UNIVERSIDADE FEDERAL DE OURO PRETO
INSTITUTO DE CIÊNCIAS EXATAS E BIOLÓGICAS
DEPARTAMENTO DE COMPUTAÇÃO

ARTUR BERMOND TORRES

STOCHASTICALLY GENERATING WELL-TYPED CPS-CALCULUS TERMS: A CONTEXT-DIRECTED ALGORITHM

Ouro Preto
2026

ARTUR BERMOND TORRES

STOCHASTICALLY GENERATING WELL-TYPED CPS-CALCULUS TERMS: A CONTEXT-DIRECTED ALGORITHM

Monografia apresentada ao Curso de Ciência da Computação da Universidade Federal de Ouro Preto como parte dos requisitos necessários para a obtenção do grau de Bacharel em Ciência da Computação.

Orientador: Prof. Dr. Rodrigo Geraldo Ribeiro
Coorientador: Me. Paulo Torrens

Ouro Preto
2026

SISBIN - SISTEMA DE BIBLIOTECAS E INFORMAÇÃO

T693s Torres, Artur Bermond.
Stochastically generating well-typed CPS-calculus terms [manuscrito]: a context-directed algorithm. / Artur Bermond Torres. - 2026.
135 f.: il.: color., gráf..

Orientador: Prof. Dr. Rodrigo Geraldo Ribeiro.
Coorientador: Prof. Me. Paulo Torrens.
Monografia (Bacharelado). Universidade Federal de Ouro Preto. Instituto de Ciências Exatas e Biológicas. Graduação em Ciência da Computação .

1. Compiladores (Programas de computador). 2. Programação (Computadores). 3. Algorítmos computacionais. I. Ribeiro, Rodrigo Geraldo. II. Torrens, Paulo. III. Universidade Federal de Ouro Preto. IV. Título.

CDU 004.43

Bibliotecário(a) Responsável: Renata Mara de Almeida - CRB-7: 6328

MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DE OURO PRETO
REITORIA
INSTITUTO DE CIENCIAS EXATAS E BIOLOGICAS
DEPARTAMENTO DE COMPUTACAO

FOLHA DE APROVAÇÃO

Artur Bermond Torres

STOCHASTICALLY GENERATING WELL-TYPED CPS-CALCULUS TERMS:

A CONTEXT-DIRECTED ALGORITHM

Monografia apresentada ao Curso de Ciência da Computação da Universidade Federal
de Ouro Preto como requisito parcial para obtenção do título de
Bacharel em Ciência da Computação

Aprovada em 03 de Março de 2026

Membros da banca

Dr. Rodrigo Geraldo Ribeiro - Orientador(a) - Universidade Federal de Ouro Preto
Dr. - Elton Máximo Cardoso - Universidade Federal de Ouro Preto
MSc. Guilherme Augusto Anício Drummond do Nascimento - Universidade Federal de Ouro Preto

Rodrigo Geraldo Ribeiro, orientador do trabalho, aprovou a versão final e autorizou seu depósito na Biblioteca Digital de Trabalhos de Conclusão de Curso da UFOP em 04/03/2026

Documento assinado eletronicamente por **Rodrigo Geraldo Ribeiro**, **PROFESSOR 3 GRAU**, em 04/03/2026, às 07:24, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

A autenticidade deste documento pode ser conferida no site http://sei.ufop.br/sei/controlador_externo.php?acao=documento_conferir&id_orgao_acesso_externo=0 , informando o código verificador **1068897** e o código CRC **FF8880D7**.

R. Diogo de Vasconcelos, 122, - Bairro Pilar Ouro Preto/MG, CEP 35402-163
Telefone: 3135591692 - www.ufop.br

In loving memory of my mom

Program testing can be used to show the presence of bugs, but never to show their absence!

— Edsger W. Dijkstra (DIJKSTRA, 1970)

Resumo

Continuações são frequentemente empregadas no contexto de compiladores, em especial como fundamento para representações intermediárias baseadas em *continuation-passing style* (CPS). Um campo emergente de pesquisa é o desenvolvimento de um cálculo baseado em CPS, visando-se criar uma base teórica para essas representações intermediárias. O trabalho atual tem o objetivo de contribuir a essa linha de pesquisa implementando uma variante de CPS-calculus em PLT Redex, formalizando um gerador de termos bem-tipados para essa linguagem e apresentando uma maneira de unir esses dois aspectos do trabalho em Racket, programando-se propriedades em Redex e depois testando-as usando o gerador de termos implementado em Rackcheck. Primeiramente, diversas semânticas de CPS-calculus foram implementadas, possibilitando o teste de reduções, tipagem e traduções entre CPS-calculus e λ-calculus, tanto por *call-by-name* quanto por *call-by-value*. Após, um gerador estocástico de termos bem-tipados foi formalizado usando-se árvores de provas e pseudo-código puramente funcional e depois implementado em Rackcheck, que possui suporte a *shrinking* automático de termos. Por último, foram testadas algumas propriedades sobre a imagem do gerador, além da propriedade de preservação de tipos da semântica operacional *jump reduction* e das traduções de λ-calculus para CPS-calculus. Com todos esses desenvolvimentos, o trabalho aqui apresentado obteve êxito em criar um caminho para possibilitar testes baseados em propriedades de CPS-calculus, algo que poderá ser de utilidade a futuras explorações acadêmicas no tópico, em especial como um prelúdio para provas formais que visa diminuir o tempo gasto com tentativas de se provar algo falso.

Palavras-chave: CPS-calculus. Testes baseados em propriedades. Geração aleatória de programas bem-tipados.

Abstract

Continuations are frequently employed in the context of compilers, specially as the foundation for intermediary representations based on continuation-passing-style (CPS). An emerging field of study is in the development of a calculus based on CPS, with the goal of creating a theoretical base for these intermediary representations. The present work aims to contribute to this field of research by implementing a variant of CPS-calculus in PLT Redex, formalizing a generator of well-typed CPS-calculus terms and then joining these two aspects of the work in Racket, implementing properties in Redex and then testing them using the generator implemented in Rackcheck. First, several operational semantics were implemented, allowing reductions, typing and translations to and from CPS-calculus and λ-calculus, both in call-by-name and call-by-value, to be tested. Next, a stochastic generator of well-typed terms was formalized through judgments and purely functional pseudo-code and then implemented using Rackcheck, which has support for automatic shrinking of terms. Lastly, some properties about the generator's image were tested, alongside the type preservation property of the operational semantics jump reduction and of the translations from λ-calculus to CPS-calculus. With these developments, the work here presented was successful in creating a path for property-based testing to take place on CPS-calculus, something that may be useful to future academic endeavors on the topic, specially as a prelude to formal proofs, aiming to reduce the time spent with attempts to prove a false property.

Keywords: CPS-calculus. Property-based testing. Random generation of well-typed programs.

Lista de Abreviaturas e Siglas

ABNT	Associação Brasileira de Normas Técnicas
DECOM	Departamento de Computação
UFOP	Universidade Federal de Ouro Preto
CPS	Continuation-passing style
IR	Intermediate Representation
CFG	Context-free grammar
BNF	Backus–Naur form
FV	Free variables
DAG	Directed Acyclic Graph
CBN	Call by Name
CBV	Call by Value
DSL	Domain-specific language
ICFP	International Conference on Functional Programming
w.l.o.g	Without loss of generality
i.e.	*id est* (“as is”)

Lista de Símbolos

$\equiv$	Equivalent
$\not\equiv$	Not equivalent
$\approx$	Bissimilar
$\in$	Is in
$\notin$	Is not in
$\cup$	Set union
$\cap$	Set intersection
$\setminus$	Set difference
$\forall$	For all
$\mapsto$	Maps to
$\top$	Logical true
$\bot$	Logical false
$\neg$	Logical negation
$\vee$	Logical disjunction
$\wedge$	Logical conjunction
$\longrightarrow$	Logical implication
$\rightarrow$	Small-step reduction relation
$\Downarrow$	Big-step reduction relation
:	Typing relation
$\square$	Empty
$[\![\ ]\!]$	λ-calculus translations into CPS

Sumário

1 Introduction

This work seeks to explore the potential utility of randomly generated programs for automated property-based testing of Continuation Passing Style (CPS) calculus' semantics, and to formalize a correct and diverse stochastic generator of well-typed CPS-calculus terms. This first chapter contains the motivations behind this goal, alongside precise objectives that should be achieved by the end of this project. It also lays out the structure for the rest of this text.

1.1 Motivations

Computer scientists are invited to look not only at the end-user software that can be implemented to fulfill a specific task, from the higher-level database management systems to the lower-level compression methods they use, but also at the mathematical models behind the algorithms themselves. For instance, what makes an algorithm possible or not possible? What are the limitations of certain models of computation? How do different models relate? What do they allow us to do? (CUTLAND, 1980)

Related to that, it's within computer science's attributions to research, analyze and develop compilers, tools that are able to translate code from one language to another, often (but not always) with the objective of ending the process at executable machine instructions. It's the compiler's job to bridge the gap between human understandable logic and machine understandable instructions, and as such they play an integral part of high-level programming languages (ALFRED; MONICA; JEFFREY, 2007).

In the context of mathematical models of computation and compiler implementations, we find fields such as programming language theory and compiler theory. These areas are central not only to the work here presented, but also to any work involving programming, and as such, related academic advancements may have an impact in all fields of study that require, or at least benefit from, computer programs.

This work relates to the study of the *continuation-passing style calculus* (CPS-calculus) (THIELECKE, 1997b), a formal system of computation. It is based on the *continuation-passing style* (CPS), a method of programming that make up the foundation behind some functional compilers (APPEL, 2007). As a calculus, it may be used as a theoretical foundation for an intermediate representation (IR) language to be used in compilers, due to its ability to extract an imperative program from a functional one (TORRENS; ORCHARD; VASCONCELLOS, 2024). When used in this context, it would aid in the process of creating a common IR for these two paradigms, so that optimizations may be conceived with a lesser loss of generality.

Advancements on this area have the potential to create an impact in compiler theory, as it

may allow for the intuitive unification of functional and procedural paradigms, something that alludes back to λ-calculus' equivalence relationship with Turing machines (TURING, 1937).

1.2 Objectives

This work's main objetive is to assist in the academic development of CPS-calculus as a theoretical foundation for compiler IRs. The path through which this goal is to be pursued is an investigation on the usefulness of automated property-based testing for a certain set of operational semantics acting on the CPS-calculus.

More specifically, this work aims to correctly implement a specification of the CPS-calculus in Redex, a Racket domain-specific language used to formalize, test and debug operational semantics (FELLEISEN; FINDLER; FLATT, 2009). The chosen specification is the one defined in Kennedy (2007), which is polyadic in its parameters and non-recursive. Using that implementation, it's desired to:

- Implement the reduction semantics specified for that variant in Torrens, Orchard e Vasconcellos (2024).
- Implement the typing system defined in Thielecke (1997a).
- Implement the translations to and from λ-calculus and CPS-calculus, in both call-by-name and call-by-value forms, and in both untyped and typed semantics. These translations are defined in Thielecke (1997a) and Torrens, Orchard e Vasconcellos (2024).
- Test the type preservation property of the jump reduction (TORRENS; ORCHARD; VASCONCELLOS, 2024).
- Test the type preservation property of the translation from simply-typed λ-calculus to simply-typed CPS-calculus (TORRENS; ORCHARD; VASCONCELLOS, 2024).

In order to achieve these goals, it is also within this work's scope to develop a stochastic generator of well-typed and well-syntaxed CPS-calculus terms. This generator's image should be as comprehensible as possible while still avoiding useless terms. Property tests of this generator should be developed as well.

1.3 Structure of this work

After this introductory chapter, this work is going to partake in an extensive bibliographic review, where every concept and tool used in this project is going to be explained and explored. Afterwards, in that same chapter, related works will be listed and subjected to having their similarities and differences to this work highlighted.

Then, a polyadic CPS-calculus specification and some of its semantics are going to be mechanized in PLT Redex. This will offer a mean to execute, visualize and test semantics of CPS-calculus and λ-calculus. That chapter will heavily reference corresponding sections of the theoretical foundations as it showcases their implementation, as to avoid having to repeat a lot of information.

Afterwards, there's going to be a chapter formalizing a stochastic generation algorithm for well-typed terms in CPS-calculus. That chapter will not have any real code, instead being purely focused in formalizing an algorithm through judgments or, sometimes, purely functional Haskell-like pseudo-code. The generator formalized is already implemented, but its code is omitted from the main text due to its size.

After a specification of CPS-calculus is mechanized and a generator for it is formalized and implemented, there's going to be a chapter that connects these two units by performing several tests of properties. Properties that have already been proven correct in the literature will be tested, not to try to find counter-examples to the properties themselves, but to test the language mechanization and the generator implementation. Properties about the generator's image will also be implemented.

Following from these tests, a chapter regarding the obtained results can be found. This chapter will include not only a summary of whether some counter-examples were found or not, but also statistics regarding the generator, such as sensibility to arguments with regards to term size, among others.

Finally, there's going to be a chapter containing the final remarks for this work. This chapter's goal is to give an overview of the academic advancements this project was able to achieve and also give prospects for future work.

Every chapter, including the current one, is going to have an introduction before the first section, one that is meant to give an overview of what the chapter aims to achieve without anachronistically mentioning information that is about to but have yet to be explored. Every chapter, but this time except for the current one, is also going to have a conclusion section at the end, where a summary of what was just done will be given, this time allowing for the recently explored concepts or details to be mentioned.

1.3.1 List of chapters

Chapter 1: Introduction.

Chapter 2: Background

Chapter 3: CPS-calculus Implementation

Chapter 4: CPS-calculus Generator

2 Background

In this chapter, we are first going to explore the theoretical foundations required to fully understand the developments provided throughout the remainder of this work. These foundations will include basic concepts, such as explanations for what context-free grammars and operational semantics are, and also more specific subjects to this work, such as what is the Continuation Passing Style Calculus and how it operates under certain semantics, and also what are some of the tools used in this work, such as Racket, PLT Redex and Rackcheck.

After that, related works are going to be discussed, divided in two subsections. First, projects pertaining to the mechanization of languages and their operational semantics in PLT Redex will be listed and explored. Second, the same will be done to works around the subject of stochastic generation of well-typed terms of a language for automated property-based testing of compilers or operational semantics.

2.1 Theoretical foundations

2.1.1 Context-free grammars

Context-free grammars (CFG) are a type of grammar whose production rules do not take into consideration the context in which a non-terminal symbol is found. One of their applications in computer science is the formalization of programming language syntaxes, which are then used to make parsers in compilers (ALFRED; MONICA; JEFFREY, 2007).

For every context-free grammar, there is a non-deterministic push-down automaton that is able to process it (HOPCROFT, 2001). Typically, a representation of that push-down automaton can be found within parsers, usually in the form of a stack and a pair of action and goto tables (ALFRED; MONICA; JEFFREY, 2007).

In order to represent CFGs throughout the remainder of this work, the Backus–Naur Form (BNF) will be used. The BNF is a way of writing CFGs such that every production rule is written as $[Nonterminal] ::= [Expression]$.

A non-terminal symbol is **not** a part of the grammar's alphabet, and instead serves as an identifier for the rules they map. This is in contrast with terminal symbols, which **are** present in the alphabet, and therefore actually represent a part of the language. An expression is a concatenation of zero or more symbols, which can be non-terminals or terminals.

A non-terminal can be deemed **nullable** if it can produce λ, a special symbol which represents an empty sequence. This happens if at least one of this non-terminal's expressions is a sequence of exactly zero terminals and all of its non-terminals are also nullable (HOPCROFT,

2001). If a non-terminal has more than one production rule, the BNF allows for the usage of the pipe symbol | between multiple expressions, each representing a possible production derivable from said non-terminal. Below is an example of a CFG in BNF, where non-terminals are enclosed between angle brackets and terminals are not:

$$\begin{aligned}\langle E\rangle ::=\ & \langle A\rangle \vee \langle A\rangle \\ |\ & \langle A\rangle \wedge \langle A\rangle \\ |\ & \langle A\rangle\end{aligned} \qquad \begin{aligned}\langle A\rangle ::=\ & \neg\ \langle A\rangle \\ |\ & (\ \langle E\rangle\ ) \\ |\ & \langle B\rangle\end{aligned} \qquad \begin{aligned}\langle B\rangle ::=\ & \top \\ |\ & \bot\end{aligned}$$

This grammar is able to produce boolean logic terms, with support for conjunction ($\wedge$), disjunction ($\vee$), negation ($\neg$) and parentheses.

2.1.2 Evaluation and Reduction

Evaluation and reduction are the processes of performing calculations on terms of a language by following a set of well-defined rules. These two terms can be treated interchangeably here, but some authors do differentiate them by choosing to use "reduction" for small-step semantics and "evaluation" for big-step semantics (PIERCE, 2002). More on small-step and big-step semantics soon.

As an example of evaluation, consider a grammar B that produces simple boolean expressions. Reducing $\bot \vee \top$ (read: "false or true"), matched as a term of the language B, could result in the new term $\top$ after following one *computation step*. In order to do so, one would need a *reduction relation* for the logical disjunction (the OR operator). A reduction relation is just a set of *reduction rules* that give instructions on how to process a term. Such a set is provided below for the disjunction example, using B as a variable and with the name of each rule being given between parentheses on the right-hand side:

$$\frac{}{\top \vee B \to \top}\ (\vee\text{—}True) \qquad \frac{}{\bot \vee B \to B}\ (\vee\text{—}False)$$

It's common to use an arrow to represent a reduction relation, usually a left-to-right simple arrow ($\rightarrow$) in small-step relations and a top-to-bottom double arrow ($\Downarrow$) in big-step relations; although the exact type of arrow may vary. When used as a midfix, it intuitively shows the original term t on the left and the new term t' on the right, with the second being the result of running the relation through the first. This is often used not only as a way of *defining* the reduction rules, which in small-step is often done through pattern matching on the original term, like here, but also to create the relation's *evaluation statements* (PIERCE, 2002). Examples of rule definitions are already provided above, but to give an example of how the arrow is used to create statements, we can represent the same idea of $\bot \vee \top$ resulting in $\top$ simply as $\bot \vee \top \to \top$, which is a statement (also called a *judgment*) under the $\rightarrow$ relation. On this note, $\bot \vee \top \to \top$ is a *derivable* statement, because there is a rule in $\rightarrow$ that will return $\top$ when given $\bot \vee \top$ (in this case, $\vee\text{—}False$).

Another statement under $\rightarrow$ could be $\bot \vee \top \rightarrow \bot$, but this statement is **not** derivable, as there is no rule that can produce it (PIERCE, 2002).

Although the provided rules can be a good way of understanding the concept of evaluation, they alone can only be used to represent non-recursive terms, which is hardly useful in practice. For example, how would that relation process the term $(\top \vee \top) \vee \bot$? The answer is that it wouldn't, because it cannot match this term with any of its rules. More specifically, although the sub-term $\top \vee \top$ could be matched with the B in the reduction rules if it were on the right-hand side, it cannot match with neither the constants $\top$ nor $\bot$ present on the left-hand side, where the sub-term is actually located. As such, the evaluation cannot happen.

To fix this, a third, recursive rule, needs to be introduced:

$$\frac{A \rightarrow A'}{A \vee B \rightarrow A' \vee B} \ (\vee\text{—}Cong)$$

This rule, unlike the first two, requires a premise to be true in order for it to be applicable. What this rule does is it allows for reductions on sub-terms to happen, and it does so by stating that the original $A \vee B$ evaluates to $A' \vee B$ if $A \rightarrow A'$ is derivable, allowing a computation step to be done on the sub-term A. Now, when putting this rule to use in the given example $(\top \vee \top) \vee \bot$, the newly modified relation will now reduce this term to $\top \vee \bot$, because $\top \vee \top \rightarrow \top$ is derivable and therefore matches the premise. The name of the rule $(\vee\text{—}Cong)$ comes from the fact that reduction rules with premises are sometimes called *congruence rules*, as opposed to those without premises, which are called *computation rules* (PIERCE, 2002).

The relation described here is a small-step relation, meaning it reduces terms one step at a time (PIERCE, 2002). The arrow notation $\rightarrow$ indicates a single reduction, so $(\top \vee \top) \vee \bot \rightarrow \top \vee \bot$ is derived by just one computational step. Sometimes, however, it may be useful to represent the process of recursively applying the relation $\rightarrow$ until it is no longer possible, and such idea is expressed by super-scripting a star to the right of the arrow. Therefore, a multi-step $\rightarrow$ is written as $\rightarrow^\star$. With this, $(\top \vee \top) \vee \bot \rightarrow^\star \top$ is now valid.

2.1.2.1 Derivation trees

The syntax presented here is still very simple, but with the introduction of a congruence rule, proof trees start to become very useful to prove the derivability of terms. As an example, consider a slightly more complex term than the one just given above: $((\top \vee \top) \vee \bot) \vee \bot$. The proof tree below proves that the evaluation statement $((\top \vee \top) \vee \bot) \vee \bot \rightarrow (\top \vee \bot) \vee \bot$ is derivable by showing a valid *derivation tree* for it (PIERCE, 2002):

$$\dfrac{\dfrac{\dfrac{}{\top \vee \top \rightarrow \top}\ (\vee\text{—}True)}{(\top \vee \top) \vee \bot \rightarrow \top \vee \bot}\ (\vee\text{—}Cong)}{((\top \vee \top) \vee \bot) \vee \bot \rightarrow (\top \vee \bot) \vee \bot}\ (\vee\text{—}Cong)$$

This tree is showing that the relation works by repeatedly applying the congruence rule to the sub-statements until a computation rule is matched. When this happens, the process ends, and the proof is done, confirming that the provided evaluation statement is derivable. If, however, the right-hand side of the provided statement was a term different than $(\top \vee \bot) \vee \bot$, but the left-hand side was the same, then there would be no way of finishing this derivation tree, because at some level there would be no rule capable of matching with the sub-statement. As an example of that, consider the statement $((\top \vee \top) \vee \bot) \vee \bot \rightarrow (\top \vee \top) \vee \bot$. If we try to build the derivation tree, we get:

$$\dfrac{\dfrac{\text{!!UNDERIVABLE!!}}{(\top \vee \top) \vee \bot \rightarrow \top \vee \top}}{((\top \vee \top) \vee \bot) \vee \bot \rightarrow (\top \vee \top) \vee \bot} (\vee\text{—}Cong)$$

This proof can go no further than the first congruence rule, because in order to match the conclusion of the rule a second time, we need both right-hand sides of the $\vee$ operator to be equal, as per the congruence rule's definition. To show what this means, the rule's conclusion reads $A \vee B \rightarrow A' \vee B$, so the term bound by B needs to be the same on both sides for the pattern matching to work, but the left-hand side has $\bot$ in the place of B and the right-hand side has $\top$ in the place of B, and these are, clearly, different terms. Therefore, as no rules apply and no computation rule has been reached, the provided statement is **not** derivable.

Lastly, it may be useful to think of these trees not only as a proof of derivability, but as a tool to reason about the execution of an evaluation relation when it is not initially known what is the term t_2 that the original term t_1 evaluates to.

2.1.2.2 Normal form and values

When a term can no longer be reduced by a relation, it is said to be in **normal form** (PIERCE, 2002).

Frequently, the goal of evaluation is to correctly reduce a term all the way down into an appropriate **value**. The subset of terms that can be considered values is language-specific, meaning there is no universal rule to define what is or isn't a value in all languages. For example, the language for booleans described above could define its values as being either $\top$ or $\bot$, and so any other term of the language that is neither of these terms, regardless of whether or not they are in normal form, are not values (PIERCE, 2002).

If no reduction rules can be applied to a term (i.e. it's in normal form), but that term is not a value, then the computation process is said to be *stuck*. Usually when this happens in a programming language, it results in what is known as a *Runtime Error*. This can happen for various reasons, including a poorly-defined language semantics, a faulty implementation of a semantically well-defined language or (most commonly) a term that is outside of a reduction's domain (for example, attempting to divide by zero). The last reason can be mitigated, although

rarely fully eliminated, through a robust *typing system*, replacing some runtime errors with errors that can be reported at compilation time (PIERCE, 2002).

2.1.2.3 Operational semantics

Operational semantics is one of the semantic styles through which we can formalize the behavior of a language. This style of formalization specifies an abstract machine, such that the state of the machine is usually given by a term of the language and its transition functions either perform a step of simplification on the term of the current state, moving the machine to a new one, or halt the machine, by moving it into a final state with no transitions (PIERCE, 2002).

Consider the untyped arithmetic expressions language defined in Pierce (2002), named *exprU* in the present work, where the *U* stands for "Untyped". It is defined by the following CFG, such that the non-terminal $\langle t \rangle$ represents the language's terms and the non-terminals $\langle v \rangle$ and $\langle nv \rangle$ its values:

$\langle t \rangle$ ::= true
| false
| if $\langle t \rangle$ then $\langle t \rangle$ else $\langle t \rangle$
| zero
| succ $\langle t \rangle$
| pred $\langle t \rangle$
| iszero $\langle t \rangle$

$\langle v \rangle$ ::= true
| false
| $\langle nv \rangle$

$\langle nv \rangle$ ::= zero
| succ $\langle nv \rangle$

In this language, *true*, *false* and *zero* are all constants, and *if*, *succ*, *pred* and *iszero* are all functions that need to take more terms to give results, as is reflected through the recursions on $\langle t \rangle$ in the grammar above.

There are two main styles of operational semantics, namely **Small-step** and **Big-step**. What follows is a brief explanation of each strategy and also a list of the evaluation rules provided by Pierce (2002) to formalize *exprU*'s small-step and big-step semantics.

Small-step semantics perform reductions one simplification at a time, such that the outcome of the simplification isn't necessarily a value of the language, meaning that the output term may or may not be able to get simplified again. This shows every change the original term goes through until it becomes a value (PIERCE, 2002). Usually, small-step semantics use the syntax $t_1 \rightarrow t_2$ to mean "t_1 reduces to t_2 in one step" and $t_1 \rightarrow^\star t_2$ to signify "t_1 reduces to t_2 in zero or more steps such that t_2 is in normal form". It comes from both definitions that the single-step $t_1 \rightarrow t_2$ reduction will fail if t_1 is already in normal form, but $t_1 \rightarrow^\star t_2$ will not, as performing zero steps is valid for the second but not for the first.

Below is the small-step reduction relation for the language *exprU* (PIERCE, 2002):

$$\frac{}{\text{if } true \text{ then } t_2 \text{ else } t_3 \rightarrow t_2} \text{ (E-IfTrue)} \qquad \frac{}{\text{pred (succ } nv_1) \rightarrow nv_1} \text{ (E-PredSucc)}$$

$$\frac{}{\text{if } false \text{ then } t_2 \text{ else } t_3 \rightarrow t_3} \text{ (E-IfFalse)} \qquad \frac{t_1 \rightarrow t_1'}{\text{pred } t_1 \rightarrow \text{pred } t_1'} \text{ (E-Pred)}$$

$$\frac{t_1 \rightarrow t_1'}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 \rightarrow \text{if } t_1' \text{ then } t_2 \text{ else } t_3} \text{ (E-If)} \qquad \frac{}{\text{iszero } zero \rightarrow true} \text{ (E-IsZeroZero)}$$

$$\frac{t_1 \rightarrow t_1'}{\text{succ } t_1 \rightarrow \text{succ } t_1'} \text{ (E-Succ)} \qquad \frac{}{\text{iszero (succ } nv_1) \rightarrow false} \text{ (E-IsZeroSucc)}$$

$$\frac{}{\text{pred } zero \rightarrow zero} \text{ (E-PredZero)} \qquad \frac{t_1 \rightarrow t_1'}{\text{iszero } t_1 \rightarrow \text{iszero } t_1'} \text{ (E-IsZero)}$$

Now, big-step semantics will directly represent the idea of a term evaluating into a value, and it does so in just **one** step. The derivation trees that result from this type of semantic are not meant to prove that the original term t_1 is able to perform one specific simplification, but that it is able to converge into a value by following a chain of simplifications (PIERCE, 2002). The syntax often used to represent big-step evaluation relations is similar to $t_1 \Downarrow v_1$, which reads "t_1 evaluates to the value v_1". There is a tendency to use a top-down arrow, like $\Downarrow$, but like with the small-step syntax, this may vary.

Like was done previously, here's the big-step evaluation relation for the language *exprU*, again as provided by Pierce (2002):

$$\frac{}{v \Downarrow v} \text{ (B-Value)} \qquad \frac{t_1 \Downarrow zero}{\text{pred } t_1 \Downarrow zero} \text{ (B-PredZero)}$$

$$\frac{t_1 \Downarrow true \qquad t_2 \Downarrow v_2}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 \Downarrow v_2} \text{ (B-IfTrue)} \qquad \frac{t_1 \Downarrow \text{succ } nv_1}{\text{pred } t_1 \Downarrow nv_1} \text{ (B-PredSucc)}$$

$$\frac{t_1 \Downarrow false \qquad t_3 \Downarrow v_3}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 \Downarrow v_3} \text{ (B-IfFalse)} \qquad \frac{t_1 \Downarrow zero}{\text{iszero } t_1 \Downarrow true} \text{ (B-IsZeroZero)}$$

$$\frac{t_1 \Downarrow nv_1}{\text{succ } t_1 \Downarrow \text{succ } nv_1} \text{ (B-Succ)} \qquad \frac{t_1 \Downarrow \text{succ } nv_1}{\text{iszero } t_1 \Downarrow false} \text{ (B-IsZeroSucc)}$$

When comparing the two styles of semantics, it's possible to point out some differences. For instance, in big-step semantics there is now no more distinction between a computation and a congruence rule: every rule except B-Value has a premise, evaluating the appropriate subterms into values that are then used to evaluate the conclusion. A good example to visualize this relationship may be the rule B-IsZeroZero, which says that after t_1 has finished evaluating

and has as its result the value $zero$, the term (iszero t_1) located in the conclusion is going to be evaluated to $true$.

It is possible to see the $\Downarrow$ as a delimiter between an input channel and an output channel of the evaluation. Initially the right-hand side at every level is empty, and it remains empty as the computation goes up the evaluation tree until the rule B-Value is reached, which takes the term on the left-hand side and copies it to the right-hand side **iff** it is a value according to the grammar of the language. After that is done, that branch of the evaluation tree can start moving down the tree to perform the appropriate calculations using the values obtained at every level (FELLEISEN; FINDLER; FLATT, 2009).

To give a simple example, consider the term (succ (pred $zero$)). This is the step-by-step generation of its evaluation tree in big-step, with all the missing values filled with $\square$, the symbol this work is going to use for a value that is still unknown at a given moment of the execution. Read from top to bottom first and only then from left to right:

$$\text{Start}\quad \text{succ (pred } zero) \Downarrow \square$$

$$\text{B-Value}\ \frac{\dfrac{\dfrac{}{zero \Downarrow zero}}{\text{pred } zero \Downarrow \square}}{\text{succ (pred } zero) \Downarrow \square}$$

$$\frac{\text{Evaluating (pred } zero)\quad \text{pred } zero \Downarrow \square}{\text{succ (pred } zero) \Downarrow \square}$$

$$\text{B-PredZero}\ \frac{\dfrac{\dfrac{}{zero \Downarrow zero}}{\text{pred } zero \Downarrow zero}}{\text{succ (pred } zero) \Downarrow \square}$$

$$\frac{\dfrac{\text{Evaluating } zero\quad zero \Downarrow \square}{\text{pred } zero \Downarrow \square}}{\text{succ (pred } zero) \Downarrow \square}$$

$$\text{B-Succ}\ \frac{\dfrac{\dfrac{}{zero \Downarrow zero}}{\text{pred } zero \Downarrow zero}}{\text{succ (pred } zero) \Downarrow \text{succ } zero}$$

After the B-Succ step, the final result is (succ (pred $zero$) $\Downarrow$ succ $zero$). Do notice that, as opposed to small-step, big-step semantics don't fully commit to a specific rule at the time a recursion step is added to the tree, because it needs to wait for the premises to be fully evaluated in order to decide on which rule to use. For example, the expression (pred $zero$) at first isn't processed by either B-PredZero or B-PredSucc, because it needs to fully evaluate the sub-term $zero$ first to see if it results in the value $zero$ or a value of the form (succ t). Once the B-Value rule is used on this sub-term $zero$, however, it's possible to match the new premise ($zero \Downarrow zero$) with B-PredZero and evaluate (pred $zero$) to the value $zero$.

The terms of *exprU* may get stuck during the execution when following these sets of rules, resulting in a normal form that is not a value. In this language, this will happen anytime a computation requiring either $true$ or $false$, like the guard in the conditionals, receives a term of the form $zero$ or (succ nv), and also anytime a function meant to process natural numbers, like iszero, receives a boolean value.

In operational semantics, a term being *stuck* can be understood as the abstract machine representing the relation reaching a *meaningless state*, one that is not final, but that has no suitable transitions to take a step further (PIERCE, 2002).

One last example will show this happening in both small-step and big-step. Consider the term (iszero (iszero $zero$)). In small-step, the first step happens through E-IsZero followed by E-IsZeroZero, which results in the term (iszero $true$). This term is not a value, but it is in normal form, as there is no sequence of rules that can be applied to it that will result in a reduction. Therefore, the term is *stuck*.

In big-step, it may be more useful to reason through a sequence of prooftrees, like this:

$$\text{Start}\quad \text{iszero (iszero } zero\text{)} \Downarrow \square$$

$$\text{B-Value}\ \frac{\dfrac{\dfrac{}{zero \Downarrow zero}}{\text{iszero } zero \Downarrow \square}}{\text{iszero (iszero } zero\text{)} \Downarrow \square}$$

$$\frac{\text{Evaluating (iszero } zero\text{)}\quad \text{iszero } zero \Downarrow \square}{\text{iszero (iszero } zero\text{)} \Downarrow \square}$$

$$\frac{\dfrac{\text{Evaluating } zero\quad zero \Downarrow \square}{\text{iszero } zero \Downarrow \square}}{\text{iszero (iszero } zero\text{)} \Downarrow \square}$$

$$\text{B-IsZeroZero}\ \frac{\dfrac{\dfrac{}{zero \Downarrow zero}}{\text{iszero } zero \Downarrow true}}{\text{iszero (iszero } zero\text{)} \Downarrow \square}$$

The tree ends with the value of the root still filled by $\square$, as there is no value that can result from the term (iszero t_1) when ($t_1 \Downarrow true$). This shows the original term (iszero (iszero $zero$)) fails to evaluate to any value.

A possible solution to this problem is the implementation of a typing mechanism, which is discussed later on.

2.1.2.4 Evaluation Contexts

Another way of viewing congruence rules is through *evaluation contexts*. A term can be rewritten as a context by replacing a fragment of its recursive structure with a *hole*, written here as [—]. The location of the hole indicates where the next reduction is going to happen through a computation rule (PIERCE, 2002).

Where the hole can be placed and the order of operations that must be followed depends on how the context is defined in the language's grammar. As an example, here's a suitable context for *exprU*'s:

$\langle C \rangle$ ::= [—]
| if $\langle C \rangle$ then $\langle t \rangle$ else $\langle t \rangle$
| succ $\langle C \rangle$

| pred $\langle C \rangle$
| iszero $\langle C \rangle$

Remarkably, the rule for the *if* only allows for contexts to continue their recursion through the conditional guard, and never inside the branches. As a consequence, the hole can only be placed inside the guard as well, which forces reductions on sub-terms to only happen there. This is done like this to correctly represent the E-If congruence rule, and in fact, to define an evaluation context is to essentially encode a language's congruence rules directly into the grammar of the language.

We write $C[t]$ to mean the action of replacing (or "filling") the hole in a context C with the term t (PIERCE, 2002). For example, if $C \equiv$ (if [—] then (succ zero) else zero), then $C[(\text{iszero } zero)]$ results in the term (if (iszero $zero$) then (succ zero) else zero).

These contexts can then be used to reason about reductions. The statement $C[\text{iszero } zero] \rightarrow C[true]$ means that the reduction (iszero $zero \rightarrow zero$) happens in the hole of a context C.

2.1.3 λ-calculus

λ-calculus is a formal system of computation formulated by Alonzo Church (CHURCH, 1936). There are multiple variants, such as the untyped or simply typed, but this section will only deal with the untyped version. In these systems, quoting Pierce (2002), "all computation is reduced to the basic operations of function definition and application". This means that λ-calculus performs computation steps by chaining together function calls. λ-calculus terms can be generated by following this CFG:

$\langle t \rangle$ ::= x
| $\lambda x.\langle t \rangle$
| $\langle t \rangle \langle t \rangle$

In the grammar above, $\langle t \rangle$ is a λ-calculus term, and ignoring the angle brackets from now on, x is any *variable* or *atomic constant*, $\lambda x.t$ is a λ-*abstraction*, and $t\ t$ is an *application*. Abstractions can be understood as functions: they take in one parameter as input, which is the variable present between λ and the dot (the left-hand side of the abstraction), and what comes after that is the term in which it might be possible to replace said input into (the right-hand side), given an application (which is like a function call) (HINDLEY; SELDIN, 2008).

2.1.3.1 Variable binding and α-equivalence

Consider the identity function: it takes one input, and returns the exact same input back, without any modification. This function can be created in λ-calculus with a λ-abstraction such as $(\lambda x.x)$. When called with an input, this function simply replaces the x in the right-hand side with the input provided.

However, now consider a constant function: it also takes one input, but it discards it and returns a constant value. A corresponding λ-abstraction can be $(\lambda x.y)$. When called, it replaces any x on the right-hand side with the input, but as there is no x on the right, only an y, the result of an application on this abstraction is always y for all x provided.

This happens because what the left-hand side in the abstraction is doing is the process of *binding* the variable given between the λ and the dot, in this case x, to any of its occurrences after the dot. We say that the variable x in the right-hand side of the identity function is *bound*, but the variable y in the constant function is *free*, because it is not bound by any abstractions in its scope (HINDLEY; SELDIN, 2008). Free variables are always constants, because they will never change based on what the input is.

Sometimes it's useful to get the set of free variables in a term, and for any term t_1 this set is returned by the function call $FV(t_1)$, defined in λ-calculus as follows (PIERCE, 2002):

$$\begin{aligned} FV(x) &\equiv \{x\} \\ FV(\lambda x.t_1) &\equiv FV(t_1) \setminus \{x\} \\ FV(t_1 t_2) &\equiv FV(t_1) \cup FV(t_2) \end{aligned}$$

When a variable is bound, any parameter provided to its *binding abstraction* is going to change the variable's representation inside the abstraction into the parameter provided. For example, the application $(\lambda x.xx)\ y$ will result in yy, because all occurrences of x in the right-hand side are going to be replaced by the parameter provided, in this case y (HINDLEY; SELDIN, 2008).

Still about binding, it is possible to chain abstractions one inside another, and the bound variable from the outer abstraction are still bound within its inner abstractions (HINDLEY; SELDIN, 2008). For example, consider this function: $(\lambda x.(\lambda y.xy))$. The outer abstraction is $\lambda x.t_1$, and the inner is $\lambda y.xy$, where t_1 is the inner abstraction. In isolation, the inner abstraction would only have y as a bound variable, and x would be free; however, because x is already bound by the outer abstraction and the inner one exists within it, x remains bound. This is because binding happens within the entire body (right-hand side) of the abstraction, which becomes the *scope* of the binding (HINDLEY; SELDIN, 2008).

This behavior of λ-abstractions allow the calculus to represent multi-argument functions by nesting λ-abstractions together, even though each abstraction can only accept one parameter. This was showcased in the example above, when two abstractions were used, each being representative of a single-argument function, in order to define an operation that relies on 2 parameters. This process of representing polyadic (multiple arguments) functions as a sequence of monoadic (single argument) functions, each returning another monoadic function except for the last one, which actually returns the value being computed, is called *currying*. As every λ-abstraction has exactly one argument, all of them are in *curried form* (PIERCE, 2002).

One final note on binding is that although an abstraction binding has influence over its entire inner scope, pure λ-calculus results in the possibility for *shadowing*, which is the rebinding of an already bound variable by an inner abstraction. Using the concatenation example above, if instead of $(\lambda x.(\lambda y.xy))$, which at the end has both x and y bound by the outer and the inner scope, respectively, we had $(\lambda x.(\lambda y.(\lambda x.xy)))$, two things would change: first, this abstraction would represent a 3-parameters function instead of a 2-parameters function; second, the first parameter, which appears bound to x, would actually be discarded at the end, because x has been rebound in the third abstraction by shadowing the first binding, and there is no usage of the first x that exists outside the scope of the third abstraction but inside the scope of the first one. As a result, in order to represent ab, instead of supplying a first and b second, like one would do in the previous abstraction, one would need to supply, in order, Tba, where T is any value (PIERCE, 2002).

Lastly, it's useful to define α-*equivalence*: Two λ-calculus terms are said to be α-equivalent if they represent the same function, which is to say that given the same input, both terms will always return the same output. The name of bound variables does not matter for α-equivalence, so both $(\lambda x.x)$ and $(\lambda y.y)$, for example, represent the same function, as they only differ in the name of the chosen parameters (PIERCE, 2002). Lastly, the process of renaming a binding variable in a λ-abstraction in a way that preserves α-equivalence is called α-*conversion*.

2.1.3.2 Function application and β-reduction

An application of the form $(\lambda x.t_1)\ t_2$ is a *reducible expression*, a name which is often seen in its syllabic abbreviation form *redex* (PIERCE, 2002). As the name suggests, redexes are still reducible, meaning it's possible to perform at least one more step of computation on them, if desired, by applying the term t_2 onto the abstraction in the left-hand side (PIERCE, 2002). This computation step in λ-calculus is known as β-reduction, which is the replacement of all free occurrences of the bound variable x in t_1 by the term t_2, generating a new term as a result (HINDLEY; SELDIN, 2008). Redexes can be seen as function calls, where β-reduction is the process of executing the function on the left with the parameter on the right. When there are no more redexes left in a term, this term is in β-normal form, which is to say it is in normal form with respects to β-reduction (HINDLEY; SELDIN, 2008).

β-reduction is necessarily done in a *capture-avoiding* manner, meaning all free variables in t_2 will remain being free, even if a naive substitution into t_1 would bind them. For example, a capturing (and therefore incorrect) substitution done in $(\lambda x.(\lambda z.x))\ z$ would result in $(\lambda z.z)$ (PIERCE, 2002). This is incorrect because the binder z, located in the inner abstraction, is semantically different from the free variable z being used as an argument to the function, but because they have the same name, replacing x with the free variable z binds this initially free variable to the inner abstraction, which should not happen.

To make this clearer, think about what would happen if we reduced the outer abstraction

using a parameter different than z, and w.l.o.g let's say this parameter is a: in that case, the outer redex would reduce to $(\lambda z.a)$, which is a constant function that returns the value originally supplied to the outer abstraction. This should be the behavior for all applications, because the name of a bound variable should not matter to the abstraction's image. However, if z is supplied, then instead of a constant function, the identity function is returned. This is because the free variable x (free with regards to the inner abstraction, that is), when substituted into the supplied free variable z, is actually captured by the binder z and becomes a bound variable. This phenomenon is called *variable capture* (PIERCE, 2002).

In order to fix this, β-reduction requires the binding z from the original term to not be a free variable in the term x is being replaced with, in this case only a free z. This solves the issue, but if nothing more is done, it also turns the substitution function partial. If the binder **is** a free variable in the right-hand term in the application, like in the given example, then instead of doing an incorrect substitution, like before, nothing would happen (PIERCE, 2002).

To make it total again, this case gets an α-conversion step, renaming the binding variable causing this issue in a way that preserves α-equivalence. Given an abstraction $(\lambda x.(\lambda y.P))N$, if $y \in FV(N)$ then α-conversion will replace y with any new variable k such that $k \notin FV(NP)$ (HINDLEY; SELDIN, 2008). Doing this allows the originally problematic application to result in $\lambda k.z$, which is an α-equivalent constant function, just as desired.

There are multiple possible syntaxes for capture-avoiding substitutions, but in this work the one that is going to be used is $[x \mapsto t_2]\ t_1$, where x is the binding variable, t_1 is the body of the abstraction the substitution is going to act on, and t_2 is the parameter provided and that will replace x in t_1 (PIERCE, 2002).

Using that syntax, β-reduction can be defined as follows (HINDLEY; SELDIN, 2008) (CURRY; FEYS; CRAIG, 1974):

$$
\begin{aligned}
[x \mapsto N]x &\equiv N \\
[x \mapsto N]a &\equiv a, \quad \text{if } a \neq x \\
[x \mapsto N](\lambda x.P) &\equiv (\lambda x.P) \qquad \text{– shadowing} \\
[x \mapsto N](\lambda y.P) &\equiv \lambda y.[x \mapsto N]P, \\
&\qquad \text{if } x \neq y \wedge y \notin FV(N) \\
[x \mapsto N](\lambda y.P) &\equiv \lambda z.[x \mapsto N]([y \mapsto z]P), \\
&\qquad \text{if } x \neq y \wedge y \in FV(N), \\
&\qquad \text{given a } \ z \ \text{ s.t. } \ z \notin FV(NP) \\
[x \mapsto N](PQ) &\equiv (([x \mapsto N]P)\ ([x \mapsto N]Q))
\end{aligned}
$$

Then, for a redex like $(\lambda x.A)\ B$, the application using β-reduction is called as $[x \mapsto B]A$.

