Programação funcional com estilo e sem custo: Transformação CPS "à la carte"*

Tiago Roxo[1], Mário Pereira[2], and Simão Melo de Sousa[1]

[1] UBI, Covilhã, Portugal
[2] NOVA – LINCS

Resumo Neste artigo explora-se o uso de CPS como resultado de transformações automáticas de programas. CPS é um estilo de programação que, entre outras características, permite transformar qualquer função recursiva não terminal numa recursiva terminal e usa espaço constante da pilha implícita. Por estas razões, erros de tipo *Stack Overflow* são assim mitigados neste estilo de programação, admitindo que o compilador subjacente otimiza chamadas recursivas terminais. A modificação de código de programa para este estilo foi conseguido por recurso a uma extensão sintática do OCaml, designada por PPX. Neste artigo são apresentados os mecanismos associados à automatização de transformação para CPS, de uso transparente para o programador, avaliando as diferenças de desempenho derivadas desta.

1 Introdução

Programação para resoluções de problemas, de forma eficiente, nem sempre é facilmente conseguido, exigindo muitas vezes o repensar da abordagem a tomar. Este tipo de problemas é acrescido quando o número de dados a lidar é considerável e problemas como *Stack Overflow* podem advir do estilo de implementação escolhido. Dentro do contexto pedagógico, onde se insere este artigo, a utilização de um estilo de programação onde seja possível a visualização, avaliação e implementação de mecanismos automáticos de instrumentação é outro dos desafios apresentados. Um estilo de programação, capaz de reponder a estes desafios e garantir a eficiente implementação de soluções é o *Continuation-Passing Style* (CPS).

O uso de CPS permite eliminar a redundância aparente entre as diferentes chamadas e retorno da mesma função [4]. Por outras palavras, uma função nunca retorna em CPS até completar o seu cálculo. O conceito de uma função não aninhar as suas chamadas para o cálculo de valores intermédios deriva do facto de, em vez de retornar um valor `p`, passamos este para uma continuação. Podemos interpretar a continuação como sendo uma avaliação de contexto, representada por uma abstração λ, e a cada abstração λ é passada a continuação, para além dos restantes argumentos da função . Deste modo, todos os resultados intermédios

* Financiado por *Tezos Foundation*, incluído no projeto FACTOR: *Functional ApproaCh Teaching pOrtuguese couRses*.

são passados à continuação e portanto todas as chamadas a funções CPS serão recursivas terminais [5].

Outra característica do CPS é a definição da ordem de avaliação de expressões. A avaliação de expressões λ, em geral, depende da ordem de avaliação dos seus argumentos. Neste sentido, a independência de ordem de avaliação foi uma das motivações associadas ao uso continuações [6,7], tendo o CPS sido desenvolvido, inicialmente, como um avaliador, independente de ordem, de expressões λ [8,9].

A utilização de CPS tem a si associado o tornar da pilha explícita. De facto, a particularidade de CPS não necessitar de controlo de pilha e não deixar à decisão de qual a ordem de avaliação de subexpressões do programa, como referido anteriormente, foi uma das motivações associadas ao seu uso em compilador de linguagem [10]. A noção de tornar a pilha explícita está associada à ideia que, em CPS, um programa usa espaço constante na pilha implícita, e a continuação, alocada na memória dinâmica (*heap*), desempenhará as funções de pilha. Neste sentido, um programa deixa de ser realmente recursivo, passando a ser iterativo. Por estas características, erros de tipo *Stack Overflow* são assim mitigados neste estilo de programação. As transformações CPS são transformações clássicas, refira-se por exemplo [14], ou ainda uma versão otimizada desta transformação [15]. Para uma revisão completa e atual deste tema, convidamos o leitor a consultar o trabalho de F. Pottier [16].

Um dos problemas do uso deste estilo é a exigência imposta ao programador de escrever uma implementação pouco natural ou intuitiva. Esta noção ganha ainda maior importância se contextualizarmos o artigo, no projeto *Functional ApproaCh Teaching pOrtuguese couRses* (FACTOR), que visa a implementação desta técnica no contexto de ensino. Neste cenário, pretendemos manipular programas escritos por alunos, sem os obrigar a escrever diretamente em CPS. Concentramos os esforços dos alunos na formulação natural de soluções, delegando a extensão sintática desenvolvida para fazer a tradução para CPS. Neste ambiente é também importante destacar a facilidade que o CPS propicia a nível de manipulação algébrica e manipulação de programas, via *Abstract Syntax Tree* (AST), uma estrutura interna de dados.

A contribuição deste artigo é o facultar de uma extensão, incorporada no processo de compilação OCaml, capacitada para promover a transformação automática de expressões do programa do utilizador e de alunos, considerando o contexto de pedagógico onde o artigo se insere. Esta extensão faz uso de tecnologias de modificação de código e de expressões sintáticas (e.g %CPS), presente em OCaml, criadas para o uso de reescrita de código.

