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
2025

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
2025

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, 2025.
51 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, 2025.

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
UDESC

Joinville, Novembro de 2025

A todos que me apoiaram nesses anos!

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

RESUMO

No contexto de compiladores, as representações intermediárias desempenham um papel fundamental, especialmente em ambientes de produção. O *Continuation Passing Style* (CPS) é uma dessas representações, notável para linguagens funcionais devido às otimizações avançadas que permite. Contudo, há uma escassez de pesquisas e implementações dessa IR com termos devidamente tipados, o que impede a detecção de uma gama de erros durante as etapas de compilação. Este trabalho, portanto, explora a formalização de um sistema de tipos para o CPS e o desenvolvimento de um algoritmo de inferência de tipos para essa representação, juntamente com sua implementação na linguagem de programação Haskell.

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

ABSTRACT

In the field of compiler, intermediate representations (IR) play a fundamental role, particularly in production environmnents. Continuation Passing Style (CPS) is one such representation, notable for functional languages due to the advanced optimizations it enables. However, research and implementations of CPS with well-typed terms remain limited, which restricts the detection of certain classes of errors during compilation stages. This work investigates the formalization of a type system for CPS and develops a type inference algorithm for this representation, along with its implementation in the Haskell programming language.

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

LISTA DE ILUSTRAÇÕES

LISTINGS

LISTA DE ABREVIATURAS E SIGLAS

IR	Intermediate Representation
CPS	Continuation Passing Style
SSA	Static Single Attribution
ANF	Administrative Normal Form
CBN	Call by Name
CBV	Call by Value

SUMÁRIO

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, apesar de existirem sistemas de tipos para CPS, como o proposto por Thielecke (1997) - um sistema de tipos monomórfico, muitas implementações deste, 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, este trabalho propõe apresentar e desenvolver uma formalização de um sistema de tipos para CPS, bem como um algoritmo de inferência de tipos para o mesmo. A escolha da linguagem de programação para a solução proposta será Haskell. Por ser 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 tem como objetivo formalizar uma extensão para o sistema de tipos para CPS proposto por Thielecke (1997), adicionando polimorfismo e investigar a possibilidade de propor um algoritmo de inferência, para esta representação intermediária.

1.2 OBJETIVOS ESPECÍFICOS

- Formalizar um sistema de tipos para CPS com suporte a polimorfismo;
- Propor e implementar em Haskell um algoritmo de inferência de tipos para CPS;
- 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 no estudo de sistemas de tipos, com o objetivo de proporcionar uma compreensão completa ao autor. A segunda etapa trata da formulação do algoritmo de inferência para o cálculo de continuações, junto com sua implementação.

No escopo deste trabalho, a validação do algoritmo se dará por meio de testes de implementação. Em etapa posterior, serão necessárias as provas de consistência e de completude do algoritmo em relação ao sistema de tipos proposto.

1.4 ESTRUTURA DO TRABALHO

Esta primeira etapa consistiu principalmente na fundamentação teórica e revisão bibliográfica no estudo de CPS e sistemas de tipos. Em razão disto, o Capítulo 2 contém os conceitos

e definições necessários para entendimento do tema. Este é separado em seções, tal que a Seção 2.1 aborda representação intermediária, com um aprofundamento em CPS na Subseção 2.1.1. Teoria de tipos é então apresentada na Seção 2.2, detalhando o Cálculo Lambda Simplesmente Tipado na Seção 2.3 Um aprofundamento no sistema Damas-Hindley-Milner na Seção 2.4, discutindo de maneira mais específica o algoritmo W na Subseção 2.4.1. O Capítulo 3 conta com o desenvolvimento do trabalho descrito e proposto.

2 FUNDAMENTAÇÃO TEÓRICA

2.1 REPRESENTAÇÃO INTERMEDIÁRIA DE CÓDIGO

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 aplicadas 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.

2.1.1 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 continuação, que representa o próximo passo a ser executado no programa. Assim, toda chamada de função se torna uma chamada de

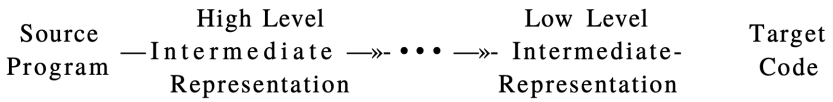

Figura 1 – Sequência de representações intermediárias
Fonte: (AHO et al., 2008)

cauda.

Uma chamada de cauda (do inglês *tail call*) ocorre quando a última instrução executada em uma função é uma chamada a outra função, sem que restem computações adicionais a serem feitas após essa chamada (MUCHNICK, 1997). Isso permite que a função atual libere seu quadro de ativação, otimizando o uso de memória, já que o compilador não precisa manter o estado da função anterior na pilha. Em constraste, uma chamada que não é de cauda ocorre quando ainda restam operações após a chamada, como somas ou multiplicações, o que exige que o quadro de ativação da função atual permaneça na pilha até a conclusão dessas operações.

Na Figura 2.1, o exemplo da função fatorial demonstra uma chamada que não é de cauda, pois a chamada recursiva `factorial(n - 1)` não é a última operação a ser realizada. A função precisa aguardar o retorno desta chamada para, então, multiplicar o resultado por `n`, o que impede a liberação do quadro de ativação até o término da multiplicação.

Em constrate, na Figura 2.2 como exemplo, tem-se uma versão da função fatorial que utiliza chamada de cauda. A função auxiliar `go` acumula o valor do cálculo diretamente em seu argumento `a`, e a chamada recursiva `go (n - 1) (a * n)` é a última instrução a ser executada. Como não há operações pendentes após a chamada recursiva, o compilador pode otimizar a função, reutilizando o quadro de ativação da função `go` para a chamada subsequente, tornando o cálculo mais eficiente.

```
1 factorial :: Int -> Int
2 factorial 0 = 1
3 factorial n = n * factorial (n - 1)
```

Listing 2.1 – Função fatorial em Haskell

```
1 go :: Int -> Int -> Int
2 go 1 a = a
3 go n a = go (n - 1) (a * n)
4
5 factorial :: Int -> Int
6 factorial 0 = 1
7 factorial n = go n 1
```

Listing 2.2 – Função fatorial em Haskell com chamada de cauda

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.

No cálculo lambda, as variáveis podem ser classificadas como livres ou ligadas, dependendo de seu contexto em um termo. Variáveis são consideradas livres quando não estão associadas a uma abstração de função. Por exemplo, no termo $\lambda x.y$, a variável y é livre, pois não está ligada a nenhum parâmetro introduzido. Em contraste, no termo $(\lambda x.x)y$, a variável x está ligada dentro do corpo da abstração, enquanto y permanece livre. Um termo sem variáveis livres é denominado fechado ou combinador; por exemplo, $\lambda x.\lambda y.xy$ é um combinador, pois todas as variáveis estão ligadas às suas respectivas abstrações.

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.

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

Note que, $e[y/x]$ indica a substituição de todas as ocorrências ligadas de x por y em e, desde que y não ocorra livre em e. Por exemplo, $\lambda x.x + z \to_\alpha \lambda y.y + z$.

β-redução: Aplicação de função.

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

Aqui, $e_1[e_2/x]$ denota a substituição de todas as ocorrências ligadas de x em e_1 por e_2. Para evitar conflitos com variáveis livres em e_2, aplica-se α-redução prévia. Por exemplo, em $(\lambda x.\lambda y.x + y)(y + 1)$, renomeia-se y para z na abstração interna:

$$(\lambda x.\lambda z.x + z)(y + 1) \to_\beta \lambda z.(y + 1) + z.$$

η-redução: Expansão de função.

$$\lambda x.(e\,x) \to_\eta e \quad \text{se } x \notin \mathrm{FV}(e) \tag{4}$$

Tal que $\mathrm{FV}(e)$ denota as variáveis livres em e. Por exemplo, $\lambda x.(\lambda y.y + 1)\,x \to_\eta \lambda y.y + 1$.

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

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 $x+1$. Ao converter funções para CPS, o fluxo de controle do programa passa a ser gerenciado de forma explícita através de chamadas de função que encadeiam as etapas subsequentes. Essa transformação traz diversas vantagens no contexto de compiladores, pois torna explícitas as informações normalmente implícitas na pilha de execução e, assim, abre espaço para a aplicação de uma série de otimizações (APPEL, 1992).

Em primeiro lugar, o CPS expõe todas as chamadas de cauda, permitindo a eliminação de recursividade em cauda (TCO, do inglês *tail-call optimization*). Essa otimização possibilita que chamadas recursivas sejam realizadas sem o acúmulo de quadros de ativação na pilha, reduzindo o uso de memória e evitando estouros de pilha em programas fortemente recursivos.

Além disso, a natureza explícita das chamadas em CPS favorece a expansão *inline* de funções. Como cada chamada é clara e separada, o compilador pode substituir uma chamada de função por sua definição diretamente, reduzindo o *overhead* de chamadas e potencialmente expondo outras oportunidades de otimização.

Outro ponto importante é a representação de *closures*. Em CPS, como todo o contexto necessário à computação é passado de maneira explícita, o compilador consegue construir e manipular *closures* com maior eficiência, uma vez que o ambiente da função está sempre disponível e bem definido.

A transformação para CPS também facilita a alocação de registradores. Com o fluxo de controle explicitado, o compilador pode prever melhor o tempo de vida das variáveis e organizar os dados de forma que o uso de registradores seja maximizado e o número de acessos à memória minimizado (APPEL, 1992).

Por fim, como todo o fluxo de execução é expresso por chamadas encadeadas, o código gerado permite análises estáticas mais precisas. O compilador pode, por exemplo, detectar com mais facilidade padrões de execução, dependências entre expressões e oportunidades de reordenação ou eliminação de código redundante.

Dessa forma, a conversão para CPS não apenas preserva a semântica da computação original, mas também fornece ao compilador uma estrutura rica e detalhada para aplicar otimizações de maneira eficaz.

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 (do inglês *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 (do inglês *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. Vale ressaltar que Appel e Jim (1997) possuem uma sintaxe diferente para o cálculo de continuações, onde os termos são respectivamente representados como $k(\vec{x})$ e `let` $k(\vec{x}) = c$ `in` b.

A tradução para CPS converte um código escrito em estilo direto (onde o controle de fluxo é implícito) para o estilo de passagem de continuações (FLANAGAN et al., 1993). A principal ideia por trás dessa transformação é modificar as funções para que elas não retornem um valor diretamente, mas, em vez disso, passem o resultado para uma continuação.

