Verificação de Programas OCaml Imperativos de Ordem Superior, através de Desfuncionalização

Tiago Soares and Mário Pereira

NOVA LINCS – Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa

Resumo Neste artigo, apresentamos a técnica de desfuncionalização como uma abordagem prática e eficaz à verificação automática de programas imperativos de ordem superior. Tais programas, anotados com uma linguagem de especificação própria, são, em primeiro lugar, desfuncionalizados e posteriormente traduzidos para uma ferramenta de verificação deductiva automática, capaz de raciocinar sobre programas de primeira-ordem. Aplicamos esta técnica na análise de código escrito em linguagem OCaml, uma linguagem multi-paradigma que permite facilmente combinar computação de ordem superior com aspectos imperativos. Todos os casos de estudo reportados neste artigo foram inteiramente verificados em Cameleer, uma plataforma para a verificação deductiva de programas OCaml, a qual estendemos com a capacidade de desfuncionalização.

1 Introdução

Escrever código complexo é uma tarefa monumental, independentemente de quão hábil seja o programador. Se quisermos atacar as falhas presentes no nosso código antes que elas se tornem vulnerabilidades, temos apenas duas estratégias: testes e prudência; mas a prudência nunca será suficiente e testagem, nas palavras de Djikstra, *"can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence"* [5]. Logo, se quisermos ter a certeza que o nosso código funciona correctamente, teremos de utilizar técnicas baseadas no rigor matemático, isto é, provas formais de programas.

Existem vários famílias de propriedades de um programa que podem ser verificadas: ausência de erros de tipificação, transbordamentos aritméticos, exceções que não são tratadas explicitamente ou terminação. Mais geralmente, podemos querer relacionar o *input* do programa com o *output*/estado. A metodologia que adotamos neste trabalho é a verificação dedutiva [6] por duas razões fundamentais: (i) permite-nos raciocinar sobre propriedades arbitrariamente complexas (*e.g.*, correção funcional, segurança de execução); (ii) permite-nos raciocinar diretamente sobre código, em detrimento de modelos abstratos.

A adoção de uma linguagem funcional é normalmente considerada como um meio por excelência para obter, facilmente, programas correctos e seguros por construção. O carácter normalmente puro das implementações funcionais aproximam-nas de definições puramente lógicas, o que simplifica qualquer prova de correcção. No entanto, linguagens da família funcional, como por exemplo OCaml, permitem a combinação elegante entre computações que manipulam

funções como valores de primeira classe e efeitos imperativos. Esta flexibilidade acarreta, porém, uma dificuldade acrescida na verificação desta classe de programas, a mais óbvia sendo modelar a memória do programa utilizando construções lógicas. É precisamente sobre a prova de programas OCaml que combinam estas duas visões que nos centramos neste artigo.

De modo a especificar código com efeitos, precisamos de uma lógica de programas capaz de raciocinar sobre os ditos efeitos. Uma das alternativas mais populares é a Lógica de Separação [16], que nos permite escrever asserções sobre o conteúdo da memória. Devido ao seu poder expressivo, escrever e ler provas desta lógica é frequentemente demasiado complicado. Logo, utilizaremos GOSPEL (*Generic Ocaml SPEcification Language*) [3], uma alternativa de alto nível para descrever o estado do programa. De modo a provar que um determinado programa adere a um especificação GOSPEL utilizaremos Cameleer [13], uma ferramenta dedicada à prova dedutiva de programas escritos directamente em OCaml. Resumidamente, esta ferramenta funciona por tradução de um programa OCaml anotado para um equivalente em WhyML, a linguagem de especificação e verificação da plataforma de verificação Why3 [7]. Uma das características distintivas do Why3 é a sua interface com múltiplos *solvers* SMT, o que nos conduz naturalmente a um elevado grau de automação em provas.