2 Transformação para CPS

Nesta secção, apresentamos as tecnologias e abordagens seguidas para promover a transformação automática de um código OCaml em CPS. Na secção 2.1 será apresentada a extensão OCaml usada para que, aquando do processo de compilação, fossem promovidas as alterações necessárias para produzir um código em

CPS. Nas secções 2.2 e 2.3 apresentamos uma transformação intermédia para CPS, denominada *Administrative Normal Form* (ANF), sendo enunciados os propósitos desta escolha e de que forma esta permitiu uma transformação mais eficiente. Por fim, na secção 2.4, são ilustradas as particularidades associadas à transformação de código, em CPS, quando este contém exceções.

A definição da semântica a utilizar, aquando da modificação de código, é um aspecto importante a considerar. A definição formal de um sub-conjunto da linguagem OCaml, assim como as diferentes funções de transformação de código para CPS, encontram-se no anexo A.

2.1 *PreProcessor eXtensions* (PPX)

Para implementar transformação de código para CPS, recorreu-se a uma extensão sintática de OCaml, conhecida por PreProcessor eXtensions (PPX), que permite adicionar novas sintaxes e funcionalidades à linguagem OCaml. Esta extensão procura por pequenas "âncoras"sintáticas, (%CPS, e.g) que sinalizam a esta que deve modificar o código nesse local. Um exemplo de aplicação desta "âncora" pode ser visualizada no cabeçalho da função, no excerto de codigo 1.1.

Excerto de código 1.1: Aplicação de "âncora" sintática, visando a transformação automática para CPS.

```
let %CPS rec fact x = if x = 0 then 1 else x * fact (x - 1)
```

Neste sentido, a linguagem OCaml tem uma sintaxe desenhada para uso de reescrita [1], usando PPX, nomeadamente "nodos de extensão", que seguem a nomenculatura [%nome_extensão]. Esta abordagem foi a implementada neste artigo e, assim, para que um utilizador queira ter parte do seu código transformado em CPS, de forma transparente e automática, simplesmente terá de adicionar a sintaxe referida na expressão a transformar. Exemplos de transformação são ilustrados na secção 3.

PPX no processo de compilação. Feita a exposição de como invocar a transformação automática para CPS, importa agora referir de que modo esta transformação é feita ao nível de compilador.

O processo de compilação OCaml passa por diferentes fases. A figura 1 exibe os principais passos associados a este processo, evidenciando o local de incorporação de PPX para promoção de transformação CPS. Note-se que, aquando da leitura de um programa, o primeiro passo que compilador faz é converter código fonte numa estrutura interna de dados, denominada por AST. A implementação de PPX avalia a AST criada e procura por construtores associados (sintaxes do tipo nodo de extensão, referidos anteriormente), visando transformação de código. Ao encontrar tais construtores, a extensão promove a reescrita de código, segundo as funcionalidades associadas a esta, convertendo uma AST válida (proveniente do *parser* do compilador OCaml) numa outra AST válida. Uma vez que a AST é construída antes da avaliação de tipos, otimização e geração de código,

como observado na figura 1, as transformações imposta por PPX são puramente sintáticas.

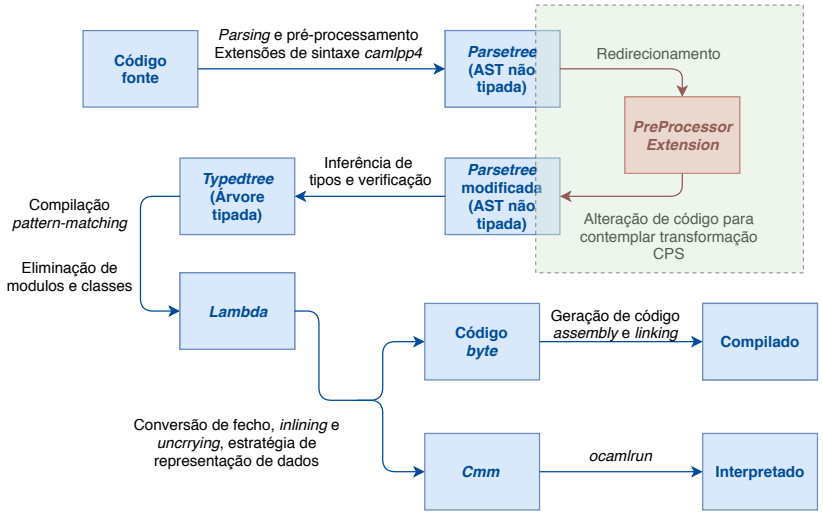

Figura 1: Pipeline de compilação em OCaml, evidenciando a incorporação de PPX, visando a transformação em CPS. Adaptado de [2].

2.2 Administrative Normal Form (ANF)

Administrative Normal Form (ANF), introduzida por Sabry e Felleisen no início da decada de 90 [11], é uma técnica de programação, usada como representação intermédia em alguns compiladores, que tem como finalidade tornar a ordem de avaliação de argumentos de chamadas explícita; em ANF, todos os componentes de cada aplicação são expressões atómicas. Neste sentido, surge como uma alternativa ao CPS, caso seja desejável apenas tornar explícita a ordem de avaliação [12].