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

Listing 2.3 – Função soma em Haskell em Estilo Direto

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

Listing 2.4 – Função soma em Haskell em Estilo Direto

Por exemplo, a Figura 2.3 apresenta um programa na linguagem Haskell que soma dois números no estilo direto, retornando o valor após realizar o cálculo. Já a Figura 2.4 mostra um programa equivalente em CPS. Nesta versão, o controle de fluxo do programa é explícito, pois a função k é chamada para processar o resultado da soma dos argumentos.

```haskell
sub :: Int -> Int -> Int
sub x y = x - y

mult :: Int -> Int -> Int
mult x y = x * y

factorial :: Int -> Int
factorial 0 = 1
factorial n = mult n (factorial (sub n 1))
```

Listing 2.5 – Função fatorial em Haskell em Estilo Direto

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

```
3
4  multCps :: Int -> Int -> (Int -> r) -> r
5  multCps x y k = k (x * y)
6
7  factorialCps :: Int -> (Int -> r) -> r
8  factorialCps 0 k = k (1)
9  factorialCps n k =
10   subCps n 1 (\nMinus1 ->
11     factorialCps nMinus1 (\factNMinus1 ->
12       multCps n factNMinus1 k))
```

Listing 2.6 – Função fatorial em Haskell em CPS

Para ilustrar melhor, a Figura 2.5 apresenta um programa em Haskell que calcula o fatorial no estilo direto, utilizando funções definidas para multiplicação e subtração. Na Figura 2.6, um programa similar em CPS é definido, com as funções auxiliares também transformadas para CPS. Na função `factorialCps` é possível notar duas funções lambda (continuações), `nMinus1` e `factNMinus1`. A primeira continuação guarda o resultado da operação $n-1$, enquanto a segunda recebe recursivamente o cálculo do fatorial de $n-1$, multiplica por n e finalmente passa o resultado para a continuação k.

Outro fato importante a ser observado nos códigos apresentados é a tipagem das funções. Na função de soma, definida na Figura 2.3, a função tem tipo $Int \to Int \to Int$, ou seja, ela recebe dois inteiros e retorna um inteiro. Já a função de soma em CPS, definida na Figura 2.4, possui o tipo $Int \to Int \to (Int \to r) \to r$. Isso significa que a função recebe dois inteiros e uma continuação, que é uma função de tipo $Int \to r$, onde `r` pode ser qualquer tipo, e retorna esse mesmo tipo `r`.

Essa transformação de tipo reflete a diferença fundamental entre o estilo direto e o CPS: em vez de retornar um valor diretamente, a função em CPS recebe uma continuação que especifica o próximo passo da computação. O mesmo padrão pode ser observado nas funções para o cálculo do fatorial nas Figuras 2.5 e 2.6. No estilo direto, a função `factorial` tem o tipo $Int \to Int$, enquanto na versão CPS, a função `factorialCps` tem o tipo $Int \to (Int \to r) \to r$.

Essa correspondência entre os tipos não é uma coincidência. Como discutido por (TORRENS, 2019), uma função em estilo direto com tipo $A \to B$ pode ser transformada em uma função em CPS com o tipo $A \to (B \to \bot) \to \bot$. Aqui, $\bot$ representa o tipo dos valores que nunca retornam, uma característica associada ao estilo de passagem de continuações, onde as funções são compostas de forma a encadear continuações até que a execução termine de maneira explícita.

Este exemplo simples da função fatorial em CPS ilustra as dificuldades inerentes ao uso de continuações explícitas, como a verbosidade do código, complexidade de compreensão e a propensão a erros. No entanto, apesar desses desafios, o CPS se mostra extremamente adequado para a aplicação de otimizações, sendo uma escolha eficiente para representações intermediárias, especialmente em cenários onde o desempenho é essencial.

2.2 TEORIA DE TIPOS

A Teoria de Tipos, conforme apresentada por (COQUAND, 2022), foi introduzida por Russell em 1908 ao encontrar um paradoxo na Teoria de Conjuntos, conhecido atualmente como o Paradoxo de Russell:

$$\text{Seja } R = \{x \mid x \notin x\}, \text{ então } R \in R \iff R \notin R \tag{6}$$

Ou seja, considere R como o conjunto dos conjuntos que não contêm a si mesmos. A contradição surge ao observar que, se o conjunto R contém a si mesmo, isso implica que R não contém a si mesmo, e vice-versa.

Outra maneira de descrever esse paradoxo é através do Paradoxo do Barbeiro: imagine uma cidade com apenas um barbeiro, onde ele somente barbeia aqueles que não se barbeiam. O paradoxo surge quando perguntamos: "Quem barbeia o barbeiro?" Ele não pode fazer sua própria barba, pois barbeia apenas aqueles que não fazem a própria barba. No entanto, se ele não faz sua própria barba, então pertence ao grupo daqueles que devem ser barbeados pelo barbeiro, logo, ele deveria barbear-se. Essa situação gera uma contradição semelhante ao Paradoxo de Russell.

Assim como os paradoxos na Teoria de Conjuntos expuseram a necessidade de fundamentos mais rigorosos para a matemática, a Teoria de Tipos surgiu como uma estrutura lógica para evitar inconsistências. Na computação, essa ideia se reflete nos sistemas de tipos modernos, que impedem comportamentos paradoxais ou indefinidos em programas. Por exemplo, ao restringir operações a tipos específicos, evita-se que funções sejam aplicadas a entidades incompatíveis — análogo a evitar que o barbeiro pertença ao conjunto que gera a contradição.

Atualmente, a principal aplicação da Teoria de Tipos está na formalização de sistemas de tipos para linguagens de programação. Um sistema de tipos garante a ausência de certos comportamentos dos programas classificando os valores computados em cada uma de suas sentenças (PIERCE, 2002). Além disso, atribuir e verificar tipos para cada construção presente nos programas têm várias utilidades, como fornecer informações para auxiliar na modularização de programas, otimização de código executada pelo compilador, além de poder ser usada como documentação do código.

No contexto das linguagens de programação, podemos distinguir três categorias principais de tipos: tipos simples, tipos polimórficos e tipos dependentes (PIERCE, 2002). Tipos simples atribuem um tipo fixo a cada termo, enquanto tipos polimórficos introduzem a noção de generalidade, permitindo que funções possam ser aplicadas a argumentos de diferentes tipos sem a necessidade de serem redefinidas para cada um. Já os tipos dependentes permitem que tipos dependam de valores.

Um exemplo de tipo simples é uma função que opera sobre números inteiros. Esta função recebe um número inteiro e retorna outro número inteiro. Seu tipo, portanto, é representado como $Int \to Int$, indicando que tanto a entrada quanto a saída são do tipo inteiro.

Um exemplo de polimorfismo é a função identidade, que recebe um elemento de qualquer tipo e retorna o mesmo elemento. Seu tipo é expresso como $a \to a$, onde a pode ser qualquer tipo, caracterizando o polimorfismo paramétrico. Nesse caso, a função mantém o mesmo comportamento para todos os tipos, sem necessidade de reimplementação.

Em linguagens com suporte a tipos dependentes, um exemplo seria o de um vetor cujo comprimento (número de elementos) faz parte de seu tipo. Nesse caso, uma função de concatenação de vetores deve garantir que somente vetores com tipos compatíveis em relação ao comprimento possam ser concatenados. O tipo da função de concatenação seria algo como[1] $Vector(n) \to Vector(m) \to Vector(n+m)$, onde n e m são valores que representam os comprimentos dos vetores e fazem parte da definição de tipo.

No contexto do polimorfismo, Pierce (2002) define duas principais variedades: o polimorfismo paramétrico, (como é o caso da função identidade), e o polimorfismo com sobrecarga. No primeiro, uma única definição opera genericamente, mantendo o mesmo comportamento para todos os tipos. Já no segundo, o comportamento varia conforme o tipo dos argumentos, permitindo múltiplas implementações — como na sobrecarga de operadores, onde a função selecionada depende dos tipos dos operandos.

O polimorfismo desempenha um papel crucial na inferência de tipos. Em linguagens como Haskell, o sistema de tipos deduz automaticamente tipos genéricos sempre que possível, permitindo que funções como a identidade ($\lambda x.x$, tipada como $\forall\alpha.\alpha \to \alpha$) sejam usadas de forma universal. Por outro lado, funções com restrições específicas ilustram o polimorfismo com sobrecarga. Por exemplo, a função de soma no Listing 2.3, originalmente definida como $Int \to Int \to Int$, pode ser generalizada para operar sobre quaisquer tipos numéricos utilizando classes de tipos. Isso é alcançado ao substituir a tipagem explícita por uma restrição como `Num a => a -> a -> a`, onde `Num a` indica que `a` deve pertencer à classe de tipos numéricos.

Em Haskell, classes de tipos são mecanismos que habilitam o polimorfismo de sobrecarga. A função `sumList`, por exemplo, (Listing 2.7) é definida com a restrição `Num a`, permitindo que opere sobre listas de inteiros, `Float`, `Double` ou outros tipos numéricos. Essa abordagem combina flexibilidade e segurança: o polimorfismo com sobrecarga garante que a função generalize seu comportamento apenas dentro de um domínio específico (*e.g.*, números), evitando inconsistências.

```
1 sumList :: (Num a) => [a] -> a
2 sumList [] = 0
3 sumList (x : xs) = x + sumList xs
```

Listing 2.7 – Função somatório de elementos de lista em Haskell

[1] A notação exata pode variar entre diferentes linguagens de programação que suportam tipos dependentes. A estrutura apresentada serve apenas como uma ilustração conceitual do comportamento esperado.

2.3 CÁLCULO LAMBDA SIMPLESMENTE TIPADO

O Cálculo Lambda Simplesmente Tipado é uma das primeiras e mais simples variantes do Cálculo Lambda que incorpora tipos em sua estrutura (CHURCH, 1940). Enquanto o cálculo lambda original não faz distinção entre diferentes tipos de dados, no Cálculo Lambda Simplesmente Tipado os termos são anotados com tipos. Cada função recebe e retorna valores de tipos específicos, o que permite prevenir uma série de erros comuns em programas, como a aplicação de funções a argumentos incorretos. Além disso, o sistema de tipos serve como uma ferramenta de verificação durante a compilação de programas, assegurando que erros de tipo sejam detectados antes da execução. Dessa forma, ele não apenas facilita a criação de software mais robusto, mas também oferece uma base formal para o estudo de linguagens de programação (PIERCE, 2002).

A sintaxe básica do Cálculo Lambda Simplesmente Tipado inclui:

- Variáveis: $x, y, z, \ldots$
- Tipos: $T ::= \mathbf{Int} \mid \mathbf{Bool} \mid T \to T$
- Termos: $\lambda x : T.\tau \mid \tau_1 \tau_2 \mid x$

No Cálculo Lambda Simplesmente Tipado, cada variável possui um tipo atribuído e os termos são construídos com base nesses tipos. Por exemplo, a abstração de função $\lambda x : T.\tau$ define uma função onde a variável x é de tipo T e o corpo da função, τ, é um termo. A aplicação de função $\tau_1 \tau_2$ indica que τ_1 é uma função que é aplicada ao argumento τ_2, o qual deve ter um tipo compatível com o tipo esperado por τ_1. Essa formalização facilita a composição de funções e o raciocínio sobre a estrutura dos programas, pois cada termo pode ser avaliado dentro de um contexto de tipagem. A sintaxe dos tipos, como $T \to T$, define uma função que aceita um argumento do tipo T e retorna um valor também do tipo T.

