apply`, gerada durante a desfuncionalização, procede por análise de casos sobre cada função presente no programa original, onde cada ramo corresponde à definição da abstracção correspondente. Assim, cada um destes ramos deve devolver um valor do mesmo tipo, o que é apenas verdade quando todas as abstracções do programa têm o mesmo tipo $\tau_1 \to \tau_2$. Para resolver este problema, Pottier e Gauthier [13] propuseram a utilização de *tipos algébricos generalizados* (do inglês *generalized algebraic data types*, GADT) para desfuncionalizar um programa contendo abstracções de diferentes tipos. Tal solução interessa-nos e por isso pretendemos estudar possíveis mecanismos para estender a linguagem e lógica do sistema Why3 com suporte para os GADT.

Finalmente, para melhor avaliar a sua utilidade, desejamos aplicar a prova por desfuncionalização a um caso de estudo mais complexo. Um bom candidato é o algoritmo de Koda-Ruskey [9] para a enumeração de todos os ideais de um conjunto parcialmente ordenado. É nossa intenção partir da implementação de ordem superior existente deste algoritmo [5], e provar a sua correcção através da nossa metodologia.

Referências

1. Ager, M.S., Biernacki, D., Danvy, O., Midtgaard, J.: A functional correspondence between evaluators and abstract machines. In: Proceedings of the 5th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, 27-29 August 2003, Uppsala, Sweden. pp. 8–19. ACM (2003)
2. Charguéraud, A.: Characteristic formulae for the verification of imperative programs. In: Chakravarty, M.M.T., Hu, Z., Danvy, O. (eds.) Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming (ICFP). pp. 418–430. ACM, Tokyo, Japan (September 2011)
3. Danvy, O., Millikin, K.: Refunctionalization at work. Science of Computer Programming **74**(8), 534–549 (2009), special Issue on Mathematics of Program Construction (MPC 2006)
4. Danvy, O., Nielsen, L.R.: Defunctionalization at work. In: Proceedings of the 3rd ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming. pp. 162–174. PPDP '01, ACM Press (2001)
5. Filliâtre, J.C.: La supériorité de l'ordre supérieur. In: Journées Francophones des Langages Applicatifs. pp. 15–26. Anglet, France (Jan 2002)
6. Filliâtre, J.C., Paskevich, A.: Why3 — where programs meet provers. In: Felleisen, M., Gardner, P. (eds.) Proceedings of the 22nd European Symposium on Programming. Lecture Notes in Computer Science, vol. 7792, pp. 125–128. Springer (Mar 2013)
7. Kanig, J.: Spécification et preuve de programmes d'ordre supérieur. Thèse de doctorat, Université Paris-Sud (2010)
8. Kanig, J., Filliâtre, J.C.: Who: A Verifier for Effectful Higher-order Programs. In: ACM SIGPLAN Workshop on ML. Edinburgh, Scotland, UK (Aug 2009)
9. Koda, Y., Ruskey, F.: A Gray Code for the Ideals of a Forest Poset. Journal of Algorithms (15), 324–340 (1993)
10. Nanevski, A., Morrisett, G., Shinnar, A., Govereau, P., Birkedal, L.: Ynot: Reasoning with the awkward squad. In: Proceedings of ICFP'08 (2008)
11. Pereira, M.: Défonctionnaliser pour prouver. In: Boldo, S., Signoles, J. (eds.) Vingt-huitièmes Journées Francophones des Langages Applicatifs. Gourette, France (Jan 2017)
12. Plotkin, G.D.: Call-by-name, call-by-value and the lambda-calculus. Theor. Comput. Sci. **1**(2), 125–159 (1975)
13. Pottier, F., Gauthier, N.: Polymorphic typed defunctionalization and concretization. Higher-Order and Symbolic Computation **19**, 125–162 (Mar 2006)
14. Régis-Gianas, Y., Pottier, F.: A Hoare logic for call-by-value functional programs. In: Proceedings of the Ninth International Conference on Mathematics of Program Construction (MPC'08). pp. 305–335 (2008)
15. Reynolds, J.C.: Definitional interpreters for higher-order programming languages. Higher-Order and Symbolic Computation **11**(4), 363–397 (Dec 1998)