2.1.3.3 Evaluation strategies

Given a λ-term that is not in β-normal form, there are distinct ways of doing computation on the remaining redexes, and these make up the different *evaluation strategies* of λ-calculus. Depending on the strategy, not every redex is going to necessarily be executed, and regarding the ones that are, their order of execution may also differ. It is beyond the scope of this work to discuss all the strategies available, but two of them should be briefly mentioned: *call by name* and *call by value*.

- **Call by Name** (CBN) reduces the leftmost, outermost redex first, and does not allow any reductions inside abstractions, even if they are possible. If a function call is done using this method of evaluation, its argument will be sent in the format it is found in, without trying to reduce it to a value first. This is significant if the parameter applied is also a redex, as the entire redex will be sent and will only be reduced if and when it is required to be, potentially avoiding computation that might be unnecessary or would result in a stuck term, but at the cost of potentially having to evaluate the same expression multiple times instead of only once. The evaluation process done by this strategy is known as *lazy evaluation* (PIERCE, 2002). Below is an example of a CBN evaluation, adapted from Pierce (2002):

$$\begin{aligned}(\lambda x.x)((\lambda y.y)(\lambda z.(\lambda k.k)z)) &\rightarrow \\ (\lambda y.y)(\lambda z.(\lambda k.k)z) &\rightarrow \\ (\lambda z.(\lambda k.k)z)\end{aligned}$$

- **Call by Value** (CBV) also prioritizes reducing the leftmost, outermost redex first, and it also does not allow for available reductions inside abstractions, but on top of that it also requires the parameter in the application to be reduced to a value first. If the same function call is done with this method, its argument will first be evaluated into a value, and only then will the function be executed. Opposite to the call by name strategy, this means that parameter redexes will be executed regardless of whether they will be used or not, and they will be executed at function call time. This might result in some unnecessary work in the case the parameter is not actually used, but this work is only going to be done once, as it's not at risk of being repeated in redexes such as $(\lambda x.xx)(fy)$, where (fy) is another redex. The evaluation process that uses this strategy is known as *eager evaluation* (PIERCE, 2002). Below is the previous example, but performing CBV evaluation:

$$\begin{aligned}(\lambda x.x)((\lambda y.y)(\lambda z.(\lambda k.k)z)) &\rightarrow \\ (\lambda x.x)(\lambda z.(\lambda k.k)z) &\rightarrow \\ (\lambda z.(\lambda k.k)z)\end{aligned}$$

Different evaluation strategies might result in different normal-forms, but semantically the results that terminate should be the same (PIERCE, 2002). For example, a strategy that does allow

reductions inside abstractions could reduce the starting abstraction in the example to $(\lambda z.z)$, yielding a different normal form. However, as soon as another application is done on either of the CBN or CBV normal forms above, z is getting substituted inside the abstraction, allowing for the $(\lambda k.k)\ a$ redex to be evaluated, with a being the term z is substituted with. In the end, one more application to any of these terms will result in the same final value.

2.1.3.4 Lambda calculus and Turing machines

λ-calculus is an specially useful abstraction in the context of functional programming, given its essence of representing computation as chained mathematical functions, forgoing the necessity of holding a state. However, it is significant to note that λ-calculus is also *Turing complete* (PIERCE, 2002), meaning that it's expressive enough so that it can be used to simulate any Turing machine, and therefore be used to execute any computation a Turing machine is able to perform.

2.1.4 Language safety and Typing

However useful λ-calculus and Turing machines may be, they are still only mathematical formalisms, so they are not practical to be used directly by developers. Instead, they serve as part of the theoretical foundations used to design, build, and study more practical tools to perform computation, such as high level programming languages (PIERCE, 2002).

When designing a programming language, it is possible to use the mathematical formalisms already found and studied as a backbone, and focus on making design choices that can, among other things, implement useful semantic abstractions that are relevant to the programmer, such as a typing system or well-defined operations on arrays. These abstractions are what makes a language *high-level* to begin with. However, if the implementation of the language does not protect its abstractions, then the language is *unsafe* (PIERCE, 2002). Using an example given by Pierce (2002), a programmer expects an array to be an abstraction of the computer memory that can only have its values altered by operations on that array; but an unsafe language may end up allowing, whether as a feature, a bug, or an unwanted consequence of a critical aspect of the language, the values of that array to be changed by writing past the ending of another data structure. This is the case in some C implementations, due to some *undefined behaviors* regarding pointers.

If the language is safe, the programmer can use the abstractions and not have to worry about lower-level details. But if the language is **not** safe, then the programmer does need to know about things that go beyond the abstractions of the language, such as memory layout or the implementations details of the language (PIERCE, 2002).

2.1.4.1 Static type systems

An option to increase the safety of a language is a *static type system*. Below is a definition for type system, adapted from Pierce (2002):

Definition 2.1.1. *A type system is an automatic method to prove the absence of certain program behaviors by classifying its terms based on the kinds of values they compute.*

A point that deserves attention in that definition is how type systems can only prove the absence of certain program behaviors, but **not** their presence. This means type systems are *conservative*, and will therefore also reject some programs that behave well at run time (PIERCE, 2002).

Consider the usage of this ternary operator in pseudo-code: (**int** x $=$ (5,10) if $False$ else 2) (example inspired by Pierce (2002)). Some compilers could perform an optimization step and change this code in a way that removes the ternary operator and leaves only the number 2 as a constant to be assigned, and in that case, no issues arise. However, assuming there is no compile-time optimizations and that this language is statically typed, then this command would likely result in an error. This happens because, despite always actually returning an **int**, the actual type of the expression in the right-hand side cannot be inferred by the *type-checker* at compile-time to an **int** without actually executing the code. Instead, this expression either cannot be typed at all or has a type similar to **Either((int,int), int)**, which differs from the type **int** of the left-hand side of the assignment (PIERCE, 2002).

Static typing is very useful to catch a variety of errors at compile-time, and that can aid software development at any stage, even if the programmers don't go out of their way to make use of the more robust features in a language's type system. However, spending some time to make the best use possible out of the typing mechanisms, specially if they're expressive and extendable enough to allow for problem-specific abstractions to be encoded as types in a program, tends to result in code that is much safer, as it helps the type-checker in finding a larger range of possible issues. Static typing alone cannot guarantee language safety, however, because there are issues it cannot catch at compilation time, such as division-by-zero and out-of-bounds array accesses, so these are usually handled dynamically at run-time (PIERCE, 2002).

2.1.4.2 Typed terms

To start, return to the grammar definition of the language *exprU* at 2.1.2.3, and then consider the following simply-typed variant of this language and grammar, also provided by Pierce (2002):

$\langle t \rangle$ ::= true
| false
| if $\langle t \rangle$ then $\langle t \rangle$ else $\langle t \rangle$

| zero
| succ $\langle t \rangle$
| pred $\langle t \rangle$
| iszero $\langle t \rangle$

$\langle v \rangle$ ::= true
| false
| $\langle nv \rangle$

$\langle nv \rangle$ ::= zero
| succ $\langle nv \rangle$

$\langle T \rangle$ ::= Bool
| Nat

This new language is going to be named *exprT*, with the T standing for Typed. The only change in grammar is the inclusion of a new non-terminal $\langle T \rangle$, which can be either Bool or Nat and will represent the possible types in the language.

Working on this new non-terminal, a typing relation _ : _ is defined, such that $t_1 : T_1$ means "t_1 is of type T_1". This typing relation is more naturally represented as big-step, because there's no clear way to separate steps from one another (as there was in the case of reduction relations: a reduced sub-term). This relation has a set of inference rules that assign types to terms (PIERCE, 2002):

$$\frac{}{\text{true} : \text{Bool}}\ \text{(T-True)} \qquad \frac{t_1 : \text{Nat}}{\text{succ}\ t_1 : \text{Nat}}\ \text{(T-Succ)}$$

$$\frac{}{\text{false} : \text{Bool}}\ \text{(T-False)}$$

$$\frac{t_1 : \text{Nat}}{\text{pred}\ t_1 : \text{Nat}}\ \text{(T-Pred)}$$

$$\frac{}{\text{zero} : \text{Nat}}\ \text{(T-Zero)}$$

$$\frac{t_1 : \text{Bool} \qquad t_2 : T_1 \qquad t_3 : T_1}{\text{if}\ t_1\ \text{then}\ t_2\ \text{else}\ t_3 : T_1}\ \text{(T-If)} \qquad \frac{t_1 : \text{Nat}}{\text{iszero}\ t_1 : \text{Bool}}\ \text{(T-IsZero)}$$

This relation checks whether a term is *well-typed* or *ill-typed*. More specifically, if there is a T_1 such that $t_1 : T_1$ is derivable, then t_1 is *typable* (PIERCE, 2002). Also, in this language specifically, as proved by Pierce (2002), every typable term has exactly one possible type, because there's only one typing derivation that satisfies the relation.

This typing relation is able to check typability to every term without needing to evaluate it. This is because this relation acts only on the term's syntax, and not on its semantics. Due to this, it's possible to do it statically, at compile-time.

If this typechecker was in a compiler for *exprT*, an ill-typed term could be set to not compile at all. This would make sure only well-typed terms are allowed to even begin executing, reducing the domain of the reduction/evaluation relations to terms that are known for sure to not cause a run-time type error. This increases language-safety and avoids stuck terms (PIERCE, 2002). In the absence of a compiler to concisely separate compilation from execution, however, consider that there is a type-checker step before evaluation, and that only well-typed terms are part of the domain of the reduction/evaluation relations.

Earlier, a provided example caused an issue in the untyped version of this language. That term was (iszero (iszero $zero$)), and when attempting to evaluate it, neither the closure of the small-step relation nor the big-step relation were able to reduce it to a value in the language.

Consider the same term, but now with *exprT* as its language. Before attempting to evaluate it, the term goes through a type-checking step...

$$\text{Start}\quad \text{iszero (iszero } zero\text{)} : \square$$

$$\text{T-Zero}\ \frac{\dfrac{\dfrac{}{zero : \text{Nat}}}{\text{iszero } zero : \square}}{\text{iszero (iszero } zero\text{)} : \square}$$

$$\frac{\text{Typechecks (iszero } zero\text{)}\quad \text{iszero } zero : \square}{\text{iszero (iszero } zero\text{)} : \square}$$

$$\frac{\text{Typechecks } zero\quad \dfrac{zero : \square}{\text{iszero } zero : \square}}{\text{iszero (iszero } zero\text{)} : \square}$$

$$\text{T-IsZero}\ \frac{\dfrac{\dfrac{}{zero : \text{Nat}}}{\text{iszero } zero : \text{Bool}}}{\text{iszero (iszero } zero\text{)} : \square}$$

...and the type-checker fails to find a suitable type. This happens because the only typing rule for (iszero t_1) requires the type of t_1 to be Nat, but in the provided term, t_1 has type Bool. Therefore, the term (iszero (iszero $zero$)) is untypable.

As it's untypable, this term falls outside of the relations' newly established domains, and won't be evaluated. This allows these kinds of issues to be caught early, due to the type-checker's static and conservative nature (PIERCE, 2002).

Lastly, typing relations can either perform *checks* or *inferences*. Type checking gets both the term and the type as an input and then verifies if the provided type is suitable for that term. Type inference, on the other hand, only gets a term, and then tries to find a possible suitable type for it.

2.1.5 Unification

The unification problem, in its most general form, is the question of whether a finite set of equations is contradictory or not. If it is not, it may also be a part of the problem to return a

set of substitutions that, when applied to the variables on both sides of any equation from the set, result in them obeying some definition of equivalence.

There are various types of unification problems, but the one relevant to this work is syntactic unification. In this variant, both sides of the equations are made out of terms that should be syntactically equivalent to one another after a solution set of variable substitutions are applied to each of them. This set is called an *unifier* of the equations.

The difficulty behind this problem lies on the fact that the variables present in the equations can have restrictions regarding what values they're allowed to have, and these restrictions are unknown at first. As a brief first example, consider this set containing just one equation syntactically relating functions, using lowercase Latin letters for constant symbols and uppercase Latin letters for non-nullable variables:

$$\{f(X, h()) = f(g(h()), X)\}$$

From this equation, we can match the pattern to see two equalities: $X = g(h())$ and $h() = X$. It follows then that for this to have an unifier, it must be true that $h() = g(h())$. However, these are clearly different terms, and therefore this example does **not** have an unifier. Now, as a second example, consider the following set of equations between functions:

$$\{f(g(X), g(Y)) = f(Y, g(g(Z))),\ g(Z) = g(h())\}$$

There are three variables in the example, X, Y and Z, and the first equation already gives information as to what values some of these may be allowed to have: $g(X) = Y$ and $g(Y) = g(g(Z))$. Some simplifications are already possible, because if $g(X) = Y$, then $g(Y) = g(g(Z)) \implies g(g(X)) = g(g(Z))$. Following from the simplification, it's also possible to see that $X = Z$. So, by just looking at the first equation, it's possible to say that if there's a valid unifier for this set of equations, it must be at least as specific as $\{X \mapsto Z,\ Y \mapsto g(Z)\}$. The second equation says that $g(Z) = g(h())$, so $Z = h()$. This further specializes the variables, resulting in the following final unifier: $\{X \mapsto h(),\ Y \mapsto g(h()),\ Z \mapsto h()\}$.

All variables have been mandatorily assigned to a constant, so this is an unique unifier. However, if this example had stopped at the first equation, then it would have an infinite number of possible unifiers, because any set of substitutions that follows the patterns provided by and that are at least as specific as $\{X \mapsto Z,\ Y \mapsto g(Z)\}$ would validate the equations. In other words, if the example had stopped at the first equation, then $\forall Z.\{X \mapsto Z,\ Y \mapsto g(Z)\}$ would form a pattern for all valid unifiers.

Actually, it's possible to keep on specializing that pattern while still avoiding getting to constants. For example, $\forall Z'.\{X \mapsto g(Z'),\ Y \mapsto g(g(Z')),\ Z \mapsto g(Z')\}$ would also be a pattern for valid unifiers. However, this one is more specific than the prior, which means there is a set of substitutions γ that can be applied together with $\{X \mapsto Z,\ Y \mapsto g(Z)\}$ such that these substitutions become equivalent to the set $\{X \mapsto g(Z'),\ Y \mapsto g(g(Z')),\ Z \mapsto g(Z')\}$ (PIERCE, 2002). In this case, γ is $\{Z \mapsto g(Z')\}$.

If it's not possible to recreate a valid unifier σ by applying together a substitution γ with another valid unifier σ', then σ is the *principal unifier*, also known as the *most general unifier* (PIERCE, 2002).

Several algorithms have been proposed to solve the unification problem. One of them was defined in Martelli e Montanari (1982), which is able to return, in linear time, either a failure or a set of equations in solved form (defined below) equivalent to the set of equations provided as input. It's also proved that it always terminates.

A set of equations is in solved form iff all equations are of the form $X = \alpha$, where X is a variable and α is a term, and every X on the left side is unique, meaning it never appears again, neither on the left nor on the right side of any equation in the set (MARTELLI; MONTANARI, 1982).

Given a set of equations in solved form, its most general unifier is just the set, but with the equality relation being replaced with a substitution relation (MARTELLI; MONTANARI, 1982). For example, the unifier of $\{X = g(Z),\ Y = h()\}$ is simply $\{X \mapsto g(Z),\ Y \mapsto h()\}$.

As the algorithm returns a set of equations in solved form **equivalent** to the original set of equations, the easily obtained trivial unifier is also the most general unifier for the original set (MARTELLI; MONTANARI, 1982).

With this background done, the algorithm is now going to be defined, using lowercase Latin letters for literal symbols, uppercase Latin letters for variables, lowercase Greek letters for any term (variable or not, and containing a variable or not), and the uppercase Greek letter Π for a set of equations. It works by applying transformations to the equations in Π one at a time until none more are available, in which case it successfully returns the transformed set Π, now in solved form, or until one of the failure rules are reached:

$$\Pi \cup \{\alpha = X\} \equiv \Pi \cup \{X = \alpha\}, \quad \text{if } \alpha \text{ is not a variable} \tag{1}$$

$$\Pi \cup \{X = X\} \equiv \Pi \tag{2}$$

$$\Pi \cup \{f(\alpha_1, \ldots, \alpha_n) = g(\beta_1, \ldots, \beta_m)\} \equiv \bot, \quad \text{if } ((f \neq g) \vee (n \neq m)) \tag{3}$$

$$\Pi \cup \{f(\alpha_1, \ldots, \alpha_n) = f(\beta_1, \ldots, \beta_n)\} \equiv \Pi \cup \{\alpha_1 = \beta_1, \ldots, \alpha_n = \beta_n\} \tag{4}$$

$$\Pi \cup \{X = \alpha\} \equiv \bot, \quad \text{if } (X \in \text{Vars}(\alpha)) \tag{5}$$

$$\Pi \cup \{X = \alpha\} \equiv ([X \mapsto \alpha]\ \Pi) \cup \{X = \alpha\}, \quad \text{if } (X \in \text{Vars}(\Pi)) \tag{6}$$

For these rules, consider that the current set of equations is Π' and that $\Pi' \equiv \Pi \cup E$, where E is an equation with a suitable transformation randomly sampled from Π' at each step. Also, $[X \mapsto \alpha]\ \Pi$ indicates a substitution of every instance of X in Π with the term α. Finally, the function Vars returns a set of every variable in a provided term or set of equations.

Rule (1) merely applies the symmetric property of equality in order to rewrite equalities relating variables and non-variables such that the variable is on the left-hand side and the non-

variable is on the right-hand side.

Rule (2) erases identical variables. Do note that variable names matter here, as this rule would **not** be applied if the sampled equation was $X = Y$.

Rule (3) is the first failure condition, which happens when there is a mismatch between two terms' structure. This mismatch is manifested either through the application of different root functions ($f \neq g$) or through the supply of a different number of arguments ($n \neq m$).

On the other hand, if both $f = g$ and $n = m$, then rule (4) can be used, applying what Martelli e Montanari (1982) calls *term reduction*. The term reduction splits every corresponding pair of arguments into a new equation and erases the original top-level equation. Despite the syntax in the rule, it's possible to have $n = 0$, and in such case just the original equation is erased without any new ones being added.

Rule (5) is another failure condition, and it's triggered when a left-hand side variable X occurs within the same equation's right-hand side. This means X is an infinitely recursive term, which is not allowed here. Note that preceding rules have precedence over subsequent ones, and as such if $\alpha \equiv X$, then rule (2) would be taken, instead of resulting in a failure.

Rule (6) applies *variable elimination* (MARTELLI; MONTANARI, 1982), replacing every instance of X in the remaining set of equations Π with the term α. Note that $X \notin \text{Vars}(\alpha)$, as that would result in rule (5) being taken, and as such $X \notin \text{Vars}(\Pi)$ after this transformation is applied. The sampled equation is not subjected to the substitution, but it remains in the set, so that subsequent variable eliminations can take place in its right-hand side. Finally, the condition that $X \in \text{Vars}(\Pi)$ is there to guarantee termination.

To showcase this algorithm being applied, consider again the second example:

$$\{f(g(X), g(Y)) = f(Y, g(g(Z))),\ g(Z) = g(h())\}$$

Let's say that the second equation is selected first: in this case, rule (4) is taken, and the new set is $\{f(g(X), g(Y)) = f(Y, g(g(Z))),\ Z = h()\}$

Now let's say the first equation from the new set is selected: rule (4) is taken again, and the new set is $\{g(X) = Y,\ g(Y) = g(g(Z)),\ Z = h()\}$.

First equation leads to rule (1): $\{Y = g(X),\ g(Y) = g(g(Z)),\ Z = h()\}$.

Third equation leads to rule (6): $\{Y = g(X),\ g(Y) = g(g(h())),\ Z = h()\}$.

First equation to rule (6) again: $\{Y = g(X),\ g(g(X)) = g(g(h())),\ Z = h()\}$.

Second equation leads to rule (4): $\{Y = g(X),\ g(X) = g(h()),\ Z = h()\}$.

Second equation again, and again leading to rule (4): $\{Y = g(X),\ X = h(),\ Z = h()\}$.

Second equation once more, but now to rule (6): $\{Y = g(h()),\ X = h(),\ Z = h()\}$.

And now there are no more possible transformations that can be applied, and as no failure

conditions have been reached, $\{Y \mapsto g(h()),\ X \mapsto h(),\ Z \mapsto h()\}$ is the returned unifier.

2.1.6 Compiler's Intermediate Representation languages

It's beyond the scope of this work to talk about compilers as a whole, but one part of the field deserves mention: Intermediate Representations (IR).

An IR is a compiled code that is in a language other than the compiler's final target, but is procedurally generated by the compiler nonetheless as an intermediate step. This can aid in the optimization process, as one can design the IR language to be much lower-level than the source code's high-level programming language. This gives much more flexibility to optimizations, as now there is no more need to conform to the source language's abstractions (ALFRED; MONICA; JEFFREY, 2007). The process of reducing the level of a language during the compilation process is called *lowering*.

2.1.7 Racket and PLT Redex

PLT Redex is a domain-specific language (DSL) for the Racket programming language (FELLEISEN; FINDLER; FLATT, 2009). Racket is a Lisp dialect, and it offers powerful pattern matching capabilities and functional paradigm support, being often used for programming language research (KLEIN et al., 2012). As the DSL is contained within Racket, it also uses a Lisp-like syntax, which means everything is written using S-expressions, forgoing infix notation in favor of explicit parentheses-guided function calls. For example, to evaluate the expression $2+3*5$ in a Lisp-like syntax, one would write $(+\ 2\ (*\ 3\ 5))$. This avoids ambiguity by explicitly writing expressions like the programmer would want them to be derived by the language's grammar.

Before continuing with this section, it's of note that the name Redex is derived from "reducible expression", a concept already discussed in this work and that is often given the name "redex". To differentiate between the Racket DSL and the concept of reducible expressions, the former will always be capitalized and the latter will never be (except in cases where capitalization is grammatically mandatory).

Moving on, Redex was made with the intent of providing language semantics modeling functionalities. It allows the user to mechanically represent a language's grammar using the BNF syntax, program the relations the language executes on its terms, perform randomized tests on the language, etc.

In order to showcase Redex's functionalities, consider the previously defined *exprT* language. This language is going to be implemented using the DSL. To start, a racket file is created and Redex imported through **(require redex)**.

To establish the language's syntax, the **define-language** function is provided:

```
(define-language exprT
  (t ::= true
         false
         zero
         (if0 t t t)
         (succ t)
         (pred t)
         (iszero t))
  (v ::= true
         false
         nv)
  (nv ::= zero
          (succ nv))
  (T ::= Bool
         Nat))
```

This creates the language with terms t, v, nv and T, just like the language specification. The only difference is how the *if* name was changed to *if0*, to avoid a naming conflict with Racket's own conditional function.

This language does not offer an opportunity to showcase this, but when defining a language, it's possible to set *binding forms*. This will tell Redex what term is responsible for the binding (and therefore represents the scope) of another term. As a small example of this, consider a possible binding rule for λ-calculus: $(\lambda x.t$ #:refers-to $x)$. The #:refers-to clause binds the term on the left (in this case, t) to the one on the right (in this case, x). Essentially, this clause is stating that every instance of x inside t is not an independent variable, but the same one, which is the binding variable of the abstraction (FELLEISEN; FINDLER; FLATT, 2009).

In Racket, it's possible to bind the value of an expression to an id with the **define** function. This can be used together with Redex's **term** function, which manually creates a term. To test if a given term can match with a language's grammar, the **redex-match?** function can be used. All three of these functions are demonstrated below:

```
(define eg_if (term (if0 (iszero zero) zero (succ zero))))
(redex-match? exprT t eg_if)
```

With the language defined, it's possible to implement the big-step type-checker. Typing relations are usually codified as big-step semantics, and in Redex, big-step semantics are programmed as *judgments*. To create a judgment, it's possible to use the function **define-judgment-form**:

```
(define-judgment-form exprT
    #:mode (type-infer I O)
    #:contract (type-infer t T)

    [------------------------------ "T-True"
     (type-infer true Bool)]

    [------------------------------ "T-False"
     (type-infer false Bool)]

    [------------------------------ "T-Zero"
     (type-infer zero Nat)]
```

```
13
14      [(type-infer t_1 Bool)
15       (type-infer t_2 T_1)
16       (type-infer t_3 T_1)
17       ------------------------------ "T-If"
18       (type-infer (if0 t_1 t_2 t_3) T_1)]
19
20      [(type-infer t_1 Nat)
21       ------------------------------ "T-Succ"
22       (type-infer (succ t_1) Nat)]
23
24      [(type-infer t_1 Nat)
25       ------------------------------ "T-Pred"
26       (type-infer (pred t_1) Nat)]
27
28      [(type-infer t_1 Nat)
29       ------------------------------ "T-IsZero"
30       (type-infer (iszero t_1) Bool)])
```

Two important options: mode and contract. Mode indicates the inputs and outputs channels of the judgment according to parameter order. Contract is the non-terminal signature, so the first parameter is a t (a term of the language) and the second parameter is a T (a type of the language). Interpreting both options together, this judgment is going to receive a term as an input and determine a possible type as an output.

After that, there is a list of the language's typing rules, as defined in previous sections. This is done in a syntax that resembles natural deduction, so in T-If, for example, the first 3 lines are premises that need to be matched (t_1 needs to be of type Bool and t_2 and t_3 must have the same type), then a separator line with the name is placed, and lastly the conclusion, which assigns type T_1 to the *if*.

After defining the typing judgment, it's possible to randomly generate terms that satisfy the typing restrictions, test if a term has a given type, or define reduction relations as to only allow terms that satisfy this relation.

Before setting up the language's reduction relation, it's required to set up a *context*. Contexts can be thought of as the outer terms in which the congruence rules in the relation will act on. For example, consider the following congruence rule of *exprT*:

$$\frac{t_1 \rightarrow t_1'}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 \rightarrow \text{if } t_1' \text{ then } t_2 \text{ else } t_3} \text{ (E—If)}$$

This t_1 could be any other t term: it could be a pred, another if, or even another value[1]. But regardless of what t_1 is, the execution of E-If is the same. This creates a reduction context, where certain sub-terms need to be reduced first as a prerequisite to reduce the original term.

[1] Although because of the computation rules matching with $true$ and $false$, the only way t_1 could be a value is if the execution has reached a nonsensical state, like (if $zero\ zero\ zero$), precisely the kind of issue the typing in this language is made to avoid

In Redex, an evaluation context is created as an extension of the main language with an added non-terminal meant to represent the context. This extension can be defined by **define-extended-language**, and the new non-terminal must contain two pieces of information: a hole and where in the grammar it's possible to place the hole. The context abstracts congruence rules into the grammar, and the hole is the sub-term a computation rule can be applied in:

```
(define-extended-language
  exprT-Ctx exprT
  (C ::= hole
     (if0 C t t)
     (succ C)
     (pred C)
     (iszero C)))
```

The term (if0 C t t) shows what was just said: when no computation rules apply, the first parameter t_1 is the only one that can have a hole, so it's the only one that must be reduced as part of a congruence rule, not t_2 and not t_3. Actually, congruence rules are not defined explicitly in the relations, and instead have their behavior coded through contexts, like above.

Below, the reduction relation:

```
(define ->t
  (reduction-relation
   exprT-Ctx
   #:domain t
    (-->t (if0 true t_1 t_2)
          t_1
          "E-IfTrue"
          (judgment-holds (type-infer (if0 true t_1 t_2) T)))
    (-->t (if0 false t_1 t_2)
          t_2
          "E-IfFalse"
          (judgment-holds (type-infer (if0 false t_1 t_2) T)))
    (-->t (pred zero)
          zero
          "E-PredZero"
          (judgment-holds (type-infer (pred zero) T)))
    (-->t (pred (succ nv_1))
          nv_1
          "E-PredSucc"
          (judgment-holds (type-infer (pred (succ nv_1)) T)))
    (-->t (iszero zero)
          true
          "E-IsZeroZero"
          (judgment-holds (type-infer (iszero zero) T)))
    (-->t (iszero (succ nv_1))
          false
          "E-IsZeroSucc"
          (judgment-holds (type-infer (iszero (succ nv_1)) T)))
    with
    [(--> (in-hole C a) (in-hole C b))
     (-->t a b)]))
```

The domain option tells Redex this reduction relation acts on t. Then the computation rules are listed, with the left-hand side, right-hand side, name and a judgment restriction being

written for each rule. Afterwards, there is a *with* clause, which defines a pattern that was used in the relation definition. This **judgment-holds** option is to make sure ill-typed terms won't be executed, even when there is an execution path to be performed. For example, given the term *(if0* $true\ zero\ true$*)*, a reduction relation without a typing restriction would just allow this to reduce to $zero$, even though this term is ill-typed, because the *then* and *else* clauses don't share the same type. With the restriction, this kind of term would not be allowed to execute. However, including a judgment-holds in that position may not be the best choice, because it makes the check only for the current sub-term, and not for the entire term as a whole.

About the *with* clause: in order to represent a reduction rule, a special function called –> is used. This function holds information on the rules, like its left-hand side, right-hand side, and name. However, in order to include contexts in this relation, another function is required: **in-hole**. This function receives a context and a term, and it returns the context but with the hole filled by the provided term. When used in the definition of reduction rules, this serves as a way to generalize the computation rules as to allow them to act within the given context.

The function in-hole simply replaces the hole of a context with the term provided by a. This is done because now a is free to be pattern matched on in the reductions, and this is what is happening. The *with* clause creates a new binding –>t that receives a and b and then surrounds both parameters with in-holes and a fitting context. The first in-hole is used to allow pattern-matching on a, as stated, so the rules can make reductions *inside* the original term, regardless of what the provided context is. The second in-hole is done because as the reduction pattern matches on a, it returns the result of the reduction performed on a instead of on the original term, so (in-hole C b) places the returned b back into the hole of the original term.

For example, the term (iszero (pred $zero$)) could be contextualized as (iszero $hole$), which means a is (pred $zero$). Then, pattern matching is performed on a, which chooses the rule E-PredZero to perform. After reduction, (pred $zero$) turns into $zero$, which is bound to b, and then this b is placed back into the same hole a was taken from. So the original term (iszero (pred $zero$)) becomes, after the reduction, (iszero $zero$).

With the reduction relation established, it's possible to apply it to a term with **apply-reduction-relation** for one step or **apply-reduction-relation*** for multiple steps.

Also, Redex allows for the definition of new reduction relations based on other reduction relations by, for example, incorporating a closure to them. For example, the function **context-closure** creates a closure on the possible execution contexts, which is useful in case there is more than one context possible at any given time. The result is a reduction relation that applies a step to every context.

Redex also incorporates metafunctions in their toolkit, which are functions that act on terms of a language. Below is the definition of a metafunction that, given a term t, applies the context closure of the established reduction relation until all contexts are in normal-form, and

then returns the first value in a tuple. This metafunction takes in a term and attempts to return a value.

```
(define ->cbv (context-closure ->t exprT-Ctx C))
(define-metafunction exprT
    expr-eval : t -> v
    [(expr-eval t)
     ,(car (apply-reduction-relation* ->cbv (term t)))])
```

Finally, there is the function **redex-check**, which creates randomized and automated testing for languages. Consider this definition of redex-mach:

```
(redex-check
   exprT
   #:satisfying (type-infer t T)
   (redex-match? exprT-Ctx v (term (expr-eval t)))
   #:attempts 1000)
```

This checking instantiates 1000 *exprT* terms that satisfy the judgment *type-infer*, i.e. typable terms, and for each one of them it checks if all the normal forms reached through the context closure are values of the language. This check specifically is a way of testing if the typing relation is sound, but it can be used to test many other aspects of the language.

2.1.8 Rackcheck

Despite Redex offering options for randomly generating well-typed terms, its generation capacities are not sufficient for the work here presented. To fill this gap, the library Rackcheck is going to be used to implement a generator that is specific to CPS-calculus.

Rackcheck has tools that allow for the creation of complex generators through the composition of simpler ones, while still allowing for all of Racket's features to be used within a generator definition. For example, consider a generator that outputs an x such that either $x \in \mathbb{N}$ or $x \in [0, 10]$. This can be achieved by using a choice generator on a generator for natural numbers and a modified generator for real numbers (which by default only outputs $x \in [0, 1]$):

```
(gen:choice
  gen:natural
  (gen:map gen:real (lambda (r) (* 10 r))))
```

It's then possible to use the sampled value in another generator through *gen:bind*. In the example below, the generator first samples either a natural number or a real number between 0 and 10, as explained above. Then it binds the sampled number to n and enters into a conditional: if $n < 5$, then generate a boolean, else generate an even integer between -10 and 10. It's defined like this:

```
(gen:bind
  (gen:choice
    gen:natural
    (gen:map gen:real (lambda (r) (* 10 r))))
  (lambda (n)
```

```
    (if (< n 5)
        gen:boolean
        (gen:filter
          (gen:integer-in -10 10)
          (lambda (i)
            (= 0 (modulo i 2)))))))
```

We can define a property in Rackcheck by using the define-property function, and then test it by randomly sampling values through a generator by using the check-property function. For example, given a generator that returns a list of naturals of any length, it's possible to check the property that the reverse of a reverse of a list is the original list itself (that is, that the reverse function is its own inverse). This is done like this, taken directly from Rackcheck's documentation:

```
; taken from https://docs.racket-lang.org/rackcheck/index.html
(check-property
  (property ([xs (gen:list gen:natural)])
    (check-equal? (reverse (reverse xs)) xs)))
```

It's possible to create a custom generator for CPS-calculus commands. One advantage of using Rackcheck for this purpose is its automatic *shrinking* functionality. Upon encountering a term t that serves as a counter-example for a property being tested, instead of just returning the term as is, Rackcheck also performs shrinking steps, reducing the size of t as much as possible in an attempt to output a minimum counter-example.

2.1.9 Continuations

Continuations are functions that express what comes next in the execution of a program, i.e. how to *continue* the computation, therefore serving as an abstraction of a program's control flow (APPEL, 2007). As functions, continuations don't return anything, and instead operate as a tail call, receiving a value and then passing that value along, never returning back to the context in which it was called.

Continuations can be written in the Racket programming language in different ways. One of which is through the function *call/cc*, a function that captures the current continuation and passes it as an argument to a procedure. Calling this continuation inside the procedure in which it is bound to will jump back to the point at which the continuation was captured and replace the procedure with a new one (FELLEISEN; FINDLER; FLATT, 2009). This property of going back to a previous moment allows continuations to be used to implement loops, exceptions, early returns, and other abstractions that rely on manipulating the control flow of the program.

Below is an example of call/cc being used, demonstrating how it can jump from one part of the program to another:

```
(println (call/cc (lambda (cont)
                    (string-append ``Hello '' ``World''))))
(println (call/cc (lambda (cont)
                    (string-append ``Hello '' (cont ``World'')))))
```

These represent two distinct function calls, and both create a continuation as the argument for the function *println*. This continuation is now bound to the name cont and can be called as a function. Initially, what comes next for the program is a concatenation of the strings “Hello ” and “World”.

In the first line, cont is not called, so “Hello World” is printed on the screen exactly as if call/cc was never used.

The second line, however, results in only the text “World” being print on the screen. This happens because when (cont “World”) is called, the execution goes back to the point when the continuation was captured and replaces its original procedure, in this case the concatenation of the two strings, with the value “World”. In this case, this will essentially discard the original instruction to concatenate both strings, and will result in just “World” being print.

To reiterate, the call to the continuation does not return the string “World”. Instead, it changes the initial closure it previously captured into a new one, modifying the flow of the program as it is executed. Also, call/cc is not the only way to use continuations in Racket, and in fact, another way to use them is to pass them as an argument to the function it is a continuation of. This is called *continuation-passing style* (APPEL, 2007).

2.1.9.1 Continuation-passing style

Continuation-passing style (CPS) is a program notation that passes a continuation as an argument to functions, making it such that the function does not return, but instead makes a call to the continuation using the calculated output. This makes the control and data flow explicit, and is a notation more closely related to λ-calculus, where a program is composed by a chain of applications (APPEL, 2007). CPS is mainly used in some compilers' Intermediate Representation, as it helps to further lower the program by exposing its control flow. As an example, consider a function that outputs the next even number after an input number x. Below is an example of a program in Racket written in the usual way, so not in CPS:

```
1 (define is-even (lambda (x) (= (modulo x 2) 0)))
2 (define next-even (lambda (x) (if (is-even x) (+ x 2) (+ x 1))))
```

To rewrite this example in CPS, it's necessary to also make some auxiliary functions first to represent the basic operations of addition, modulo and equality in the style. Following the definition, instead of returning the values calculated, these functions will use the values as an input to a continuation, given as an argument:

```
1 (define add-cps (lambda (x y c) (c (+ x y))))
2 (define mod-cps (lambda (x y c) (c (modulo x y))))
3 (define equal-cps (lambda (x y c) (c (= x y))))
```

These functions use the corresponding primitives available (the functions that do return a value) and pass their outputs to a continuation c, given by the function calling them. Now back to the original functions is-even and next-even, below is an equivalent function to is-even in CPS:

```
(define is-even-cps (lambda (x c)
                      (mod-cps x 2 (lambda (k)
                                     (equal-cps k 0 c)))))
```

First, this function calculates the modulo of x with 2 using mod-cps, inputting the result to a newly defined continuation. This continuation is a lambda function with one parameter k, which is going to be the result of the modulo, following from mod-cps' definition. This function then calls equal-cps to check if the modulo is equal to 0 and, following from the definition of equal-cps, inputs the boolean result to the parameter continuation c.

Finally, the next-even function in CPS:

```
(define next-even-cps (lambda (x c)
                        (is-even-cps x (lambda (b)
                                         (if b
                                             (add-cps x 2 c)
                                             (add-cps x 1 c))))))
```

This function calls the is-even-cps function, inputting x and a new continuation, a lambda function whose input is used as the guard in an if statement. Recalling the definition of is-even-cps, this is the continuation that equal-cps calls. On the clauses, the addition is done with the next-even-cps continuation.

In the non-CPS version of this function, one could call it for the number 2 by writing (next-even 2). However, the CPS version requires a continuation to be provided, which will indicate what comes next for the computation. As in this example next-even-cps is the end-goal, the identity function can be used to finish the process. Then, calling (next-even-cps 2 identity) will pass the identity function to add-cps, which will make it behave similarly to the regular + operator.

Writing functions like this results in values not being directly returned, but instead forming a function concatenation chain in which one value returns only when **all** values return. The only case where this is not true is with primitive functions, like add-cps, mod-cps and equal-cps (APPEL, 2007).

2.1.10 CPS-calculus

CPS-calculus is a name-passing calculus formalized in Thielecke (1997b), using as inspiration the CPS-based IR developed in Appel (1992). Its terms can be referred to as *commands* (TORRENS; ORCHARD; VASCONCELLOS, 2024), and are generated by only two grammar rules: one representing a jump to a continuation and the other a bind of one. The grammar rules are defined below (THIELECKE, 1997b):

$$\begin{aligned} \langle \mathrm{M} \rangle ::= & \; x\langle \vec{x} \rangle \\ & \mid \langle \mathrm{M} \rangle \; \{ x\langle \vec{x} \rangle = \langle \mathrm{M} \rangle \} \end{aligned}$$

The first rule is a jump to the continuation x with parameter list $\vec{x}$. The second rule binds the continuation named x to the outer command on the left and the parameters in $\vec{x}$ to the inner command on the right (THIELECKE, 1997b). An important feature of CPS commands is that they do not return, a property shared by functions written in continuation-passing style (TORRENS; ORCHARD; VASCONCELLOS, 2024).

The version of CPS-calculus studied in this work is polyadic, meaning the jumps and binds can take multiple parameters, as opposed to the previously elaborated upon monoadic structure of λ expressions. It's also non-recursive, so in a bind such as $c_1\ \{k\langle\vec{x}\rangle = c_2\}$, the variable k is not bound in c_2. The recursive variant of the calculus is often written in the literature with $\Leftarrow$ instead of $=$ for bind commands (THIELECKE, 1997a).

Following the terminology used in Merro (2010), but adapting it to the polyadic version of the calculus, given a jump $k\langle\vec{x}\rangle$, it's said that k is in *subject* position and the arguments $x \in \vec{x}$ are said to be in *object* position. This same terminology is extended to include such k and x in binds of the form $c_1\{k\langle\vec{x}\rangle = c_2\}$ as well.

As inspired by an IR, the CPS-calculus was born in the context of compilers, and it has been theoretically explored as a possible candidate to be used as a foundational calculus behind IR implementations (TORRENS; ORCHARD; VASCONCELLOS, 2024).

2.1.10.1 Evaluation Contexts

Context rules for the monoadic version of CPS-calculus were given in Merro (2010). The rules take into account how a bind has two recursive spots, so the hole could be located inside either of them. Here, a hole is given by the [—] symbol. An adaption of those rules to the polyadic grammar is immediate and is as follows:

$$\begin{aligned}\langle \mathrm{C}\rangle ::= {} & [—] \\ \mid {} & \langle \mathrm{C}\rangle\ \{x\langle\vec{x}\rangle = \langle \mathrm{M}\rangle\} \\ \mid {} & \langle \mathrm{M}\rangle\ \{x\langle\vec{x}\rangle = \langle \mathrm{C}\rangle\}\end{aligned}$$

When a context is generated only by applying the first two rules, then it's said to be a *static context* (MERRO, 2010), and its hole (or the command it captures) is in *head position* (TORRENS; ORCHARD; VASCONCELLOS, 2024). If the jump in head position is reducible, then it can also be called a *head redex*.

2.1.10.2 Axiomatic Semantics

Before showing axioms of CPS-calculus, it's necessary to define a few functions that act on CPS terms. First, the definition for the *FV* function, which extracts the set of a command's

free variables. The rules are adapted from Thielecke (1997b):

$$\text{FV}(k\langle\vec{x}\rangle) \equiv \{k\} \cup \vec{x}$$
$$\text{FV}(M\{k\langle\vec{x}\rangle = N\}) \equiv (\text{FV}(M) \setminus \{k\}) \cup (\text{FV}(N) \setminus \vec{x})$$

And now the definition for the *dom* function, which extracts the bound variables in a context. The rules are adapted from Torrens, Orchard e Vasconcellos (2024):

$$\text{dom}([\text{—}]) = \{\emptyset\}$$
$$\text{dom}(\text{C}\ \{k\langle\vec{x}\rangle = cmd\}) \equiv \{k\} \cup \text{dom}(\text{C})$$
$$\text{dom}(cmd\ \{k\langle\vec{x}\rangle = \text{C}\}) \equiv \vec{x} \cup \text{dom}(\text{C})$$

We utilize $[x \mapsto y]c$ to represent a capture-avoiding substitution of variable x with variable y in term c. The syntax $[\vec{x} \mapsto \vec{y}]c$ is syntactic sugar to perform multiple substitutions at once, and it requires $\vec{x}$ and $\vec{y}$ to have the same number of elements.

Finally, the axiomatic theory, which serves as the basis for all semantics to be developed for the CPS-calculus (MERRO, 2010). The rules are taken from Thielecke (1997a) and pertain to this specific version of the calculus:

$$L\{m\langle\vec{x}\rangle = M\}\{n\langle\vec{y}\rangle = N\} \equiv L\{n\langle\vec{y}\rangle = N\}\{m\langle\vec{x}\rangle = M\{n\langle\vec{y}\rangle = N\}\}, \quad \text{(DISTR)}$$
$$m \neq n \wedge m, \vec{x} \notin \text{FV}(N)$$
$$k\langle\vec{y}\rangle\{n\langle\vec{z}\rangle = N\} \equiv k\langle\vec{y}\rangle, \quad n \notin \text{FV}(k\langle\vec{y}\rangle) \quad \text{(GC)}$$
$$n\langle\vec{y}\rangle\{n\langle\vec{z}\rangle = N\} \equiv [\vec{z} \mapsto \vec{y}]N \quad \text{(JMP)}$$
$$M\{n\langle\vec{x}\rangle = n'\langle\vec{x}\rangle\} \equiv [n \mapsto n']M \quad \text{(ETA)}$$

The (JMP) rule is what drives computation, performing a jump to the subject n by replacing the redex jump with N, but modified such that every instance of its parameters are substituted with the object arguments provided. However, that rule requires that the subject continuation is adjacent to the jump, which is not always the case. The (DISTR) rule is able to reorder the binds, such that the (JMP) rule is applicable, but in order to do so it needs to duplicate binds internally, increasing the size of the command. The (GC) rule removes unused binds, which can help clear up the copies made by (DISTR). Lastly, (ETA) performs eta reduction, removing continuations whose only purpose is to pass along its parameters to a different continuation (TORRENS; ORCHARD; VASCONCELLOS, 2024). Most of these rules were first devised in Appel (1992).

