Embora linguagens intermediárias sejam pervasivas em compiladores, elas são muitas vezes usadas em uma forma não-tipada, descartando assim importantes informações contidas na linguagem fonte do compilador, e impedindo que garantias sejam preservadas em fases posteriores do processo de compilação. Esse trabalho pretende apresentar um cálculo de continuações contendo tipos dependentes, que deve ser capaz de manter as propriedades lógicas de um programa durante as fases de otimização de um compilador, e uma discussão sobre quais necessidades existem para usá-lo como uma linguagem intermediária.

Orientador: Prof. Dr. Cristiano Damiani Vasconcellos

Joinville, 2019

ANO
2019

PAULO HENRIQUE TORRENS | UM CÁLCULO DE CONTINUAÇÕES COM TIPOS DEPENDENTES

UNIVERSIDADE DO ESTADO DE SANTA CATARINA – UDESC
CENTRO DE CIÊNCIAS TECNOLÓGICAS - CCT
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO APLICADA

DISSERTAÇÃO DE MESTRADO

UM CÁLCULO DE CONTINUAÇÕES COM TIPOS DEPENDENTES

PAULO HENRIQUE TORRENS

JOINVILLE, 2019

UNIVERSIDADE DO ESTADO DE SANTA CATARINA - UDESC
CENTRO DE CIÊNCIAS TECNOLÓGICAS - CCT
MESTRADO EM COMPUTAÇÃO APLICADA

PAULO H. TORRENS

UM CÁLCULO DE CONTINUAÇÕES COM TIPOS DEPENDENTES

JOINVILLE

2018

PAULO H. TORRENS

UM CÁLCULO DE CONTINUAÇÕES COM TIPOS DEPENDENTES

Dissertação submetida ao Programa de Pós-Graduação em Computação Aplicada do Centro de Ciências Tecnológicas da Universidade do Estado de Santa Catarina, para a obtenção do grau de Mestre em Computação Aplicada.

Orientador: Dr. Cristiano Damiani Vasconcellos

JOINVILLE

2018

Ficha catalográfica elaborada pelo programa de geração automática da

Biblioteca Setorial do CCT/UDESC,

com os dados fornecidos pelo(a) autor(a)

Torrens, Paulo Henrique
Um Cálculo de Continuações com Tipos Dependentes / Paulo Henrique Torrens. -- 2019.
94 p.

Orientador: Cristiano Damiani Vasconcellos
Dissertação (mestrado) -- Universidade do Estado de Santa Catarina, Centro de Ciências Tecnológicas, Programa de Pós-Graduação em Computação Aplicada, Joinville, 2019.

1. compiladores. 2. linguagens de programação. 3. linguagens intermediárias. 4. tipos dependentes. 5. tipos lineares. I. Vasconcellos, Cristiano Damiani . II. Universidade do Estado de Santa Catarina, Centro de Ciências Tecnológicas, Programa de Pós-Graduação em Computação Aplicada. III. Título.

Um Cálculo de Continuações com Tipos Dependentes

por

Paulo Henrique Torrens

Esta dissertação foi julgada adequada para obtenção do título de

Mestre em Computação Aplicada

Área de concentração em "Ciência da Computação",
e aprovada em sua forma final pelo

CURSO DE MESTRADO ACADÊMICO EM COMPUTAÇÃO APLICADA
DO CENTRO DE CIÊNCIAS TECNOLÓGICAS DA
UNIVERSIDADE DO ESTADO DE SANTA CATARINA.

Banca Examinadora:

Prof. Dr. Cristiano Damiani Vasconcellos
CCT/UDESC (Orientador/Presidente)

Profa. Dra. Karina Girardi Roggia
CCT/UDESC

Prof. Dr. Rodrigo Geraldo Ribeiro
UFOP

Joinville,SC, 19 de dezembro de 2018.

AGRADECIMENTOS

O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Código de Financiamento 001.

Em especial, agradeço ao meu orientador, Cristiano Damiani Vasconcellos, que lidou com meus constantes atrasos durante os últimos anos.

"*There is (gentle reader) nothing (the works of God only set apart) which so much beautifies and adorns the soul and mind of man as does knowledge of the good arts and sciences. Many arts there are which beautify the mind of man; but of all none do more garnish and beautify it than those arts which are called mathematical, [...].*"

John Dee

RESUMO

Embora linguagens intermediárias sejam pervasivas em compiladores, elas são muitas vezes usadas em uma forma não-tipada, descartando assim importantes informações contidas na linguagem fonte do compilador, e impedindo que garantias sejam preservadas em fases posteriores do processo de compilação. Esse trabalho pretende apresentar um cálculo de continuações contendo tipos dependentes, que deve ser capaz de manter as propriedades lógicas de um programa durante as fases de otimização de um compilador, e uma discussão sobre quais necessidades existem para usá-lo como uma linguagem intermediária.

Palavras-chaves: compiladores, linguagens de programação, linguagens intermediárias, tipos dependentes, tipos lineares

ABSTRACT

Although intermediate languages are pervasive in compilers, they are usually employed in an untyped setting, thus discarding important information present in the compiler's source language, and keeping a few guarantees from being preserved in later compilation steps. The current work aims to present a calculus of continuations with dependent types, which should be able to keep logical properties of a program through the optimization steps of a compiler, along with a discussion about what would be the needs to use it as an intermediate language.

Keywords: compilers, programming languages, intermediate languages, dependent types, linear types

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE SIGLAS E ABREVIATURAS

ANF forma normal administrativa (do inglês, *administrative normal form*)

CFG grafo de fluxo de controle (do inglês, *control flow graph*)

CoC cálculo de construções (do inglês, *calculus of constructions*)

CPS estilo de passagem de continuação (do inglês, *continuation-passing style*)

SSA atribuição única estática (do inglês, *static single assignment*)

SUMÁRIO

> "*A PL designer used to be able to design some syntax and semantics for their language, implement a compiler, and then call it a day. Today, however, languages are supported by sophisticated environments that, when designed together with languages, have the potential to significantly improve the programmer's overall experience.*" **(Sean McDirmid)**

1 INTRODUÇÃO

As áreas de teoria de linguagens de programação e compiladores se desenvolveram grandemente desde seu início na década de 60, e cada vez mais sistemas complexos são construídos para auxiliar e agilizar o desenvolvimento de *software* modernos. Os requisitos para um compilador moderno também se tornaram mais complexos, os quais necessitam ser além de simples caixas pretas capazes de gerar código de forma arbitrária.

O processo de desenvolvimento de um compilador (AHO et al., 2006, onde é melhor abordado) é um trabalho longo, e costuma ser dividido em diversas fases, como análise léxica, análise sintática, análise semântica, otimizações (e demais transformações desejáveis), e emissão de código. De interesse particular para este trabalho é a fase de otimização, a qual faz uso de uma linguagem intermediária, essencial para o uso em escala de um compilador.

Há muito tempo sabe-se que linguagens de programação convencionais, tais como são usadas por programadores humanos, não são adequadas para otimizações (FLANAGAN et al., 1993, por exemplo). Apesar de variarem grandemente quanto ao seu paradigma e seu estilo de escrita, linguagens de programação apresentam abstrações sobre detalhes úteis ao compilador; tal característica criou a necessidade de linguagens intermediárias: linguagens de programação projetadas para serem usadas por máquinas (mais especificamente, por compiladores), com uma semântica adequada para otimizações automáticas de programas.

Para compiladores de linguagens imperativas, a escolha popular de linguagem intermediária é a forma de atribuição única estática (SSA), representada dentro de um grafo de fluxo de controle (CFG) (CYTRON et al., 1991). Programas imperativos são notoriamente difíceis de se otimizar, em particular devido à falta de transparência referencial: não é possível, arbitrariamente, trocar variáveis por suas definições, visto que elas podem ser alteradas conforme a execução do programa, e cujas definições podem causar efeitos colaterais ao mesmo.

Programas em SSA transformam funções escritas na linguagem fonte[1] em blocos básicos, cada um contendo uma sequência de atribuições, seguidas por um

[1] Linguagem de entrada do compilador, na qual foi escrita o programa que está sendo compilado.

desvio de fluxo, tendo suas variáveis atribuídas uma única vez (fato responsável pelo nome SSA). Nota-se que, devido à sua formulação, CFG/SSA é considerado uma linguagem de programação funcional (APPEL, 1998).

Compiladores de linguagens funcionais, entretanto, apresentam uma linguagem intermediária baseada no cálculo lambda. Algumas linguagens intermediárias existem, mas, dentre elas, é sempre presente o conceito de continuações (KENNEDY, 2007). O cálculo lambda com CPS é uma escolha popular, tornando as continuações explícitas dentro do código, tendo sido alvo de vários estudos, tanto a favor (KENNEDY, 2007) quanto contra (MAURER et al., 2017) seu uso (e a favor de alternativas tidas como mais simples), e sendo fundamentalmente a base para compilação de linguagens funcionais (MORRISETT et al., 1999).

Continuações representam o resto de uma computação, isto é, o que fazer com o resultado de uma função ao invés de retorná-lo. Essa informação costuma ser implícita nas linguagens de programação convencionais, que fazem o uso de abstrações como uma pilha de chamada[2], mas é explícita em linguagens de baixo nível; continuações podem ser vistas, por exemplo, como o endereço de retorno em um programa *assembly*, que é passado explicitamente como um parâmetro adicional à função chamada (KENNEDY, 2007).

Tradicionalmente, compiladores são desenvolvidos para linguagens de programação particulares, e, tal qual, técnicas de compilação são desenvolvidas especificamente para paradigmas particulares, a fim de lidar com as abstrações necessárias para a implementação. Como tal, pesquisas nas áreas de otimizações costumam ser segmentadas; entretanto, se torna cada vez mais comum a criação de *software* escrito com mais de uma linguagem de programação, com implementações e ambientes distintos (AHMED, 2015).

Essa distinção, entretanto, não é uma necessidade: ambas as linguagens são equivalentes: é possível uma tradução entre o cálculo lambda em estilo de passagem de continuação e a SSA *form*, conforme apresentado em (KELSEY, 1995), onde os autores argumentam: “desenvolvedores de compiladores podem usar uma única representação e obter vantagens de ambos trabalhos feitos para otimizações intraprocedurais que foram desenvolvidos para SSA e trabalhos feitos para otimizações interprocedurais usando CPS” (em uma tradução livre). Embora essa relação tenha sido aproveitada em alguns projetos (FLUET; WEEKS, 2001), a segregação de técnicas de otimização ainda é comum.

Ambas as linguagens, SSA e CPS, entretanto, possuem uma similaridade pouco desejada: não é incomum que sejam representadas em uma forma sem tipa-

[2] Exceto nos dias antigos, quando FORTRAN ainda era escrito apenas em caixa alta.

gem nos compiladores convencionais (MORRISETT et al., 1999). Sistemas de tipos, tais como usados em linguagens de programação convencionais, representam propriedades do programa que, além de auxiliarem o programador e aumentarem a segurança do código ao restringir algumas classes de erro, podem ser usados para guiar otimizações e servir de evidência que tais otimizações são válidas (MORRISETT et al., 1999), e não é desejável que sejam descartadas rapidamente dentro de um compilador.

Dentre informações contidas em sistemas de tipos, cujos benefícios podem se estender para linguagens intermediárias, estão, por exemplo, as informações contidas em sistemas de tipos dependentes[3] e sistemas de tipos lineares. Nota-se também que o desejo de se garantir propriedades dentro de linguagens intermediárias não é uma novidade; trabalhos recentes como (ZHAO et al., 2012) começam a apresentar formalizações para linguagens intermediárias, com o principal objetivo de se garantir que otimizações introduzidas por um compilador, provadamente, não alterem o significado do código conforme foi escrito pelo programador.

Sistemas de tipos dependentes, presentes em linguagens como Idris e Agda, são notáveis por serem capazes de representar e garantir propriedades semânticas de um programa em seu sistema de tipos; por exemplo, é possível criar uma função de ordenação que, além de retornar uma lista ordenada, retorna uma prova de que a ordenação está correta, onde o compilador efetivamente também possui o papel de assistente de provas. Tipos dependentes foram usados, por exemplo, para a escrita do CompCert (LEROY, 2009), um compilador de C provadamente correto. Ao se apagar informações sobre tipos dependentes na linguagem intermediária, qualquer prova de comportamento é juntamente apagada e não há garantia de que as mesmas são preservadas através das transformações na linguagem intermediária.

Sistemas de tipos lineares, presentes em linguagens como Clean, funcionam como um cálculo de recursos para o programa. Eles permitem limitar o uso de variáveis, especificamente, garantir que uma variável seja usada uma única vez e podem ser usados, por exemplo, para impedir que se tente ler um arquivo já fechado. Ao se apagar informações sobre tipos lineares na linguagem intermediária, qualquer garantia de utilização de recursos é juntamente apagada: manipulações dentro do compilador através da linguagem intermediária podem, possivelmente por acidente, adicionar ou eliminar usos de uma variável linear.

Em particular, a perda de tipos é problemática na fase de *linking* de programas com unidades de compilação distintas: nesse ponto, com o código transformado

[3] Embora alguns sistemas com tipos dependentes façam uma distinção entre informações computacionalmente irrelevantes, que são propositadamente apagadas durante a compilação, e informações computacionalmente relevantes.

em uma linguagem intermediária, as características desejadas de outras partes do código (contidas no sistema de tipos original) são perdidas. Conforme (BOWMAN et al., 2018), investigar a compilação de linguagens com tipos dependentes é importante pois pode ser usada para garantir que as entradas para uma função sejam apenas válidas caso formem um contexto válido na linguagem fonte; esse trabalho motiva que o mesmo argumento é válido para tipos lineares.

O objetivo desse trabalho é apresentar os primeiros passos para a criação de uma linguagem intermediária capaz de representar programas fonte imperativos e funcionais, utilizando-se da equivalência entre SSA e CPS, e que preserve informações sobre seus tipos, permitindo que se garanta propriedades mais complexas de programas fonte ainda na etapa intermediária de otimização do compilador. A principal motivação de tal linguagem é a capacidade de reaproveitar otimizações projetadas por ambas escolas de pesquisa em relação a compiladores, e que o faça de uma forma segura quanto às propriedades dos programas fonte.

As contribuições desse trabalho são listadas a seguir:

- Um cálculo de continuações adequado para o uso como uma linguagem intermediária, apresentando tipos dependentes e, de forma limitada, tipos lineares. De certa forma, o objetivo do cálculo é responder a pergunta: “como se pareceria uma lógica em estilo de passagem de continuações?”.
- Uma implementação de um algoritmo de checagem de tipos para o cálculo de continuações apresentado, a fim de explorar empiricamente a proposta.
- Uma discussão sobre as necessidades e futuros passos para a formalização de uma linguagem intermediária capaz de representar programas reais, tanto de linguagens fonte imperativas quanto funcionais.

Os seguintes capítulos estão organizados a seguir. O Capítulo 2 apresenta as fundamentações teóricas necessárias para esse trabalho. O Capítulo 3 apresenta o desenvolvimento de uma linguagem intermediária atendendo aos pontos iniciais desse trabalho, explorando um protótipo de implementação na Seção 3.2. O Capítulo 4 apresenta uma discussão sobre a linguagem proposta. Finalmente, o Capítulo 5 conclui.

"*Trying to outsmart a compiler defeats much of the purpose of using one.*" **(Brian Kernighan)**

2 FUNDAMENTAÇÃO

Este capítulo apresentará as devidas fundamentações teóricas necessárias para o desenvolvimento desse trabalho. Dentre elas estão presentes a classificação de linguagens de programação, noções básicas sobre sistemas de tipos e teoria de tipos, em especial tipos dependentes e tipos lineares, e linguagens intermediárias.

2.1 LINGUAGENS DE PROGRAMAÇÃO

Para os fins desse trabalho, é necessário classificar linguagens de programação em determinados grupos. Das principais formas de classificar linguagens de programação encontradas na literatura, iremos usar principalmente classificações por seu nível de abstração e seu paradigma. Também é possível classificá-las como linguagens de programação de sistemas, e de acordo com seu sistema de tipos (conforme será abordado na Seção 2.1.4).

2.1.1 Nível de Abstração

Linguagens podem ser divididas conforme seu nível de abstração em relação à máquina que irá executá-la. Dizemos que linguagens de baixo nível possuem pouca ou nenhuma abstração, enquanto linguagens de alto nível possuem um nível grande de abstração.

Os exemplos canônicos de linguagens de baixo nível são as linguagens de montagem (do inglês, *assembly language*). Tais linguagens não possuem abstrações (salvo *macros*) em relação a uma arquitetura de *hardware* para a qual foi projetada, contendo apenas uma lista de instruções (*opcodes*) tais quais o processador pode executar, dando acesso aos seus recursos físicos. Além de variarem pela arquitetura, linguagens de montagem variam pela sintaxe específica do montador (do inglês, *assembler*) usado. Como tal, linguagens de montagem específicas podem ser descritas como uma 2-tupla no formato (*arch*, *assm*), onde *arch* é a arquitetura e *assm* é o montador. Por exemplo, (x86, GAS) é a linguagem de montagem para processadores Intel x86, com a sintaxe do montador GAS (GNU Assembler). A consequência direta da sua relação a arquiteturas de *hardware* é que linguagens de montagem são inerentemente não-portáveis.

Em contrapartida, linguagens de alto nível englobam um número muito grande de possíveis abstrações que devem ser traduzidas para uma linguagem de baixo ní-

vel por um compilador[4]. Tais abstrações podem ser simples como controles de fluxo estruturados (condicionais, *loops*, exceções), e podem ser complexas como gerenciamento automático de memória, controle de paralelismo, e até mesmo operações em um banco de dados.

Embora popularmente a linguagem C seja considerada de “baixo nível”, não a consideraremos como tal nesse trabalho. Apesar de possuir características normalmente associadas a linguagens de baixo nível, como a aritmética de ponteiros e saltos incondicionais (o infame `goto`), isso é apenas válido para *hardwares* específicos. O padrão ISO C foi projetado para uma máquina abstrata (CHEN, 2013), não para uma real, fato que leva programadores a fazerem suposições erradas em relação ao padrão da linguagem, escrevendo código não-conformante e inseguro (FEATHER, 2004). O desejo por parte de desenvolvedores de *hardware* e de compiladores de se manter tais suposições (i.e., permitir que programadores C acreditem que estão lidando com algo “próximo ao metal”) é atribuído como causa de problemas de segurança recentemente descobertos, como o Spectre e o Meltdown (CHISNALL, 2018).

Tal como abordado em (ERTL, 2015), usaremos doravante o nome C para nos referirmos à linguagem dos programas em C conformantes (termo retirado do padrão da linguagem, *conforming program*), que faz uso de recursos específicos de algum compilador, e usaremos o nome “C” para nos referirmos à linguagem dos programas em C estritamente conformantes (conforme o padrão, *strictly conforming program*), que utiliza apenas os recursos descritos e garantidos pelo padrão da linguagem, o que maximiza a portabilidade (mas não oferece algumas garantias que programadores C normalmente esperam).

2.1.2 Paradigmas

Discutivelmente, a classificação mais comum para linguagens de programação é por seu paradigma; paradigmas definem características semânticas da linguagem, e, por consequência, afetam diretamente a forma idiomática de se programar: duas linguagens com o mesmo paradigma costumam ter estilos similares de uso. Os principais paradigmas serão discutidos nessa sub-sessão; para informações mais detalhadas, recomenda-se a leitura de (ROY, 2012).

Nota-se que linguagens costumam apresentar mais de um paradigma; para esse trabalho, quando uma linguagem for classificada como pertencendo a um paradigma, isso significará que a linguagem foi projetada para acomodar um estilo de programação que conforme com o paradigma, sem a necessidade interferência de outros paradigmas, similar à definição dada em (ROY, 2012).

[4] Ou, então, ser interpretadas.

A principal divisão de paradigma considerada é pelo controle de estado oferecido pela linguagem; linguagens imperativas, com um controle explícito de estado, e linguagens declarativas, implicitamente através de notações equacionais; o paradigma declarativo pode ser sub-dividido em funcional, lógico e de *dataflow*.

Linguagens imperativas, como C , “C” e Fortran, são caracterizadas pelo controle de estado, onde variáveis podem (e de fato costumam) ser mutáveis, e funções podem ter efeitos colaterais (como alterar memória global, arquivos, etc) de maneira irrestrita, através de uma sequência de instruções. Linguagens de montagem, por seu controle sequencial sobre os recursos de *hardware*, são linguagens imperativas[5].

Linguagens declarativas, por sua vez, apresentam dados de forma equacional, com a intenção de serem referencialmente transparentes: o uso de um objeto pode ser trocado livremente por sua definição, similar a definições matemáticas. Para que tal seja possível, funções não devem apresentar efeitos colaterais, sendo então chamadas de funções puras (onde múltiplas chamadas da mesma função, com os mesmos argumentos, sempre retornam o mesmo resultado).

Linguagens funcionais tratam funções como objetos de primeira classe (funções podem receber como argumento, ou retornar, outras funções), fazendo grande uso de composição de funções e de *closures* (funções que capturam variáveis de outros escopos). Várias linguagens como Common Lisp e Standard ML são classificadas como funcionais pelo seu estilo idiomático de programação, embora também possuam variáveis mutáveis e permitam um estilo imperativo, que costuma (por convenção) ser isolado em pedaços de código e usados para otimizações. Dizemos que uma linguagem é puramente funcional quando a mesma não permite a escrita de código imperativo, como por exemplo Haskell, onde ações que envolvem o sistema (arquivos, rede, etc) e são inerentemente mutáveis são representadas através de técnicas como mônadas, efeitos algébricos, ou tipos lineares (os quais serão vistos na Seção 2.5).

Um subconjunto das linguagens funcionais, chamadas de linguagens funcionais totais, motivadas em (TURNER, 2004), são linguagens puramente funcionais que, em tempo de compilação, garantem que todas as funções aceitas são totais, isto é, todas as funções provadamente terminam[6]. Tais linguagens são particularmente interessantes para a construção de assistentes de provas e linguagens de contratos inteligentes para criptomoedas (O’CONNOR, 2017). Funções que não são provadamente totais e as linguagens que as suportam são chamadas de parciais. O espaço de *design* para linguagens imperativas totais foi pouco explorado, porém é possível

[5] Embora algumas arquiteturas de *hardware* como DEC Alpha possuam modelos de memória fracos, e precisem de tratamento especial para garantir ordem.

[6] Em linguagens funcionais totais, todos os algoritmos aceitos terminam, porém nem todos os algoritmos que terminam são aceitos (afinal, o problema da parada é indecidível).

através de semântica de jogos (CHURCHILL; LAIRD; MCCUSKER, 2011).

Linguagens lógicas, como Prolog, são baseadas em alguns tipos de lógica matemática (como lógica de primeira ordem) e apresentam o programa na forma de cláusulas e relações entre variáveis que devem, então, ser solucionadas através de um algoritmo de *backtracking* ou solucionador similar. Linguagens de *dataflow*, como Lucid, apresentam variáveis como um fluxo de valores através do tempo e suas dependências; valores são automaticamente recalculados caso uma de suas dependências mude (e, por isso, podem limitar ou então apresentar problemas em definições cíclicas, geralmente chamados de *glitches*).

Ortogonal aos paradigmas acima está o paradigma orientado a objetos. Dizemos que uma linguagem é orientada a objetos quando suas principais entidades primitivas são objetos, variáveis capazes de armazenar atributos (estado) e uma lista de operações, apresentando alguma forma de polimorfismo (geralmente por subtipagem). Embora geralmente associado a linguagens imperativas, existem linguagens orientadas a objetos declarativas como OCaml e Scala. Dizemos que uma linguagem é puramente orientada a objetos quando suas únicas entidades primitivas são objetos (isto é, “tudo é um objeto”), como por exemplo em SmallTalk e Self.