Embora GOSPEL seja quase tão expressiva como Lógica de Separação, esta linguagem, tal como WhyML, não consegue capturar funções de ordem superior com efeitos. Estender GOSPEL, e as próprias ferramentas Cameleer e Why3, para suportarem nativamente esta classe de problemas é uma tarefa não trivial. Muito provavelmente, seria necessário introduzir uma noção de quantificação universal sobre todos os triplos de Hoare possíveis de uma função de ordem superior, o que rapidamente tornaria a prova automática impraticável senão mesmo impossível. Esta é a abordagem utilizada em lógicas de programa bastante ricas (*e.g.*, Iris [9]), onde não existe qualquer ambição de automação de provas e por isso estão, normalmente, fora do alcance do programador-*lambda*.

Neste artigo, apresentamos uma visão alternativa à prova de programas de ordem superior com efeitos. Sendo a ordem superior a nossa principal fonte de dificuldade, o que propomos é ultrapassar as limitações das ferramentas de prova automática através do uso da técnica de desfuncionalização [4, 15]. Esta técnica de transformação de programas permite a conversão de qualquer programa de ordem superior num equivalente de primeira ordem, que pode então ser fornecido a uma ferramenta de prova como o Why3. Ao contrário do que descrevemos em trabalho prévio [12], neste artigo apresentamos uma implementação da desfuncionalização para programas escritos numa linguagem de programação realista. Todo o fluxo de trabalho é automatizado através da incorporação desta técnica como uma extensão ao processo de tradução da ferramenta Cameleer. Assim, o utilizador apenas deverá preocupar-se em escrever uma implementação OCaml de ordem superior que será automaticamente desfuncionalizada, e posteriormente enviada ao Why3. Finalmente, reportamos a nossa utilização, bem sucedida, do *pipeline* proposto para a verificação automática de diversos programas OCaml de ordem superior.

2 Prova por Desfuncionalização

Nesta secção definimos a nossa técnica de prova baseada em desfuncionalização. O mecanismo de tradução que adotamos segue aquilo que é apresentado por Pottier e Gauthier [14]. Assim, focamo-nos aqui exclusivamente sobres aspectos particulares à nossa tradução. A principal novidade da nossa abordagem é a introdução de efeitos de escrita em memória, assim como a junção de elementos de especificação lógica. A nossa gramática de expressões é definida como se segue:

$$e ::= \ldots \mid \texttt{letrec}\ x = e\ \texttt{in}\ e \mid x := e \mid !x \mid \texttt{fun}\ x : \tau\, \bar{\mathcal{S}} \xrightarrow{\epsilon} e$$

As três primeiras construções são utilizadas, respectivamente, para introduzir um referência local com o nome `x`, para actualizar o valor da referência `x` e para aceder ao conteúdo atual de `x`. Por motivos de apresentação, restringimos os efeitos da nossa linguagem à manipulação de referências. Finalmente, a construção `fun` introduz uma função anónima com parâmetro formal `x`.

Dois aspetos na nossa definição sintática de funções anónimas merecem uma explicação mais detalhada: primeiro, a introdução de $\mathcal{S}$ como um *placeholder* para especificação lógica; segundo, a anotação ϵ sobre a seta. Começando por este último, ϵ representa o efeito produzido pela aplicação desta função. A nossa definição de efeito é, neste contexto, relativamente simples, sendo ϵ definido como o conjunto de referências cujo valor é alterado pela execução da função[1]. Quanto à especificação $\mathcal{S}$, esta representa o contrato da função, da seguinte forma:

$$\mathcal{S} ::= \texttt{requires}\ t \mid \texttt{ensures}\ t$$

As anotações `requires` e `ensures` são usadas, respectivamente, como cláusulas de pré-condição e a pós-condição da função. Já a variável `t` representa uma linguagem de termos, extraída de uma lógica de programas de primeira ordem, como por exemplo WhyML, JML [11] ou ACSL [1]. A única restrição que colocamos ao nível da linguagem lógica é que esta forneça o par de predicados lógicos `pre/post`, tal como introduzido por Régis-Gianas e Pottier [17], e Kanig e Filliâtre [10]. Resumidamente, estes predicados representam de forma abstracta, no interior de uma fórmula lógica, os contratos de uma função de programa. Assim, o contrato de uma função f do tipo $\tau_1 \to \tau_2$ é representada como se segue:

$$\texttt{pre}_f : \tau_1 \to \epsilon_i \to \texttt{prop}$$
$$\texttt{post}_f : \tau_1 \to \epsilon_i \to \epsilon_f \to \tau_2 \to \texttt{prop}$$