O uso de ANF neste artigo serve o propósito de fornecer uma transformação intermédia para CPS. Nesse sentido, a transformação ANF não foi aplicada em toda a sua extensão (ou seja, nem todos os componentes de todas as aplicações serão valores) mas apenas para os casos que benefeciariam a transformação para CPS. O excerto de código em 1.2, evidencia a transformação de código direto para ANF.

Excerto de código 1.2: Cálculo de altura de árvore em estilo direto e ANF.

```
type t = E | N of t * t

(* Direto *)
let rec height = function
```

```
  | E ->  0
  | N (a,b) -> 1 + max (height a) (height b)

(* ANF *)
let rec height matchArg =
  match matchArg with
  | E -> 0
  | N (a,b)  ->
      let heightANF_0 = height a  in
      let heightANF_1 = height b  in
      1 + (max heightANF_0 heightANF_1)
```

A transformação para ANF promove uma avaliação de ordem específica, resultando, neste contexto, na criação de expressões do tipo `let in`, sendo a expressão associada ao `let in` uma aplicação cujo o primeiro parâmetro é um identificador igual ao nome da função envolvente (no exemplo do excerto de código, `height a`). Esta estruturação de código será aproveitada para a implementação CPS, abordada na subsecção seguinte.

2.3 Implementação de CPS

A transformação para CPS aproveita a transformação intermédia para ANF, presente no excerto de código 1.2. Este aspecto pode ser observado através do reaproveitamento de expressão de `let in` associada a ANF (`heightaux a`), sendo esta seguida de `fun`, com nome de atributo de ANF (`height_aux a (fun heightANF_0 -> ...)`). Deve-se então proceder de forma análoga para sucessivos `let in`'s aninhados, até encontrar outra expressão que não seja `let in` (no exemplo usado, `1 + (max heightANF_0 heightANF_1)`), local onde será aplicada a continuação.

Excerto de código 1.3: Cálculo de altura de árvore em estilo CPS.

```
(* CPS *)
let rec height matchArg =
  let rec height_aux matchArg kFunction =
    match matchArg with
    | E  -> kFunction 0
    | N (a,b) ->
        height_aux a (fun heightANF_0  ->
        height_aux b (fun heightANF_1  ->
          kFunction (1 + (max heightANF_0 heightANF_1))))
in  height_aux matchArg (fun s -> s)
```

É evidente a reorganização de estrutura entre ANF e CPS e o reaproveitamento de código que existe entre estas, o que minimiza a dificuldade de implementação de transformação de código em estilo direto para CPS.

2.4 Tratamento de exceções

Com o objetivo de ter sempre o controlo do fluxo do programa, via continuações, optou-se por ter um tratamento particular com as exceções. Ao invés de deixar que seja o programa a lidar com excepções, como ocorre no estilo de programação direto, iremos tratar estas via continuações, mantendo a coerência do estilo CPS.

Caso o código a transformar contenha aplicações cujo o primeiro parâmetro desta seja um identificador associado a exceções (e.g. `raise, failwith`), então essa porção de código será passada a uma continuação de exceção e não a uma de retorno; um exemplo de continuação de retorno pode ser observado em 1.3, com denominação `kFunction`. Naturalmente, enquanto que haverá apenas uma continuação de retorno, resultante da transformação automática de código para CPS, haverá tantas continuações de exceções quantas as existentes no código inicial (em estilo direto). O excerto de código 1.4 exibe um exemplo com exceção e invocação de transformação automática de código. Note-se a presença do retorno associado a `Example`, no ramo `with`, que irá ser associado à continuação de exceção, aquando da transformação.

Excerto de código 1.4: Exemplo de aplicação de exceção, usando `try` *e* `with`, com invocação de transformação automática para CPS.

```
exception Example
let %CPS rec func x = if x < 0 then raise Example else x
let %CPS n = try func (-2) with
  Example -> Format.eprintf "Exception@."; exit 0
```

Excerto de código 1.5: Resultado de transformação automática para CPS de exemplo de código com exceção.

```
exception Example
let rec func x =
  let rec func_aux x kFunction kException0 =
    if x < 0 then kException0 () else kFunction x  in
  func_aux x (fun s  -> s)
  (fun ()  -> Format.eprintf "Exception@."; exit 0)
let n = func (-2)
```

No excerto de código 1.5 é visível a transformação de código de 1.4, ilustrando o tratamento de exceções, a interação com a `try` *e* `with` e os retornos de exceções. Com a inclusão dos retornos associados a exceções (presentes nos ramos de `with`), resultantes da avaliação de *e*, faz sentido incorporar estes retornos no retorno das continuações de exceções, aquando da transformação para CPS. Derivado desta consideração, podemos concluir que a nossa linguagem-alvo não contém as construções `try` *e* `with` ou `raise`.

Maior detalhe, relativamente ao tratamento de exceções, será apresentado na subsecção 3.2, onde será exibido um exemplo de transformação de código mais completo, em CPS, contendo diferentes exceções.

