UNIVERSIDADE DO ESTADO DE SANTA CATARINA – UDESC
CENTRO DE CIÊNCIAS TÉCNOLÓGICAS – CCT
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO – BCC

VINÍCIOS BIDIN SANTOS

INFERÊNCIA DE TIPOS PARA CPS

JOINVILLE
2024

VINÍCIOS BIDIN SANTOS

INFERÊNCIA DE TIPOS PARA CPS

Trabalho de conclusão de curso submetido à Universidade do Estado de Santa Catarina como parte dos requisitos para a obtenção do grau de Bacharel em Ciência da Computação

Orientador: Cristiano Damiani Vasconcellos
Coorientador: Paulo Henrique Torrens

JOINVILLE
2024

Para gerar a ficha catalográfica de teses e
dissertações acessar o link:
https://www.udesc.br/bu/manuais/ficha

Santos, Vinícios Bidin
Inferência de tipos para CPS / Vinícios Bidin Santos.
- Joinville, 2024.
24 p. : il. ; 30 cm.

Orientador: Cristiano Damiani Vasconcellos.
Coorientador: Paulo Henrique Torrens.
Trabalho de Conclusão de Curso - Universidade
do Estado de Santa Catarina, Centro de Ciências
Tecnológicas, Bacharelado em Ciência da Computação,
Joinville, 2024.

1. Inferência de Tipos. 2. Estilo de Passagem de
Continuação. 3. Algoritmo W. 4. Damas-Milner. 5. Haskell.
I. Vasconcellos, Cristiano Damiani . II. Torrens, Paulo
Henrique . III. Universidade do Estado de Santa Catarina,
Centro de Ciências Tecnológicas, Bacharelado em Ciência
da Computação. IV. Título.

VINÍCIOS BIDIN SANTOS

INFERÊNCIA DE TIPOS PARA CPS

Trabalho de conclusão de curso submetido à Universidade do Estado de Santa Catarina como parte dos requisitos para a obtenção do grau de Bacharel em Ciência da Computação

Orientador: Cristiano Damiani Vasconcellos
Coorientador: Paulo Henrique Torrens

BANCA EXAMINADORA:

Orientador:

Dr. Cristiano Damiani Vasconcellos
UDESC

Coorientador:

Me. Paulo Henrique Torrens
University of Kent

Membros:

Dra. Karina Girardi Roggia
UDESC

Me. Gabriela Moreira

Joinville, Novembro de 2024

Um salve pra geral que me apoiou nesses anos!

"*Eu sou a senhora Marocas.*" (Senhora Marocas – Toy Story, [1995])

RESUMO

Este trabalho propõe uma investigação sobre a inferência de tipos para o Estilo de Passagem de Continuação (CPS) - representação intermediária amplamente utilizada em compiladores de linguagens funcionais. A pesquisa se concentra na extensão do algoritmo W, tradicionalmente usado para inferência de tipos no sistema Damas-Milner, para abranger o cálculo de continuações. A proposta inclui a implementação dessa extensão na linguagem Haskell e a validação do algoritmo por meio de programas de teste, assegurando que os tipos inferidos estejam corretos.

Palavras-chave: Inferência de Tipos, Estilo de Passagem de Continuação (CPS), Algoritmo W, Damas-Milner, Haskell, Sistema de Tipos.

ABSTRACT

This work proposes an investigation into type inference for Continuation Passing Style (CPS) - an intermediate representation widely used in compilers for functional languages. The research focuses on extending the W algorithm, traditionally used for type inference in the Damas-Milner system, to encompass continuation calculus. The proposal includes implementing this extension in the Haskell programming language and validating the algorithm through test programs, ensuring that the inferred types are correct.

Keywords: Type Inference, Continuation Passing Style (CPS), W Algorithm, Damas-Milner, Haskell, Type System.