Algumas linguagens, como C , D, Fortran, PL/I e Rust, são classificadas como linguagens de programação de sistemas por serem projetadas com o intuito de se poder controlar diretamente o *hardware*, permitindo a escrita de sistemas operacionais e *drivers* em linguagens de alto nível (tarefa classifcamente reservada para linguagens de baixo nível). Para tal, oferecem recursos como o acesso a posições arbitrárias de memória[7] e permitem a escrita de código de montagem diretamente dentro delas (do inglês, *inline assembly*). Como consequência, caso tais recursos sejam usados, o programa deixa de ser portável pois depende de informações da implementação.

2.1.3 Linguagens Intermediárias

A maioria das linguagens de programação não possui uma semântica adequada para a otimização de programas, o que dificulta uma tradução direta de uma linguagem de alto nível para uma linguagem de baixo nível de forma eficiente. As chamadas linguagens intermediárias são usadas internamente por compiladores para a otimização do código, antes da conversão para uma linguagem de montagem. Isto é, código fonte escrito pelo programador é traduzido para uma sequência de linguagens intermediárias, otimizações são aplicadas, e só então o programa é traduzido para a linguagem final.

[7] Nota-se que “C” não possui tal característica; o padrão da linguagem C não garante ou exige que os ponteiros da linguagem sejam endereços reais para a memória, e a conversão de números literais para ponteiros é dependente da implementação.

Duas linguagens intermediárias comuns, o cálculo lambda em estilo de passagem de continuação e a *static single assignment form* serão abordadas na Seção 2.6. Essas duas linguagens são, respectivamente, usadas com frequência em compiladores para linguagens funcionais e imperativas.

2.1.4 Sistemas de Tipos

Sistemas de tipos são um conjunto de regras que designam “tipos” para entidades computacionais (como variáveis, expressões e funções), limitando os valores que elas podem representar em uma linguagem de programação (PIERCE, 2002). O propósito principal de sistemas de tipos é reduzir as chances de comportamentos errados (e a introdução de *bugs*) em um programa, limitando operações a seus respectivos tipos (CARDELLI, 1996). O estudo formal de sistemas de tipos é chamado de teoria de tipos, que será abordado na Seção 2.2.

Todas as linguagens definem tipos para suas entidades computacionais; há linguagens de programação que não apresentam um sistema de tipos aparente e, convencionalmente, costumam ser chamadas de linguagens não-tipadas (do inglês, *untyped*). Uma interpretação alternativa válida é de que todas as entidades, na verdade, possuem um mesmo e único tipo, tornando a linguagem unitipada (do inglês, *unityped*) (CARDELLI, 1996). Exemplos de linguagens unitipadas são as linguagens de montagem convencionais e Forth, onde o único tipo é a palavra binária do processador (que pode ser interpretada como inteiro, ponto flutuante, etc).

Embora não haja consenso sobre a nomenclatura na literatura (CARDELLI, 1996), há linguagens, chamadas informalmente de dinamicamente tipadas, que, similar às linguagens unitipadas, não fazem uma verificação estática (em tempo de compilação) sobre o sistema de tipos; entretanto, durante o tempo de execução, áreas de memória possuem um tipo designado, e verificações de tipagem são feitas (como, por exemplo, em Scheme e Ruby). Essa verificação em tempo de execução também é chamada de checagem dinâmica (ao invés de tipagem dinâmica) a fim de evitar a ambiguidade (CARDELLI, 1996), e linguagens que apresentam ambas checagens estáticas e dinâmicas usam uma informação dinâmica de tipos derivada da estática, como por exemplo em Java (com a *keyword* `instanceof`) e C++ (com a *keyword* `typeid`).

O termo fortemente tipado (do inglês, *strongly typed*), também bastante ambíguo na literatura, é aplicado a sistemas de tipos onde cada área de memória possui um tipo distinto, e cada processo da linguagem tem seus requerimentos baseados nesses tipos. Em outras palavras, cada objeto na memória, sendo ele designado estaticamente ou dinamicamente, possui um tipo específico. O termo recíproco, fracamente tipado (do inglês, *weakly typed*), é usado informalmente para se referir a linguagens onde um objeto na memória possa ser interpretado como mais de um tipo, como por

exemplo em C , através da tática de *type punning*[8] (o que pode causar erros no programa, e é grandemente limitada em “C”, onde objetos na memória são fortemente tipados, e seu uso indevido resulta em *undefined behavior*).

Conforme (CARDELLI, 1996), erros em um programa podem ser divididos em erros capturados (do inglês, *trapped error*), e em erros não-capturados (do inglês, *untrapped error*). O primeiro se refere a erros que podem ser detectados imediatamente, como uma divisão por zero ou uma exceção, e o segundo se refere a falhas que podem passar despercebidas por uma quantidade arbitrária de tempo, como o acesso a posições inválidas de memória. Sistemas de tipos (incluindo dinâmicos) podem definir um conjunto de erros proibidos, que costumam englobar todos os erros não-capturados. Quando um sistema de tipos rejeita qualquer erro não-capturado em todos os programas possíveis (de forma estática ou dinâmica), chamamos sua linguagem de segura[9].

Linguagens podem oferecer sistemas de tipos simples, como C e “C”, onde objetos são apenas classificados dentre inteiros, pontos flutuantes, pontos fixos, etc, ou sistemas de tipos que oferecem garantias de segurança, como Rust, cujo sistema de tipos é seguro e garante que, caso um programa compile sem erros, jamais haverão condições de corrida em tempo de execução. Sistemas de tipos mais complexos podem ainda oferecer garantias comportamentais: caso um programa compile, pôde ser provado que não haverão falhas na execução do programa; a Seção 2.3 abordará a relação entre sistemas de tipos e sistemas de lógica.

Uma característica importante de sistemas de tipos (estáticos) é a necessidade de serem decidíveis: é necessário que exista um algoritmo, chamado de algoritmo de checagem de tipo (do inglês, *typechecking algorithm*), capaz de verificar todos os programas possíveis para a linguagem e aceitá-los ou rejeitá-los de acordo com as regras do sistema de tipos. Um programa rejeitado será um programa onde não foi possível provar as propriedades garantidas pelo sistema de tipos.

2.2 TEORIA DE TIPOS

A teoria de tipos foi inicialmente proposta por Bertrand Russell em 1908 como um método para a fundamentação da matemática. Russell havia percebido que tentativas de formalização do que hoje chamamos de teoria ingênua de conjuntos apresentavam contradições; o então chamado paradoxo de Russell pode ser expressado na

[8] Também chamada de *coercion*, os *bits* de um objeto de um tipo são interpretados como sendo de outro tipo arbitrário. Nota-se que o mesmo é diferente de uma conversão, onde um novo objeto é criado.

[9] Por questões de otimização e para a programação de sistemas, linguagens seguras costumam oferecer métodos para a execução de código inseguro, como a função `unsafeCoerce` em Haskell, a *keyword* `unsafe` em C# e Rust, e a infame classe `sun.misc.Unsafe` em Java.

seguinte forma:

$$\text{Considere } R = \{x \mid x \ / \ x\}, \text{ então } R \quad R \Leftrightarrow R \ / \ R$$

Em outras palavras, “considere R o conjunto de todos os conjuntos que não contém a si mesmos, então caso R contenha a si mesmo implicará que R não contém a si mesmo (e vice-versa)”, o que é logicamente contraditório, e, portanto, inconsistente. Dizemos que um sistema lógico é *consistente* quando o mesmo não apresenta contradições (isto é, não podemos derivar ambos um teorema α e sua negação $\neg\alpha$); o princípio da explosão diz que, em um sistema com contradições, tudo pode ser provado.

O paradoxo pode também ser descrito em termos do chamado paradoxo do barbeiro: há uma cidade com apenas um barbeiro, e este barbeiro faz a barba daqueles (e apenas daqueles) que não fazem a própria barba. A pergunta a ser feita: quem barbeia o barbeiro? O barbeiro não pode fazer a própria barba pois apenas pode barbear aqueles que não fazem a própria barba; segue que, como o barbeiro não faz a própria barba, ele se enquadra no grupo de homens que devem ser barbeadas pelo barbeiro, e portanto ele deve fazer a própria barba. Há uma clara contradição. Tal problema pode, por exemplo, ser descrito na linguagem de programação lógica Prolog, conforme a Figura 1.

Figura 1 – Paradoxo do barbeiro em Prolog

```
1 shaves(barber, X) :- person(X), not shaves(X, X).
2 person(barber).
```

Fonte: o autor

Nota-se que, caso esse código seja executado, resultará em um *loop* infinito, incapaz de fornecer uma resposta.

O principal uso atual da teoria de tipos, embora possua aplicabilidade em áreas próprias da matemática (Univalent Foundatiosn Program, 2013), é a formalização de sistemas de tipos para linguagens de programação dentro da ciência da computação. As aplicabilidades são diversas; dentre elas, a inferência automática de tipos (estáticos, porém sem a necessidade de anotações feitas pelo programador), a prevenção de falhas, e a prova de propriedades arbitrárias sobre programas. Em algumas linguagens funcionais totais, como Agda e Gallina[10], cujos sistemas de tipos representam um sistema lógico, a distinção entre as duas áreas é bem menos evi-

[10] Linguagem usada pelo assistente de provas Coq.

dente, e programas são usados efetivamente como provas de teoremas descritos pelo próprio sistema de tipos (conforme será abordado na Seção 2.2.4).

No contexto da teoria de tipos, as formalizações de sistemas de tipos costumam ser apresentadas em um sistema de reescrita de termos (a chamada semântica operacional), como o cálculo pi, o cálculo sigma, e, especialmente, o cálculo lambda. Sistemas de reescrita de termos são sistemas formais onde os objetos trabalhados, os termos, são descritos indutivamente a partir de uma sintaxe, podendo conter subtermos; o sistema, então, fornece regras de reescrita, fornecendo transformações válidas de termos para termos dentro do sistema (chamadas de reduções).

2.2.1 O Cálculo Lambda

O cálculo lambda, apresentado por Alonzo Chuch em 1936, é um sistema formal usado para representar computações através de abstrações e aplicações, usando o conceito de reduções (PIERCE, 2002), relações entre termos na linguagem. Apesar de sua simplicidade, ele é capaz de representar qualquer computação, tornando-o equivalente a uma máquina de Turing.

Como um sistema de reescrita de termos, o cálculo lambda pode ser descrito a partir da seguinte sintaxe:

$$e ::= x \mid \lambda x.e \mid e\ e$$

Na descrição acima, x é uma meta-variável[11] que representa possíveis variáveis (identificadores) dentro do sistema. A própria notação da sintaxe define a meta-variável e classe sintática e representa possíveis termos dentro do sistema, que indutivamente possuem apenas uma de três possíveis formas: uma variável, uma abstração (também chamada de função lambda), ou uma aplicação, respectivamente. Quanto à notação, aplicações possuem a maior precedência, e associam para a esquerda. Isto é, o termo $a\ b\ c$ é equivalente a $((a\ b)\ c)$ e o termo $\lambda x.y\ z$ é equivalence a $(\lambda x.(y\ z))$; parênteses podem ser adicionados para controlar precedência, por exemplo para os termos $a\ (b\ c)$ e $(\lambda x.y)\ z$, onde os parânteses são necessários.

As abstrações são definidas por um identificador, que efetivamente representa o parâmetro para a função, e por seu retorno (um termo arbitrário); isso é similar às funções encontradas em linguagens de programação modernas, embora as restringindo a um único argumento. Para se obter funções com dois ou mais parâmetros, uma abstração deve retornar uma nova abstração. Por exemplo, o termo $\lambda a.\lambda b.a\ b$ é uma função lambda que recebe um parâmetro, a, e retorna uma função lambda com

[11] Uma variável que representa demais possíveis variáveis em uma linguagem, dentro de uma classe sintática.

um parâmetro b. A transformação de funções de múltiplos argumentos para uma série de funções de um único argumento é chamada de *currying*.

Variáveis que existem em um termo são chamadas de variáveis livres quando não estão ligadas a um parâmetro introduzido por uma função lambda que contém tal termo como parte de seu corpo, similar ao conceito de escopo em linguagens de programação modernas. Por exemplo, o termo lambda $a\ b$ possui duas variáveis livres, a e b, que não estão ligadas a uma abstração. Entretanto, o termo $\lambda a.b\ a$ liga a variável a dentro do seu corpo; avaliando o termo como um todo, apenas a variável b está livre. Nota-se que funções apenas podem ligar variáveis dentro de seu corpo, portanto o termo $(\lambda a.b)\ a$ possui duas variáveis livres (visto que o subtermo a não está contido na abstração que liga tal variável). Um termo que não possui variáveis livres é chamado de fechado, como por exemplo $\lambda a.\lambda b.a\ b$.

A semântica do cálculo lambda, ou, em outros termos, a forma de se interpretar programas escritos nele, é dada por três tipos de redução: α, β e η. Reduções podem ser representadas pelo símbolo $\rightarrow$, significando que um termo pode ser reduzido por qualquer uma das três reduções a outro termo. Tal símbolo também pode ser anotado com a(s) redução(ões) específica(s) caso desejado, por exemplo $\rightarrow_\alpha$ representando apenas reduções α ou $\rightarrow_{\beta\eta}$ representando reduções β ou η. A notação $\twoheadrightarrow$ é a relação de redução fechada sobre reflexividade e transitividade, que representa a redução de um termo a outro em zero ou mais passos; isto é, $\twoheadrightarrow$ é uma função tal que 1) se $e_1 \rightarrow e_2$, então $e_1 \twoheadrightarrow e_2$, 2) para todo termo e, $e \twoheadrightarrow e$ (a função é reflexiva), e 3) caso $e_1 \twoheadrightarrow e_2$ e $e_2 \twoheadrightarrow e_3$, então $e_1 \twoheadrightarrow e_3$ (a função é transitiva). Termos que podem ser reduzido são chamados de redexes (do inglês, *reducible expression*).

A redução α é usada para renomear variáveis ligadas dentro de uma abstração (que são α-redexes). Por exemplo, $\lambda a.a\ x \rightarrow_\alpha \lambda b.b\ x$: o parâmetro a foi renomeado para b, e todas as ocorrências de a dentro do corpo da abstração são renomeadas de acordo. Nota-se que variáveis livres não são renomeadas, como por exemplo $a\ \lambda a.a \rightarrow_\alpha a\ \lambda x.x$, pois o primeiro a é livre, enquanto o segundo é ligado.

Existem casos onde o mesmo parâmetro pode ser usada de forma aninhada, situação onde as variáveis livres de um subtermo são ligadas à abstração mais próxima. Por exemplo, no termo $\lambda a.(\lambda a.a)$ (parênteses adicionados para claridade), a variável a é ligada ao subtermo entre parênteses, que não possui variáveis livres. Como consequência, $\lambda a.(\lambda a.a) \rightarrow_\alpha \lambda a.(\lambda b.b)$ e $\lambda a.(\lambda a.a) \rightarrow_\alpha \lambda b.(\lambda a.a)$.

Tal situação, onde o mesmo parâmetro é usado de forma aninhada, pode causar confusão ao leitor, e dificultar a implementação em *software*. Uma solução propõe trocar o uso de identificadores ligados como variáveis por índices numéricos, chama-

dos de índices de de Bruijn[12], que representam a proximidade da variável à abstração que a liga. Por exemplo, o termo $\lambda z.(\lambda y.y\ (\lambda x.x))\ (\lambda x.z\ x)$ é anotado com índices de de Brujin como $\lambda(\lambda 1\ (\lambda 1))\ (\lambda 2\ 1)$, conforme ilustrado abaixo:

Esse tipo de notação elimina a necessidade de reduções α, visto que variáveis ligadas não são mais representadas por identificadores (PIERCE, 2002).

A redução β é usada para a aplicação de funções; aplicações cujo subtermo à esquerda é uma abstração são β-redexes. Um termo $(\lambda x.e_1)\ e_2$ é β-reduzido para um termo e_1 com todas variáveis x livres à abstração substituído por e_2 (lembrando que, considerando e_1 isoladamente, todas as variáveis x que estariam ligadas pela abstração $\lambda x.e_1$ são variáveis livres, visto que o propósito da abstração é propriamente ligar todas as variáveis x no subtermo e_1); a notação $a[b/x]$, onde a e b são dois termos arbitrários, representa a substituição de quaisquer ocorrências da variável livre x dentro de a por b (o que pode acarretar reduções α para manter o significado quando índices de de Bruijn não são usados). Portanto, temos que $(\lambda x.e_1)\ e_2 \rightarrow_\beta e_1[e_2/x]$, e, por exemplo, $(\lambda a.a\ b)\ c \rightarrow_\beta c\ b$.

A redução η é usada para "prever" posições que se tornaram β-redexes no futuro, e nem sempre é considerada em teorias baseadas no cálculo lambda. Um η-redex tem a forma de uma abstração $\lambda x.e\ x$, cujo corpo é uma aplicação, onde o subtermo à direita é a variável ligada pela abstração, e tal variável não aparece livre no subtermo à esquerda. Tal redex, então, η-reduz para o subtermo e; por exemplo, $\lambda a.b\ a \rightarrow_\eta b$. É fácil perceber seu significado; por exemplo, temos que $(\lambda a.b\ a)\ x \rightarrow_\beta b\ x$, o mesmo que teríamos com a η-redução $(\lambda a.b\ a)\ x \rightarrow_\eta b\ x$, embora a η-redução possa ser usada mesmo considerando apenas a abstração isoladamente (onde não existe um $\beta-$redex).

É comum o uso da notação $\texttt{let}\ x\ =\ a\ \texttt{in}\ b$ para se representar o termo $(\lambda x.b)\ a$, que é claramente um β-redex; tal estrutura também pode existir como primitiva em versões extendidas do cálculo lambda, onde o sistema de tipos pode dar tratamentos diferentes a expressões `let` e a β-redexes. Nesse caso, temos a chamada ζ-redução, similar à β-redução esperada para esse caso, tendo que $\texttt{let}\ x\ =\ a\ \texttt{in}\ b \rightarrow_\zeta b[a/x]$.

O cálculo lambda, conforme descrito acima, é o suficiente para representar quaisquer computações; é possível, por exemplo, representar números, cálculos nu-

[12] Nomeado em honra ao matemático Nicolaas Govert de Bruijn.

méricos, tuplas, e até mesmo recursão (esta através do combinador Y[13]). Por praticidade, e por questões de otimização quando implementado em um computador, não é incomum que o cálculo lambda seja extendido com novas formas para seus termos.

Por exemplo, podemos descrever uma sintaxe para o cálculo lambda extendido com números (com adição e subtração) e pares:

$$e ::= x \mid \lambda x.e \mid e\ e \mid n \mid e + e \mid e - e \mid \ e,\ e\ \mid \pi_1\ e \mid \pi_2\ e$$

Nessa linguagem, a meta-variável n representa os números inteiros. Reduções adicionais, então, podem ser adicionadas para se operar em números e pares. Por exemplo, poderíamos definir que $10 + (20 - 5) \rightarrow 25$. Pares podem ser construídos usando os parênteses angulares, por exemplo, $\langle 10,\ \lambda x.x \rangle$. Podemos também definir reduções para pares, onde $\pi_1\ \langle a,\ b \rangle$ e $\pi_2\ \langle a,\ b \rangle$ são β-redexes, β-reduzindo como $\pi_1\ \langle a,\ b \rangle \rightarrow_\beta a$ e $\pi_2\ \langle a,\ b \rangle \rightarrow_\beta b$ respectivamente. Tal tipo de projeção para pares se chama projeção forte, e será melhor discutida na Seção 2.2.4.

Devido ao poder computacional do cálulo lambda, a literatura descreve um grande número de sistemas de tipos para o mesmo, que são usados para restringir a quantidade de termos que são aceitos pelo cálculo, similar ao feito nas linguagens de programação (e, na verdade, precedendo as mesmas). O primeiro exemplo foi o cálculo lambda simplesmente tipado, introduzido por Alonzo Church em 1940.

2.2.2 A linguagem dos Sistemas de Tipos

A notação convencional para se representar sistemas de tipos, em especial para o cálculo lambda, se baseia em julgamentos, relações que podem ser feitas pelas regras do sistema, utilizando dedução natural. A existência de uma relação de julgamento dentro de um sistema de tipos, convencionalmente, pode ser representada conforme a seguir. Tal notação demonstra o formato da relação em um sistema de tipos; por exemplo, podemos declarar o julgamento a seguir:

$\Gamma \quad T$:

De acordo com as convenções adotadas na literatura, tal julgamento pode ser lido como “no contexto Γ, T é um tipo”. Aqui, Γ é a meta-variável que representa o contexto atual, T é a meta-variável que representa tipos.

O cálculo lambda simplesmente tipado, por exemplo, é definido utilizando um conjunto de tipos básicos, também chamados de tipos primitivos ou atômicos. Se considerarmos o cálculo lambda simplesmente tipado com números naturais, para os quais daremos o tipo N, o sistema de tipos terá o seguinte julgamento:

$\Gamma \quad \mathrm{N}$:

[13] Cuja definição pode ser dada como $\lambda f.(\lambda x.f\ (x\ x))\ (\lambda x.f\ (x\ x))$, e onde $Y\ f \rightarrow^* f\ (Y\ f)$.

Que pode ser lido como “em qualquer contexto Γ, N é um tipo”. Tal informação pode ser usada para construir tipos mais complexos, conforme será visto na Seção 2.2.3.

Contextos, usados para representar informações derivadas em sistemas de tipos, representam um conjunto de informações, mapeando, por exemplo, variáveis a seus tipos. Evidentemente, um identificador pode ter tipos diferentes em contextos diferentes (similar a como podemos usar o mesmo nome para duas variáveis de tipos diferentes em pontos diferentes de um programa). A sintaxe dos contextos, que pode variar dentre sistemas de tipos, pode por exemplo ser definida como:

$$\Gamma ::= \cdot \mid \Gamma, x\colon T$$

Nota-se que contextos costumam representar sequências, visto que um elemento pode depender de demais elementos mais à esquerda; entretanto, é comum aproveitar-se de notações sintáticas usadas para conjuntos quando o significado pode ser inferido, e é usada meramente por comodidade. Caso tal informação não esteja clara, ela pode ser informada explicitamente pelo autor.

Além de definições de quais tipos são válidos, sistemas de tipo tem o papel de permitir a derivação de tipos para termos da linguagem. Um julgamento essencial em sistemas de tipos é a asserção de que um termo, em certo contexto, possui um tipo:

$$\boxed{\Gamma \quad e\colon T}$$

Tal julgamento pode ser lido como “no contexto Γ, o termo e tem tipo T”. Por exemplo, novamente assumindo o cálculo lambda simplesmente tipado com números naturais, onde n é a meta-variável representando números naturais, tal julgamento fará parte do sistema de tipos:

$$\Gamma \quad n\colon \mathsf{N}$$

Variáveis, o caso base para a sintaxe do cálculo lambda, tem seu tipo derivado diretamente do contexto. Tal julgamento pode ser representado da seguinte forma, através de dedução natural:

$$\frac{x\colon T \quad \Gamma}{\Gamma \quad x\colon T}$$

Acima da linha constam as condições para o julgamento, e abaixo da linha estão as conclusões. A regra acima pode ser lida como “caso $x\colon T$ pertença ao contexto Γ, então, no contexto Γ, x tem tipo T”. Usando o abuso de notação mencionado acima, a mesma regra acima pode ser simplificada, eliminando suas pré-condições:

$$\Gamma, x\colon T \quad x\colon T$$

Termos mais complexos (como funções e pares), então, necessitam de tipos mais complexos. Sistemas de tipos permitem derivar tipos compostos que podem ser construídos dentro da linguagem. Dentre os tipos compostos comuns se encontram os tipos de função, tipos de produto e tipos de soma.