No primeiro caso, o conjunto ϵ_i representa o estado da computação antes da chamada à função; no caso do predicado `post` este é parametrizado pelo estado antes da chamada à função e pelo estado após a sua execução (ϵ_i e ϵ_f, respectivamente). Nos casos de estudo que apresentaremos nas secções seguintes,

[1] Na realidade, o conjunto ϵ é uma sobre-aproximação dos efeitos realmente produzidos pelo corpo da função. Na presença de certas construções sintáticas (*e.g.*, condicionais da forma `if..then..else`), é impossível determinar, estaticamente, o conjunto efectivo de efeitos produzidos por determinada computação.

utilizaremos GOSPEL como linguagem de especificação, a qual estendemos para a utilização do par `pre/post`.

Na nossa linguagem, uma declaração, representada por d, pode tomar duas formas: ou uma expressão de programa ou uma declaração lógica (*e.g.*, predicados ou lemas auxiliares). Um programa p é uma lista de declarações, representada por $\bar{d}$. Assim, o nosso mecanismo de desfuncionalização irá gerar, a partir de p (programa de ordem superior), o seguinte programa de primeira ordem:

$$\texttt{predicate pre}_{\tau_1 \to \tau_2} = \texttt{fun } f : \texttt{Arrow}_{\tau_1 \to \tau_2} \to \texttt{fun arg} : \tau_1 \to \texttt{fun } \epsilon_i : \epsilon \to$$
$$\texttt{case } f \texttt{ of } \overline{c_{\texttt{pre}}}$$

$$\texttt{predicate post}_{\tau_1 \to \tau_2} = \texttt{fun } f : \texttt{Arrow}_{\tau_1 \to \tau_2} \to \texttt{fun arg} : \tau_1 \to$$
$$\texttt{fun } \epsilon_i : \epsilon \to \texttt{fun } \epsilon_f : \epsilon \to \texttt{fun res} : \tau_2 \to$$
$$\texttt{case } f \texttt{ of } \overline{c_{\texttt{post}}}$$

$$\texttt{letrec apply}_{\tau_1 \to \tau_2} = \texttt{fun } f : \texttt{Arrow}_{\tau_1 \to \tau_2} \to \texttt{fun arg} : \tau_1$$
$$\texttt{requires pre } f \texttt{ arg old}(\epsilon)$$
$$\texttt{ensures post } f \texttt{ arg old}(\epsilon)\ \epsilon\ \texttt{result} \to$$
$$\texttt{case } f \texttt{ of } \overline{c}_p$$

$$p'$$

Neste esquema, `p'` representa a desfuncionalização do corpo do programa original `p`, nomeadamente a substituição de expressões `fun` pela sua versão desfuncionalizada e da aplicação de funções por chamadas à função `apply`. Seguindo a apresentação original de Pottier e Gauthier, o tipo `Arrow` representa o tipo de funções desfuncionalizadas. Aqui, como não tratamos polimorfismo, anotamos cada tipo `Arrow` com $\tau_1 \to \tau_2$, representando assim a família de tipos de primeira ordem resultante da desfuncionalização de um termo `fun` de tipo $\tau_1 \to \tau_2$. Finalmente, o corpo gerado para os predicados `pre` e `post` realiza uma análise por casos sobre a forma da função f desfuncionalizada. Para cada termo `fun` do programa original anotado com o predicado `post` (resp. `pre`), a sequência $\overline{c_{\texttt{post}}}$ (resp. $\overline{c_{\texttt{pre}}}$) contém um ramo da forma $\texttt{m}\,\bar{\tau} \mapsto \texttt{post} \ldots$ (resp. `pre`), em que m representa a `tag` única atribuída a cada função desfuncionalizada e `post ...` (resp. `pre`) uma chamada ao predicado desfuncionalizado. De notar, ainda, que utilizamos a *keyword* `result` para nomear o resultado devolvido por uma função.