LISTA DE ILUSTRAÇÕES

Figura 1 – Sequência de representações intermediárias 15
Figura 2 – Função soma em Haskell em Estilo Direto 17
Figura 3 – Função soma em Haskell em CPS 18

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

IR	Intermediate Representation
CPS	Continuation Passing Style
SSA	Static Single Attribution
ANF	Administrative Normal Form

SUMÁRIO

1	**INTRODUÇÃO**	**12**
1.1	OBJETIVO GERAL	**13**
1.2	OBJETIVOS ESPECÍFICOS	**13**
1.3	METODOLOGIA	**13**
1.4	ESTRUTURA DO TRABALHO	**13**
2	**REPRESENTAÇÃO INTERMEDIÁRIA DE CÓDIGO**	**15**
2.1	CPS	**16**
3	**TEORIA DE TIPOS**	**19**
4	**SISTEMA DAMAS-MILNER**	**20**
4.1	ALGORITMO W	**20**
5	**FORMALIZAÇÃO**	**21**
6	**IMPLEMENTAÇÃO**	**22**
7	**CONCLUSÃO**	**23**
	REFERÊNCIAS	**24**

1 INTRODUÇÃO

A compilação de programas envolve diversas fases, cada uma com funções específicas, como análise léxica, análise sintática, análise semântica, otimizações, e, finalmente, a geração de código. Uma etapa crítica nesse processo é a otimização, que frequentemente se baseia em representações intermediárias (IRs). Essas representações atuam como ponte entre o código fonte e o código de máquina, permitindo que transformações e otimizações sejam aplicadas de maneira mais eficaz (PLOTKIN, 1975).

As representações intermediárias variam conforme o paradigma da linguagem de programação. Para linguagens imperativas, a Representação em Atribuição Única Estática (SSA) é amplamente adotada. Já em linguagens funcionais, a Forma Normal Administrativa (ANF) e o Estilo de Passagem de Continuação (CPS) se destacam. Este trabalho foca especificamente no CPS, uma IR que oferece vantagens particulares em termos de otimização e simplicidade na geração de código.

Essas características do CPS se tornam ainda mais evidentes quando comparamos como diferentes linguagens de lidam com o fluxo de execução. Em linguagens de alto nível, por exemplo, a pilha de chamadas atua como uma abstração fundamental para gerenciar o controle de retorno das funções. No entanto, em linguagens de baixo nível, como *assembly*, não há tal abstração, exigindo que o controle do fluxo seja manualmente tratado por meio de endereços de retorno.

Nesse contexto, o CPS se destaca ao tornar as continuações explícitas no código. Ao invés de confiar na pilha de chamadas para gerenciar retornos, o CPS introduz um parâmetro adicional em cada função, representando a continuação — isto é, o que deve ser feito com o resultado da função (KENNEDY, 2007). Desta forma, em vez de simplesmente retornar um valor diretamente, a função invoca essa continuação, transferindo explicitamente o controle à próxima etapa da computação. Isso elimina a dependência da pilha de chamadas, simplificando o modelo de execução e tornando-o mais alinhado com as necessidades de linguagens de baixo nível.

Além disso, a adoção do CPS como representação intermediária vai além da tradução de linguagens de alto nível para código de máquina. O CPS facilita a aplicação de otimizações avançadas, como a eliminação de chamadas de cauda e a fusão de funções, além de permitir uma correspondência mais direta com o código gerado em linguagens de montagem (FLANAGAN et al., 1993).

Por outro lado, um ponto importante a ser considerado é que muitas implementações do CPS optam por uma representação não tipada (MORRISETT et al., 1999). Embora essa abordagem simplifique a implementação inicial, ele pode comprometer a segurança e a correção do código. Um sistema de tipos robusto pode não apenas garantir a correção de certas transformações e otimizações, mas também identificar uma classe inteira de erros antes da execução, proporcionando assim maior confiabilidade ao processo de compilação.

Diante dessas considerações, a linguagem de programação Haskell foi escolhida para

a implementação deste trabalho. Haskell, sendo uma linguagem funcional pura fortemente tipada, possui características desejáveis, como transparência referencial (SØNDERGAARD; SESTOFT, 1990) e um sistema de tipos robusto para explorar as vantagens do CPS e aplicar o sistema de tipos de maneira rigorosa. Dessa forma, a escolha de Haskell não apenas facilita o desenvolvimento de uma implementação segura e eficiente do CPS, como também conta com garantias de seguranças que são fundamentais para o sucesso deste trabalho.