The condition for a CPS term to be *stuck* with respect to these axioms is to have a command that should result in a jump, like $k\langle\vec{x}\rangle\{k\langle\vec{y}\rangle = N\}$, but that has mismatch between the lengths of the argument list $\vec{x}$ and the parameter list $\vec{y}$ (TORRENS; ORCHARD; VASCONCELLOS, 2024).

One last development useful to this work is an extension to these axioms that was given by Thielecke (1997a) for the linear variant of the CPS-calculus. This extension comprises two rules, (FLOAT-L) and (FLOAT-R), and is meant to represent structural equivalence through the reordering of binds, similarly to (DISTR). These rules can be derived for the non-linear version of the calculus by using the axioms that were already defined (TORRENS; ORCHARD; VASCONCELLOS, 2024). Their rules can be found below, and are valid up to α-convertibility:

$$L\{m\langle\vec{x}\rangle = M\}\{n\langle\vec{y}\rangle = N\} \equiv L\{n\langle\vec{y}\rangle = N\}\{m\langle\vec{x}\rangle = M\}, \qquad \text{(FLOAT-L)}$$
$$m \not\equiv n \wedge n \notin \text{FV}(M)$$
$$L\{m\langle\vec{x}\rangle = M\}\{n\langle\vec{y}\rangle = N\} \equiv L\{m\langle\vec{x}\rangle = M\{n\langle\vec{y}\rangle = N\}\}, \qquad \text{(FLOAT-R)}$$
$$m \not\equiv n \wedge n \notin \text{FV}(L)$$

2.1.10.3 Operational Semantics

In Merro e Sangiorgi (1998), the Head Reduction relation was presented, which only reduces the CPS-calculus in static contexts, i.e. only redexes in head position. This reduction, as will be the case with every CPS-calculus semantics discussed from this point on, is built on top of the axiomatic semantics of a CPS-calculus variant.

Later, Torrens, Orchard e Vasconcellos (2024) expanded on this development by formalizing the Jump Reduction ($\rightsquigarrow_j$), which allows jumps to be performed in all contexts. In other words, Jump Reduction reduces any redex, not only the one in head-position. It's defined as the compatible closure of this rule, adapted from Torrens, Orchard e Vasconcellos (2024):

$$\text{C}\,[k\langle\vec{x}\rangle]\{k\langle\vec{y}\rangle = c\} \rightsquigarrow_j \text{C}\,[[\vec{y} \mapsto \vec{x}]c]\{k\langle\vec{y}\rangle = c\}$$
$$\text{w.l.o.g., } k \notin \text{dom(C)}$$

This reduction works by searching inside a context for a jump that can be performed to a continuation immediately outside of it. When such a jump exists, and if that jump's id is not bound somewhere else in the context (this makes sure the jump's bound to the continuation outside of C), a capture-avoiding substitution is performed on cmd_0, replacing the continuation's defined parameters with the ones provided by the jump.

Alongside the Jump Reduction, the authors also define a Garbage Collection Reduction ($\rightsquigarrow_{gc}$), based on the (GC) axiom and aimed at counteracting Jump Reduction's property of potentially increasing the size of the term after it's applied, at worst almost doubling it in one step (TORRENS; ORCHARD; VASCONCELLOS, 2024). It's defined as the compatible closure of this rule:

$$b\{k\langle\vec{y}\rangle = c\} \rightsquigarrow_{gc} b$$
$$\text{given } k \notin \text{FV}(b)$$

All this does is remove continuation binds that are never used in the scope to which they are bound.

Combining these two relations, Full Reduction is defined ($\rightsquigarrow \equiv \rightsquigarrow_j \cup \rightsquigarrow_{gc}$) (TORRENS; ORCHARD; VASCONCELLOS, 2024).

2.1.10.4 Typing Relation

A typing system was developed for the CPS-calculus in Thielecke (1997a). This system consists of an inductive polyadic negation type and a set of base types *X*. The types are defined by the grammar below:

$$\tau \ ::= \ \neg(\tau_1, \ \ldots, \ \tau_n) \quad | \quad X$$

As previously mentioned, CPS-calculus commands don't return. A consequence of this is that they also don't have types, unlike the simply typed λ-calculus, for example. However, their free variables do, and that's the scope in which the typing relation operates on. Notably, no term of the non-recursive version of CPS-calculus is closed, i.e., all non-recursive terms have at least one free variable (TORRENS; ORCHARD; VASCONCELLOS, 2024).

The goal of a checking typing relation for the CPS-calculus is to verify if a given free variables environment, which is an associative list mapping a variable name with its type, makes the term well-typed. Alternatively, an inferring typing relation finds such an environment, not necessarily containing exact types, but general unified ones that represent a minimum pattern that needs to be followed in order for the type to obey all the typing constraints.

Considering Γ to be the free variables environment, the typing rules are defined below (THIELECKE, 1997a):

$$\frac{}{\Gamma, \ k : \neg\vec{\tau}, \ \vec{y} : \vec{\tau} \ \vdash \ k\langle\vec{y}\rangle} \text{ (Jump)} \qquad \frac{\Gamma, \ n : \neg\vec{\tau} \ \vdash \ M \qquad \Gamma, \ \vec{y} : \vec{\tau} \ \vdash \ N}{\Gamma \ \vdash \ M\{n\langle\vec{y}\rangle = N\}} \text{ (Bind)}$$

With the exception of a jump with no arguments or a bind with no parameters, like $k\langle\rangle$, typing rules calculate a variable's type with relation to other variables. In a jump, like in $k\langle x, y\rangle$, the outer variable k's type is $\neg(X, Y)$, where X and Y are the types for x and y, respectively. Similarly in a bind, like in $cmd_1\{k\langle x, y\rangle = cmd_2\}$, the newly defined continuation k's type is also $\neg(X, Y)$.

A consequence of this behavior is that sometimes calculated types are not specific, and instead merely provides patterns indicating dependencies. To illustrate this, consider this command: $k\langle x\rangle$. If an inferring typing judgment is run on it, the output is something like: $k : \neg(A)$ and $x : A$. Any type A works to make this command well-typed. These typing holes, like A, are closely related to variables being present on the right-hand side of substitutions in an unifier. In fact, the unification algorithm presented in 2.1.5 will be used later on to implement this typing relation.

Now consider this other command: $k\langle x\rangle\{x\langle\rangle = y\langle\rangle\}$. After running the typing judgment, the types calculated for k and x are now specific, with $k : \neg(\neg())$ and $x : \neg()$. This happens

because x is defined as a continuation with zero parameters, forcing what was once a general A to be typed as $\neg()$.

Note that this is different than the types being polymorphic within a command as a whole. This is a feature of the inferring typing judgment, which happens due to a command lacking information to further specialize a variable's type, but not a feature of the types themselves. If a variable with a general type gets specialized, any further usage of this variable must obey this same specialization, even if using a different type would not result in the term getting stuck (for example, the parameter for a continuation representing a constant function), as to preserve the type system's conservatism.

2.1.10.5 CPS-calculus $\leftrightarrow$ λ-calculus Translations

Finally, CPS-calculus translations to and from λ-calculus exist, and can be useful to integrate the corpus of knowledge already derived regarding λ-calculus with CPS-calculus.

In Thielecke (1997b), a translation from CPS-calculus to λ-calculus was defined, written as cmd°, where cmd is a CPS-calculus command. It has these rules:

$$k\langle y_1,\ \ldots,\ y_n\rangle^\circ \equiv k y_1 \ldots y_n$$
$$M\{k\langle y_1,\ \ldots,\ y_n\rangle = N\}^\circ \equiv (\lambda k.M^\circ)(\lambda y_1 \ldots y_n.N^\circ)$$

Note that the second rule contains a polyadic abstraction, but here this is just used as syntactic sugar to represent currying.

In Thielecke (1997a), translations going the other way (from λ-calculus to CPS-calculus) were defined. Specifically, there are two translations, one for the call-by-name evaluation of λ-calculus (written as $[\![—]\!]_N$) and the other for the call-by-value evaluation (written as $[\![—]\!]_V$). The CBN translation is defined like this:

$$[\![x]\!]_N \equiv x\langle k\rangle$$
$$[\![\lambda x.e]\!]_N \equiv k\langle v\rangle\{v\langle x,\ k\rangle = [\![e]\!]_N\}$$
$$[\![f\ e]\!]_N \equiv [\![f]\!]_N\{k\langle f\rangle = f\langle v, k\rangle\{v\langle k\rangle = [\![e]\!]_N\}\}$$

And the CBV translation is defined like this:

$$[\![x]\!]_V \equiv k\langle x\rangle$$
$$[\![\lambda x.e]\!]_V \equiv k\langle v\rangle\{v\langle x,\ k\rangle = [\![e]\!]_V\}$$
$$[\![f\ e]\!]_V \equiv [\![f]\!]_V\{k\langle f\rangle = [\![e]\!]_V\{k\langle v\rangle = f\langle v, k\rangle\}\}$$

And now expanding the scope to the simply-typed λ-calculus, typed translations from it to CPS-calculus were defined in Torrens, Orchard e Vasconcellos (2024). Consider λ-calculus' functional types being defined as follows:

$$A ::= A \to A \quad | \quad X$$

Then, rules for transforming a functional type A into a CPS-calculus type τ is as follows, again differentiating between CBN and CBV and starting with CBN:

$$[\![X]\!]_N \equiv \neg X$$
$$[\![A \to B]\!]_N \equiv \neg(\neg(\neg([\![A]\!]_N),\ [\![B]\!]_N))$$

And now the CBV type translation:

$$[\![X]\!]_V \equiv X$$
$$[\![A \to B]\!]_V \equiv \neg([\![A]\!]_V,\ \neg([\![B]\!]_V))$$

And lastly, given a λ-calculus well-typed judgment $\vec{x} : \vec{A} \vdash_\lambda e : B$, it's possible to fully translate the judgment to CPS-calculus by applying the appropriate transformations on both λ-calculus expressions and types, as follows (TORRENS; ORCHARD; VASCONCELLOS, 2024):

$$[\![x_1 : A_1,\ \ldots,\ x_n : A_n \vdash_\lambda e : B]\!]_N \equiv x_1 : \neg([\![A_1]\!]_N),\ \ldots,\ x_n : \neg([\![A_n]\!]_N),\ k : [\![B]\!]_N \vdash [\![e]\!]_N$$
$$[\![x_1 : A_1,\ \ldots,\ x_n : A_n \vdash_\lambda e : B]\!]_V \equiv x_1 : [\![A_1]\!]_V,\ \ldots,\ x_n : [\![A_n]\!]_V,\ k : \neg([\![B]\!]_V) \vdash [\![e]\!]_V$$

2.2 Related works

2.2.1 Lightweight mechanization using Redex

Redex (FELLEISEN; FINDLER; FLATT, 2009) is a DSL for Racket that is used to test, analyze and study the syntax and semantics of a language. For that purpose, it allows the user to mechanically define languages and calculi, and then use these definitions to establish relations, metafunctions, etc.

It is within this work's objectives to implement the CPS-calculus in Redex and create test suites to mechanically verify some of its semantics. To this end, Klein et al. (2012) offers important insight on testing of published papers. In that work, the authors differentiate between heavyweight and lightweight mechanization.

Heavyweight mechanization involves using tools such as Coq or Agda to create machine-checked proofs of theorems and properties of the language (KLEIN et al., 2012). This is extremely useful and academically rigorous, but it demands a lot of work on the part of the researcher, and if the definitions developed contain issues, they also might spend a long time trying to prove something that is, unbeknowst to them at this point, false.

Lightweight mechanization offers an alternative; or perhaps, a prelude. With that form of mechanization, researchers can implement the language in a programming environment and create tests for its semantics. This type of mechanization could involve creating interpreters and typecheckers in a functional programming language, such as Haskell (KLEIN et al., 2012).

Then, this mechanized model of the language could be tested through randomized property-based testing libraries, like Haskell's QuickCheck (CLAESSEN; HUGHES, 2000). However, differences between the researched language's mathematical model and the programming language's own syntax can lead to errors in the implementation; also, the process of implementing all the tools necessaries for this mechanization to work can lead to implementation errors that are not necessarily related to the language's theoretical definitions at all, besides being a time-consuming process (KLEIN et al., 2012).

Redex is another option to program lightweight language models. It uses the Racket programming language, but its specific syntax is similar to what one would see in a programming language theory paper, allowing for quick translation between the theoretical source's typesetting and the programmed model. Also, because Redex is a DSL designed specifically for semantics engineering, it bolsters an extensive suite of tools for implementing and testing the mechanized language (KLEIN et al., 2012) (FELLEISEN; FINDLER; FLATT, 2009).

In Klein et al. (2012), the authors took nine papers from the International Conference on Functional Programming 2009 proceedings and mechanized their content in Redex to look for errors. They found errors in all nine papers, which were then confirmed by the authors of the papers themselves.

Of special interest to this work, three out of the nine papers studied in Klein et al. (2012) had continuations in their scope of study. Namely, these papers are McCarthy (2009), Rompf, Maier e Odersky (2009), and Midtgaard e Jensen (2009). Two of these, Rompf, Maier e Odersky (2009) and Midtgaard e Jensen (2009), involved CPS transformations in their work.

Even though none of these three involve CPS-calculus specifically, the Redex implementation of these papers, which were developed and made available by Klein et al. (2012), will serve as important reference tools to this work's own implementation of CPS-calculus in Redex.

2.2.2 Stochastic generation of well-typed programs

This work's goal of mechanizing a version of CPS-calculus' operational semantics stems from the utility of randomized testing as a prelude to rigorous proofs about a language. In other words, it's useful being able to test properties of a language, so that some errors can be found early, before someone attempts to prove a negative. However, mechanized operational semantics are only one side of the testing relationship. It's also needed to explore the other: the test cases.

Efficiently generating a diverse set of programs for automated testing is an active field of research, specially when the programs generated must obey certain criteria, such as being well-typed or to fit into a notion of "usefulness"(for example, minimizing the number of functions or variables that are defined but never referenced).

In Yang et al. (2011), a generator for C programs was developed with the purpose of finding bugs in compilers. This generator, Csmith, is able to create C programs while avoiding

undefined or unspecified behaviors that influences the meaning of the program, which is important to determine that a found anomaly is indeed a bug with respects to one or more expected behaviors. The paper reported having found and reported more than 325 new bugs to compiler developers, including open source and closed source compilers. Their test procedure includes *randomized differential testing*, which runs randomly generated valid C programs with several different compilers and then compares all of their outputs. Through a voting system, the ones that return a result shared by the majority are deemed as not buggy, while those that return an outlier output are deemed as buggy (YANG et al., 2011).

Something similar was done in Graeff et al. (2024), but with Haskell, a purely functional programming language. Contrary to the Csmith paper (YANG et al., 2011), the goal of this project is to find bugs in just one compiler, the Glasgow Haskell Compiler (GHC), but now with different optimization flags enabled, searching for both compilation errors and also for different outputs on different optimization levels, something that should not happen for an optimization to be deemed correct (ALFRED; MONICA; JEFFREY, 2007). This project developed a generator of Haskell programs based on an extended version of λ-calculus, such that its generation process is guided by the λ-calculus' type system (GRAEFF et al., 2024). This is specially significant due to the fact CPS-calculus has well-defined translations to-and-from simply-typed λ-calculus (THIELECKE, 1997a) (THIELECKE, 1997b), making the theoretical foundations behind that project relevant for the work here presented. Furthermore, the authors employed Haskell's Quickcheck library (CLAESSEN; HUGHES, 2000) to implement the generator and to perform tests on it, which is exactly what is to be done in this work, but with Racket's Rackcheck library instead. The generator in question works by first creating a type accepted by the language and then by creating an expression that has this type, and if the generated expression needs sub-expressions of certain types, then a recursive step is taken, continuing with the recursion until there are no more required sub-expressions or until the recursive depth exceeds a maximum depth parameter (GRAEFF et al., 2024). The authors generated 10.000 Haskell programs, half of them with maximum depth 5 and half of them with maximum depth 15, and found 2 and 55 errors, respectively, all of them in the lower optimization levels O0 and O (GRAEFF et al., 2024).

Another relevant work is what was done in Feitosa, Ribeiro e Bois (2019). There, the authors presented a generator of well-typed Featherweight Java programs, also written using Haskell's Quickcheck (CLAESSEN; HUGHES, 2000). Featherweight Java (IGARASHI; PIERCE; WADLER, 2001) (FJ) is a small calculus meant to abstract away several Java features in order to create a calculus representation of programs in that language. The generation is broken down into two phases: first, a set of classes are defined to make up an environment. Then, an expression is built using the classes defined in the previous step. Like what was done in Graeff et al. (2024), this generation process is also guided by the typing system, first selecting a desired type for the expression, and then using FJ's typing semantics to guide the recursion towards an expression whose type matches the desired one.

The exact type-directed procedure of first generating a valid type and then creating an expression that fills in this type, like what was done in Feitosa, Ribeiro e Bois (2019) and Graeff et al. (2024), is not possible for CPS-calculus. This is because its terms don't have types, only the variables inside a term do (TORRENS; ORCHARD; VASCONCELLOS, 2024). However, these ideas are still very relevant for the development of a suitable generator, because they show ways to use a typing system to guide program generation. On that front, another relevant project is Fetscher et al. (2015), which uses PLT Redex (FELLEISEN; FINDLER; FLATT, 2009) to define a language-agnostic procedure to create well-typed terms based on a language's typing semantics. Our current goal is aiming for more specificity towards CPS-calculus, but the ideas behind their general method can be taken as a basis for some of the decisions taken in ours, such as the usage of the unification algorithm to solve typing constraints.

2.3 Conclusion

In this chapter, most of the concepts relevant to what comes next in this work were explored. The theoretical foundations presented can be used as a reference throughout later chapters.

3 CPS-calculus Implementation

The contributions this work seeks to provide require a mechanized implementation of CPS-calculus. Regarding the tool to perform for such mechanization, PLT Redex (FELLEISEN; FINDLER; FLATT, 2009) has been picked. What follows in this chapter is a PLT Redex implementation of a syntax for CPS-calculus, followed by several of its semantics as explored in section 2.1.10.

3.1 CPS-calculus' syntax

As previously mentioned, the version of CPS-calculus that was chosen to be implemented is the polyadic and non-recursive variant. Its syntax was defined in 2.1.10. This is how it was implemented in Redex:

```
(define-language CPS-Calculus
  (cmd ::= (id ⟨ args ⟩)
           (cmd \{ id ⟨ args ⟩ = cmd \}))
  (id ::= variable-not-otherwise-mentioned)
  (args ::= (id ...))
  #:binding-forms
  (cmd_0 #:refers-to id_0 \{ id_0 ⟨ (id_1 ...) ⟩ =
          cmd_1 #:refers-to (shadow id_1 ...) \}))
```

Three Redex details deserve mentioning: First, the special name *variable-not-otherwise-mentioned* is a catch-all in Redex, as this rule will match with any variable name that is not part of the non-terminals of the language. Second, the syntax $(id...)$ represents a list of ids of any size. Third, there are two binding forms: the first is binding the name of the continuation to the outer command and the second is binding the parameters of that same continuation to its body, the inner command.

Now, we define the context in which reductions can take place. Explanations of how CPS-calculus contexts work are given in section 2.1.10.1. Its Redex implementation is located below:

As a small example of the recursion, here's how the command $k\langle a\rangle\{k\langle x\rangle = x\langle\rangle\}$ is rendered as a term in Redex:

((k ⟨ (a) ⟩) {k ⟨ (x) ⟩ = (x ⟨ () ⟩) })

```
(define-extended-language
  CPS-Calculus-Ctx
  CPS-Calculus
  (C ::= hole
         (C \{ id ⟨ args ⟩ = cmd \})
         (cmd \{ id ⟨ args ⟩ = C \})))
```

3.2 Metafunctions

Now that the language's syntax is defined, it's possible to create some functions that act on the terms of the language, also called *metafunctions*. To start, the union and difference of sets (actually implemented on lists, but for their usage this will make no difference):

```
;Set functions (actually done on lists)
;source:
; https://docs.racketlang.org/redex/Other_Relations.html)

(define-metafunction CPS-Calculus
  ∪ : (id ...) ... -> (id ...)
  [(∪ (id_1 ...) (id_2 ...) (id_3 ...) ...)
   (∪ (id_1 ... id_2 ...) (id_3 ...) ...)]
  [(∪ (id_1 ...))
   (id_1 ...)]
  [(∪) ()])

(define-metafunction CPS-Calculus
  - : (id ...) (id ...) -> (id ...)
  [(- (id ...) ()) (id ...)]
  [(- (id_1 ... id_2 id_3 ...) (id_2 id_4 ...))
   (- (id_1 ... id_3 ...) (id_2 id_4 ...))
   (side-condition (not (memq (term id_2) (term (id_3 ...)))))]
  [(- (id_1 ...) (id_2 id_3 ...))
   (- (id_1 ...) (id_3 ...))])
```

These metafunctions are going to be used to calculate the set of free variables of a term. The function to do exactly that, FV, is defined below (THIELECKE, 1997b):

```
(define-metafunction CPS-Calculus
  FV : cmd -> (id ...)
  [(FV (id_0 ⟨ args_0 ⟩)) (∪ (id_0) args_0)]
  [(FV (cmd_0 \{ id_0 ⟨ args_0 ⟩ = cmd_1 \}))
   (∪ (- (FV cmd_0) (id_0)) (- (FV cmd_1) args_0))])
```

Another set that is needed for this work is called *dom*. It is the set containing the variables a context C captures/binds in its hole (TORRENS; ORCHARD; VASCONCELLOS, 2024):

```
(define-metafunction CPS-Calculus-Ctx
  dom : C -> (id ...)
  [(dom hole) ()]
  [(dom (C_0 \{ id_0 ⟨ args_0 ⟩ = cmd_0 \}))
   (∪ (id_0) (dom C_0))]
  [(dom (cmd_0 \{ id_0 ⟨ args_0 ⟩ = C_0 \}))
   (∪ args_0 (dom C_0))])
```

The definition of both of these sets can be found in section 2.1.10.2.

3.3 Typing Semantics

Here we implement the typing system developed in Thielecke (1997a) for the CPS-calculus. Details regarding this typing system can be found in section 2.1.10.4.

First, however, it's necessary to implement an unification algorithm. It was chosen the one formalized in Martelli e Montanari (1982), due to its ability to return the most general unifier in linear time. Details on this algorithm can be found in section 2.1.5. What was implemented was an adaptation of that algorithm to fit CPS-calculus' types, where all functions are just ¬. The implementation was done as a metafunction and uses an extended version of the CPS-calculus language, now holding types, type environments and unification constraints:

```
(define-extended-language
  CPS-Calculus-Typed
  CPS-Calculus-Ctx
    (τ ::= id (¬ (τ ...)))
    (Γ ::= · (Γ id τ))
    (constraints ::= #f · (constraints τ τ)))
```

The unification algorithm defined in Martelli e Montanari (1982) is non-deterministic, because it processes the equations in any order. To make it deterministic, the equations were organized as a list and then processed in that order, with some changes to the rule (6) in 2.1.5. The metafunction receives two lists of constraints, the first one being the original list of equations to be processed and the second one being empty at first, to be populated as the iterations go. Instead of rule (6) adding back the (w.l.o.g) $X = \alpha$ equation to the original list, it adds it to the second list instead, and the $([X \mapsto \alpha]\ \Pi)$ acts on both lists instead of only the first (but still not on $X = \alpha$).

When the first list is empty, it is guaranteed that the second list is already in solved form, so the function just returns it. This guarantee is not a new property, but derived directly from the original algorithm, specifically how the rule (6) marks an equation in the set as solved, by acting on it only if it's of the form $X = \alpha$ and by then modifying any previously solved and non-solved equations to not include X. The implementation is as follows:

```
(define-metafunction CPS-Calculus-Typed
  unification : constraints constraints -> constraints
  ; return solved
  [(unification · constraints_1) constraints_1]
  ; rule (1)
  [(unification (constraints_0 (¬ (τ_0 ...)) id_0) constraints_1)
   (unification (constraints_0 id_0 (¬ (τ_0 ...))) constraints_1)]
   ; rule (2)
  [(unification (constraints_0 id_0 id_0) constraints_1)
   (unification constraints_0 constraints_1)]
   ; rule (3)
  [(unification (constraints_0 (¬ (τ_0 ..._!_0)) (¬ (τ_1 ..._!_0)))
                constraints_1) #f]
  ; rule (4)
  [(unification (constraints_0 (¬ (τ_0 ..._0)) (¬ (τ_1 ..._0)))
                constraints_1)
   (unification (decompose-types constraints_0 (τ_0 ...) (τ_1 ...))
                constraints_1)]
   ; rule (5)
  [(unification (constraints_0 id_0 τ_0) constraints_1)
   #f
   (side-condition (memq (term id_0) (term (FV-types τ_0))))]
   ; rule (6) modified
```

```
  [(unification (constraints_0 id_0 τ_0) constraints_1)
   (unification (substitute constraints_0 id_0 τ_0)
                ((substitute constraints_1 id_0 τ_0) id_0 τ_0))])
```

This implementation calls two other metafunctions: FV-types and decompose-types. These are implemented as such:

```
(define-metafunction CPS-Calculus-Typed
  FV-types : τ -> (id ...)
  [(FV-types id_0) (id_0)]
  [(FV-types (¬ (τ_0 ...))) (∪ (FV-types τ_0) ...)])

(define-metafunction CPS-Calculus-Typed
  decompose-types : constraints (τ ...) (τ ...) -> constraints
  [(decompose-types constraints_0 () ()) constraints_0]
  [(decompose-types constraints_0 (τ_0 τ_1 ...) (τ_2 τ_3 ...))
   ((decompose-types constraints_0 (τ_1 ...) (τ_3 ...)) τ_0 τ_2)])
```

With this, it's possible to implement the typing semantics. The implementation corresponds to the process of inferring a free variables environment, rather than checking if a provided one correctly types the command. It's able to find the most general type for each variable by finding the principal unifier for the set of typing constraints. This most general type is called that variable's *principal type*, much like the most general unifier is called the principal unifier.

As the types in the inferred environment Γ are the most general types for each variable, turning this into a checker for a provided environment $\hat{\Gamma}$ only involves checking if $(\hat{\Gamma} \subseteq \Gamma) \wedge (\exists\gamma \,.\, \forall(k,\tau) \in \Gamma \,.\, \gamma(\tau) \equiv \hat{\Gamma}(k))$, where γ is a potentially empty set of substitutions and $\gamma(\tau)$ is an application of those substitutions to τ. In other words: to check if every variable in the inferred Γ also exists in the provided $\hat{\Gamma}$ and if there exists a set of substitutions γ that can transform every type in the inferred Γ to the corresponding provided one in $\hat{\Gamma}$, i.e., if it's possible to specialize the types in Γ by substituting typing variables in a consistent way across all free variables in the environment in order to make these types equal to their corresponding ones in $\hat{\Gamma}$.

Now back to the inference relation. It takes in a CPS-calculus command and either returns false, if the command is not typable, or a free variables environment mapping each free variable to a principal type. It follows the two rules originally given by Thielecke (1997a), but with some modifications on the one for binds to incorporate unification at every step. Specifically, after the two recursive premises of the original rule and given a bind of the form $M\{n\langle\vec{y}\rangle = N\}$, the algorithm checks if the union of the environments inferred on both premises with a new constraint relating the new bind n with the type $\neg\vec{y}$ has an unifier. If it doesn't, then the command is deemed not typable and the judgment fails. If it does, then the principal unifier is used to transform every type in the environment to be returned, after removing n and $\vec{y}$, into its principal type.

Both the Jump (used without any modification) and the Bind rules are explained as they are originally defined in section 2.1.10.4. The judgment implementation also uses several auxiliary metafunctions, which will be defined shortly. The judgment is implemented as follows:

```
(define-judgment-form CPS-Calculus-Typed
  #:mode (typing O I)
  #:contract (typing Γ cmd)

  [------------------------------------------------------- "J"
   (typing ((args-to-env args_0) id_0 (¬ args_0))
           (id_0 ⟨ args_0 ⟩))]

  [(typing Γ_0 cmd_0)
   (typing Γ_1 cmd_1)
   (where constraints_0
          (unification ((union-envs Γ_0 Γ_1) id_0 (¬ args_0)) ·))
   (where #t (check-if-united constraints_0))
   (where Γ_2 (remove-duplicates-in-env
               (substitute-into-env
                 (union-envs
                   (remove-from-env id_0 Γ_0)
                   (remove-args-from-env args_0 Γ_1))
                 (flatten-constraints constraints_0 ())) ()))
   ------------------------------------------------------- "B"
   (typing Γ_2 (cmd_0 \{ id_0 ⟨ args_0 ⟩ = cmd_1 \}))])
```

This judgment uses some new auxiliary metafunctions. Starting with the rule for jumps, there is one called args-to-env. This metafunction instantiates a new environment whose only elements are the arguments in the jump. Actually, this metafunction shows an interesting implementation detail: at first, the type of every argument is just the name of the argument. Its implementation is as follows:

```
(define-metafunction CPS-Calculus-Typed
  args-to-env : args -> Γ
  [(args-to-env ()) ·]
  [(args-to-env (id_0 id_1 ...))
   ((args-to-env (id_1 ...)) id_0 id_0)])
```

Now going to the rule for binds, there's the check-if-united, remove-from-env, remove-args-from-env, union-envs, flatten-constraints and substitute-into-env metafunctions. The check-if-united metafunction only checks if the unification was successful or not:

```
(define-metafunction CPS-Calculus-Typed
  check-if-united : constraints -> boolean
  [(check-if-united #f) #f]
  [(check-if-united constraints_0) #t])
```

Then about the remove-from-env and remove-args-from-env. These do basically the same thing: remove one or multiple, respectively, entries from the variables environment based on the identifier key.

```
(define-metafunction CPS-Calculus-Typed
  remove-from-env : id Γ -> Γ
  [(remove-from-env id_0 ·) ·]
  [(remove-from-env id_0 (Γ_0 id_0 τ_0))
   (remove-from-env id_0 Γ_0)]
  [(remove-from-env id_0 (Γ_0 id_1 τ_0))
   ((remove-from-env id_0 Γ_0) id_1 τ_0)])
```

```
(define-metafunction CPS-Calculus-Typed
  remove-args-from-env : args Γ -> Γ
  [(remove-args-from-env () Γ_0) Γ_0]
  [(remove-args-from-env (id_0 id_1 ...) Γ_0)
   (remove-args-from-env (id_1 ...) (remove-from-env id_0 Γ_0))])
```

The union-envs metafunction merely concatenates two environments:

```
(define-metafunction CPS-Calculus-Typed
  union-envs : Γ Γ -> Γ
  [(union-envs Γ_0 ·) Γ_0]
  [(union-envs Γ_0 (Γ_1 id_0 τ_0)) ((union-envs Γ_0 Γ_1) id_0 τ_0)])
```

The flatten-constraints takes the recursive definition of a constraint and returns it as a list. To do so, it takes in a nested constraint as the first argument and an accumulator (initially an empty list) as the second argument:

```
(define-metafunction CPS-Calculus-Typed
  flatten-constraints : constraints ((id τ) ...) -> ((id τ) ...)
  [(flatten-constraints · ((id_1 τ_1) ...)) ((id_1 τ_1) ...)]
  [(flatten-constraints (constraints_0 id_0 τ_0) ((id_1 τ_1) ...))
   (flatten-constraints constraints_0 ((id_0 τ_0) (id_1 τ_1) ...))])
```

The substitute-into-env uses the list of equations in solved form obtained from the unification to modify the environment's types:

```
(define-metafunction CPS-Calculus-Typed
  substitute-into-env : Γ ((id τ) ...) -> Γ
  [(substitute-into-env · ((id_1 τ_1) ...)) ·]
  [(substitute-into-env (Γ_0 id_0 τ_0) ((id_1 τ_1) ...))
   ((substitute-into-env Γ_0 ((id_1 τ_1) ...)) id_0
    (substitute τ_0 (id_1 τ_1) ...))])
```

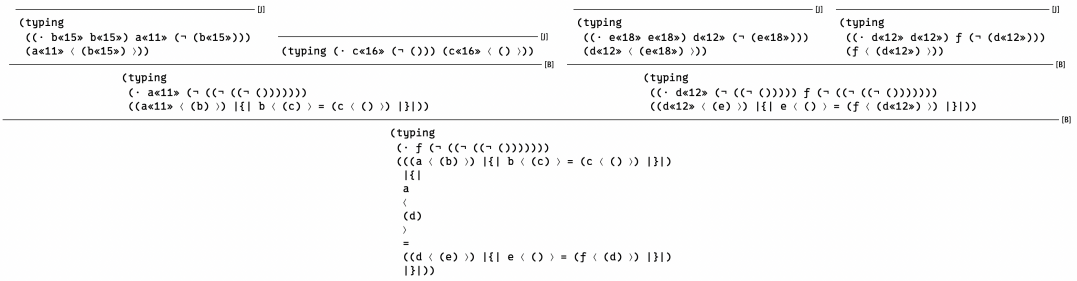
And finally, remove-duplicates-in-env recreates the environment, but only with unique ids. Comparing the type is not required, because if there were two names that could be substituted into two different types following the substitutions, the unification would have failed. This is the final metafunction:

```
(define-metafunction CPS-Calculus-Typed
  remove-duplicates-in-env : Γ (id ...) -> Γ
  [(remove-duplicates-in-env · (id_1 ...)) ·]
  [(remove-duplicates-in-env (Γ_0 id_0 τ_0) (id_1 ...))
   (remove-duplicates-in-env Γ_0 (id_1 ...))
   (side-condition (memq (term id_0) (term (id_1 ...))))]
  [(remove-duplicates-in-env (Γ_0 id_0 τ_0) (id_1 ...))
   ((remove-duplicates-in-env Γ_0 (id_0 id_1 ...)) id_0 τ_0)])
```

To give an example of the typing relation working, let's define a small term:

(((a ⟨ (b) ⟩) { b ⟨ (c) ⟩ = (c ⟨ () ⟩) }) { a ⟨ (d) ⟩ = ((d ⟨ (e) ⟩) { e ⟨ () ⟩ = (f ⟨ (d) ⟩) }) })

Here is its typing judgment:



3.4 Operational Semantics

In this section, the Jump Reduction, Garbage Collection Reduction and Full Reduction, as defined in Torrens, Orchard e Vasconcellos (2024) and explored in section 2.1.10.3, will be implemented. First, however, it's necessary to define one additional metafunction, one that does a step of capture-avoiding substitution on a CPS-calculus command for each parameter x in $args$ in a jump:

```
(define-metafunction CPS-Calculus
  jump-reduce : cmd (id ..._1) (id ..._1) -> cmd
  [(jump-reduce cmd_1 (id_x ...) (id_y ...))
   (substitute cmd_1 [id_x id_y] ...)])
```

Now it's possible to start implementing the reduction relations. The Jump Reduction is implemented in Redex as follows:

```
(define ->jr
  (compatible-closure
    (reduction-relation
     CPS-Calculus-Typed
     #:domain cmd
     #:codomain cmd
      (--> ((in-hole C_0 (id_0 ⟨ args_0 ⟩)))
           \{ id_0 ⟨ args_1 ⟩ = cmd_0 \})
           ((in-hole C_0 (jump-reduce cmd_0 args_1 args_0))
           \{ id_0 ⟨ args_1 ⟩ = cmd_0 \})
        "Jump␣Reduction"
        (side-condition (not (memq (term id_0)
                                   (term (dom C_0)))))
        (judgment-holds
          (typing Γ ((in-hole C_0 (id_0 ⟨ (id_1 ...) ⟩)))
            \{ id_0 ⟨ (id_2 ...) ⟩ = cmd_0 \}))))))
    CPS-Calculus-Typed cmd))
```

The added judgment-holds clause ensures that a CPS command will only be a part of this reduction's domain if it's well-typed. If that clause was absent, then the reduction could lead to stuck terms.

Now, the Garbage Collection Reduction is implemented as follows:

```
(define ->gc
  (compatible-closure
    (reduction-relation
```

```
    CPS-Calculus-Typed
    #:domain cmd
    #:codomain cmd
     (--> (cmd_0 \{ id_0 ⟨ args_0 ⟩ = cmd_1 \})
          cmd_0
          "Garbage␣Collection␣Reduction"
          (side-condition (not (memq (term id_0)
                                     (term (FV cmd_0)))))))
   CPS-Calculus-Typed cmd))
```

Lastly, the full reduction, which is just the union of the two rules above:

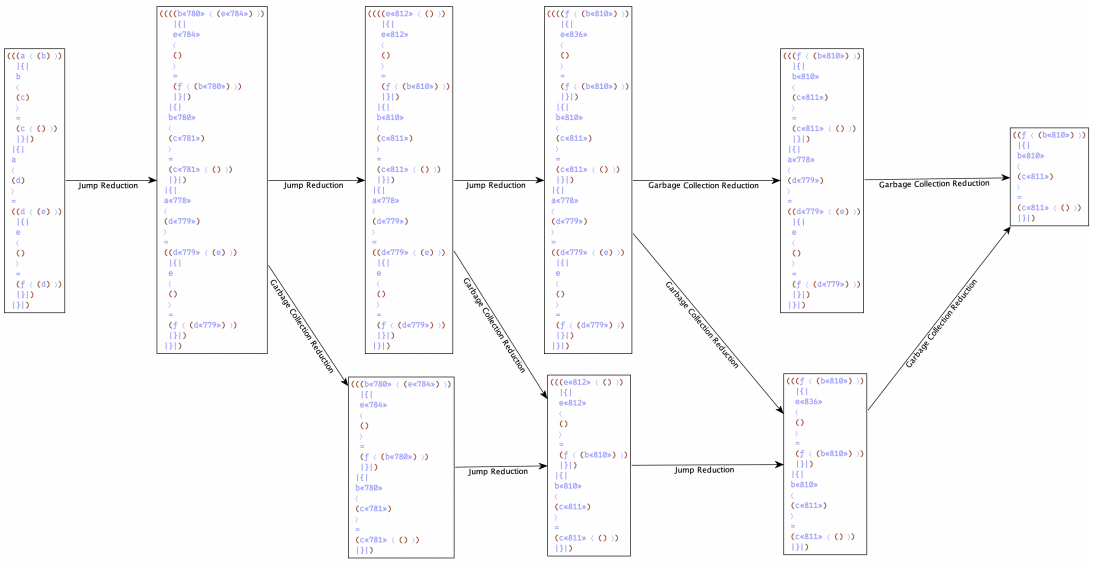
```
(define ->fr
  (union-reduction-relations ->jr ->gc))
```

These reductions rules can be applied in any order and anywhere, and as proved in Torrens, Orchard e Vasconcellos (2024), they will all lead to the same normal form, i.e., the Full Reduction is confluent.

A minor example showcasing this is the application of the Full Reduction on the following term, which is the same used when discussing the typing semantics:

$$(((a\ \langle\ (b)\ \rangle)\ \{\ b\ \langle\ (c)\ \rangle = (c\ \langle\ ()\ \rangle)\ \})\ \{\ a\ \langle\ (d)\ \rangle = ((d\ \langle\ (e)\ \rangle)\ \{\ e\ \langle\ ()\ \rangle = (f\ \langle\ (d)\ \rangle)\ \})\ \})$$

Using Redex's traces function (FELLEISEN; FINDLER; FLATT, 2009), we can obtain all possible reduction paths of this term under continuous applications of the full reduction, represented as a graph:



3.5 CPS-calculus Translations

To implement the CPS-calculus translations, both to and from λ-calculus, first it's required to extend the language to include λ-calculus expressions:

```
(define-extended-language
  CPS-Calculus-λ
```

```
CPS-Calculus-Typed
  (e ::= id                ; Variable
         (e e)             ; Application
         (λ id \. e))      ; Abstraction
  (val ::= (λ id \. e))    ; Abstraction value
  (Cλ ::= hole             ; Context
          (Cλ e)
          (val Cλ))
  (A ::= id (A -> A))      ; Functional types
  (λΓ ::= (λΓ e A))        ; Type Env
  #:binding-forms
  (λ id \. e #:refers-to id))
```

This extended language also includes values contexts, types and typing environments for the λ-calculus. The theory behind these transformations were explored in detail in section 2.1.10.5.

3.5.1 From CPS-calculus to λ-Calculus

In order to translate the polyadic CPS-calculus into a monoadic λ-calculus expression, it is first required to implement some metafunctions to nest bind parameters and jump arguments into the correct structure. This makes it so that arguments are supplied in the correct order and every abstraction is in curried form.

```
(define-metafunction CPS-Calculus-λ
  nest-args-jump-λ : id args -> e
  [(nest-args-jump-λ id_0 ()) id_0]
  [(nest-args-jump-λ id_0 (id_1 ... id_2))
   ((nest-args-jump-λ id_0 (id_1 ...)) id_2)])

(define-metafunction CPS-Calculus-λ
  curry-params-bind-λ : e args -> e
  [(curry-params-bind-λ e_0 ()) e_0]
  [(curry-params-bind-λ e_0 (id_0 id_1 ...))
   (λ id_0 \. (curry-params-bind-λ e_0 (id_1 ...)))])
```

And now the translation itself as defined by Thielecke (1997b) and also mapped as a metafunction:

```
(define-metafunction CPS-Calculus-λ
  CPS-to-λ : cmd -> e
  [(CPS-to-λ (id_0 ⟨ args_0 ⟩))
   (nest-args-jump-λ id_0 args_0)]
  [(CPS-to-λ (cmd_0 \{ id_0 ⟨ args_0 ⟩ = cmd_1 \}))
   ((λ id_0 \. (CPS-to-λ cmd_0))
     (curry-params-bind-λ (CPS-to-λ cmd_1) args_0))])
```

As a small example of it working, consider the following CPS-calculus term:

$$k\langle x, y\rangle \ \{m\langle n\rangle = n\langle\rangle\} \ \{k\langle a, b\rangle = a\langle b\rangle\}$$

Then, applying CPS-to-λ on that term results in the following expression:

$$((\lambda k.((\lambda m.((k\ x)\ y))(\lambda n.n)))(\lambda a.(\lambda b.(a\ b))))$$

By repeatedly performing β-reduction steps on this expression, the β-normal form calculated is going to be $(x\ y)$. This expression coincides with the $x\langle y\rangle$ that is obtained by repeatedly performing Full Reduction steps on the original CPS-calculus term.

3.5.2 From λ-Calculus to CPS-calculus

These translations were once again implemented as metafunctions. Starting with the untyped translations defined in Thielecke (1997a), the CBN and CBV implementations are as follows, first considering CBN:

```
(caching-enabled? #f)

(define-metafunction CPS-Calculus-λ
  λCBN-to-CPS : id e -> cmd
  [(λCBN-to-CPS (name k id) id_0)
   (id_0 ⟨ (k) ⟩)]
  [(λCBN-to-CPS (name k id) (λ id_0 \. e_0))
   ,(let ([next-k (gensym "k")] [v (gensym "v")])
      (term ((k ⟨ (,v) ⟩) \{ ,v ⟨ (id_0 ,next-k) ⟩ =
        (λCBN-to-CPS ,next-k e_0) \})))]
  [(λCBN-to-CPS (name k id) (e_0 e_1))
   ,(let ([left-k (gensym "k")]
          [right-k (gensym "k")]
          [v (gensym "v")]
          [f (gensym "f")])
      (term ((λCBN-to-CPS ,left-k e_0) \{ ,left-k ⟨ (,f) ⟩ =
        ((,f ⟨ (,v k) ⟩) \{ ,v ⟨ (,right-k) ⟩ =
          (λCBN-to-CPS ,right-k e_1) \}) \})))])
```

The (caching-enabled? #f) line is not mandatory, but encouraged for this implementation. Redex metafunctions by default caches the output of each seen arrange of inputs, but as identifiers generated by gensym are guaranteed to be distinct, all recursive calls will result in a cache miss. This caching option remains set for the CBV version of this transformation, which, by the way, is implemented as such:

```
(define-metafunction CPS-Calculus-λ
  λCBV-to-CPS : id e -> cmd
  [(λCBV-to-CPS (name k id) id_0)
   (k ⟨ (id_0) ⟩)]
  [(λCBV-to-CPS (name k id) (λ id_0 \. e_0))
   ,(let ([next-k (gensym "k")]
          [v (gensym "v")])
      (term ((k ⟨ (,v) ⟩) \{ ,v ⟨ (id_0 ,next-k) ⟩ =
        (λCBV-to-CPS ,next-k e_0) \})))]
  [(λCBV-to-CPS (name k id) (e_0 e_1))
   ,(let ([left-k (gensym "k")]
          [right-k (gensym "k")]
          [v (gensym "v")]
          [f (gensym "f")])
      (term ((λCBV-to-CPS ,left-k e_0) \{ ,left-k ⟨ (,f) ⟩ =
        ((λCBV-to-CPS ,right-k e_1) \{ ,right-k ⟨ (,v) ⟩ =
          (,f ⟨ (,v k) ⟩) \}) \})))])
```

If taking into account α-convertibility, it's not required to name the f and v variables as something distinct at all (TORRENS; ORCHARD; VASCONCELLOS, 2024). However, it was decided that making them all fresh is the better approach in this instance, both because it may help disambiguate the names when the output is being read by a human, and also because this way there is no need to rely on α-convertibility, if that ever turns out to be relevant.