3 Breve Descrição da Extensão à Ferramenta Cameleer

De forma resumida, a ferramenta Cameleer irá transformar as anotações de GOSPEL e o programa OCaml num programa WhyML equivalente. Anotações GOSPEL são colocadas como comentário no final de funções ou, no caso de funções anónimas, entre a *keyword* `fun` e a lista de argumentos:

```
fun (*@ ensures Q *) arg... -> exp
```

Embora GOSPEL tenha uma sintaxe bastante rica, neste artigo utilizaremos apenas duas construções, as cláusulas *ensures* e *requires* que introduzem pós e

pré-condições, respetivamente. De modo a perceber como a nossa extensão de GOSPEL lida com funções de ordem superior, analisemos o seguinte exemplo:

```
let f g = ...
(*@ r = f g
      ensures r = g 42 *)
```

Esta é uma especificação de uma função `f` que recebe um argumento `g`, outra função, e cujo resultado, `r`, é igual à aplicação de `g` a 42. Embora, superficialmente, esta especificação pareça legítima, nós utilizamos, num contexto lógico, uma função de programa executável. Tal pode resultar em incoerências lógica se `g` tiver algum comportamento impuro, como, por exemplo, modificação de estruturas mutáveis ou divergência. De modo a resolver este problema, utilizaremos o predicado `post`, como se segue:

```
(*@ r = f g
      ensures post g 42 r *)
```

Na tradução que implementamos em Cameleer, o predicado `post` que utilizaremos terá a seguinte estrutura quando traduzido para WhyML:

```
predicate postN (k : kontN) (arg : int) (result : int) =
  match k with
  | K -> let x = arg in ... (* pos-condicao equivalente *)
```

Um detalhe importante sobre o predicado é o seu nome: este não tem nome de `post`, mas sim `postN`. Isto verifica-se porque teremos um predicado `post` para cada tipo de função presente no nosso programa; uma das responsabilidades do nosso tradutor é, dado uma aplicação de `post`, substituí-la pelo predicado desfuncionalizado correto. O predicado `post` é construído de forma semelhante à que construimos o `apply`, mas em vez de cada ramo ter o código correspondente a cada função, esta terá a sua pós-condição.

No caso de querermos descrever mudanças no estado do programa, o predicado `post` terá como argumentos suplementares o estado inicial do programa quando a continuação foi chamada e o estado final. Para o exemplo seguinte

```
let n = ref 0 in
let m = ref 0 in

let f (k : unit -> unit) : unit =
  let n := !m in
  k ()
```

se soubermos que `k` pode modificar a referência `n`, devemos explicitar o seu estado inicial e final quando a continuação é chamada. Uma vez que o seu estado inicial é igual ao estado inicial de `m` e o seu estado final é igual ao seu estado no fim do nosso programa, a sua pós condição será:

```
(*@ f k
      ensures post k () (old !m) !n () *)
```

A *keyword* `old` denota o valor lógico de `!m` antes da função ser chamada. Porém, a nossa prova ainda está incompleta, pois a nossa ferramenta não saberá a que variável estamos a referir, `n` ou `m`. De modo a resolver esta ambiguidade, introduzimos uma nova função lógica `state` para produzir um *snapshot* completo do estado do nosso programa. Esta função espera um número arbitrário de argumentos (deve ser instanciado segundo a natureza do estado de cada programa), em que cada argumento é um par, contendo o nome de uma referência e o seu valor correspondente. Esta função cria um tuplo que contém o valor de todas as variáveis de estado. Se não especificamos o valor, o `state` preenche com o valor antes ou depois de função ser executada, dependendo do contexto. Dado isto, re-escrevemos o exemplo anterior como se segue:

```
(*@ f k
      ensures post k () (state (n, old !m)) (state (n, !n)) () *)
```

Dado que não especificamos o valor de `m` em nenhuma das chamadas, o nosso tradutor irá utilizar, por defeito, `(m, old !m)` na primeira aplicação de `state` e `(m, !m)` na segunda. De modo a finalizar esta secção, notamos que a função `state`, tal como o predicado `post`, não façam parte da definição original da linguagem WhyML. Trata-se de uma meta-função que o nosso tradutor utiliza de modo a facilitar a conversão e manter as definições de estado o mais simples que possível. A ferramenta Cameleer irá assim gerar a seguinte definição em WhyML:

```
predicate postN (k: kontN) (arg: unit) (old_state: int, int)
  (state: int, int) (result: unit) =
  match state with
  | n, m ->
      match old_state with
      | old_n, old_m ->
          match k with
          | K -> ...  (* pos-condicao equivalente *)
```