A inferência de tipos no Cálculo Lambda Simplesmente Tipado assegura que cada expressão tenha um tipo bem-definido, baseado nas regras de tipagem. A tipagem de termos é feita através de um conjunto de regras formais que garantem a consistência dos tipos no programa. Por exemplo, a regra de tipagem para abstrações lambda é a seguinte:

$$\frac{\Gamma, x : T_1 \vdash \tau : T_2}{\Gamma \vdash (\lambda x : T_1.\tau) : T_1 \to T_2}$$

Isso significa que, se o termo τ possui o tipo T_2 sob o contexto onde x possui o tipo T_1, então a abstração $\lambda x : T_1.\tau$ tem o tipo $T_1 \to T_2$. Essa verificação de tipo garante que, ao aplicar a função, o tipo do argumento corresponde ao tipo esperado pela função.

O Cálculo Lambda Simplesmente Tipado está intimamente relacionado com a lógica intuicionista proposicional. Esse vínculo é formalizado pela Correspondência Curry-Howard, que estabelece uma correspondência direta entre proposições lógicas e tipos, e entre provas e programas. Em outras palavras, tipos podem ser interpretados como proposições lógicas, e

termos tipados como provas dessas proposições (PIERCE, 2002). Por exemplo, o tipo $A \to B$ no Cálculo Lambda Simplesmente Tipado pode ser visto como a implicação lógica "se A, então B". Assim, uma função que aceita um argumento do tipo A e retorna um valor do tipo B é equivalente a uma prova de que A implica em B. Esse princípio permite usar ferramentas da teoria de tipos para construir provas formais de teoremas em lógica intuicionista, fornecendo uma base teórica robusta para assistentes de prova automatizados, como o Coq (COQUAND; HUET, 1988).

Além disso, a Correspondência Curry-Howard não apenas conecta tipos e lógica, mas também oferece um método sistemático para projetar e raciocinar sobre sistemas de inferência de tipos, garantindo que programas tipados sejam corretos em relação às especificações lógicas. A inferência de tipos desempenha um papel fundamental na programação funcional moderna, sendo inicialmente introduzida com a linguagem ML por Damas e Milner (1982), com o algoritmo W. A linguagem Haskell extende o sistema Damas-Milner, adicionando principalmente o suporte a sobrecarga de funções.

2.4 SISTEMA DAMAS-MILNER

O sistema Damas-Milner, introduzido por Robin Milner e posteriormente formalizado em maior detalhe por Luis Damas (MILNER, 1978; DAMAS; MILNER, 1982), é um dos sistemas de tipos mais influentes para linguagens funcionais. Este sistema tem como principal característica a inferência automática de tipos polimórficos, sem a necessidade de anotações explícitas por parte do programador, ocorrendo em linguagens como ML, Haskell e OCaml. A sua base é o cálculo lambda com polimorfismo paramétrico, introduzido via `let`, permitindo que funções possam operar sobre múltiplos tipos de maneira genérica.

A introdução do sistema Damas-Milner trouxe duas contribuições principais: a definição de um sistema de tipos robusto e a criação de um algoritmo, o Algoritmo W, capaz de inferir o tipo mais geral (também chamado de *principal type-scheme*), conforme demonstrado em Damas (1984). O algoritmo é consistente e completo em relação ao sistema de tipos: a consistência assegura que todo tipo inferido é correto, ou seja, pode ser derivado pelo sistema de tipos; já a completude garante que qualquer tipo derivado pelo sistema será uma instância do tipo inferido pelo algoritmo. Como resultado, a linguagem ML e suas derivadas se tornaram notórias por fornecer ao programador a capacidade de escrever programas sem erros de tipo detectáveis durante a compilação, permitindo um desenvolvimento mais seguro e robusto (MILNER, 1978; DAMAS, 1984).

A sintaxe do sistema Damas-Milner define as expressões e os tipos usados no processo de inferência. Abaixo, segue a gramática das expressões e tipos:

Variáveis	x
Expressões	$e ::= x \mid e\ e' \mid \lambda x.e \mid \texttt{let}\ x = e\ \texttt{in}\ e'$
Variáveis de tipo	α
Tipos primitivos	ι
Tipos	$\tau ::= \alpha \mid \iota \mid \tau \to \tau\alpha$
Schemes	$\sigma ::= \forall\alpha.\sigma \mid \tau$

Na sintaxe, x representa variáveis que podem ser nomes de qualquer identificador, e e descreve expressões que podem ser variáveis, aplicações de função, funções anônimas ou declarações `let`, que introduzem polimorfismo através de generalização de tipos. α é usado para representar variáveis de tipos. Os tipos primitivos ι são usados para representar tipos constantes. Tipos τ podem ser tanto variáveis de tipo quanto funções entre tipos. Por fim, σ denota *schemes*, ou tipos polimórficos, que podem quantificar variáveis de tipo, permitindo reutilização de variáveis de tipos em diferentes contextos.

O polimorfismo no sistema Damas-Milner é introduzido pelas expressões `let`, que permitem a generalização de tipos. Ao declarar uma variável ou função usando `let`, o tipo inferido é generalizado para ser utilizado de maneira polimórfica na expressão que ocorre após o `in`. Isso significa que, ao declarar uma função como `let id` $= \lambda x.x$, o sistema deduz o tipo mais geral $\forall\alpha.\alpha \to \alpha$, que pode ter sua variável de tipo α instanciada para diferentes tipos conforme for necessário.

A inferência de tipos envolve dois processos principais: generalização e instanciação. A generalização ocorre quando o sistema identifica que uma expressão pode ser tipada com um tipo mais geral, permitindo que seja reutilizada de maneira polimórfica. Já a instanciação ocorre quando um tipo polimórfico é aplicado a um tipo concreto, especializando-o para um uso específico. Esse mecanismo garante a flexibilidade do sistema, ao mesmo tempo que mantém a segurança garantida pela inferência de tipos. Por exemplo, considere a expressão `let id` $= \lambda x.x$ `in (id 1, id 'a')`. O sistema generaliza o tipo de `id` para $\forall\alpha.\alpha \to \alpha$, e instancia este tipo tanto para inteiros quanto para caracteres nas duas aplicações subsequentes.

Outro conceito presente no sistema Damas-Milner é a substituição de tipos, onde estes são mapeados para outros tipos ou para variáveis de tipo. Formalmente, uma substituição de tipos é representada como um mapeamento finito de variáveis de tipo para tipos, denotado por S, e pode ser escrito na forma $[\alpha_1 \mapsto \tau_1, \alpha_2 \mapsto \tau_2, \ldots, \alpha_n \mapsto \tau_n]$. Aqui, α_i são variáveis de tipo distintas e τ_i são os tipos correspondentes. Em outras palavras, S associa cada variável de tipo α_i a um tipo τ_i específico.

A aplicação de uma substituição S em um tipo τ, denotada por $S\tau$, resulta na substituição de todas as ocorrências livres de α_i em τ por τ_i. Esse conceito de substituição é fundamental para o processo de instanciação de tipos, que será discutido a seguir. A definição formal da aplicação de substituições é dada por:

$$
\begin{aligned}
S\alpha_i &\equiv \tau_i, \\
S\alpha &\equiv \alpha, \quad \text{se } \alpha \notin \{\alpha_1, \alpha_2, \ldots, \alpha_n\}, \\
S(\tau_1 \to \tau_2) &\equiv S\tau_1 \to S\tau_2, \\
S(\forall\alpha.\sigma) &\equiv S'\sigma, \quad \text{onde } S' = S \setminus [\alpha \mapsto \_].
\end{aligned}
$$

Figura 2 – Regras de aplicação da substituição de tipos
Fonte: (SILVA, 2019)

onde o símbolo de subtração de conjuntos ($\setminus$) indica que a substituição S' é a substituição S restrita ao conjunto de mapeamentos que não envolvem a variável α.

A instanciação de tipos é um processo em que um esquema de tipo $\sigma = \forall\alpha_1 \ldots \alpha_m.\tau$ é transformado em um tipo específico substituindo suas variáveis quantificadas por tipos concretos. Se S é uma substituição, então $S\sigma$ é o esquema de tipo obtido substituindo cada ocorrência livre de α_i em σ por τ_i, renomeando as variáveis genéricas de σ, se necessário. O tipo resultante $S\sigma$ é chamado de uma instância de σ (DAMAS; MILNER, 1982). Esse processo é essencial para adaptar esquemas de tipos polimórficos a situações específicas em um programa, mantendo a flexibilidade e segurança do sistema de tipos.

Um esquema de tipo também pode ter uma instância genérica $\sigma' = \forall\beta_1 \ldots \beta_n.\tau'$, se existir uma substituição $[\tau_i/\alpha_i]$ tal que $\tau' = [\tau_i/\alpha_i]\tau$, e as variáveis β_j não aparecem livres em σ. Nesse caso, escrevemos $\sigma > \sigma'$, indicando que σ é mais geral do que σ'. Vale notar que a instanciação atua sobre variáveis livres, enquanto a instanciação genérica lida com variáveis ligadas.

O sistema de tipos de Damas e Milner é definido por um conjunto de regras de inferência de tipos, apresentadas na Figura 3, que são usadas para determinar os tipos das expressões no sistema. Essas regras são representadas por meio de julgamentos de tipos da forma $\Gamma \vdash e : \sigma$, onde Γ é o contexto, um conjunto de suposições com pares de variáveis e seus tipos: (e, σ), e é a expressão sendo tipada, e σ ou τ são o tipo inferido no contexto.

As regras de inferência são interpretadas de baixo para cima. Por exemplo, na regra da tautologia (TAUT):

$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma}$$

significa que, se em um contexto Γ, a variável x possui o tipo σ, então podemos concluir que x tem o tipo σ no mesmo contexto. Isso reflete o fato de que a associação de tipos no contexto é preservada.

Na regra de generalização (GEN), a condição de que α não seja livre em Γ assegura que o tipo generalizado não dependa de nenhum tipo específico presente no contexto. Isso permite que o tipo $\forall\alpha.\sigma$ seja usado de forma polimórfica em diferentes partes do programa.

Essas regras garantem a solidez do sistema, preservando a segurança dos tipos ao inferir automaticamente os tipos mais gerais possíveis para as expressões.