As a minimal example, consider the following λ-calculus expression: $((\lambda x.x)\ (\lambda y.z))$

After running λCBN-to-CPS on it and supplying the symbol k at the top-level, the following command is returned[1]:

$$\begin{aligned}&\mathrm{k1}\langle \mathrm{v1}\rangle\{\mathrm{v1}\langle \mathrm{x}, \mathrm{k2}\rangle = \mathrm{x}\langle \mathrm{k2}\rangle\}\\ &\quad\{\mathrm{k1}\langle \mathrm{f1}\rangle = \mathrm{f1}\langle \mathrm{v2}, \mathrm{k}\rangle\\ &\qquad\{\mathrm{v2}\langle \mathrm{k3}\rangle = \mathrm{k3}\langle \mathrm{v3}\rangle\{\mathrm{v3}\langle \mathrm{y}, \mathrm{k4}\rangle = \mathrm{z}\langle \mathrm{k4}\rangle\}\}\}\end{aligned}$$

And after doing the same with λCBV-to-CPS:

$$\begin{aligned}&\mathrm{k1}\langle \mathrm{v1}\rangle\{\mathrm{v1}\langle \mathrm{x}, \mathrm{k2}\rangle = \mathrm{k2}\langle \mathrm{x}\rangle\}\\ &\quad\{\mathrm{k1}\langle \mathrm{f1}\rangle = \mathrm{k3}\langle \mathrm{v2}\rangle\\ &\qquad\{\mathrm{v2}\langle \mathrm{y}, \mathrm{k4}\rangle = \mathrm{k4}\langle \mathrm{z}\rangle\}\{\mathrm{k3}\langle \mathrm{v3}\rangle = \mathrm{f1}\langle \mathrm{v3}, \mathrm{k}\rangle\}\}\end{aligned}$$

And now moving on to the typed translations. First, the translations of the actual types, from a λ-calculus' functional type A to a CPS-calculus' negation type τ, also differentiating between CBN and CBV:

```
(define-metafunction CPS-Calculus-λ
  λCBN-to-CPS-type : A -> τ
  [(λCBN-to-CPS-type id_0) (¬ (id_0))]
  [(λCBN-to-CPS-type (A_0 -> A_1))
   (¬ ((¬ ((¬ ((λCBN-to-CPS-type A_0)))
    (λCBN-to-CPS-type A_1)))))])

(define-metafunction CPS-Calculus-λ
  λCBV-to-CPS-type : A -> τ
  [(λCBV-to-CPS-type id_0) id_0]
  [(λCBV-to-CPS-type (A_0 -> A_1))
   (¬ ((λCBV-to-CPS-type A_0)
    (¬ ((λCBV-to-CPS-type A_1)))))])
```

The rule that translated an entire typing statement in λ-calculus to a typing statement in CPS-calculus will be given in chapter 5.

3.6 Conclusion

In this chapter, the CPS-calculus was mechanized using PLT Redex, with support for operational semantics, typing semantics, and translations to-and-from λ-calculus.

[1] The variable names were changed, because gensym appends each created symbol with 7 digits and that made the original command too big to typeset.

With just this, it's already possible to start testing properties by using Redex's own term generation functionalities. It's possible, for example, to generate commands that satisfy the typing judgment for the CPS-Calculus. However, the generalist nature of this method results in a set of terms that lack diversity, as most of the generated commands are somewhat trivial.

To make meaningful usage of the mechanization presented in this chapter, a specialized generator of CPS-Calculus commands was devised. This generator is to be formalized in the next chapter.

4 CPS-calculus Generator

The CPS-calculus terms generator was designed with the intent of balancing two competing characteristics: diversity and usefulness. When it comes to diversity, the generator's image should include as many different command patterns as possible, as to maximize coverage when using it to test a property. However, and now about usefulness, it should also minimize patterns that don't contribute much to the operational semantics, for example defining continuations that are never used, even if avoiding those types of terms may reduce the image's overall diversity.

An implementation of the algorithm formalized in this chapter was created using Rackcheck, a Racket library. The output should be a term immediately understandable by the CPS-calculus Redex implementation explored in the previous chapter.

4.1 Overview

As previously mentioned, CPS-calculus commands don't have types. Instead, the typing relation defined in Thielecke (1997a) and explored in this work in section 2.1.10.4 is only applicable to the command's variables (TORRENS; ORCHARD; VASCONCELLOS, 2024). This makes creating trivial well-typed commands relatively easy at first glance, as any command that doesn't utilize any of its parameters or defined continuations and instead purely relies on free variables will be well-typed. However, that is neither diverse nor useful.

The generator developed here tries to maximize the proportion of parameters used, and it also guarantees that every continuation defined is used at least once, as they are defined on demand. Also, every variable has its full type saved in the variables' environment during the generation, and the algorithm makes a light attempt to use these variables in such a way that the principal types obtained through the inference judgment implemented in section 3.3 are as close to the original full types as possible. However, this is not a best possible attempt, and as such this is a feature that is left to be further explored in future works.

The command generation is guided by random walks performed on a sampled finite type environment. This is inspired by type-directed generation procedures, like what was done in Feitosa, Ribeiro e Bois (2019) and Graeff et al. (2024). It's not possible to replicate these type-directed ideas as they are because the type in question is the term's, but CPS-calculus commands don't have types.

The alternative to this was to instead guide the generation by variable types, or in other words, to create a context-directed generation procedure. The way this was done was by sometimes using a newly added parameter's type as the root of a new random walk along a finite type environment, such that the current type can move to either one of its super-types or one of its sub-

types. This walk can then be processed, starting from where it ended and going backwards until the initial step is reached. Each step on the walk is representative of a jump, and the parameter that created this walk will be used on the last jump as one of its variables, either in subject position or in object position. This approach can be seen as instantiating, for every parameter that originates a new walk, a sequence of prerequisite jumps that must be done before that parameter's guaranteed use is invoked, essentially delaying its main usage by a minimum of w jumps, where w is the number of steps in the walk.

The reason walks were chosen instead of allowing steps to be taken from any type to any other type is twofold: first, by keeping adjacent jumps' variable types somewhat close together on the type environment, more binds and parameters of relevant types are going to be added to the variables environment, which can increase the chances of them being re-utilized even for small commands, which can greatly increase diversity. Second, and related to the first point, after a certain number of steps have been taken, further walks will only move to sub-types in an attempt to end the process at a base type, specializing the inferable variables' types as much as possible. However, this process can't guarantee that the least general types will be able to get calculated, specially given this is not even a best effort at that, and as such, inducing a denser web of type dependencies by more frequent re-utilization of variables is desirable from the standpoint of getting the inferable types to be as close as possible to the full types used in the generation process.

In the following sections, the algorithm is going to be formalized. An implementation based on this formalization was done in Racket using the Rackcheck library. Due to it's size, this implementation will be omitted from the main text, but it can be publicly found on this project's GitHub repository[1].

4.2 Used Notation

Most operations will be defined as big-step relations, with input and output channels being separated by a $\vdash$ symbol containing the relation name sub-scripted, like in $\vdash_{CPS}$. Inputs and outputs will be separated by a semicolon.

Some very simple and pure functions will be defined as a Haskell-like pseudo-code, instead of as big-step relations. These functions allow partial application, so if a function that receives two parameters is provided with only one, it returns a function that receives the remaining parameter and uses the first parameter as a constant.

Standard functions will be used without a proper definition, like the reverse operation for lists and some higher-order functions, like *map*, *filter*, *fold*, among others. These higher-order functions may be used both on lists or sets indiscriminately. A limitation on that front is that whenever the *fold* function is applied on a set, the order of processed elements should not impact

[1] https://github.com/bermondd/cps-calculus-generator

the final accumulator. Also, *filter-keys* and *filter-values* can be used to filter an associative list based on its key or value, respectively. The *partition* function gets a predicate and a list or set and it returns a tuple where the first element is the result of applying filter with that predicate and the second element is the result of applying filter with the negation of that predicate. The *fold* function gets a function as its second parameter, an initial accumulator as its third and one or more lists starting at its fourth, but the function provided has the accumulator as the last parameter.

The standard function *cons* will be written with the Haskell syntax, using the infix function $\_ : \_$, which gets an element of type A on the left and a list of type $[A]$ on the right. List elements will always be separated using the colon symbol. Tuple values, on the other hand, are separated by commas.

The conditional (if $cond$ then $branch_1$ else $branch_2$) will be written as ($cond$? $branch_1$: $branch_2$), to save space. Also, a common dot can be used to represent function composition and a (let *assignment* in *expression*) expression can be used to create a local binding for quick access to tuple elements in lambda functions.

The Greek letter ξ is going to be used to abstract randomness in several different ways, like what was done in Feitosa, Ribeiro e Bois (2019). The ways this symbol is used is located below:

- ξ :: Foldable $t \Rightarrow t\ a \rightarrow a$: If used as a function on a collection of elements, it returns a random element from the provided collection. This function is used with parentheses, like $\xi(L)$.
- ξ :: Foldable $t \Rightarrow t\ a \rightarrow \{n \in \mathbb{N} \mid n \leq |t|\} \rightarrow t\ a$: If alongside the foldable a natural number n is also provided, then it returns n distinct random elements from the original collection as a new one of the same type. If it also has an order (like lists), the order of the returned elements are random. Natural numbers that are larger than the size of the collection are considered outside the domain of this function. This function is also used with parentheses.
- $\xi_{shuffle} :: [a] \rightarrow [a]$: Randomly shuffles the list given as a parameter. Used with parentheses.
- $\xi_{id} :: String$: Returns a fresh unique identifier, never seen before anywhere else in the command. For this function, the parentheses are omitted.

The meaning behind all further Greek letters will be explained as they are introduced.

Three different equality symbols are employed, and two of them mean similar but different things. The first is $lhs := rhs$, which creates a lazy alias for the operation described in the rhs to the name lhs. For that to be usable, the rhs needs to either be a pure operation or be used only once, as the lhs can be seen as a macro that is replaced with its expression in all locations in which it's used. The $:=$ is the preferred symbol whenever applicable.

The second is $lhs = rhs$, which is an eager assignment of the value returned by the rhs operation to the name lhs. This is used exclusively when randomness is involved and the same randomly sampled value must be used in two or more places, as to guarantee that the output of a ξ function is the same everywhere.

The third type of equality is $lhs \equiv rhs$, which is an equivalence comparison operator. The symbol $lhs \not\equiv rhs$ is also valid and evaluates to true iff lhs and rhs are different.

All of these equality symbols support pattern matching, which can be used to bind values through tuple or list unpacking. The symbol $\not\equiv$ may be used to indicate a condition that a variable does **not** match with a certain pattern. The underscore _ can be used in pattern matching locations to signal that a value is present there, but that it does not matter, so it can be discarded.

In judgments, lambda functions will be used extensively, due to space constraints. The syntax used is $(I \to O)$, for example $(x \to x + 2)$. There can be multiple inputs, but only one output, and multiple inputs are separated by a semicolon.

Sets with restrictions can be represented in multiple ways. The explicit form is like this, for example: $\{x \in \mathbb{R} \mid x \leq 10\}$. But two shorthands are also used: $[a..{=}b]$ means the same as $\{n \in \mathbb{N} \mid a \leq n \leq b\}$, $[a..b]$ means $\{n \in \mathbb{N} \mid a \leq n < b\}$, and $[a, b]$ means $\{x \in \mathbb{R} \mid a \leq x \leq b\}$, with parentheses instead of square brackets being used to signal open intervals.

Due to the prevalence of a specific set, consider for the remainder of this work that $\mathbb{R}^u$ is the subset of real numbers within the closed unit interval, that is, $\mathbb{R}^u := \{x \in \mathbb{R} \mid 0 \leq x \leq 1\}$. Also, as is standard in the literature, $\mathbb{N}^+$ means the set of non-zero natural numbers.

Finally, for brevity sake, rules for unreachable states will be omitted. Getting to such a state would mean that either this formalization or one of its implementations is incorrect.

4.3 Term Generator ($\vdash_{CPS}$)

This is the judgment called to generate a command. It receives 16 numerical parameters, all of them natural numbers and grouped together with the symbol ψ here due to space constraints, and a set of sets of base types. The 16 numerical parameters are:

1. MNST ($\mathbb{N}^+$): Upper bound for the maximum number of arguments a continuation can have. In terms of CPS types, this then indicates the Maximum number of Sub-Types.
2. MBSF ($\mathbb{N}^+$): Upper bound for the Maximum Batch Size of types for the First execution of $\vdash_{type-env}$.
3. MTSF ($\mathbb{N}^+$): Upper bound for the Maximum Type Size for the First execution of $\vdash_{type-env}$.
4. MBSR ($\mathbb{N}^+$): Upper bound for the Maximum Batch Size of types for Recursive executions of $\vdash_{type-env}$.

5. MTSR ($\mathbb{N}^+$): Upper bound for the Maximum Type Size for the Recursive executions of $\vdash_{type-env}$.
6. MTRB ($\mathbb{N}^+$): Upper bound for the maximum number of new base types the recursive steps of $\vdash_{type-env}$ will sample. The name stands for Maximum Type Recursion Bases.
7. MTRS ($\mathbb{N}$): Upper bound for the maximum number of recursive steps allowed for $\vdash_{type-env}$. The name stands for Maximum Type Recursion Steps.
8. MWLM ($\mathbb{N}^+$): Upper bound for the maximum main walk length, to be used in $\vdash_{cmd}$ right before a call to $\vdash_{walk}$.
9. BWLM ($\mathbb{N}^+$): Upper bound for the maximum branched walk length, to be used in $\vdash_{walk}$ before a recursive call.
10. FLMD ($\mathbb{N}$): Upper bound for the maximum number of fold dependencies in $\vdash_{walk}$.
11. BRMD ($\mathbb{N}$): Upper bound for the maximum number of borrow dependencies in $\vdash_{walk}$.
12. LNMD ($\mathbb{N}$): Upper bound for the maximum number of lend dependencies in $\vdash_{walk}$.
13. MCDA ($\mathbb{N}$): Upper bound for the maximum number of dependencies allowed in a component. Used in $\vdash_{walk}$.
14. MCWA ($\mathbb{N}^+$): Upper bound for the maximum number of connected walks in a single component. Used in $\vdash_{walk}$.
15. MD ($\mathbb{N}$): Upper bound for the maximum depth of branching recursive calls, which indicates the maximum recursive depth of $\vdash_{walk}$.
16. MSBR ($\mathbb{N}$): Upper bound for the maximum walk length accumulator.

And the judgment returns a CPS command. All of these numerical parameters will be inputted to $\vdash_{params}$ to create a set of parameters Ψ that is valid only for this generation instance and that will be resampled for every subsequent command.

The $\vdash_{CPS}$ judgment only has one rule, defined below:

$$\frac{\begin{array}{c}\text{b} = \xi(\text{cases}) \cup \{\neg()\} \\ \psi \vdash_{params} \Psi \\ \Psi \text{ ; b ; } \{\emptyset\} \text{ ; } \Psi(\text{MTRS}) \vdash_{type-env} \Theta \\ \text{hd-nf?} := \xi(\{\top, \bot\}) \text{ ? halt : } \cdot \\ \text{root-typ} := \xi(\text{weighted-repeat-}\tau\ ((\Theta \setminus \text{b}) \cup \{\neg()\})) \\ \Psi \text{ ; } \Theta \text{ ; } \{\emptyset\} \text{ ; b ; hd-nf? ; root-typ ; } 0 \text{ ; } 0 \text{ ; } 1 \vdash_{walk} \Delta \text{ ; \_} \\ \Psi \text{ ; } \Theta \text{ ; } \Delta \text{ ; } \{\emptyset\} \text{ ; } \Psi(\text{MSBR}) \vdash_{cmd} \text{cmd ; \_}\end{array}}{\text{bases ; } \psi \vdash_{CPS} \text{cmd}}$$

Where weighted-repeat-τ is defined as follows, following a functional Haskell-like pseudo-code:

$$
\begin{aligned}
&\textbf{weighted-repeat-}\tau :: [\tau] \to [\tau] \\
&\text{weighted-repeat-}\tau \quad nil \;=\; nil \\
&\text{weighted-repeat-}\tau \quad (\tau : \vec{\tau}) \\
&\qquad\qquad = \text{ append (repeat } \tau \text{ (get-type-size } \tau)) \text{ (weighted-repeat-}\tau\ \vec{\tau}) \\
&\qquad\qquad\quad \text{where} \\
&\quad \textbf{get-type-size} :: \tau \to \mathbb{N} \\
&\quad \text{get-type-size} \quad \neg\vec{\tau} \;=\; \text{fold } (\tau \text{ ; acc} \to \text{acc} + (\text{get-type-size } \tau)) \ 1\ \vec{\tau} \\
&\quad \text{get-type-size} \quad X \;=\; 1
\end{aligned}
$$

Below is an explanation of these steps, line by line:

1. Sample a set of base types, and add $\neg()$ to it.
2. Generate several random parameters with $\vdash_{params}$, to improve diversity across multiple runs.
3. Generate a finite type environment with $\vdash_{type-env}$. This environment is represented as two graphs: a directed acyclic graph and its transpose[2]. This generation is recursive, and the number of recursive steps is determined by the parameter $\Psi(\text{MTRS})$.
4. Generate a boolean. If true, create a symbol called *halt*. If false, create $\cdot$. This symbol is the name given for the root variable of the walk done in the next step.
5. Select a non-base starting type from the type environment, with $\neg()$ as an option. This sampling is weighted on the size of the type, which is expressed by a repetition of that type in the list being sampled from. For example, $\neg()$ will be in that list only once, but $\neg(\neg(), \neg())$ will be there three times.
6. With the sampled type, perform a one step random walk to one of its sub-types[3] with $\vdash_{walk}$. This small walk is going to initialize a walk environment. If hd-nf? is halt, then the command generated is guaranteed to be in head-normal form.
7. Start processing the walk environment with $\vdash_{cmd}$, sending in the steps accumulator as determined by $\Psi(\text{MSBR})$. At first, the walk environment is only going to have one step at most, representing one jump. As this is the start of the algorithm, no binds have been done yet, so the variables environment is empty. Unless the initial type is $\neg()$, there will be at least one bind with at least one parameter. Each of these parameters can instantiate a

[2] The transpose of a directed graph is that same graph but with the direction of its edges reversed

[3] The only case where no sub-types exist is if this initial type is $\neg()$, and in that case it's a walk with no steps

new walk with its root set as the parameter's type, and these walks are now able to move not only to the parameter's sub-types but also to the super-types in the type environment. The $\vdash_{cmd}$ judgment is recursive, such that its first output will already be the full command, which is then returned.

The last judgment will keep on processing the walks and creating new walks until all of the available steps (determined by $\Psi(\text{MSBR})$ and called by default in the rest of this formalization as w_{acc}) are spent. At that point, only going to sub-types is allowed, ending at a base type. Whenever a new walk with w steps is performed, w_{acc} gets decremented by w, and at the end this guarantees termination and somewhat controls the size of the command being generated.

4.4 Parameter Generator ($\vdash_{params}$)

This function's purpose is to process the provided general parameters into new command-specific parameters. It receives the ψ described previously, and returns an associative list Ψ, mapping names to numbers.

The numerical values provided to $\vdash_{CPS}$ and grouped into ψ indicate a maximum possible value for the metric they represent. For example, the first parameter indicates the maximum number of arguments a continuation can have, and it acts across **all** terms generated with that call. However, to increase diversity, it may be useful to make it so that a certain command has its maximum number of arguments set to a different, smaller number, which changes with every call. Then, this first parameter can be seen as a ceiling for the generated maximums, increasing diversity while still leaving the generation bounded.

The $\vdash_{params}$ judgment performs a sampling using the generator arguments as a maximum. It takes the same ψ arguments as the top-level $\vdash_{CPS}$ judgment. This judgment returns an associative list Ψ, which maps a name to a numerical parameter that is valid for the current command generation. Subsequent generations will sample Ψ again, leading to greater diversity of outputs. It only has one rule, defined below:

$$
\begin{aligned}
\Psi(\text{MNST}) &= \xi([1..=\text{MNST}]) \\
\Psi(\text{MBSF}) &= \xi([1..=\text{MBSF}]) \\
\Psi(\text{MTSF}) &= \xi([1..=\text{MTSF}]) \\
\Psi(\text{MBSR}) &= \xi([1..=\text{MBSR}]) \\
\Psi(\text{MTSR}) &= \xi([1..=\text{MTSR}]) \\
\Psi(\text{MTRB}) &= \xi([1..=\text{MTRB}]) \\
\Psi(\text{MTRS}) &= \xi(\{0,1\}) * \xi([0..=\text{MTRS}]) \\
\Psi(\text{EST}) &= \xi(\mathbb{R}^u) \\
\Psi(\text{MD}) &= \xi([0..=\text{MD}]) \\
\Psi(\text{NBTT}) &= \xi(\mathbb{R}^u)^2 \\
\Psi(\text{MGT}) &= \xi(\mathbb{R}^u)^2 \\
\Psi(\text{BNT}) &= \xi(\mathbb{R}^u)^{\frac{1}{4}} \\
\Psi(\text{BWLM}) &= \xi([1..=\text{BWLM}]) \\
\Psi(\text{VIT}) &= \xi(1, \xi(\mathbb{R}^u)^{\frac{1}{8}}, \xi(\mathbb{R}^u)^{\frac{1}{4}}) \\
\Psi(\text{FLT}) &= \xi(\mathbb{R}^u)^{\frac{1}{2}} \\
\Psi(\text{BRT}) &= \xi(\mathbb{R}^u)^{\frac{1}{4}} \\
\Psi(\text{LNT}) &= \xi(\mathbb{R}^u)^{\frac{1}{4}} \\
\Psi(\text{FLMD}) &= \xi([0..=\text{FLMD}]) \\
\Psi(\text{BRMD}) &= \xi([0..=\text{BRMD}]) \\
\Psi(\text{LNMD}) &= \xi([0..=\text{LNMD}]) \\
\Psi(\text{MDBA}) &= \xi([0..=\text{MCDA}]) \\
\Psi(\text{MDOA}) &= \xi([0..=\text{MCDA}]) \\
\Psi(\text{MCWA}) &= \xi([1..=\text{MCWA}]) \\
\Psi(\text{ADTT}) &= \xi(\{-1, 1, \xi(\mathbb{R}^u)\}) \\
\Psi(\text{LTWT}) &= \xi(\{-1, 1, \xi(\mathbb{R}^u)\}) \\
\Psi(\text{MWLM}) &= \xi([1..=\text{MWLM}]) \\
\Psi(\text{MBLT}) &= \xi(\mathbb{R}^u) \\
\Psi(\text{WDB}) &= \xi(\mathbb{R}^u) \\
\Psi(\text{FRT}) &= \xi(\mathbb{R}^u) \\
\Psi(\text{FB}) &= \xi([0..=10]) \\
\Psi(\text{BB}) &= \xi([0..=10]) \\
\Psi(\text{CSW}) &= \xi([0..=5]) \\
\Psi(\text{WOCB}) &= 2 * \xi(\mathbb{R}^u) \\
\Psi(\text{MSBR}) &= \xi([0..=\text{MSBR}]) \\
\hline
\psi \vdash_{params} \Psi &
\end{aligned}
$$

The exponents after some of the real-valued samples are there to provide bias across generations. First consider $\Psi(\text{FRT})$, standing for Float Right Threshold: the lower the value, the more likely it is for the Float Right operation (defined in $\vdash_{cmd}$) to be executed. With no exponents, the distribution of thresholds is uniform, so having a very high threshold of T ≥ 0.9 (which results in only 10% chance for the float right operation to be executed when prompted, for that

specific generation) is exactly as likely as having a very low threshold of T ≤ 0.1 (which results in a probability of 90%).

Sometimes, however, it's desirable for some operations to have a high probability of having a higher or lower threshold across the generations. Consider $\Psi(\text{MGT})$, which stands for Merge Threshold: it's desirable for the Merge operation (defined in $\vdash_{walk}$) to be used more often than not in most programs. By squaring its generated value, we skew the distribution of the threshold to make it more likely for it to result in smaller numbers. Specifically, only 50% of programs will have MT valued over 0.25, and already low thresholds will get severely reinforced in that direction. For example, a sample of T ≤ 0.1, which happens with a probability of 10%, ends up with a threshold of 0.01, which turns a 90% chance of Merging into a 99%. On the other hand, the image of the generation itself remains unchanged, as it's still possible to sample a very high number, and in which case the squaring has minimal impact. For example, a sample of T ≤ 0.99 still leads, after squaring, to a threshold of 0.9801, a difference of less than 1%.

A list of all these parameters can be found below. It's meant to be used for future reference, as it references parts of the generation that have not yet been explained:

- MNST: Maximum number of arguments in a continuation. From a CPS types perspective, this represents the Maximum number of Sub-Types. Used in $\vdash_{type-env}$.
- MBSF: Maximum Batch Size of types for the first $\vdash_{type-env}$ step.
- MTSF: Maximum Type Size for the first $\vdash_{type-env}$ step.
- MBSR: Maximum Batch Size of types for the recursive $\vdash_{type-env}$ steps.
- MTSR: Maximum Type Size for the recursive $\vdash_{type-env}$ steps.
- MTRB: Maximum number of new base types used in recursive calls to $\vdash_{type-env}$. This stands for Maximum Type Recursive Base.
- MTRS: Number of possible recursive steps of $\vdash_{type-env}$.
- EST: Threshold that needs to be met in order to allow for a maximum number of continuation arguments greater than 4. The name stands for Extended Subtypes Threshold. Used in $\vdash_{type-env}$.
- MD: Maximum Depth of branching allowed. Used in $\vdash_{cmd}$.
- NBTT: No BackTracking Threshold. When met and if possible, removes the backtracking vertex as an option in the random walk phase of $\vdash_{walk}$.
- MGT: Threshold that needs to be met in order to execute the MerGe operation of $\vdash_{walk}$.
- BNT: Threshold that needs to be met in order to execute the BraNch operation of $\vdash_{walk}$.

- BWLM: Maximum branched walk length. Used in $\vdash_{walk}$.
- VIT: Threshold that needs to be met in order to execute the VoId operation of $\vdash_{walk}$.
- FLT: Threshold that needs to be met in order to execute the FoLd operation of $\vdash_{walk}$.
- BRT: Threshold that needs to be met in order to execute the BorRow operation of $\vdash_{walk}$.
- LNT: Threshold that needs to be met in order to execute the LeNd operation of $\vdash_{walk}$.
- FLMD: Maximum number of FoLd Dependencies that can be created in one step. Used in $\vdash_{walk}$.
- BRMD: Maximum number of BorRow Dependencies that can be created in one step. Used in $\vdash_{walk}$.
- LNMD: Maximum number of LeNd Dependencies that can be created in one step. Used in $\vdash_{walk}$.
- MDBA: Maximum number of Depended-By dependencies Allowed in a component at any given time. Used in $\vdash_{walk}$.
- MDOA: Maximum number of Depends-On dependencies Allowed in a component at any given time. Used in $\vdash_{walk}$.
- MCWA: Maximum number of Connected Walks Allowed together. Used in $\vdash_{walk}$.
- ADTT: Allow Different Types Threshold. Determines the threshold that needs to be beat in order to lift the equality of types as a filter criteria for extrinsic dependencies. Used in $\vdash_{walk}$.
- LTWT: Label Tip of Walk Threshold. Determines the threshold that needs to be beat in order for the end vertex of walks to be labeled with a special symbol that prevents one of its components from being considered a side component. This results in that component not originating branches and not being the object of collides. Used in $\vdash_{walk}$.
- MWLM: Main Walk Length Max. Used in $\vdash_{cmd}$.
- MBLT: Threshold that needs to be met in order to move a base type to the left in the component filling order. Used in $\vdash_{cmd}$.
- WDB: Walk distribution bias. The higher this value, the higher the weight the number of unprocessed nodes has in the distribution of the remaining walk steps, and the lower the weight of randoms scores. Used in $\vdash_{cmd}$.
- FRT: Threshold that needs to be met in order to execute the Fold Right operation. Used in $\vdash_{cmd}$.

- FB: Fetch bias. The higher this value, the more likely it is for an already defined continuation or parameter to be fetched from the environment and used. Used in $\vdash_{cmd}$.
- BB: Bind bias. The higher this value, the more likely it is to create a new bind to be used in a jump, even if a variable with the same type is present in the environment. Used in $\vdash_{cmd}$.
- CSW: Collide on the Same Walk bias. The higher this is, the more likely it is to perform a collide on a component whose node originated from the same walk as the current argument. Used in $\vdash_{cmd}$.
- WOCB: Walk Over Collide Bias. The higher this value, the more likely it is for a new argument to originate a new walk, rather than a collide. Used in $\vdash_{cmd}$.
- MSBR: Steps before returning. This value is inputted as a parameter to several judgments, and it is continuously reduced until it becomes a non-positive number, at which point further walks will only allow backward steps. This is the parameter that indicates when to start the termination phase of the generation.

4.5 Type Environment Generator ($\vdash_{type-env}$)

There are infinite CPS-calculus types, as each type is unbounded both on its width (how many sub-types it can have) and depth (how many layers of recursion its sub-types can have). Before a command is generated, a finite typing environment is first created.

The $\vdash_{type-env}$ judgment receives the following as input:

- Ψ: Associative list containing the parameters for that generation.
- first?: A boolean indicating whether this is the first step or if it's a recursive step.
- b: Set of base types.
- θ: An accumulator of types.
- x: Natural number controlling the recursion.

It outputs an associative list Θ, which maps a type to a 2-tuple whose values contain a set of types each, one representing the key's super-types in the environment and the other representing its sub-types. This output can be viewed as two directed acyclic graph (DAG), where the first element in the outer tuple is the source vertex of an edge and the two sets in the second tuple element are the source's adjacency's lists (or in this case, sets), each corresponding to a different graph. Importantly, these two graphs are actually the transpose of each other.

This judgment has two rules. Starting with the base case, it happens when $x \equiv 0$ and calls the $\vdash_{type-list}$ judgment, as does the recursive case, and returns the type environment after turning it into two graphs by calling to-graph-vertex. Its rule is defined below:

$$\frac{\Psi\ ;\ \text{first?}\ ;\ \text{b}\ ;\ \xi([1..=(\text{first?}\ ?\ \Psi(\text{MBSF}) : \Psi(\text{MBSR}))])\ \vdash_{type-list}\ \vec{\tau} \qquad \theta' := \text{fold}\ (\tau\ ;\ acc\ \rightarrow\ acc\ \cup\ (\text{explode}\ \tau))\ \theta\ \vec{\tau}}{\Psi\ ;\ \text{first?}\ ;\ \text{b}\ ;\ \theta\ ;\ zero\ \vdash_{type-env}\ \text{map}\ (\text{to-graph-vertex}\ \theta')\ \theta'}$$

And now the recursive case. It starts like the base case, generating a list of types and then calling Explode on it to join with the current accumulator. After that, however, it creates a new set of base types by sampling from the accumulator, and then does a recursive call with what was sampled. This has the effect of turning the new base types into hub nodes in the output graph, that is, nodes with a much higher number of connections than average. This case's rule is defined below:

$$\frac{\begin{array}{c}\Psi\ ;\ \text{first?}\ ;\ \text{b}\ ;\ \xi([1..=(\text{first?}\ ?\ \Psi(\text{MBSF}) : \Psi(\text{MBSR}))])\ \vdash_{type-list}\ \vec{\tau} \\ \theta' := \text{fold}\ (\tau\ ;\ \text{acc}\ \rightarrow\ \text{acc}\ \cup\ (\text{explode}\ \tau))\ \theta\ \vec{\tau} \\ \text{b}' = \xi(\theta',\ \xi([1..=\Psi(\text{MTRB})])) \\ \Psi\ ;\ \text{b}'\ ;\ \theta'\ ;\ \text{x}\ \vdash_{type-env}\ \Theta\end{array}}{\Psi\ ;\ \text{first?}\ ;\ \text{b}\ ;\ \theta\ ;\ succ(\text{x})\ \vdash_{type-env}\ \Theta}$$

4.5.1 Auxiliary functions

First the function explode. It gets a type as its parameter and return a set containing that type, all of its sub-types, all of its sub-types' sub-types, etc.

$$\begin{aligned}&\textbf{explode}\ ::\ \tau \rightarrow \{\tau\} \\ &\text{explode}\ \neg\vec{\tau}\ =\ \text{fold}\ (\tau\ ;\ \text{acc}\ \rightarrow\ \text{acc}\ \cup\ (\text{explode}\ \tau))\ \{\neg\vec{\tau}\}\ \vec{\tau} \\ &\text{explode}\ \ X\ =\ \{X\}\end{aligned}$$

And now the function to-graph-vertex. It receives a set of types θ and a type τ, and it returns a 2-tuple consisting of the given type τ as its first element and another 2-tuple as its second element. This second 2-tuple has the set of the τ's super-types in θ as its first element and the set of τ's sub-types in θ as its second element.

$$\begin{aligned}&\textbf{to-graph-vertex}\ ::\ \{\tau\}\ \rightarrow\ \tau\ \rightarrow\ (\tau,\ (\{\tau\},\ \{\tau\})) \\ &\text{to-graph-vertex}\ \Theta\ \neg\vec{\tau}\ =\ (\neg\vec{\tau},\ (\text{filter}\ (\tau'\ \rightarrow\ (\tau' \equiv \neg\vec{\tau}')\ \wedge\ (\neg\vec{\tau} \in \vec{\tau}'))\ \Theta,\ \text{toSet}\ \vec{\tau})) \\ &\text{to-graph-vertex}\ \Theta\ \ X\ =\ (X,\ (\text{filter}\ (\tau'\ \rightarrow\ (\tau' \equiv \neg\vec{\tau}')\ \wedge\ (X \in \vec{\tau}'))\ \Theta,\ \{\emptyset\}))\end{aligned}$$

4.5.2 List of Types ($\vdash_{type-list}$)

This judgment creates, as the name suggests, a list of CPS-calculus types. It takes as parameters:

- Ψ: Associative list of parameters.
- first?: A boolean indicating whether this is being called from the first step of $\vdash_{type-env}$ or if it's from a recursive step.

- b: Set of base types.
- x: Natural number indicating how many more types to create. This parameter controls the recursion.

This judgment outputs a list of CPS-calculus types.

The base case simply returns nil when $x \equiv zero$:

$$\frac{}{\Psi\ ;\ \text{first?}\ ;\ \text{b}\ ;\ zero\ \vdash_{type-list}\ nil}$$

And the recursive case calls $\vdash_{type}$ and $\vdash_{type-list}$, concatenating the outputs of each:

$$\frac{\begin{array}{c}\Psi\ ;\ \text{b}\ ;\ \xi([0..=(\text{first?}\ ?\ \Psi(\text{MTSF}) : \Psi(\text{MTSR}))])\ \vdash_{type}\ \tau \\ \Psi\ ;\ \text{b}\ ;\ \text{x}\ \vdash_{type-list}\ \vec{\tau}\end{array}}{\Psi\ ;\ \text{first?}\ ;\ \text{b}\ ;\ succ(\text{x})\ \vdash_{type-list}\ (\tau : \vec{\tau})}$$

4.5.3 Types ($\vdash_{type}$)

This judgment creates a CPS-calculus type. It gets the following parameters:

- Ψ: Associative list of parameters.
- b: Set of base types.
- x: Natural indicating the remaining size of the type being generated. The recursion is based on this parameter.

When x $\equiv zero$, this judgment returns a random base type from b. This is the base case:

$$\frac{}{\Psi\ ;\ \text{b}\ ;\ zero\ \vdash_{type}\ \xi(\text{b})}$$

When x $> zero$, the recursive step is taken. First, a number m is generated, indicating how many subtypes are going to be created. Then, a size k_i is generated for each one of the prospect sub-types, such that $\sum_{i=0}^{\text{m}} \text{k}_i \equiv \text{x} - 1$. A recursive call is performed for every one of the m sub-types with their corresponding sizes. Lastly, being τ_i the outputs of the recursive calls, the returned type is the negation of the list of the outputs, like $\neg(\tau_1,\ \ldots,\ \tau_{\text{m}})$. This recursive step is defined below:

$$\frac{\begin{array}{c}\text{max\_subtypes} := (\xi(\mathbb{R}) > \Psi(\text{EST}))\ ?\ \Psi(\text{MNST}) : \min(4, \Psi(\text{MNST})) \\ \text{m} = \xi([1..=\text{max\_subtypes}]) \\ \text{k}_i = \xi(\mathbb{N})\ ,\quad \forall i \in \mathbb{N}\ .\ 1 \leq i \leq \text{m} \\ \sum_{i=1}^{\text{m}} \text{k}_i \equiv \text{x} \\ \Psi\ ;\ \text{b}\ ;\ \text{k}_i\ \vdash_{type}\ \tau_i\ ,\quad \forall i \in \mathbb{N}\ .\ 1 \leq i \leq \text{m}\end{array}}{\Psi\ ;\ \text{b}\ ;\ succ(\text{x})\ \vdash_{type}\ \neg(\tau_1, \cdots, \tau_{\text{m}})}$$

4.6 Walk Generator ($\vdash_{walk}$)

The type environment obtained by $\vdash_{type-env}$, as previously stated, are two DAGs represented as adjacency lists, and these two DAGs are related in that one is the transpose of the other.

The core idea behind this generation algorithm lies in performing random walks on these graphs, and then using these walks as an execution plan for commands, guiding the generation by processing them. Explanation of how these walks are processed will be given later, as this section is focused on explaining how these walks are created in the first place, and all the optional transformations they can be subjected to.

When a new parameter is added to the variables environment, it can either trigger a *Collide* (explained together with $\vdash_{cmd}$) or attempt to start a random walk. When it starts a random walk, this parameter's type is its root and the parameter itself is bound to be used by the last jump that walk encodes.

A step going to a super-type ($\tau_1 \to \neg(\ldots, \tau_1, \ldots)$) is going to be referred to as moving *forward*, and a step going to a sub-type ($\neg(\ldots, \tau_1, \ldots) \to \tau_1$) is going to be referred to as moving *backwards*. The arrows will have a label indicating this, like this:

$$\neg(\neg(), X) \xrightarrow{\text{B}} \neg() \xrightarrow{\text{F}} \neg(\neg())$$

The $\vdash_{walk}$ judgment gets the following as parameters:

- Ψ: Associative list of parameters.
- Θ: Type environment.
- Δ: Current walk environment, consisting of four structures. The first is an associative list of walks (Δ_W), the second an associative list of nodes (Δ_N), the third an associative list of components (Δ_C), and the fourth a directed acyclic graph of dependencies (Δ_G). These pieces of the walk environment will be explained in more detail later in this section. The new walk will be integrated here.
- b: The set of base types in the type environment. Could be obtained by filtering Θ under the condition that a given type does not have sub-types, but it's included in the parameters nonetheless as to avoid cluttering the formalization with repeated filters.
- vn: The name of the parameter that started this walk.
- τ: The type of the parameter that started this walk. This is going to be the root of the walk.
- d: Controls the allowed maximum depth for the $\vdash_{branch}$ transformation. This parameter controls the recursion in that judgment.

- w_{acc}: The remaining number of steps before the algorithm prohibits further walks to move forward (in the direction of super-types). Effectively, this bounds the size of the generated term, guaranteeing termination.
- w: The maximum number of steps that can be taken by this walk. Usually the walk will have exactly w steps, but if $\mathrm{w}_{acc} \leq 0$, it's possible for the walk to have less than that by reaching a type without sub-types in less than w steps.

And it returns a new walk environment Δ', consisting of three associative lists and a graph, and an updated walk steps accumulator.

The judgment has two rules, and one of them represents a special case. The judgment first performs a random walk using $\vdash_{steps}$ in both cases, initially returning a list of steps and labels. If the list has only one element, then the walk had 0 steps, and the single element is the root type. A walk with 0 steps is only allowed if the root type is $\neg()$, but may also happen when it's a base type and $\mathrm{w}_{acc} \leq 0$, although those cases should **not** be permitted. The first rule covers such an event and is the mentioned special case, triggered when the list of steps has length 1 and the root type is different than $\neg()$, meaning it must be a base type. The rule simply discards the singleton step and returns the current walk environment unchanged:

$$\frac{\begin{array}{c}\hat{\Theta} := (\mathrm{w}_{acc} > 0)\ ?\ \Theta : (\mathrm{map}\ ((\mathrm{k},\ (\Theta_F,\ \Theta_B)) \rightarrow (\mathrm{k},\ (\{\emptyset\},\ \Theta_B)))\ \Theta) \\ \Psi\ ;\ \hat{\Theta}\ ;\ \tau\ ;\ \cdot\ ;\ \mathrm{w} \vdash_{steps} \mathrm{stps} \\ \mathrm{length\ stps} \equiv 1 \\ \tau \not\equiv \neg()\end{array}}{\Psi\ ;\ \Theta\ ;\ \Delta\ ;\ \mathrm{b}\ ;\ \mathrm{vn}\ ;\ \tau\ ;\ \mathrm{d}\ ;\ \mathrm{w}_{acc}\ ;\ \mathrm{w} \vdash_{walk} \Delta\ ;\ \mathrm{w}_{acc}}$$

The second case follows from the walking steps. It processes this walk, changing its structural representation and performing optional transformations. It also integrates the new walk into the walk environment. The rule is defined below:

$$\hat{\Theta} := (\mathrm{w}_{acc} > 0)\ ?\ \Theta : (\mathrm{map}\ ((\mathrm{k}, (\Theta_F,\ \Theta_B)) \to (\mathrm{k}, (\{\emptyset\},\ \Theta_B)))\ \Theta)$$

$$\Psi\ ;\ \hat{\Theta}\ ;\ \tau\ ;\ \cdot\ ;\ \mathrm{w} \vdash_{steps} \mathrm{stps}$$

$$(\mathrm{length\ stps} > 1) \vee (\tau \equiv \neg())$$

$$\Psi\ ;\ \mathrm{vn}\ ;\ nil\ ;\ \cdot\ ;\ \mathrm{stps} \vdash_{assemble} \mathrm{assembled}$$

$$\Psi\ ;\ \xi(\mathbb{R}^u)\ ;\ \mathrm{w}_{acc} - \lceil \tfrac{\mathrm{length\ stps}}{2} \rceil\ ;\ \mathrm{assembled} \vdash_{merge} \mathrm{merged}\ ;\ \mathrm{w}'_{acc}$$

$$\Psi\ ;\ \hat{\Theta}\ ;\ \mathrm{b}\ ;\ \mathrm{d}\ ;\ \mathrm{w}'_{acc}\ ;\ \mathrm{merged} \vdash_{branch} \mathrm{branched}\ ;\ \mathrm{w}''_{acc}$$

$$\Psi\ ;\ \mathrm{branched} \vdash_{void} \mathrm{voided}$$

$$\mathrm{w\text{-}id} = \xi_{id}$$

$$\mathrm{w\text{-}id}\ ;\ \mathrm{voided} \vdash_{transform} \delta_N\ ;\ \delta_C\ ;\ \delta_G$$

$$\Psi\ ;\ \xi(\mathbb{R}^u)\ ;\ \delta_C\ ;\ \delta_G\ ;\ \delta_C \vdash_{fold} \delta'_C\ ;\ \delta'_G$$

$$\Psi\ ;\ (\Delta_W \cup \{(\mathrm{w\text{-}id}, \{\mathrm{w\text{-}id}\})\})\ ;\ \Delta_N\ ;\ \Delta_C\ ;\ (\Delta_G \cup \delta'_G)\ ;\ \mathrm{w\text{-}id}\ ;\ \{\emptyset\}\ ;\ \delta'_C \vdash_{reach} \Delta'_W\ ;\ \Delta'_C\ ;\ \Delta'_G$$

$$\frac{\Delta'_N := \mathrm{map}\ (\mathrm{ready?}\ \Delta'_C)\ (\Delta_N \cup \delta_N)}{\Psi\ ;\ \Theta\ ;\ \Delta\ ;\ \mathrm{b}\ ;\ \mathrm{vn}\ ;\ \tau\ ;\ \mathrm{d}\ ;\ \mathrm{w}_{acc}\ ;\ \mathrm{w} \vdash_{walk} \Delta'\ ;\ \mathrm{w}''_{acc}}$$

Where the function ready? just represents the operation below, mapped with a singular case. Due to this formalization preferring to avoid index-based access on tuples, tuple unpacking is often the preferred syntax to perform operations on some tuple elements:

$$\begin{aligned}
&\mathrm{ready?}\ \Delta'_C\ (\mathrm{k}, (\mathrm{wn}, \tau, \mathrm{cs}, \mathrm{r?})) \\
&\quad \equiv\ (\mathrm{k}, (\mathrm{wn}, \tau, \mathrm{cs}, (\bigwedge_{\mathrm{cid} \in \mathrm{cs}} \mathrm{let}\ ((\mathrm{nid}, \mathrm{vn}, \mathrm{ty}, \mathrm{dep\text{-}by}, \mathrm{dep\text{-}on}, \mathrm{col?}) := \Delta'_C(\mathrm{cid})) \\
&\qquad\qquad \mathrm{in}\ (\mathrm{dep\text{-}on} \equiv \{\emptyset\}))))
\end{aligned}$$

Going back to the judgment's second rule, below is a brief summary of its steps:

1. Determine if moving forward (to super-types) is still allowed or not. If not, remove the forward side of the type environment graph.