2.2.3 Construtores de Tipos

Uma característica fundamental em sistemas de tipos é a habilidade de construir novos tipos a partir de tipos já existentes dentro do sistema. Tais tipos compostos podem então ser usados para termos mais complexos da linguagem, como, por exemplo, funções e pares.

Tipos de função (do inglês, *function types*, também chamados de *arrow types* ou *exponential types* devido a sua relação com a teoria de conjuntos), geralmente representados por uma seta ($\rightarrow$), representam os tipos de objetos que mapeiam um parâmetro de tipo A a um retorno de tipo B (equivalentes ao domínio e ao codomínio, respectivamente, de uma função na teoria de conjuntos). Por exemplo, podemos definir a seguinte regra:

$$\frac{\Gamma \quad A: \qquad \Gamma \quad B:}{\Gamma \quad A \rightarrow B :}$$

Isto é, caso A e B sejam tipos válidos no contexto Γ, então o tipo $A \rightarrow B$ também é válido no contexto Γ, e representa funções que mapeiam um parâmetro de tipo A para um retorno de tipo B. Tais tipos de regra são chamads de regras de formação (do inglês, *formation rules*).

Por exemplo, considerando o cálculo lambda simplesmente tipado com números naturais e valores booleanos, onde N é o tipo primitivo dos números naturais e B é o tipo primitivo dos valores booleanos, poderíamos derivar que o tipo N $\rightarrow$ B $\rightarrow$ N (onde $\rightarrow$ associa para a direita) existe no sistema, através de dedução natural:

$$\frac{\Gamma \quad \text{N}: \qquad \dfrac{\Gamma \quad \text{B}: \qquad \Gamma \quad \text{N}:}{\Gamma \quad \text{B} \rightarrow \text{N} :}}{\Gamma \quad \text{N} \rightarrow \text{B} \rightarrow \text{N} :}$$

O sistema de tipos, então, fornece uma ou mais regras, chamadas de regras de introdução, para construir termos de tais tipos. Por exemplo, para introduzirmos um elemento com um tipo de função no cálculo lambda simplesmente tipado:

$$\frac{\Gamma, a: A \quad b: B}{\Gamma \quad \lambda a.b : A \rightarrow B}$$

Ou seja, caso no contexto $\Gamma, a\colon A$ a expressão b tenha tipo B, então em Γ a expressão $\lambda a.b$ é válida e terá tipo $A \to B$, sendo assim possível a criação de funções.

Note que a regra para funções lambda, conforme ilustrada acima, não apresenta informações sobre o tipo do parâmetro, apenas que, se com algum tipo A a função retorne B, então ela pode ter tipo $A \to B$. Isso também permite que a mesma abstração tenha mais de um tipo possível; chamamos isso de estilo Curry (PIERCE, 2002). Uma alternativa, chamada de estilo Church[14] (PIERCE, 2002), altera a sintaxe da linguagem para incluir, explicitamente, o tipo do argumento na abstração. Nesse caso, a regra seria anotada da seguinte forma:

$$\frac{\Gamma, a\colon A \quad b\colon B}{\Gamma \quad \lambda a\colon A.b : A \to B}$$

Além das regras de introdução, o sistema de tipos fornece uma ou mais regras de eliminação, com o objetivo oposto: operar, e efetivamente desfazer, objetos criados a partir de uma regra de introdução. Por exemplo, para eliminarmos um elemento com um tipo de função:

$$\frac{\Gamma \quad f\colon A \to B \qquad \Gamma \quad a\colon A}{\Gamma \quad f\ a\colon B}$$

Isto é, caso f e a tenham tipos $A \to B$ e A respectivamente no contexto Γ, então a expressão $f\ a$ (uma aplicação de função) é válida e terá tipo B em Γ, representando efetivamente o retorno da função.

Tipos de produto (do inglês, *product types*), que aparecem em diversas linguagens como pares, representam a junção de dois tipos, ou seja, representam um par de elementos com os tipos usados para a construção. Sua regra de formação pode ser descrita da seguinte forma:

$$\frac{\Gamma \quad A\colon \qquad \Gamma \quad B\colon}{\Gamma \quad A \times B :}$$

Cuja interpretação é simples; caso A e B sejam tipos no contexto Γ, podemos então derivar o tipo $A \times B$ como válido no contexto Γ. Para usarmos tais tipos, necessitamos de uma regra de introdução; ao introduzir um par, dois termos de tipos distintos podem ser unidos em um único termo:

$$\frac{\Gamma \quad a\colon A \qquad \Gamma \quad b\colon B}{\Gamma \quad a, b\ : A \times B}$$

[14] Os estilos foram nomeados em honra aos matemáticos Haskell Curry e Alonzo Church.

Para extrairmos os subtermos de um par, podemos usar as regras de eliminação. Duas regras de eliminação, chamadas de projeções, podem ser usadas em um par, para se recuperar o primeiro e o segundo termo usado na introdução:

$$\frac{\Gamma \quad x\colon A \times B}{\Gamma \quad \pi_1 x : A} \qquad\qquad \frac{\Gamma \quad x\colon A \times B}{\Gamma \quad \pi_2 x : B}$$

Tipos de soma (do ingês, *sum types*), que aparecem em diversas linguagens como enumerações ou uniões disjuntas, representam alternativas entre dois tipos, isto é, um termo de um tipo de soma será representado por apenas um termo de um dos dois tipos usados na formação, mas não ambos (como seria o caso para tipos de produto). Sua regra de formação pode ser descrita como:

$$\frac{\Gamma \quad A\colon \qquad \Gamma \quad B\colon}{\Gamma \quad A + B :}$$

De uma forma dual ao tipo de produtos, descritos acima com uma regra de introdução e duas de eliminação (pois contém dois subtermos, unidos na introdução, que podem ser eliminados separadamente), tipos de soma terão duas formas de introdução (pois contém apenas um subtermo) e apenas uma forma de eliminação (que efetivamente funciona como uma condicional, recebendo duas *branches*):

$$\frac{\Gamma \quad x\colon A}{\Gamma \quad \iota_1\ x : A + B} \qquad\qquad \frac{\Gamma \quad x\colon B}{\Gamma \quad \iota_2\ x : A + B}$$

$$\frac{\Gamma \quad x\colon A + B \qquad \Gamma \quad f\colon A \to C \qquad \Gamma \quad g\colon B \to C}{\Gamma \quad [x,\ f,\ g] : C}$$

De forma similar aos tipos de função e de produto descritos, as regras acima se referem à tipagem dos termos. Em um cálculo lambda com somas, é possível a existência de regras de redução tal que $[\iota_1\ a,\ f,\ g] \to_\beta f\ a$ e $[\iota_2\ b,\ f,\ g] \to_\beta g\ b$.

Digressão: a teoria de tipos tem suas origens na teoria de conjuntos, fato que explica parte da sua nomenclatura. Não é incomum o uso de números para representar tipos, especificamente os números 0, 1 e 2, para representar tipos com esses exatos números de elementos (onde, por exemplo, 2 representa o tipo booleano, cujos valores apenas podem ser verdadeiro e falso). Esses números representam a cardinalidade de conjuntos, e os construtores de tipos representam a cardinalidade dos tipos criados; por exemplo, 2×2 (o tipo de produto de tipos booleanos) representa um tipo com 4 possíveis valores (nominalmente, `false`, `false` , `true`, `false` , `false`, `true` , e `true`, `true`).

2.2.4 Tipos Dependentes

Os construtores descritos acima na Seção 2.2.3 são comuns em linguagens de programação convencionais. Nos três casos descritos, há a combinação de dois tipos previamente existentes no sistema, porém sempre de forma fixa; por exemplo, nos tipos de funções, o tipo de retorno não varia de acordo com o seu argumento (em outras palavras, o codomínio não depende do domínio).

Sistemas de tipos mais complexos ampliam essas possibilidades; por exemplo, o Sistema F permite criar abstrações em tipos, criando assim funções polimórficas, e o Sistema F_ω fornece abstrações lambda no nível dos tipos, permitindo assim tipos genéricos, como por exemplo listas (PIERCE, 2002).

Tipos dependentes apresentam generalizações dos tipos acima, tal que o segundo tipo usado passa a ser uma família de tipos, dependendo do primeiro elemento. Os dois casos gerais são os tipos Π, e os tipos Σ. Embora essa generalização permita polimorfismo e tipos genéricos, o termo tipo dependente costuma ser usado para designar, especificamente, quando a linguagem permite uma abstração cujo parâmetro é um termo, e o retorno é um tipo; isto é, a linguagem permite, por exemplo, vetores de tamanho n, onde n é uma variável dentro do próprio programa.

Um tipo Π, também chamado de tipo generalizado de produto, representa uma função dependente: o parâmetro fornecido à função pode fazer parte do tipo de retorno (e, por consequência, parâmetros diferentes podem resultar em tipos de retorno diferentes). Um tipo Π é criado da seguinte forma[15]:

$$\frac{\Gamma \quad A: \qquad \Gamma,\ a : A \quad B:}{\Gamma \quad \Pi a{:}\ A.B\ :}$$

Suas regras de introdução e eliminação são similares às de tipos de função (visto que esses são um caso específico de tipos Π):

$$\frac{\Gamma,\ a{:}\ A \quad b : B}{\Gamma \quad \lambda a{:}\ A.b : \Pi a{:}A.B} \qquad\qquad \frac{\Gamma \quad f : \Pi a{:}\ A.B \qquad \Gamma \quad x : A}{\Gamma \quad f\ x : B[x/a]}$$

Deve-se notar que, na eliminação, a variável a pode ser livre em B, e será substituída pelo argumento fornecido. O exemplo canônico de tipos dependentes, usando tipos Π, é o tipo de vetores de tamanho conhecido em tempo de compilação. Por exemplo, f: Πn : N.V n é uma função que recebe um número natural n, e retorna um vetor de n elementos; chamamos, então, V n de um tipo dependente. Como consequência, f 5 terá tipo V 5, e f 10 terá tipo V 10, tipos distintos.

[15] Usamos a notação de Church pois a checagem de tipos para tipos dependentes com estilo de Curry é indecidível (BARTHE; HATCLIFF; SØRENSEN, 1999).

Na presença de tipos Π, é comum o uso da notação $A \to B$ para designar tipos $\Pi a\colon A.B$ caso a não apareça livre em B (ou seja, o retorno é fixo, tal que B não dependende de a, como em tipos de função convencionais, justificando a notação).

Da mesma forma que tipos Π generalizam funções, tipos Σ, também chamados de tipos generalizados de soma, generalizam pares, tal que o tipo do segundo elemento do par pode depender do primeiro. Tipos Σ podem ser representados da seguinte forma:

$$\frac{\Gamma \quad A\colon \qquad \Gamma,\ a\colon A \quad B\ :}{\Gamma \quad \Sigma a\colon A.B\ :}$$

Tal tipo, quando existente como primitivo em um sistema de tipos, é chamado de tipo sigma forte (do inglês, *strong sigma type*). Tipos Σ podem, entretanto, ser representado através de tipos Π (conforme será visto na Seção 2.4.1), onde podem ser chamados de tipo sigma fraco. As regras de introdução e eliminação de tipos Σ fortes podem ser representadas da seguinte forma, similares às de tipos de produtos (visto que esses são um caso específico de tipos Σ):

$$\frac{\Gamma \quad a\colon A \qquad \Gamma,\ a\colon A \quad b\colon B}{\Gamma \quad a, b\ _{\Sigma a\colon A.B}\ :\ \Sigma a\colon A.B}$$

$$\frac{\Gamma \quad x\colon \Sigma a\colon A.B}{\Gamma \quad \pi_1 x\colon A} \qquad\qquad \frac{\Gamma \quad x\colon \Sigma a\colon A.B}{\Gamma \quad \pi_2 x\colon B[\pi_1 x/a]}$$

Note que, devido a possíveis ambiguidades, a introdução de pares dependentes costuma ser anotada com o tipo desejado; isso será ilustrado na Seção 3.1.

Através disso é possível, por exemplo, representar uma tupla de um número e um vetor cujo tamanho é esse número através do tipo $\Sigma n\colon \mathsf{N}.\mathtt{Vec}\ n$. Tipos de somas também podem ser usados para criar subconjuntos de tipos, definindo o segundo elemento como uma proposição (conforme será descrito na Seção 2.3). Por exemplo, o tipo $\Sigma n\colon \mathsf{N}.\mathtt{IsOdd}\ n$ é o tipo que representa os números naturais ímpares (contendo um número, n, e uma prova de que esse número é ímpar), semanticamente representando um subconjunto dos números naturais.

De forma similar aos tipos Π, quando o segundo elemento do par não depende do primeiro, a notação $A \times B$ (como em tipos de produtos convencionais) pode ser usada para o tipo $\Sigma a\colon A.B$. Além disso, como é possível usá-los para representar subconjuntos, é comum o uso da notação $\{\ a\colon A \mid \mathsf{P}\ a\ \}$ ao invés de $\Sigma a\colon A.\mathsf{P}\ a$ para representar tipos Σ quando o segundo elemento é uma proposição referente ao primeiro elemento.

Deve-se notar que os tipos Σ descritos acima são classificados como pequenos (do inglês, *small sigma type*), pois ambos os elementos usados na construção são tipos dentro do sistema (tendo *kind*). Em contra-partida, alguns sistemas de tipos podem incluir tipos Σ grandes como ΣT: $.T$, que representa um par de um tipo arbitrário e um elemento desse tipo. Entretanto, caso tal tipo ΣT: $.T$ tenha *kind* , tal que ele possa ser usado como primeiro elemento para ele mesmo, o sistema de tipos se torna inconsistente no cálculo de construções (BOWMAN et al., 2018). Tipos Σ grandes são componentes importantes de alguns sistemas de tipos, como por exemplo no sistema de módulos de Standard ML e OCaml.

Algoritmos de checagem de tipos para sistemas com tipos dependentes se tornam consideravelmente mais complexos (JIA et al., 2010), devido ao problema de equivalência de termos, o que pode custar sua decidibilidade. Existe a possibilidade de, ao se comparar a igualdade de dois tipos, termos a necessidade de compararmos dois termos dos quais os tipos dependem.

2.2.5 Equivalência de Termos

Um dos pontos mais importantes para o *design* de qualquer sistemas de tipos é a equivalência de tipos: o algoritmo de checagem de tipos precisa verificar os tipos fornecidos em certos pontos de um programa, e os tipos que eram esperados naquela posição. Para sistemas de tipos dependentes, a equivalência de tipos passa a depender da equivalência de termos, visto que os mesmos podem estar presentes em tipos (JIA et al., 2010). Nota-se, portanto, que a decidibilidade da equivalência de tipos, e consequentemente do algoritmo de checagem de tipos, depende totalmente da decidibilidade da equivalência de termos.

Em muitas linguagens, como no caso do cálculo de construções, apresentado na Seção 2.4.1, tal equivalência é feita a partir da normalização de termos: assumindo uma linguagem funcional total, termos presentes no sistema de tipos são, então, reduzidos até não apresentarem mais nenhum redex, e as formas normais resultantes são comparadas por α-equivalência (PIERCE, 2005; JIA et al., 2010).

Essa, entretanto, não é a única possível noção de equivalência para termos em uma linguagem (JIA et al., 2010). Algumas linguagens, como por exemplo Idris, permitem o uso de tipos dependentes para funções que usam recursão geral, limitando assim quaisquer reduções durante o algoritmo de checagem de tipos apenas a sub-termos aos quais é possível se provar terminação (BRADY, 2013); a linguagem Zombie (SJÖBERG; WEIRICH, 2015) requer que o programador explicitamente decida em que pontos um termo deve reduzir durante a checagem de tipos. Em alguns casos, como o da linguagem Cayenne, o algoritmo de checagem de tipos é propositadamente indecidível, sendo então responsabilidade do programador fazer com que

tipos dependam apenas de termos cuja redução eventualmente termina (AUGUSTSSON, 1998).

2.3 CORRESPONDÊNCIA DE CURRY-HOWARD

A correspondência de Curry-Howard, resultante das observações iniciais de Haskell Curry e de William Alvin Howard, e atualmente estendida por vários autores, é uma relação existente entre alguns sistemas de lógica matemática e cálculos computacionais: tipos podem ser interpretados como teoremas, e programas podem ser interpretados como provas. Com isso, programas em sistemas de tipos específicos passam a ter, além de suas propriedades computacionais, propriedades lógicas.

Algumas das relações encontradas podem ser descritas conforme abaixo:

- Implicações são equivalentes a funções: tipos de funções, como $A \to B$, significam que, tendo um objeto de tipo A, podemos construir um objeto de tipo B; logo, tal função equivale a uma implicação lógica de que "se A, então B" (e, por consequência, parâmetros são equivalentes a suposições). Tal equivalência se extende para tipos Π, que são equivalentes ao quantificador "para todos" ( ) na lógica clássica e intuicionista. Ou seja, Πa: $A.B$ representa A, B.

- *Modus ponens* é equivalente a aplicações: em sistemas de lógica, o *modus ponens* diz que "caso A implique B, e A seja verdadeiro, então B deve ser verdadeiro", o que equivale a aplicação de uma função f de tipo $A \to B$ a um objeto do tipo A (resultando em um objeto do tipo B). Por exemplo:

$$\frac{\Gamma \quad A \quad B \qquad \Gamma \quad A}{\Gamma \quad B} \qquad\qquad \frac{\Gamma \quad f\colon A \to B \qquad \Gamma \quad a\colon A}{\Gamma \quad f\ a : B}$$

- Conjunções e disjunções são equivalentes a tipos de produtos e tipos de soma, respectivamente: uma conjunção lógica nos diz que ambos teoremas são verdadeiros, e uma disjunção lógica nos diz que ao menos um dos dois teoremas são verdadeiros; isso equivale a produtos (temos um objeto contendo uma prova de cada teorema) e a somas (temos um objeto que contém a prova de um dos dois teoremas).

- Existenciais são equivalentes a pares dependentes: de forma dual aos tipos Π, tipos Σ são equivalentes ao quantificador "existe" ( ) na lógica. Computacionalmente ele representa um elemento específico da proposição A tal que foi possível construir (ou provar) B. Ou seja, Σa: $A.B$ representa A, B.

- Falsidade é equivalente a *bottom types*: um teorema que é falso (e, por consequência, não pode ser provado) é equivalente a um *bottom type* (represen-

tado por 0 ou), um tipo sem construtores, do qual nenhuma instância pode ser construída. Como um programa fechado de tal tipo não deve existir, tal teorema não pode ser provado em um sistema consistente.

Essa relação é justificada em linguagens totais, onde toda execução precisa, eventualmente, terminar. Isso é necessário pois, caso contrário, uma função que nunca termina poderia oferecer uma prova para qualquer teorema (chamando a si mesma, recursivamente, e jamais retornando), o que tornaria a lógica inconsistente, explicando o exemplo do paradoxo do barbeiro fornecido na Seção 2.2. Também nota-se que não devem existir valores válidos para todos os tipos (como `null` em algumas linguagens de programação), pois tal permitiria o uso de teoremas absurdos.

A lógica intuicionista (também chamada de lógica construtivista) é uma restrição da lógica clássica, onde provas para teoremas precisam ser construídas. Enquanto na lógica clássica assumimos apenas dois valores possíveis para um teorema (verdadeiro ou falso, a chamada lei do terceiro excluído), a lógica intuicionista cria instâncias de uma prova como evidência da mesma, e teoremas são considerados verdadeiros quando tal evidência pôde ser construída. Para se provar que algo é falso, então, é necessário se provar que tal evidência não pode ser construída (a negação intuicionista define $\neg A$ como $A \rightarrow$). A prova, então, é equivalente a um programa que constrói tal evidência.

Nota-se, entretanto, que, conforme demonstrado por Wadler em (WADLER, 2003), a lei do terceiro excluído também possui um significado computacional sob a correspondência de Curry-Howard: continuações de primeira classe. Uma prova que utilize lógica clássica, ao ser tratada como um programa, possui a habilidade de “voltar no tempo”, retornando duas vezes do mesmo lugar, através de um operador clássico. Tal operador existe na linguagem de programação Scheme como a função `call/cc`, que possui o tipo $((A \rightarrow B) \rightarrow A) \rightarrow A$, tipo equivalente à lei de Peirce, com a qual é possível se derivar a lei do terceiro excluído mesmo dentro da lógica intuicionista.

Alguns sistemas de tipos em particular, como os sistemas de tipos puros presentes dentro do cubo lambda, abordado a seguir, possuem uma grande relação com sistemas de lógica intuicionista.

2.4 SISTEMAS DE TIPOS PUROS

Sistemas de tipos puros (do inglês, *pure type systems*) são um formalismo apresentado para a representação de alguns sistemas de tipos para o cálculo lambda em estilo Church, seguindo um conjunto de regras comuns (PIERCE, 2005). Vários sistemas de tipos podem então ser descritos através desse conjunto de regras, em particular os sistemas do cubo lambda (descrito a seguir).

Sistemas de tipos puros são parametrizados por três constantes: S (*sorts*), A (*axioms*), e R (*rules*). O parâmetro S define os *sorts* existentes no sistema de tipos. O parâmetro A define axiomas para o sistema, que representam noções de tipagem para *sorts*. Finalmente, o parâmetro R define regras que permitem a derivação dos tipos de funções dentro do cálculo lambda. As regras de sistemas de tipos são descritos na Figura 2, e sua sintaxe é dada a seguir:

$$e ::= s \mid x \mid \lambda x{:}\ e.e \mid \Pi x{:}\ e.e \mid e\ e$$

As variações dentre sistemas de tipos puros se encontram nas regras (1) e (3) dos mesmos. A regra (1) simplesmente afirma, sem pré-condições, um *sort* s_1 possui tipo s_2, caso a 2-tupla (s_1, s_2) exista no parâmetro A (e, portanto, sistemas de tipos puros diferentes podem conter axiomas diferentes). A regra (3) representa o ponto central dos sistemas de tipos puros: que tipos Π podemos construir dentro do sistema. O parâmetro R contém 3-tuplas cujos valores, (s_1, s_2, s_3), representam respectivamente um conjunto válido de dominio (tipo de parâmetro), codomínio (tipo de retorno), e *kind* para tipos Π.

Na regra (4) temos a introdução de funções no sistema. Caso seja derivável que $\Pi A{:}\ A.B$ é um tipo válido dentro do sistema, através da regra (3), e que, no mesmo contexto, com a adição de uma variável $a{:}\ A$, o termo b seja derivável e tenha tipo B, então, nesse contexto, podemos derivar uma função $\lambda a{:}\ A.B$, cujo tipo é $\Pi A{:}\ A.B$. A variável $a{:}\ A$, então, é o parâmetro da função.