Antes de apresentar o Algoritmo W, é importante observar que ele é uma implementação

$$\text{TAUT:} \quad \frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma}$$

$$\text{ABS:} \quad \frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash (\lambda x.e) : \tau \to \tau'}$$

$$\text{APP:} \quad \frac{\Gamma \vdash e : \tau' \to \tau \quad \Gamma \vdash e' : \tau'}{\Gamma \vdash (e\ e') : \tau}$$

$$\text{LET:} \quad \frac{\Gamma \vdash e : \sigma \quad \Gamma, x : \sigma \vdash e' : \tau}{\Gamma \vdash (\texttt{let}\ \ x = e\ \texttt{in}\ e') : \tau}$$

$$\text{INST:} \quad \frac{\Gamma \vdash e : \sigma}{\Gamma \vdash e : \sigma'} \quad (\sigma > \sigma')$$

$$\text{GEN:} \quad \frac{\Gamma \vdash e : \sigma}{\Gamma \vdash e : \forall \alpha \sigma} \quad (\alpha \text{ não livre em } \Gamma)$$

Figura 3 – Regras de Inferência do sistema Damas-Milner
Fonte: o autor. Adaptado de (DAMAS; MILNER, 1982)

prática das regras de inferência aqui descritas, usando o conceito de unificação para resolver as equações de tipo geradas durante a inferência. A seguir, será discutido em detalhes o funcionamento do Algoritmo W.

2.4.1 Algoritmo W

O Algoritmo W, introduzido em Damas e Milner (1982), é um algoritmo eficiente[2] para inferência de tipos em linguagens de programação funcional. Ele se baseia no processo de unificação para solucionar equações de tipos geradas durante a análise de expressões, atribuindo os tipos mais gerais possíveis, ou seja, os tipos mais polimórficos que ainda garantem a consistência do sistema.

A unificação é o processo de encontrar uma substituição de variáveis de tipo que torna dois tipos dados equivalentes. Formalmente, dados dois tipos τ_1 e τ_2, a unificação procura uma substituição S tal que $S\tau_1 = S\tau_2$. Se tal substituição existe, os tipos são considerados unificáveis e S é chamada de solução unificadora. Caso contrário, os tipos são incompatíveis.

[2] Embora seja eficiente na grande maioria dos casos, há situações em que o Algoritmo W apresenta desempenho exponencial, como ocorre em expressões com recursão polimórfica, conforme discutido em (VASCONCELLOS, 2004).

O algoritmo de unificação, *unify*, descrito na Figura 4, gera a unificação mais geral possível, operando recursivamente sobre a estrutura dos tipos. Ele verifica se os tipos são idênticos, se uma variável de tipo pode ser substituída por outro tipo, ou, no caso de tipos compostos, se suas partes podem ser unificadas independentemente.

Figura 4 – Algoritmo de unificação no formato de função.

```
unify(α, α) =
    retorna [ ]
unify(α, τ) =
    se occurs(α, τ), então
        falha
    senão
        retorna [α ↦ τ]
unify(τ, α) =
    se occurs(α, τ), então
        falha
    senão
        retorna [α ↦ τ]
unify(τ1 → τ2, τ1' → τ2') =
    retorna unify(τ1, τ1') ∘ unify(τ2, τ2')
```

Fonte: autor. Adaptado de (RIBEIRO; CAMARÃO, 2016)

O algoritmo *unify* faz uso da função de verificação de ocorrência *occurs* apresentado na Figura 5, que por sua vez, tem como propósito evitar substituições que introduzam ciclos, como $[\alpha \mapsto \alpha \to \alpha]$, que resultaria em inconsistência no sistema de tipos (RIBEIRO; CAMARÃO, 2016). Esta função verifica recursivamente se uma variável de tipo α aparece em um tipo τ.

Figura 5 – Algoritmo de verificação de ocorrência no formato de função.

```
occurs(α, τ1 → τ2) =
    retorna occurs(α, τ1) ∨ occurs(α, τ2)
occurs(α, α) =
    retorna Verdadeiro
occurs(α, τ) =
    retorna Falso
```

Fonte: autor. Adaptado de (RIBEIRO; CAMARÃO, 2016)

O Algoritmo W, descrito na Figura 6 é um método utilizado para inferência de tipos em expressões de linguagens funcionais. Ele atribui os tipos mais gerais possíveis a cada

subexpressão, garantindo a consistência com as operações definidas. O algoritmo combina a unificação com regras de inferência de tipos para deduzir o tipo de uma expressão, explorando o polimorfismo de forma eficiente.

Figura 6 – Algoritmo W no formato de função.

```
W(Γ, x) =
    se Γ(x) = ∀ α1 ... αn . τ, então
        retorna ([αi ↦ τi]τ, Id), onde α'i fresh
    senão
        falha

W(Γ, e e') =
    (τ, S1) ← W(Γ, e)
    (τ', S2) ← W(S1Γ, e')
    S ← unify(S2τ, τ' → α), onde α fresh
    retorna (Sα, S ∘ S2 ∘ S1)

W(Γ, λx.e) =
    (τ, S) ← W(Γ[x : α], e), onde α fresh
    retorna (S(α → τ), S)

    (τ, S1) ← W(Γ, e)
    (τ', S2) ← W(S1 Γ[x : gen(S1Γ, τ)], e')
    retorna (τ', S2 ∘ S1)
```

Fonte: (SILVA, 2019)

O funcionamento pode ser analisado para os diferentes tipos de expressões a seguir. Para uma variável x, o algoritmo verifica se existe uma atribuição de tipo para x no contexto Γ. Se x estiver associada a um tipo polimórfico da forma $\forall\alpha_1 \ldots \alpha_n.\tau$, realiza-se a substituição das variáveis ligadas α_i por novos tipos frescos, que não aparecem em outros lugares do contexto, e retorna-se o tipo resultante, juntamente com a substituição identidade. Caso x não esteja no contexto, a inferência falha.

Para uma aplicação de função $e\ e'$, o algoritmo infere recursivamente os tipos das subexpressões e e e'. A partir desses tipos, unifica o tipo de e com um tipo função $\tau' \to \alpha$, onde α é um novo tipo variável introduzido durante a unificação. A substituição resultante é então aplicada ao tipo inferido de e e o algoritmo retorna o tipo correspondente à aplicação.

Quando a expressão é uma abstração $\lambda x.e$, o algoritmo atualiza o contexto adicionando uma nova variável de tipo para x e procede inferindo o tipo de e. O tipo função resultante, $\alpha \to \tau$,

é então retornado como o tipo inferido para a abstração.

No caso de expressões do tipo `let`, onde uma variável é definida localmente, o algoritmo primeiro infere o tipo da expressão vinculada, seguido pelo tipo do corpo da expressão. O contexto é atualizado para incluir a variável definida com um tipo generalizado, permitindo polimorfismo na expressão resultante. A generalização é aplicada ao tipo inferido, de forma que as variáveis de tipo que não estão presentes no contexto sejam quantificadas, garantindo assim um nível adequado de polimorfismo na inferência de tipos.

Essas etapas garantem que o Algoritmo W seja capaz de inferir tipos de forma eficiente, atribuindo os tipos mais polimórficos possíveis para expressões em linguagens funcionais e explorando as capacidades do sistema de tipos.

3 DESENVOLVIMENTO

Será abordado, neste capítulo, a contribuição prática do trabalho, apresentando formalização do sistema de tipos proposto, sua implementação na linguagem Haskell, e os resultados experimentais obtidos. No que diz respeito ao sistema de tipos, em virtude da limitação de tempo e da grandeza deste trabalho, não foi feito a prova de que este é correto ou completo, há somente indícios que levam a crer que o trabalho feito está correto, junto de resultados corretos em testes efetuados. Uma explicação mais detalhada sobre esta parte será dada na Seção 3.1.

Na Seção 3.2, será descrito as decisões de projeto que orientaram a implementação. O ambiente de execução utilizado foi o compilador *The Glorious Glascow Haskell Compilation System* (GHC), na versão 9.12.2 em conjunto com o gerenciador de projetos Cabal na versão 3.14.1. O código-fonte completo está disponível publicamente no repositório *cps-type-inferer*[1] do GitHub. Na Seção 3.2, terá um aprofundamento maior na implementação.

Considerações, como resultados e descobertas importantes, serão entradas em detalhes na Seção 3.3, onde será discutido sobre a relevância das contribuições e próximos passos a serem tomados em relação a esta pesquisa na seção 3.4.

3.1 FORMALIZAÇÃO

Em razão da natureza mais prática deste trabalho, a notação utilizada para representar o cálculo de continuações será a mesma utilizada por (APPEL; JIM, 1997), a sintaxe do 'let'. O sistema de tipos formalizado aqui foi fortemente inspirado no sistema de Damas e Milner, explicado na Seção 2.4, onde suas regras foram adaptadas de modo que elas se enquadrem no sistema polimórfico baseado em continuações. Em particular, o contexto Γ associa variáveis a politipos e define julgamentos distintos para representar átomos e comandos. A distinção destes se mostra necessária uma vez que é levado em consideração o comportamento não retornável das continuações.

A sintaxe do sistema conta com expressões e tipos usados no processo de tipagem e de inferência de tipos. Abaixo, segue a gramática das expressões e tipos presentes[2]:

$$
\begin{array}{lrcl}
\text{Átomos} & a & ::= & x \mid n \\
\text{Comandos} & b & ::= & x(\vec{a}) \mid \texttt{let}\ x(\vec{x}) = b\ \texttt{in}\ b \\
\\
\text{Monotipo} & \tau & ::= & \alpha \mid \texttt{int} \mid \neg\vec{\tau} \\
\text{Politipo} & \sigma & ::= & \forall\vec{\alpha}.\tau \\
\text{Contexto} & \Gamma & ::= & \cdot \mid \Gamma,\, x\colon \sigma
\end{array}
$$

[1] <https://github.com/bidinpithecus/cps-type-inferer>

[2] Vale destacar que, todas as formalizações presentes aqui nesta seção, foram feitas pelo coorientador em reunião juntamente do autor, onde esse explicava suas motivações para atingir o resultado. Ainda, no momento da produção deste trabalho, não foi feita uma publicação contendo estas formalizações para que seja devidamente referenciada.

Na sintaxe apresentada, a representa os átomos. Isto é, variáveis do programa (x) e literais inteiros (n) formam os elementos primitivos do sistema. Os comandos b, por sua vez, são as expressões, explicadas com mais detalhes na Seção 2.1.1, sendo a primeira o *jump*, e a segunda o *bind*. Três elementos distintos compõem os tipos presentes neste sistema. Os monotipos (τ), são os tipos que não possuem quantificação (monomórficos), podendo ser variáveis de tipo (α), tipos numéricos inteiros (`int`), ou ainda, tipos negados ($\neg\vec{\tau}$), usados para representar funções que retornam absurdos. Já os politipos (σ), são responsáveis por garantir a quantificação universal de variáveis de tipos (polimórficos). Por fim, o contexto (Γ) contém o mapeamento de cada variável para um politipo (σ).