2. Walk at most w steps and group the traversed types into a list, which is interleaved with B or F flags that indicate the type of step it was.

3. Determine if the steps list is valid for this judgment, that is, if the length of said list is more than one or if the root type is $\neg()$.

4. Calls $\vdash_{assemble}$ to build the walk into a recursive data structure, which consists of a node hosting a number of components, and each component can in turn host another node inside of it. A node represents a step in the walk, and it will result in a jump command when processed later on. Every node has at least one component, representing a variable position in the jump. For example, a node that results in the jump $a\langle b, c\rangle$ has 3 components, one for each variable location.

5. Calls $\vdash_{merge}$ to have a chance to do an operation that merges two adjacent nodes. Before that, however, subtract the number of steps taken from w_{acc}. Currently, the steps still have labels between each type, so its length approximates double, hence the division.

6. Calls $\vdash_{branch}$ to have a chance to recursively create sub-walks, forking the walk into two or more different branches. Initially, a node could only have 0 or 1 components with children nodes. Branching is the operation that allows a node to surpass this limitation.

7. Calls $\vdash_{void}$ to have a chance to assign a free variable to a component created by the walk. This forces affected components to have their variable position be filled with a free variable, instead of allowing for environment lookups or bind creations.

8. Creates an identifier for the new walk and then calls $\vdash_{transform}$ to once again change its data structure, now turning it into a flat 3-tuple consisting of associative lists for nodes and components and a graph of dependencies. These dependencies are related to the previous recursive data structure, where node A could host inside one of its components a different node B, and therefore one depends on the other.

9. Calls $\vdash_{fold}$ to have a chance to create dependencies between different parts of the same walk. This forces parts of the walk to have to wait for other parts to finish processing before it can go. This allows for a strict order of execution that would otherwise be extremely unlikely to happen, in terms of probability, hence increasing diversity.

10. Calls $\vdash_{reach}$ to have a chance to create dependencies between the current walk and other walks in the environment. Similar to $\vdash_{fold}$, but this time the relationship happens between nodes of different walks. This last judgment also has a second purpose: to add every component in δ_C to Δ_C as they're being processed.

11. Finally, this map checks, for every node in the environment, if it's ready to be processed next. A node is ready if it has no unprocessed dependencies.

The next subsections will elaborate on each of these judgments, defining exactly how they operate and showing each of their effects on the walk.

4.6.1 Walk along the environment's types ($\vdash_{steps}$)

This judgment performs a second-order random walk, and that means it also takes into account the previous vertex when deciding on the next, instead of only relying on the current one. This is done by introducing the $\Psi(\text{NBTT})$ parameter, which indicates the probability of removing the immediately previous source vertex from the options for the next step. The higher this probability, the less likely it is for the random walk to backtrack. However, this does not have any effect if the backtracking vertex is the only possible option, as removing it would result in the walk getting stuck.

The $\vdash_{steps}$ judgment takes the following parameters:

- Ψ: An associative list of parameters.

- Θ: The type environment, on top of which the random walk will occur.
- τ_{curr}: The current vertex. If there will be a next node, it's going to be a neighbor of this vertex, and that means it's either going to be one of its super-types or sub-types.
- τ_{prev}: The previous vertex. Depending on the probability in $\Psi(\text{NBTT})$ and if there are other options available for the current vertex, this vertex could be removed in order to reduce the likelihood of backtracking. If this is the beginning of the walk, there is no previous vertex and therefore this parameter will have as its value the special symbol $\cdot$.
- w: A natural indicating the remaining walk length. The recursion is based on this parameter.

This judgment outputs a list consisting of CPS-calculus types, interleaved with F or B labels.

First the base case, when w $\equiv zero$:

$$\frac{}{\Psi\ ;\ \Theta\ ;\ \tau_{curr}\ ;\ \tau_{prev}\ ;\ zero\ \vdash_{steps}\ (\tau_{curr} : nil)}$$

Then a special case, which can happen when w_{acc} is non-positive, as this leads to $\vec{\tau}_F \equiv \{\emptyset\}$ and therefore an empty set of possible next steps when the current vertex is a base type (as base types have $\vec{\tau}_B \equiv \{\emptyset\}$):

$$\frac{\begin{array}{c}(\vec{\tau}_F,\ \vec{\tau}_B) := \Theta(\tau_{curr}) \\ \tau_{env} := \vec{\tau}_F\ \cup\ \vec{\tau}_B \\ |\tau_{env}| \equiv 0\end{array}}{\Psi\ ;\ \Theta\ ;\ \tau_{curr}\ ;\ \tau_{prev}\ ;\ succ(w)\ \vdash_{steps}\ (\tau_{curr} : nil)}$$

And finally, the recursive case:

$$\frac{\begin{array}{c}(\vec{\tau}_F,\ \vec{\tau}_B) := \Theta(\tau_{curr}) \\ \tau_{env} := \vec{\tau}_F\ \cup\ \vec{\tau}_B \\ |\tau_{env}| > 0 \\ \tau_{next} = ((\tau_{env} \not\equiv \{\tau_{prev}\}) \wedge (\xi(\mathbb{R}^u) > \Psi(\text{NBTT}))\ ?\ \xi(\tau_{env} \backslash \{\tau_{prev}\}) : \xi(\tau_{env})) \\ L\ :=\ ((\tau_{next} \equiv \neg\vec{\tau}_{next-subtypes}) \wedge (\tau_{curr} \in \vec{\tau}_{next-subtypes}))\ ?\ F : B \\ \Psi\ ;\ \Theta\ ;\ \tau_{next}\ ;\ \tau_{curr}\ ;\ \text{w}\ \vdash_{steps}\ \text{wlk}\end{array}}{\Psi\ ;\ \Theta\ ;\ \tau_{curr}\ ;\ \tau_{prev}\ ;\ succ(\text{w})\ \vdash_{steps}\ (\tau_{curr} : L : \text{wlk})}$$

Note that a recursive step adds two elements to the list: the current vertex τ_{curr} and a flag which indicates, considering the known next vertex τ_{next}, if the step taken is forward or backwards. It does not add the next vertex, as that responsibility is pushed to the next call.

4.6.2 Assemble the walk data structure ($\vdash_{assemble}$)

After the random walk, it's necessary to transform the list of steps into a specific recursive data structure. This is meant mainly to facilitate the formalization of the algorithm and to reason about it. For such, consider the following grammar:

$$
\begin{aligned}
\text{walk} &::= \text{node} \\
\text{node} &::= (\text{iden typ } (\text{component } ...+)) \\
\text{component} &::= (\text{iden iden? iden? node?}) \\
\text{typ} &::= \text{iden} \mid \neg(\text{typ } ...) \\
\text{iden?} &::= \text{iden} \mid \cdot \\
\text{node?} &::= \text{node} \mid \cdot \mid \circ \\
\text{iden} &::= \text{unique-identifier}
\end{aligned}
$$

Every node is going to represent a jump, and each of its components will represent a variable of that jump, such that the component in head position is the continuation being jumped to (in subject position) and the rest are the parameters being passed to it (in object position). Almost every vertex in the walk is going to be assigned to a component, and as such there's a relation not only of a node A leading to a next node B, but also of a component from A leading to a component from B. The only vertex that gets assigned to no components is the last one, except in the special case when it's the only vertex in the walk, i.e. when the walk has 0 steps and is rooted at $\neg()$, or when a sampled variable $\alpha \sim \mathbb{U}[0, 1]$ is one such that $0 \leq \Psi(\text{LTWT}) < \alpha \leq 1$, where $\Psi(\text{LTWT})$ is a threshold that changes with every command generated. The α variable is independently sampled for each node, and this pattern continues on with the next transformations in this section.

A node has an unique identifier serving as its internal name, the type of the continuation in subject position, and a list of components. A component also has an unique identifier serving as its internal name, and then three optional fields: a variable name, the internal name of the component from the previous node that led to the current one, and a nested node. The absence of a variable name and of a previous component will always be labeled with the symbol $\cdot$, but the absence of a nested node may be represented in two ways: most of the times, the symbol $\cdot$ is going to be used, like the others, but sometimes the final vertex in a walk will be labeled with a $\circ$, as dictated by the threshold parameter $\Psi(\text{LTWT})$. The reason for that will be explained later.

A component having a variable name is a promise that, when (or if) the node containing that component is processed, the jump will use that variable in that component's position. Initially, only one component has a variable name: the parameter that originated this walk, and in essence this means that by creating a walk, a parameter is already planning one of its usages. The parameter can be used in other places to fill components of the same type that don't have a variable name, but by creating a walk, it makes an attempt to have at least one usage. Ways to fill in the variable

field will be mentioned later on, but the most significant one is an operation called *collide*, where a new parameter assigns itself to a component of an existing walk instead of generating a new one.

A very important detail is that the length of the component list in the node rule is semantically well-defined: it's the number of parameters the continuation in subject position takes plus one, where the first component represents a variable with type *typ* and all of the components that come after represent its sub-types in order.

For example, if typ $\equiv \neg(X, \neg())$, then the node that has this typ represents a jump like $a\langle b, c\rangle$, where a has type $\neg(X, \neg())$, b has type X and c has type $\neg()$. For that node, the first component will represent a, the second b and the third c.

The judgment $\vdash_{assemble}$ builds a walk, which at this point is just a node, by generating a node for every step taken and then nesting them inside each other. It takes these parameters:

- Ψ: An associative list of parameters.
- name: Initially, this has the variable name of the parameter that generated this walk. But this is the case only for the very first call to $\vdash_{assemble}$ in a given walk, and the others all have the symbol $\cdot$ in this place. Sub-walks and the very first walk of a generation also have $\cdot$ here, even for the first call. The $\cdot$ symbol will often be used to represent an absence of a value.
- acc: An accumulator of nodes. The ones in this accumulator don't have other nodes inside of their components, but at the end of the process this accumulator will be processed in order to create this nesting.
- prev: An identifier of a random component from the previous step that has the same type as the head of the remaining list of steps, i.e., the step currently being processed. If it's the first step in the walk, then this argument is $\cdot$ instead.
- stps: The list of steps and labels. The recursion is based on this parameter.

And it outputs a node, as mentioned.

For a walk with n steps there will be n nodes created, and as such, walks with 0 steps are generally not allowed, which also makes sense because a walk with 0 steps is ambiguous as to whether the root type (the only vertex in the walk) is meant to be the type of a variable in subject position (as a continuation being jumped to, which would be a 1-step walk with label B) or in object position (as a parameter, which would be a 1-step walk with label F). The only exception to this is if the root type is $\neg()$, in which case a walk with 0 steps is allowed and its root type is interpreted as the type of the continuation being jumped to (subject position). This special case is the responsible for mapping jumps of the form $k\langle\rangle$, and outputs only one node, which has only one component. Here's its rule:

$$\Psi\ ;\ \text{name}\ ;\ nil\ ;\ \cdot\ ;\ (\neg() : nil)\ \vdash_{assemble}\ (\xi_{id},\ \neg(),\ ((\xi_{id},\ \text{name},\ \cdot,\ \cdot) : nil))$$

And now the base case, which takes the list of nodes accumulator, and groups these nodes into a single nested one. It uses the standard function to reverse a list and another function called nestNodes, to be defined right after the rule. It also samples an $\alpha \sim \mathbb{U}[0, 1]$, and if $\alpha > \Psi(\text{LTWT})$, it labels the component from the most recent node in acc whose identifier is equal to prev by changing its child node field from $\cdot$ to $\circ$. It does so using a function called labelHead, also to be defined after this rule:

$$\frac{\begin{array}{c}\alpha := \xi(\mathbb{R}^u) \\ \text{acc}' := (\alpha > \Psi(\text{LTWT}))\ ?\ (\text{labelHead prev acc}) : \text{acc}\end{array}}{\Psi\ ;\ \text{name}\ ;\ \text{acc}\ ;\ \text{prev}\ ;\ (\tau_{last} : nil) \vdash_{assemble} (\text{nestNodes . reverse acc})}$$

```
nestNodes :: [node] → node
nestNodes ((iden_1 typ_1 comps_1) : (iden_2 typ_2 comps_2) : ns)
        = (iden_1 typ_1 (nestNodesInternal comps_1 (findPrev comps_2)
            ((iden_2 typ_2 comps_2) : ns)))
nestNodes (nd : nil) = nd
            where
    findPrev :: [component] → iden
    findPrev ((_ _ · _) : cs) = findPrev cs
    findPrev ((ident _ prev _) : _) = prev
    nestNodesInternal :: [component] → iden → [node] → [component]
    nestNodesInternal ((parent varname prev ·) : cs) parent nodes
            = ((parent varname prev (nestNodes nodes)) : cs)
    nestNodesInternal ((notParent varname prev child?) : cs) parent nodes
            = ((notParent varname prev child?) : (nestNodesInternal cs parent nodes))
```

```
labelHead :: iden → [node] → [node]
labelHead prev ((n-id n-typ n-comps) : nds)
        = ((n-id n-typ (labelComp prev n-comps)) : nds)
            where
    labelComp :: iden → [component] → [component]
    labelComp prev ((prev · · ·) : comps) = ((prev · · ○) : comps)
    labelComp prev ((other · · ·) : comps) = ((other · · ·) : (labelComp prev comps))
```

This judgment has 2 distinct recursive rules, one for the F label and the other for the B label. The purpose of both is to create a new node, add it to the accumulator and follow to the next recursive call, but they differ in some ways.

Starting with the rule for the F label. As the step is going to a super-type, the source vertex is the type for one of the jump's parameters, and the destination vertex is the type for the jump's head. As such, the new node being created has type equal to the destination vertex. Also, the name given to the head component is the one passed along to serve as the *prev* name for the next call, and the current *prev* is assigned to a random tail component of the same type as the source vertex. Here is the rule:

$$\frac{\begin{array}{c}\text{dest-iden} = \xi_{id} \\ \text{iter} := \text{zip } [0..|\vec{\tau}_{dst}|]\ \vec{\tau}_{dst} \\ \text{path-idx} := \text{first . } \xi(\text{filter (i ; } \tau \to \tau \equiv \tau_{src}) \text{ iter)} \\ \text{comps} := \text{map (i ; } \tau \to \text{(i} \equiv \text{path-idx) ? } (\xi_{id}, \text{ name, prev, } \cdot) : (\xi_{id}, \cdot, \cdot, \cdot)) \text{ iter} \\ \text{new-acc} := ((\xi_{id}, \neg\vec{\tau}_{dst}, ((\text{dest-iden}, \cdot, \cdot, \cdot) : \text{comps})) : \text{acc}) \\ \Psi\ ;\ \cdot\ ;\ \text{new-acc}\ ;\ \text{dest-iden}\ ;\ (\neg\vec{\tau}_{dst} : \text{stps}) \vdash_{assemble} \text{out}\end{array}}{\Psi\ ;\ \text{name}\ ;\ \text{acc}\ ;\ \text{prev}\ ;\ (\tau_{src} : F : \neg\vec{\tau}_{dst} : \text{stps}) \vdash_{assemble} \text{out}}$$

And now for the B label. Very similar to the F label, but with some of the logic reversed. The step is going to a sub-type, so the source vertex is the type for the jump's head, and the destination vertex is the type for one of the jump's parameters. The new node has type equal to the source vertex. About the names, the head component gets the *prev*, and the next *prev* is the name of a random tail component of the same type as the destination vertex. Here is the rule:

$$\frac{\begin{array}{c}\text{nodes} = (\text{map (iden} \to \text{(iden, } \cdot, \cdot, \cdot)) \text{ (repeat } \xi_{id}\ |\vec{\tau}_{src}|)) \\ \text{dest-iden} := (\text{first . first . } \xi(\text{filter (nd, } \tau \to \tau \equiv \tau_{dst}) \text{ (zip nodes } \vec{\tau}_{src})) \\ \text{new-acc} := ((\xi_{id}, \neg\vec{\tau}_{src}, ((\xi_{id}, \text{ name, prev, } \cdot) : \text{nodes})) : \text{acc}) \\ \Psi\ ;\ \cdot\ ;\ \text{new-acc}\ ;\ \text{dest-iden}\ ;\ (\tau_{dst} : \text{stps}) \vdash_{assemble} \text{out}\end{array}}{\Psi\ ;\ \text{name}\ ;\ \text{acc}\ ;\ \text{prev}\ ;\ (\neg\vec{\tau}_{src} : B : \tau_{dst} : \text{stps}) \vdash_{assemble} \text{out}}$$

4.6.3 Merge neighboring nodes ($\vdash_{merge}$)

Before explaining $\vdash_{merge}$, consider the following example walk: $\tau_1 \xrightarrow{\text{F}} \neg(\tau_1, \tau_2) \xrightarrow{\text{B}} \tau_2 \xrightarrow{\text{F}} \neg(\tau_2, \tau_3) \xrightarrow{\text{B}} \tau_3$.

From the way $\vdash_{assemble}$ is done, the first two steps in are going to result in two distinct adjacent jumps to continuations of the same type, that type being $\neg(\tau_1, \tau_2)$. This is a pattern that happens every time a F-step is followed by a B-step, and also happens whenever a B-step is followed by an F-step if those two steps represent a backtrack. This repetition is therefore extremely common, and may impact diversity.

To solve this, the $\vdash_{merge}$ transformation was designed. It is applied if two neighboring nodes have the exact same type **and** if a sampled $\alpha \sim \mathbb{U}[0, 1]$ exceeds a threshold $0 \leq \Psi(\text{MGT}) \leq 1$.

First, here's an auxiliary function that is needed by this transformation:

compNextNode? :: [component] $\rightarrow$ component?
compNextNode? ((ident varname prev $\cdot$) : cs) $=$ compNextNode? cs
compNextNode? ((ident varname prev $\circ$) : cs) $=$ compNextNode? cs
compNextNode? ((ident varname prev node) : cs) $=$ (ident varname prev node)
compNextNode? nil $=$ $\cdot$

This function gets a node's list of components and returns the component that has the nested child node, if it exists. Note that although there are multiple components, at this stage only zero or one of them will contain a child node.

Now, listing the judgment's parameters:

- Ψ: associative list of parameters.
- α: random variable sampled from an uniform distribution of the closed real unit range. Used to determine if a merge is going to happen or not.
- w_{acc}: the current steps accumulator. If a merge is performed, w_{acc} will get refunded one unit, as a node is removed.
- node: the node currently being analyzed. The recursion is based on this parameter.

The judgment returns two values: the potentially modified node and w_{acc}.

The base case, when the current node is a leaf:

$$\frac{\cdot \equiv \text{(compNextNode? comps)}}{\Psi \ ; \ \alpha \ ; \ \text{w}_{acc} \ ; \ \text{(ident, typ, comps)} \ \vdash_{merge} \ \text{(ident, typ, comps)} \ ; \ \text{w}_{acc}}$$

Now starting the recursive cases. The first one represents the event of a merge not happening on a non-leaf node. This happens either when the current node and its child are of different types, making the merge impossible, or when $\alpha \leq \Psi(\text{MGT})$. This rule leaves the current node unchanged and the algorithm follows to the next one, resampling α:

$$\frac{\begin{array}{c}\text{(comp-id, \_, \_, (nxt-id, nxt-typ, nxt-comps))} \equiv \text{(compNextNode? comps)} \\ (\text{nxt-typ} \not\equiv \text{typ}) \vee (\alpha \leq \Psi(\text{MGT})) \\ \Psi \ ; \ \xi(\mathbb{R}^u) \ ; \ \text{w}_{acc} \ ; \ \text{(nxt-id, nxt-typ, nxt-comps)} \ \vdash_{merge} \ \text{new-node} \ ; \ \text{w}'_{acc} \\ \text{f} := \text{((idt, vn, pv, nd?)} \rightarrow \text{(idt, vn, pv, ((idt} \equiv \text{comp-id) ? new-node : nd?)))} \\ \text{comps' := (map f comps)}\end{array}}{\Psi \ ; \ \alpha \ ; \ \text{w}_{acc} \ ; \ \text{(ident, typ, comps)} \ \vdash_{merge} \ \text{(ident, typ, comps')} \ ; \ \text{w}'_{acc}}$$

The other recursive step deals with the event of the merge happening. For that to be possible, the current node must be a non-leaf, its child must have the same type as the current

node, and $\alpha > \Psi(\text{MGT})$. A merge creates a new node, using the internal identifier of the parent and a list of merged components. Each merged component will have the identifier of the corresponding child component (so its child will have a prev pointing to an existing component), the variable name of the parent (so the promise at the root parameter is not lost), the previous identifier of the parent (as using the child's prev would now point to a non-existent component), and the child node of the child (as the child node of the parent is currently being merged with the parent):

$$\frac{\begin{array}{c}\text{(comp-id, \_, \_, (nxt-id, nxt-typ, nxt-comps))} \equiv \text{(compNextNode? comps)} \\ \text{nxt-typ} \equiv \text{typ} \\ \alpha > \Psi(\text{MGT}) \\ \text{f} := \text{((id1, vn1, pv1, nd1) ; (id2, vn2, pv2, nd2)} \rightarrow \text{(id2, vn1, pv1, nd2))} \\ \text{comps'} := \text{map f comps nxt-comps} \\ \Psi \ ; \ \xi(\mathbb{R}^u) \ ; \ \text{w}_{acc} \ ; \ \text{(ident, typ, comps')} \vdash_{merge} \text{new-node ; w}'_{acc}\end{array}}{\Psi \ ; \ \alpha \ ; \ \text{w}_{acc} \ ; \ \text{(ident, typ, comps)} \vdash_{merge} \text{new-node ; } (\text{w}'_{acc} + 1)}$$

4.6.4 Branch into sub-walks rooted on empty components ($\vdash_{branch}$)

Consider, for example, this 1-step walk: $\tau_1 \xrightarrow{F} \neg(\tau_1, \tau_2, \tau_3)$. The node originated from this step will host 4 components: the one associated with τ_1 will almost always be linked to a variable, that being the parameter that started this walk[4], but the other three will not have any variables associated with them, and neither will they have a previous component nor a child node, as a 1-step walk will only generate one node.

Components that have the symbol $\cdot$ in the places of the predecessor identifier and the child node can be considered *side components*. They are not considered to be a direct part of the walk that originated the node structure. If a component in the last node of a walk has had its $\cdot$ replaced with $\circ$ in $\vdash_{assemble}$, then that component will **not** be considered a side component. If a side component also does not have an associated variable, it can be also considered an *empty component*. It's worth mentioning that at this point, almost every side component is also an empty component, with the only exception being the special case of main (not branched) walks with 0 steps rooted at $\neg()$ and that were given a variable.

Empty components can be seen as adjacent to the main walk. This is not a problem, but it opens up the possibility for sub-walks, with the goal of increasing diversity. The process of starting a sub-walk rooted on an empty component is called *branching*.

Two judgments will be defined, and they will act in a mutually recursive manner. This pattern will continue through the next transformations as well, such that the external judgment acts on nodes and the internal judgment acts on components.

First defining the node-level branch judgment. Its parameters are:

[4] If the step had label B it would be possible for it to not have a variable associated, specifically in the case where it's the first walk of the generation

- Ψ: Associative list of parameters.
- Θ: Type environment.
- b: The set of base types in the type environment. Could be obtained by filtering Θ under the condition that a given type does not have sub-types, but it's included in the parameters nonetheless as to avoid cluttering the formalization with repeated filters.
- d: Natural that indicates the remaining maximum depth. If d $\equiv zero$, then further branching is not allowed. If d $> zero$, then branching is allowed, and the sub-walks are gonna have $d' = d - 1$.
- w_{acc}: The remaining steps accumulator. Sub-walk steps also consume from it, however they do so in a slightly different way, a fact which is going to be elaborated upon shortly.
- node: The recursion is based on this parameter.

This judgment returns a potentially modified node and a new w_{acc}.

Base case: When d $\equiv zero$, no branching happens.

$$\frac{}{\Psi \text{ ; } \Theta \text{ ; b ; } zero \text{ ; } \text{w}_{acc} \text{ ; node} \vdash_{branch} \text{node ; } \text{w}_{acc}}$$

Recursive step: Calls $\vdash_{branch-comp}$ for all of the current node's components, grouping the outputs with the rest of the original node and returning it alongside the minimum w_{acc} among all of the recursive calls.

$$\frac{\begin{array}{c}(\text{ident}, \neg\vec{\tau}, (\text{comp}_0 : \ldots : \text{comp}_n : nil) \equiv \text{nd} \\ \Psi \text{ ; } \xi(\mathbb{R}^u) \text{ ; } \Theta \text{ ; b ; } succ(\text{d}) \text{ ; t ; } \text{w}_{acc} \text{ ; comp}_i \vdash_{branch-comp} \text{comp}'_i \text{ ; } (\text{w}_{acc})'_i \\ \forall (i, t) \in (\text{zip } [0..{=}n] \ \ (\neg\vec{\tau} : \vec{\tau})) \\ \text{nd}' := (\text{ident, typ, } (\text{comp}'_0 : \ldots : \text{comp}'_n : nil)) \\ \text{out-w}_{acc} := \min(\{(\text{w}_{acc})'_i \mid \forall i \in \mathbb{N} \ . \ 0 \leq i \leq n\})\end{array}}{\Psi \text{ ; } \Theta \text{ ; b ; } succ(\text{d}) \text{ ; } \text{w}_{acc} \text{ ; nd} \vdash_{branch} \text{nd}' \text{ ; out-w}_{acc}}$$

Now to define the component-level branching judgment. These are its parameters:

- Ψ: Associative list of parameters.
- α: Random probability score sampled from an uniform distribution over the closed unit interval. It must be larger than the Branch Threshold Ψ(BT) in order for the branching to occur.
- Θ: The type environment.
- b: The set of base types in the type environment.

- d: Same parameter as the node-level branching.
- w_{acc}: The accumulator of steps. It is replicated at every branch and then at the end the minimal across all branches is returned.
- comp: The current component. This parameter is the basis of the recursion.

This judgment returns a potentially modified component and w_{acc}.

Base case: The component does not have a child node, but branching does not happen, because of at least one of these conditions:

- The score was not enough: $\alpha \leq \Psi(\mathrm{BNT})$
- It's not an empty component: $(\mathrm{vn?} \not\equiv \cdot) \vee (\mathrm{pv?} \not\equiv \cdot) \vee (\mathrm{nd?} \not\equiv \cdot)$
- The generated walk would have 0 steps: $(\mathrm{w}_{acc} \leq 0) \wedge (\mathrm{typ} \in (\{\neg()\} \cup \mathrm{b}))$

In those cases, simply return the component and w_{acc} unchanged:

$$\frac{\begin{array}{c}(\text{iden, vn?, pv?, child?}) := \text{nd?} \\ (\text{child?} \equiv \cdot) \vee (\text{child?} \equiv \circ) \\ \text{score-fail} := (\alpha \leq \Psi(\mathrm{BNT})) \\ \text{empty-fail} := (\text{vn?} \not\equiv \cdot) \vee (\text{pv?} \not\equiv \cdot) \vee (\text{child?} \not\equiv \cdot) \\ \text{steps-fail} := ((\mathrm{w}_{acc} \leq 0) \wedge (\text{typ} \in \mathrm{b})) \\ \text{score-fail} \vee \text{empty-fail} \vee \text{steps-fail}\end{array}}{\Psi \ ; \alpha \ ; \Theta \ ; \mathrm{b} \ ; \mathit{succ}(\mathrm{d}) \ ; \mathrm{typ} \ ; \mathrm{w}_{acc} \ ; \mathrm{nd?} \vdash_{branch-comp} \mathrm{nd?} \ ; \mathrm{w}_{acc}}$$

Recursive case 1: The component has a child node, which implies in it not being an empty component. The recursion continues along the child node:

$$\frac{\begin{array}{c}(\text{iden, vn?, pv?, child?}) := \text{nd?} \\ (\text{n-id}, \neg\vec{\tau}, \text{comps}) \equiv \text{child?} \\ \Psi \ ; \Theta \ ; \mathrm{b} \ ; \mathit{succ}(\mathrm{d}) \ ; \mathrm{w}_{acc} \ ; \text{child?} \vdash_{branch} \text{child}' \ ; \mathrm{w}'_{acc} \\ \text{nd'} := (\text{iden, vn?, pv?, child}')\end{array}}{\Psi \ ; \alpha \ ; \Theta \ ; \mathrm{b} \ ; \mathit{succ}(\mathrm{d}) \ ; \mathrm{typ} \ ; \mathrm{w}_{acc} \ ; \mathrm{nd?} \vdash_{branch-comp} \mathrm{nd}' \ ; \mathrm{w}'_{acc}}$$

Recursive case 2: $\alpha > \Psi(\mathrm{BNT})$, the component is an empty component and it's possible to have a walk with at least one step. In this case, a branching walk will happen and will be attached to the current walk, effectively creating a forking point. The Branching Walk Length Max is determined by the parameter $\Psi(\mathrm{BWLM})$, instead of the regular $\Psi(\mathrm{MWLM})$ used for main walks. This case is still being considered as recursive because it calls $\vdash_{branch}$ in the internal judgment $\vdash_{branch-walk}$ for the sub-walk, but it's not recursive on the current walk, like the previous case is.

$$\frac{\begin{array}{c}(\text{iden}, \cdot, \cdot, \cdot) \equiv \text{nd?} \\ \alpha > \Psi(\text{BNT}) \\ (\text{w}_{acc} > 0) \vee (\text{typ} \notin (\{\neg()\} \cup \text{b})) \\ \Psi \text{ ; } \Theta \text{ ; b ; typ ; d ; w}_{acc} \vdash_{branch-walk} (\text{n-id}, \neg\vec{\tau}, \text{comps}) \text{ ; w}'_{acc} \\ \text{next} = \text{first . first . } \xi(\text{filter ((c, t)} \rightarrow \text{t} \equiv \text{typ) (zip comps } (\neg\vec{\tau} : \vec{\tau}))) \\ \text{comps}' := \text{map ((i, } \cdot\text{, } \cdot\text{, n)} \rightarrow \text{(i, } \cdot\text{, ((i} \equiv \text{next) ? iden : } \cdot\text{), n)) comps} \\ \text{child} := (\text{iden}, \ \cdot, \ \cdot, \ (\text{n-id}, \ \neg\vec{\tau}, \ \text{comps}')) \\ \text{nd}' := (\text{iden}, \cdot, \cdot, \text{child})\end{array}}{\Psi \text{ ; } \alpha \text{ ; } \Theta \text{ ; b ; } succ(\text{d}) \text{ ; typ ; w}_{acc} \text{ ; nd?} \vdash_{branch-comp} \text{nd}' \text{ ; w}'_{acc}}$$

Lastly, it's required to define the new internal judgment $\vdash_{branch-walk}$. It works similarly to the $\vdash_{walk}$ judgment previously defined, but it stops after its $\vdash_{branch}$ operation, and there's also no discard case, as the branched walk will always have at least one step. Therefore, it only has a single rule, which is defined below:

$$\frac{\begin{array}{c}\hat{\Theta} := (\text{w}_{acc} > 0) \text{ ? } \Theta : (\text{map } ((\text{k}, (\Theta_F, \ \Theta_B)) \rightarrow (\text{k}, (\{\emptyset\}, \ \Theta_B))) \ \Theta) \\ \text{w} := \xi(\{n \in \mathbb{N}^+ \mid n \leq \Psi(\text{BWLM})\} \\ \Psi \text{ ; } \hat{\Theta} \text{ ; } \tau \text{ ; } \cdot \text{ ; w} \vdash_{steps} \text{stps} \\ \Psi \text{ ; } \cdot \text{ ; } nil \text{ ; } \cdot \text{ ; stps} \vdash_{assemble} \text{assembled} \\ \Psi \text{ ; } \xi(\mathbb{R}^u) \text{ ; w}_{acc} - \lceil \frac{\text{length stps}}{2} \rceil \text{ ; assembled} \vdash_{merge} \text{merged ; w}'_{acc} \\ \Psi \text{ ; } \hat{\Theta} \text{ ; d ; w}'_{acc} \text{ ; merged} \vdash_{branch} \text{branched ; w}''_{acc}\end{array}}{\Psi \text{ ; } \Theta \text{ ; b ; } \tau \text{ ; d ; w}_{acc} \vdash_{branch-walk} \text{branched ; w}'_{acc}}$$

4.6.5 Assign a free variable into some components ($\vdash_{void}$)

Due to the way $\vdash_{cmd}$ operates on walks, if a component does not have a variable assigned to it at the time its node is being processed, then either the algorithm is going to fetch an already defined variable of the corresponding type from the variable environment and assign it to the component, or it's going to create a new binding, or sometimes it's going to use a free variable. It's rare that it uses a free variable, as that should only happen under a few select conditions. It may be interesting, however, to assign a free variable to a component during the walk generation, so that there is a greater variety of places free variables may be encountered. This is what $\vdash_{void}$ does: it assigns free variables to empty components.

This operation's likelihood is controlled by the $\Psi(\text{VIT})$ parameter. Performing this very frequently may have substantial impacts on the type-checker's ability to infer specific types, so it was decided to decrease this likelihood by both stimulating the threshold to be a high value and to set it at a minimum of 0.75 (meaning at most the void operation will have a 25% chance of being applied).

The $\vdash_{void}$ judgment takes the following parameters:

- Ψ: Associative list of parameters.

- node?: A node. This is the parameter the recursion is based on.

And it returns a potentially modified node.

This operation is also separated in node-level and component-level judgments. The node-level judgment just calls the component-level one for all of its components, replacing the old components with potentially modified ones:

$$\frac{\begin{array}{c}(\text{comp}_0 : \ldots : \text{comp}_n : nil) := \text{comps} \\ \Psi\ ;\ \xi(\mathbb{R}^u)\ ;\ \text{comp}_i \vdash_{void-comp} \text{comp}'_i \qquad \forall i \in \mathbb{N}\ .\ 0 \leq i \leq \text{n} \\ \text{comps}' := (\text{comp}'_0 : \ldots : \text{comp}'_n : nil)\end{array}}{\Psi\ ;\ (\text{iden, typ, comps}) \vdash_{void}\ (\text{iden, typ, comps}')}$$

And now the judgment that acts at component-level. It takes these parameters:

- Ψ: Associative list of parameters.
- α: Uniformly sampled score between 0 and 1. This is compared with $\Psi(\text{VIT})$ to decide whether or not the void operation is going to be applied.
- comp: A component. This is the parameter the recursion is based on.

And now the cases, of which there are three. First, if $\alpha > \Psi(\text{VIT})$ and the current component has $\cdot$ for the variable name, previous component and next node, i.e., it is an empty component, the void operation is performed by assigning a fresh free variable to the variable field. Notice that after the procedure, the component stops being an empty component, but it's still a side component. This is a base case, and it is defined like this:

$$\frac{\alpha > \Psi(\text{VIT})}{\Psi\ ;\ \alpha\ ;\ (\text{iden}, \cdot, \cdot, \cdot) \vdash_{void-comp} (\text{iden}, \xi_{id}, \cdot, \cdot)}$$

A second base case is representative of when there is no child node, but either the component is not an empty one or $\alpha \leq \Psi(\text{VIT})$, and therefore a void operation is not done:

$$\frac{\begin{array}{c}(\text{nd?} \equiv \cdot) \vee (\text{nd?} \equiv \circ) \\ \text{score-fail} := \alpha \leq \Psi(\text{VIT}) \\ \text{empty-fail} := (\text{vn?} \not\equiv \cdot) \vee (\text{pv?} \not\equiv \cdot) \vee (\text{nd?} \not\equiv \cdot) \\ \text{score-fail} \vee \text{empty-fail}\end{array}}{\Psi\ ;\ \alpha\ ;\ (\text{iden, vn?, pv?, nd?}) \vdash_{void-comp} (\text{iden, vn?, pv?, nd?})}$$

The recursive case happens whenever there is a child node, so it just continues along the child node, calling the node-level $\vdash_{void}$ and leaving the current component's vn? unchanged:

$$\frac{\begin{array}{c}(\text{n-id, typ, comps}) \equiv \text{nd?} \\ \Psi\ ;\ \text{nd?} \vdash_{void} \text{nd}'\end{array}}{\Psi\ ;\ \alpha\ ;\ (\text{iden, vn?, pv?, nd?}) \vdash_{void-comp} (\text{iden, vn?, pv?, nd}')}$$

4.6.6 Flatten the recursive node structure ($\vdash_{transform}$)

The structure of the walk is going to change again here. Instead of the nodes being present in a recursively defined structure, their relationship will now be flattened. Specifically, now the walk will become a 3-tuple, where the first two elements are associative lists for nodes and components and the last element is a dependency graph of nodes, represented as a collection of edges. This graph is a DAG, and the algorithm not getting stuck relies on this fact.

It makes more sense to first explain the graph creation. If previously a certain node X stores within itself a node Y in one of its components, then (Y,X) $\in$ G, where G is the new graph. The reversal of the expected edge direction comes from the way the algorithm processes walks: it starts from one of its ending vertices (only one, if no branching was done) and goes backwards from that until it reaches the starting vertex of the walk. Throughout this work, this graph may be handled as if it's represented as a list of edges or adjacency lists. The adjacency list approach is the more theoretically precise, as isolated nodes still exist and they can't be represented using a list of edges alone. The processing order of nodes in this walk will follow a valid topological sorting of this graph.

The first associative list contains all of this walk's nodes, mapping an internal node identifier to a 4-tuple containing the following elements:

1. walk-id: An unique identifier for the walk this node belongs to. This identifier is created at this step and is replicated across all nodes.
2. typ: The type of the subject continuation.
3. node-comps: Instead of saving the entire components, like was done up until this point, this is only a list of their unique identifiers in order.
4. ready?: A boolean that indicates whether this node is able to be processed at this point. This is only true iff processing this node next would respect a valid topological sorting of the graph.

The second associative list contains all of this walk's components, mapping an internal component identifier to a 6-tuple containing the following elements:

1. node-ident: The identifier of the node this component belongs to.
2. varname: The name of the variable this component is going to use. If no variable has been associated, the value is $\cdot$.
3. comp-typ: The type this component's variable needs to have.
4. depended-by: Set of tuples. The first element in each tuple is the identifier of a component in a different node that is waiting for the current component's node to be executed in order

for them to be labeled ready. The second element is a boolean flag that indicates whether this dependency is intrinsic ($\top$) or extrinsic ($\bot$). At this point, all dependencies are going to be intrinsic, which means they derive from the steps in the walk, instead of being obtained artificially through future transformations. More on the differences between these two types of dependencies in the next subsection.

5. depends-on: Set of tuples. Similar to depended-by, but this one stores the identifier and boolean flag corresponding to a component the current node is waiting for. A node's ready? flag is obtained by checking whether this set is empty for all of its components.

6. collideable?: A boolean that indicates whether this component is able to be subjected to a Collide operation. This operation is explained in $\vdash_{cmd}$. This is going to be true iff this component is an empty component.

The components in depended-by and depends-on are at this point always of the same type as the current component, because these dependencies originate from the prev field, which always points to a component of the same type in the immediately previous node in the walk. This pattern of dependencies always having the same type may or may not continue throughout the walk generation as new dependencies are added to this set, depending on the value of the parameter $\Psi(\text{ADTT})$ (Allow Different Types Threshold). More on that parameter later.

The main advantage of forcing dependencies to be of the same type is that new variables of that type may be introduced into the environment and may be used by the components that depended on the one whose node introduced this new variable. However, neither that introduction nor its usage is guaranteed.

The main advantage of allowing dependencies to be of different types is to create a more diverse system of branch execution blockage, where any component can block the execution of any other component.

After this step is done, its output should reflect the following updated grammar:

$$
\begin{aligned}
\text{walk} &::= (((\text{iden node}) ...+) ((\text{iden component}) ...+) ((\text{iden } ...+) ...)) \\
\text{node} &::= (\text{iden typ (iden } ...+) \text{ bool}) \\
\text{component} &::= (\text{iden iden? typ ((iden bool) } ...) ((\text{iden bool})) \text{ bool}) \\
\text{typ} &::= \text{iden} \mid \neg(\text{typ } ...) \\
\text{iden?} &::= \text{iden} \mid \cdot \\
\text{iden} &::= \text{unique-identifier} \\
\text{bool} &::= \top \mid \bot
\end{aligned}
$$

And the main judgment below. It uses two internal judgments, one to find the graph edges and the other to create the associative lists, and the graph is created by joining these edges

with the node-level associative list's keys. As $\vdash_{DAG}$ only finds edges, isolated vertices would be deleted from the graph if nothing additional is done, and so to make sure every node has a corresponding vertex, the edges are transformed into an adjacency associative list where every node is mapped to a list containing the vertices it has an edge with, with that list being empty in the case of isolated nodes. The rule is below:

$$\frac{\begin{array}{c}\text{nd} \vdash_{DAG} \text{edges} \\ \text{w-id ; nd} \vdash_{flatten} \text{nodes ; comps} \\ \text{adj-assoc} := \text{map } (((\text{src, dst}) : \text{tl}) \rightarrow (\text{src, dst} : (\text{map second tl}))) \ (\text{group-by first edges}) \\ \text{G} := \text{map } (\text{k} \rightarrow (\text{k, } ((\text{k} \in \text{adj-assoc}) \ ? \ \text{adj-assoc(k)} : nil))) \ (\text{keys nodes})\end{array}}{\text{w-id ; nd} \vdash_{transform} (\text{nodes, comps, G})}$$

The first internal judgment, $\vdash_{DAG}$, also calls its own internal judgment, $\vdash_{DAG-comp}$. These operate at a node-level and component-level, respectively. Both use a graph union operation, which first concatenates all of the edges and then removes duplicates.

The node-level judgment takes just a node as a parameter, and it returns a graph represented as a set of edges. The input is actually a node as defined in the old grammar, so it's still a recursive structure. This judgment's single rule is defined below:

$$\frac{\text{ident ; comp}_i \vdash_{DAG-comp} \text{G}_i \ , \ \forall i \in \mathbb{N} \ . \ 1 \leq i \leq n}{(\text{ident, typ, } (\text{comp}_0 : \ldots : \text{comp}_n : nil)) \vdash_{DAG} \bigcup_{i=1}^{n} \text{G}_i}$$

Now for the component-level operation. It takes a node identifier and a component as an input, and it outputs a graph as a set of edges. Like the node-level judgment, the component provided as input here is also as defined in the previous grammar, so it's still representative of a recursive structure.