Figura 2 – Regras de checagem de tipos para sistemas de tipos puros

$\boxed{\Gamma \quad e{:}\ e}$

$$(1)\ \frac{}{\Gamma \quad s_1{:}\ s_2}\ \text{se } (s_1, s_2) \quad A \qquad\qquad (2)\ \frac{}{\Gamma,\ x{:}\ T \quad x{:}\ T}$$

$$(3)\ \frac{\Gamma \quad A{:}\ s_1 \qquad \Gamma,\ a{:}\ A \quad B{:}\ s_2}{\Gamma \quad \Pi a{:}\ A.B\ :\ s_3}\ \text{se } (s_1, s_2, s_3) \quad R$$

$$(4)\ \frac{\Gamma,\ a{:}\ A \quad b{:}\ B \qquad \Gamma \quad \Pi x{:}\ A.B\ :\ s}{\lambda a{:}\ A.b\ :\ \Pi a{:}\ A.B}$$

$$(5)\ \frac{\Gamma \quad f{:}\ \Pi a{:}\ A.B \qquad \Gamma \quad x{:}\ A}{\Gamma \quad f\ x\ :\ B[x/a]}$$

Fonte: o autor

As regras (2) e (5) apresentam verificação de variáveis e aplicação de funções, respectivamente, e são similares às regras apresentadas previamente.

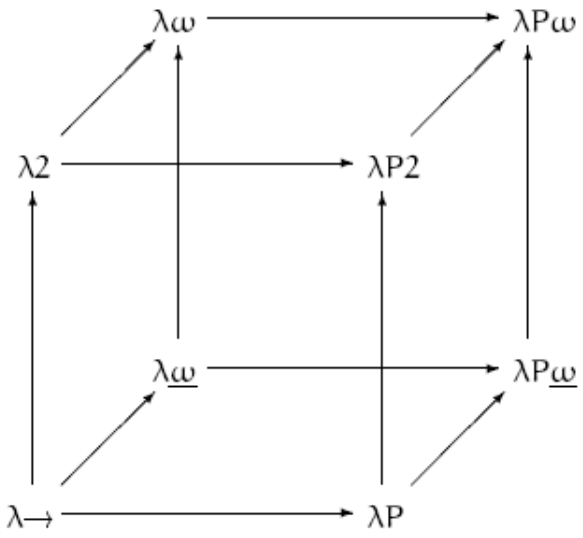
Alguns sistemas de tipos, como o cálculo lambda simplesmente tipado e o Sistema F_ω (usado como base para os sistemas de algumas linguagens como OCaml, Standard ML e Haskell) são sistemas de tipos puros e puderam ser generalizados no cubo lambda (visto na Figura 3), onde cada eixo representa um valor adicional no parâmetro R do sistema de tipos. Todos os oito sistemas do cubo possuem apenas dois *sorts*, S = *{ , }*, e um único axioma, A = *{(,)}*. Portanto a regra (1) dos sistemas de tipos puros pode ser simplificada para o cubo lambda como:

$$\frac{}{\Gamma \quad :}$$

Os sistemas do cubo variam de acordo com o parâmetor R, especificando os tipos de abstrações possíveis, conforme ilustrado na Tabela 1.

Todos os sistemas do cubo possuem uma regra em comum: (, ,); isso significa que todos os sistemas podem fazer funções que dependem de valores e que retornam valores. O sistema $\lambda\rightarrow$, contendo apenas essa regra, é o cálculo lambda simplesmente tipado. Os três eixos adicionam abstrações possíveis ao sistema de tipos.

Figura 3 – O cubo lambda



Fonte: Wikipédia, a Enciclopédia Livre™

Tabela 1 – Sistemas de tipos do cubo lambda

Sistema	**Regras**	(topo)	(direita)	(trás)
$\lambda\!\rightarrow$	$R = \{(\ ,\ ,\ )\}$			
$\lambda 2$	$R = \{(\ ,\ ,\ )$	$(\ ,\ ,\ )\}$		
λP	$R = \{(\ ,\ ,\ )$		$(\ ,\ ,\ )\}$	
$\lambda\underline{\omega}$	$R = \{(\ ,\ ,\ )$			$(\ ,\ ,\ )\}$
$\lambda P2$	$R = \{(\ ,\ ,\ )$	$(\ ,\ ,\ )$	$(\ ,\ ,\ )\}$	
$\lambda\omega$	$R = \{(\ ,\ ,\ )$	$(\ ,\ ,\ )$		$(\ ,\ ,\ )\}$
$\lambda P\underline{\omega}$	$R = \{(\ ,\ ,\ )$		$(\ ,\ ,\ )$	$(\ ,\ ,\ )\}$
$\lambda P\omega$	$R = \{(\ ,\ ,\ )$	$(\ ,\ ,\ )$	$(\ ,\ ,\ )$	$(\ ,\ ,\ )\}$

Fonte: o autor

- O lado traseiro introduz operadores de tipos: o valor $(\ ,\ ,\ )$ é adicionado a R, permitindo a criação de tipos genéricos que dependem de outros tipos.
- O lado superior introduz funções polimórficas: o valor $(\ ,\ ,\ )$ é adicionado a R, permitindo que uma função possa ser executada para qualquer tipo.
- O lado direito introduz tipos dependentes: o valor $(\ ,\ ,\ )$ é adicionado a R, permitindo o uso de tipos dependentes (isto é, tipos que dependem de valores).

Assim, temos que o cálculo lambda simplesmente tipado é o sistema mais simples, e o cálculo de construções (representado no cubo como $\lambda P\omega$) é o mais completo (do qual todos os outros são subconjuntos). Nota-se que o cálculo de construções é fortemente normalizável (isto é, todas as reduções bem-tipadas eventualmente terminam), e, por consequência, todos os subsistemas do cubo também são. Particularmente, a igualdade de termos (e tipos) no sistema é dada quando dois termos são idênticos (ignorando α-reduções) em sua forma normal.

Formalmente, podemos então definir que:

$$\frac{\Gamma \quad a\colon A \qquad A \rightarrow \ B}{\Gamma \quad a\colon B}$$

Ou seja, “se a tiver tipo A, e A reduzir em um número arbitrário de passos para B, então temos que a também tem tipo B”, demonstrando a equivalência de A e B.

2.4.1 O Cálculo de Construções

O CoC é o mais geral dos oito cálculos do cubo, e a base do assistente de provas Coq (PIERCE, 2005).

É possível considerarmos o CoC isoladamente ao fixarmos os 4 tipos possíveis de abstrações (e, consequentemente, de aplicações) do cubo lambda, definindo

uma versão estratificada do cálculo de construções (BARTHE; UUSTALU, 2002), especializando as regras dos sistemas de tipos puros para o uso específico do CoC. As regras estratificadas podem ser vistas na Figura 4, e sua sintaxe pode ser dada a seguir:

$$A ::= \quad | \; \Pi X\colon A.A \; | \; \Pi x\colon T.A$$

$$T ::= X \; | \; \lambda X\colon A.T \; | \; \lambda x\colon T.T \; | \; T \; T \; | \; T \; e \; | \; \Pi X\colon A.T \; | \; \Pi x\colon T.T$$

$$e ::= x \; | \; \lambda X\colon A.e \; | \; \lambda x\colon T.e \; | \; e \; T \; | \; e \; e$$

Na sintaxe acima, a meta-variável X representa variáveis de tipos, e a meta-variável x, como anteriormente, representa variáveis de termos. Note que não consideramos como parte da linguagem, visto que esse termo não possui um tipo no sistema. O contexto passa a ser dado pela seguinte gramática, podendo ligar variáveis de tipos e variáveis de termos:

$$\Gamma ::= \cdot \; | \; \Gamma,\; X\colon A \; | \; \Gamma,\; x\colon T$$

O CoC é, de acordo com a correspondência de Curry-Howard, diretamente equivalente a um subconjunto da lógica intuicionista, representado dentro do cálculo lambda com apenas um construtor de tipos (nominalmente, pi). Teoremas, então, são descritos como tipos impredicativos, isto é, tipos cuja definição polimórfica abranje todos os teoremas que podem ser descritos no sistema, inclusive a si mesmos. Por exemplo, o tipo dos números naturais pode ser descrito em CoC como:

$$\mathsf{N} \equiv \Pi T\colon \quad .\Pi z\colon T.\Pi s\colon T \to T.T$$

Que pode ser lido como: “para todo teorema T, se tendo uma prova de T, se tendo uma prova de que com T podemos provar T, podemos provar de T”. Sabemos, graças às regras do sistema, que tal tipo possui *sort* (e, portanto, poderia ser passado como primeiro argumento para uma função de tal tipo).

Para provar tal teorema (isto é, representar números dentro do sistema), apenas é preciso criar objetos com o tipo acima. Neste caso, temos que zero, de tipo N, pode ser definido como:

$$\lambda T\colon \quad .\lambda z\colon T.\lambda s\colon T \to T.z$$

E o sucessor, de tipo N → N, pode ser definido como:

$$\lambda n\colon \mathsf{N}.\lambda T\colon \quad .\lambda z\colon T.\lambda s\colon T \to T.s \; (n \; T \; z \; s)$$

O CoC por si só, embora útil por sua pequena definição formal, não atende alguns requisitos importantes para seu uso como lógica. Por exemplo, sabe-se que

Figura 4 – Regras de checagem de tipos para o cálculo de construções

$\boxed{\Gamma \quad A: \quad}$

$$(1)\ \frac{}{\Gamma \quad :}$$

$$(2)\ \frac{\Gamma \quad A: \qquad \Gamma,\ X: A \quad B:}{\Gamma \quad \Pi X: A.B :} \qquad (3)\ \frac{\Gamma \quad T: \qquad \Gamma,\ x: T \quad B:}{\Gamma \quad \Pi x: T.B :}$$

$\boxed{\Gamma \quad T: A}$

$$(4)\ \frac{}{\Gamma,\ X: A \quad X: A}$$

$$(5)\ \frac{\Gamma \quad A: \qquad \Gamma,\ X: A \quad T: B}{\Gamma \quad \lambda X: A.T : \Pi X: A.B} \qquad (6)\ \frac{\Gamma \quad T: \qquad \Gamma,\ x: T \quad U: B}{\Gamma \quad \lambda x: T.U : \Pi x: T.B}$$

$$(7)\ \frac{\Gamma \quad T: \Pi X: A.B \qquad \Gamma \quad U: A}{\Gamma \quad T\ U : B[U/X]} \qquad (8)\ \frac{\Gamma \quad T: \Pi x: U.B \qquad \Gamma \quad e: U}{\Gamma \quad T\ e : B[e/x]}$$

$$(9)\ \frac{\Gamma \quad A: \qquad \Gamma,\ X: A \quad T:}{\Gamma \quad \Pi X: A.T :} \qquad (10)\ \frac{\Gamma \quad T: \qquad \Gamma,\ x: T \quad U:}{\Gamma \quad \Pi x: T.U :}$$

$\boxed{\Gamma \quad e: T}$

$$(11)\ \frac{}{\Gamma,\ x: T \quad x: T}$$

$$(12)\ \frac{\Gamma \quad A: \qquad \Gamma,\ X: A \quad e: T}{\Gamma \quad \lambda X: A.e : \Pi X: A.T} \qquad (13)\ \frac{\Gamma \quad T: \qquad \Gamma,\ x: T \quad e: U}{\Gamma \quad \lambda x: T.e : \Pi x: T.U}$$

$$(14)\ \frac{\Gamma \quad f: \Pi X: A.T \qquad \Gamma \quad U: A}{\Gamma \quad f\ U : T[U/X]} \qquad (15)\ \frac{\Gamma \quad f: \Pi x: T.U \qquad \Gamma \quad e: T}{\Gamma \quad f\ e : U[e/x]}$$

Fonte: o autor

não é possível se provar que $0 = 1$ usando a codificação impredicativa acima, além da ausência dos princípios indutivo e coindutivo (BARTHE; UUSTALU, 2002).

Outra limitação conhecida no CoC é a impossibilidade de se definir eliminações dependentes. Por exemplo, embora tipos Π sejam o único construtor existente no sistema, tipos Σ podem ser representados de forma impredicativa:

$$\Sigma a: A.B \equiv \Pi R: \quad .(\Pi a: A.B \to R) \to R$$

Em codificações similares à acima, o tipo R é chamado de tipo de resposta.

É trivial se provar que tal representação, que representa um tipo sigma forte, é isomórfica a tipos Σ fortes (primitivos). A projeção fraca, então, elimina ambos subtermos do par simultaneamente (ao invés das projeções fortes π_1 e π_2 vistas na Seção 2.2.4). Nominalmente:

$$\texttt{let}\ \ a,\ b\ \ = x\ \texttt{in}\ e \equiv x\ R\ (\lambda a\colon A.\lambda b\colon B.e)$$

Onde R é o tipo do termo e, e o termo x tem tipo $\Sigma a\colon A.B$.

Assim, é possível se escrever provas existenciais no CoC; o problema surge quando se deseja extrair os elementos do par. É possível definir a projeção forte do primeiro elemento (π_1) de forma simples:

$$\lambda p\colon (\Sigma a{:}A.B).p\ A\ (\lambda a\colon A.\lambda b : B.a)$$

Ou, com a notação introduzida acima:

$$\texttt{let}\ \ a,\ b\ \ = x\ \texttt{in}\ a$$

Não é, entretanto, possível definir a projeção forte do segundo elemento (π_2) caso ele dependa do primeiro (isto é, caso a seja livre em B) a partir da projeção fraca (BARTHE; UUSTALU, 2002). Nota-se, também, pela definição, que tipos Σ grandes, embora possam ser construídos, não podem ser eliminados de nenhuma forma (fato que tornaria o CoC inconsistente; isso se torna simples de observar no caso do tipo sigma fraco, pois, para eliminar o primeiro elemento, seria necessário instanciar R acima com , resultando em : , uma conhecida fonte de inconsistência).

O assistente de provas Coq resolve esses problemas através do cálculo de construções (co)indutivas (do inglês, *calculus of (co)inductive constructions*, doravante CIC), onde o CoC é estendido com tipos indutivos e coindutivos, além de uma hierarquia predicativa infinita de universos (*sorts*). Tipos indutivos e coindutivos, como o caso do tipo sigma[16], podem ser codificados no CoC, mas sofrem dos mesmos problemas.

2.5 O CÁLCULO LAMBDA LINEAR

Sistemas de tipos lineares apresentam uma lógica de recursos, e são baseados na lógica linear de Girard (PIERCE, 2005). O principal uso de um sistema de tipos lineares é a restrição do uso de um objeto que é tido como um recurso: uma variável

[16] Na realidade, tipos Σ podem ser descritos como tipos indutivos, e, de fato, isso é feito no assistente de provas Coq, ao invés de existir como um construtor primitivo.

linear só pode, e deve, ser usada uma única vez. Tal limitação é desejável para implementação segura de alguns protocolos computacionais, tendo sido incorporado em linguagens como Clean e, mais recentemente, como uma extensão para a linguagem Haskell (BERNARDY et al., 2017).

Um dos principais elementos da lógica linear é a implicação linear, representada pelo símbolo (, também chamado de “operador pirulito” (do inglês, *lollipop operator*). Uma implicação linear T (U é similar a uma implicação intuicionista, porém impõe uma restrição de linearidade em T: uma função f: T (U cria uma prova de U a partir de uma prova de T, porém sua suposição T é usada exatamente uma única vez.

Por exemplo, os combinadores S e K, termos lambda conhecidos (tendo sua origem na lógica de combinadores), podem ser representados da seguinte forma no cálculo lambda:

$$S = \lambda x.\lambda y.\lambda z.x\ z\ (y\ z)$$

$$K = \lambda x.\lambda y.x$$

Seus tipos são, respectivamente, $(A \to B \to C) \to (A \to B) \to A \to C$, e $A \to B \to A$. Tais termos não podem ser termos lineares; se considerássemos suas implicações como lineares, isto é, com tipos (A (B (C) ((A (B) (A (C e A (B (A, estes se tornariam inválidos: em S, o parâmetro z é duplicado, e, em K, o parâmetro y é descartado. Em contrapartida, os combinadores B, C e I podem ser representados em um sistema de tipos linear. Os termos são definidos como:

$$B = \lambda x.\lambda y.\lambda z.x\ (y\ z)$$

$$C = \lambda x.\lambda y.\lambda z.x\ z\ y$$

$$I = \lambda x.x$$

Estes termos, com tipos, respectivamente, $(B \to C) \to (A \to B) \to A \to C$, $(A \to B \to C) \to B \to A \to C$ e $A \to A$, podem ser representados como teoremas em uma lógica linear: cada um de seus parâmetros é usado exatamente uma vez.

Como limitar variáveis a um único uso é uma restrição muito forte, programas em sistemas de tipos totalmente lineares são inexpressivos, e incapazes de representar uma grande quantidade de programas desejáveis (WADLER, 1990). Devido a isso, variantes do cálculo lambda com tipos lineares os unem a tipos intuicionistas: alguns termos são lineares, e devem ser usados uma única vez, enquanto alguns termos são irrestritos, e podem ser usados um número arbitrário de vezes (PIERCE, 2005). Tal restrição é representada pelo sistema de tipos.

A lógica linear, tradicionalmente, não apresenta implicações tradicionais, onde um parâmetro (ou melhor, uma suposição) possa ser usado qualquer número de vezes. Existe, entretanto, o operador exponencial, representado como ! (lê-se "naturalmente", ou “*bang*”), que permite a cópia e o descarte de suposições. Assim tem-se que o teorema $T \rightarrow U$ pode ser codificado como $!T \multimap U$ (BERNARDY et al., 2017). A implicação intuicionista, entretanto, costuma ser usada como primitiva em sistemas de tipos lineares (que então não usam o operador exponencial).

Note que, de acordo com a representação acima, tal como foi usada na extensão Linear Haskell, não faz uma diferença essencial entre tipos lineares e tipos intuicionistas, a diferença existindo apenas no tipo de seta usada para representar o tipo de uma função. Nesse caso, onde $f\colon T \multimap U$, temos que x (onde $x\colon T$) só será usado uma única vez se, e apenas se, $f\ x$ for usado uma única vez (BERNARDY et al., 2017). Isto é, a linearidade é restringida ao corpo da função tipada com a implicação linear, e um parâmetro fornecido para uma função linear não precisa ser um termo linear no contexto onde a função é chamada. Como consequência, não é possível forçar a linearidade do retorno de uma função (BERNARDY et al., 2017).

Essa, entretanto, não é a única forma de se considerar a implicação intuicionista. Alguns sistemas de tipos, como apresentado em (WADLER, 1990) e em (PIERCE, 2005), consideram o tipo $T \rightarrow U$ como sendo equivalente a $!(T \multimap U)$ na lógica linear: o significado da implicação linear, então, é alterado; ao invés de tornar o parâmetro linear dentro do contexto da função apenas, a função passa a ser um termo linear em si, e o que passa a definir se um parâmetro é ou não linear é seu tipo (havendo, assim, uma distinção natural entre tipos lineares e tipos intuicionistas, permitindo, por exemplo, o retorno de um termo linear de uma função). Tal representação também pode ser chamada de *uniqueness typing*; essa representação, embora mais distante da lógica linear original, tem utilidades para programação (BERNARDY et al., 2017).

A variante *uniqueness typing*, usada por exemplo na linguagem de programação Clean, é particularmente útil para programas puramente funcionais. Como sabemos que um termo será universalmente linear, sabemos também que, quando executado por um computador, existirá apenas uma referência na memória para o objeto por vez (WADLER, 1990). Funções que transformam termos lineares, então, são livres para alterar o termo *in-place*, oferecendo, efetivamente, mutabilidade para linguagens puramente funcionais, como por exemplo a alteração de vetores de forma eficiente (WADLER, 1990). A ideia de usar tipos lineares para representar mutabilidade em programas puramente funcionais não é incomum na literatura, e apresenta algumas similaridades à lógica de Hoare (KRISHNASWAMI; PRADIC; BENTON, 2015).

Nessa representação, uma função apenas precisará ser linear caso use uma variável linear de seu contexto. A intuição por trás dessa restrição é simples: caso uma função convencional, intuicionista, possa ser feita contendo termos lineares dentro de si, ao ser descartada ou duplicada, consequentemente os termos lineares contidos em seu corpo também seriam (WADLER, 1990). A mesma lógica se aplica, consequentemente, a pares: caso um par contenha um subtermo linear, o par necessariamente precisa ser linear para se evitar o descarte ou duplicação da informação (PIERCE, 2005).

Um fato notável é que ambas as representações para a implicação intuicionista na lógica linear, conforme descritas acima, são, na verdade, duais (BERNARDY et al., 2017). Uma técnica sugerida para garantir a existência única de um termo linear em Linear Haskell é a dupla negação: para uma função linear f: T (U não há a restrição de que seu retorno seja usado uma única vez; e, de fato, caso seja usado mais de uma vez, seu argumento linear também será usado mais de uma vez. É possível negar a função duas vezes, isto é, g: $A \rightarrow$ (B (R) (R; ao invés de retornar um tipo B ao usuário, o usuário precisa fornecer uma continuação que irá usar o tipo de retorno como desejar, contanto que o use exatamente uma única vez. O tipo R, tal como foi usado, é conhecido como tipo de resposta, que será elaborado com mais detalhes na Seção 2.6.2.

Continuações, por sua vez, são extremamente comuns no que se refere a linguagens intermediárias; funções deixam de retornar um valor, e, ao invés disso, recebem como parâmetro adicional uma função que irá receber o antes retorno como um parâmetro, efetivamente contendo o resto da computação.

2.6 CONTINUAÇÕES E CONTROLE

A fim de facilitar transformações e análise sobre código, linguagens intermediárias tornam explícitos detalhes tratados de forma implícita em linguagens fonte; em particular, o fluxo de controle, como por exemplo a pilha de chamada com a qual programadores estão habituados, é representada de forma explícita em linguagens intermediárias.

Quanto a compiladores de linguagens funcionais, algumas linguagens intermediárias foram propostas, como o cálculo lambda monádico, o cálculo lambda em forma normal administrativa (ANF), e o cálculo lambda em CPS. Embora existam argumentos a favor do uso de ANF, que usa restrições sintáticas para eliminar a necessidade de continuações explícitas (seu logo é: “pense em continuações, escreva em estilo direto”), ainda há argumentos a favor do uso de CPS diretamente (KENNEDY, 2007).

Figura 5 – Função fatorial em Scheme em estilo direto

```
1 ; Função com tipo Int → Int
2 (define (factorial n)
3   (if (= n 0)
4     1
5           ; Assume-se que a chamada irá retornar para esse ponto
6     (* n (factorial (- n 1)))))
```

Fonte: o autor

2.6.1 Estilo de Passagem de Continuação

Convencionalmente, programas são escritos no chamado estilo direto, onde o controle de fluxo da aplicação é implícito (funções podem ser chamadas e retornam para o local de onde foram chamadas, a conhecida pilha de chamada). O CPS, introduzido inicialmente na linguagem Scheme, é um estilo de programação onde o controle da aplicação é passado explicitamente: funções nunca retornam, e, por consequência, toda chamada é uma chamada de cauda.

Funções escritas em CPS recebem um parâmetro adicional: uma continuação explícita. No ponto onde a função deveria retornar, ela explicitamente chama a continuação com o valor desejado, que deve continuar a computação. O objetivo é, essencialmente, eliminar a pilha de chamadas; o compilador passa a ser livre da distinção entre funções que retornam e funções que não retornam, visto que, em estilo de continuação, nenhuma função deve retornar.

Por exemplo, a Figura 5 contém um programa na linguagem Scheme para calcular o fatorial de um número em estilo direto, e a Figura 6 contém programa similar em CPS[17]. Na versão CPS, o controle de fluxo de código é passado explicitamente; a função que inicialmente chama `factorial&`[18] informa o que fazer com o resultado da função (através do parâmetro `k`), e os resultados são “retornados” à função original através dessa continuação. O mesmo comportamento é usado pela própria função ao passar uma continuação para sua chamada recursiva.