3 Avaliação experimental

Nesta secção serão exibidos exemplos de aplicação da transformação automática desenvolvida, apresentando o código direto inicial e o código resultante da transformação, em CPS. O método de transformação segue os mecanismo referidos na secção anterior sendo, ainda, demonstrados os efeitos ocorridos aquando da execução do código transformado, nomeadamente a nível da mitigação de erros derivados de código ineficiente (e.g. *Stack Overflow*).

Exemplos de transformação, usando árvores, não serão exibidos nas subseções seguintes, visto tal já ter sido apresentado nos excertos de código 1.2 e 1.3, na secção anterior. Embora neste não seja evidente a invocação da transformação automática, é visível a transformação final, em CPS. Assim. será dado enfâse a exemplos que usem outras estruturas de dados.

3.1 Transformação usando listas

Neste primeiro exemplo de transformação de código, em CPS, é vísivel a invocação de transformação automática, usando para tal `%CPS`, aquando da definição da função de somatório de elementos de uma lista, no excerto 1.6.

Excerto de código 1.6: Função somatória de elementos de lista, com invocação de transformação automática para CPS.

```
let %CPS rec sum = function [] -> 0 | el::rx -> el + sum rx
```

O resultado da transformação de código em 1.6 pode ser obervador no excerto de código 1.7. Este novo código apresenta-se em CPS e tem a particularidade de não modificar a chamada de função transformada. Por outras palavras, `sum` está agora em CPS mas para ser invocada apenas terá de receber uma lista (tal como seria se estivesse em estilo direto, no excerto de código 1.6), sendo a aplicação de parâmetros extra feita internamente a esta. Neste exemplo, incorporou-se uma função `sum_aux`, que contém todos os argumentos de `sum`, além da continuação, sendo incorporado nesta função o novo código, em CPS.

Excerto de código 1.7: Função somatória de elementos de lista, em CPS, resultante da transformação automática de código em 1.6.

```
let rec sum matchArg =
  let rec sum_aux matchArg kFunction =
    match matchArg with
    | [] -> kFunction 0
    | el::rx -> sum_aux rx (fun sumANF_0  ->
        kFunction (el + sumANF_0))
in sum_aux list (fun s  -> s)
```

A pertinência desta transformação pode ser observada quando a função `sum` recebe como argumento uma lista com 1 milhão de elementos. No excerto de

código 1.8, podemos observar a invocação de função `sum` e a apresentação do resultado do somatório dos elementos da lista. Como é visível, neste excerto, `sum` é a implementação em estilo direto do cálculo do somatório, o que culmina na apresentação de erro *StackOverflow*.

Excerto de código 1.8: Somatório de elementos de lista, com 1 milhão de elementos, em código direto, e exibição de resultado.

```
let rec sum = function [] -> 0 | el::rx -> el + sum rx

let () = Format.eprintf "sum: %d@." (sum l_1_milhao)
```

Output: `Fatal error: exception Stack_overflow`

Por comparação, o *output* de 1.6 pode ser visto no excerto de código 1.9. Note-se que código presente no excerto 1.6 é exatamente o mesmo que em 1.8, com a exceção de o primeiro ter a invocação de transformação automática para CPS, por inclusão de `%CPS`, na expressão a modificar.

Excerto de código 1.9: *Output* de somatório de elementos de lista, com 1 milhão de elementos, com transformação automática para CPS aplicada.

Output: `sum: 500000500000`

Este exemplo é uma boa referência para exibir a facilidade de aplicação da âncora sintática (%CPS) que culmina na transformação automática de código em CPS. Adicionalmente, podemos observar que apenas nas funções que o utilizador deseje serão aplicadas as transformações para CPS.

3.2 Transformação usando exceções

O tratamento de exceções, aquando da transformação automática, foi anteriormente mencionado em 2.4. Esta subsecção visa reforçar as noções referidas por recurso a um exemplo mais completo de transformação de código para CPS. Este exemplo será também usado para apresentar outras características associadas à transformação de código com exceções, nomeadamente, a estrutura da continuação de exceção e os possíveis retornos associados a esta.

Excerto de código 1.10: Cálculo de fatorial, usando exceções, com invocação de transformação automática de código.

```
exception Negative

let %CPS rec fact x =
  if x < 0 then raise Negative
  else if x > 20 then failwith "Valor de x demasiado elevado!"
  else if x = 0 then 1 else x * fact (x -1)
```

```
let %CPS result = try fact 5 with
  Negative -> Format.eprintf "Numero negativo!@."; exit 0
```

O excerto de codigo 1.10 exibe um exemplo de utilização de exceções, com uso de `try` *e* `with`, onde em `with` está contido o retorno associado a uma exceção declarada pelo utilizador; neste caso fará um *print* da mensagem "Numero negativo", terminando o programa. Este exemplo tem também a particularidade de, no interior da função, ser invocada uma exceção (usando `failwith`), cujo retorno não está presente nos ramos de `with`. Este cenário permite perceber de que forma a transformação automática de código para CPS lida com diferentes casos de exceções.