The component-level judgment has two rules, one for when there is a child node and the other for when there isn't. If there is a child node, then it calls the node-level judgment again, adding an edge to the output graph if that edge is not present already. This is the recursive case, and it's defined below:

$$\frac{(\text{inner, typ, comps}) \vdash_{DAG} \text{G}}{\text{outer ; (iden, vn?, iden?, (inner, typ, comps))} \vdash_{DAG-comp} \{(\text{inner, outer})\} \cup \text{G}}$$

And now for the base case. If there isn't a child node, then simply return an empty set:

$$\frac{(\text{nd?} \equiv \cdot) \vee (\text{nd?} \equiv \circ)}{\text{outer ; (iden, vn?, iden?, nd?)} \vdash_{DAG-comp} \{\emptyset\}}$$

The second internal judgment of $\vdash_{transform}$, $\vdash_{flatten}$, also has its own internal judgment meant to process components. The node-level operation gets as input an old node and a walk identifier and outputs associative lists for both nodes and components. It does this by calling its

internal judgment for every one of the node's components, which returns not only an updated component, but also associative lists representative of that component's recursive structure. It then joins these associative lists and adds one new entry in the nodes one and the updated components in the other, now reflecting the updated grammar. Its rule is defined below:

$$\frac{\begin{array}{c}\text{(ident, } \neg\vec{\tau}\text{, (comp}_0 : \ldots : \text{comp}_n : nil)) \equiv \text{node} \\ \text{walk-id ; comp}_i \text{ ; } \tau \text{ ; ident} \vdash_{flatten-comp} \text{comp}'_i \text{ ; n-assoc}'_i \text{ ; c-assoc}'_i \\ \forall(\tau, i) \in (\text{zip } (\neg\vec{\tau} : \vec{\tau})\ [0..{=}n]) \\ \text{comps}' := (\text{comp}'_0 : \ldots : \text{comp}'_n : nil) \\ \text{comps}'\text{-ids} := (\text{map first comps}') \\ \text{comps}'\text{-vals} := (\text{map second comps}') \\ \text{ready?} := \bigwedge_{\text{(nid, vn, ctyp, d-by, depends-on, c?)} \in \text{comps}'\text{-vals}} \text{depends-on} \equiv \{\emptyset\} \\ \text{nodes-assoc} := (\{(\text{ident, (walk-id, } \neg\vec{\tau}\text{, comps}'\text{-ids, ready?}))\} \cup (\bigcup_{i=0}^{n} \text{n-assoc}'_i)) \\ \text{comps-assoc} := ((\text{to-set comps}') \cup (\bigcup_{i=0}^{n} \text{c-assoc}'_i))\end{array}}{\text{walk-id ; node} \vdash_{flatten} \text{nodes-assoc ; comps-assoc}}$$

The component-level judgment gets a component, its type, the node identifier and the walk identifier. It returns, in this order, the processed component as a suitable member of the component associative list and the outputs of a recursive call to $\vdash_{flatten}$ using the child node, those being the associative list for nodes and the associative list for components. If there is no child node, return instead two empty lists as the second and third return output. This judgment's recursive rule is as follows and maps the case in which there is a child node:

$$\frac{\begin{array}{c}\text{(outer, vn?, pv?, (inner, typ, child-comps))} \equiv \text{comp} \\ \text{walk-id ; (inner, typ, child-comps)} \vdash_{flatten} \text{nodes-assoc ; comps-assoc} \\ \text{depended-by} := (\text{pv?} \equiv \cdot)\ ?\ \{\emptyset\} : \{(\text{pv}, \top)\} \\ \text{depends-on} := \{((\text{getPrevComp child-comps}), \top)\} \\ \text{comp}' := (\text{outer, (node-id, vn?, } \tau\text{, depended-by, depends-on, } \bot))\end{array}}{\text{walk-id ; comp ; } \tau \text{ ; node-id} \vdash_{flatten-comp} \text{comp}' \text{ ; nodes-assoc ; comps-assoc}}$$

It uses the following auxiliary function, which is very similar to the already defined findPrev, returning the identifier of a component that has the prev field set. Notice it does not define a case for when there is no prev different than $\cdot$, and that's because getPrevComp is always called on a list that represents a node's child components, so it's never going to be called on the first node's components, the only case where it would not find any prev:

getPrevComp :: [component] → iden

getPrevComp ((_ _ · _) : cs) = getPrevComp cs

getPrevComp ((ident _ prev _) : _) = ident

Now going back to the judgment. It has one base case, which still updates the component, but now returns empty lists as the two last outputs. The new component's depends-on field is

set to empty, as there is no child node, and instead of simply setting the collideable? flag to $\perp$, it makes a check to see if the current component is an empty component or not. The check is as expected, returning $\top$ iff $(\text{vn?} \equiv \cdot) \wedge (\text{pv?} \equiv \cdot) \wedge (\text{nd?} \equiv \cdot)$, and $\perp$ otherwise. Note that if a component in the last node of the walk has had its nd? set to $\circ$, this component will also have $\perp$ in the collideable? field. The rule for this base case is as follows:

$$\frac{\begin{array}{c}\text{(outer, vn?, pv?, nd?)} \equiv \text{comp} \\ (\text{nd?} \equiv \cdot) \vee (\text{nd?} \equiv \circ) \\ \text{depended-by} := (\text{pv?} \equiv \cdot)\ ?\ \{\emptyset\} : \{(\text{pv},\ \top)\} \\ \text{collideable?} := ((\text{vn?} \equiv \cdot) \wedge (\text{pv?} \equiv \cdot) \wedge (\text{nd?} \equiv \cdot)) \\ \text{comp}' := \text{(outer, (node-id, vn?, } \tau\text{, depended-by, } \{\emptyset\}\text{, collideable?))}\end{array}}{\text{walk-id ; comp ; } \tau \text{ ; node-id} \vdash_{flatten-comp} \text{comp}' \text{ ; } nil \text{ ; } nil}$$

4.6.7 Intermission: Intrinsic x Extrinsic Dependencies

Before following on to the next transformation, it may be worth to define what are intrinsic and extrinsic dependencies in the context of this algorithm.

A component's **intrinsic** dependencies are determined by the recently flattened recursive structure of nodes. A node A containing another node B defines an intrinsic dependency, which is brought to the component-level by relating which of A's components has B and which of B's components has the *prev?* variable set to the identifier of the parent component from A[5].

On the other hand, **extrinsic** dependencies are derived from transformations outside of that recursive definition of a node, and are therefore optional additions to the dependency system, rather than being a defining characteristic of it or of the walk's graph structure as given by the graph obtained from $\vdash_{DAG}$.

As opposed to intrinsic dependencies, extrinsic ones can happen at any distance, which means that a component from node A can create an extrinsic dependency between itself and a component from node B, regardless of how far away these nodes were from one another in the recursive node structure, or even if there was a path to reach one starting from the other at all. On that last point, only extrinsic dependencies can create relationships between nodes from two different branches or two different walks entirely.

Conditions on extrinsic dependencies will be detailed later on when discussing the transformations that can create them, but one that is common among all of them and that should be anticipated is the restriction that two components can only add an extrinsic dependency between each other **if the addition of an edge representing this dependency at node-level will maintain the acyclic property of the walk environment's graph**. Other conditions could be lifted or changed to achieve different transformations, but not following this restriction will cause the algorithm to be stuck.

[5] Actually comparing the *prev?* variable to the parent component from A is unnecessary, as a node can have at most one component with a *prev?* different than $\cdot$

4.6.8 Create extrinsic dependencies within the same walk ($\vdash_{fold}$)

The fold transformation creates extrinsic dependencies between different components belonging to the same walk. By creating these dependencies, the algorithm may force an entire section of the walk to only start to execute when another section finishes. It gets the following parameters:

- Ψ: Associative list of parameters.
- α: Random score sampled from $\alpha \sim \mathbb{U}[0, 1]$. Compared with $\Psi(\text{FLT})$ to determine if a fold is going to happen or not.
- Δ_C: Associative list of components.
- Δ_G: Graph representing the walk's node-level dependencies.
- comps: An iterable containing the remaining components to be processed by this transformation. The recursion is based on this parameter.

And it returns two values:

- Δ'_C: The potentially modified associative list of components, with any new dependencies added.
- Δ'_G: The potentially modified graph, with new edges representing any new dependencies at a node-level.

First the base case, triggered when there are no remaining components to be processed. It simply outputs the Δ_C and Δ_G:

$$\frac{\text{comps} \equiv \{\emptyset\}}{\Psi \text{ ; } \alpha \text{ ; } \Delta_C \text{ ; } \Delta_G \text{ ; comps} \vdash_{fold} \Delta_C \text{ ; } \Delta_G}$$

And now the first recursive case. This maps the event when the fold does not happen due to the α score not exceeding the threshold:

$$\frac{\begin{array}{c}\text{comps} \not\equiv \{\emptyset\} \\ \text{(comp-id, (node-id, varname, typ depended-by depends-on collide?))} = \xi(\text{comps}) \\ \text{next-}\alpha = \xi(\mathbb{R}^u) \\ \alpha \leq \Psi(\text{FLT}) \\ \Psi \text{ ; next-}\alpha \text{ ; } \Delta_C \text{ ; } \Delta_G \text{ ; (remove comp-id comps)} \vdash_{fold} \Delta'_C \text{ ; } \Delta'_G\end{array}}{\Psi \text{ ; } \alpha \text{ ; } \Delta_C \text{ ; } \Delta_G \text{ ; comps} \vdash_{fold} \Delta'_C \text{ ; } \Delta'_G}$$

And now the second recursive case, where the fold may happen or not. It's always possible for the number of new dependencies to be added (num-deps) to get the value 0, which will result in a fold in practice not happening, but the operation remains the same regardless of the value num-deps has. Several conditions need to be simultaneously met in order for a component to be deemed as eligible for the current component to depend on. These conditions are:

- The inclusion of an edge of form (other, current) to the graph must maintain its acyclic property. This requirement is not flexible.
- The other component's variable field must not be set.
- The number of components that currently depend on the other component must be strictly less than $\Psi(\text{MDBA})$ (Max Depended-By Allowed).
- One of the following must be true: either an uniformly sampled $\alpha \sim \mathbb{U}[0, 1]$ is valued such that $\alpha > \Psi(\text{ADTT})$ (Allow Different Types Threshold) or the other component's type is the same as the current's.

This case also uses the $\Psi(\text{FLMD})$ (Fold Max Dependencies) parameter, which tells a maximum number of dependencies that can be created by one step of the fold operation. The rule is:

$$\frac{\begin{array}{c}\text{comps} \not\equiv \{\emptyset\} \\ \text{(comp-id, (node-id, varname, typ, depended-by, depends-on, collide?))} = \xi(\text{comps}) \\ \text{next-}\alpha = \xi(\mathbb{R}^u) \\ \alpha > \Psi(\text{FLT}) \\ \text{fn} := ((\text{c-id}, (\text{n-id, vn?, t, dep-by, dep-on, col?})) \to ((\text{isDAG? } (\{(\text{n-id, node-id})\} \cup \Delta_G)) \\ \wedge\ (\text{vn?} \equiv \cdot) \wedge (|\text{dep-by}| < \Psi(\text{MDBA})) \wedge ((\xi(\mathbb{R}^u) > \Psi(\text{ADTT})) \vee (\text{t} \equiv \text{typ})))) \\ \text{candidates} := \text{filter fn } \Delta_C \\ \text{num-deps} := \xi([0..=(\max(0, \min(|\text{candidates}|, \Psi(\text{FLMD}), (\Psi(\text{MDOA}) - |\text{depends-on}|))))]) \\ \text{chosen} = \xi(\text{candidates, num-deps}) \\ \text{new-deps} := \text{map } ((\text{c-id}, (\text{n-id, vn?, t, dep-by, dep-on, col?})) \to (\text{c-id}, \bot)) \text{ chosen} \\ \text{new-edges} := \text{map } ((\text{c-id}, (\text{n-id, vn?, t, dep-by, dep-on, col?})) \to (\text{n-id, node-id})) \text{ chosen} \\ \text{fold-fn} := (\text{d ; acc} \to \text{update d acc } ((\text{n-id, vn, t, dep-by, dep-on, col?}) \\ \to (\text{n-id, vn, t, (dep-by} \cup \{(\text{comp-id}, \bot)\}), \text{dep-on, col?}))) \\ \text{first-acc} := \text{update comp-id } \Delta_C \text{ ((n-id, vn, t, dep-by, dep-on, col?)} \\ \to (\text{n-id, vn, t, dep-by, (dep-on} \cup \text{new-deps), col?})) \\ \Delta'_C := \text{fold fold-func first-acc (map first new-deps)} \\ \Delta'_G := \text{new-edges} \cup \Delta_G \\ \Psi \text{ ; next-}\alpha \text{ ; } \Delta'_C \text{ ; } \Delta'_G \text{ ; (remove comp-id comps)} \vdash_{fold} \Delta''_C \text{ ; } \Delta''_G\end{array}}{\Psi \text{ ; } \alpha \text{ ; } \Delta_C \text{ ; } \Delta_G \text{ ; comps} \vdash_{fold} \Delta''_C \text{ ; } \Delta''_G}$$

4.6.9 Create extrinsic dependencies between different walks ($\vdash_{reach}$)

This transformation runs through each component in the current walk and tries to create a dependency relation between it and another one from a different walk. As opposed to $\vdash_{fold}$, which can be defined with just one judgment, $\vdash_{reach}$ needs two: one for when the component in the current walk *depends on* a component from outside the walk ($\vdash_{borrow}$), and the other for when it is *depended by* one ($\vdash_{lend}$).

This judgment also has a second purpose coupled together with the above: as the last transformation, it also joins the new components and graph into the walk environment, integrating the new walk into the scope. The nodes will be integrated outside of this judgment.

The upper-level $\vdash_{reach}$ judgment gets the following parameters:

- Ψ: Associative list of parameters.
- Δ_W: Associative list of walks from the walk environment. This list **should** already contain the new walk.
- Δ_N: Associative list of nodes from the walk environment. It does not matter whether the current walk's nodes have already been added to it or not. In this case, they have not. This judgment will also not add the new nodes to the list, so that needs to be done after $\vdash_{reach}$ returns.
- Δ_C: Associative list of components from the walk environment. This list should **not** contain the current walk's components, as they will be added at the end of the recursive process.
- Δ_G: Current graph representing the walk environment's dependencies. This graph **should** already be given as an input joined with the current walk's graph.
- w-id: Identifier of the current walk.
- comps-acc: An accumulator for processed components. At the base case of the recursion, this parameter is going to be added to Δ_C. This can't be done step-by-step because it would impact the recursive process by allowing a $\vdash_{fold}$-like operation between two components of the same walk, where one is already processed and added to the environment and the other is currently processed. To avoid that, this accumulator is introduced.
- comps: An iterable containing the remaining components to be processed by this transformation. The recursion is based on this parameter.

And it returns three values:

- Δ'_W: The updated associative list of walks, including any new dependencies that may have been added.

- Δ'_C: The updated associative list of components, containing the new dependencies added, if any, and all the current walk's components.
- Δ'_G: The updated graph, with new edges representing new dependencies at a node-level.

First its base case, when there are no more components from the current walk to process. It joins the accumulator with the walk environment's components.

$$\frac{\Delta'_C := \text{comps-acc} \cup \Delta_C}{\Psi \ ; \ \Delta \ ; \ \text{w-id} \ ; \ \text{comps-acc} \ ; \ \{\emptyset\} \vdash_{reach} \Delta_W \ ; \ \Delta'_C \ ; \ \Delta_G}$$

And now the recursive step. It first calls both $\vdash_{borrow}$ and $\vdash_{lend}$ for the current component, and then does a recursive call:

$$\frac{\begin{gathered}\text{comps} \not\equiv \{\emptyset\} \\ \text{comp} = \xi(\text{comps}) \\ \Psi \ ; \ \xi(\mathbb{R}^u) \ ; \ \text{w-id} \ ; \ \text{comp} \ ; \ \Delta \vdash_{borrow} \text{comp}' \ ; \ \Delta'_W \ ; \ \Delta'_C \ ; \ \Delta'_G \\ \Delta'_N := \Delta_N \\ \Psi \ ; \ \xi(\mathbb{R}^u) \ ; \ \text{w-id} \ ; \ \text{comp}' \ ; \ \Delta' \vdash_{lend} \text{comp}'' \ ; \ \Delta''_W \ ; \ \Delta''_C \ ; \ \Delta''_G \\ \Delta''_N := \Delta'_N \\ \text{comps-acc}' := (\text{comp}'' : \text{comps-acc}) \\ \text{comps}' := \text{remove comp comps} \\ \Psi \ ; \ \Delta'' \ ; \ \text{w-id} \ ; \ \text{comps-acc}' \ ; \ \text{comps}' \vdash_{reach} \Delta'''_W \ ; \ \Delta'''_C \ ; \ \Delta'''_G\end{gathered}}{\Psi \ ; \ \Delta \ ; \ \text{w-id} \ ; \ \text{comps-acc} \ ; \ \text{comps} \vdash_{reach} \Delta'''_W \ ; \ \Delta'''_C \ ; \ \Delta'''_G}$$

Now to define $\vdash_{borrow}$ and $\vdash_{lend}$. Starting with $\vdash_{borrow}$, this makes it so that the component from the current walk depends on components from other walks. It gets as parameters:

- Ψ: Associative list of parameters.
- α: Random score sampled from $\alpha \sim \mathbb{U}[0, 1]$. Compared with $\Psi(\text{BRT})$ to determine if a borrow is going to happen or not.
- w-id: Identifier of the current walk.
- comp: Component from the current walk that is being analyzed.
- Δ: Current walk environment.

And it returns potentially modified values for the comp, Δ_W, Δ_C and Δ_G parameters.

It has two cases, one for when $\alpha > \Psi(\text{BRT})$ and the other for when that's false. Starting with $\alpha \leq \Psi(\text{BRT})$, resulting in the parameters being returned unchanged, with the borrow operation not happening:

$$\frac{\alpha \leq \Psi(\text{BRT})}{\Psi \text{ ; } \alpha \text{ ; w-id ; comp ; } \Delta \vdash_{borrow} \text{comp ; } \Delta_W \text{ ; } \Delta_C \text{ ; } \Delta_G}$$

The other rule, as mentioned, happens when $\alpha > \Psi(\text{BRT})$. In that case, a process extremely similar to $\vdash_{fold}$ takes place, with the only differences being that it's not recursive, that the possible candidate components to borrow from (to depend on) are on the disjoint walk environment instead of on the current walk, and that Δ_W is used to limit the number of different walks being connected via dependencies. The rule is as follows:

$$\frac{\begin{array}{c}\alpha > \Psi(\text{BRT}) \\ \text{(comp-id, (node-id, var, typ, depended-by, depends-on, collide?))} \equiv \text{comp} \\ \text{fn} := (\text{(c-id, (n-id, vn?, t, dep-by, dep-on, col?))} \rightarrow (\text{(isDAG? } (\{\text{(n-id, node-id)}\} \cup \Delta_G)) \\ \wedge (\text{vn?} \equiv \cdot) \wedge (|\text{dep-by}| < \Psi(\text{MDBA})) \wedge ((\xi(\mathbb{R}^u) > \Psi(\text{ADTT})) \vee (\text{t} \equiv \text{typ})) \\ \wedge (|\Delta_W(\text{first } \Delta_N(\text{n-id})) \cup \Delta_W(\text{w-id})| \leq \Psi(\text{MCWA})))) \\ \text{candidates} := \text{filter fn } \Delta_C \\ \text{num-deps} := \xi([0..=(\max(0, \min(|\text{candidates}|, \Psi(\text{BRMD}), (\Psi(\text{MDOA}) - |\text{depends-on}|))))]) \\ \Psi \text{ ; w-id ; } \Delta_W \text{ ; } \Delta_N \text{ ; candidates ; num-deps} \vdash_{select-while-filtering} \Delta'_W \text{ ; chosen} \\ \text{new-deps} := \text{map ((c-id, (n-id, vn, t, dep-by, dep-on, col?)) } \rightarrow \text{ (c-id, } \perp)) \text{ chosen} \\ \text{new-edges} := \text{map ((c-id, (n-id, vn, t, dep-by, dep-on, col?)) } \rightarrow \text{ (n-id, node-id)) chosen} \\ \text{comp}' := \text{(comp-id, (node-id, var, typ, depended-by, (depends-on } \cup \text{ new-deps), collide?))} \\ \text{fold-fn} := \text{(d ; acc } \rightarrow \text{ update d acc ((n-id, vn, t, dep-by, dep-on, col?)} \\ \rightarrow \text{ (n-id, vn, t, (dep-by } \cup \{\text{(comp-id, } \perp)\}), \text{ dep-on, col?}))) \\ \Delta'_C := \text{fold fold-fn } \Delta_C \text{ (map first new-deps)} \\ \Delta'_G := \text{new-edges} \cup \Delta_G\end{array}}{\Psi \text{ ; } \alpha \text{ ; w-id ; comp ; } \Delta \vdash_{borrow} \text{comp}' \text{ ; } \Delta'_W \text{ ; } \Delta'_C \text{ ; } \Delta'_G}$$

Now defining $\vdash_{lend}$. It operates similarly to $\vdash_{borrow}$, but with the cardinality of the dependencies reversed. The current component is now not looking to depend on other components, but to make other components depend on it. The input parameters are identical to $\vdash_{borrow}$, and so are the outputs. The list will therefore be omitted for sake of brevity.

The first case is triggered when either $\alpha \leq \Psi(\text{LNT})$ or the component has a variable set, in which case there is no lend. The condition related to the variable is putting what was once a condition for the candidates on the current component. This first case is defined like this:

$$\frac{\begin{array}{c}\text{(comp-id, (node-id, var, typ, depended-by, depends-on, collide?))} \equiv \text{comp} \\ (\alpha \leq \Psi(\text{LNT})) \vee (\text{var} \not\equiv \cdot)\end{array}}{\Psi \text{ ; } \alpha \text{ ; w-id ; comp ; } \Delta \vdash_{lend} \text{comp ; } \Delta_W \text{ ; } \Delta_C \text{ ; } \Delta_G}$$

And the second case when $\alpha > \Psi(\text{LNT})$ and $\text{var} \equiv \cdot$. It is in some respects symmetrical to what was done in $\vdash_{borrow}$. For example, the previous filter condition that checks in $\vdash_{fold}$ and $\vdash_{borrow}$ if $(|\text{dep-by}| < \Psi(\text{MDBA}))$ is replaced here with $(|\text{dep-on}| < \Psi(\text{MDOA}))$, and the edges being added have had their direction reversed as well. Other than these types of change, nothing is different:

$$\frac{\begin{array}{c}\text{(comp-id, (node-id, var, typ, depended-by, depends-on, collide?))} \equiv \text{comp} \\ \alpha > \Psi(\text{LNT}) \\ \text{var} \equiv \cdot \\ \text{fn} := ((\text{c-id, (n-id, vn?, t, dep-by, dep-on, col?))} \rightarrow ((\text{isDAG? } (\{(\text{node-id, n-id})\} \cup \text{G})) \\ \wedge (|\text{dep-on}| < \Psi(\text{MDOA})) \wedge ((\xi(\mathbb{R}^u) > \Psi(\text{ADTT})) \vee (\text{t} \equiv \text{typ})))) \\ \wedge (|\Delta_W(\text{first } \Delta_N(\text{n-id})) \cup \Delta_W(\text{w-id})| \leq \Psi(\text{MCWA})))) \\ \text{candidates} := \text{filter fn } \Delta_C \\ \text{num-deps} := \xi([0..=(\max(0, \min(|\text{candidates}|, \Psi(\text{LNMD}), (\Psi(\text{MDBA}) - |\text{depended-by}|))))]) \\ \Psi \text{ ; w-id ; } \Delta_W \text{ ; } \Delta_N \text{ ; candidates ; num-deps} \vdash_{select-while-filtering} \Delta'_W \text{ ; chosen} \\ \text{new-deps} := \text{map ((c-id, (n-id, vn, t, dep-by, dep-on, col?)) } \rightarrow \text{ (c-id, } \perp)) \text{ chosen} \\ \text{new-edges} := \text{map ((c-id, (n-id, vn, t, dep-by, dep-on, col?)) } \rightarrow \text{ (node-id, n-id)) chosen} \\ \text{comp}' := \text{(comp-id, (node-id, var, typ, (depended-by } \cup \text{ new-deps), depends-on, collide?))} \\ \text{fold-fn} := \text{(d ; acc } \rightarrow \text{ update d acc ((n-id, vn, t, dep-by, dep-on, col?)} \\ \rightarrow \text{ (n-id, vn, t, dep-by, (dep-on } \cup \{(\text{comp-id}, \perp)\}), \text{col?}))) \\ \Delta'_C := \text{fold fold-fn } \Delta_C \text{ (map first new-deps)} \\ \Delta'_G := \text{new-edges} \cup \Delta_G \end{array}}{\Psi \text{ ; } \alpha \text{ ; w-id ; comp ; } \Delta \vdash_{lend} \text{comp}' \text{ ; } \Delta'_W \text{ ; } \Delta'_C \text{ ; } \Delta'_G}$$

And lastly, the auxiliary judgment $\vdash_{select-while-filtering}$. This judgment does something similar to ξ(collection, number), but after each selection it also filters the collection variable to remove entries that have recently become invalid. The filter condition is specified to be the last condition in the fn function in both $\vdash_{borrow}$ and $\vdash_{lend}$, therefore requiring that adding multiple components will not lead to the number of distinct walks that are connected by their nodes in Δ_G exceeding the limit Ψ(MCWA). The judgment takes the following parameters:

- Ψ: Associative list of parameters.
- w-id: The current walk identifier.
- Δ_W: The associative list mapping walk names to a set of walks that are connected to it, including itself.
- Δ_N: The associative list of nodes in the walk environment.
- cands: Collection of candidates the algorithm is going to sample from.
- x: Natural that indicates how many more candidates should be sampled. The recursion is based on this parameter.

And it returns a potentially updated Δ_W and a randomly selected subset of elements from candidates.

First the base case, which happens when either x $\equiv zero$ or when candidates $\equiv \{\emptyset\}$. In this case, simply return the inputted Δ_W and an empty list.

$$\frac{((\text{x} \equiv zero) \vee (\text{cands} \equiv \{\emptyset\}))}{\Psi \text{ ; w-id ; } \Delta_W \text{ ; } \Delta_N \text{ ; cands ; x} \vdash_{select-while-filtering} \Delta_W \text{ ; } nil}$$

And now the recursive step, which happens in all other cases. It first selects an option from candidates, then it creates a soon-to-be shared new entry for the Δ_W of both the current walk and the selected candidate's walk. This shared entry will be the union of the two, because the two walks are now connected.:

$$\frac{\begin{array}{c}\neg((\text{x} \equiv zero) \vee (\text{cands} \equiv \{\emptyset\})) \\ \text{cand} = \xi(\text{cands}) \\ (\text{c-id, (n-id, vn?, t, dep-by, dep-on, col?))} := \text{cand} \\ \text{new-group} := \Delta_W(\text{first } \Delta_N(\text{n-id})) \cup \Delta_W(\text{w-id}) \\ \Delta'_W := \text{fold (k ; acc} \rightarrow \text{set acc k new-group) } \Delta_W \text{ new-group} \\ \text{get-walk-id} := (\text{(c-id}', \text{(n-id}', \text{vn?}', \text{t}', \text{dep-by}', \text{dep-on}', \text{col?}')) \rightarrow \text{first } \Delta_N(\text{n-id}')) \\ \text{cands}' := \text{remove cand cands} \\ \text{next} := \text{filter (c} \rightarrow |\text{new-group} \cup \Delta'_W(\text{get-walk-id c})| \leq \Psi(\text{MCWA})) \text{ cands}' \\ \Psi \text{ ; w-id ; } \Delta'_W \text{ ; } \Delta_N \text{ ; next ; x} - 1 \vdash_{s.w.f} \Delta''_W \text{ ; xs}\end{array}}{\Psi \text{ ; w-id ; } \Delta_W \text{ ; } \Delta_N \text{ ; cands ; x} \vdash_{s.w.f} \Delta''_W \text{ ; (cand : xs)}}$$

This concludes the formalization for the walk procedure. When the algorithm returns to the top-level $\vdash_{walk}$, the integration of the new walk with the environment is finished by joining the node-level associative lists and doing some post-processing to determine if a node is ready to be processed or if it has active dependencies, a process that can be seen in the definition of that judgment.

4.7 Command Generator ($\vdash_{cmd}$)

With the walk environment defined and, at the point the command generator is called by the top-level $\vdash_{CPS}$, initialized, it's possible to elaborate on how these walks are actually processed and turned into CPS commands. The summary of this process goes like this:

1. If the walk environment is empty, perform a random jump with no binds. If not, sample from it a random node with no active dependencies, always giving preference to those that have at least one component without a variable name set, to allow at least one bind to be made and for the computation to proceed.

2. Determine how many components without an assigned variable will get one by fetching from the environment, and label them. The others will either generate a bind or, if the component's type is a base type, get assigned a free variable. This will create an execution plan for the current step.

3. Remove the current node and its components from the walk environment, preparing it for a recursive step. Given n planned binds, distribute the Δ_G's connected components

randomly across the n recursive commands, also moving the Δ_N and Δ_C elements to the appropriate binds. Also randomly divide the value currently held by w_{acc} to the same n recursion spots.

4. Make the binds. For every component in the node, if it doesn't have a variable and it's not labeled as a component that will get its variable by fetching from the environment, create a bind to a new continuation of the same type as the component's. If a bind is to be done, the process goes like this:

   a) For every parameter in this new bind, decide if a new walk is going to be created by using its type as the root or if a collide is going to happen, where a collide is the operation of assigning this new parameter to the variable field of an empty component from any node currently in the scope of that new bind.

   b) Then, perform a recursive step, creating the inner command contained by that bind.

   c) When it returns, build the actual bind structure, but at this point add it to a list instead of nesting one inside the other, like the CPS grammar would require. With that bind, maybe perform a (FLOAT-R) operation, moving from the right side of the equality described in 2.1.10.2 to the left, that is, moving an inner bind from the recursively generated command to the outer-level, specifically to the right of the bind that hosts said command.

   d) Recursively continue to the next component, adding the current bind and any of the floated binds to the variables environment.

   e) When it returns, output an appended list containing the binds calculated for the next components together with the current bind and the floated binds, alongside information needed to perform further FLOAT-R operations and to assign that new bind to the empty variable field of the component that originated it.

5. With the output of the bind creation procedure, fetch variables from the new environment for the components labeled in step (2), and assign the new binds to the appropriate components. Then, with all of the components hosting a variable and having all of the binds in list format, nest the command, transforming that list of binds into the recursive structure of a CPS command, with the jump placed at the end.

The above summary will now be formalized in much more detail. Starting with the top-level $\vdash_{cmd}$, it gets these parameters:

- Ψ: Associative list of parameters.
- Θ: The type environment.

- Δ: The walk environment. This is a 3-tuple containing an associative list for the nodes, an associative list for the components and a dependency directed acyclic graph connecting nodes. These can be accessed through Δ_N, Δ_C and Δ_G, respectively.
- Γ: Variables environment. This is organized differently than what is common in runtime environments, as instead of mapping a variable name to a type, this environment instead maps a type to a possibly empty set of variable names of that type.
- b: Set of base types.
- w_{acc}: Walk steps remaining. If non-positive, then only backward steps are allowed for future walks.

And it returns a CPS-calculus command and an associative list mapping the name of the binds capturable by static contexts on that command to their types.

An associative list mapping the name of a recently bound continuation to its type. This does contain floated binds, that is, binds created inside a bind's internal command that were brought to the external scope.

Before talking about the judgment, a small note on terminology. In this section, a walk environment Δ is empty if $\Delta_N \equiv \{\emptyset\}$. As there are no nodes, this also implies $\Delta_C \equiv \{\emptyset\}$ and $\Delta_G \equiv \{\emptyset\}$. Next, nodes are deemed *ready* if their last field, ready?, has the value $\top$, which means none of its components have any dependencies. At any given step, only ready nodes can be processed, and their execution can make other nodes ready by removing a dependency on the node that was just processed. A node can also be deemed *complete*, which means that all of its components have a variable name set for the vn? field. Not all complete nodes are also ready nodes, because it's possible to have extrinsic dependencies such that a complete node is waiting for an incomplete node to be processed for it to be considered ready.

This judgment has three rules. Starting with a base case, which happens when there are no more nodes to process, i.e. when the walk environment is empty. In this scenario, a jump with no binds is done, with the guidance that the arguments for this jump must all be bound variables, while still allowing the continuation being jumped to to be a free variable. Notably, a jump with no arguments is therefore always an available option. This case is defined below:

$$\frac{\begin{array}{c}\Delta_N \equiv \{\emptyset\} \\ \text{jumpable-types} := \text{filter-keys } (\mathrm{k} \to ((\neg\vec{\tau} \equiv k) \wedge (\bigwedge_{\tau\in\vec{\tau}} \Gamma(\tau) \not\equiv \{\emptyset\}))) \; \Theta \\ \neg\vec{\tau} = \xi(\text{jumpable-types}) \\ \text{head} := \xi(\{\xi_{id}\} \cup \Gamma(\neg\vec{\tau})) \\ \text{args} := \text{map } (\tau \to \xi(\Gamma(\tau))) \; \vec{\tau}\end{array}}{\Psi \; ; \; \Theta \; ; \; \Delta \; ; \; \Gamma \; ; \; \mathrm{b} \; ; \; \mathrm{w}_{acc} \vdash_{cmd} (\text{head } \langle \text{ args } \rangle) \; ; \; \{\emptyset\}}$$

Then a second base case. This one represents the event when the walk environment is not empty, but all of the ready nodes are also complete. When this happens with a walk environment

possessing more than one node in total, it results in a loss of information, because it makes it impossible to create any more binds on the current command branch and, as a result, one of these nodes is picked for a jump with no binds and the rest are discarded. Its rule is defined as follows:

$$\frac{\begin{array}{c}\Delta_N \not\equiv \{\emptyset\} \\ \text{ready-nodes} := \text{filter-values } ((\text{w-id, typ, comps, ready?}) \to \text{ready?})\ \Delta_N \\ \text{fn}_1 := ((\text{w-id, typ, comps, ready?}) \to \neg(\bigwedge_{(\text{n-id, vn, t, dep-by, dep-on, col?}) \in \text{comps}} \text{vn} \not\equiv \cdot)) \\ \text{incomplete-nodes} := \text{filter-values } \text{fn}_1 \text{ ready-nodes} \\ \text{incomplete-nodes} \equiv \{\emptyset\} \\ (\text{node-id, (w-id, typ, comps, ready?)}) = \xi(\text{ready-nodes}) \\ \text{fn}_2 := (\text{c-id} \to \text{let } ((\text{n-id, vn, t, dep-by, dep-on, col?}) := \text{comps(c-id)}) \text{ in vn}) \\ (\text{head} : \text{args}) := \text{map } \text{fn}_2 \text{ comps}\end{array}}{\Psi\ ;\ \Theta\ ;\ \Delta\ ;\ \Gamma\ ;\ \text{b}\ ;\ \text{w}_{acc} \vdash_{cmd} (\text{head } \langle\ \text{args}\ \rangle)\ ;\ \{\emptyset\}}$$

And now the recursive step, which happens when the walk environment is not empty and there is a ready node that is not complete. For the sake of organization, this judgment will be split into several smaller ones, even when no control flow decision is necessary. On the top-level, and referencing the steps given by the summary located at the start of this section, a ready but incomplete node is randomly sampled from the environment (1), and then its list of components is inputted to $\vdash_{cmd-plan}$, which results in an execution plan for the current step (2). Afterwards, this plan is used to recursively create binds (3 and 4) with $\vdash_{cmd-exec}$, resulting in 6 outputs, all of them to be explained in that judgment. Lastly, input all of these outputs directly into $\vdash_{cmd-build}$ for a final post-processing step that builds the command (5). Both of these will be returned unmodified by the top-level $\vdash_{cmd}$. The top-level recursive step is defined below:

$$\frac{\begin{array}{c}\Delta_N \not\equiv \{\emptyset\} \\ \text{ready-nodes} := \text{filter-values } ((\text{w-id, typ, comps, ready?}) \to \text{ready?})\ \Delta_N \\ \text{fn} := ((\text{w-id, typ, comps, ready?}) \to \neg(\bigwedge_{(\text{n-id, vn, t, dep-by, dep-on, col?}) \in \text{comps}} \text{vn} \not\equiv \cdot)) \\ \text{incomplete-nodes} := \text{filter-values fn ready-nodes} \\ \text{incomplete-nodes} \not\equiv \{\emptyset\} \\ (\text{node-id, (w-id, typ, comps, ready?)}) := \xi(\text{incomplete-nodes}) \\ \Psi\ ;\ \Delta\ ;\ \Gamma\ ;\ \text{comps} \vdash_{cmd-plan} \sigma \\ \Psi\ ;\ \Theta\ ;\ \Delta\ ;\ \Gamma\ ;\ \text{b}\ ;\ \text{w}_{acc}\ ;\ \text{node-id}\ ;\ \sigma \vdash_{cmd-exec} \neg\vec{\tau}\ ;\ \chi\ ;\ \varsigma\ ;\ \omega\ ;\ \Pi\ ;\ \Sigma \\ \Gamma\ ;\ \neg\vec{\tau}\ ;\ \chi\ ;\ \varsigma\ ;\ \omega\ ;\ \Pi\ ;\ \Sigma \vdash_{cmd-build} \text{command}\end{array}}{\Psi\ ;\ \Theta\ ;\ \Delta\ ;\ \Gamma\ ;\ \text{b}\ ;\ \text{w}_{acc} \vdash_{cmd} \text{command}\ ;\ \Pi}$$

The internal judgments are now going to be defined. For all the remainder of this formalization, consider γ_C to be an alias for the tuple (n-id, vn?, τ, dep-by, dep-on, col?) and γ_N to be an alias for the tuple (wn, $\neg\vec{\tau}$, cs, r?).

4.7.1 Create an execution plan for a jump ($\vdash_{cmd-plan}$)

After having picked a random incomplete node from the walk environment, the algorithm assembles an execution plan to guide the way this node is going to be processed. Specifically, for a node with n components, this execution plan is going to be a list of n elements, such that each element is a 3-tuple consisting of a component identifier, a flag and the component's entry in Δ_C. This last element is optional, as it's possible to get this entry by performing a lookup on the component-level associative list, but it's left as a member of the tuple for convenience.

Two main pieces of information are encoded in an execution plan:

- The order of components to be processed. Every element in the execution plan represents one of the node's components, but they are shuffled and rearranged to allow for binds and lookups to happen in an order that is independent of their position in the node's original component structure.
- What to do with each component. If a component has a variable name set, then that variable is going to be used. If it does not, however, multiple options are available. The second element in each of the plan's tuples is a flag that can either be a variable name or one of $\cdot$ or $\circ$. If it's a $\cdot$, a new bind is performed. If it's a $\circ$, the variables environment is searched for a suitable variable to be assigned to the component.

To create such an execution plan, the $\vdash_{cmd-plan}$ is defined. This judgment is not recursive and has only one case. Its separation from the top-level $\vdash_{cmd}$ is therefore entirely for organization reasons. It takes the following parameters:

- Ψ: Associative list of parameters.
- Δ: Walk environment.
- Γ: Variables environment.
- comps: List of the current node's component ids.

And it returns an execution plan, which is a list of 3-tuples, each containing a component identifier, a flag and an entry from Δ_C.

Before formalizing the rule, a brief overview of what is done: First, it's calculated how many components don't have a variable assigned. These are holes with the potential of becoming a bind or of being filled with a variable obtained from environment lookups. However, if there are more nodes in the environment besides the current one, filling every single hole with lookups will result in these nodes being discarded, as there would be no binds left to continue the computation in. To circumvent this issue, if there is more than a single node in the environment, the maximum

number of fills allowed will be one less than the number of holes. With this ceiling defined, a number of fills is also chosen.

After that, the component identifiers are shuffled and inputted to the $\vdash_{cmd-shift-bases}$ judgment. All this does is, depending on a comparison between an uniformly sampled α and a threshold, move base types to the left of the shuffled component list. This is done because the choice of which components are going to be filled with lookups is decided based on the first k components in the shuffled list, for a k number of fills, and as a result, moving elements to earlier positions on the list increases the likelihood of it being assigned a lookup flag, instead of a bind flag. This is desirable for base types, because there can't be a bind for them. If a component that represents a base type is assigned the flag $\cdot$, instead of performing a bind, a new free-variable will be created for it.

After the first labeling, a transformation is applied on the components that got $\circ$ as a flag. The main reason this transformation was designed was to make it so that the number of variables of a given type currently in the variables environment has an impact on the likelihood of a component being filled using that environment. It's because of this transformation that components whose types don't have any variables in the environment are not labeled to perform a lookup in search of such a variable. The parameter $\Psi(\text{FB})$ controls the Fill Bias, and the higher it is, the more weight each variable in the environment has for the current component to maintain the $\circ$ flag. There's also $\Psi(\text{BB})$, the Bind Bias, which counteracts the Fill Bias. The parameter $\Psi(\text{BB})$ does not interact with the size of the environment, but high values of it can make fills very difficult to happen, even in environments with several suitable variables.

Lastly, one final transformation is applied. All it does is assign a free variable to any element in the execution plan that has the flag $\cdot$ and that is also a base type. As mentioned, it's not possible to create binds for base types, and this transformation deals with that.

The case performing all of these steps is defined below:

$$\text{fn}_1 := (\text{c-id} \to \text{let } (\gamma_C := \Delta_C(\text{c-id})) \text{ in } ((\text{vn?} \equiv \cdot) \ ? \ 1 : 0))$$
$$\text{num-holes} := \sum (\text{map fn}_1 \text{ comps})$$
$$\text{fill-max} := (|\Delta_N| > 1) \ ? \ (\text{num-holes} - 1) : \text{num-holes}$$
$$\text{num-fills} := \xi([0..=\text{fill-max}])$$
$$\Psi \ ; \ \xi(\mathbb{R}^u) \ ; \ \Delta \ ; \ nil \ ; \ nil \ ; \ \xi_{shuffle}(\text{comps}) \vdash_{cmd-shift-bases} \text{shuffled}$$
$$\text{labeled} := \text{choose-filling-spots num-fills shuffled}$$
$$\text{fn}_{\text{FB}} := (\tau \to \text{repeat } \circ \ (\Psi(\text{FB}) * |\Gamma(\tau)|))$$
$$\text{fn}_{\text{BB}} := (\tau \to \text{repeat } \cdot \ (((\Psi(\text{BB}) \equiv 0) \wedge (\Psi(\text{FB}) * |\Gamma(\tau)| \equiv 0)) \ ? \ 1 : \Psi(\text{BB})))$$
$$\text{fn}_2 := (\tau \to \xi(\text{append } (\text{fn}_{\text{FB}} \ \tau) \ (\text{fn}_{\text{BB}} \ \tau)))$$
$$\text{fn}_3 := ((\text{c, f?}, \gamma_C) \to (\text{c, } ((\text{f?} \equiv \circ) \ ? \ (\text{fn}_2 \ \tau) : \text{f?}), \gamma_C))$$
$$\hat{\sigma} := \text{map fn}_3 \text{ labeled}$$
$$\text{fn}_4 := ((\text{c, f?}, \gamma_C) \to (\text{c, } (((\text{f?} \equiv \cdot) \wedge (\tau \not\equiv \neg\vec{\pi})) \ ? \ \xi_{id} : \text{f?}), \gamma_C))$$
$$\frac{\sigma := \text{map fn}_4 \ \hat{\sigma}}{\Psi \ ; \ \Delta \ ; \ \Gamma \ ; \ \text{comps} \vdash_{cmd-plan} \sigma}$$

4.7.1.1 Maybe move some base types to the left ($\vdash_{cmd-shift-bases}$)

This judgment takes in a shuffled list of component identifiers and two accumulators initialized as empty lists, and for each element of that identifier list, it either gets added to the left list or to the right list. At the end, these two accumulators are joined into a single list, which is returned.

The requirements to assign an identifier to the list on the left is for the component with that identifier to be of a base type and also for a random score $\alpha \sim \mathbb{U}[0, 1]$ to exceed the Move Primitive Left Threshold, that is, $\alpha > \Psi(\text{MBLT})$. Otherwise, the element is added to the list on the right.

Formally, the judgment takes these parameters:

- Ψ: Associative list of parameters.
- α: Random score sampled from $\mathbb{U}[0, 1]$.
- Δ: Walk environment.
- L: Left accumulator.
- R: Right accumulator.
- comps: List of shuffled component identifiers. The recursion is based on this parameter.

And it returns a list of component identifiers.