Embora o estilo rapidamente se torne complexo e verboso para programação manual, compiladores para linguagens funcionais costumam apresentar traduções mecânicas de estilo direto para CPS. Apesar de não ser óbvio a primeira vista, linguagens de montagem são usadas em CPS ao invés de um estilo direto; quando

[17] Para esse exemplo, as funções `if`, `=`, `-` e `*` são interpretadas como primitivas, e por isso se assemelham a funções em estilo direto.

[18] O símbolo $\&$ ao fim da função não possui tratamento especial; é apenas um caracter presente no nome da função. Por convenção, funções em CPS em Scheme terminam com um *ampersand*.

Figura 6 – Função fatorial em Scheme em CPS

```
1 ; Função com tipo Int × (Int →  ) →
2 (define (factorial& n k)
3   (if (= n 0)
4     ; Chama explicitamente a continuação,
5     ; "retornando" dessa função
6     (k 1)
7                          ; Resto da computação
8     (factorial& (- n 1) (lambda (m)
9                           (k (* n m))))))
```

Fonte: o autor

uma função é chamada, a posição atual é explicitamente colocada na pilha, como um parâmetro para a função no registro de ativação. Ao retornar, a função usa o valor na pilha para determinar onde continuar a execução do programa através de um salto. Com isso, a transformação em CPS, além de conveniente para otimizações, também auxilia a aproximar um programa escrito em uma linguagem de alto nível para uma linguagem de baixo nível que poderá ser executada pelo computador.

Código em linguagens como Scheme não costuma ser escrito como na Figura 6, em CPS. O conceito de continuações, entretanto, é importante para a linguagem visto que a mesma possui continuações de primeira classe. É possível, através da função *call-with-current-continuation* (abreviada *call/cc*) chamar uma função fornecendo a continuação atual, isto é, o estado atual do programa, como parâmetro (que então pode ser armazenado em variáveis, etc). Como discutido na Seção 2.3, tal função é equivalente a um operador clássico, permitindo que uma mesma função retorne duas vezes (WADLER, 2003).

O fato de continuações existirem como cidadãos de primeira classe (podendo ser passadas como parâmetro e retornadas de funções) em linguagens como Scheme impede o uso de uma pilha de chamada tradicional. Definimos, então, o conceito de naturalidade: uma função natural, ao receber uma continuação, não irá duplicá-la ou descartá-la, mas irá usá-la para “retornar” seu resultado (THIELECKE, 2003). Funções naturais podem ser facilmente compiladas para linguagens de montagem.

Embora continuações costumem ser analisadas no contexto de compiladores, propriamente como linguagens intermediárias (FLANAGAN et al., 1993), ou no contexto de cálculos completos em estilo direto (BOWMAN et al., 2018; MAURER et al., 2017), é possível seu estudo como um cálculo por conta própria (FÜHRMANN; THIELECKE, 2004). Por exemplo, é possível definir um cálculo inteiramente baseado em continuações; sua sintaxe pode ser definida a seguir:

$$T ::= \mathsf{N} \mid \neg T \mid T \times T$$

$$e ::= x \mid n \mid \ e, \ e \ \mid \pi_1 e \mid \pi_2 e$$

$$c ::= x \ e \ \mid c \ \{ \ x \ x{:}\, T \ = c \ \}$$

Junto a isso, algumas reduções são adicionadas, conforme foram chamadas em (FÜHRMANN; THIELECKE, 2004):

- `Distr`-redução:
  $e \ \{ \ a \ b{:}\, B \ = f \ \}\{ \ c \ d{:}\, D \ = g \ \} \rightarrow \ e \ \{ \ c \ d{:}\, D \ = g \ \}\{ \ a \ b{:}\, B \ = f \ \{ \ c \ d{:}\, D \ = g \ \} \ \}$
  (onde $a = c$ e a e b não estão livres em g)
- `GC`-redução: $a \ b \ \ \{ \ c \ d{:}\, D \ = e \ \} \rightarrow a \ b$ (onde c não está livre em $a \ b$)
- `Jmp`-redução: $a \ b \ \ \{ \ a \ c{:}\, C \ = e \ \} \rightarrow e[b/c]$
- `Eta`-redução: $e \ \{ \ a \ b{:}\, B \ = c \ b \ \ \} \rightarrow e[c/a]$

Nesse cálculo, de forma muito similar a representações intermediárias propriamente usadas em compiladores, abstrações (continuações) são introduzidas através da sintaxe $a \ \{ \ b \ x{:}\, T \ = c \ \}$, onde a continuação b é ligada (tendo tipo $\neg T$) dentro do comando a (onde pode aparecer livre); seu corpo é o comando c, onde o parâmetro x é ligado. O cálculo apresentado possui uma grande similaridade ao cálculo pi, usado para o estudo de processos (FÜHRMANN; THIELECKE, 2004).

Uma tradução em CPS, algoritmo capaz de transformar um termo lambda em estilo direto em um termo lambda em CPS, costuma ser usado por compiladores para a tradução da linguagem fonte para a linguagem intermediária. Nota-se também que o mesmo tipo de tradução pode ser usado para traduzir o cálculo lambda para um termo equivalente no cálculo pi (FÜHRMANN; THIELECKE, 2004).

2.6.2 Tradução em Passagem de Continuação

Traduções em passagem de continuação (FLANAGAN et al., 1993) são algoritmos que recebem um programa em estilo direto e, mecanicamente, o convertem para um programa em CPS; tais tipos de traduções são comuns em compiladores.

Na correspondência de Curry-Howard, a transformação de programas em estilo direto para programas em CPS é equivalente à introdução de negação dupla (WADLER, 2003): a transformação é válida em uma lógica intuicionista, e a função transformada contém o mesmo valor lógico da original. Por exemplo, uma função em estilo direto de tipo $A \rightarrow B$ pode ser mecanicamente transformada em uma função em CPS

Figura 7 – Prova de tradução CPS para funções

```
1  Theorem cps:
2       T U: Prop,
3       f: T →  U,
4    ¬(T ×  ¬U).
5  Proof.
6    intros T U f g.
7    destruct g.
8    apply n; apply f.
9    assumption.
10 Qed.
```

Fonte: o autor

de tipo $A \times (B \rightarrow \quad) \rightarrow \quad$. A interpretação da conversão é trivial: no formato original, temos que “se A, então podemos construir B”, e na versão em CPS, temos que “não é possível termos uma prova de A e, ao mesmo tempo, temos uma prova de que B é impossível”. Tal afirmação é um teorema intuicionista e pode ser trivialmente provado, conforme visto na Figura 7 (escrita na linguagem do assistente de provas Coq).

O tipo , conhecido como o tipo de resposta (do inglês, *answer type*) é o tipo resultante de uma computação. Pela sua proximidade com a lógica, é possível representar o tipo de resposta como o tipo 0, isto é, um tipo que não pode ser construído (e, se fosse, seria um absurdo), tendo assim, dentro da correspondência de Curry-Howard, o significado de uma prova duplamente negada (FÜHRMANN; THIELECKE, 2004), caso a prova por completa esteja em CPS. A única necessidade, entretanto, é que ele não apareça livre nos parâmetros, sendo limitado apenas, efetivamente, à resposta. Um tipo arbitrário poderia ser usado para, efetivamente, retornar o resultado final de uma computação em CPS, o que pode ser útil em ambientes onde provas em CPS são usadas em conjunto com provas convencionais, em estilo direto (THIELECKE, 2003).

Embora conversões para CPS já existam há bastante tempo, ainda não foram completamente estudadas em linguagens dependentemente tipadas. Em (BARTHE; HATCLIFF; SØRENSEN, 1999) é apresentada uma tradução CPS para um subconjunto de sistemas de tipos puro, incluindo todos os cálculos do cubo lambda, incluindo o cálculo de construções, capaz de corretamente traduzir tipos Π. Em (BARTHE; UUSTALU, 2002), entretanto, os autores notam que que, aplicando-se a mesma metodologia, não é possível traduzir tipos Σ fortes para CPS. Nota-se em (BOWMAN et al., 2018) que, para tipos Σ, sistemas de tipos não convencionais se fazem necessários, onde algumas regras são adicionadas a fim de garantir a preservação de tipos.

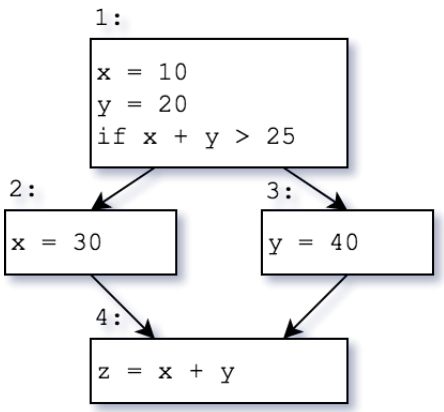
2.6.3 *Static Single Assignment Form*

O *static single assignment form* (doravante SSA), desenvolvido por pesquisadores da IBM nos anos 80 (CYTRON et al., 1991), é uma linguagem funcional (APPEL, 1998) visual[19] comum em compiladores para linguagens imperativas. Devido ao uso irrestrito de estado, linguagens imperativas costumam ser difíceis de se otimizar; uma técnica comum em compiladores então é a conversão do código imperativo para um grafo de fluxo de controle (doravante CFG), que divide a função em chamados blocos básicos, e que contém a transferência de controle durante a execução do programa. Compiladores de nível industrial, como o GCC e o Clang, fazem uso desse formato.

O CFG representa o fluxo do programa: comandos estruturados como condicionais e *loops* são separados em unidades chamadas de blocos básicos (do inglês, *basic blocks*), que representam uma série de definições de variáveis seguidas por um desvio de controle. Por exemplo, considere a seguinte função em pseudo-código:

```
1 x = 10
2 y = 20
3
4 if x + y > 25:
5     x = 30
6 else:
7     y = 40
8
9 z =  x + y
```

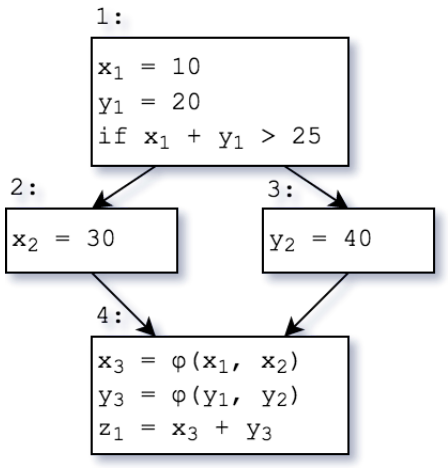
Ela pode ser representada através de um CFG da seguinte maneira:



[19] Uma linguagem de programação visual é uma linguagem cuja notação primária se dá através de elementos gráficos além de, ou ao invés de, texto corrido.

Cada bloco básico é composto de uma série de atribuições seguidas de um desvio de fluxo para um ou dois[20] blocos básicos. No exemplo acima, temos os blocos básicos 1, 2, 3 e 4. As variáveis x e y são definidas inicialmente no bloco 1, porém são alteradas, respectivamente, nos blocks 2 e 3. O bloco 4, que representa um ponto de união, representa o código alcançado pelo programa em pseudo-código após a execução da condicional: é possível alcançar o bloco 4 por dois caminhos diferentes (nominalmente, blocos 2 e 3). No bloco 4, então, não temos informações estáticas sobre os valores das variáveis x e y, visto que as mesmas podem ter sido alteradas dependendo do fluxo de execução.

Para contornar esse problema, e termos informações sobre o fluxo de dados no programa, a linguagem SSA/CFG primeiro renomeia as variáveis originais para garantir que cada variável seja definida apenas uma única vez, proibindo possíveis alterações. Pontos de união, como o bloco 4, por serem acessíveis por mais de um bloco de origem, precisam de informação dos possíveis valores de suas variáveis, inseridos em funções ϕ. O equivalente ao programa acima, em SSA/CFG, é demonstrado:



Funções ϕ (originalmente batizadas como *phony functions*) definem possíveis valores para uma variável em blocos básicos que podem ter mais de uma origem; os "parâmetros" para as funções ϕ são variáveis existentes em blocos básicos anteriores no grafo de controle, um para cada possível bloco básico de origem. Neste exemplo, caso o fluxo de execução do algoritmo para o bloco 4 venha do bloco 2, as variáveis x_3 e y_3 terão valores x_1 e y_1, respectivamente. Similarmente, caso o fluxo venha do bloco 3, os valores de x_3 e y_3 serão x_2 e y_2, respectivamente.

Em um programa SSA/CPS, há um conceito importante chamado de dominância. Dizemos que um bloco básico A domina outro bloco básico B quando todos os caminhos no fluxo de controle do programa, durante a execução, precisem passar

[20] Ou possivelmente mais, em algumas variantes, a fim de se considerar a estrutura `switch` de controle.

por A para chegar a B. Por consequência, todos os blocos básicos de uma função são dominadas por sua entrada (nesse caso, 1): nota-se que 1 domina 2, 3 e 4 pois todos os caminhos para tais passam por 1, mas nem 2 e nem 3 dominam 4.

A dominância pode ser calculada através da análise do fluxo do programa, e pode ser usada para minimizar o uso de funções ϕ onde elas não são necessárias, conforme apresentado em (CYTRON et al., 1991). Blocos básicos podem livremente se referir a variáveis definidas em blocos básicos que os dominam, pois sabemos que tal variável necessariamente existirá dentro desse fluxo de execução. Tal informação de dominância é, então, necessária para saber quais valores ainda devem estar vivos na memória do programa durante a execução e quais podem ser sobre-escritos, afetando etapas de otimização como, por exemplo, a alocação de registradores.

A conversão de programas imperativos para SSA/CFG, através de análise de dominância e renomeação de variáveis, pode ser equiparado a “extrair o programa funcional que existe dentro de um programa imperativo” (APPEL, 1998): a linguagem SSA/CPS, conforme descrita, é puramente funcional. Os algoritmos de conversão, então, costumam ser extendidos de alguma forma para considerar características de memória, como *aliasing*, vetores e ponteiros.

Um fato notável é que programas em SSA/CFG e programas em cálculo lambda em CPS são, na verdade, equivalentes (KELSEY, 1995). Blocos básicos são diretamente equivalentes a funções lambda, onde funções ϕ são, na verdade, seus parâmetros. Blocos dominados vivem no escopo das funções lambda que os dominam, e, por isso, podem fazer uso das variáveis definidas no escopo superior. Apesar do isomorfismo, não é comum o uso de continuações de primeira classe em SSA/CFG (KELSEY, 1995), o que não é um problema para seu uso como linguagem intermediária, visto que algoritmos de conversão para CPS não introduzem uso de primeira classe de continuações (KENNEDY, 2007), que só ocorre com uso de funções como `call/cc` pelo programador.

3 DESENVOLVIMENTO

O CoC fornece uma fundamentação para o desenvolvimento de teoremas, e, graças ao assistente de provas Coq, a programas com suas propriedades formalmente provadas, permitindo assim a implementação de algoritmos mais seguros. Toda essa garantia, entretanto, é perdida quando código é extraído do assistente a fim de ser compilado para um programa executável (BOWMAN et al., 2018).

Um dos primeiros passos para se preservar as garantias de alto níveis garantidas por programas provados é a tradução de sua linguagem fonte para uma linguagem intermediária em CPS, tal qual possa ser usada como um sistema lógico, preservando as garantias originais do programa. Tal fato deixa a pergunta: como se pareceria uma lógica em CPS? É possível provarmos teoremas baseados em continuações?

Conforme notado na Seção 2.6.2, traduções para CPS são, de acordo com o isomorfismo de Curry-Howard, equivalentes a introdução de negação dupla. Por exemplo, considere o seguinte isomorfismo:

$$A, B \Leftrightarrow \neg\ A, \neg B \qquad\qquad A, B \Leftrightarrow \neg\ A, \neg B$$

A interpretação das proposições acima é simples, e ambos teoremas podem ser facilmente provados[21]; duas equivalências são apresentadas:

- Se temos uma prova de que para todo A podemos provar B, então não pode existir um A tal que não possamos provar B (introduzindo uma dupla negação), e vice-versa (removendo-a).

- Se temos uma prova de que existe um A tal que possamos provar B, então não será para todo A que não poderemos provar B (introduzindo uma dupla negação), e vice versa (removendo-a).

A quantificação existencial, então, pode ser definida em termos de quantificações universais, similar a como tipos Σ fracos foram definidos na Seção 2.4.1 usando apenas tipos Π. Entretanto, se considerarmos um ambiente de passagem de continuações, não existirá mais a necessidade de um tipo de resposta para o tipo Σ, pois funções não retornam: sua definição, portanto, será exatamente a definição dada na equivalência acima, fixando o tipo de resposta como apenas :

$$\Sigma a\colon A.B \equiv \neg\Pi a\colon A.\neg B$$

[21] Embora a remoção da negação dupla necessite do operador clássico.

Ou, aplicando a definição da negação intuicionista a fim de facilitar a visualização:

$$\Sigma a\colon A.B \equiv (\Pi a\colon A.B \to \quad) \to$$

A primeira observação a ser feita: tipos Σ fracos são, por definição, negações duplas, e portanto estão em CPS; eles recebem uma continuação como argumento e a chamam com os sub-termos do par. Esse fato não deve ser uma surpresa considerando-se a relação entre traduções de CPS e negação dupla.

O uso de quantificação existencial, entretanto, pode não ser desejado; por exemplo, o mesmo não existe (como primitivo) no CoC. Por exemplo, ao negarmos duplamente uma quantificação universal, podemos aplicar a dupla negação novamente para eliminar a quantificação existencial; dessa forma, a seguinte equivalência é derivada:

$$A, B \Leftrightarrow \neg\neg\ A, \neg\neg B$$

Tal equivalência é facilmente provada na lógica clássica, tendo sua interpretação trivial. De fato, a introdução de dupla negação dessa forma é exatamente a tradução apresentada em (BARTHE; HATCLIFF; SØRENSEN, 1999).

Termos duplamente negados rapidamente se tornam complexos. Por exemplo, números naturais, conforme demonstrados na Seção 2.4.1, possuem o seguinte tipo após a converção em CPS de (BARTHE; HATCLIFF; SØRENSEN, 1999), representando a dupla negação do tipo N:

$$\begin{aligned}
&((\Pi A\colon \quad .\\
&\ ((\Pi z\colon (A \to \quad) \to \quad .\\
&\quad ((\Pi s\colon ((((A \to \quad) \to \quad) \to (A \to \quad) \to \quad) \to \quad .\\
&\qquad (A \to \quad) \to \quad) \to \quad) \to \quad) \to \quad) \to \quad) \to
\end{aligned}$$

Tal apresentação críptica é imprática. É possível reescrever esse mesmo tipo usando a definição de negação, onde ele passa a se apresentar de uma forma mais familiar:

$$\neg\neg\Pi A\colon \quad .\neg\neg\Pi z\colon \neg\neg A.\neg\neg\Pi s\colon \neg\neg(\neg\neg A \to \neg\neg A).\neg\neg A$$

E, usando a definição de tipos Σ fracos dada acima, usando como tipo de resposta, podemos escrever esse mesmo termo como:

$$\neg\Sigma A\colon \quad .\Sigma Z\colon \neg\neg A.\Sigma S\colon \neg(\Sigma x\colon \neg\neg A.\neg A).\neg A$$

A conversão para CPS do CoC efetivamente remove quantificações universais arbitrárias no nível dos termos, usando apenas negações e quantificações existenciais para representar computações: quantificações universais não são utilizadas em programas em CPS, e um cálculo que não oferecesse tipos Π no nível dos termos, então, seria incapaz de escrever programas em estilo direto (FÜHRMANN; THIELECKE, 2004).

3.1 O CÁLCULO DE CONTINUAÇÕES

O principal artefato desse trabalho, o cálculo de continuações, é descrito nessa Seção. Sua sintaxe é dada a seguir:

$$A ::= \quad \mid \Pi X\colon A.A \mid \Pi x\colon T.A$$

$$T ::= X \mid \lambda X\colon A.T \mid \lambda x\colon T.T \mid T\ T \mid T\ e \mid \neg T \mid \Sigma X\colon A.T \mid \Sigma x\colon T.T$$

$$e ::= x \mid \lambda x\colon T.c \mid \ e,\ e$$

$$c ::= e\ e \ \mid \texttt{let}\ \ X,\ x\ \ = \pi e\ \texttt{in}\ c \mid \texttt{let}\ \ x,\ x\ \ = \pi e\ \texttt{in}\ c \mid c\ \{\ x\ x\colon T\ \ = c\ \}$$

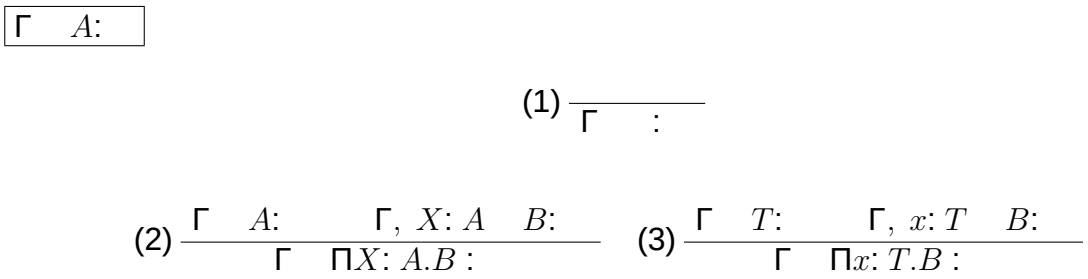
Nota-se que, na linguagem dos tipos, aplicação tem a maior precedência (associando para a esquerda), seguido pela negação, e finalmente por tipos Σ e abstrações. Sendo assim, $\Sigma X\colon \quad .T\ \neg T\ X$ deve ser lido como $\Sigma X\colon \quad .T\ (\neg(T\ X))$.

Tipos lineares são usados de uma maneira restrita no sistema a fim de limitar o uso da continuação atual, o último "parâmetro" para uma função, para a qual a função deve retornar seu resultado. Linearidade de continuações pode ser usado a fim de se obter naturalidade (THIELECKE, 2003); conjectura-se, esse trabalho, que essa condição é o suficiente para garantir que o fluxo de código funcione como uma pilha, e, portanto, possa ser compilado como tal futuramente (usando a *stack* do C). Idealmente, parâmetros lineares arbitrários poderiam ser aceitos para representar recursos (como motivado na Seção 2.5), porém isso é deixado como trabalho futuro.

O assistente de provas Coq faz uma notável distinção entre termos computacionalmente relevantes (que estão nos *sorts* `Type` e `Set`), e termos computacionalmente irrelevantes (que vivem no universo impredicativo `Prop`, comumente representado como). O cálculo proposto é baseado diretamente no CoC, e utiliza seu universo impredicativo; apesar disso, é usada uma distinção diferente. Consideramos esse universo (`Prop`) como computacionalmente relevante: termos e, cujo tipo vive em , são considerados computações existentes em tempo de execução, e, por isso, apenas existem em CPS (através de negações e pares). Tipos T, cujo tipo vive em , não representam valores existentes em tempo de execução, mas apenas tipos usados durante o algoritmo de checagem de tipos; por isso, tipos Π existem no nível de *kinds*, visto que desejamos executar funções em nível de tipos e obter seus resultados em tempo de compilação. Além disso, continuações no nível de tipos (ou seja, negações no nível de *kinds*) não são particularmente úteis para tipar termos, visto que precisaríamos eliminar uma negação dupla para recuperar o tipo resultante da continuação (BOWMAN et al., 2018), introduzindo uma complexidade desnecessária.