As regras sintáticas de tipagem do sistema de tipos, contando com elementos também presentes no Sistema Damas-Milner são ilustrados a seguir:

$\boxed{\Gamma \vdash a\colon \tau}$

$$\frac{x\colon \sigma \in \Gamma \qquad \sigma \sqsubseteq \tau}{\Gamma \vdash x\colon \tau}\,[\texttt{Var}]$$

$$\frac{}{\Gamma \vdash n\colon \texttt{int}}\,[\texttt{Int}]$$

$\boxed{\Gamma \vdash b}$

$$\frac{\Gamma \vdash k\colon \neg\vec{\tau} \qquad \Gamma \vdash \vec{a}\colon \vec{\tau}}{\Gamma \vdash k(\vec{a})}\,[\texttt{Jump}]$$

$$\frac{\Gamma, \vec{x}\colon \vec{\tau} \vdash c \qquad \Gamma,\, k\colon \overline{\Gamma}(\neg\vec{\tau}) \vdash b}{\Gamma \vdash \texttt{let}\ k(\vec{x}) = c\ \texttt{in}\ b}\,[\texttt{Bind}]$$

Aqui, para os átomos do sistema, a partir da regra [`Var`] é esperado que as variáveis, como vêm do contexto, possuam um tipo polimórfico σ. Existe ainda um tipo monomórfico τ que a relação de ordem $\sqsubseteq$ diz ser menos geral do que σ. Assim, em um contexto Γ, uma variável x terá tipo τ caso esta esteja presente no contexto. A regra [`Int`], é bem direta ao ponto. Em um contexto Γ, um literal inteiro terá um tipo `int`. Por exemplo, se $x\colon \forall\vec{\alpha}.\tau \in \Gamma$, então $\Gamma \vdash x\colon \tau$ (após renomeação adequada das variáveis de tipo).

As continuações, como discutido anteriormente na Seção 2.1.1, representam fluxos de controle que não retornam valores. Uma vez que a continuação pode ser interpretada como o próximo passo de uma computação, e a computação se dá por contradições, a continuação em si não possui um tipo, ela representa um absurdo. Então, pode se dizer que a continuação é uma testemunha de que aquilo é um absurdo.

A regra [`Jump`] portanto, diz que sob um contexto Γ, se $k\colon \neg(\tau_1, \ldots, \tau_n)$ com n argumentos e cada argumento a_i tiver um tipo correspondente τ_i, então $k(\vec{a})$ é válido, ou seja, o salto k com os

argumentos $\vec{a}$ é testemunha de uma contradição. De modo semelhante para o [Bind], as premissas c e b onde c está sob o contexto $\{ \Gamma \cup \{ \vec{x}\colon \vec{\tau} \} \}$, e b sob o contexto $\{ \Gamma \cup \{ k\colon \overline{\Gamma}(\neg \vec{\tau}) \} \}$, são testemunhas de que o comando `let` $k(\vec{x}) = c$ `in` b é uma contradição. Assim como o 'let' introduz o polimorfismo no sistema Hindley-Milner, melhor detalhado na Seção 2.4, a generalização $\overline{\Gamma}(\neg \vec{\tau})$ presente na premissa do [Bind] quantifica as variáveis livres de $\vec{\tau}$ em Γ, estendendo o polimorfismo também ao CPS.

O algoritmo de inferência de tipos segue o mesmo esquema de Hindley-Milner (algoritmo W) adaptado ao CPS. Assim como o W, este faz uso do unificador mais geral, retornando sempre que existir o tipo mais genérico das expressões pertencentes a este sistema. Abaixo, tem-se sua definição:

$\boxed{\Gamma \vdash_W a\colon \tau}$

$$\frac{x\colon \sigma \in \Gamma \qquad \tau = inst(\sigma)}{\Gamma \vdash_W x\colon \tau}$$

$$\frac{}{\Gamma \vdash_W n\colon \texttt{int}}$$

$\boxed{\Gamma \vdash_W b \Rightarrow S}$

$$\frac{\Gamma \vdash_W k\colon \tau_1 \qquad \Gamma \vdash_W \vec{a}\colon \vec{\tau_2} \qquad S = mgu(\tau_1, \neg\vec{\tau_2})}{\Gamma \vdash_W k(\vec{a}) \Rightarrow S}$$

$$\frac{\vec{\tau} = \overrightarrow{newvar} \qquad \Gamma, \vec{x}\colon \vec{\tau} \vdash_W c \Rightarrow S_1 \qquad \sigma = \overline{S_1\Gamma}(S_1 \neg \vec{\tau}) \qquad S_1\Gamma, k\colon \sigma \vdash_W b \Rightarrow S_2}{\Gamma \vdash_W \texttt{let}\ k(\vec{x}) = c\ \texttt{in}\ b \Rightarrow S_2 \circ S_1}$$

Para as regras de inferência dos átomos, tal qual o sistema de tipos definido anteriormente, o algoritmo com uma variável x de politipo σ pertencente ao contexto Γ retornará um monotipo τ sob o mesmo contexto onde τ será a instanciação deste tipo σ. Para o tipo numérico, não são necessárias premissas, algoritmo simplesmente devolve o tipo `int`.

$$\frac{a\colon \forall\alpha.\alpha \in \Gamma \qquad \alpha = inst(\forall\alpha.\alpha)}{\Gamma \vdash_W x\colon \alpha}$$

Por exemplo, esteja a variável a com tipo $\forall\alpha.\alpha$ no contexto, ou seja, $a\colon \forall\alpha.\alpha \in \Gamma$. O algoritmo então inferirá, que a variável a terá tipo α, após as devidas normalizações (redução-α).

Os comandos serão inferidos a partir de substituições, onde o algoritmo as retornará representando o absurdo para qual esses testemunham. Para o [Jump], partindo das premissas onde sob um contexto Γ, a chamada k terá monotipo τ_1, os n argumentos em $\vec{a}$ terão n monotipos τ_2, e ainda, S é a unificação mais geral entre τ_1 e $\neg\vec{\tau_2}$, o algoritmo irá então retornar esta substituição S para o salto $k(\vec{a})$.

$$\frac{\Gamma \vdash_W k\colon \alpha \qquad \Gamma \vdash_W x\colon \beta \qquad S = mgu(\alpha, \neg\beta)}{\Gamma \vdash_W k(x) \Rightarrow \{\ \alpha \mapsto \neg\beta\ \}}$$

Por exemplo, em determinado contexto Γ, seja a chamada k com tipo α, ou seja, $\Gamma \vdash_W k\colon \alpha$, e ainda sob o mesmo contexto, o argumento x com tipo β, ou seja, $\Gamma \vdash_W x\colon \beta$. A partir da unificação mais geral entre α e $\neg\beta$ é obtida a substituição S, ou seja, $S = mgu(\alpha, \neg\beta)$. O algoritmo então, irá inferir que a substituição para que o salto represente uma contradição é $\{\alpha \mapsto \neg\beta\}$.

Para a regra que garante o polimorfismo do sistema, o [Bind], a chamada k recebe $\vec{x}$ argumentos, onde estes terão $\vec{\tau}$ tipos como sendo variáveis de tipo, ou seja, $\vec{\tau} = \overrightarrow{newvar}$. O c, por se tratar de um comando, será uma substituição S_1, onde recursivamente será inferida com o contexto inicial Γ unido com os $\vec{x}$ argumentos tipados com suas $\vec{\tau}$ variáveis de tipo frescas, ou seja, $\{\ \Gamma \cup \{\ \vec{x}\colon \vec{\tau}\ \}\ \} \vdash_W c \Rightarrow S_1$. Um ponto de atenção é necessário na função de generalização $\sigma = \overline{S_1\Gamma}(S_1 \neg \vec{\tau})$. A subsituição S_1 aplicada no contexto garante que este esteja atualizado com a descoberta do comando c na premissa anterior. Como as continuações não retornam e sim somente passam o resultado da computação adiante, é necessário também que S_1 seja aplicado no tipo do argumento $S_1 \neg \vec{\tau}$, para garantir que a substituição obtida no comando anterior seja utilizada nos tipos. De maneira semelhante ao primeiro comando, o comando b é inferido recursivamente com S_1 aplicado no contexto unido com o salto k tendo o tipo polimórfico σ produzido na premissa anterior, sendo atribuido a esta inferência a substituição S_2, ou seja, $\{\ S_1\Gamma \cup \{\ k\colon \sigma\ \}\ \} \Rightarrow S_2$. O algoritmo portanto, para o comando `let` $k(\vec{x}) = c$ `in` b, irá produzir a substituição resultante da composição entre as substituições de b e c, ou seja, $S_2 \circ S_2$.

3.2 IMPLEMENTAÇÃO

Partindo para a parte prática do trabalho, os módulos e funções serão apresentados de maneira gradual, de modo a facilitar o entendimento do fluxo inteiro do programa. Vale destacar aqui que é feito também a implementação do sistema Hindley-Milner, porém, como este foi feito somente para poder usar o sistema de continuações de maneira mais cômoda, não será entrado em detalhes nos que dizem respeito à esse sistema de tipos. Inicialmente, na Subseção 3.2.1, será discutido sobre a maneira como foi representado o sistema de tipos. Em sequência, a Subseção 3.2.2 trará detalhes sobre a implementação das funções de tradução de cálculo lambda simplesmente tipado para CPS. Posteriormente, a Subseção 3.2.3 irá tratar da inferência em si, juntamente da verificação do tipo. Por fim, a geração de código abordada na Subseção 3.2.4 serve como uma maneira de se testar a tradução do código.

3.2.1 Tipos de Dados

Para representar os comandos, bem como os tipos do sistema, foram utilizados tipos de dados algébricos (ADTs, do inglês *algebraic data types*), disponíveis no Listing 3.1.

```
1 type Id = String
2
```

```
3  data Command
4    = Jump Id [Id]
5    | Bind Command Id [Id] Command
6
7  data CPSMonoType
8    = TVar Id
9    | TInt
10   | TNeg [CPSMonoType]
11
12 data CPSPolyType
13   = Forall [Id] CPSMonoType
14
15 type Context = Data.Map Id CPSPolyType
16 type Substitution = Data.Map Id CPSMonoType
```

Listing 3.1 – Definição dos tipos de dados

Para os comandos, dois construtores podem ser observados, o `Jump` e o `Bind`, sendo responsáveis por construir respectivamente os comandos de *Jump*, onde há um salto `Id` com `[Id]` parâmetros. Sendo assim, o salto $k(x)$ seria representado por este ADT: `Jump k [x]`. E o comando *Bind*, onde há outro `Command` definido recursivamente, a função `Id` com argumentos `[Id]` e por fim outro `Command` também definido recursivamente. O *bind* `let` $k(x) = k(x)$ `in` $k(x)$, portanto seria definido pelo seguinte ADT: `Bind (Jump k [x]) k [x] (Jump k [x])`.