4 Casos de Estudo

4.1 Altura de uma árvore binária

O primeiro exemplo que iremos analisar será uma função que calcula a altura de uma árvore, com o seguinte esqueleto:

```
let height_tree (t: int tree): int = ...
(*@ r = height_tree t
      ensures (height t) = r *)
```

A pós-condição da função `height_tree` utiliza a função lógica auxiliar `height` como definição matemática da altura de uma árvore binária. Embora uma função que calcula a altura de uma árvore devia ser agnóstica ao seu tipo, esta só

aceita árvores de inteiros, dado que desfuncionalizar programas com variáveis polimórficas está, por agora, fora do escopo do nosso projeto.

Em detrimento de definirmos `height_tree` seguindo a definição recursiva *by-the-book* de cálculo da altura de uma árvore binária, escrevemos aqui este programa em *CPS* (*Continuation Passing Style*), evitando assim qualquer problema de transbordamento de pilha. Primeiramente, definimos uma função auxiliar que, dado uma continuação, calcule a altura de uma árvore.

```
let rec height_tree_cps (t: int tree) (k: int -> int) : int
  = match t with
    | Empty -> k 0
    | Node(lt, _, rt) ->
        height_tree_cps lt (fun
          (*@ ensures post k (1 + max hl (height rt)) result *)
          (hl : int) ->
        height_tree_cps rt (fun
          (*@ ensures post k (1 + max hl hr) result *)
          (hr : int) -> k (1 + max hl hr)))
(*@ r = height_tree_cps t k
      ensures post k (height t) r *)
```

Por razões de simplicidade, omitimos aqui a utilização dos argumentos de estado.

Resumidamente, esta função começa por calcular a altura da sub-árvore esquerda e passa este resultado como argumento `hl` da continuação da primeira chamada recursiva. Depois de calcular a altura da sub-árvore direita, argumento `hr`, na continuação final apenas é necessário calcular o máximo entre as alturas das duas sub-árvores. Para obter uma implementação correcta, a nossa função principal deverá chamar a função `height_tree_cps` com a continuação identidade, como se segue:

```
let height_tree (t: int tree): int =
  height_tree_cps t (fun (*@ ensures result = x *) (x : int) -> x)
(*@ r = height_tree t
      ensures (height t) = r *)
```

Depois de desfuncionalizado e traduzido para WhyML pela ferramenta Cameleer, todas as condições de verificação geradas para este programa são automaticamente descartadas pelo *prover* CVC4, em menos de 1 segundo.

4.2 Interpretador *Small-step*

O próximo exemplo é um interpretador para uma pequena linguagem que apenas suporta inteiros e subtrações:

```
type exp = Const of integer | Sub of exp * exp
```

De modo a definir como expressões são avaliadas, apresentaremos duas regras semânticas. A primeira, que declara que um nodo *Sub* com dois *Const* pode ser imediatamente reduzido a um *Const*:

$$Sub(Const\ v1)(Const\ v2) \xrightarrow{e} Const(v1 - v2)$$

A outra regra, representa a típica redução contextual:

$$\frac{e \xrightarrow{e} e'}{C[e] \xrightarrow{e} C[e']}$$

onde um contexto C é representado pela seguinte gramática:

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

Resumidamente, $C[e]$ representa um contexto onde a expressão e substitui completamente o buraco $\square$. Dado isto, a nosso regra anterior pode ser lida, informalmente: se uma expressão pode ser reduzida a um e' simplificado, qualquer contexto $C[e]$ pode ser reduzido a $C[e']$.

Com o nosso interpretador definido, vamos considerar como definir os contextos num programa OCaml. Embora a abordagem mais natural seja criar um tipo algébrico com três construtores, um para cada um dos ramos, no nosso caso representamos tais contextos como funções de primeira ordem que recebem uma expressão e devolvem outra expressão. Tal abordagem conduz a um código mais elegante e modular, apesar de ordem superior.