Similar ao cálculo de construções, como foi apresentado na Seção 2.4.1, dividimos a sintaxe em níveis: a linguagem dos *sorts* (A), a linguagem dos tipos (T), e a

Figura 8 – Tipagem de *sorts* no cálculo de continuações



Fonte: o autor

linguagem dos termos (e); diferente dele, adicionamos uma classe sintática adicional, representando a linguagem dos comandos (c). Usamos X como meta-variável para variáveis de tipo (que possuem um *sort*), e usamos x como meta-variável de termos (que possuem um tipo). A linguagem dos *sorts* é trivial e idêntica à do cálculo de construções, e suas regras de checagem de tipo são descritas na Figura 8.

A linguagem dos tipos é similar à do cálculo de construções. Semelhante a ele, existem abstrações e aplicações em nível de tipos. Os tipos de produto, entretanto, que tipam abstrações no nível dos termos, são removidos, substituídos por um tipo de negação como primitivo (visto que o objetivo é que nenhuma função retorne), similar a (FÜHRMANN; THIELECKE, 2004); de forma complementar, tipos Σ impredicativos são adicionados. Suas regras de checagem de tipo são descritas na Figura 9.

As regras para checagem de tipos de *sorts* e de tipos são descritas, individualmente, a seguir. Nota-se que as regras de (1) a (8) também estão presentes no cálculo de construções, descrito na Seção 2.4.1.

A regra (1) define o principal axioma, , que representa o tipo dos tipos pequenos. As regras (2) e (3) apresentam tipos Π no nível dos *sorts*, permitindo que existam tais funções no nível dos tipos. Por brevidade, não consideramos tipos Σ ao nível de *sorts* (ou pares no nível de tipos).

A regra (4) é padrão e permite a verificação de variáveis de tipos no contexto. As regras (5) e (6) permitem a criação de abstrações no nível de tipos, cujas aplicações são dadas, respectivamente, pelas regras (7) e (8). Note que, na regra (8), é necessário que o contexto linear esteja vazio para a verificação do termo: tipos não podem depender de termos lineares.

A regra (9) é usada para representar o tipo de abstrações em nível de termos (isto é, efetivamente o programa desejado). Tipos de negação são usados para representar funções onde retornar é um absurdo; como o cálculo tem como objetivo permitir apenas programas escritos em CPS, cujo retorno sempre será um absurdo,

Figura 9 – Tipagem de tipos no cálculo de continuações

$\boxed{\Gamma \quad T : A}$

$$(4)\ \frac{}{\Gamma,\ X : A \quad X : A} \qquad (5)\ \frac{\Gamma \quad A : \qquad \Gamma,\ X : A \quad T : B}{\Gamma \quad \lambda X : A.T : \Pi X : A.B} \qquad (6)\ \frac{\Gamma \quad T : \qquad \Gamma,\ x : T \quad U : B}{\Gamma \quad \lambda x : T.U : \Pi x : T.B}$$

$$(7)\ \frac{\Gamma \quad T : \Pi X : A.B \qquad \Gamma \quad U : A}{\Gamma \quad T\ U : B[U/X]} \qquad (8)\ \frac{\Gamma \quad T : \Pi x : U.B \qquad \Gamma;\ \cdot \quad e : U}{\Gamma \quad T\ e : B[e/x]}$$

$$(9)\ \frac{\Gamma \quad T :}{\Gamma \quad \neg T :} \qquad (10)\ \frac{\Gamma \quad A : \qquad \Gamma,\ X : A \quad T :}{\Gamma \quad \Sigma X : A.T :} \qquad (11)\ \frac{\Gamma \quad T : \qquad \Gamma,\ x : T \quad U :}{\Gamma \quad \Sigma x : T.U :}$$

Fonte: o autor

não há necessidade de tipos Π ou tipos de função de retorno arbitrário. O mesmo raciocínio é empregado em (FÜHRMANN; THIELECKE, 2004), onde são usados em conjunto a tipos de produtos. A regra (10) representa tipos existenciais, isto é, tipos Σ cujo elemento à esquerda é um tipo. Note que a regra é impredicativa, o que levaria a uma inconsistência no cálculo de construções (e será melhor abordado na Seção 4.4). A regra é justificada pela forma restrita que projeções são feitas na linguagem. A regra (11) representa tipos Σ pequenos.

As regras para termos se distanciam do cálculo de construções, e são descritas na Figura 10. Como funções não devem retornar, regras auxiliares foram necessárias na forma de comandos, cujas regras são descritas na Figura 11.

A sintaxe de comandos é simples; é fácil perceber que qualquer comando será uma sequência de zero ou mais projeções (`let ... in ...`), seguido de um salto ($k\ e$ ou $e\ k$), seguido de zero ou mais ligações de blocos (`... {...}`). Ligação de blocos possui maior precedência que projeções.

Para ambas as linguagens, dois contextos são considerados, a fim de simplificar a notação: um contexto intuicionista Γ, que mapeia termos intuicionistas a seus tipos (onde variáveis de tipos também são consideradas termos intuicionistas), e um contexto linear Δ, que mapeia termos lineares a seus tipos. Suas regras de checagem de tipos são descritas, individualmente, a seguir.

Figura 10 – Tipagem de termos no cálculo de continuações

$\boxed{\Gamma;\ \Delta \quad e\colon T}$

$$(12)\ \frac{}{\Gamma,\ x\colon T;\ \cdot \quad x\colon T} \qquad (13)\ \frac{}{\Gamma;\ x\colon T \quad x\colon T}$$

$$(14)\ \frac{\Gamma \quad T\colon \qquad \Gamma;\ x\colon T \quad c\colon}{\Gamma;\ \cdot \quad \lambda x\colon T.c : \neg T}$$

$$(15)\ \frac{\Gamma \quad T\colon A \qquad \Gamma,\ X\colon A;\ \Delta \quad k\colon U[T/X]}{\Gamma;\ \Delta \quad T,\ k\ : \Sigma X\colon A.U}$$

$$(16)\ \frac{\Gamma;\ \cdot \quad e\colon T \qquad \Gamma, x\colon T;\ \Delta \quad k\colon U[e/x]}{\Gamma;\ \Delta \quad e,\ k\ : \Sigma x\colon T.U}$$

Fonte: o autor

Figura 11 – Tipagem de comandos no cálculo de continuações

$\boxed{\Gamma;\ \Delta \quad c\colon}$

$$(17)\ \frac{\Gamma;\ \cdot \quad e\colon \neg T \qquad \Gamma;\ \Delta \quad k\colon T}{\Gamma;\ \Delta \quad e\ k\ :} \qquad (18)\ \frac{\Gamma;\ \Delta \quad k\colon \neg T \qquad \Gamma;\ \cdot \quad e\colon T}{\Gamma;\ \Delta \quad k\ e\ :}$$

$$(19)\ \frac{\Gamma;\ \Delta \quad e\colon \Sigma x\colon T.U \qquad \Gamma, a\colon T;\ b\colon U[a/x] \quad c\colon}{\Gamma;\ \Delta \quad \texttt{let}\ \ a,\ b\ = \pi e\ \texttt{in}\ c\ :}$$

$$(20)\ \frac{\Gamma;\ \Delta \quad e\colon \Sigma X\colon A.U \qquad \Gamma, Y\colon A;\ b\colon U[Y/X] \quad c\colon}{\Gamma;\ \Delta \quad \texttt{let}\ \ Y,\ b\ = \pi e\ \texttt{in}\ c\ :}$$

$$(21)\ \frac{\Gamma;\ \cdot \quad e\colon \Sigma x\colon T.U \qquad \Gamma,\ a\colon T,\ b\colon U[a/x];\ \Delta \quad c\colon}{\Gamma;\ \Delta \quad \texttt{let}\ \ a,\ b\ = \pi e\ \texttt{in}\ c\ :}$$

$$(22)\ \frac{\Gamma;\ \cdot \quad e\colon \Sigma X\colon A.U \qquad \Gamma,\ Y\colon A,\ b\colon U[Y/X];\ \Delta \quad c\colon}{\Gamma;\ \Delta \quad \texttt{let}\ \ Y,\ b\ = \pi e\ \texttt{in}\ c\ :}$$

$$(23)\ \frac{\Gamma;\ a\colon \neg T \quad c\colon \qquad \Gamma,\ x\colon T;\ \Delta \quad d\colon}{\Gamma;\ \Delta \quad c\ \{\ a\ x\colon T\ = d\ \}}$$

Fonte: o autor

As regras de (12) a (16) tipam termos; termos são valores dentro da linguagem, que possuem um tipo. Termos podem possuir apenas uma de duas formas: uma função (*closure*), ou um par.

A regra (12) é padrão e permite a verificação de variáveis de termos no contexto; note que é necessário que o contexto linear esteja vazio, caso contrário variáveis lineares poderiam ser ignoradas. A regra (13) representa a verificação de uma variável linear no contexto; o contexto intuicionista é ignorado, porém a variável deve existir sozinha no contexto linear, para evitar que outras variáveis sejam ignoradas.

A regra (14) introduz funções em nível de termos. Funções lambda recebem um único parâmetro. Note que o contexto linear precisa estar vazio: abstrações lambda não podem capturar variáveis lineares de contextos superiores; tal restrição é justificada para se evitar a criação de funções com contextos lineares (conforme discutido na Seção 2.5). O parâmetro da função, entretanto, é movido para o contexto linear e deve ser usado exatamente uma vez dentro dela.

As regras (15) e (16) introduzem pares grandes e pequenos, respectivamente. Em ambos os casos, o contexto linear é ignorado à esquerda (o que é trivial para a regra (15), visto que tipos não podem depender em termos lineares), e deve ser consumido exclusivamente pela direita. Essa restrição é necessária para evitar que continuações sejam usadas como valores de primeira classe, estando estas sempre no ponto mais à direita de um parâmetro.

Uma caraterística imporante quanto à introdução de pares no cálculo de continuações é que, diferente dos pares dependentes descritos na Seção 2.2.4, o termo do par não é anotado com seu tipo, e a substituição ocorre em uma das suposições. Isso implica que o tipo de um par não é determinístico: dentro de um contexto, um par pode ter mais de um tipo.

Para ilustrar esse ponto, assuma um contexto intuicionista $\Gamma = N:\ \ , n: N$, e um contexto linear vazio. É possível, então, através das regras fornecidas, derivar dois possíveis tipos para o termo N, n :

$$\dfrac{(4)\ \dfrac{}{\Gamma \quad N:} \qquad (12)\ \dfrac{}{\Gamma, X:\ ;\ \cdot \quad \textbf{n: N[N/X]}}}{(16)\ \Gamma; \cdot \quad N, n\ : \Sigma X: \quad .N}$$

E:

$$\dfrac{(4)\ \dfrac{}{\Gamma \quad N:} \qquad (12)\ \dfrac{}{\Gamma, X:\ ;\ \cdot \quad \textbf{n: X[N/X]}}}{(16)\ \Gamma; \cdot \quad N, n\ : \Sigma X: \quad .X}$$

Como uma substituição ocorre na segunda suposição da regra (16), podemos notar que, devido ao fato que $N[N/X] = X[N/X]$, o termo n passa a ter ambos os

tipos, e, como consequência, o termpo N, n pode receber tanto o tipo $\Sigma X: \quad .N$ quanto $\Sigma X: \quad .X$. Tal não-determinismo, entretanto, não apresenta problemas para uma implementação, que pode ser feita de forma determinística; a única forma de se usar pares literais é através das regras (17) ou (18), onde aparecem como argumento para uma aplicação. O tipo de retorno, entretanto, sempre será , e o tipo do parâmetro é necessariamente anotado pela função: o algoritmo, então, deve apenas verificar se um dos tipos possíveis para o par é o tipo anotado, o que pode ser feito recursivamente.

As regras (17) e (18) apresentam saltos no fluxo, e são a única forma de aplicação de funções disponível no cálculo; note-se que, diferente do cálculo lambda e do cálculo de construções, assumimos que retornar da chamada é um absurdo. A diferença entre as regras existe em que lado da aplicação recebe o contexto linear; na regra (17) temos chamadas de cauda, onde a continuação atual, que vive no contexto linear, pode ser fornecida. Na regra (18) temos o "retorno" de uma função, fornecendo um termo não-linear como resultado para a continuação atual.

As regras (19), (20), (21) e (22) apresentam projeções para pares. Nas regras (19) e (20) assume-se que o par viverá no contexto linear (ou, se for um valor, ao invés de uma variável, conterá uma variável linear à direita), que será usada pela projeção. Como o par estava no contexto linear, baseado na regra (14), sabemos que ele representa os parâmetros da função; a continuação, eventualmente, estará na posição mais à direita, e, por isso, o segundo elemento do par será colocado no contexto linear. Nas regras (21) e (22) o par existia no contexto intuicionista, então, pela derivação das regras do sistema, sabemos que o mesmo existia como um parâmetro; o contexto linear, então, é preservado após a projeção.

A regra (23) introduz continuações de segunda-classe na linguagem, chamadas de continuações locais. A continuação é representada dentro de chaves, similar ao cálculo demonstrado na Seção 2.6.1. O contexto linear é "empurrado" para a última continuação possível, onde deverá ser tratado, e a continuação existirá como uma variável linear no comando anterior. Nota-se que parâmetros de continuações são colocados no contexto intuicionista, e não no contexto linear (análogo a como abstrações lambda são colocadas no contexto intuicionista, mas continuações são colocadas no contexto linear). Tal sintaxe deriva da sintaxe usada em (FÜHRMANN; THIELECKE, 2004), conforme descrito na Seção 2.6.1.

Nota-se que, devido à forma que as variáveis lineares são tratadas, e por serem reservadas a continuações, o único fluxo possível de código é de um comando para a continuação imediatamente abaixo dele (ou "retornar" para a continuação recebida como parâmetro, caso seja a última continuação local); a última continuação, sintaticamente, então, é responsável por retornar da função.

3.1.1 Exemplos

Para ilustrar o cálculo de continuações, usaremos algumas convenções sintáticas a fim de facilitar a leitura. Similar ao cálculo de construções, podemos usar $A \to B$ para representar $\Pi X\colon A.B$ caso X não apareça livre em B (ou $T \to B$ para representar $\Pi x\colon T.B$ caso x não apareça livre em B), como no cálculo de construções. De forma análoga, $A \times U$ pode ser usado ao invés de $\Sigma X\colon A.U$ caso X não apareça livre em U (ou $T \times B$ para representar $\Sigma x\colon T.U$ caso x não apareça livre em U). Ambos operadores associam para a direita, e possuem precedência menor que negação; isto é, $T \times \neg U \times V$ representa $T \times ((\neg U) \times V)$. A notação $a,\ b,\ \ldots$ pode ser usada para representar o termo $a,\ b,\ \ldots$. De forma similar, o comando $\texttt{let}\ \ a,\ b,\ \ldots\ = x\ \texttt{in} c$ representa $\texttt{let}\ \ a,\ x\ = x\ \texttt{in}\ \texttt{let}\ \ b,\ \ldots\ = x\ \texttt{in}\ c$. Para valores, $\lambda\ a\colon T,\ \ldots\ .c$ representa $\lambda x\colon (\Sigma a\colon T. \ldots).\texttt{let}\ \ a,\ \ldots\ = x\ \texttt{in}\ c$, e, para comandos, $c\ \{\ b\ a\colon T,\ \ldots\ = d\ \}$ representa $c\ \{\ b\ x\colon \Sigma a\colon T. \ldots\ = \texttt{let}\ \ a,\ \ldots\ = x\ \texttt{in}\ d\ \}$ e $k\ a,\ \ldots$ representa $k\ \ a,\ \ldots$.

No CoC, o termo $\lambda T\colon\ \ .\lambda z\colon T.\lambda s\colon T \to T.s\ (s\ z)$ representa o número 2, e tem tipo $\Pi T\colon\ \ .T \to (T \to T) \to T$. Podemos representar tal número no cálculo de continuações como o termo:

$$
\begin{array}{l}
\lambda\ T\colon\ \ ,\ k\colon \neg\neg\Sigma z\colon T.\neg\neg\Sigma s\colon \neg(T \times \neg T).\neg T\ . \\
\quad k\ \lambda\ z\colon T,\ k\colon \neg\neg\Sigma s\colon \neg(T \times \neg T).\neg T\ . \\
\qquad k\ \lambda\ s\colon \neg(T \times \neg T),\ k\colon \neg T\ . \\
\qquad\quad s\ z,\ b_1 \\
\qquad\quad \{\ b_1\ one\colon T\ \ =\ \ s\ one,\ b_2\ \} \\
\qquad\quad \{\ b_2\ two\colon T\ \ =\ \ k\ two\ \}
\end{array}
$$

Cujo tipo será $\neg\Sigma T\colon\ \ .\neg\neg\Sigma z\colon T.\neg\neg\Sigma s\colon \neg(T \times \neg T).\neg T$.

As duplas negações precedendo os "parâmetros" z e s acima representam um detalhe de baixo nível, por assim dizer, da função representada: a função é *curried*. Temos uma função que retorna uma *closure*, isto é, a continuação atual k espera uma função como resultado, similar ao termo original no CoC, que retorna uma nova função ao receber um único parâmetro. Embora essa seja a forma mais natural de se traduzir termos lambda, é possível também representar funções *uncurried*, que aceitam mais de um parâmetro simultaneamente. Por exemplo, podemos apresentar o número 2 da seguinte forma:

$$
\begin{array}{l}
\lambda\ T\colon\ \ ,\ z\colon T,\ s\colon \neg(T \times \neg T),\ k\colon \neg T\ . \\
\quad s\ z,\ b_1 \\
\quad \{\ b_1\ one\colon T\ \ =\ \ s\ one,\ b_2\ \} \\
\quad \{\ b_2\ two\colon T\ \ =\ \ k\ two\ \}
\end{array}
$$

Cujo tipo será $\neg\Sigma T\colon\ \_.\Sigma z\colon T.\Sigma s\colon \neg(T \times \neg T).\neg T$. Tal relação já foi percebida em trabalhos anteriores (FILINSKI, 1992).

As continuações b_1 e b_2, efetivamente, em ambos os exemplos representam *labels* na linguagem de montagem, similar à linguagem intermediária apresentada em (KENNEDY, 2007). *Closures*, então, necessariamente são representadas por abstrações lambda, considerando que continuações locais vivem no contexto linear e são cidadãos de segunda-classe.

Nota-se também que a noção de dominância entre blocos básicos é diretamente representável por continuações locais. Interpretando continuações locais como blocos básicos, temos que continuações dominam quaisquer blocos aninhados. Portanto, o exemplo acima também poderia ser representado como:

$$
\begin{array}{l}
\lambda\ T\colon\ \_,\ z\colon T,\ s\colon \neg(T \times \neg T),\ k\colon \neg T\ . \\
\quad s\ z,\ b_1 \\
\quad \{\ b_1\ one\colon T\ \ = \\
\qquad s\ one,\ b_2 \\
\qquad \{\ b_2\ two\colon T\ \ =\ \ k\ two\ \}\}
\end{array}
$$

3.2 IMPLEMENTAÇÃO

A fim de validar a proposta empiricamente, uma implementação do CoC e do cálculo de continuações descrito acima foi feita utilizando a linguagem de programação Haskell. Tal implementação é brevemente descrita nesse capítulo, com ênfase nos algoritmos de checagem de tipos.

As ferramentas Alex e Happy são usadas, respectivamente, para a implementação de um *lexer* e um *parser* para o CoC. A linguagem é definida de forma estratificada, como visto na Seção 2.4.1, utilizando quatro tipos de dados: `Sort`, `Type`, `Term` e `Expr`, o último podendo se referir a qualquer objeto na linguagem, incluindo o universo ; tal implementação, ilustrada na Figura 12, tem como objetivo auxiliar no raciocínio indutivo quanto a termos do cálculo[22]. O tipo `Context` representa os contextos usados, uma lista de tuplas de *strings* representando variáveis e seus tipos.

[22] A gramática pode ser implementada de forma semelhante à estrutura de dados; a utilização de *tokens* diferentes para variáveis de tipos (com letras maiúsculas) e de termos (com letras minúsculas) é o suficiente para manter a gramática LR(1).

Figura 12 – Tipos para o CoC

```
1  data Sort = Prop
2            | SortPi1 String Type Sort
3            | SortPi2 String Sort Sort
4
5  data Type = TypeBound Int
6            | TypeApplication1 Type Term
7            | TypeApplication2 Type Type
8            | TypeLambda1 String Type Type
9            | TypeLambda2 String Sort Type
10           | TypePi1 String Type Type
11           | TypePi2 String Sort Type
12
13 data Term = TermBound Int
14           | TermApplication1 Term Term
15           | TermApplication2 Term Type
16           | TermLambda1 String Type Term
17           | TermLambda2 String Sort Term
18
19 data Expr = Univ
20           | Sort Sort
21           | Type Type
22           | Term Term
23
24 type Context = [(String, Expr)]
```

Fonte: o autor

Conforme mencionado na Seção 2.2.1, índices de de Bruijn são usados para facilitar substituições e equivalência. O *parser* é responsável por ligar as variáveis (e substituir o identificador textual pelo devido índice); para fins de *debug*, abstrações (λ e Π) contém o nome de sua variável, embora o mesmo não seja utilizado significativamente. Variáveis são acessados em contextos a partir de seu índice (por exemplo, a variável mais recentemente ligada é representada pelo índice zero, estando este no começo da lista).

Detalhes de interesse são implementados utilizando *type classes*, mecanismo fornecido pela linguagem para polimorfismo *ad-hoc*. Algumas classes são implementadas, conforme listadas na Figura 13.

A classe `Eq`, padrão na linguagem, é usada para comparar dois termos por igualdade; consideramos dois termos iguais caso sejam α-convertíveis. Isto é, termos são comparados, recursivamente, por igualdade, desconsiderando os nomes das variáveis presentes em abstrações. Como índices de de Bruijn são usados, variáveis ligadas são representadas pela sua distância à abstração que as ligou, e, portanto, tal índice pode ser comparado numericamente.

Figura 13 – Classes usadas para o algoritmo de checagem de tipos

```
1  class LocallyNameless a where
2    shiftN :: Int → Int → a → a
3
4  class Substitutable a b where
5    subst :: Int → a → b → a
6
7  class Normalizable a where
8    normalize :: a → a
9
10 class Typeable a where
11   typeof :: Context → a → Either TypeError Expr
12
13 class CPSConv1 a where
14   cps_barthe' :: Monad m    Context → a → StateT Int m a
```

Fonte: o autor

A classe `LocallyNameless a` representa tipos `a` contendo índices de de Bruijn, e possui uma função, `shiftN`, usada para renomear uma variável desejada. Duas funções auxiliares, `shift` e `unshift`, são definidas conforme a Figura 14; tais funções, respectivamente, adicionam uma nova abstração em volta de um termo, e removem uma abstração. Também na Figura 14 é demonstrado um exemplo de instância; como índices são representados pela distância, precimos aumentar seu valor caso entremos em novas abstrações (linhas 16 e 18), e garantir que variáveis mais próximas sejam renomeadas (linha 7).

A classe `Substitutable a b` representa a possibilidade de se substituir um `b` dentro de um `a`, através de uma função `subst`; para o caso do CoC, significa que devemos implementar seis instâncias, pois é possível substituir tipos e termos dentro de *sorts*, tipos e termos. A Figura 15 demonstra um exemplo de instância (especificamente, a substituição de termos em termos).

Note que, caso o termo que queiramos substituir esteja dentro de uma abstração, é necessário o uso da função `shift` para adicionar uma nova abstração ao redor do termo substituto, que renomeará suas variáveis conforme necessário (linhas 14 e 18).