Excerto de código 1.11: Cálculo de fatorial, usando exceções, com código automaticamente transformado para CPS.

```
exception Negative
let rec fact x =
  let rec fact_aux x kFunction kException0 kException1 =
    if x < 0 then kException0 ()
    else if x > 20 then kException1 ()
      else if x = 0 then kFunction 1
        else fact_aux (x - 1) (fun factANF_0  ->
          kFunction (x * factANF_0))
          kException0 kException1
  in fact_aux x (fun s  -> s)
    (fun ()  -> Format.eprintf "Numero negativo!@."; exit 0)
    (fun ()  -> failwith "Valor de x demasiado elevado!")
```

Em 1.11 podemos observar o resultado da transformação de código para CPS. Em concordância com o referido neste subsecção e na subsecção 2.4, haverá tantas continuações de exceções, dentro da função transformada, como exceções dentro destas. A criação de exceção é feita com aplicação de uma continuação de exceção ao elemento `unit`. Os retornos associados a cada uma das exceções são observáveis depois de `(fun s -> s)`, continuação de retorno, e estes são concordantes com o retorno presente nos ramos `with`, de `try` *e* `with`, e com o retorno da exceção não declarada por utilizador (`failwith`).

Este exemplo exibe também o desaparecimento de `try` *e* `with`, sendo esta substituída por *e* (no exemplo, `fact x`). Tal acontece pois foi associada a esta expressão a âncora sintática `%CPS` e porque todos os ramos de `with` continham retornos associados a exceções de utilizador (no exemplo, `Negative`). Assim não se justifica a presença de `try` *e* `with`, pois o retorno a apresentar, aquando da presença de uma exceção, já se encontra presente nas continuações de exceções.

4 Desempenho da transformação

A transformação automática de código para CPS contempla uma enorme vantagem em termos de manuseamento de elevado número de dados mas tal transformação não é conseguida sem desvantagens. A noção de remover o tratamento

recursivo de funções da pilha implícita de OCaml e passar este para a memória dinâmica, onde todas as iterações são tratadas em continuações, não é livre de sacrifícios, de desempenho ou de determinismo comportamental. De um forma intuitiva, esta transformação remove o tratamento recursivo de funções de uma componente altamente eficiente nesse aspecto (pilha), transferindo esta tarefa para uma outra (memória dinâmica), com maior capacidade de armazenamento. Assim, embora seja possível tratar de um maior número de dados, perdemos eficiência. Adicionalmente, se os dados da função, tradicionalmente colocados na pilha, são transferidos para a *heap*, estes podem ser sujeitos à acção do *Garbagge Collector*, cujo funcionamento não é local nem facilmente previsível.

Nesta secção avaliamos o desempenho entre código em estilo direto e código transformado para CPS. A figura 2 apresenta o gráfico que relacionada o tempo de execução com a quantidade de elementos de entrada. Foram testadas duas estruturas diferentes, árvores binárias (AB) e listas (List), sendo avaliado, respetivamente, a altura de uma árvore e o somatório dos elementos de uma lista. Ambas as avaliações foram feitas em estilo direto e CPS (resultante da transformação automática de código). Mais se acrescenta que as respectivas bibliotecas *standard* OCaml (os módulos `List` e `Set`) foram totalmente instrumentadas com sucesso por esta transformação.

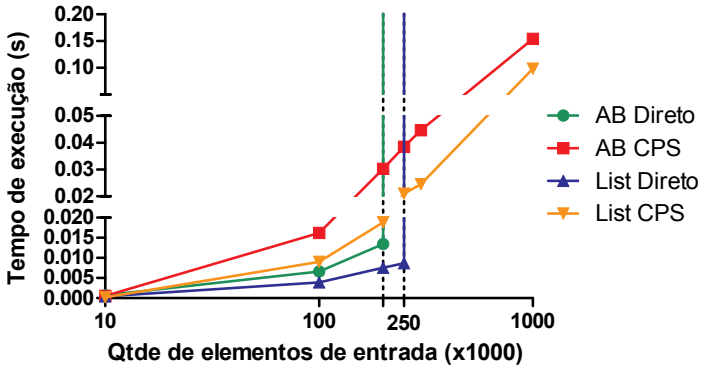

Figura 2: Avaliação de desempenho entre estilo direto e CPS para cálculo de altura de árvore (AB) e somatório de elementos de lista (List). Foi relacionado o tempo de execução (s) com a quantidade de elementos de entrada (x1000).

A análise do gráfico da figura 2 permite perceber que, abaixo de 10.000 elementos de entrada a diferença entre estilos é negligenciável. Acima deste valor, e até 200.000 elementos, o estilo direto tem uma melhor *performance* que o estilo CPS. No entanto, acima de 200.000 elementos, para o cálculo de altura de árvore, e acima de 250.000 elementos, para o cálculo de somatório de elementos de lista, o estilo direto exibe o erro *StackOverflow*, expresso no gráfico pela

subida da reta para o infinito. O estilo CPS, mesmo nestas condições, consegue efetuar os cálculos requiridos, sendo capaz de suportar quantidades de elementos superiores, como é o caso de 1.000.000 de elementos, com tempos de execução abaixo de 0.2 segundos. Mais detalhes da avaliação de desempenho podem ser vistos no anexo B.