1.1 OBJETIVO GERAL

Este trabalho visa explorar a inferência de tipos para CPS, propondo uma extensão do algoritmo W - um algoritmo de inferência de tipos para o sistema Damas-Milner - para CPS, definido em (TORRENS; ORCHARD; VASCONCELLOS, 2024) na linguagem de programação Haskell.

1.2 OBJETIVOS ESPECÍFICOS

- Formular uma extensão do algoritmo W para CPS;
- Implementar essa extensão em Haskell;
- Validar a implementação do algoritmo por meio do teste de inferência para expressões. Se possível, realizar a geração de programas para verificação de que o algoritmo infere corretamente os tipos a eles.

1.3 METODOLOGIA

A metodologia deste trabalho consistirá em duas principais etapas: pesquisa bibliográfica e implementação. A primeira etapa envolve uma extensa revisão de literatura sobre continuações e seu cálculo, bem como um aprofundamento do sistema Damas-Milner, com o objetivo de proporcionar uma compreensão completa ao autor. A segunda etapa trata da formulação do algoritmo W estendido para o cálculo de continuações, junto com sua implementação.

Para validar a implementação, será testado a inferência do algoritmo com testes. Caso seja possível, será feito ainda a geração de programas de teste, nos quais a inferência de tipos será verificada. A validação consistirá em comparar os tipos inferidos pelo algoritmo com os tipos esperados para esses programas.

1.4 ESTRUTURA DO TRABALHO

Este trabalho está estruturado de maneira que será fornecida, antes da formalização da extensão e implementação do algoritmo W, nos capítulos 5 e 6, uma fundamentação teórica. Inicialmente, será abordada a representação intermediária no capítulo 2, com um aprofundamento

em CPS na seção 2.1. Na sequência, o capítulo 3 discutirá a teoria de tipos, apresentando os conceitos e definições necessários para a compreensão do tema. O sistema Damas-Hindley-Milner será detalhado no capítulo 4, incluindo o algoritmo W, discutido de forma mais específica na seção 4.1. Por fim, o capítulo 7 trará as considerações finais, sintetizando os principais pontos e contribuições do trabalho.

2 REPRESENTAÇÃO INTERMEDIÁRIA DE CÓDIGO

Na computação, um compilador é um programa responsável por traduzir um código escrito em uma linguagem de programação para outra, geralmente do código-fonte para código de máquina, permitindo assim a execução do programa. Durante esse processo, é fundamental que o mínimo de informações seja perdido, uma vez que a semântica original deve ser preservada no processo de tradução. Uma abordagem comum utilizada para manter a integridade semântica e possibilitar otimizações, são as representações intermediárias (IR, do inglês *intermediate representation*) (COOPER; TORCZON, 2014).

Compiladores modernos, amplamente utilizados na indústria, empregam mais de uma IR para tirar proveito das vantagens de cada uma, uma vez que essas representações são projetadas para diferentes objetivos, como otimizações específicas. As IRs podem ser classificadas de acordo com o nível de abstração e são comumente aplicados em sequência. Representações com um nível maior de abstração são usadas próximas ao código-fonte, enquanto aquelas de nível mais baixo estão mais próximas do código de máquina (AHO et al., 2008), como ilustrado na figura 1.

Uma das principais informações que deve ser preservada em uma IR, é o fluxo de controle, isto é, a ordem em que as instruções do programa são executadas, como chamadas de função, *loops* e condições. Para garantir que o compilador mantenha a semântica correta do programa, o fluxo de controle deve ser repassado de alguma maneira durante o processo de tradução (COOPER; TORCZON, 2014). Uma das maneiras disso ser feito explicitamente, é com o uso de continuações, que são funções que descrevem o próximo passo de uma computação em um ponto particular da execução do programa.

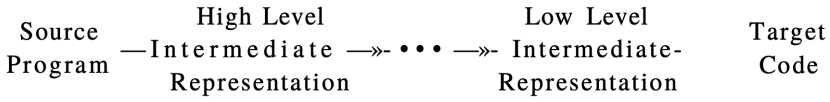

Figura 1 – Sequência de representações intermediárias
Fonte: (AHO et al., 2008)

2.1 CPS

O estilo de passagem de continuações (CPS, do inglês *continuation passing style*) é uma técnica de transformação de código que torna o fluxo de controle de um programa explícito, ao converter o estilo convencional de chamadas de função em chamadas que passam explicitamente o controle para a próxima etapa, conhecida como continuação (do inglês, *continuation*) (APPEL, 1992). Em vez de retornar diretamente o resultado de uma função, o CPS transforma cada função para que, ao finalizar sua computação, ela invoque uma *continuation*, que representa o próximo passo a ser executado no programa.