A Cálculo da Altura de uma Árvore, Desfuncionalizado

A.1 Implementação OCaml

```
type 'a tree = E | N of 'a tree * 'a * 'a tree

type 'a kont = Kid | Kleft of 'a tree * 'a kont | Kright of 'a kont * int

let rec apply (k: 'a kont) arg = match k with
  | Kid → let x = arg in x
  | Kleft (r, k) → let hl = arg in height_tree_cps r (Kright (k, hl))
  | Kright (k, hl) → let hr = arg in apply k (1 + max hl hr)
and height_tree_cps t (k: 'a kont) = match t with
  | E → apply k 0
  | N (l, x, r) → height_tree_cps l (Kleft (r, k))

let height_tree t = height_tree_cps t Kid
```

A.2 Implementação Why3

```
type kont 'a = Kid | Kleft (tree 'a) (kont 'a) | Kright int (kont 'a)

let rec height_tree_cps (t: tree 'a) (k: kont 'a) : int
  ensures { post k (height t) result }
= match t with
  | Empty → apply k 0
  | Node l _ r → height_tree_cps l (Kleft r k)
  end
with apply (k: kont 'a) (arg: int) : int
  ensures  { post k arg result }
= match k with
  | Kid → arg
  | Kleft r k → height_tree_cps r (Kright arg k)
  | Kright hl k → apply k (1 + max hl arg)
  end

let height_tree (t: tree 'a) : int
  ensures { result = height t }
= height_tree_cps t Kid
```

B Número de Elementos Distintos de uma Árvore

```
function set_of_tree_acc (t: tree 'a) (s: set 'a) : set 'a =
  match t with
  | Empty → s
  | Node l x r → set_of_tree r (set_of_tree l (add x s))
  end

function set_of_tree (t: tree 'a) : set 'a = set_of_tree_acc t empty

predicate post (k: kont 'a) (arg: unit) (old cur: state 'a) (result: unit) =
  match k with
  | Kid → let () = arg in old == cur
  | Kleft r k → let () = arg in post k () (set_of_tree_acc r old) cur result
  | Kright k → let () = arg in post k () old cur result
  end

let n_distinct_elements (t: tree 'a) : int
  ensures { result = cardinal (set_of_tree t) }
= let h = ref empty in
  let rec apply (k: kont 'a) (arg: unit) : unit
    ensures { post k arg (old !h) !h () }
  = match k with
    | Kid → let x = arg in x
    | Kleft r k → let _ = arg in distinct_elements_loop r (Kright k)
    | Kright k → let _ = arg in apply k arg
    end
  with distinct_elements_loop (t: tree 'a) (k: kont 'a) : unit
    ensures { post k () (set_of_tree_acc t (old !h)) !h () }
  = match t with
    | Empty → apply k ()
    | Node l x r →
       h := add x !h;
       distinct_elements_loop l (Kleft r k)
    end in
  distinct_elements_loop t Kid;
  cardinal !h
```

C Interpretador *small-step*

```
type exp = Const int | Sub exp exp

type context = exp → exp

function eval (e: exp) : int = match e with
  | Const n → n
  | Sub e1 e2 → (eval e1) - (eval e2)
  end

predicate is_redex (e: exp) = match e with
  | Sub (Const _) (Const _) → true
  | _ → false
  end

let head_reduction (e: exp) : exp
  requires { is_redex e }
  ensures  { eval result = eval e }
= match e with
  | Sub (Const v1) (Const v2) → Const (v1 - v2)
  | _ → absurd
  end

predicate is_value (e: exp) = match e with Const _ → true | _ → false end

let rec decompose_term (e: exp) (c: context) : (contex, exp)
  requires { not (is_value e) }
  returns  { (c', e') → is_redex e' ∧
             forall res. post c e res → post c' e' res }
= match e with
  | Const _ → absurd
  | Sub (Const v1) (Const v2) → (c, e)
  | Sub (Const v1) e → decompose_term e (fun x →
      ensures { post c (Sub (Const v) x) result } c (Sub (Const v) x))
  | Sub (e1, e2) → decompose_term e1 (fun x →
      ensures { post c (Sub x e) result } c (Sub x e2))

let decompose (e: exp) : (context, exp)
  requires { not (is_value e) }
  returns  { (c', e') → is_redex e' ∧ post c' e' e }
= decompose_term (fun x → x) e

let red (e: exp) : int
  ensures { result = eval e }
= match e with
  | Const v → v
  | _ → let (c, r) = decompose e in
     let r' = head_reduction r in red (c r')
```