5 Discussão e perspectivas

Neste artigo apresentou-se uma técnica de transformação automática de código, para CPS, fazendo uso de uma extensão sintática de OCaml, denominada PPX, dedicada à introdução de novas funcionalidades à linguagem. Este mecanismo permite que o utilizador, de forma cómoda e transparente, consiga modificar o seu código, tornando-o recursivo terminal e mitigando erros do tipo *StackOverflow*, bastando para tal a incorporação da âncora sintática `%CPS` na expressão a transformar.

As inerentes vantagens de CPS são contrabalançadas pela sua dificuldade de implementação, obrigando o utilizador a usar um estilo de programação particular, baseado em continuações. A facilidade de transformação de código, à escolha do utilizador, por inserção de `%CPS` no local a modificar, permite que o utilizador se concentre em desenhar uma implementação ao seu estilo, ainda que ineficiente, delegando ao PPX a tarefa de o tornar apropriado para lidar com elevado número de dados.

A noção de alterar apenas as expressões contendo `%CPS` tem particular importância pela diferença de desempenho que a implementação de CPS acarreta. Na secção 4 foi evidente que, para um diminuto número de elementos, a diferença entre estilo direto e CPS é inexistente. No entanto, existem intervalos de valores, relativamente ao número de elementos, onde a *performance* do estilo direto é superior ao CPS. As vantagens de transformação para CPS tornam-se mais evidentes na resolução de problemas com elevado número de elementos, situação onde o estilo direto culmina em erro *StackOverflow*. Assim, a avaliação do problema a lidar é importante para determinar a pertinência da transformação automática de código para CPS.

No contexto pedagógico, ambiente de desenvolvimento deste artigo, importa referir que a utilização de CPS potencia a implementação de instrumentação automática e incorporação de sistemas de correção automática de programas, respeitando testes de tipo "caixa branca". A transformação automática para CPS permite também o *feedback* visual automático do processo de execução de um programa, admitindo a complementaridade de *frameworks* adequadas, como é o caso de *Ocsigen*, outra componente do projeto FACTOR, onde este artigo se insere. Neste ambiente de ensino, a capacidade de manipulação de execução de programas de alunos tem valor acrescido e tal é conseguido por recurso ao CPS. No entanto, a implementação direta de algoritmos em CPS não é algo que se queira impor aos alunos. O intuito pedagógico é que estes percebam as ideias centrais de algoritmos sem se concentrarem nos detalhes de implementação. A transformação automática para CPS desenvolvida, usando PPX, surge assim nesse sentido.

Como perspectivas futuras, tenciona-se promover a transformação automática de código usando a técnica de desfuncionalização [13], que permite a resolução de problemas, com base em continuações, mas usando um programa de primeira ordem, ao invés de ordem superior, como é o caso do CPS desenvolvido neste artigo. Esta implementação tem interesse particular quando juntamos ao cenário pedagógico descrito a possibilidade de os alunos serem expostos à demonstração, por computadores, dos critérios de correção dos algoritmos implementados. Se pretendemos provas automáticas, temos de dispôr de mecanismos que transformem programas de ordem superior (de prova automática difícil) em código de primeira ordem (cuja prova automática é mais plausível). A desfuncionalização é uma de tais transformações.

Referências

1. A Guide to PreProcessor eXtensions, https://ocamlverse.github.io/content/ppx.html. Último acesso: 26 de maio de 2019.
2. Real World OCaml, 2nd Edition, https://dev.realworldocaml.org/compiler-frontend.html. Último acesso: 26 de maio de 2019.
3. A Tutorial to OCaml -ppx Language Extensions, https://victor.darvariu.me/jekyll/update/2018/06/19/ppx-tutorial.html. Último acesso: 27 de maio de 2019.
4. Functional programming and type systems, Making the stack explicit: the CPS transformation, https://gitlab.inria.fr/fpottier/mpri-2.4-public/blob/master/slides/fpottier-04.pdf. Último acesso: 22 de maio de 2019.
5. Danvy, O., Nielsen, L.R.: CPS transformation of beta-redexes. Information Processing Letters 94(5), 217-224 (2005).
6. John C. Reynolds. The discoveries of continuations. Lisp and Symbolic Computation, 6(3/4):233–247, 1993.
7. Christopher P. Wadsworth. Continuations revisited. Higher-Order and Symbolic Computation, 13(1/2):131–133, 2000.
8. Michael J. Fischer. Lambda-calculus schemata. LISP and Symbolic Computation, 6(3/4):259–288, 1993.
9. Gordon D. Plotkin. Call-by-name, call-by-value and the *lambda*-calculus. Theoretical Computer Science, 1:125–159, 1975.
10. Guy L. Steele Jr. Rabbit: A compiler for Scheme. Master's thesis, M.I.T (A.I. LAB.), Massachusetts, U.S.A, 1978.
11. Sabry, A. and Felleisen, M. Reasoning about program in continuation passing style. In Proc. ACM Conference on Lisp and Functional Programming, pp. 288-298; extended version in Lisp and Symbolic Comput. 6(3/4), 289-360, 1993.
12. Flanagan, C., Sabry, A., Duba, B. F., and Felleisen, M. The essence of compiling with continuations. In Proceedings of the ACM Sigplan Conference on Programming Language Design and Implementation (1993).
13. Danvy, O., Nielsen, L.R.: Defunctionalization at work. In: Proc. of 3rd Int. ACM SIGPLAN Conf. on Principles and Practice of Declarative Programming (2001).
14. Danvy, O. and Hatcliff, J.: CPS-transformation after strictness analysis. In: ACM Lett. Program. Lang. Syst. 3(1), 195-212, 1992.
15. Danvy, O. and Millikin, K. and Nielsen, L. R.: On One-pass CPS Transformations. In: J. Funct. Program.. 6(17), 793-812, 2007.
16. Pottier, F.: Revisiting the CPS Transformation and its Implementation, 2017, http://gallium.inria.fr/ fpottier/publis/fpottier-cps.pdf.