O cálculo lambda, definido por Church (1932), é um sistema formal que serve como base para a maioria das linguagens funcionais. Ele é capaz de representar qualquer computação utilizando abstrações e aplicações através de reduções. Sua sintaxe consiste em três regras simples que definem os elementos principais do sistema: variável, abstração e aplicação, conforme apresentados a seguir:

$$e ::= x \mid \lambda x.e \mid ee \tag{1}$$

A partir dessa sintaxe, um termo e pode possuir apenas uma das três formas. A primeira forma refere-se às variáveis, que representam identificadores no sistema. A segunda forma, chamada de abstração, define uma função lambda: uma função que associa o identificador x a um termo e, seu corpo, com x vinculado ao termo e. Finalmente, a aplicação ocorre quando um termo e é aplicado a outro e, representando a chamada de uma função.

Para avaliar expressões no cálculo lambda, usamos três tipos de redução: α, β e η, que seguem as seguintes definições:

α-redução: Renomeação de variáveis ligadas.

$$\lambda x.e[x] \to \lambda y.e[y] \tag{2}$$

β-redução: Aplicação de função.

$$(\lambda x.e_1)e_2 \to e_1[e_2/x] \tag{3}$$

η-redução: Expansão de função.

$$\lambda x.(e\,x) \to e \quad \text{se } x \text{ não ocorre livre em } e \tag{4}$$

As reduções são responsáveis pela semântica operacional do cálculo lambda. A α-redução permite a renomeação de variáveis ligadas, enquanto a β-redução descreve a aplicação de funções, substituindo o parâmetro da função por um valor passado como argumento. Por fim, a η-redução lida com a simplificação de funções quando elas aplicam diretamente seu argumento.

A transformação para CPS se baseia nessa estrutura formal. No cálculo lambda tradicional, o fluxo de execução é implícito: as funções são aplicadas e seus resultados são retornados

Figura 2 – Função soma em Haskell em Estilo Direto

Fonte: o autor

```haskell
add :: Int -> Int -> Int
add x y = x + y
```

automaticamente. No entanto, no CPS, o fluxo de controle é explicitamente representado como uma série de chamadas a funções. Cada função, em vez de retornar diretamente um valor, recebe um argumento extra, a continuação, que indica o próximo passo da computação.

Por exemplo, a expressão $\lambda x.x+1$ no cálculo lambda tradicional retornaria o valor $x+1$. Ao transformar essa expressão para CPS, ela se torna $\lambda x.\lambda k.k(x+1)$.

Aqui, k é a continuação que processa o resultado $x+1$. Essa técnica é especialmente poderosa no contexto de compiladores, uma vez que facilita várias formas de otimizações e análises, como eliminação de chamadas de função (TCO, do inglês *tail-call optimization*), expansão *inline*, representação de *closures*, alocação de registradores, entre outras (APPEL, 1992). Ao transformar o código para CPS, todo o fluxo de execução do programa é capturado como uma série de chamadas encadeadas de funções, sem depender de uma pilha de execução implícita.

O cálculo de continuações (do inglês, *CPS-calculus*), conforme definido por Thielecke (1997), é um sistema formal que leva o CPS além de seu uso tradicional como uma técnica de transformação de código, tratando-o como um modelo computacional por si só. Enquanto o CPS é utilizado como uma IR em compiladores, o cálculo de continuações oferece uma estrutura para raciocinar formalmente sobre computações onde o fluxo de controle é explicitamente representado. Os termos do cálculo de continuações, chamados de comandos, são descritos pelas seguintes regras:

$$M ::= x\langle\vec{x}\rangle \mid M\{x\langle\vec{x}\rangle = M\} \tag{5}$$

Aqui, $x\langle\vec{x}\rangle$ representa um salto (*jump*), isto é, uma chamada para a continuação x com os parâmetros $\vec{x}$, sendo essencialmente uma chamada direta para a continuação, enquanto $M\{x\langle\vec{x}\rangle = M\}$ representa um vínculo (*binding*), onde o corpo M está vinculado à continuação x com os parâmetros $\vec{x}$, isto é, uma chamada intermediária que, ao ser chamada, executará o próximo passo da computação.