Below is the base type, which happens when the comps list is empty. It appends both accumulators and returns the result:

$$\frac{\text{out} := \text{append L R}}{\Psi\ ;\ \alpha\ ;\ \Delta\ ;\ \text{L}\ ;\ \text{R}\ ;\ nil \vdash_{cmd-shift-bases} \text{out}}$$

And now a recursive case for when the comps list is not empty. This one represents the event when a shift happens, that is, when the component identifier is added to the list on the left:

$$\frac{\begin{array}{c}(\text{n-id, vn?, } \tau\text{, dep-by, dep-on, col?}) := \Delta_C(\text{c-id}) \\ ((\alpha > \Psi(\text{MBLT})) \wedge (\tau \not\equiv \neg\vec{\pi})) \\ \Psi\ ;\ \xi(\mathbb{R}^u)\ ;\ \Delta\ ;\ (\text{c-id} : \text{L})\ ;\ \text{R}\ ;\ \text{cs} \vdash_{cmd-shift-bases} \text{out}\end{array}}{\Psi\ ;\ \alpha\ ;\ \Delta\ ;\ \text{L}\ ;\ \text{R}\ ;\ (\text{c-id} : \text{cs}) \vdash_{cmd-shift-bases} \text{out}}$$

And the other recursive case, mapping the event when the component identifier is added to the list on the right:

$$\frac{\begin{array}{c}(\text{n-id, vn?, } \tau\text{, dep-by, dep-on, col?}) := \Delta_C(\text{c-id}) \\ \neg((\alpha > \Psi(\text{MBLT})) \wedge (\tau \not\equiv \neg\vec{\pi})) \\ \Psi\ ;\ \xi(\mathbb{R}^u)\ ;\ \Delta\ ;\ \text{L}\ ;\ (\text{c-id} : \text{R})\ ;\ \text{cs} \vdash_{cmd-shift-bases} \text{out}\end{array}}{\Psi\ ;\ \alpha\ ;\ \Delta\ ;\ \text{L}\ ;\ \text{R}\ ;\ (\text{c-id} : \text{cs}) \vdash_{cmd-shift-bases} \text{out}}$$

4.7.1.2 Label components to fetch from environment (choose-filling-spots)

This function has two related purposes. First, it turns a list of identifiers into a list of tuples, which is the execution plan that was previously elaborated upon. Second, it assigns the flags to these tuples, which will indicate if a certain component already has a variable name, and if not, whether that absence will result in a new bind ($\cdot$) or in a lookup to the variables environment ($\circ$). The definition below is in a Haskell-like functional pseudo-code:

choose-filling-spots $:: \Delta \to \Gamma \to \mathbb{N} \to [\text{identifier}] \to [(\text{identifier}, \text{flag}, \text{component})]$

$$
\begin{aligned}
&\text{choose-filling-spots } \Delta\ \Gamma\ \text{num } nil && = nil \\
&\text{choose-filling-spots } \Delta\ \Gamma\ \text{num (c-id : cs)} && \text{let } (\gamma_C := \Delta_C(\text{c-id})) \text{ in} \\
&\quad | \ (\text{vn?} \not\equiv \cdot) && = (\text{c-id vn? } \gamma_C) : (\text{choose-filling-spots } \Delta\ \Gamma\ \text{num cs}) \\
&\quad | \ ((\text{num} > 0) \wedge (|\Gamma(\tau)| > 0)) && = (\text{c-id} \circ \gamma_C) : (\text{choose-filling-spots } \Delta\ \Gamma\ (\text{num} - 1)\ \text{cs}) \\
&\quad | \ \text{otherwise} && = (\text{c-id} \cdot \gamma_C) : (\text{choose-filling-spots } \Delta\ \Gamma\ \text{num cs})
\end{aligned}
$$

4.7.2 Follow the execution plan to generate new binds ($\vdash_{cmd-exec}$)

With the execution plan available, it's time to process it in order. At first, only binds will be processed, which is what guides the recursion through the generation of an inner CPS command for each new bind being created. Then, the jump is gonna be built, which is done by using the previously defined binds together with the already preset variables in the components and then by fetching the remaining variables from the environment. The non-recursive $\vdash_{cmd-exec}$ judgment is defined for this purpose. It takes the following parameters

- Ψ: Associative list of parameters.
- Θ: Type environment.
- Δ: Walk environment.
- Γ: Variables environment.
- b: Set of base types.
- w_{acc}: Remaining walk steps.
- n-id: Identifier of the current node being processed.
- σ: The execution plan.

And it returns 6 values. They are:

- $\neg\vec{\tau}$: The node's type. In this judgment, the name of the return is set as typ.

- χ: A list containing the identifiers for the node's components. In this judgment, the name of the return is set as comps.
- ς: An associative list obtained by removing the component element from the 3-tuples in σ. This maps a component identifier to its flag in the execution plan.
- ω: List of binds. Each bind has format $(\{k\langle\vec{x}\rangle = N\})$, where k is an identifier, $\vec{x}$ is a list of identifiers and N is an inner command. Notice that this is not a CPS command, as it doesn't have the outer command on the left. This list of binds is going to later be nested into the expected recursive structure. This list does not contain the jump.
- Π: An associative list mapping the name of a recently bound continuation to its type. This does contain floated binds, that is, binds created inside a bind's internal command that were brought to the external scope.
- Σ: An associative list mapping the component-id to the name of the continuation it defined with a bind. This does not contain floated binds.

The judgment has two rules. Despite efforts taken through the selection of incomplete components and through the creation of an execution plan that often reserves at least one unfilled spot for a bind and that prioritizes filling base type components first, it's still possible for $\vdash_{cmd-exec}$ to be called with a plan that does not involve any binds, specially considering the impossibility of creating a bind to a continuation of a base type. The first rule covers this case, and it simply stops the execution without engaging in the recursive bind-creating procedure. The last three outputs are provided as empty structures. The rule for this case is defined as follows:

$$\frac{\begin{array}{c}\text{(w-id, typ, comps, ready?)} := \Delta_N\text{(n-id)} \\ \varsigma := \text{map } ((\text{c, f?}, \gamma_C) \to (\text{c, f?}))\ \sigma \\ \text{num-binds} := \sum(\text{map } ((\text{c, f?}, \gamma_C) \to (((\text{f?} \equiv \cdot) \wedge (\tau \equiv \neg\vec{\pi}))\ ?\ 1 : 0))\ \sigma) \\ \text{num-binds} \equiv 0\end{array}}{\Psi\ ;\ \Theta\ ;\ \Delta\ ;\ \Gamma\ ;\ \text{b}\ ;\ \text{w}_{acc}\ ;\ \text{n-id}\ ;\ \sigma \vdash_{cmd-exec} \text{typ}\ ;\ \text{comps}\ ;\ \varsigma\ ;\ nil\ ;\ \{\emptyset\}\ ;\ \{\emptyset\}}$$

The second case is when there are binds to be done. After the num-binds > 0 check, 3 internal judgments are called sequentially. The first, $\vdash_{cmd-remove-current}$, removes the current node from the walk environment. The second, $\vdash_{cmd-split}$, distributes the walks and the remaining walk steps among the planned binds. The third, $\vdash_{cmd-bind}$, actually creates these binds and is responsible for calling $\vdash_{cmd}$ internally to create their commands. The rule is defined as such:

$$\frac{\begin{array}{c}\text{(w-id, typ, comps, ready?)} := \Delta_N\text{(n-id)} \\ \varsigma := \text{map } ((\text{c, f?}, \gamma_C) \to (\text{c, f?}))\ \sigma \\ \text{num-binds} := \sum(\text{map } ((\text{c, f?}, \gamma_C) \to (((\text{f?} \equiv \cdot) \wedge (\tau \equiv \neg\vec{\pi}))\ ?\ 1 : 0))\ \sigma) \\ \text{num-binds} > 0 \\ \Delta\ ;\ \text{n-id}\ \vdash_{cmd-remove-current-node}\ \Delta' \\ \Psi\ ;\ \Delta'\ ;\ \text{w}_{acc}\ ;\ \text{n-id}\ ;\ \text{num-binds} \vdash_{cmd-split} \Omega_G\ ;\ \varphi_w \\ \Psi\ ;\ \Theta\ ;\ \Delta'\ ;\ \Gamma\ ;\ \text{b}\ ;\ \text{w-id}\ ;\ \Omega_G\ ;\ \varphi_w\ ;\ 0\ ;\ \sigma \vdash_{cmd-bind} \omega\ ;\ \Pi\ ;\ \Sigma\end{array}}{\Psi\ ;\ \Theta\ ;\ \Delta\ ;\ \Gamma\ ;\ \text{b}\ ;\ \text{w}_{acc}\ ;\ \text{n-id}\ ;\ \sigma \vdash_{cmd-exec} \text{typ}\ ;\ \text{comps}\ ;\ \varsigma\ ;\ \omega\ ;\ \Pi\ ;\ \Sigma}$$

4.7.2.1 Remove the current node from the walk environment ($\vdash_{cmd-remove-current-node}$)

This judgment's only purpose is to remove the current node from the walk environment, which includes modifications to all four elements of the Δ tuple. It gets the walk environment and a node identifier as an input and outputs a modified walk environment, without the corresponding node, its components, its graph vertex or any of its associated edges. This judgment only has one case and is not recursive, and as such it was only created for organization purposes. Its singular rule is defined below:

$$\frac{\begin{array}{c}\text{(w-id, typ, comps, ready?)} := \Delta_N\text{(node-id)} \\ \text{comp-updt} := (\gamma_C \to (\text{n-id, vn?}, \tau, \text{dep-by, (dep-on} \setminus \text{comps), col?})) \\ \text{comp-fold-fn} := (\text{k ; acc} \to \text{update acc k comp-updt}) \\ \text{comp-initial-acc} := \text{filter-values } (\gamma_C \to \text{n-id} \not\equiv \text{node-id}) \text{ comps} \\ \text{comp-iterable} := \bigcup_{\text{c}\in\text{comps}} \text{let } (\gamma_C := \Delta_C\text{(c)}) \text{ in (map ((id, t)} \to \text{id) dep-by)} \\ \Delta'_C := \text{fold comp-fold-fn comp-initial-acc comp-iterable} \\ \text{node-updt} := (\gamma_N \to (\text{wn}, \neg\vec{\tau}, \text{cs}, (\textstyle\bigwedge_{\text{c}\in\text{cs}} \text{let } (\gamma_C := \Delta'_C\text{(c)}) \text{ in (dep-on} \equiv \{\emptyset\})))) \\ \text{node-fold-fn} := (\text{k ; acc} \to \text{update acc k node-updt}) \\ \text{node-initial-acc} := \text{remove } \Delta_N \text{ node-id} \\ \text{node-iterable} := \mathcal{N}_{\Delta_G}\text{(node-id)} \\ \Delta'_N := \text{fold node-fold-fn node-initial-acc node-iterable} \\ \Delta'_G := \text{remove node-id } \Delta_G \\ \text{walk-fold-fn} := (\text{wgc ; acc} \to \text{fold (w ; acc2} \to \text{set acc2 w } (toSet\ \text{wgc})) \text{ acc wgc}) \\ \text{walk-initial-acc} := (\textstyle\bigvee_{\text{c}\in\text{comps}} \text{let } (\gamma_C := \Delta_C\text{(c)}) \text{ in } (\bigvee_{(\_,i)\in\text{dep-by}} i))\ ?\ \Delta_W : (\Delta_W \setminus \{\text{w-id}\}) \\ \Delta'_G\text{-comps} := \text{connected-components } \Delta'_G \\ \text{walk-iterable} := \text{map (gc} \to \text{unique (map (n} \to \text{let } (\gamma_N := \Delta'_N\text{(n)}) \text{ in wn) gc))} \Delta'_G\text{-comps} \\ \Delta'_W := \text{fold walk-fold-fn walk-initial-acc walk-iterable}\end{array}}{\Delta\ ;\ \text{node-id} \vdash_{cmd-remove-current-node} \Delta'}$$

The function $\mathcal{N}_{\Delta_G}$(node-id) is used to get the out-neighbors of vertex node-id in Δ_G, where the out-neighbors of a vertex v in a directed graph are all vertices w with an edge coming from v to w, so (v, w).

4.7.2.2 Split graph components and w_{acc} among planned binds ($\vdash_{cmd-split}$)

The creation of binds is triggered by a node being processed, but often that node is not going to be the only one present in the walk environment. As the processing of a node results in a jump at the leftmost command position in the sequence of binds it created, future nodes will need to be processed inside one of the binds. Then, there needs to be a decision of which bind is going to process what nodes.

Besides that, sharing the same w_{acc} among all binds could lead to a command growing exponentially with the size of w_{acc}. To avoid that, the current value of w_{acc} should be split among the binds as well.

This judgment, $\vdash_{cmd-split}$, partitions the nodes and w_{acc} among the planned binds. It only has one rule, and as such it was defined only for organizational purposes. It takes the following parameters:

- Ψ: Associative list of parameters.
- Δ: Walk environment.
- w_{acc}: Remaining walk steps.
- node-id: Identifier of the current node being processed.
- num-binds: A natural number indicating how many binds are planned.

And it outputs the following:

- Ω_G: A list of tuples containing an index at first position and a set of vertices representative of a connected graph component. All of these vertices are from Δ_G and they are disjoint among each other. This list's length is the number of connected components present in Δ_G.
- φ_w: A list of numbers, indicating the w_{acc} for each bind. This list's length is the number of binds.

It works like this: first, the connected components of an undirected version of Δ_G are obtained. These are then partitioned between final and non-final graph components, named this way because a final graph component is one with a single node and such that that node does not generate any binds. In other words, processing the node a final graph component has results in the end of the recursive process. The condition involves checking if the component is a singleton and if all of that node's components either have a variable name set or have its type being a base type.

After, the ready final components are uniquely allocated among the binds. This allocation is unique because sending two final components to the same bind results in the additional ones

always being discarded, as there are no more binds to process them in. To reduce the likelihood of such cases, if a component goes to the first bind, then no final components can go to that same bind also, for example. If there are more final components than there are planned binds, the leftovers are returned.

Then the non-final are freely allocated to the binds, and after that the final leftovers of the unique allocation are allocated, but this time limited to the binds in which there is at least one non-final component.

The results from all these three allocations are finally appended into Ω_G, which hosts a list of 2-tuples. The first element of these tuples is the index a graph component was allocated in, and the second element is that graph component itself.

After that, it's time to spread the w_{acc}. First, real scores x are uniformly sampled for each bind, such that $0 < x \leq 1$. These are then normalized, to make it so that $\sum_{w \in \vec{w}} w \equiv 1$. If there are no more nodes left, i.e. $|\Delta_N| \equiv 0$, then those normalized scores are multiplied by the w_{acc} and returned. Otherwise, a second procedure is also applied.

This second procedure works like this: for every bind, get all of the graph components assigned to it and sum all of their sizes, then divide that sum by the total number of vertices across all graph components. This could indicate that a certain bind is being assigned, for example, 60% of the total number of nodes being distributed. Instead of relying purely on randomness to determine a bind's scores, it's possible to assign it a value proportionate to how much of the walk environment it's getting.

The final scores use both methods, relying on a Walk Distribution Bias variable to determine their weight, such that $0 \leq \Psi(\mathrm{WDB}) \leq 1$. Specifically, considering w_1 to be the randomly sampled score and w_2 to be the proportional score, the final score for each bind is calculated as: $w_1 * \Psi(\mathrm{WDB}) + w_2 * (1 - \Psi(\mathrm{WDB}))$. These scores are then assigned to φ_w.

The rule for the judgment implementing this is as follows:

$$\text{cc-G} := \text{connected-components (to-undirected } \Delta_G)$$
$$\text{fn}_1 := (\text{c-id} \rightarrow \text{let } (\gamma_C := \Delta_C(\text{c-id})) \text{ in } ((\text{vn?} \not\equiv \cdot) \vee (\tau \not\equiv \neg\vec{\pi})))$$
$$\text{fn}_2 := (\text{n-id} \rightarrow \text{let } (\gamma_N := \Delta_N(\text{n-id})) \text{ in } (\bigwedge_{\text{c-id} \in \text{cs}} \text{fn}_1 \text{ c-id}))$$
$$(\text{G-final, G-nonfinal}) := \text{partition } (\text{gc} \rightarrow (\mathcal{V}(\text{gc}) \equiv \{\text{singleton}\}) \wedge (\text{fn}_2 \text{ singleton})) \text{ cc-G}$$
$$\text{num-allocs} := \min(\text{num-binds}, \ |\text{G-final}|)$$
$$\text{G-final ; } [0..\text{num-binds}] \text{ ; num-allocs} \vdash_{split-unique} \text{alloc-complete ; free-complete}$$
$$\text{alloc-incomplete} := \text{map } (\text{gc} \rightarrow (\xi([0..\text{num-binds}]), \ \text{gc})) \text{ G-nonfinal}$$
$$\text{incomplete-indices} := \text{unique } (\text{map } ((\text{i, c}) \rightarrow \text{i}) \text{ alloc-incomplete})$$
$$\text{free-indices} := (\text{empty? incomplete-indices}) \ ? \ [0..\text{num-binds}] : \text{incomplete-indices}$$
$$\text{alloc-remaining} := \text{map } (\text{gc} \rightarrow (\xi(\text{free-indices}), \ \text{gc})) \text{ free-complete}$$
$$\Omega_G := \text{append alloc-complete alloc-incomplete alloc-remaining}$$
$$\text{w-scores} = \text{repeat } \xi(\{x \in \mathbb{R} \mid 0 < x \leq 1\}) \text{ num-binds}$$
$$\text{w-normed} := \text{map } (\text{w} \rightarrow \tfrac{\text{w}}{\sum \text{w-scores}}) \text{ w-scores}$$
$$\text{fn}_1 := (\text{idx} \rightarrow |\bigcup \text{map } ((\text{i, gc}) \rightarrow \mathcal{V}(\text{gc})) \text{ (filter } ((\text{i, gc}) \rightarrow \text{i} \equiv \text{idx}) \ \Omega_G)|)$$
$$\text{fn}_2 := (\text{w ; i} \rightarrow \text{w} * \Psi(\text{WDB}) + \tfrac{\text{fn}_1 \text{ i}}{|\Delta_N|} * (1 - \Psi(\text{WDB})))$$
$$\text{w-biased?} := (|\Delta_N| \equiv 0) \ ? \ \text{w-normed} : (\text{map fn}_2 \text{ w-normed } [0..\text{num-binds}])$$
$$\frac{\varphi_w := \text{map } (\text{w} \rightarrow \text{w} * \text{w}_{acc}) \text{ w-biased?}}{\Psi \text{ ; } \Delta \text{ ; } \text{w}_{acc} \text{ ; node-id ; num-binds} \vdash_{cmd-split} \Omega_G \text{ ; } \varphi_w}$$

4.7.2.3 Assign complete nodes to different binds ($\vdash_{split-unique}$)

This judgment was already briefly explained in the section about $\vdash_{cmd-split}$. This judgment, $\vdash_{split-unique}$, allocates a graph component to a bind index, disallowing repeated assignments to the same index. It is recursive on the number of allocations it should perform, which is assumed to be the minimum between the number of binds and the number of final graph components. It takes the following parameters:

- graphs: Graph components.
- indices: Bind indices.
- n: Remaining number of allocations. The recursion is based on this parameter.

And it returns a list of allocations, each of them being a tuple mapping an index to a graph component, and the graph components that were not able to be allocated.

The base case:

$$\frac{}{\text{graphs ; indices ; } zero \vdash_{split-unique} nil \text{ ; graphs}}$$

The recursive step:

$$\frac{\begin{array}{c}\text{comp} = \xi(\text{graphs}) \\ \text{idx} = \xi(\text{indices}) \\ (\text{graphs} \setminus \{\text{comp}\}) \text{ ; } (\text{indices} \setminus \{\text{idx}\}) \text{ ; n} \vdash_{split-unique} \text{allocated ; free} \\ \text{allocated}' := ((\text{idx, comp}) : \text{allocated})\end{array}}{\text{graphs ; indices ; } succ(\text{n}) \vdash_{split-unique} \text{allocated}' \text{ ; free}}$$

4.7.2.4 Create the new binds required by the execution plan ($\vdash_{cmd-bind}$)

With the graph components Ω_G and remaining walk steps φ_w been created and assigned to binds, it's possible to actually create them now. This is done with the $\vdash_{cmd-bind}$ judgment, which takes the following parameters:

- Ψ: Associative list of parameters.
- Θ: Type environment.
- Δ: Walk environment.
- Γ: Variables environment.
- b: Set of base types.
- w-id: Identifier for the current node's walk.
- Ω_G: List of tuples consisting of a bind index in the first position and a graph component in the second.
- φ_w: List of remaining steps, one for each planned bind.
- idx: Index of the current bind. Starts at zero and monotonically increases as binds are performed.
- plan: The execution plan. This is a list of tuples, where the first element is a component identifier, the second element is its flag in the plan, and the third element is its entry in the walk environment. The recursion is based on this parameter.

And outputs the following:

- ω: List of binds. Each bind is a tuple of the format $(\{k\langle\vec{x}\rangle = N\})$, where k is an identifier, $\vec{x}$ is a list of identifiers, N is an inner command, and the other symbols are constant literals. Notice that this is not a CPS command, as it doesn't have the outer command on the left. This list of binds is going to later be nested into the expected recursive structure. This list does not contain the jump.
- Π: An associative list mapping the name of a recently bound continuation to its type. This does contain floated binds, that is, binds created inside a bind's internal command that were brought to the external scope. The Float transformation is explained later.

- Σ: An associative list mapping the component-id to the name of the continuation it defined with a bind. This does not contain floated binds.

Starting with the base case, when all of the execution plan has been processed and all binds have been created. Just return empty structures at all output positions:

$$\frac{}{\Psi\ ;\ \Theta\ ;\ \Delta\ ;\ \Gamma\ ;\ \text{b}\ ;\ \text{w-id}\ ;\ \Omega_G\ ;\ \varphi_w\ ;\ \text{idx}\ ;\ nil \vdash_{cmd-bind} nil\ ;\ \{\emptyset\}\ ;\ \{\emptyset\}}$$

Then, the case when the bind is not performed. This happens either because the flag is different than $\cdot$ or because the component's type is a base type. Just skip this component and process the next:

$$\frac{\begin{array}{c}(\text{c-id, f?, (n-id, vn?, } \tau\text{, dep-by, dep-on, col?)) := c} \\ ((\text{f?} \not\equiv \cdot) \vee (\tau \not\equiv \neg\vec{\pi})) \\ \Psi\ ;\ \Theta\ ;\ \Delta\ ;\ \Gamma\ ;\ \text{b}\ ;\ \text{w-id}\ ;\ \Omega_G\ ;\ \varphi_w\ ;\ \text{idx}\ ;\ \text{cs} \vdash_{cmd-bind} \omega\ ;\ \Pi\ ;\ \Sigma\end{array}}{\Psi\ ;\ \Theta\ ;\ \Delta\ ;\ \Gamma\ ;\ \text{b}\ ;\ \text{w-id}\ ;\ \Omega_G\ ;\ \varphi_w\ ;\ \text{idx}\ ;\ (\text{c} : \text{cs}) \vdash_{cmd-bind} \omega\ ;\ \Pi\ ;\ \Sigma}$$

And lastly, the case mapping the event when a bind is to be generated. As a brief overview of how it works:

1. First, the judgment first creates some necessary structures for the bind, such as the internal command's walk environment, which is filtered from the original walk environment based on the nodes that are a part of a given bind's assigned graph components, and the name for each of the bind's variables.
2. Then, for each parameter in the bind, it's decided if that parameter will originate a walk, a collide or nothing at all, which is done by the judgment $\vdash_{walk-or-collide}$.
3. Next, a command for that bind is generated with $\vdash_{cmd}$, effectively creating the recursion for the entire command generation process.
4. After that returns, a Float-R transformation can be applied to the binds immediately inside the command, bringing them to the outside, which is done with the judgment $\vdash_{float-R}$.
5. Next, the current bind is created and appended to the list of floated binds, if there are any.
6. The new continuations are added to the variables environment and a recursive call to $\vdash_{cmd-bind}$ is performed, going to the next element in the execution plan.
7. Finally, join the outputs obtained from the recursive call with what was obtained from processing the current step, and return.

The judgment rule is defined like this:

$$\frac{\begin{array}{c}\text{(c-id, f?, (node-id, varname, cont-type, depended-by, depended-on, collideable?))} := \text{c} \\ ((\text{f?} \equiv \cdot) \wedge (\text{cont-type} \equiv \neg\vec{\pi})) \\ (\text{cont-name} : \text{arg-names}) = \text{repeat } \xi_{id}\ ((\text{length } \vec{\pi}) + 1) \\ (\text{curr-}\Omega_G,\ \text{next-}\Omega_G) := \text{partition } ((\text{bind-idx}, \_) \to \text{bind-idx} \equiv \text{idx})\ \Omega_G \\ (\text{curr-}\varphi_w : \text{next-}\varphi_w) := \varphi_w \\ \text{curr-vertices} := \bigcup(\text{map } ((\text{i, gc}) \to \mathcal{V}(\text{gc}))\ \text{curr-}\Omega_G) \\ (\text{curr-}\Delta_N,\ \text{next-}\Delta_N) := \text{partition-keys } (\text{n} \to \text{n} \in \text{curr-vertices})\ \Delta_N \\ (\text{curr-}\Delta_C,\ \text{next-}\Delta_C) := \text{partition-values } (\gamma_C \to \text{n-id} \in \text{curr-vertices})\ \Delta_C \\ (\text{curr-}\Delta_G,\ \text{next-}\Delta_G) := \text{partition } (v \to v \in \text{curr-vertices})\ \Delta_G \\ \text{curr-walks} := \text{unique } (\text{map } (\text{n} \to \text{let } (\gamma_N := \text{curr-}\Delta_N(\text{n})) \text{ in wn})\ \text{curr-vertices}) \\ (\text{curr-}\Delta_W,\ \text{next-}\Delta_W) := \text{partition-keys } (\text{w} \to \text{w} \in \text{curr-walks})\ \Delta_W \\ \text{args} = \xi_{shuffle}(\text{map } (\pi\ ;\ \text{x} \to (\pi, \text{x}))\ \vec{\pi}\ \text{arg-names}) \\ \Psi\ ;\ \Theta\ ;\ \text{curr-}\Delta\ ;\ \text{b}\ ;\ \text{curr-}\varphi_w\ ;\ \text{w-id}\ ;\ \text{args} \vdash_{walk-or-collide} \text{inner-}\Delta\ ;\ \text{inner-w}_{acc} \\ \text{inner-}\Gamma := \text{fold } ((\pi, \text{x})\ ;\ \text{acc} \to \text{update acc } \pi\ (\text{xs} \to (\text{x} : \text{xs})))\ \Gamma\ \text{args} \\ \Psi\ ;\ \Theta\ ;\ \text{inner-}\Delta\ ;\ \text{inner-}\Gamma\ ;\ \text{inner-w}_{acc} \vdash_{cmd} \text{inner-cmd}\ ;\ \text{inner-}\Pi \\ \Psi\ ;\ \xi(\mathbb{R}^u)\ ;\ \text{(to-set arg-names)}\ ;\ \text{inner-cmd} \vdash_{float-R} \text{new-inner-cmd}\ ;\ \text{floated-binds} \\ \text{curr-bind} := (\{\text{cont-name}\langle\text{arg-names}\rangle = \text{new-inner-cmd}\}) \\ \text{curr-}\omega := (\text{curr-bind} : (\text{reverse floated-binds})) \\ \text{cont-}\Gamma := \text{update } \Gamma\ \text{cont-type } (\text{xs} \to (\text{cont-name} : \text{xs})) \\ \text{fn-updater} := ((\{k\langle\vec{x}\rangle = m\})\ ;\ \text{acc} \to \text{update acc inner-}\Pi(k)\ (\text{xs} \to (k : \text{xs}))) \\ \text{next-}\Gamma := \text{fold fn-updater cont-}\Gamma\ \text{floated-binds} \\ \Psi\ ;\ \Theta\ ;\ \text{next-}\Delta\ ;\ \text{next-}\Gamma\ ;\ \text{w-id}\ ;\ \text{next-}\Omega\ ;\ \text{next-}\varphi_w\ ;\ (\text{idx} + 1)\ ;\ \text{cs} \\ \vdash_{cmd-bind} \text{next-}\omega\ ;\ \text{next-}\Pi\ ;\ \text{next-}\Sigma \\ \omega := \text{append next-}\omega\ \text{curr-}\omega \\ \text{floated-}\Pi := \text{map } (\{x\langle\vec{y}\rangle = m\}) \to (x, \text{inner-}\Pi(x)))\ \text{floated-binds} \\ \Pi := \text{next-}\Pi \cup \{(\text{cont-name}, \tau)\} \cup \text{floated-}\Pi \\ \Sigma := \text{next-}\Sigma \cup \{(\text{c-id}, \text{cont-name})\} \end{array}}{\Psi\ ;\ \Theta\ ;\ \Delta\ ;\ \Gamma\ ;\ \text{b}\ ;\ \text{w-id}\ ;\ \Omega_G\ ;\ \varphi_w\ ;\ \text{idx}\ ;\ (\text{c} : \text{cs}) \vdash_{cmd-bind} \omega\ ;\ \Pi\ ;\ \Sigma}$$

4.7.2.5 Instantiate a new walk or collide into an existing component ($\vdash_{walk-or-collide}$)

This judgment is called by $\vdash_{cmd-bind}$ to decide, for each parameter in the new bind, if it's going to originate a new walk or if it's going to start a collide. Walks have been thoroughly defined in 4.6, but collides have not, and so they will be defined now.

A collide is the process by which a new parameter searches the walk environment for an empty component of the same type so that it can assign itself into the component's variable field. Specifically, if a component has no variable set, is of the same type, has the collideable? field set to true, and does not depend and is not depended by any other components, then that component is eligible to be the target of a collide. If a component is chosen for a collide, then its variable field is set to the name of the parameter that started the operation. Effectively, this binds the parameter to an usage, where if that node is ever processed, then that parameter is guaranteed to be used in its jump.

The $\vdash_{walk-or-collide}$ judgment takes the following parameters:

- Ψ: Associative list of parameters.
- Θ: Type environment.
- Δ: Walk environment.
- b: Set of base types.
- w_{acc}: Remaining steps before walks are only allowed to take backward steps.
- walk-id: The walk identifier for the node being currently processed.
- args: The list of 2-tuples, containing a type as a tuple's first element and the name of a parameter as the second. The recursion is based on this parameter.

And outputs the following:

- Δ': A modified walk environment, with the extensions provided by any new walks and collides.
- w'_{acc}: Updated w_{acc} to reflect new walks.

The base case happens when the list of args is empty, and it simply returns the provided Δ and w_{acc} unmodified. It's defined like this:

$$\frac{}{\Psi\ ;\ \Theta\ ;\ \Delta\ ;\ \mathrm{b}\ ;\ \mathrm{w}_{acc}\ ;\ \text{walk-id}\ ;\ nil \vdash_{walk-or-collide} \Delta\ ;\ \mathrm{w}_{acc}}$$

The recursive step first does some processing and then calls an internal judgment to actually perform conditional control flow checks, called $\vdash_{w \vee c-cond}$. Specifically, this initial processing can be summarized like this:

1. First, the rule gets all of the components that are collideable with the current argument, grouping their names in a list. These names are inserted $\Psi(\mathrm{CSW})$ times if the component is part of the same walk as the node that is being processed right now, and 1 time otherwise. If the $\Psi(\mathrm{CSW})$ parameter is 0, then collides are only possible in different walks. If it's 1, then the parameter has no effect. If it's greater than 1, then there's a positive bias for collides to happen on the same walk. Regardless, it's still always possible to collide on different walks.
2. Second, the symbol $\cdot$ is used to indicate a new walk. It's going to be repeated in a list a number of times, calculated as the maximum between 1 and the number obtained with this formula: $\lceil \mathrm{w}_{acc} * \Psi(\mathrm{WOCB}) \rceil + 1$. Essentially, the higher w_{acc} is, the more likely it is to

perform a walk. This is influenced by the Walk Over Collide Bias, $\Psi(\text{WOCB})$, which is always in the range $(0, 2]$. Note that w_{acc} can be negative, due to the way it's deduced in a walk, but because of the mentioned maximum between the calculated number and 1, there is always a minimum of one $\cdot$ in the list, so a walk is always possible.

3. Then, these two lists are concatenated and a random element is sampled from them. If it's $\cdot$, then it should be a walk, and if it's an identifier, then it should be a collide. This is interpreted in the $\vdash_{w \vee c-cond}$ judgment, which is called next and has its returns passed along the outputs of the current $\vdash_{walk-or-collide}$ call.

The rule described is defined like this:

$$\text{fn-deps} := (\gamma_C \to (\text{dep-by} \equiv \{\emptyset\}) \wedge (\text{dep-on} \equiv \{\emptyset\}))$$
$$\text{collideable-}\Delta_C := \text{filter-values } (\gamma_C \to (\text{vn?} \equiv \cdot) \wedge (\tau \equiv \text{type}) \wedge (\text{fn-deps } \gamma_C) \wedge \text{col?})\ \Delta_C$$
$$\text{fn-csw} := (\text{n-id} \to \text{let } ((\text{wn, t, cs, r?}) := \Delta_N(\text{n-id})) \text{ in } ((\text{wn} \equiv \text{walk-id})\ ?\ \Psi(\text{CSW}) : 1))$$
$$\text{collide-holes} := \text{append-all } (\text{map } ((\text{c-id} ; \gamma_C) \to \text{repeat c-id } (\text{fn-csw n-id}))\ \text{collideable-}\Delta_C)$$
$$\text{num-bind-holes} := \max(1, \lceil \text{w}_{acc} * \Psi(\text{WOCB}) \rceil + 1)$$
$$\text{all-holes} := \text{append collide-holes } (\text{repeat} \cdot \text{num-bind-holes})$$
$$\text{filled-id?} = \xi(\text{all-holes})$$
$$\frac{\Psi\ ;\ \Theta\ ;\ \Delta\ ;\ \text{b}\ ;\ \text{w}_{acc}\ ;\ \text{walk-id}\ ;\ \text{filled-id?}\ ;\ \text{type}\ ;\ \text{name}\ ;\ \text{next} \vdash_{w \vee c-cond} \Delta'\ ;\ \text{w}'_{acc}}{\Psi\ ;\ \Theta\ ;\ \Delta\ ;\ \text{b}\ ;\ \text{w}_{acc}\ ;\ \text{walk-id}\ ;\ ((\text{type, name}) : \text{next}) \vdash_{walk-or-collide} \Delta'\ ;\ \text{w}'_{acc}}$$

And now for the internal judgment $\vdash_{w \vee c-cond}$. It gets the following parameters:

- Ψ: Associative list of parameters.
- Θ: Type environment.
- Δ: Walk environment.
- b: Set of base types.
- w_{acc}: Remaining steps before walks are only allowed to take backward steps.
- walk-id: The walk identifier for the node being currently processed.
- filled-id?: Either a $\cdot$, indicating a walk, or the identifier for a component to collide with.
- type: The current argument's type.
- name: The current argument's name.
- next: The next arguments to be processed.

And outputs the following:

- Δ': A modified walk environment, with the extensions provided by any new walks and collides.
- w'_{acc}: Updated w_{acc} to reflect new walks.

The first case represents a walk. For it to happen, it's required that filled-id? $\equiv \cdot$ and that either $\mathrm{w}_{acc} > 0$ or the argument type is not a base type. In that case, a new walk is created, rooted at the current argument's type and with that argument's name being assigned to a component in the first node. After the walk is done, the computation continues to the next argument, calling back the original $\vdash_{walk-or-collide}$ judgment. The number of steps in the walk is bounded by the Main Walk Length Maximum ($\Psi(\text{MWLM})$), and the branching depth is sampled from a geometric distribution with probability $\frac{1}{2}$. That is, there's a 50% chance that the depth is 0, then a 25% chance that it's 1, then a 12.5% chance that it's 2, and so on. That depth is then bounded by the Maximum Depth ($\Psi(\text{MD})$). This case is defined below:

$$\frac{\begin{array}{c}\text{filled-id?} \equiv \cdot \\ ((\mathrm{w}_{acc} > 0) \vee (\text{type} \equiv \neg\vec{\pi})) \\ \text{geom-depth} \sim_{\xi} \text{Geom}(\frac{1}{2}) \\ \text{depth} := \min(\text{geom-depth}, \Psi(\text{MD})) \\ \text{num-steps} := \xi([((\text{type} \equiv \neg()) \ ?\ 0 : 1)..{=}\Psi(\text{MWLM})]) \\ \Psi\ ;\ \Theta\ ;\ \Delta\ ;\ \text{b}\ ;\ \text{name}\ ;\ \text{type}\ ;\ \text{depth}\ ;\ \mathrm{w}_{acc}\ ;\ \text{num-steps} \vdash_{walk} \Delta'\ ;\ \mathrm{w}'_{acc} \\ \Psi\ ;\ \Theta\ ;\ \Delta'\ ;\ \text{b}\ ;\ \mathrm{w}'_{acc}\ ;\ \text{walk-id}\ ;\ \text{next} \vdash_{walk-or-collide} \Delta''\ ;\ \mathrm{w}''_{acc}\end{array}}{\Psi\ ;\ \Theta\ ;\ \Delta\ ;\ \text{b}\ ;\ \mathrm{w}_{acc}\ ;\ \text{walk-id}\ ;\ \text{filled-id?}\ ;\ \text{type}\ ;\ \text{name}\ ;\ \text{next} \vdash_{w \vee c-cond} \Delta''\ ;\ \mathrm{w}''_{acc}}$$

This second case could have been a walk, but due to w_{acc} being non-positive and the type being a base type, it's not possible to create one. The accumulator being non-positive makes it so that only backward steps are allowed, but as the type is a base, there are no backward steps available. A 0 step walk is only allowed when rooted on the type $\neg()$, which here is not being considered a base type. The computation simply continues to the next argument:

$$\frac{\begin{array}{c}\text{filled-id?} \equiv \cdot \\ \neg((\mathrm{w}_{acc} > 0) \vee (\text{type} \equiv \neg\vec{\pi})) \\ \Psi\ ;\ \Theta\ ;\ \Delta\ ;\ \mathrm{w}_{acc}\ ;\ \text{walk-id}\ ;\ \text{next} \vdash_{walk-or-collide} \Delta'\ ;\ \mathrm{w}_{acc}\end{array}}{\Psi\ ;\ \Theta\ ;\ \Delta\ ;\ \text{b}\ ;\ \mathrm{w}_{acc}\ ;\ \text{walk-id}\ ;\ \text{filled-id?}\ ;\ \text{type}\ ;\ \text{name}\ ;\ \text{next} \vdash_{w \vee c-cond} \Delta'\ ;\ \mathrm{w}'_{acc}}$$

Lastly, the case in which a collide happens. Simply replace the collided component's variable name with the parameter and then continue to the next argument:

$$\frac{\begin{array}{c}\text{filled-id?} \not\equiv \cdot \\ \text{fn-col} := ((\text{n-id}, \cdot, \tau, \text{dep-by}, \text{dep-on}, \text{col?}) \to (\text{n-id}, \text{name}, \tau, \text{dep-by}, \text{dep-on}, \text{col?})) \\ \text{collided-}\Delta := (\Delta_W, \Delta_N, (\text{update}\ \Delta_C\ \text{filled-id?}\ \text{fn-col}), \Delta_G) \\ \Psi\ ;\ \Theta\ ;\ \text{collided-}\Delta\ ;\ \mathrm{w}_{acc}\ ;\ \text{walk-id}\ ;\ \text{next} \vdash_{walk-or-collide} \Delta'\ ;\ \mathrm{w}'_{acc}\end{array}}{\Psi\ ;\ \Theta\ ;\ \Delta\ ;\ \text{b}\ ;\ \mathrm{w}_{acc}\ ;\ \text{walk-id}\ ;\ \text{filled-id?}\ ;\ \text{type}\ ;\ \text{name}\ ;\ \text{next} \vdash_{w \vee c-cond} \Delta'\ ;\ \mathrm{w}'_{acc}}$$

4.7.2.6 Maybe move binds to the outer scope, so they're usable to future binds ($\vdash_{float-R}$)

After a bind has been done, its internal command can be subjected to a Float-R transformation. This is a transformation based on the (FLOAT-R) rule described in 2.1.10.2. Specifically, it works by applying that rule, moving from the right side of the equality to the left side. The condition that the moved bind's continuation is not present in the outer command is lifted, and therefore the commands before and after the transformation are different.

This transformation was defined in an attempt to make it so that a bind can be brought to a larger scope than the one it was created in. Actually, if this transformation is not applied, then every bind captured by static contexts in a term will be used immediately at the jump at the end of the chain. By bringing some of these internal binds to an outside scope, it's possible for them to be used by other commands that are yet to be generated, which can increase diversity.

The judgment $\vdash_{float-R}$ performs this transformation, and it takes the following parameters:

- Ψ: Associative list of parameters.
- α: Uniformly sampled random score in the range $[0, 1]$.
- Ξ: List of taboo variables. For a bind to be floated to the outside, none of the variables used inside its internal command can be in this list.
- cmd: The CPS calculus command currently being processed. The recursion is on this parameter.

And it returns a modified command, without the floated binds, and a list containing the binds that were floated.

The base case is for a jump. It's the end of the recursion, and the same jump is returned together with an empty list.

$$\frac{}{\Psi\ ;\ \alpha\ ;\ \Xi\ ;\ (k\langle\vec{x}\rangle) \vdash_{float-R} (k\langle\vec{x}\rangle)\ ;\ nil}$$

The first recursive case deals with the event of a bind that was not floated. This can happen either because $\alpha \leq \Psi(\text{FRT})$ (Fold Right Threshold) or because one of the variables used in the bind's internal command is present in the forbidden list Ξ. In this case, sample a new α, add the current continuation name to Ξ and move to the next bind, following the static context:

$$\frac{\begin{array}{c}((\alpha \leq \Psi(\text{FRT})) \vee (\Xi \cap (\text{get-free-vars N}) \not\equiv \{\emptyset\})) \\ \Psi\ ;\ \xi(\mathbb{R}^u)\ ;\ \Xi \cup \{k\}\ ;\ \text{M} \vdash_{float-R} \text{M}'\ ;\ \text{floated}\end{array}}{\Psi\ ;\ \alpha\ ;\ \Xi\ ;\ (\text{M}\ \{k\langle\vec{x}\rangle = \text{N}\}) \vdash_{float-R} (\text{M}'\ \{k\langle\vec{x}\rangle = \text{N}\})\ ;\ \text{floated}}$$

The second recursive case deals with a bind that was floated. In this case, sample a new α and continue to the next command, but do not add the continuation name to the forbidden

list, because floated continuations are still accessible by the inner commands in which they were originally present. At the end, remove the floated bind from the original command and add it to the floated list:

$$\frac{\begin{array}{c}\neg((\alpha \leq \Psi(\text{FRT})) \vee (\Xi \cap (\text{get-free-vars N}) \not\equiv \{\emptyset\})) \\ \Psi \; ; \; \xi(\mathbb{R}^u) \; ; \; \Xi \; ; \; \text{M} \vdash_{float-R} \text{M}' \; ; \; \text{floated}\end{array}}{\Psi \; ; \; \alpha \; ; \; \Xi \; ; \; (\text{M} \; \{k\langle\vec{x}\rangle = \text{N}\}) \vdash_{float-R} \text{M}' \; ; \; ((\{k\langle\vec{x}\rangle = \text{N}\}) : \text{floated})}$$

Lastly, the get-free-vars function is just an implementation of the FV function as defined in 2.1.10.2, but this time outside of Redex. It's defined below, using Haskell-like functional pseudocode:

get-free-vars :: command $\rightarrow$ {identifier}

get-free-vars $(k\langle\vec{x}\rangle) \; = \; \{k\} \cup (\text{to-set } \vec{x})$

get-free-vars (outer $\{k\langle\vec{x}\rangle = \text{inner}\}$)

$= \; ((\text{get-free-vars outer}) \setminus \{k\}) \cup ((\text{get-free-vars inner}) \setminus (\text{to-set } \vec{x}))$

4.7.3 Organize the jump and its binds into a CPS command ($\vdash_{cmd-build}$)

At this point, all the binds for a jump have been generated and organized into a list. All that's left to do is to create the jump, join it with the binds and then nest everything into a command as defined by the CPS-Calculus CFG. In order to do that, $\vdash_{cmd-build}$ is defined. It only has a single case, and takes the following parameters:

- Γ: The variables environment.
- $\neg\vec{\tau}$: The node's type.
- χ: A list containing the identifiers for the node's components.
- ς: An associative list mapping a component identifier to its flag in the execution plan.
- ω: List of binds. Each bind has format $(\{k\langle\vec{x}\rangle = N\})$, where k is an identifier, $\vec{x}$ is a list of identifiers and N is an inner command.
- Π: An associative list mapping the name of a recently bound continuation to its type. This contains only continuations defined by the current node's execution plan and floated binds that are at the same level as these continuations.
- Σ: An associative list mapping a component-id to the name of the continuation it defined with a bind. This does not contain floated binds.

And it returns a CPS-calculus command, as given by the CFG in 2.1.10.

It performs these steps: first, it creates a list of assigned names, where for each component identifier $\mathrm{c} \in \chi$, if it originated a bind, then return the name of that bind, and otherwise return its flag in the execution plan. The elements in the resulting list will therefore consist of either a name, coming from either the variable name field in the component or from the bind that component originated, or a $\cdot$ or $\circ$ symbol to indicate an environment fetch. The $\cdot$ originally indicates a bind, but if the type of that component is a base type, then a bind is impossible, and in that case these are also going to be considered as a fetch here.

After, the variables environment is extended to include the new bound continuations. Both those created from the execution plan and those that were floated are included in this extension.

Next, fetch the remaining names from the environment. If the extended environment has at least one variable of that type, randomly pick from the options. If there are none, then use a free variable. The free variable case can only happen with free variables, so when $\mathrm{f?} \equiv \cdot$.