Os tipos, foram representados com dois tipos algébricos diferentes, um para os monotipos `CPSMonoType`, e uma para os politipos `CPSPolyType`. As variáveis do monotipo são construídas a partir dos construtores `TVar Id` e `TInt`, onde no primeiro, o `Id` serve para obter a variável de tipo atribuída àquela variável, enquanto que as funções que não retornam são representadas as partir do construtor de negação `TNeg [CPSMonoType]` sendo os argumentos dela definidos recursivamente sobre si. O contexto por sua vez, é um tipo que utiliza o `Data.Map` disponível no pacote *containers* para mapear uma variável para um tipo polimórfico. As substituições são representadas utilizando o mesmo `Data.Map`, onde desta vez é mapeado uma variável de tipo para um monotipo.

3.2.2 Traduções

O processo de tradução, no contexto de linguagens é a etapa em que um código feito em uma linguagem é transcrito para outra, onde idealmente, semântica alguma deve ser perdida. Neste caso, afim de facilitar os testes, e ainda de se aproximar mais de um uso real, onde um código é escrito em cálculo lambda, este é traduzido para o cálculo de continuações. Esta implementação da tradução foi feita seguindo a definição apresentada em (TORRENS; ORCHARD; VASCONCELLOS, 2024).

$\boxed{[\![\, e \,]\!]_N = b}$ *Plotkin's call-by-name translation*

$$
\begin{array}{lcl}
[\![\, x \,]\!]_N & = & x\langle k \rangle \\
[\![\, \lambda x.\, e \,]\!]_N & = & k\langle v \rangle \; \{\; v\langle x,\, k \rangle = [\![\, e \,]\!]_N \;\} \\
[\![\, f\ e \,]\!]_N & = & [\![\, f \,]\!]_N \; \{\; k\langle f \rangle = f\langle v,\, k \rangle \; \{\; v\langle k \rangle = [\![\, e \,]\!]_N \;\} \;\} \\
[\![\, \texttt{let}\ x = f\ \texttt{in}\ e \,]\!]_N & = & [\![\, e \,]\!]_N \; \{\; x\langle k \rangle = [\![\, f \,]\!]_V \;\}
\end{array}
$$

$\boxed{[\![\, e \,]\!]_V = b}$ *Plotkin's call-by-value translation*

$$
\begin{array}{lcl}
[\![\, x \,]\!]_V & = & k\langle x \rangle \\
[\![\, \lambda x.\, e \,]\!]_V & = & k\langle v \rangle \; \{\; v\langle x,\, k \rangle = [\![\, e \,]\!]_V \;\} \\
[\![\, f\ e \,]\!]_V & = & [\![\, f \,]\!]_V \; \{\; k\langle f \rangle = [\![\, e \,]\!]_V \; \{\; k\langle v \rangle = f\langle v,\, k \rangle \;\} \;\} \\
[\![\, \texttt{let}\ x = f\ \texttt{in}\ e \,]\!]_V & = & [\![\, f \,]\!]_V \; \{\; k\langle x \rangle = [\![\, e \,]\!]_N \;\}
\end{array}
$$

Note que, duas traduções distintas com base na estratégia de avaliação são apresentadas para o cálculo-λ, a chamada por nome (CBN, do inglês *call-by-name*) representada por $[\![-]\!]_N$, que diz respeito ao cálculo-λ_n e a chamada por valor (CBV, do inglês *call-by-value*) representada por $[\![-]\!]_V$, que diz respeito ao cálculo-λ_v. Ainda, as variáveis f, v e k que não aparecem no termo de origem são consideradas frescas, onde as duas primeiras são imediatamente ligadas, enquanto que k é esperada que seja livre e por ser vinculada às traduções dos subtermos, um único k é necessário (TORRENS; ORCHARD; VASCONCELLOS, 2024).

```
initialCont :: Id
initialCont = "k"
```

Listing 3.2 – Continuação inicial

Para todas as computações, um contexto inicial precisa conter a continuação inicial. Este será o objeto a ter seu tipo inferido. Afim de praticidade, esta continuação será sempre a mesma, dada por 'k'.

```
callByName :: Expr -> Id -> FreshM Command
callByName (Var x) k = return $ Jump x [k]

callByName (Lam x e) k = do
  k' <- freshCont
  v <- freshVar
  let jump = Jump k [v]
  eCall <- callByName e k'
  return $ Bind jump v [x, k'] eCall

callByName (App f e) k = do
  fCont <- freshCont
  fVar <- freshVar
  argCont <- freshCont
```

```
15    argVar <- freshVar
16    fCall <- callByName f fCont
17    let appCmd = Jump fVar [argVar, k]
18    eCall <- callByName e argCont
19    let innerBind = Bind appCmd argVar [argCont] eCall
20    return $ Bind fCall fCont [fVar] innerBind
21
22  callByName (Let x b a) k = do
23    k' <- freshCont
24    bCall <- callByName b k'
25    aCall <- callByName a k
26    return $ Bind aCall x [k'] bCall
27
28
29  cbnExprTrans :: Expr -> Command
30  cbnExprTrans expr = evalState (callByName expr initialCont) (0, 0)
```

Listing 3.3 – Tradução das expressões para CBN

Neste código, são apresentadas as funções responsáveis para traduzir as expressões para CBN. O ponto de partida desta computação será a função `cbnExprTrans :: Expr → Command`, onde ela irá receber a expressão em cálculo-λ e retornará o comando traduzido. Sua responsabilidade é criar as mônadas de estado que farão o controle do índice das variáveis frescas necessárias e chamar as funções de tradução passando a continuação inicial.

Tomando como exemplo a função identidade em cálculo-λ ($\lambda x.\ x$), é possível perceber como os termos crescem em CPS. Isto torna o desenvolvimento diretamente neste cálculo não apropriado, mas ainda, nota-se que a implementação da função de tradução é bastante direta em relação a sua definição formal. Os únicos pontos de atenção são em relação à geração das variáveis frescas, mas que como foi dito anteriormente, não eram completamente necessários, visto que eles são ligados imediatamente. Ao traduzir então a função, tem-se que o equivalente em CPS é:

```
1  let v0(x, k0) =
2      x(k0)
3  in
4      k(v0)
```

Listing 3.4 – Tradução da função identidade em CBN

Este comportamento fica ainda mais visível quando uma função um pouco maior é traduzida, por exemplo o numeral de Church dois ($\lambda f.\ \lambda x.\ f\ (f\ x)$). Sua tradução portanto é dada a seguir:

```
1  let v0(f, k0) =
2      let v1(x, k1) =
```

```
        let k2(v2) =
            let v3(k3) =
                let k4(v4) =
                    let v5(k5) =
                        x(k5)
                    in
                        v4(v5, k3)
                in
                    f(k4)
            in
                v2(v3, k1)
        in
            f(k2)
    in
        k0(v1)
in
    k(v0)
```

Listing 3.5 – Tradução do numeral de Church “2” em CBN

De maneira semelhante, foram feitas as mesmas funções utilizadas no *call-by-name*, porém adaptadas para o CBV, respeitando as diferenças presentes na definicão formal da função.

```
callByValue :: Expr -> Id -> FreshM Command
callByValue (Var x) k = return $ Jump k [x]

callByValue (Lam x e) k = do
  k' <- freshCont
  v <- freshVar
  let bindBody = Jump k [v]
  body <- callByValue e k'
  return $ Bind bindBody v [x, k'] body

callByValue (App f e) k = do
  fCont <- freshCont
  fVar <- freshVar
  argCont <- freshCont
  argVar <- freshVar
  fCall <- callByValue f fCont
  eCall <- callByValue e argCont
  let appCmd = Jump fVar [argVar, k]
  let argBind = Bind eCall argCont [argVar] appCmd
  return $ Bind fCall fCont [fVar] argBind
```

```
21
22 callByValue (Let x b a) k = do
23   k' <- freshCont
24   bCall <- callByValue b k'
25   aCall <- callByValue a k
26   return $ Bind bCall k' [x] aCall
27
28
29 cbvExprTrans :: Expr -> Command
30 cbvExprTrans expr = evalState (callByValue expr initialCont) (0, 0)
```

Listing 3.6 – Tradução das expressões para CBV

A partir dessas diferenças nas definições, nota-se também particularidades nas traduções destas, por exemplo ao traduzir a mesma função identidade, em CBV, é obtido o seguinte resultado:

```
1 let v0(x, k0) =
2     k0(x)
3 in
4     k(v0)
```

Listing 3.7 – Tradução da função identidade em CBV

Neste exemplo da função identidade, pouca diferença entre as duas estratégias de avaliação pode ser notada. Isso se deve não ao tamanho da expressão, e sim dos elementos desta. Neste caso, há somente uma abstração lambda com uma variável. A seguir, é exibido novamente o numeral de Church dois, porém para o CBV, onde mais diferenças podem ser observadas. O motivo disto é os elementos da função, que diferentemente da identidade, conta com mais construtores para representá-la:

```
1  let v0(f, k0) =
2      let v1(x, k1) =
3          let k2(v2) =
4              let k3(v3) =
5                  v2(v3, k1)
6              in
7                  let k4(v4) =
8                      let k5(v5) =
9                          v4(v5, k3)
10                     in
11                         k5(x)
12                 in
13                     k4(f)
14         in
15             k2(f)
```

```
16      in
17          k0(v1)
18  in
19      k(v0)
```

Listing 3.8 – Tradução do numeral de Church “2” em CBV

TODO: Citar aqui o problema do let no cbv.

Além da tradução de expressões, uma vez que o ambiente sendo trabalhado é tipado, é necessário que a tradução preserve a tipagem dos programas, Torrens, Orchard e Vasconcellos (2024) apresentam a tradução tipada do CPS, onde, para o tipo funcional $\mathtt{A} ::= \mathtt{A} \to \mathtt{A} \mid \mathtt{X}$, as seguintes traduções são definidas. Note que aqui também as funções são definidas diferentemente de acordo com estratégia de avaliação adotada:

$$
\begin{aligned}
[\![X]\!]_N &= \neg X \\
[\![A \to B]\!]_N &= \neg\neg(\neg[\![A]\!]_N, [\![B]\!]_N) \\
[\![\vec{x}\colon \vec{A} \vdash e\colon B]\!]_N &= \vec{x}\colon \overrightarrow{\neg[\![A]\!]_N}, k\colon [\![B]\!]_N \vdash [\![e]\!]_N
\end{aligned}
$$

$$
\begin{aligned}
[\![X]\!]_V &= X \\
[\![A \to B]\!]_V &= \neg([\![A]\!]_V, \neg[\![B]\!]_V) \\
[\![\vec{x}\colon \vec{A} \vdash e\colon B]\!]_V &= \vec{x}\colon \overrightarrow{[\![A]\!]_V}, k\colon \neg[\![B]\!]_V \vdash [\![e]\!]_V
\end{aligned}
$$

Esta função de tradução tem como propósito, mostrar que se o termo lambda é tipado, então o termo traduzido para CPS também é tipado. É importante notar que isto se aplica somente para os tipos simples, como definido no tipo funcional a partir do tipo primitivo $\mathtt{X}$ e do *arrow type* $\mathtt{A} \to \mathtt{A}$. O sistema de tipos proposto neste trabalho, entretanto, é polimórfico, ou seja, aqui há a adição de variáveis de tipos quantificadas. Sendo assim, esta função de tradução, se aplicando neste caso de uso, não necessariamente retornará o tipo mais geral de uma expressão, mas sempre um subtipo deste.