A partir das duas classes acima podemos definir uma β-redução. A redução $(\lambda x{:}\ T.b)\ y \ \rightarrow_\beta \ b[y/x]$ é computada por três passos: 1. adicionar uma abstração a `y` (nominalmente, a variável `x`), 2. substituir a variável 0 em `b` (ou seja, ocorrências de `x`, que é a variável ligada mais recentemente) pelo resultado, e finalmente 3. remover uma abstração do resultado (a variável `x`, que não é mais usada).

Figura 14 – Exemplo de renomeação de índice

```
shift = shiftN 1 0

unshift = shiftN (-1) 0

instance LocallyNameless Term where
  shiftN d n (TermBound m) =
    if n ≤ m then
      TermBound (m + d)
    else
      TermBound m
  shiftN d n (TermApplication1 f x) =
    TermApplication1 (shiftN d n f) (shiftN d n x)
  shiftN d n (TermApplication2 f x) =
    TermApplication2 (shiftN d n f) (shiftN d n x)
  shiftN d n (TermLambda1 s t b) =
    TermLambda1 s (shiftN d n t) (shiftN d (n + 1) b)
  shiftN d n (TermLambda2 s t b) =
    TermLambda2 s (shiftN d n t) (shiftN d (n + 1) b)
```

Fonte: o autor

Figura 15 – Exemplo de substituição de termos

```
instance Substitutable Term Term where
  subst n (TermBound m) y =
    if n == m then
      y
    else
      TermBound m
  subst n (TermApplication1 f x) y =
    TermApplication1 (subst n f y) (subst n x y)
  subst n (TermApplication2 f x) y =
    TermApplication2 (subst n f y) (subst n x y)
  subst n (TermLambda1 s t b) y =
    TermLambda1 s (subst n t y) b'
    where
      b' = subst (n + 1) b (shift y)
  subst n (TermLambda2 s t b) y =
    TermLambda2 s (subst n t y) b'
    where
      b' = subst (n + 1) b (shift y)
```

Fonte: o autor

Figura 16 – Exemplo de normalização de termos

```
1  normalize (TermApplication1 f x) =
2    case normalize f of
3      TermLambda1 _ _ b →
4        -- Beta reduction
5        normalize (unshift b')
6        where
7          x' = shift x
8          b' = subst 0 b x'
9      g →
10       TermApplication1 g (normalize x)
```

Fonte: o autor

A classe `Normalizable a` implica que o tipo `a` pode ser reduzido. A função `normalize` recebe um objeto de tipo `a` e aplica β-reduções enquanto houverem redexes no termo. Como sabemos que o CoC é normalizável, tais reduções irão eventualmente terminar caso a função seja chamada em um objeto bem tipado. Tal função, além de útil por si só para normalizar termos bem tipados, é utilizada posteriormente para o algoritmo de checagem de tipos.

A única redução considerada no protótipo é a β-redução; não existe a necessidade de α-reduções devido ao uso de índices de de Bruijn, e, embora seja possível adicionarmos η-redução ao CoC sem a perda das propriedades de interesse (ABEL, 2010), ela não foi considerada pois η-equivalência não é preservada após a tradução em CPS (conforme discutido a seguir). Podemos ver na Figura 16 um exemplo de instância; como consideramos apenas β-redexes, os mesmos apenas aparecem em aplicações, com outros tipos de termos apenas recursivamente normalizando seus respectivos sub-termos.

A classe `Typeable a` efetivamente implementa o algoritmo de checagem de tipos para o CoC. O algoritmo representa uma inferência de tipos para termos no CoC, baseado fielmente nas suas regras, conforme ilustradas na Seção 2.4.1. Nota-se que isso é possível pois as regras são determinísticas: termos apresentam no máximo um tipo válido dentro de um contexto, e sua dedução depende apenas do tipo de seus subtermos, indutivamente.

Na Figura 17 são ilustrados alguns dos casos da checagem de tipo, a fim de comparação. Nesses exemplos, o tipo de uma variável é apenas o tipo que consta no contexto recebido (utilizando o operador `!!` da linguagem). Conforme as regras de tipagem, para tiparmos uma aplicação de termos $f\ x$ precisamos nos certificar que f é uma função (e portanto possui tipo $\Pi x\colon T.U$, cujas variáveis são representadas

Figura 17 – Exemplo de inferência de tipo

```
1  typeof g (TermBound n) =
2    let (s, Type t) = g !! n in
3    return $ Type t
4
5  typeof g (TermApplication1 f x) = do
6    Type ft ← fmap normalize (typeof g f)
7    case ft of
8      TypePi1 s t b → do
9        Type xt ← fmap normalize (typeof g x)
10       unless (t == xt) (throw g (TypeMismatch (Type t) (Type xt)))
11       return $ Type $ normalize (unshift (subst 0 b (shift x)))
12     _ →
13       throw g (NotAFunction $ Term f)
```

Fonte: o autor

como `s`, `t` e `b` na linha 8), e que o tipo inferido normalizado do parâmetro `x` é igual ao tipo normalizado esperado (linha 10). O resultado é o tipo de retorno da função `f`, com ocorrências dependentes propriamente substituídas pelo parâmetro fornecido (e, assim, utilizamos as classes `Eq`, `LocallyNameless`, `Substitutable` e `Normalizable`).

Finalmente, o último ponto de interesse no protótipo com relação ao CoC é a sua tradução em CPS, implementada pela classe `CPSConv1`. A tradução é descrita em (BARTHE; HATCLIFF; SØRENSEN, 1999) para um conjunto de sistemas de tipos puros, apresentando uma tradução *call-by-name*. O protótipo implementa uma versão especializada do algoritmo, fazendo proveito da estratificação do cálculo. Uma descrição formal da tradução, conforme usada, é ilustrada na Figura 18. Note que a tradução de termos faz uso de seu tipo.

Similar ao algoritmo original, assumimos que termos que estão sendo traduzidos já estão em forma normal. Além disso, assumimos que as variáveis k e j usadas no lado direito da tradução são novos e não são usados no termo que está sendo traduzido (claro que, devido aos índices de de Bruijn, isso é meramente um detalhe sintático ao leitor, e não afeta o algoritmo).

Observamos que, confome mencionado no começo do Capítulo 3, a tradução transforma funções dependentes como $\Pi x\colon T.U$ em pares dependentes. O caso 10 demonstra sua tradução para $\Pi x\colon \neg\neg T.\neg\neg U$, equivalente definicionalmente a $\Sigma x\colon \neg\neg T.\neg U$. Seu uso, visto nos casos 11, 14 e 15, adiciona uma negação extra, criando assim o tipo $\neg\Sigma x\colon \neg\neg T.\neg U$ para cada função no código original. O parâmetro de tipo T é duplamente negado pela natureza *call-by-name* da tradução: ele representa um *thunk* com

[23] O artigo original apresenta um pequeno erro de digitação nessa regra.

Figura 18 – Tradução em CPS para o CoC

1	$[\![\ ]\!]_\Gamma$	=			
2	$[\![\Pi X\colon A.B]\!]_\Gamma$	=	$\Pi X\colon [\![A]\!]_\Gamma.[\![B]\!]_{\Gamma,X:A}$		
3	$[\![\Pi x\colon T.A]\!]_\Gamma$	=	$\Pi x\colon \neg\neg[\![T]\!]_\Gamma.[\![A]\!]_{\Gamma,x:T}$		
4	$[\![X]\!]_\Gamma$	=	X		
5	$[\![\lambda X\colon A.T]\!]_\Gamma$	=	$\lambda X\colon [\![A]\!]_\Gamma.[\![T]\!]_{\Gamma,X:A}$		
6	$[\![\lambda x\colon T.U]\!]_\Gamma$	=	$\lambda x\colon \neg\neg[\![T]\!]_\Gamma.[\![U]\!]_{\Gamma,x:T}$		
7	$[\![T\ U]\!]_\Gamma$	=	$[\![T]\!]_\Gamma\ [\![U]\!]_\Gamma$		
8	$[\![T\ e]\!]_\Gamma$	=	$[\![T]\!]_\Gamma\ [\![e]\!]_\Gamma$		
9	$[\![\Pi X\colon A.T]\!]_\Gamma$	=	$\Pi X\colon [\![A]\!]_\Gamma.\neg\neg[\![T]\!]_{\Gamma,X:A}$		
10	$[\![\Pi x\colon T.U]\!]_\Gamma$	=	$\Pi x\colon \neg\neg[\![T]\!]_\Gamma.\neg\neg[\![U]\!]_{\Gamma,x:T}$		
				onde:	
11	$[\![x]\!]_\Gamma$	=	$\lambda k\colon \neg[\![T]\!]_\Gamma.x\ k$	Γ	$x\colon T$
12	$[\![\lambda X\colon A.e]\!]_\Gamma$	=	$\lambda k\colon \neg[\![T]\!].k\ (\lambda X\colon [\![A]\!]_\Gamma.[\![e]\!]_{\Gamma,X:A})$	Γ	$\lambda X\colon A.e : T$
13	$[\![\lambda x\colon T.e]\!]_\Gamma$	=	$\lambda k\colon \neg[\![U]\!].k\ (\lambda x\colon \neg\neg[\![T]\!]_\Gamma.[\![e]\!]_{\Gamma,x:T})$[23]	Γ	$\lambda x\colon T.e : U$
14	$[\![e\ T]\!]_\Gamma$	=	$\lambda k\colon \neg[\![U[T/X]]\!]_\Gamma.[\![e]\!]_\Gamma\ (\lambda j\colon [\![\Pi X\colon A.U]\!]_\Gamma.j\ [\![T]\!]_\Gamma k)$	Γ	$e\colon \Pi X\colon A.U$
15	$[\![e\ f]\!]_\Gamma$	=	$\lambda k\colon \neg[\![U[f/x]]\!]_\Gamma.[\![e]\!]_\Gamma\ (\lambda j\colon [\![\Pi x\colon T.U]\!]_\Gamma.j\ [\![f]\!]_\Gamma k)$	Γ	$e\colon \Pi x\colon T.U$

Fonte: o autor, baseado em (BARTHE; HATCLIFF; SØRENSEN, 1999)

o valor desejado. Uma tradução *call-by-value* para o CoC ainda não foi apresentada pela literatura (BOWMAN et al., 2018).

3.2.1 Não-determinismo

O cálculo de continuações descrito na Seção 3.1 é implementado de forma similar no protótipo, fazendo uso das mesmas classes, visto que foi desenvolvido com base no CoC. Os tipos de dados, também definidos de forma estratificada, são ilustrados na Figura 19, onde um tipo adicional para comandos é necessário.

O ponto chave de diferença entre a implementação do algoritmo de checagem de tipos apresentado para o CoC e o algoritmo de checagem de tipos para o cálculo de continuações é o não determinismo: o algoritmo não pode ser aplicado conforme descrito nas regras da linguagem. Notavelmente, pares na linguagem possuem mais de um tipo para o mesmo contexto, conforme exemplificado na Seção 3.1.

É comum o uso de não-determinismo ao se descrever sistemas de tipos lineares. O problema é discutido em (PIERCE, 2005). Como na regra (13) do cálculo de continuações, é comum que, para a verificação de um tipo linera, o contexto linear contenha apenas a variável desejada e nenhuma a mais (WADLER, 1990; BERNARDY et al., 2017). Isso é parte da técnica utilizada para, através das regras, garantir que variáveis sejam usadas uma única vez. A outra parte, entretanto, aparece em regras

Figura 19 – Tipos para o cálculo de continuações

```
1  data Sort = Prop
2            | SortPi1 String Type Sort
3            | SortPi2 String Sort Sort
4
5  data Type = TypeBound Int
6            | TypeApplication1 Type Term
7            | TypeApplication2 Type Type
8            | TypeLambda1 String Type Type
9            | TypeLambda2 String Sort Type
10           | TypeNegation Type
11           | TypeSigma1 String Type Type
12           | TypeSigma2 String Sort Type
13
14 data Term = TermBound Int
15           | TermLambda String Type Command
16           | TermPair1 Term Term
17           | TermPair2 Type Term
18
19 data Command = CommJump Term Term
20              | CommProjection1 String String Term Command
21              | CommProjection2 String String Term Command
22              | CommBind Command String String Type Command
23
24 data Expr = Univ
25           | Sort Sort
26           | Type Type
27           | Term Term
28           | Void
```

Fonte: o autor

de aplicação: o contexto linear é dividido entre os dois sub-termos.

Tais regras, envolvendo a divisão do contexto, não são determinísticas: não sabemos, a priori, quais variáveis serão usadas por cada sub-termo. Entretanto, ainda é possível implementar um algoritmo determinístico para isso: conforme (PIERCE, 2005), podemos alterar as regras para que, além de “retornar” o tipo de expressões, retorne o tipo e as variáveis que não foram usadas. Por exemplo, considerando apenas um contexto linear, a seguinte regra não-determinística:

$$\boxed{\Delta \quad e : T}$$

$$\frac{\Delta_1 \quad f : A \to B \qquad \Delta_2 \quad x : A}{\Delta_1 \quad \Delta_2 \quad f\ x : B}$$

Onde representa a união de dois contextos (assumimos que nenhuma variável presente em Δ_1 estará também presente em Δ_2, isto é, são disjuntos), pode ser descrita de forma algorítmica e determinística:

$$\boxed{\Delta \quad e : T;\ \Delta}$$

$$\frac{\Delta_1 \quad f : A \ (\ B;\ \Delta_2 \qquad \Delta_2 \quad x : A;\ \Delta_3}{\Delta_1 \quad f\ x : B;\ \Delta_3}$$

Onde Δ_3 representa as variáveis ainda não usadas pela aplicação; note que o contexto resultante da primeira suposição é usada para verificar a segunda: uma variável linear só pode ser usada pelo segundo sub-termo se não foi usada pelo primeiro.

O cálculo de continuações apresentado possui tipos lineares de uma maneira restrita; especificamente, apenas existe um termo linear por vez (a continuação atual). A técnica acima, entetanto, pode ser usada de forma similar para verificar se tal termo linear foi ou não usado exatamente uma vez nos sub-termos.

A segunda fonte de não-determinismo no cálculo, pares, também pode ser implementada em um algoritmo de forma prática e determinística. O ponto central é: embora não possamos inferir um tipo único para um par num contexto, os mesmos aparecerão apenas em posições que esperam um tipo único (com uma exceção).

No primeiro caso, pares podem aparecer em projeções projeções, e, para a tipagem estar correta, o elemento projetado precisa ser uma variável, ou um par literal (isto é, $a,\ b$ ou $T,\ a$). Caso o elemento seja uma variável, seu tipo é único e consta no contexto, e portanto não será um problema. Caso o elemento seja um par literal, a projeção apresenta um β-redex óbvio; na implementação, o *parser* implementado aceita apenas projeções em variáveis, rejeitando esse caso. Tal caso problemático pode apenas aparecer internamente durante reduções, mas jamais em forma normal (por se tratar de um redex).

O segundo caso, mais interessante, aparece quando invocamos uma continuação, passando um par como parâmetro. Novamente, dois casos podem aparecer, o primeiro sendo que o par está em uma variável e seu tipo, único, está no contexto, e por isso é determinístico. O segundo caso, onde um par literal, que pode ter múltiplos tipos, aparece como parâmetro para uma chamada de continuação, é resolvido de uma forma simples: como o cálculo é apresentado em estilo Church, sabemos qual é o tipo (único) esperado para o parâmetro.

A partir disso, ao invés de inferirmos um tipo T para um termo e, verificamos se, na verdade, o termo e pode possuir o tipo T (o qual já sabemos, por ser o tipo negado). A Figura 20 ilustra esse detalhe na implementação. Um erro é retornado ao usuário caso o mesmo tente inferir o tipo de um par, porém o tipo de funções pode ser inferido, fazendo uso desse truque para checar o tipo de pares usados internamente.

Figura 20 – Tipagem de pares para o cálculo de continuações

```
typeof g (TermPair1 _ _) =
  throw g PairsHaveNoPrincipalType

typeof g (TermPair2 _ _) =
  throw g PairsHaveNoPrincipalType

typeof g (CommJump k x) = do
  Type kt ← fmap normalize (typeof g k)
  case kt of
    TypeNegation pt → do
      mayHaveType g x pt
      return Void
    _ →
      throw g (NotAFunction $ Term k)

mayHaveType g e@(TermPair1 a b) t = do
  case t of
    TypeSigma1 _ at bt → do
      mayHaveType g a at
      mayHaveType g b (unshift (subst 0 bt (shift a)))
      return ()
    _ →
      throw g (CantHaveType (Term e) (Type t))

mayHaveType g e@(TermPair2 a b) t = do
  -- ...

mayHaveType g e t = do
  Type et ← fmap normalize (typeof g e)
  unless ((normalize t) == et)
    (throw g (TypeMismatch (Type t) (Type et)))
  return ()
```

Fonte: o autor

3.2.2 Experimentos

A implementação, disponível em <http://www.github.com/takanuva/cps>, foi utilizada para realizar alguns experimentos a fim de validar a proposta. A princípio os mesmos envolveram simplesmente a criação de programas no CoC, com uma tradução para CPS mecânica (ilustrada na Figura 18), e então traduções manuais para o cálculo de continuações.

Após a tradução em CPS, termos não triviais se tornam extremamente grandes, e, em razão disso, são apresentados aqui apenas exemplos simples encontrados junto com a implementação. Por exemplo, assumindo números naturais de tipo N como descritos anteriormente, temos que vetores V: → N → . Uma função `cons`, ilustrada na Figura 21 usando a sintaxe do protótipo (que permite a inclusão de arquivos con-

Figura 21 – *Consing* de vetores, conforme oferecido ao programa

```
1  -- Arquivo ./coc/Nat
2  \/(Nat: *).
3  \/(succ: Nat -> Nat).
4  \/(zero: Nat).
5  Nat
6
7  -- Arquivo ./coc/succ
8  \(n: ./coc/Nat).
9  \(Nat: *).
10 \(succ: Nat -> Nat).
11 \(zero: Nat).
12 succ (n Nat succ zero)
13
14 -- Arquivo ./coc/Vec
15 \(T: *).
16 \(n: ./coc/Nat).
17 \/(Vec: * -> ./coc/Nat -> *).
18 \/(base: Vec T ./coc/zero).
19 \/(cons: \/(n: ./coc/Nat).T -> Vec T n -> Vec T (./coc/succ n)).
20 Vec T n
21
22 -- Arquivo ./coc/cons
23 \(T: *).
24 \(n: ./coc/Nat).
25 \(t: T).
26 \(v: ./coc/Vec T n).
27 \(Vec: * -> ./coc/Nat -> *).
28 \(base: Vec T ./coc/zero).
29 \(cons: \/(n: ./coc/Nat).T -> Vec T n -> Vec T (./coc/succ n)).
30 cons n t (v Vec base cons)
```

Fonte: o autor

tendo sub-termos), pode ser usada para adicionar um item a um vetor, tendo tipo ΠT: .Πn: N.T → V T n → V T ($succ$ n). Tal função, de interesse por apresentar os 4 tipos possíveis de abrações do CoC, traduzida em CPS apresenta 44 continuações apenas em seu tipo, e apresenta 416 continuações em sua forma normal.

Dentre os testes executados, a tradução em CPS para CoC preservou tipos: isto é, termos bem-tipados originalmente mantinham-se bem-tipados após a conversão (conforme esperado, visto que a tradução apresentada em (BARTHE; HATCLIFF; SØRENSEN, 1999) prova a preservação de tipos; nota-se que, ao conhecimento do autor, essa é a primeira implementação da mesma). Alguns termos foram testados e estão incluídos no repositório, como booleanos, números naturais, listas e vetores. Além disso, em todos os testes, a continuação atual foi usada linearmente (o que também pode ser deduzido pelas regras da tradução).

Além disso, todos os testes apresentaram o uso apenas de pares e negações[24], conforme pode ser observado pelas regras da tradução dadas acima. Em seguida, alguns termos (em particular, números e listas) foram traduzidos manualmente do cálculo de construções (com a ajuda da versão em CPS do tipo, traduzida mecanicamente) para o cálculo de continuações; em todos os casos, houve a preservação de tipos.

Entretanto, a tradução dos termos, embora com auxílio dos tipos traduzidos pelo algoritmo de (BARTHE; HATCLIFF; SØRENSEN, 1999), é feita utilizando *call-by-value*, a qual pode ser vista de uma forma mais primitiva do que *call-by-name* em CPS (FILINSKI, 1992). Por esse motivo, a tradução do CoC para o cálculo de continuações foi feita manualmente ao invés de mecanicamente. Por esse motivo, conjectura-se a partir dos experimentos que uma tradução mecânica com preservação de tipos é possível.

[24] O que foi descoberto por acidente; o uso de tipos Σ foi adicionado como açúcar sintático antes da conclusão que eram usados na tradução.

4 DISCUSSÃO

O cálculo de continuações, apresentado no Capítulo 3, apresenta passos inicias para o desenvolvimento de uma lógica de continuações, expandindo a linguagem de continuações apresentada em (FÜHRMANN; THIELECKE, 2004) para suportar tipos dependentes. Entretanto, alguns detalhes de interesse não são abordados.

Nesse capítulo, uma discussão adicional sobre o cálculo desenvolvido é elaborada. Detalhes adicionais, os quais motivariam o uso do cálculo como lógica e como linguagem intermediária, são abordados, junto a trabalhos relacionados e possíveis trabalhos futuros.

4.1 CONTINUAÇÕES LINEARES

A ideia do uso linear de continuações não é nova, tendo sido previamente estudada (FILINSKI, 1992; THIELECKE, 2003). Em particular, nesse trabalho, o uso de linearidade se restringe a limitar o uso de continuações a cidadãos de segunda-classe.

Conforme visto em (KELSEY, 1995), a linguagem SSA/CFG é equivalente a um subconjunto do cálculo lambda em CPS onde continuações não são tratadas como cidadãos de primeira classe, e, embora CPS permita o uso de continuações de primeira classe e operadores de controle, isso nem sempre é desejável (KENNEDY, 2007). Traduções em CPS convencionais podem receber tipos lineares, onde continuações se comportam de forma equivalente ao seu uso em SSA/CFG.

Nesse trabalho, devido às regras do sistema de tipos, o uso de continuações se restringe à chamada posteriores. Nota-se que desvios de fluxo apenas ocorrem fornecendo continuações locais definidas imediatamente a seguir, a fim de se manter a linearidade. Em (BERDINE et al., 2002) nota-se que é possível extender as possibilidades de fluxo de código, permitindo saltos para continuações locais definidas anteriormente (*backward jumps*) sem abrir mão de linearidade. Trabalhos futuros incluem extender o cálculo de continuações apresentado para suportar definições mais flexíveis de fluxo.

Embora continuações possam ser usadas e estudadas sem a consideração de operadores de controle (KENNEDY, 2007), é relevante notar a existência de tais. Embora a lei de Peirce, responsável pelo operador clássico `call/cc`, não possa ser provada dentro da lógica intuicionista, sua versão duplamente negada pode, visto que a continuação passa a ser usada explicitamente.

O tipo do operador clássico é $((A \to B) \to A) \to A$, conforme visto na Seção 2.6.1; sua versão duplamente negada recebe o tipo $\neg(\neg(\neg A \times \neg A) \times \neg A)$, e é um teorema intuicionista. Tal teorema pode ser facilmente provado no CoC (usando a codificação apresentada no começo do Capítulo 3) como o termo $\lambda g.g\ (\lambda f.\lambda k.f\ (\lambda h.h\ k\ k))$, ignorando anotações de tipo.