O próximo passo será implementar a segunda regra de tradução (não iremos mostrar a implementação da primeira dado que ela é trivial). A nossa função recebe uma expressão e um contexto de redução, simplificando a expressão e criando um novo contexto, provando que $C[e] = C'[e']$.

```
let rec decompose_term (e: exp) (c : exp -> exp) :
  ((exp -> exp) * exp)
= match e with
  | Const _ -> assert false
  | Sub (Const (v1 : int), Const (v2 : int)) -> (c, e)
  | Sub (Const (v : int), (e : exp)) -> decompose_term e
      (fun (*@ ensures post c (Sub (Const v) x) result *)
        (x : exp) -> c (Sub (Const v, x)))
  | Sub ((e1 : exp), (e2 : exp)) -> decompose_term e1
      (fun (*@ ensures post c (Sub x e2) result *)
        (x : exp) -> c (Sub (x, e2)))
(*@ c_res, e_res = decompose_term e c
      requires not (is_value e)
      ensures  is_redex e_res &&
        forall res. post c e res -> post c_res e_res res *)
```

O primeiro ramo deste *pattern-matching* assegura que o caso em que `e` é uma constante é um ponto inatingível no código. Tal é verdade dada a pré-condição `not (is_value e)`, isto é, a expressão `e` ainda não está totalmente avaliada.

Tal como no caso de estudo anterior, criamos uma função que chama a nossa auxiliar com a função identidade:

```
let decompose (e: exp) : (exp -> exp) * exp =
  decompose_term e (fun (*@ ensures result = x *) (x : exp) -> x)
(*@ c_res, e_res = decompose e
      requires not (is_value e)
      ensures  is_redex e_res && post c_res e_res e *)
```

Visto que já dispomos de todas as ferramentas para decompor as nossas expressões, o último passo é simplesmente traduzi-las. Dada uma expressão, se não for uma constante, o nosso programa irá a decompô-la num par formado por um contexto e uma expressão. Como esta expressão é uma subtração entre duas constantes, podemos aplicar a primeira regra de redução e aplicar o resultado ao contexto. Repetimos este processo até chegarmos até a uma constante.

```
let rec red (e : exp) : int = match e with
  | Const (v : int) -> v
  | _ -> let (c, r) = decompose e in
         let r = head_reduction r in red (c r)
(*@ r = red e
      ensures r = eval e *)
```

Todas as condições de verificação geradas para a versão desfuncionalizada deste algoritmo são automaticamente descartadas. De modo a provar a pós-condição da função `red` precisamos, porém, de introduzir o seguinte lema auxiliar:

```
lemma post_eval:
  forall c: exp -> exp, arg1: exp, arg2: exp, r1: exp, r2 : exp.
    eval arg1 = eval arg2 -> post c arg1 r1 -> post c arg2 r2 ->
    eval r1 = eval r2
```

A prova deste lema requer uma análise por casos no contexto `c`, feita diretamente no IDE interactivo do Why3.

4.3 Número de elementos distintos numa árvore binária

Concluindo esta seccção, analisamos o exemplo da prova de uma implementação CPS com efeitos, algo em falta nos exemplos anteriores. Este programa calcula o número de elementos distintos numa árvore binária utilizando uma referência a um conjunto:

```
module S = Set.Make(struct type t = int let compare = compare end)

(*@ function fset_of_tree (t: int tree) (s: int S.fset) : int S.fset =
      match t with
      | Empty -> s
```

```
      | Node l x r ->
          let s1 = fset_of_tree l (S.add x s) in
          fset_of_tree r s1 *)

let h : S.t ref = ref (S.empty ())

let rec distinct_elements_loop (t: int tree) (k: unit -> unit) =
  match t with
  | Empty -> k ()
  | Node((l : int tree), (x : int), (r : int tree)) ->
      h := S.add x !h;
      distinct_elements_loop l (fun
      (*@ ensures post k () (state (h, set_of_tree r (old !h)))
            (state (h, !h)) () *) () -> distinct_elements_loop r k)
(*@ r = distinct_elements_loop t k
      ensures post k () (state (h, set_of_tree t (old !h)))
                (state (h, !h)) ()*)
```