Mostrado anteriormente no Listing 3.4, a expressão CPS resultante da tradução da função identidade em CBN difere da mesma traduzida em CBV, presente no Listing 3.7. O mesmo pode ser observado ao traduzir a função identidade tipada. Por exemplo, ao executar a função para a identidade em CBN, o tipo obtido é $\neg\neg(\neg\neg\alpha, \neg\alpha)$. Já em CBV, para a mesma função, tem-se que o tipo traduzido é $(\alpha, \neg\alpha)$.

```
1  cbnTypeTranslation :: LambdaMonoType -> CPSPolyType
2  cbnTypeTranslation st =
3    let (transMono, freeVars) = runState (cbnTrans st) S.empty
4    in Forall (S.toList freeVars) transMono
5    where
6      cbnTrans :: LambdaMonoType -> State (S.Set String) CPSMonoType
7      cbnTrans = \case
8        Lambda.Typing.TVar varId -> do
```

```
9           modify (S.insert varId)
10          return $ TNeg [CPS.Typing.TVar varId]
11        TArr a b -> do
12          a' <- cbnTrans a
13          b' <- cbnTrans b
14          return $ TNeg [TNeg [TNeg [a'], b']]
```

Listing 3.9 – Tradução dos tipos para CBN

Aqui, a função responsável por traduzir um tipo utilizando a estratégia *call-by-name*, a função `cbvTypeTranslation` :: `LambdaMonoType` → `CPSPolyType` irá receber um tipo simples no cálculo-λ simplesmente tipado, e retornar um tipo polimórfico em CPS. Perceba que na definição, o retorno era um tipo simples em CPS, essa diferença é justificada ao se observar o corpo desta função, onde há a definição da função `cbvTrans`. Esta é quem efetivamente faz a computação, pois é nela que acontece o casamento de padrões para determinar o tipo sendo traduzido. Ainda, esta função retorna um tipo polimórfico pelo fato de que em momento posterior a essa tradução, a subtipagem do tipo traduzido e do tipo inferido precisa ser verificada.

```
1  cbvTypeTranslation :: LambdaMonoType -> CPSPolyType
2  cbvTypeTranslation st =
3    let (transMono, freeVars) = runState (cbvTrans st) S.empty
4        finalMono = TNeg [transMono]
5    in Forall (S.toList freeVars) finalMono
6    where
7      cbvTrans :: LambdaMonoType -> State (S.Set String) CPSMonoType
8      cbvTrans = \case
9        Lambda.Typing.TVar varId -> do
10         modify (S.insert varId)
11         return $ CPS.Typing.TVar varId
12       TArr a b -> do
13         a' <- cbvTrans a
14         b' <- cbvTrans b
15         return $ TNeg [a', TNeg [b']]
```

Listing 3.10 – Tradução dos tipos para CBV

Para que essas duas funções de tradução de tipos (CBN e CBV) sejam usadas do mesmo modo, elas seguem a messma assinatura. Seus comportamentos são o mesmo, a diferir somente nas diferenças das definições da função, isto é, como é feita a tradução.

3.2.3 Inferência

A inferência de tipos é a etapa onde, em uma linguagem onde tipos são presentes, um termo tem seu tipado inferido sem anotação prévia. Isto é, sem explicitar o tipo de um termo, este

tem seu tipo deduzido. Em um ambiente tipado com polimorfismo, a maior utilidade do inferidor de tipos é que este seja sempre o mais geral possível, tal que possa ser especializado para cada uso. O algoritmo de inferência proposto para este sistema de tipos é capaz de inferir o tipo mais geral possível, conforme explicado mais detalhadamente na Seção 3.1. Seu desenvolvimento foi bem direto ao ponto como pode ser visto abaixo no Listing 3.11.

```
inferAtom :: Context -> Id -> TI CPSMonoType
inferAtom ctx x =
  case readMaybe x :: Maybe Integer of
    Just _  -> return TInt
    Nothing -> case Map.lookup x ctx of
                 Just poly -> do
                   t <- instantiate poly
                   return t
                 Nothing -> throwError $ UnboundVariable x ctx


inferCommand :: Context -> Command -> TI Substitution
inferCommand ctx (Jump k xs) = do
  t1 <- inferAtom ctx k
  t2 <- mapM (inferAtom ctx) xs
  s <- mgu t1 (TNeg t2)
  return s

inferCommand ctx (Bind b y ys c) = do
  paramTypes <- mapM (const freshTVar) ys
  let ctxParams = extendContextWithParams ctx ys paramTypes
  s1 <- inferCommand ctxParams c
  let contType = applySubst s1 (TNeg paramTypes)
  let ctxSubst = applySubstToContext s1 ctx
  let sigma = generalize ctxSubst contType
  let ctx' = Map.insert y sigma ctxSubst
  s2 <- inferCommand ctx' b
  return (composeSubst s2 s1)


inferWithCtx :: Command -> TI CPSPolyType
inferWithCtx cmd = do
  initialType <- freshTVar
  let ctx = Map.singleton initialCont (Forall [] initialType)

  subst <- inferCommand ctx cmd
```

```
37  let ctx' = applySubstToContext subst ctx
38
39  case Map.lookup initialCont ctx' of
40    Just (Forall _ monoType) ->
41      let gen = generalize (Map.delete initialCont ctx') monoType
42          normalized = normalizePolyType gen
43      in return normalized
44    Nothing -> throwError (UnboundVariable initialCont ctx)
```

Listing 3.11 – Função principal de Inferência

Muitas das funções necessárias para a inferência do CPS são iguais as do Hindley-Milner, por não ser o foco deste trabalho, explicações sobre estas serão omitidos. Desta forma, o processamento para se inferir o tipo das continuações foi dividido em três principais funções.

A função `inferAtom :: Context → Id → TI CPSMonoType`, como seu nome e assinatura indica, é a função responsável por inferir os átomos do termo, isto é, buscar as variáveis do contexto e retorná-las caso sejam literais, instanciá-las se forem variáveis polimórficas, ou então retornar erro caso esta não esteja presente. Ao executar a função passando o contexto $\{ x: \forall\alpha.\alpha \}$ e a variável x, a função irá instanciar uma nova variável de tipo β (se esta for a próxima ainda não utilizada) e retornar este monotipo. Ou ainda, caso a função seja chamada com o contexto $\{ x: \forall\alpha.\alpha \}$ e a variável k, um erro `UnboundVariable` "k" $\{$ x: $\forall\alpha.\alpha$ $\}$ será exibido.

Já a função `inferCommand :: Context → Command → TI Substitution`, é quem, a partir do contexto e comando, irá retornar a substituição que indica o tipo da continuação. Tal qual a função anterior, nenhuma dificuldade grande foi encontrada aqui. A inferência do salto foi a mais simples delas, onde cada linha da função se refere às premissas na definição, retornando então a unificação mais geral entre τ_1 e $\vec{\tau_2}$. Já para o *binding*, algumas funções auxiliares foram necessárias principalmente para manipular o contexto ao extender este em diferentes momentos. Além destas, funções para a generalização e por fim para a composição das subsituições também foram precisas.

O ponto de partida da inferência, `inferWithCtx :: Command → TI CPSPolyType`, além de definir o contexto inicial com a continuação inicial k, recebendo um tipo α qualquer, esta função aplicará a substituição obtida na inferência e a aplicará no contexto, de modo que o contexto final esteja atualizado com o tipo inferido da continuacão k. Após isto, é feita a normalização do politipo da substituição, isto é, limpar as variáveis de tipo utilizadas durante o processo que não são mais ncessárias, afim de promover consistência e facilitar o entendimento. Por exemplo, no caso onde o politipo final seja $\forall\delta.\delta$, o retornado seria $\forall\alpha.\alpha$.

Uma etapa que não está diretamente relacionada com a inferência, e sim com os tipos em si, é a verificação da subtipagem do tipo traduzido e do tipo inferido. Isto é, uma verificação se o tipo traduzido é um subtipo do tipo inferido, indicando diretamente se o algoritmo de inferência foi implementado corretamente.

```haskell
isSubtypeOfPoly :: CPSPolyType -> CPSPolyType ->
   Either TypeError (Maybe Substitution)
isSubtypeOfPoly (Forall vars1 t1) (Forall vars2 t2) =
  runTI $ do
    fv1 <- replicateM (length vars1) freshTVar
    let s1 = Map.fromList (zip vars1 fv1)
    fv2 <- replicateM (length vars2) freshTVar
    let s2 = Map.fromList (zip vars2 fv2)
    pure $ isSubtypeOf (applySubst s1 t1) (applySubst s2 t2)


isSubtypeOf :: CPSMonoType -> CPSMonoType -> Maybe Substitution
isSubtypeOf t1 t2 = match t1 t2 Map.empty
  where
    match :: CPSMonoType -> CPSMonoType -> Substitution ->
      Maybe Substitution
    match (TVar a) t subst =
        case Map.lookup a subst of
            Just tExisting ->
                if tExisting == t
                    then Just subst
                    else Nothing
            Nothing ->
                if occursCheck a t
                    then Nothing
                    else Just (Map.insert a t subst)
    match TInt TInt subst = Just subst
    match (TNeg ts1) (TNeg ts2) subst
        | length ts1 == length ts2 =
          foldM
            (\s (t1', t2') -> match t1' t2' s)
            subst
            (zip ts1 ts2)
        | otherwise = Nothing
    match _ _ _ = Nothing
```

Listing 3.12 – Verificação de Subtipagem

TODO: EXPLICAÇÃO MAIS ALTO NÍVEL DESTAS FUNÇÕES Se retornar uma substituição, deu boa; se não, deu pau

3.2.4 Geração de Código

Uma vez que nenhuma prova a respeito da validade do sistema de tipos proposto foi desenvolvida, uma quantidade considerável de testes foi feita. Ainda, afim de propor testes mais confiáveis para o código, especificamente para a tradução, com base no teorema de Plotkin que diz que **TODO:** EXPLICAR TEOREMA QUE COMPUTA EM CPS. Para aqueles programas que computem um número, seguindo a tipagem dos numerais de Church, este é, $\forall\alpha(\alpha \to \alpha) \to \alpha \to \alpha$, foi implementado um mecanismo de geração de código que gera código em CPS e faz essa computação a partir deste.

```
-- Auto-generated by Main.hs
-- For input {{ Input file name }}
-- This code works only for Church Encoding expressions
-- It will compute the ...
-- ... generated CPS translated expression Church enconding

------------------------------------------------------------

cbn k = -- {{ CBN Translated Expression }}

cbv k = -- {{ CBV Translated Expressiond }}

------------------------------------------------------------

inc_cbv :: (Int, Int -> r) -> r
inc_cbv (n, k) = k (1 + n)

test_cbv =
    cbv (\f -> f (inc_cbv, \x -> x (0, id)))

thunk :: a -> (a -> r) -> r
thunk x k = k x

inc_cbn :: (Int -> r, Int) -> r
inc_cbn (k, n) =
    inc_cbv (n, k)

test_cbn =
    cbn (\f -> f (thunk inc_cbn, \x -> x (id, 0)))

main :: IO ()
main = do
```

```
33     print $ (test_cbn, test_cbv)
```

Listing 3.13 – Geração de código para computação de numerais de Church

O código Haskell gerado pode ser dividido em três partes principais, o cabeçalho informativo, que explicita o programa em cálculo-λ de entrada para ter gerado aquele programa em CPS. Em seguida, há a definição das funções `cbn` e `cbv`, ou seja, os programas correspondente em CPS para as duas traduções daquela entrada. Por fim, a última parte é responsável pela computação do numeral de Church, as funções definidas irão calcular o número representado pelas expressões em CPS e retornar por fim uma tupla contendo o resultado do calculado pelo *call-by-name* e *call-by-value* respectivamente.