Sintaticamente, um termo de tal tipo pode ser representado no cálculo de continuações:

$$\begin{array}{l} \lambda\ f\colon \neg(\neg A \times \neg A),\ k\colon \neg A\ . \\ \quad f\ k,\ k \end{array}$$

Entretanto, tal termo será rejeitado pelo algoritmo de checagem de tipos, visto que a continuação não é usada linearmente (nominalmente, a continuação k é duplicada).

Em (THIELECKE, 2003) nota-se que é possível o uso de um sistema de efeitos para, em tempo de compilação, verificar que funções são naturais, e que funções não são por usarem o operador clássico. De um ponto de vista de baixo nível, tal informação é relevante pois funções não-naturais não podem viver na pilha de chamada do *hardware* (a "pilha do C"). Trabalhos futuros incluem o estudo do cálculo de continuações onde continuações possam ser usadas como cidadãos de primeira classe, efetivamente permitindo que o operador clássico ilustrado acima seja aceito.

4.2 TIPOS DEPENDENTES E LINEARES

Alguns trabalhos recentes abordam sobre sistemas de tipos que integram o uso de tipos dependentes e tipos lineares (MCBRIDE, 2016; ATKEY, 2018). Uma óbvia pergunta surge rapidamente ao se trabalhar com a união de tais sistemas de tipo: se um termo linear deve aparecer exatamente uma única vez, é possível que ele seja usado em um tipo? Tal uso irá consumir o termo?

Um detalhe importante sobre resultados da literatura sobre a união de ambas categorias[25] de tipos é que não há a necessidade de se responder essa pergunta: termos lineares não podem aparecer em tipos; tal característica surge da relação entre a semântica dos sistemas de tipos conforme estudados pela teoria de categorias (KRISHNASWAMI; PRADIC; BENTON, 2015).

O cálculo de continuações, conforme apresentado, mantém essa característica. O construtor responsável pela existência de tipos dependentes, sigma, existe em dois formatos. Um deles é totalmente intuicionista, que pode ser interpretado como apresentado no começo do Capítulo 3. O segundo formato, entretanto, é sempre adicionado ao contexto linear, e seu segundo elemento é considerado linear;

[25] Trocadilho intencional.

desse jeito, o tipo Σx: T.U pode ser interpretado, fixando o tipo de resposta, como (Πx: T.U ( ) (. Note que o elemento x, de tipo T, existe no contexto intuicionista, podendo aparecer livre no tipo U (que por sua vez pode ser um tipo Σ também), porém o tipo U é tratado como linear, e, por sua posição à direita, não poderá ser usado como dependência para um tipo. O segundo elemento, e o par em si, entretanto, vivem no contexto linear.

A existência de tal tipo, um par dependente com um elemento intuicionista à esquerda e um elemento linear à direita, que por si só é um tipo linear, já foi previsto pela literatura. Em (KRISHNASWAMI; PRADIC; BENTON, 2015), tal tipo é chamado de tipo F, uma referência ao funtor F da categoria intuicionista para a categoria linear. Nessa definição, os tipos Σ apresentados na Seção 3.1 subsumam ambos tipos Σ tradicionais, intuicionistas, e tipos F.

Análogos a tipos F, (KRISHNASWAMI; PRADIC; BENTON, 2015) formaliza tipos G, uma referência ao funtor G da categoria linear para a categoria intuicionista. Para um tipo linear X, o tipo $G\ X$ representa um termo de tipo X fechado em relação ao contexto linear (isto é, ele não captura variáveis lineares de contextos superiores). No cálculo de construções, ao fixarmos um tipo de resposta, temos que o tipo ¬T é aproximadamente equivalente a G (T ( ); o parâmetro de abstrações lambda é acrescentado ao contexto linear e deve ser usado uma única vez (sendo ele uma continuação linear ou um tipo F), retornando (ou, de outro ponto de vista, jamais retornando, pois o mesmo seria absurdo), porém sendo linearmente fechado.

4.3 EXTENSÕES AO SISTEMA DE TIPOS

O sistema de tipos apresentado para o cálculo de continuações é relativamente simples, porém não apresenta algumas funcionalidades desejáveis para seu uso tanto como lógica quanto como linguagem intermediária para compiladores.

Embora a linguagem SSA/CFG seja funcional, como apresentada na Seção 2.6.3, por sua natureza costuma receber extensões imperativas para trabalhar com efeitos colaterais. Tipos lineares são uma das possíveis respostas para se trabalhar com efeitos colaterais em linguagens puramente funcionais, sem abrir mão da transparência referencial (WADLER, 1990).

O cálculo de continuações, conforme foi representado, apresenta pares dependentes intuicionistas (tipos Σ), e pares entre valores dependentes e lineares (tipos F). Em particular, um tipo com um tipo linear à esquerda e um tipo intuicionista à direita não é interessante; existe, entretanto, a possibilidade de pares entre dois tipos lineares.

O produto tensorial, representado pelo símbolo , representa um par entre dois tipos lineares; intuitivamente, conforme discutido acima, não faz sentido que o mesmo seja um tipo dependente. O produto tensorial existe na lógica linear e em alguams variantes do cálculo lambda linear, e seria uma extensão óbvia para o cálculo de continuações.

Quanto ao seu uso, a adição de parâmetros lineares, através do produto tensorial, no cálculo de continuações, permitiria o controle de recursos. Por exemplo, conforme (WADLER, 1990), um objeto linear pode representar o “mundo”, efetivamente permitindo alterações, por exemplo, na memória. A ideia é simples: uma operação de memória (leitura ou escrita) consume um mundo (que é um valor linear), retornando um par entre o resultado da operação e um novo mundo. Tal formalização força uma sequência de operações, preservando operações imperativas em um ambiente puramente funcional.

Um trabalho futuro considerado é a adição de produtos tensoriais ao cálculo de continuações. Além disso, uma oportunidade de trabalho futuro é o estudo das implicações de parâmetros lineares no cálculo, pois tais operações se assemelham à lógica de Hoare, uma conhecida forma de se verificar propriedades de programas imperativos (KRISHNASWAMI; PRADIC; BENTON, 2015).

O assistente de provas Coq, além de linguagens dependentemente tipadas como Idris e Agda, apresentam tipos indutivos e coindutivos como primitivos. Tais tipos efetivamente representam tipos recursivos, satisfazendo as equações $X = \mu X.\ T$ e $X = \nu X.\ T$ respectivamente, onde X pode aparecer livre em T.

Conforme discutido na Seção 2.4.1, tais tipos são usados em favor dos princípios indutivo e coindutivo, respectivamente. Embora traduções em CPS dos mesmos já tenham sido estudadas (BARTHE; UUSTALU, 2002), ao conhecimento do autor, não existem estudos sobre traduções em CPS de suas versões dependentes. Como trabalho futuro, o estudo da tradução em CPS de tipos indutivos para o cálculo de continuações aumentaria significativamente sua utilidade como lógica.

Em particular, tipos de soma, conforme mostrados na Seção 2.2.3, são um exemplo de tipos indutivos que representam uniões disjuntas; isto é, $A + B$ representa um termo que ou é um elemento de A, ou um elemento de B. Sua versão duplamente negada, então, seria “não é possível que não tenhamos A e não tenhamos B”, ou seja, temos o tipo $\neg(\neg A \times \neg B)$.

Tal conectiva já existe exatamente, e é presente dentro da lógica linear: a multiplicação disjuntiva, representada pelo símbolo $\parr$. O tipo $A \parr B$ é um tipo linear que recebe duas continuações lineares (ao invés de apenas uma), e executa uma delas, de acordo com o subtermo armazenado na criação do termo. Tal conectiva já foi pre-

viamente estudada em traduções de CPS (FILINSKI, 1992). Como trabalho futuro, a adição da multiplicação disjuntiva aumentaria a expressividade como lógica do cálculo de continuações, e sua usabilidade como linguagem intermediária, permitindo o uso de condicionais (`if`/`else`).

Por fim, como trabalho futuro, é proposto o estudo de codificações para sistemas de objetos. Não existe uma relação óbvia entre sistemas de tipos de linguagens orientadas de objetos para sistemas de lógica; entretanto, dado o seu uso em linguagens de programação, é motivado o estudo de traduções CPS para objetos, e suas implicações enquanto um sistema de lógica.

4.4 TIPOS Σ IMPREDICATIVOS

Conforme mencionado na Seção 2.2.4, impredicatividade em tipos Σ é uma conhecida fonte de inconsistência no CoC. Tal fato motiva o uso da eliminação fraca no cálculo de continuações apresentado: limitamos o uso de pares a termos, e os mesmos não podem ser eliminados dentro de tipos, similar ao que é possível fazer com a codificação impredicativa dentro do CoC apresentada na Seção 2.4.1.

Uma pergunta aberta, entretanto, é deixada: qual a relação dos tipos Π na prova de inconsistência? O cálculo de continuações não permite tipos Π, ou, em outros termos, não permite funções arbitrárias, tendo que retornar de uma função é um absurdo. Dada tal restrição ao sistema de tipos, onde funções não retornam, impredicatividade de tipos Σ ainda representaria um problema à consistência do cálculo?

Trabalhos futuros incluem o estudo de projeções dentro da linguagem dos tipos, e projeções fortes para tipos Σ impredicativos dentro de um cálculo de continuações. Nota-se, entretanto, que o uso de tipos Σ (mesmo que não impredicativos) pode acarretar o uso de regras não-convencionais para o sistema de tipo, embora ainda sejam possíveis (BOWMAN et al., 2018).

4.5 CONSISTÊNCIA E NORMALIZAÇÃO

Embora o autor conjecture que o cálculo de continuações, conforme apresentado na Seção 3.1, seja consistente (isto é, todas as funções terminem), baseado na sua similaridade à codificação impredicativa no CoC, tal afirmação não foi provada no primeiro momento. Trabalhos futuros incluem uma prova formal de terminação para o cálculo.

Entretanto, caso tal sistema apresente normalização, outro problema surge; tal propriedade não é desejável ao se considerar uma linguagem intermediária, visto que, no caso geral, linguagens de programação apresentam recursão geral.

O autor deixa em aberto a pergunta: como conciliar recursão geral em um cálculo de continuações? Possíveis soluções incluem o uso de um algoritmo de verificação de terminação, como na linguagem de programação Idris, e limitar reduções durante a execução do algoritmo de checagem de tipos para termos que provadamente terminam, e a verificação literal de igualdade para termos cuja terminação não pôde ser provada estaticamente.

Outra possibilidade considerada é o estudo da bissimilaridade de termos, similar a (FÜHRMANN; THIELECKE, 2004). Como termos coindutivos podem ser potencialmente infinitos, a noção tradicional de igualdade não pode ser considerada; existe a noção de bissimilaridade, isto é, a verificação que, tendendo ao infinito, ambos objetos sempre irão se comportar da mesma forma, sendo indistinguíveis ao observador. Tal possibilidade foi considerada em (JIA et al., 2010), conforme mencionado na Seção 2.2.5. Ao considerar recursão geral, uma pergunta aberta é se é possível a existência de um algoritmo de checagem de tipos que considere dois termos iguais usando bissimilaridade, a qual já foi previamente estudada no contexto de continuações (FÜHRMANN; THIELECKE, 2004) devido à relação entre um cálculo de continuações e um cálculo de processo (onde o uso de bissimilaridade foi amplamente estudado).

5 CONCLUSÕES

Linguagens de programação, tais quais usadas por programadores, não são adequadas para a fase de otimização em um compilador. Por tal motivo, compiladores traduzem a linguagem fonte para uma linguagem intermediária; entretanto, tais representações muitas vezes não são tipadas, perdendo informações existentes no programa original. Além de descartar possibilidades de otimização baseadas nos tipos, são criadas possibilidades de *bugs* na fase de *linking* do programa (BOWMAN et al., 2018).

Nessa pesquisa motivou-se a tradução em estilo de passagem de continuação para o cálculo de construções, a fim de preservar os tipos originais do programa (e, consequentemente, os teoremas por ele provado) e se manter em um formato adequado para manipulação por um compilador. Um cálculo de continuações, baseado no cálculo de construções, é apresentado, que permite a prova de teoremas através de dupla negação. A principal estratégia para isso é a eliminação de tipos Π em nível de termos, adicionando tipos Σ e tipos ¬ de negação primitivos no seu lugar, tal que os únicos termos que possam ser descritos são termos em estilo de passagem de continuação.

Resultados baseados em um protótipo de implementação do algoritmo de checagem de tipos sugerem a possibilidade de uma tradução mecânica do cálculo de construções para o cálculo de continuações, possibilitando seu uso como linguagem intermediária, preservando os tipos. Em particular, a tradução não utiliza tipos Π em nível de termos, e as continuações são usadas de forma linear (isto é, uma única vez).

Com isso, esse trabalho apresentou os passos iniciais para o desenvolvimento de uma linguagem intermediária com tipos dependentes e lineares, combinação que se mostra uma área ativa de pesquisa no momento (KRISHNASWAMI; PRADIC; BENTON, 2015; MCBRIDE, 2016; ATKEY, 2018). Além disso, também há recentes desenvolvimentos para uma compilação com preservação de tipos para linguagens como o cálculo de construções (BOWMAN et al., 2018; BOWMAN; AHMED, 2018).

Trabalhos futuros incluem a prova de propriedades (como normalização) do cálculo proposto, que foram conjecturadas e exploradas empiricamente. Além disso, para o uso como linguagem intermediária de propósito geral, extensões ao cálculo proposto que vão além dos necessários para a tradução do cálculo de construções se fazem necessários, como indução e disjunção.

REFERÊNCIAS

ABEL, A. Towards normalization by evaluation for the $\beta\eta$-calculus of constructions. In: **Functional and Logic Programming**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010. p. 224–239.

AHMED, A. Verified compilers for a multi-language world. In: **1st Summit on Advances in Programming Languages, SNAPL 2015, May 3-6, 2015, Asilomar, California, USA**. [S.l.: s.n.], 2015. p. 15–31.

AHO, A. V. et al. **Compilers: Principles, Techniques, and Tools (2Nd Edition)**. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2006.

APPEL, A. W. Ssa is functional programming. **SIGPLAN Not.**, ACM, New York, NY, USA, v. 33, n. 4, p. 17–20, abr. 1998. ISSN 0362-1340. Disponível em: <http://doi.acm.org/10.1145/278283.278285>.

ATKEY, R. Syntax and semantics of quantitative type theory. In: **Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science**. New York, NY, USA: ACM, 2018. (LICS '18), p. 56–65. ISBN 978-1-4503-5583-4. Disponível em: <http://doi.acm.org/10.1145/3209108.3209189>.

AUGUSTSSON, L. Cayenne—a language with dependent types. **SIGPLAN Not.**, ACM, New York, NY, USA, v. 34, n. 1, p. 239–250, set. 1998. ISSN 0362-1340. Disponível em: <http://doi.acm.org/10.1145/291251.289451>.

BARTHE, G.; HATCLIFF, J.; SØRENSEN, M. H. B. CPS translations and applications: The cube and beyond. **Higher Order Symbol. Comput.**, Kluwer Academic Publishers, Hingham, MA, USA, v. 12, n. 2, p. 125–170, set. 1999. ISSN 1388-3690. Disponível em: <https://doi.org/10.1023/A:1010000206149>.

BARTHE, G.; UUSTALU, T. CPS translating inductive and coinductive types. **SIGPLAN Not.**, ACM, New York, NY, USA, v. 37, n. 3, p. 131–142, jan. 2002. ISSN 0362-1340. Disponível em: <http://doi.acm.org/10.1145/509799.503043>.

BERDINE, J. et al. Linear continuation-passing. **Higher Order Symbol. Comput.**, Kluwer Academic Publishers, Hingham, MA, USA, v. 15, n. 2-3, p. 181–208, set. 2002. ISSN 1388-3690. Disponível em: <https://doi.org/10.1023/A:1020891112409>.

BERNARDY, J. et al. Linear Haskell: practical linearity in a higher-order polymorphic language. **CoRR**, abs/1710.09756, 2017. Disponível em: <http://arxiv.org/abs/1710.09756>.

BOWMAN, W. J.; AHMED, A. Compiling dependent types without continuations. 2018. Disponível em: <https://www.williamjbowman.com/downloads/wjb-paper-anf-sigma.pdf>.

BOWMAN, W. J. et al. Type-preserving cps translation of Σ and Π types is not not possible. **PACMPL**, v. 2, n. POPL, jan. 2018. Disponível em: <https://www.williamjbowman.com/resources/cps-sigma.pdf>.

BRADY, E. Idris, a general-purpose dependently typed programming language: Design and implementation. **Journal of Functional Programming**, v. 23, p. 552–593, 9 2013. ISSN 1469-7653. Disponível em: <http://journals.cambridge.org/article_S095679681300018X>.

CARDELLI, L. Type systems. **ACM Comput. Surv.**, 1996.

CHEN, R. **The C language specification describes an abstract computer, not a real one**. 2013. Disponível em: <https://blogs.msdn.microsoft.com/oldnewthing/20130328-00/?p=4823>. Acesso em: 25/06/2017.

CHISNALL, D. C is not a low-level language. **Queue**, ACM, New York, NY, USA, v. 16, n. 2, 2018. ISSN 1542-7730. Disponível em: <https://queue.acm.org/detail.cfm?id=3212479>. Acesso em: 12/08/2018.

CHURCHILL, M.; LAIRD, J.; MCCUSKER, G. Imperative programs as proofs via game semantics. **2011 IEEE 26th Annual Symposium on Logic in Computer Science**, p. 65–74, 2011.

CYTRON, R. et al. Efficiently computing static single assignment form and the control dependence graph. **ACM Trans. Program. Lang. Syst.**, ACM, New York, NY, USA, v. 13, n. 4, p. 451–490, out. 1991. ISSN 0164-0925. Disponível em: <http://doi.acm.org/10.1145/115372.115320>.

ERTL, M. A. What every compiler writer should know about programmers. In: **18. Kolloquium Programmiersprachen und Grundlagen der Programmierung (KPS'13)**. [S.l.: s.n.], 2015.

FEATHER, C. D. W. **Defect Report #260**. 2004. Disponível em: <http://www.open-std.org/jtc1/sc22/wg14/www/docs/dr_260.htm>. Acesso em: 25/06/2017.

FILINSKI, A. Linear continuations. In: **Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages**. New York, NY, USA: ACM, 1992. (POPL '92), p. 27–38. Disponível em: <http://doi.acm.org/10.1145/143165.143174>.

FLANAGAN, C. et al. The essence of compiling with continuations. **SIGPLAN Not.**, ACM, New York, NY, USA, v. 28, n. 6, p. 237–247, jun. 1993. ISSN 0362-1340. Disponível em: <http://doi.acm.org/10.1145/173262.155113>.

FLUET, M.; WEEKS, S. Contification using dominators. **SIGPLAN Not.**, ACM, New York, NY, USA, v. 36, n. 10, p. 2–13, out. 2001. ISSN 0362-1340. Disponível em: <http://doi.acm.org/10.1145/507546.507639>.

FÜHRMANN, C.; THIELECKE, H. On the call-by-value CPS transform and its semantics. **Inf. Comput.**, Academic Press, Inc., Duluth, MN, USA, v. 188, n. 2, p. 241–283, jan. 2004. ISSN 0890-5401. Disponível em: <http://dx.doi.org/10.1016/j.ic.2003.08.001>.

JIA, L. et al. Dependent types and program equivalence. **SIGPLAN Not.**, ACM, New York, NY, USA, v. 45, n. 1, p. 275–286, jan. 2010. ISSN 0362-1340. Disponível em: <http://doi.acm.org/10.1145/1707801.1706333>.

KELSEY, R. A. A correspondence between continuation passing style and static single assignment form. **SIGPLAN Not.**, ACM, New York, NY, USA, v. 30, n. 3, p. 13–22, mar. 1995. ISSN 0362-1340. Disponível em: <http://doi.acm.org/10.1145/202530.202532>.

KENNEDY, A. Compiling with continuations, continued. **SIGPLAN Not.**, ACM, New York, NY, USA, v. 42, n. 9, p. 177–190, out. 2007. ISSN 0362-1340. Disponível em: <http://doi.acm.org/10.1145/1291220.1291179>.

KRISHNASWAMI, N. R.; PRADIC, P.; BENTON, N. Integrating linear and dependent types. **SIGPLAN Not.**, ACM, New York, NY, USA, v. 50, n. 1, p. 17–30, jan. 2015. ISSN 0362-1340. Disponível em: <http://doi.acm.org/10.1145/2775051.2676969>.

LEROY, X. A formally verified compiler back-end. **J. Autom. Reason.**, Springer-Verlag New York, Inc., Secaucus, NJ, USA, v. 43, n. 4, p. 363–446, dez. 2009. ISSN 0168-7433. Disponível em: <http://dx.doi.org/10.1007/s10817-009-9155-4>.

MAURER, L. et al. Compiling without continuations. **SIGPLAN Not.**, ACM, New York, NY, USA, v. 52, n. 6, p. 482–494, jun. 2017. ISSN 0362-1340. Disponível em: <http://doi.acm.org/10.1145/3140587.3062380>.

MCBRIDE, C. I got plenty o' nuttin'. In: **A List of Successes That Can Change the World**. [S.l.: s.n.], 2016.

MORRISETT, G. et al. From System F to typed assembly language. **ACM Trans. Program. Lang. Syst.**, ACM, New York, NY, USA, v. 21, n. 3, p. 527–568, maio 1999. ISSN 0164-0925. Disponível em: <http://doi.acm.org/10.1145/319301.319345>.

O'CONNOR, R. Simplicity: A new language for blockchains. **Proceedings of the 2017 Workshop on Programming Languages and Analysis for Security**, abs/1711.03028, 2017.

PIERCE, B. C. **Types and Programming Languages**. [S.l.]: MIT Press, 2002.

PIERCE, B. C. (Ed.). **Advanced Topics in Types and Programming Languages**. [S.l.]: MIT Press, 2005.

ROY, P. V. **Programming Paradigms for Dummies: What Every Programmer Should Know**. 2012.

SJÖBERG, V.; WEIRICH, S. Programming up to congruence. **SIGPLAN Not.**, ACM, New York, NY, USA, v. 50, n. 1, p. 369–382, jan. 2015. ISSN 0362-1340. Disponível em: <http://doi.acm.org/10.1145/2775051.2676974>.

THIELECKE, H. From control effects to typed continuation passing. **SIGPLAN Not.**, ACM, New York, NY, USA, v. 38, n. 1, p. 139–149, jan. 2003. ISSN 0362-1340. Disponível em: <http://doi.acm.org/10.1145/640128.604144>.

TURNER, D. A. Total functional programming. **Journal of Universal Computer Science**, 2004.

Univalent Foundatiosn Program, T. **Homotopy Type Theory: Univalent Foundations of Mathematics**. <http://homotopytypetheory.org/book/>, 2013. Disponível em: <http://homotopytypetheory.org/book/>.

WADLER, P. Linear types can change the world! In: **PROGRAMMING CONCEPTS AND METHODS**. [S.l.]: North, 1990.

WADLER, P. Call-by-value is dual to call-by-name. **SIGPLAN Not.**, ACM, New York, NY, USA, v. 38, n. 9, p. 189–201, ago. 2003. ISSN 0362-1340. Disponível em: <http://doi.acm.org/10.1145/944746.944723>.

ZHAO, J. et al. Formalizing the LLVM intermediate representation for verified program transformations. **SIGPLAN Not.**, ACM, New York, NY, USA, v. 47, n. 1, p. 427–440, jan. 2012. ISSN 0362-1340. Disponível em: <http://doi.acm.org/10.1145/2103621.2103709>.