Apêndice A Semântica usada na transformação para CPS

Consideramos uma linguagem estilo ML, constituída por variáveis, construções `let in`, aplicação de funções (com múltiplos argumentos), estrutura `if then else` e, finalmente, definições locais de funções. Esta liguagem é representada pela seguinte gramática:

$$e ::= x \mid c \mid \texttt{let}\ x = e\ \texttt{in}\ e \mid f\ e\ \ldots\ e \mid \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e \mid \texttt{let}\ f\ x\ \ldots\ x = e\ \texttt{in}\ e$$

$$c ::= () \mid n$$

Na tradução de uma expressão e para uma expressão em estilo CPS equivalente, a primeira linguagem intermédia que apresentamos trata-se de uma variante da linguagem anterior, em que todos os argumentos de aplicações são expressões atómicas. Desta forma, qualquer programa aceite pela gramática seguinte encontra-se, por construção, em Forma Normal Administrativa:

$$t ::= x \mid c \mid \texttt{let}\ x = t\ \texttt{in}\ t \mid f\ a\ \ldots\ a \mid \texttt{if}\ a\ \texttt{then}\ t\ \texttt{else}\ t \mid \texttt{let}\ f\ x\ \ldots\ x = t\ \texttt{in}\ t$$

$$a ::= x \mid c$$

$$c ::= () \mid n$$

Tradução para Forma Normal Administrativa

Denotamos por $\mathcal{A}[\![\cdot]\!]$ a seguinte função, que converte uma expressão arbitrária e numa expressão equivalente t:

$$\boxed{\mathcal{A}[\![\cdot]\!] : e \to t}$$

$$\begin{aligned}
\mathcal{A}[\![x]\!] &= x \\
\mathcal{A}[\![c]\!] &= c \\
\mathcal{A}[\![\texttt{let}\ x = e_1\ \texttt{in}\ e_2]\!] &= \texttt{let}\ x = \mathcal{A}[\![e_1]\!]\ \texttt{in}\ \mathcal{A}[\![e_2]\!] \\
\mathcal{A}[\![f\ e_1\ \ldots\ e_n]\!] &= \texttt{let}\ x_1 = \mathcal{A}[\![e_1]\!]\ \texttt{in} \\
&\quad\ \vdots \\
&\quad\ \texttt{let}\ x_n = \mathcal{A}[\![e_n]\!]\ \texttt{in} \\
&\quad\ f\ x_1\ \ldots\ x_n \\
\mathcal{A}[\![\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3]\!] &= \texttt{let}\ x = \mathcal{A}[\![e_1]\!]\ \texttt{in} \\
&\quad\ \texttt{if}\ x\ \texttt{then}\ \mathcal{A}[\![e_2]\!]\ \texttt{else}\ \mathcal{A}[\![e_3]\!] \\
\mathcal{A}[\![\texttt{let}\ f\ x_1\ \ldots\ x_n = e_1\ \texttt{in}\ e_2]\!] &= \texttt{let}\ f\ x_1\ \ldots\ x_n = \mathcal{A}[\![e_1]\!]\ \texttt{in}\ \mathcal{A}[\![e_2]\!]
\end{aligned}$$

Esta função procede por casos sobre a estrutura da expressão e e segue uma definição homomórfica, excepto para os casos da aplicação e da estrutra

`if then else`. Para as ambas as construções procedemos de forma semelhante: convertemos as sub-expressões necessárias para Forma Normal Administrativa, substituindo-as na expressão final pelas respectivas variáveis resultantes da transformação.

Tradução para CPS

Denotamos por t^{CPS} a linguagem-alvo final da nossa tradução. Trata-se de uma extenção à definição de t com a introdução de funções anónimas, como se segue:

$$t^{\text{CPS}} ::= \ldots \mid \texttt{fun}\, x \to t^{\text{CPS}}$$

Denotamos por $\mathcal{CPS}[\![\cdot]\!]_k$ a seguinte função, que converte uma expressão t numa expressão equivalente t^{CPS}:

$$\boxed{\mathcal{CPS}[\![\cdot]\!]_k : t \to t^{\text{CPS}}}$$

$$\begin{aligned}
\mathcal{CPS}[\![x]\!]_k &= k\, x \\
\mathcal{CPS}[\![c]\!]_k &= k\, c \\
\mathcal{CPS}[\![\texttt{let}\, x = t_1\, \texttt{in}\, t_2]\!]_k &= \mathcal{CPS}[\![t_2]\!]_{(\texttt{fun}\, x \to \mathcal{CPS}[\![t_1]\!]_k)} \\
\mathcal{CPS}[\![f\, a_1\, \ldots\, a_n]\!]_k &= f\, a_1\, \ldots\, a_n\, k \\
\mathcal{CPS}[\![\texttt{if}\, a\, \texttt{then}\, t_1\, \texttt{else}\, t_2]\!]_k &= \texttt{if}\, a\, \texttt{then}\, \mathcal{CPS}[\![t_1]\!]_k\, \texttt{else}\, \mathcal{CPS}[\![t_2]\!]_k \\
\mathcal{CPS}[\![\texttt{let}\, f\, x_1\, \ldots\, x_n = t_1\, \texttt{in}\, t_2]\!]_k &= \texttt{let}\, f\, x_1\, \ldots\, x_n\, k = \mathcal{CPS}[\![t_1]\!]_k\, \texttt{in}\, \mathcal{CPS}[\![t_2]\!]_k
\end{aligned}$$

A tradução para estilo CPS é parameterizada por uma continuação k. O caso mais interessante desta transformação é o da estrutura `let in`, em que uma nova continuação é construída para a transformação recursiva da expressão t_2.

Tradução de Excepções

A fim de permitir a transformação de excepções, extendemos a definição de t com os construtores `raise` e `try with`, como se segue:

$$t^{\mathcal{X}} ::= \ldots \mid \texttt{raise}\, E\, a\, \ldots\, a \mid \texttt{try}\, t\, \texttt{with}\, \overline{E\bar{x} \Rightarrow t}$$

Denotamos por $\mathcal{X}[\![\cdot]\!]_{\mathcal{F}}$ a seguinte função, que converte uma expressão $t^{\mathcal{X}}$ numa expressão equivalente t^{CPS}:

$$\boxed{\mathcal{CPS}[\![\cdot]\!]_{\mathcal{F}} : t^{\mathcal{X}} \to t^{\text{CPS}}}$$

$$\begin{aligned}
\mathcal{X}[\![\texttt{raise}\, E\, a_1\, \ldots\, a_n]\!]_{\mathcal{F}} &= f_E(a_1, \ldots, a_n), \quad f_E \in \mathcal{F} \\
\mathcal{X}[\![\texttt{try}\, t_0\, \texttt{with}\, \overline{E\bar{x} \Rightarrow t}]\!]_{\mathcal{F}} &= \mathcal{X}[\![t_0]\!]_{\mathcal{F} \biguplus \{\texttt{fun}\, \overline{x_i} \to \mathcal{X}[\![t_i]\!]_{\mathcal{F}}\}}
\end{aligned}$$

Omitimos nesta definição os casos de transformação para expressões t. Esta transformação é parameterizada por uma família de funções $\mathcal{F}$ que contém todas as continuações excepcionais necessárias. De notar que não é necessário estender a linguagem-alvo desta transformação com suporte para excepções.

Apêndice B Avaliação de desempenho

Os testes foram desenvolvidos em *GraphPad Prim*, versão 5.01, no formato de tabela XY, com replicação de 3 valores para Y (tempos de execução), sendo feito o gráfico com base no *Standard Error of the Mean* (SEM). O tipo de gráfico usado foi *Points & connecting line*. Os testes foram desenvolvidos em Ubuntu 18.04.2 LTS, usando um *Samsung* de processador Intel® Core™ i7- 3630QM CPU 2.40GHZ x 8, com 8GB RAM. A versão OCaml utilizada foi a 4.05, o IDE usado foi *Visual Studio Code* e o registo do tempo de execução foi conseguido por recurso ao módulo OCaml *Sys*, usando a função `Sys.time()`.

A avaliação de desempenho foi feita com base em árvores binárias e listas. A avaliação de listas segue o modelo referido na subsecção 3.1, onde se realizou o somatório de uma determinada quantidade (`n`) de elementos de uma lista, cujos elementos variam entre 1 e `n`. A avaliação do desempenho entre os dois estilos, relativamente à estrutura de dados árvores binárias, foi feita sobre o cálculo da altura destas. O excerto de código 1.12 ilustra o teste realizado para esta estrutura e serve como exemplo da obtenção de tempos de execução.

Excerto de código 1.12: Avaliação do tempo de excecução, associado ao cálculo de altura de uma árvore binária, em estilo CPS.

```
type t = E | N of t * t

(* Criar arvore *)
let rec leftist_tree t = function
  | 0 -> t
  | n -> leftist_tree (N (t, E)) (n - 1)

(* Variaveis a usar *)
let t = leftist_tree E 100_000

(* Função timer *)
let time f x name =
    let t = Sys.time() in
    let _ = f x in
    Format.eprintf "Execution time of %s: @. %fs @." name (Sys.time() -. t)

let %CPS rec height = function
  | E ->  0
  | N(a,b) -> 1 + max (height a) (height b)

let () = time height t "height (t)"
```