Ao se traduzir o numeral de Church 0 representado em cálculo-λ por $\lambda f.\lambda x.\ x$, para CBN e CBV, tem-se:

```
let v0(f, k0) =
  let v1(x, k1) =
    x(k1)
  in
    k0(v1)
in
  k(v0)
```

Listing 3.14 – Tradução do numeral de Church “0” em CBN

```
let v0(f, k0) =
  let v1(x, k1) =
    k1(x)
  in
    k0(v1)
in
  k(v0)
```

Listing 3.15 – Tradução do numeral de Church “0” em CBV

Desta forma, o código gerado ao se traduzir esta função, é ilustrado a seguir:

```
-- Auto-generated by Main.hs
-- For input church-zero
-- This code works only for Church Encoding expressions
-- It will compute the ...
-- ... generated CPS translated expression Church enconding

------------------------------------------------------------

cbn k = let v0(f, k0) = let v1(x, k1) = x(k1) in k0(v1) in k(v0)
```

```
10
11 cbv k = let v0(f, k0) = let v1(x, k1) = k1(x) in k0(v1) in k(v0)
12
13 ------------------------------------------------------------
14
15 inc_cbv :: (Int, Int -> r) -> r
16 inc_cbv (n, k) = k (1 + n)
17
18 test_cbv =
19     cbv (\f -> f (inc_cbv, \x -> x (0, id)))
20
21 thunk :: a -> (a -> r) -> r
22 thunk x k = k x
23
24 inc_cbn :: (Int -> r, Int) -> r
25 inc_cbn (k, n) =
26     inc_cbv (n, k)
27
28 test_cbn =
29     cbn (\f -> f (thunk inc_cbn, \x -> x (id, 0)))
30
31 main :: IO ()
32 main = do
33   print $ (test_cbn, test_cbv)
```

Listing 3.16 – Código gerado ao traduzir o numeral de Church “0”

Ao executar o código e chamar a função `main` deste programa, o resultado obtido é justamente a computação do numeral para as duas traduções, ou seja, $(0, 0)$.

3.2.5 Fluxo Principal

O fluxo completo de execução do programa principal contempla todas as funções apresentadas nesta seção, com a adição de funções auxiliares. Essas são aplicadas em sequência, de modo a realizar uma série de ações.

Inicialmente, é passado o caminho de um arquivo contendo um programa em cálculo-λ com adição do ‘let’. O conteúdo então é processado pelo *parser* e representado pelos tipos de dados algébricos para o cálculo lambda. Uma vez que o programa já está sendo representado pelos ADTs, e ainda tem seu tipo inferido, é possível iniciar o processamento descrito pelas funções apresentadas. A primeira delas é a tradução para CPS tanto em *call-by-name* quanto em *call-by-value*, as exibindo logo em seguida. Com as traduções da expressão feitas o código Haskell já pode ser gerado, salvando assim no diretório `output` com mesmo nome do arquivo de

entrada. Como passo posterior, tem-se a tradução dos tipos para ambas as estratégias de avaliação. Os passos finais envolvem a inferência de ambas as traduções, juntamente da verificação de subtipagem, onde esta informará se o tipo traduzido é um subtipo do inferido.

```
$ cabal run
Input file path:
> input/church-zero.in
Expression:
λf. λx. x
Type:
α → β → β
Call-by-Name Translation:
Command:
let v0(f, k0) =
  let v1(x, k1) =
    x(k1)
  in
    k0(v1)
in
  k(v0)
Expected Continuation Type:
∀α,β. ¬¬(¬¬α, ¬¬(¬¬β, ¬β))
Inferred Continuation Type:
∀α,β. ¬¬(α, ¬¬(¬β, β))
Do the types match?
Yes

Call-by-Value Translation:
Command:
let v0(f, k0) =
  let v1(x, k1) =
    k1(x)
  in
    k0(v1)
in
  k(v0)
Expected Continuation Type:
∀α,β. ¬¬(α, ¬¬(β, ¬β))
Inferred Continuation Type:
∀α,β. ¬¬(α, ¬¬(β, ¬β))
Do the types match?
Yes
```

```
CPS expression saved in output/church-zero.hs
```

Listing 3.17 – Execução do programa principal

Ao executar o programa com o comando `cabal run`, passando também o arquivo de entrada `input/church-zero.in`, é processado e exibida todas as informações que foi citada anteriormente, inclusive a geração do código Haskell em `output/church-zero.hs`. É possível perceber que na saída do programa, é mostrado o tipo traduzido (na sequência da mensagem "*Expected Continuation Type:*") e o tipo inferido (que sucede a mensagem "*Inferred Continuation Type:*"). Logo em seguida, o questionamento "*Do the types match?*" é o trecho da saída que compete à subtipagem, retornando "*Yes*" caso esse seja um subtipo deste, o que indica uma inferência compatível com a tradução ou "*No*" caso a verificação falhe, indicando uma inferência incorreta.

```
$ ghc output/church-zero.hs
[1 of 2] Compiling Main             ( output/church-zero.hs, ... )
[2 of 2] Linking output/church-zero
$ ./output/church-zero
(0,0)
```

Listing 3.18 – Execução do programa gerado

Ainda, a compilação e execução do código Haskell gerado pode ser conferida no Listing 3.18 acima, ilustrando exatamente o comportamento detalhado anteriormente.

3.3 RESULTADOS

3.4 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 15 e 18.

APPEL, Andrew W; JIM, Trevor. Shrinking lambda expressions in linear time. **J. Funct. Prog.**, Cambridge University Press (CUP), v. 7, n. 5, p. 515–540, set. 1997. Citado 2 vezes nas páginas 19 e 31.

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

CHURCH, Alonzo. A formulation of the simple theory of types. **J. Symb. Log.**, Cambridge University Press (CUP), v. 5, n. 2, p. 56–68, 1940. Citado na página 23.

COOPER, Keith D; TORCZON, Linda. **Contruindo Compiladores**. 2th. ed. [S.l.]: Elsevier, 2014. Citado na página 15.

COQUAND, Thierry. Type Theory. In: ZALTA, Edward N.; NODELMAN, Uri (Ed.). **The Stanford Encyclopedia of Philosophy**. [S.l.]: Metaphysics Research Lab, Stanford University, 2022. Citado na página 21.

COQUAND, Thierry; HUET, Gérard. The calculus of constructions. **Information and Computation**, v. 76, n. 2, p. 95–120, 1988. Citado na página 24.

DAMAS, Luis. **Type assignment in programming languages**. Tese (Doutorado) — University of Edinburgh, 1984. Citado na página 24.

DAMAS, Luis; MILNER, Robin. **Principal type-schemes for functional programs**. Tese (Doutorado) — University of Edinburgh, Scotland, 1982. Citado 3 vezes nas páginas 24, 26 e 27.

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 19.

KENNEDY, Andrew. Compiling with continuations, continued. In: . [S.l.]: Association for Computing Machinery, 2007. Citado na página 12.

MILNER, Robin. A theory of type polymorphism in programming. In: **Journal of Computer and System Sciences**. [S.l.]: Elsevier, 1978. v. 17, n. 3, p. 348–375. Citado na página 24.

MORRISETT, Greg et al. From system f to typed assembly language. **ACM Trans. Program. Lang. Syst.**, v. 21, n. 3, p. 527–568, 1999. Citado na página 12.

MUCHNICK, Steven S. **Advanced Compiler Design and Implementation**. Oxford, England: Morgan Kaufmann, 1997. Citado na página 16.

PIERCE, Benjamin C. **Types and Programming Languages**. [S.l.]: The MIT Press, 2002. Citado 4 vezes nas páginas 21, 22, 23 e 24.

PLOTKIN, G.D. Call-by-name, call-by-value and the λ-calculus. **Theoretical Computer Science**, v. 1, n. 2, p. 125–159, 1975. Citado na página 12.

RIBEIRO, Rodrigo; CAMARÃO, Carlos. A mechanized textbook proof of a type unification algorithm. In: CORNÉLIO, Márcio; ROSCOE, Bill (Ed.). **Formal Methods: Foundations and Applications**. Cham: Springer International Publishing, 2016. p. 127–141. ISBN 978-3-319-29473-5. Citado na página 28.

SILVA, Rafael Castro Gonçalves. **Uma Certificação em Coq do Algoritmo W Monádico**. Dissertação (Mestrado) — UDESC, 2019. Citado 2 vezes nas páginas 26 e 29.

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

THIELECKE, Hayo. **Categorical Structure of Continuation Passing Style**. 1997. Citado 3 vezes nas páginas 12, 13 e 18.

TORRENS, Paulo; ORCHARD, Dominic; VASCONCELLOS, Cristiano. On the operational theory of the cps-calculus: Towards a theoretical foundation for irs. **Proc. ACM Program. Lang.**, 2024. Citado 3 vezes nas páginas 35, 36 e 40.

TORRENS, Paulo Henrique. **Um Cálculo de Continuações com Tipos Dependentes**. Dissertação (Mestrado) — UDESC, 2019. Citado na página 20.

VASCONCELLOS, Cristiano Damiani. **Inferência de Tipos com Suporte para Sobrecarga Baseada no Sistema CT**. Tese (Doutorado) — Universidade Federal de Minas Gerais, 2004. Citado na página 27.