Em suma, o algoritmo coloca todos os elementos num conjunto e no final, verifica a sua cardinalidade. A ideia geral deste algoritmo é bastante semelhante à do primeiro exemplo: adicionamos o elemento ao conjunto e depois chamamos a função novamente passando o ramo direito e uma continuação que percorre o ramo esquerdo. A grande diferença é que estas continuações vão produzir efeitos na memória do programa, logo necessitamos de especificar o estado inicial e final em que chamamos `k`. A continuação `k` é apenas chamada quando percorremos a árvore inteira. Logo, na pós condição de `distinct_elements_loop`, diremos que o estado inicial será o conjunto inicial, ao qual juntamos os elementos restantes da árvore. Na continuação que criamos na função, como só a chamamos após de percorrer a árvore direita, todos os elementos da mesma já estão incluídos em `h`. Assim, só nos resta adicionar os elementos de `r`. Dado que a última computação que esta função fará é chamar o `k`, o estado final de `k` é igual ao estado final de todas as continuações.

Finalmente, criamos a função principal que passará a `distinct_elements_loop` a função identidade de modo a produzir o resultado esperado:

```
  let n_distinct_elements (t: int tree) : int=
    h := S.empty ();
    distinct_elements_loop t (fun (*@ ensures !h = (old !h) *)
      (x : unit) -> x);
    S.cardinal !h
  (*@ r = n_distinct_elements t
        ensures r = S.cardinal (fset_of_tree t S.empty) *)
```

De notar a inicialização da referência `h` com o conjunto vazio, antes de ser iniciado o cálculo principal do algoritmo. Todas as condições de prova geradas para este programa são automaticamente provada pelo *prover* Alt-Ergo, em menos de 1 segundo.

5 Trabalho Relacionado

Embora este seja, tanto quanto sabemos, o primeiro trabalho a combinar desfuncionalização com provas de programas de ordem superior, houve já vários projetos que visam provar esta mesma classe de programas A ferramenta Who [10], por exemplo, estende Lógica de Hoare de modo a podermos raciocinar sobre efeitos de funções de ordem superior. Esta abordagem, todavia, é bastante densa e difícil de utilizar. Alternativamente, uma das vantagens da desfuncionalização é o facto de ser relativamente simples de implementar, dado que podemos reutilizar a infraestrutura que utilizamos para provar programas de primeira ordem; ademais, é bastante simples de utilizar.

Existe também CFML [2], a única outra ferramenta que conhecemos que utiliza verificação deductiva para provar programas escritos directamente em OCaml. Esta traduz programas para Coq, um assistente de prova interativo, e valida-os utilizando Lógica de Separação, conseguindo provar algoritmos com funções de ordem superior e com efeitos.

Finalmente, a Lógica de Separação Concorrente Iris [9] utiliza Coq, de modo a provar programas de ordem superior. Esta é uma das ferramentas mais poderosas que existe na esfera de verificação de programa, mas a sua expressividade vem com a desvantagem de ser quase impossível escrever provas sem ser um perito em verificação. Adicionalmente, é extremamente difícil automatizar provas com o Iris e CFML, esta sendo a razão principal pelo qual ambos saem do escopo do nosso projeto.

6 Conclusão e Perspectivas

Neste artigo, propomos uma extensão à ferramenta Cameleer que permite verificar programas que combinam ordem superior com efeitos. Para este efeito, a nossa extensão começa por desfuncionalizar uma implementação OCaml, anotada como elementos GOSPEL, de modo a obter um programa equivalente de primeira ordem. Finalmente, este programa é enviado à ferramenta de prova Why3, perfeitamente habilitada ao raciocínio sobre programas de primeira ordem com efeitos. Tanto quanto estamos cientes, esta é a primeira vez que a desfuncionalização é utilizada como uma técnica de prova para programas escritos numa linguagem realista. Os resultados obtidos são bastante promissores.