A tradução para CPS converte um código escrito em estilo direto (programas com o controle de fluxo da aplicação implícito) no estilo de passagem de continuações (FLANAGAN et al., 1993). A ideia principal por trás da transformação para CPS é converter toda função para que ela não retorne um valor diretamente, e sim que passe o resultado para uma continuação.

Por exemplo, a Figura 2 contém um programa na linguagem Haskell para somar dois números em estilo direto, retornando o valor após o calcular. De maneira similar, na Figura 3 contém um programa similar em CPS. Nesta versão, entretanto, o controle de fluxo do programa

Figura 3 – Função soma em Haskell em CPS

Fonte: o autor

```haskell
addCps :: Int -> Int -> (Int -> r) -> r
addCps x y k = k (x + y)
```

é explícito, uma vez que é chamada a função *k* contendo o resultado da soma entre os argumentos.

3 TEORIA DE TIPOS

4 SISTEMA DAMAS-MILNER

4.1 ALGORITMO W

5 FORMALIZAÇÃO

6 IMPLEMENTAÇÃO

7 CONCLUSÃO

REFERÊNCIAS

AHO, Alfred V et al. **Compiladores: Princípios, técnicas e ferramentas**. 2th. ed. São Paulo, SP, Brasil: Pearson Education, 2008. Citado na página 15.

APPEL, Andrew W. **Compiling with continuations**. USA: Cambridge University Press, 1992. ISBN 0521416957. Citado 2 vezes nas páginas 16 e 17.

CHURCH, Alonzo. A set of postulates for the foundation of logic. **Annals of mathematics**, JSTOR, p. 346–366, 1932. Citado na página 16.

COOPER, Keith D; TORCZON, Linda. **Contruindo Compiladores**. 2th. ed. Rio de Janeiro, RJ, Brasil: Elsevier, 2014. Citado na página 15.

FLANAGAN, Cormac et al. The essence of compiling with continuations. **SIGPLAN Not.**, Association for Computing Machinery, New York, NY, USA, v. 28, n. 6, p. 237–247, jun 1993. ISSN 0362-1340. Disponível em: <https://doi.org/10.1145/173262.155113>. Citado 2 vezes nas páginas 12 e 17.

KENNEDY, Andrew. Compiling with continuations, continued. In: **Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming**. New York, NY, USA: Association for Computing Machinery, 2007. (ICFP '07), p. 177–190. ISBN 9781595938152. Disponível em: <https://doi.org/10.1145/1291151.1291179>. Citado na página 12.

MORRISETT, Greg et al. From system f to typed assembly language. **ACM Trans. Program. Lang. Syst.**, Association for Computing Machinery, New York, NY, USA, v. 21, n. 3, p. 527–568, may 1999. ISSN 0164-0925. Disponível em: <https://doi.org/10.1145/319301.319345>. Citado na página 12.

PLOTKIN, G.D. Call-by-name, call-by-value and the λ-calculus. **Theoretical Computer Science**, v. 1, n. 2, p. 125–159, 1975. ISSN 0304-3975. Disponível em: <https://www.sciencedirect.com/science/article/pii/0304397575900171>. Citado na página 12.

SØNDERGAARD, Harald; SESTOFT, Peter. Referential transparency, definiteness and unfoldability. **Acta Inform.**, Springer Nature, v. 27, n. 6, maio 1990. Citado na página 13.

THIELECKE, Hayo. **Categorical Structure of Continuation Passing Style**. [S.l.]: University of Edinburgh. College of Science and Engineering. School of Informatics., 1997. Citado na página 17.

TORRENS, Paulo; ORCHARD, Dominic; VASCONCELLOS, Cristiano. On the operational theory of the cps-calculus: Towards a theoretical foundation for irs. **Proc. ACM Program. Lang.**, Association for Computing Machinery, New York, NY, USA, v. 8, n. ICFP, aug 2024. Disponível em: <https://doi.org/10.1145/3674630>. Citado na página 13.