Embora o nosso tradutor suporte já um sub-conjunto interessante de programas OCaml, queremos continuar a explorar os limites desta abordagem. Um dos objetivos futuros neste projeto será utilizar a nossa ferramenta para provar um exemplo maior que os nossos casos de estudo actuais, possivelmente o algoritmo Koda-Ruskey [8], ou uma biblioteca OCaml realista. Ademais, pretendemos também resolver algumas das limitações com o nosso provador, nomeadamente, adicionar suporte para funções polimórficas. Para concretizar este objetivo, necessitaremos de extender Why3 com *Generalized Algebraic Data Types* [14]. Tal implica, sem dúvida, uma extensão profunda do *core* da plataforma Why3, nomeadamente o seu sistema de tipos e gerador de condições de verificação.

Referências

1. Baudin, P., Filliâtre, J.C., Marché, C., Monate, B., Moy, Y., Prevosto, V.: ACSL: ANSI/ISO C Specification Language, version 1.4 (2009), http://frama-c.cea.fr/acsl.html
2. Charguéraud, A.: Characteristic formulae for the verification of imperative programs **46**(9), 418–430 (Sep 2011), https://doi.org/10.1145/2034574.2034828
3. Charguéraud, A., Filliâtre, J., Lourenço, C., Pereira, M.: GOSPEL — Providing OCaml with a Formal Specification Language. In: 23rd International Symposium on Formal Methods (2019), 10.1007/978-3-030-30942-8_29
4. Danvy, O., Nielsen, L.R.: Defunctionalization At Work. In: International Conference on Principles and Practice of Declarative Programming (PPDP) (2001). https://doi.org/10.1145/773184.773202
5. Dijkstra, E.W.: The Humble Programmer, p. 1972. Association for Computing Machinery, New York, NY, USA (2007), https://doi.org/10.1145/1283920.1283927
6. Filliâtre, J.C.: Deductive Software Verification. International Journal on Software Tools for Technology Transfer (STTT) **13**(5), 397–403 (Aug 2011), 10.1007/s10009-011-0211-0
7. Filliâtre, J.C., Paskevich, A.: Why3 — Where Programs Meet Provers. In: Programming Languages and Systems. pp. 125–128. Springer Berlin Heidelberg (2013)
8. Filliâtre, J.C., Pereira, M.: Producing All Ideals of a Forest, Formally (Verification Pearl) (May 2016), https://hal.inria.fr/hal-01316859, working paper or preprint
9. Jung, R., Krebbers, R., Jourdan: Iris From The Ground Up: A Modular Foundation For Higher-order Concurrent Separation Logic. Journal of Functional Programming **28** (2018)
10. Kanig, J., Filliâtre, J.C.: Who: A Verifier for Effectful Higher-order Programs. In: Workshop on ML. Edinburgh, Scotland (Aug 2009), 10.1145/1596627.1596634
11. Leavens, G.T., Baker, A.L., Ruby, C.: Preliminary design of JML: a behavioral interface specification language for java. ACM SIGSOFT Softw. Eng. Notes **31**(3), 1–38 (2006), https://doi.org/10.1145/1127878.1127884
12. Pereira, M.: Desfuncionalizar para provar. In: 11° INForum – Simpósio de Informática (2019), arXivpreprintarXiv:1905.08368
13. Pereira, M., Ravara, A.: Cameleer: a Deductive Verification Tool for OCaml. In: 33rd International Conference on Computer Aided Verification (2021), To appear
14. Pottier, F., Gauthier, N.: Polymorphic typed defunctionalization and concretization. Higher-Order and Symbolic Computation **19**, 125–162 (Mar 2006), http://cambium.inria.fr/~fpottier/publis/fpottier-gauthier-hosc.pdf
15. Reynolds, J.C.: Definitional Interpreters for Higher-Order Programming Languages. In: Proceedings of the ACM Annual Conference - Volume 2. p. 717–740. ACM '72 (1972), 10.1145/800194.805852
16. Reynolds, J.C.: Separation logic: A Logic For Shared Mutable Data Structures. In: Symposium on Logic in Computer Science. pp. 55–74. IEEE (2002)
17. Régis-Gianas, Y., Pottier, F.: A Hoare Logic for Call-by-Value Functional Programs. In: International Conference on Mathematics of Program Construction (MPC) (Jul 2008), http://dx.doi.org/10.1007/978-3-540-70594-9_17