And lastly, nest the binds and create the jump at the end by using a recursive auxiliary function, to be defined after the judgment rule. The judgment is formalized as follows:

$$\frac{\begin{array}{c}\mathrm{fn}_1 := (\mathrm{c}\ ;\ \tau \to ((\varsigma(\mathrm{c}) \equiv \cdot) \wedge (\tau \equiv \neg\vec{\pi}))\ ?\ \Sigma(\mathrm{c}) : \varsigma(\mathrm{c})) \\ \text{assigned-names} := \text{map}\ \mathrm{fn}_1\ \chi\ (\neg\vec{\tau} : \vec{\tau}) \\ \Gamma' := \text{fold}\ (\mathrm{k}\ ;\ \text{env} \to \text{update env}\ \Pi(\mathrm{k})\ (\mathrm{ks} \to (\mathrm{k} : \mathrm{ks})))\ \Gamma\ (\text{keys}\ \Pi) \\ \mathrm{fn}_2 := (\tau \to \xi((|\Gamma'(\tau)| > 0)\ ?\ \Gamma'(\tau) : \{\xi_{id}\})) \\ \mathrm{fn}_3 := (\mathrm{f?}\ ;\ \tau \to (((\mathrm{f?} \equiv \cdot) \vee (\mathrm{f?} \equiv \circ))\ ?\ (\mathrm{fn}_2\ \tau) : \mathrm{f?})) \\ \text{jump-ids} := \text{map}\ \mathrm{fn}_3\ \text{assigned-names}\ (\neg\vec{\tau} : \vec{\tau}) \\ \text{out-cmd} := \text{nest-command jump-ids (reverse}\ \omega) \end{array}}{\Gamma\ ;\ \neg\vec{\tau}\ ;\ \chi\ ;\ \varsigma\ ;\ \omega\ ;\ \Pi\ ;\ \Sigma \vdash_{cmd-build} \text{command}}$$

And to create a CPS command, the nest-command function is used. It gets as parameters the identifiers to be used for the jump and the list of binds and it returns a CPS command. It's defined below using Haskell-like functional pseudo-code:

$$\begin{array}{lll} \textbf{nest-command} :: [\text{identifier}] \to [\text{binds}] & \to \text{command} & \\ \text{nest-command}\ (\text{cont} : \text{args})\ nil & = & (\text{cont}\ \langle\ \text{args}\ \rangle) \\ \text{nest-command jmp}\ ((\{\mathrm{k}\langle\vec{\mathrm{x}}\rangle = \mathrm{N}\}) : \text{b-tl}) & = & ((\text{nest-command jmp b-tl})\ \{\mathrm{k}\langle\vec{\mathrm{x}}\rangle = \mathrm{N}\}) \end{array}$$

4.8 Conclusion

In this chapter, an algorithm for the stochastic generation of CPS-calculus commands was formalized. This formalization did not aim to be directly translatable into an efficient implementation, but to provide the theoretical foundations for reasoning about the algorithm.

This algorithm was context-directed, basing itself on performing random walks with $\vdash_{walk}$ and then processing them step by step with $\vdash_{cmd}$, creating new walks in the process and

moving on to a next step, ideally until the walks reach a base type (including $\neg()$ in that category here). Going towards the base types is an attempt to progressively add usage constraints to the variables such that, when inferring their types, they are found to be as specialized as possible, bringing their inferrable types closer to the "real"type used in the generation procedure for that variable.

By developing $\vdash_{params}$, it's possible to greatly increase command diversity across generation instances, as new parameters related to boundaries and probabilities are processed with every call. Likewise with the generation of a type environment with $\vdash_{type-env}$, as this results in every command being allowed to explore a different subset of CPS-calculus type space.

5 Property Testing

With a CPS-calculus variant mechanized and a CPS terms generator for that variant formalized and implemented, it's possible to use these two together to test semantic properties of the calculus. First, however, it may be useful to test a few properties of the generator itself, for instance if the terms created by it are indeed syntactically recognized as CPS commands and if all terms outputted by it are well-typed.

The first section in this chapter will inform about the common top-level parameters supplied to the generator across all tests and of the hardware these properties were tested on. The number of performed tests will change between the experiments, so that metric will be discussed in the following sections.

The second section will be reserved to describe tests regarding the generator itself to check if its image has some desired properties, such as only generating well-typed terms, for example. The generator calls resulting from these tests will also serve to collect some metrics, which can inform the impact different parameters have on variable re-usage, ratio of discarded nodes or term size.

The third section will be about using the generator to test the type preservation property of the jump reduction and also of the call-by-name and call-by-value translations from the λ-calculus to the CPS-calculus. These properties have already been proven correct (TORRENS; ORCHARD; VASCONCELLOS, 2024), so this experiment is less about making a breakthrough in an unexplored topic and more about showing the ways this generator can be used to test properties.

This chapter's goal is to merely define the tests being performed and the test environments, as the results obtained from them are discussed only in the next chapter, although some of the findings are briefly anticipated here as well.

5.1 Experiment settings

All experiments discussed were run in DrRacket IDE using a memory limit of 8192MB and with the Chez Scheme Racket implementation set to version 9.0. The main machine utilized was a 16GB M4 MacBook Air running MacOS. Select tests were also run on a 16GB NixOS Linux laptop running on an Intel i7-11390H, due to it having a large enough swap partition to assist in the debugging of memory-related problems. All of the metrics analyzed were collected using the M4 machine.

The generator across all tests was configured with the following parameters:

1. MNST ($\mathbb{N}^+$) = 8
2. MBSF ($\mathbb{N}^+$) = 16
3. MTSF ($\mathbb{N}^+$) = 64
4. MBSR ($\mathbb{N}^+$) = 4
5. MTSR ($\mathbb{N}^+$) = 4
6. MTRB ($\mathbb{N}^+$) = 8
7. MTRS ($\mathbb{N}$) = 1
8. MWLM ($\mathbb{N}^+$) = 8
9. BWLM ($\mathbb{N}^+$) = 4
10. FLMD ($\mathbb{N}$) = 4
11. BRMD ($\mathbb{N}$) = 2
12. LNMD ($\mathbb{N}$) = 2
13. MCDA ($\mathbb{N}$) = 8
14. MCWA ($\mathbb{N}^+$) = 16
15. MD ($\mathbb{N}$) = 1
16. MSBR ($\mathbb{N}$) = 64

There was a noticeably high sensitivity to parameters regarding memory consumption during the generation process, with some calls originally resulting in out-of-memory errors by going above the limit set in DrRacket. Further experimentation and profiling are needed in order to identify the reasons for such a behavior and how to fix it, but that is left for future works utilizing this specific implementation of the generator algorithm. To counter this in the present work, a term size-based failure trigger was introduced to the generator, where if a term that is currently being sampled exceeds a number of binds, then that sampling generation fails and a new one is called in its place. This fail-safe was utilized for a second purpose as well: in some tests, it was reduced to a much lower value, not because of memory constraints, but because the Redex CPS-calculus typing judgment was deemed too slow for very large terms. This limit changes across experiments, so it will be mentioned progressively.

5.2 Generator properties

Four properties of the generator's image were tested. They are:

1. All terms are recognized by the Redex mechanization as CPS commands.
2. All terms are in normal form with respect to the Garbage Collection Relation (TORRENS; ORCHARD; VASCONCELLOS, 2024).
3. All terms are well-typed (THIELECKE, 1997a).
4. All of the free variables in a term either have the free variable prefix or is a "halt".

About the last one, although this isn't present in the formalization, the implementation prefixes all variable names with either "cont-", "param-"or "FV-", to signal that they were defined as a continuation, a parameter or that they are being used as a free variable, respectively. As a special case, one free variable can also be named "halt", specifically the variable in subject position of the jump in head position, if it's decided in the beginning of the generation that the command is going to be in head normal-form.

Starting with the first one: the Redex function redex-match? is used to check if the output of the generator is a CPS command. This property is tested a total of 1000 times, each of them resulting in a new sample, and with a limit of 8192 binds. Uniquely to this test, some information regarding its generation process is stored in order to be analyzed, specifically:

- The final command.
- The elapsed time that was taken to generate the command.
- The parameter table Ψ.
- The type environment Θ.
- The first type of the generation, sampled in the top level $\vdash_{CPS}$ to initialize the walk environment.

All of these except for the final command were also separately saved for failures, where the number of binds exceeded the imposed limit.

The second test checked if the Garbage Collection Relation implemented in 3.4, when used with Redex's apply-reduction-relation together with any term outputted by the generator, does not lead to any different term. This property was tested 100 times and with a limit of 256 binds. It's worth noting that being in normal form with respects to the garbage collection means that all binds that were defined were also used at least once, but it's still possible to define a continuation with unused parameters. Also, it may be desirable to have commands with unused

binds, and this can be achieved by appending this generator with a swap transformation, replacing a bind from the jump that defined it with either a free variable or another continuation of the same type in the environment. This, however, is left for future works.

The third experiment is related to typing. It was also tested 100 times and with a limit of 256 binds.

The fourth and final experiment for this section was also tested 100 times and with a limit of 256 binds. This test did not use any of the Redex implementation, instead manipulating the CPS calculus terms as Racket symbols directly.

No counterexample was found across any of the tests. The data obtained from the $1\,000$ commands generated in the first test will be analyzed in the next chapter.

5.3 Type preservation

With the generator itself tested, it's time to use it to perform experiments on CPS-calculus. In this section, all of the tested properties are related to its typing semantics, more specifically, regarding type preservation.

5.3.1 Jump reduction

The first property being tested is the type preservation of the jump reduction (TORRENS; ORCHARD; VASCONCELLOS, 2024). Specifically, given a well-typed CPS command t_1, if t_1 is not in jump normal form, then the t_2 obtained from $t_1 \rightsquigarrow_j t_2$ is also well-typed.

When a reduction relation is applied in Redex, all of the possible reductions are returned as a list. In other words, all possible evaluation contexts are reduced independently and returned as a list of possible new terms. The property being checked tests all of the returned possible t_2 for a single t_1, and as such it's a very expensive check. Due to this cost related to time constraints, the maximum size of the terms was set to 128 binds, and 100 initial commands were generated.

All of the tests were successful.

5.3.2 Translations from λ-calculus to CPS-calculus

The second property being tested is the type preservation of the translations from λ-calculus to CPS-calculus, in both call-by-name and call-by-value forms. In order to implement the tests in this section, an inference judgment for the λ-calculus typing context was implemented. It's defined below:

```
(define-judgment-form CPS-Calculus-λ
  #:mode (λinfer O I O)
  #:contract (λinfer λΓ e A)

  [(where A_0 ,(gensym "typ-"))
```

```
   ------------------------------------------------------------------------------ "Var"
   (λinfer ((id_0 A_0)) id_0 A_0)]

  [(λinfer λΓ_0 e_0 A_1)
   (where A_2 ,(gensym "typ-"))
   (where A_0 ,(second (or (assoc (term id_0) (term λΓ_0)) (term (id_0 A_2)))))
   (where λΓ_1 ,(remf (compose1 (curry equal? (term id_0)) car) (term λΓ_0)))
   ------------------------------------------------------------------------------ "Abs"
   (λinfer λΓ_1 (λ id_0 \. e_0) (A_0 -> A_1))]

  [(λinfer ((id_2 A_2) ...) e_0 A_0)
   (λinfer ((id_3 A_3) ...) e_1 A_1)
   (where A_6 ,(gensym "typ-"))
   (where id_4 ,(gensym "app-"))
   (where ((id_5 A_5) ...)
          (λunification ((A_0 (A_1 -> A_6)) (id_2 A_2) ... (id_3 A_3) ...) ()))
   (where A_4
          ,(second (or (assoc (term id_4) (term ((id_5 A_5) ...))) (term (A_6 A_6)))))
   (where λΓ_0 ,(remove-duplicates (term (λsubstitute-into-env
                                          ((id_2 A_2) ... (id_3 A_3) ...)
                                          ((id_5 A_5) ...)))))
   ------------------------------------------------------------------------------ "App"
   (λinfer λΓ_0 (e_0 e_1) A_4)])
```

This implementation was inspired by Algorithm W (MILNER, 1978), an inference algorithm formalized for the Hindley–Milner type system of λ-calculus. It differs from it on some points, however, for example in having the typing context $\lambda\Gamma$ as an output rather than input and also in forgoing the substitutions set as a fourth variable in the judgment's contract in favor of using the two independently outputted contexts in the application rule to perform the required unification.

The properties being tested here are closely related to each other, and in fact each is built on top of the prior. All of them will have a generator size limit set to only 128 binds, due to the expensive nature of the operations being performed.

Starting with the simplest test, which will be the base for further ones: using only 20 samples, it's desired to check if λ expressions returned by CPS-to-λ can have their types inferred by λinfer. In other words, to check if it's possible to find both a context and a suitable type for the expression, provided only the expression itself. The relation that maps this is defined below:

```
(define-relation CPS-Calculus-λ
  CPS-to-λ-well-typed ⊆ cmd
  [(CPS-to-λ-well-typed cmd_0)
   (where e_0 (CPS-to-λ cmd_0))
   (judgment-holds (λinfer λΓ e_0 A))])
```

The tests were successful. Afterwards, this relation is going to be extended to perform the translation back to CPS-calculus and then check if the CPS command obtained is also well-typed, as in if it can have its types inferred. Both call-by-name and call-by-value versions of this relation were defined, each using the corresponding translations, as implemented in 3.5. In this chapter, only the CBN versions will be written, but both were tested and the only difference is in the CBN/CBV functions being called:

```
(define-relation CPS-Calculus-λ
  CPS-to-λCBN-to-CPS-well-typed ⊆ cmd
  [(CPS-to-λCBN-to-CPS-well-typed cmd_0)
   (where e_0 (CPS-to-λ cmd_0))
   (judgment-holds (λinfer λΓ_0 e_0 A_0))
   (where cmd_1 (λCBN-to-CPS k e_0))
   (judgment-holds (typing Γ_1 cmd_1))])
```

Both were tested with another 20 samples each, all successful. Then, each gave origin to a test about type preservation, which was proven in Torrens, Orchard e Vasconcellos (2024). This property is related to the last translations in 2.1.10.5, where an entire λ-calculus typing judgment is transformed into a corresponding CPS-calculus one. Specifically, if the statement $\lambda\Gamma \vdash_\lambda e : B$ is derivable, then those translations give a way to process the context $\lambda\Gamma$ and the expression type B by creating a Γ that types the CPS-calculus command obtained by translating the expression e. In other words, the property states that if $\lambda\Gamma \vdash_\lambda e : B$ is derivable, then the translation of that statement is also derivable (TORRENS; ORCHARD; VASCONCELLOS, 2024). As opposed to the previous relations in this section, this property was defined as a judgment:

```
(define-judgment-form CPS-Calculus-λ
  #:mode (CPS-to-λCBN-to-CPS-type-preserving I)
  #:contract (CPS-to-λCBN-to-CPS-type-preserving cmd)

  [(where e_0 (CPS-to-λ cmd_0))
   (λinfer λΓ_0 e_0 A_0)
   (where Γ_0 (λCBN-to-CPS-env λΓ_0 A_0))
   (where cmd_1 (λCBN-to-CPS k e_0))
   (typing Γ_1 cmd_1)
   (where ((id_0 τ_0) ...) (unnest-constraints (unification (union-envs Γ_0 Γ_1) ·)))
   (where Γ_3 (substitute-into-env Γ_0 ((id_0 τ_0) ...)))
   (side-condition (equal? Γ_0 Γ_3))
   ----------------------------------------------------------------
   (CPS-to-λCBN-to-CPS-type-preserving cmd_0)])
```

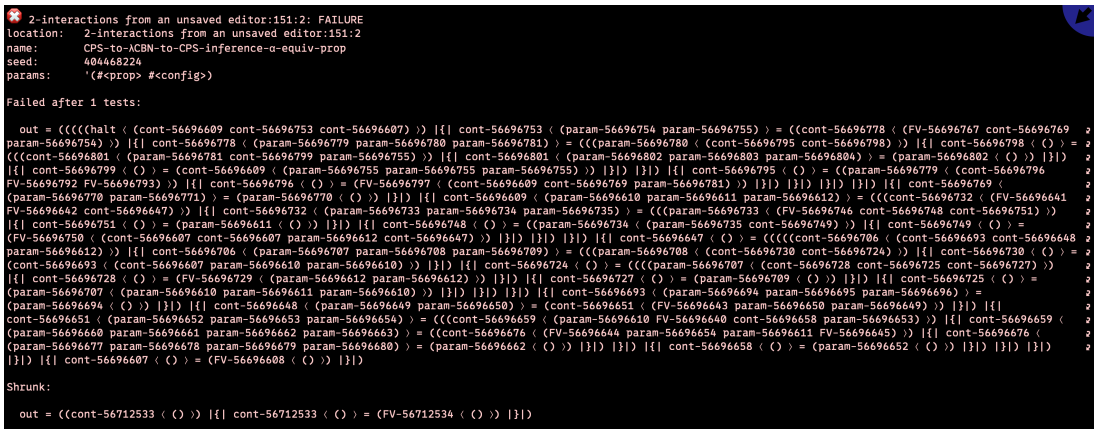
This one was tested 100 times each, and again all tests were successful. So far these successes were expected from a semantics standpoint, as this property has already been proved, but these experiments are still valuable to try and spot errors in either the formalization or in some required functions, such as the type checkers.

Regarding the experiment itself, it starts by first translating a CPS command to a λ expression and then typing it, as was done in previous tests. If it's well-typed, call the translations to CPS-calculus to obtain a typing context and a command and then type that command. The type inference will return another typing context that must unify with the one obtained by the translation. Lastly, the unified constraints in solved form, when substituted into the Γ obtained by translating the λ-calculus inferred context, must not change that context. Essentially this is checking if the translated context is either equal to or an unifiable specialization of the inferred one, because if it is and the inferred context types the command, then the translated context also types it, therefore checking the property.

Lastly, one more property is going to be tested, which is very similar to the previous one. This one aims to see if the inferred context $\hat{\Gamma}$ and the translated context Γ can be considered α-equivalent under an order-agnostic version of equivalence. The test will be a success if:

- Γ and $\hat{\Gamma}$ have the same number of elements, and the same collection of keys (variable names) in the context. The keys don't need to be in the same order.
- For every k in any of them, $\Gamma(k) \equiv_\alpha \hat{\Gamma}(k)$.

This test quickly failed, but in doing so, Rackcheck's automatic shrinking capacity can be showcased. As the generator was implemented in a pure and functional way, Rackcheck is able to correctly reduce the size of the generated failing term without previous side effects interfering with the process.



The bigger command on top is what was generated, which didn't validate the property being tested. The much smaller command on the bottom is what was obtained by continuously shrinking the size of the upper command in a way that maintains the test failure, repeating until it's no longer possible to do that. This makes analyzing failing terms way easier, as it's possible to look only at a minimal example.

That failing property is a stricter version of the type preservation one, such that it's not sufficient for the translated context to be either equal to or an specialization of the context that was inferred based on the translated command. Now they must obey a more restrictive element-wise α-equivalence instead. The Redex encoding of the call-by-name version of this property can be found below:

```
(define-judgment-form CPS-Calculus-λ
  #:mode (CPS-to-λCBN-to-CPS-inference-α-equiv I)
  #:contract (CPS-to-λCBN-to-CPS-inference-α-equiv cmd)

  [(where e_0 (CPS-to-λ cmd_0))
   (λinfer λΓ_0 e_0 A_0)
   (where Γ_0 (λCBN-to-CPS-env λΓ_0 A_0))
   (where cmd_1 (λCBN-to-CPS k e_0))
   (typing Γ_1 cmd_1)
   (where (id_0 ...) ,(get-keys-env (term Γ_0)))
   (where (id_1 ...) ,(get-keys-env (term Γ_1)))
```

```
(side-condition ,(= (length (term (id_0 ...)))
                    (length (term (id_1 ...)))))
(side-condition ,(set=? (list->set (term (id_0 ...)))
                        (list->set (term (id_1 ...)))))
(side-condition ,(andmap (lambda (k)
                           (alpha-equivalent? CPS-Calculus-λ
                                              (get-from-env (term Γ_0) k)
                                              (get-from-env (term Γ_1) k)))
                         (term (id_0 ...))))
------------------------------------------------------------------- "Var"
(CPS-to-λCBN-to-CPS-inference-α-equiv cmd_0)])
```

The purpose behind this last experiment was not to test the type preservation property, but rather the relationship between the inference algorithms and the transformations. Specifically, as a counterexample was found near the end of the chain of premises and it's assumed that the testing code is correct, it's plausible to conclude that

$$\exists\, e\,.\, \lambda\Gamma \vdash_{\lambda-\text{infer}} e : B \implies \text{CPS-infer} \circ [\![e]\!]_{\{N,V\}} \not\equiv_\alpha [\![\lambda\Gamma,\ k : B]\!]_{\{N,V\}}$$

Where λ-infer is the λ-calculus inference judgment and CPS-infer is the CPS-calculus equivalent. That is, that there exists a λ expression that is typable by the λ inference judgment, but that when translated into a CPS command and then have that command typed through the CPS inference, the resulting typing context differs from the context obtained by simply translating the $\lambda\Gamma$ and B into a CPS Γ. Do consider in the above that if the N translation was chosen on the left side of the equivalence relation, then the right side also picks N, and likewise with V.

A counterexample was found, and therefore the above statement is true. Specifically, if the two contexts are obtained independently, they are unifiable, but they aren't the exact same. One of them is more specific than the other, as can be seen in the image below (variable names changed):

```
> (let* ([e (term (CPS-to-λ ((a ⟨ () ⟩)) |{| a ⟨ () ⟩ = (FVar ⟨ () ⟩) |}|)))]
         [Γ0
          (car (judgment-holds
                (λinfer λΓ ,e A)
                (λCBN-to-CPS-env λΓ A)))]
         [Γ1
          (car (judgment-holds
                (typing Γ (λCBN-to-CPS k ,e))
                Γ))])
    (values Γ0 Γ1))
'((· k (¬ (app-56758965))) FVar (¬ ((¬ (app-56758965)))))
'((· k k56758971«111425») FVar (¬ (k56758971«111425»)))
```

Interestingly, if the call-by-name functions are swapped for their call-by-value counterparts for this specific command, the obtained types are structurally equal (up to α-convertibility):

```
> (let* ([e (term (CPS-to-λ ((a ⟨ () ⟩)) |{| a ⟨ () ⟩ = (FVar ⟨ () ⟩) |}|)))]
         [Γ0
          (car (judgment-holds
                (λinfer λΓ ,e A)
                (λCBV-to-CPS-env λΓ A)))]
         [Γ1
          (car (judgment-holds
                (typing Γ (λCBV-to-CPS k ,e))
                Γ))])
    (values Γ0 Γ1))
'((· k (¬ (app-56759307))) FVar app-56759307)
'((· FVar a«111443») k (¬ (a«111443»)))
```

However, the property in general is also false for call-by-value. What this failure means for the inference algorithms, the translations or CPS-calculus' typing semantics is left for future inquiries.

5.4 Conclusion

In this chapter, tests utilizing the Redex mechanization and the Rackcheck command generator were devised. These tests were divided into two categories: generator tests and semantics tests.

The lack of counterexamples for the generator experiments suggests (but does not prove) that the CPS-calculus command generator is behaving as intended, at least within the scope of the tests.

On the other hand, not being able to falsify semantic experiments show the more problematic side of testing alone: the absence of counterexamples is frequently ambiguous, as it can signal a number of different things. Maybe the test code is incorrect and leads to false positives, or the random samples aren't expressive enough to cover faulty scenarios (which is specially concerning in the test concerning the type preservation for the translations, given the expressions being tested are actually derived from a CPS to λ translation), or the property is indeed correct and there are no counterexamples. This ambiguity of interpretation in the event of a success is the reason testing alone is only capable of proving that errors exist, but not that they do not (DIJKSTRA, 1970). However, the absence of counterexamples can still lead to some insights related to the possibility of a property being correct, which can be the first step towards a proof of its correctness.

Most of the semantic properties tested here were not falsified, but they were already previously proved, so that's to be expected. One of them, however, was not proved, and it resulted in a term that falsifies it, shrunk to its smallest size.

6 Results

This work's objectives included three pillars: mechanization, generation and property testing. All three were successfully explored, but at different levels of depth. In this chapter, the results obtained with respect to these goals will be analyzed.

6.1 Mechanization

The first goal was to implement a variant of the CPS-calculus in PLT Redex. Using the framework, the syntax for the non-linear, non-recursive and polyadic variant was mechanized (KENNEDY, 2007), alongside the Full Reduction (TORRENS; ORCHARD; VASCONCELLOS, 2024), typing judgment (THIELECKE, 1997a), and translations to and from λ-calculus (THIELECKE, 1997b) (THIELECKE, 1997a). All of these developments can be found in 3.

The typing judgment differed slightly from the specification, as it was expanded to include context inference. Research on this typing variant's properties are left as future work.

With this implementation, research on CPS-calculus using Redex is now possible, and as such the mechanization goals were successful.

6.2 Generation

Redex offers a procedure to generate random terms that satisfy a judgment, such as a type checker, but that procedure was found to be too general to work reliably for the polyadic variant studied in this work. As such, it was a major goal to formalize a custom generator for well-typed CPS-calculus commands.

This generator was formalized in chapter 4, and then implemented using Rackcheck. As tested later, this generator outputs only CPS-calculus commands that are well-typed and in normal-form with respects to the Garbage Collection Relation (TORRENS; ORCHARD; VASCONCELLOS, 2024), ensuring no binds generated are useless. It was formalized as a context-directed procedure, as CPS commands don't have types, but the variables in the commands do.

It works by performing random walks on a sampled finite type environment and then processing it backwards, creating a jump with every step and instantiating new binds on-demand, continuing the recursive procedure.

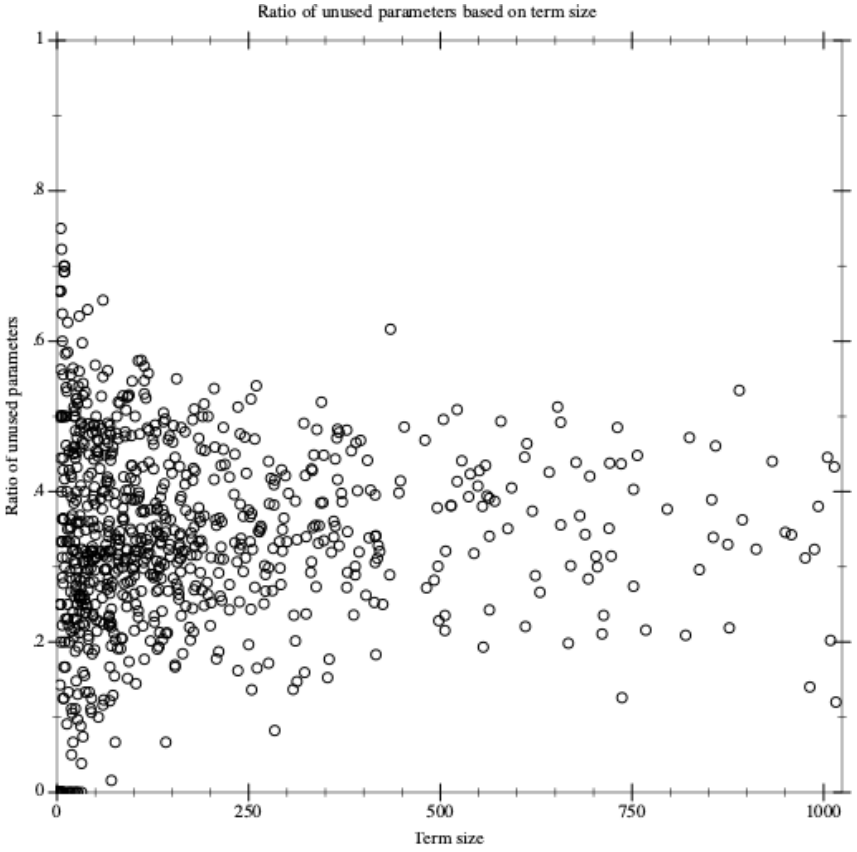
This goal was successfully met, although several improvements could be made. Despite generating commands of multiple sizes, its implementation is very memory-intensive on larger

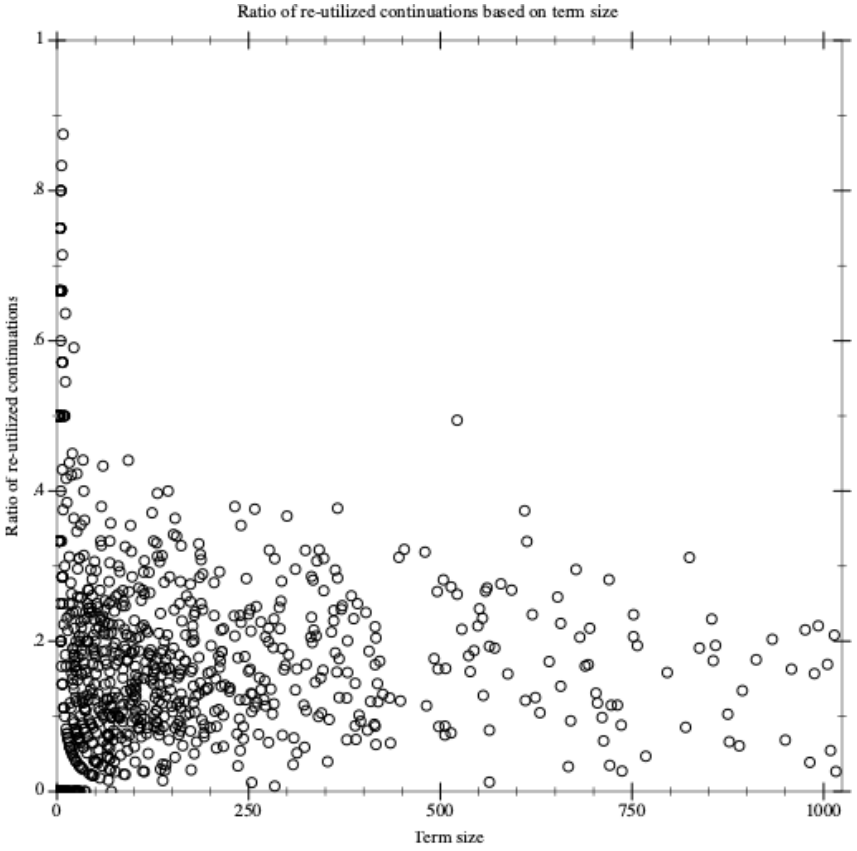
commands. However, this is just an area of improvement and it did not significantly interfere with the usability of the generator.

Concerning generator metrics, the first 1000 generated commands were found to have the following metrics:

- Quartiles indicating the size of the starting type: $(6, 12, 28)$.
- Quartiles for the size of the largest type in the type environment: $(15, 33, 57)$.
- Quartiles for the number of types in the type environment: $(10, 21, 38)$.
- Quartiles for the number of binds in a command, the metric used to represent term size: $(21, 82, 232)$.
- A total of 35 commands with size 0 were generated, the minimum command size.
- The largest generated command had 7052 binds.
- With a size maximum set to 8192 binds, in order to successfully generate 1000 commands, there were 52 failed attempts.
- The median time taken to successfully generate a command was 27.87ms, while the median time taken for generation failures (large commands that exceed the maximum number of binds) was 4566.58ms.

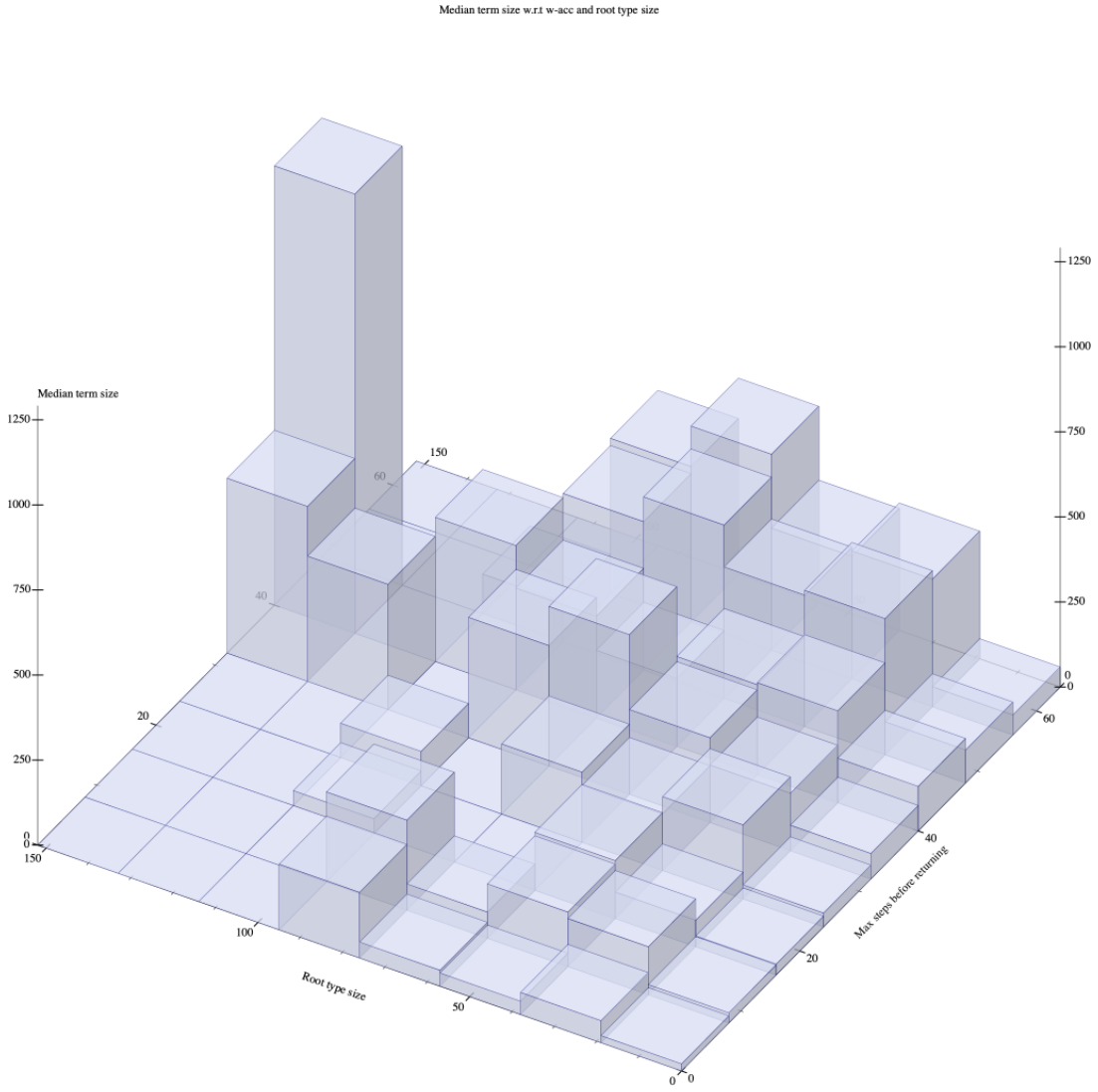
Regarding parameter utilization, the following graphs showcase the relationship between term size and parameters being unutilized and continuations being re-utilized, respectively:





Surprisingly, these graphs showcase a weaker than expected relationship between a term being large and it having a larger ratio of parameters left unused. The graph about binds being re-utilized did not break expectations.

Lastly, regarding term size and parameter sensitivity, the 3D graph shown below illustrates the command size impacts from the Maximum Steps Before Returning parameter (MSBR, which is the w-acc in the walk generator) and the size of the root type. This graph was created on a different set of 1000 commands:



As it can be seen, both the size of the initial type and the number of steps to be taken before returning have a positive influence on the size of the term.

6.3 Property testing

In chapter 5, several tests were devised. Some of them concerned properties of the generator's image itself, rather than a property about CPS-calculus, but others were indeed about its operational semantics.

Specifically, properties related to type preservation were tested. First, there was an attempt to falsify the jump reduction's type preservation. Then, efforts moved to do the same to the λ-calculus to CPS-calculus translations, using both call-by-name and call-by-value. These properties

were already proven correct (TORRENS; ORCHARD; VASCONCELLOS, 2024), but they were still tested as a proof of concept, showcasing the possibility of defining a property in Redex and then checking it using the generator's Rackcheck implementation.

All of the tests were devised in the previous chapter, and their successes or failures were also mentioned there. Regardless of the outcome of the tests, the ability to integrate the generator with Redex relations and judgments successfully connected the two other aspects of this work, and thus this goal was met.

7 Final Remarks

This work has made a number of developments in the topic of property-based testing for the CPS-calculus, meeting every objective set to be achieved and expanding on the corpus of knowledge in the field this work takes place in, aiding in future research projects.

By encoding CPS-calculus operational semantics in Redex, it was possible to demonstrate the utility of the newly formalized generator to automatically test properties of the calculus. Even if a future work leaves Racket and Redex, the general syntax used to formalize the generation algorithm allows for the re-implementation in other languages and frameworks, hence surpassing the scope here explored.

7.1 Conclusion

Property-based testing is very useful even in areas where a much greater emphasis is placed on rigorous mathematical proofs, because testing can catch false statements before an impossible proof is attempted. However, manually creating test cases in a way that maximizes test coverage is limited by bias and creativity, as one would often need to think of a possible path for falsifiability when creating the test cases, when hidden paths that would not have been thought of can exist.

To solve this, it's usually beneficial to be able to automatically generate random test cases. This work successfully achieved that for the CPS-calculus by formalizing a generator of well-typed terms. The algorithm presented in this work is language-agnostic, and as such can be implemented in any programming language. In this work, it was built in Racket and using Rackcheck, with the goal of integrating it with a CPS-calculus implementation that was mechanized using Redex. This mechanization involves reduction and typing semantics and translations to and from λ-calculus, all of which can be used to test properties.

This combination showcased how to use the developed stochastic generator to aid researchers in testing properties of CPS-calculus, which was the main goal behind this project.

7.2 Future Work

Several aspects of this work could be improved upon. First, the current Racket implementation of this generator is somewhat memory-intensive and could be optimized, as this could benefit its usability to create much larger commands. Related, the Redex mechanization is also somewhat slow, which hindered this work's ability to explore a higher number of samples during testing.

Still about implementation, it could be fruitful to move the generator to a proof assistant language, for example Rocq and using QuickChick (DÉNÈS et al., 2014). This would allow proofs to be made about the generator itself and the testing code being utilized together with it, all re-utilizing the same mechanization used to prove formal properties.

Other next steps would be to utilize what was built to test properties that so far have not yet been proven, therefore actually using the generator for the main purpose it was devised for. Also, it could be interesting to expand the ideas behind the generator to other variants of CPS-calculus, such as the recursive variant, or to other domains entirely.

It could also be useful to extend the generator by refining some aspects of it that interact with the calculus. Specially, a major point of improvement could be to type the variables as the generation happens in order to guide variable choices in jumps towards options that specialize the types the most. As tested during this work, all terms generated by the algorithm are well-typed, but when the types are independently inferred, type variables are often added, which represents either a base type or parts of the full types that were not able to be inferred through its usages. Because the algorithm attempts to make each type fully inferrable by moving parameters to base types or $\neg()$, but the current types are not tracked, several backward walks can be deemed unnecessary from the point of view of type specialization, which greatly increases command size and can reduce diversity.

Lastly, another possible next step would be to further study the CPS-calculus context inference algorithm utilized here. It differs from the checking rules presented in Thielecke (1997a), although it uses it as a base, and as such it could be interesting to study if it's a fully coherent extension to the original judgment.

Referências

ALFRED, V. A.; MONICA, S. L.; JEFFREY, D. U. *Compilers principles, techniques & tools*. [S.l.]: pearson Education, 2007.

APPEL, A. *Compiling with Continuations*. Cambridge University Press, 1992. ISBN 9780521416955. Disponível em: <https://books.google.com.br/books?id=0Uoecu9ju4AC>.

APPEL, A. W. *Compiling with continuations*. [S.l.]: Cambridge university press, 2007.

CARVALHO, M. de. *Construindo o saber: técnicas de metodologia científica*. [S.l.]: Papirus Editora, 1989. ISBN 9788530800710.

CHURCH, A. An unsolvable problem of elementary number theory. *American journal of mathematics*, JSTOR, v. 58, n. 2, p. 345–363, 1936.

CLAESSEN, K.; HUGHES, J. Quickcheck: a lightweight tool for random testing of haskell programs. In: *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*. [S.l.: s.n.], 2000. p. 268–279.

CURRY, H.; FEYS, R.; CRAIG, W. *Combinatory Logic, Volume 1 (Third printing)*. [S.l.]: North-Holland, 1974.

CUTLAND, N. *Computability: An introduction to recursive function theory*. [S.l.]: Cambridge University Press, 1980.

DÉNÈS, M.; HRITCU, C.; LAMPROPOULOS, L.; PARASKEVOPOULOU, Z.; PIERCE, B. C. Quickchick: Property-based testing for coq. In: *The Coq Workshop*. [S.l.: s.n.], 2014. v. 125, p. 126.

DIJKSTRA, E. W. Notes on structured programming. 1970.

FEITOSA, S. da S.; RIBEIRO, R. G.; BOIS, A. R. D. Generating random well-typed featherweight java programs using quickcheck. *Electronic Notes in Theoretical Computer Science*, Elsevier, v. 342, p. 3–20, 2019.

FELLEISEN, M.; FINDLER, R. B.; FLATT, M. *Semantics engineering with PLT Redex*. [S.l.]: Mit Press, 2009.

FETSCHER, B.; CLAESSEN, K.; PAŁKA, M.; HUGHES, J.; FINDLER, R. B. Making random judgments: Automatically generating well-typed terms from the definition of a type-system. In: SPRINGER. *European Symposium on Programming Languages and Systems*. [S.l.], 2015. p. 383–405.

GRAEFF, G.; MELLO, B.; DUARTE, D.; BRAGA, A.; BRAGA, S.; FEITOSA, S. A random generation of haskell programs applied to optimization testing in compilers. *Revista de Informática Teórica e Aplicada*, v. 31, n. 2, p. 20–29, 2024.

HINDLEY, J. R.; SELDIN, J. P. *Lambda-calculus and combinators: an introduction*. [S.l.]: Cambridge University Press, 2008.

HOPCROFT, J. *Introduction to Automata Theory, Languages, and Computation*. [S.l.]: Addison-Wesley, 2001.

IGARASHI, A.; PIERCE, B. C.; WADLER, P. Featherweight java: a minimal core calculus for java and gj. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, ACM New York, NY, USA, v. 23, n. 3, p. 396–450, 2001.

KENNEDY, A. Compiling with continuations, continued. In: *Proceedings of the 12th ACM SIGPLAN international conference on Functional programming*. [S.l.: s.n.], 2007. p. 177–190.

KLEIN, C.; CLEMENTS, J.; DIMOULAS, C.; EASTLUND, C.; FELLEISEN, M.; FLATT, M.; MCCARTHY, J. A.; RAFKIND, J.; TOBIN-HOCHSTADT, S.; FINDLER, R. B. Run your research: on the effectiveness of lightweight mechanization. *ACM SIGPLAN Notices*, ACM New York, NY, USA, v. 47, n. 1, p. 285–296, 2012.

MARTELLI, A.; MONTANARI, U. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, ACM New York, NY, USA, v. 4, n. 2, p. 258–282, 1982.

MCCARTHY, J. A. Automatically restful web applications: marking modular serializable continuations. In: *Proceedings of the 14th ACM SIGPLAN international conference on Functional programming*. [S.l.: s.n.], 2009. p. 299–310.

MERRO, M. On the observational theory of the cps-calculus. *Acta informatica*, Springer, v. 47, n. 2, p. 111–132, 2010.

MERRO, M.; SANGIORGI, D. On asynchrony in name-passing calculi. In: SPRINGER. *International Colloquium on Automata, Languages, and Programming*. [S.l.], 1998. p. 856–867.

MIDTGAARD, J.; JENSEN, T. P. Control-flow analysis of function calls and returns by abstract interpretation. In: *Proceedings of the 14th ACM SIGPLAN international conference on Functional programming*. [S.l.: s.n.], 2009. p. 287–298.

MILNER, R. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, v. 17, n. 3, p. 348–375, 1978. ISSN 0022-0000. Disponível em: <https://www.sciencedirect.com/science/article/pii/0022000078900144>.

PIERCE, B. C. *Types and Programming Languages*. [S.l.]: The MIT Press, 2002. ISBN 9780262303828.

RAMPAZZO, L. *Metodologia científica*. [S.l.]: Edições Loyola, 2005. ISBN 9788515024988.

ROMPF, T.; MAIER, I.; ODERSKY, M. Implementing first-class polymorphic delimited continuations by a type-directed selective cps-transform. In: *Proceedings of the 14th ACM SIGPLAN international conference on Functional programming*. [S.l.: s.n.], 2009. p. 317–328.

THIELECKE, H. Categorical structure of continuation passing style. University of Edinburgh. College of Science and Engineering. School of ..., 1997.

THIELECKE, H. Continuation semantics and self-adjointness. *Electronic Notes in Theoretical Computer Science*, Elsevier, v. 6, p. 348–364, 1997.

TORRENS, P.; ORCHARD, D. A.; VASCONCELLOS, C. On the operational theory of the cps-calculus: Towards a theoretical foundation for irs. *Proceedings of the ACM on Programming Languages*, ACM, 2024.

TURING, A. M. Computability and λ-definability. *The Journal of Symbolic Logic*, Cambridge University Press, v. 2, n. 4, p. 153–163, 1937.

YANG, X.; CHEN, Y.; EIDE, E.; REGEHR, J. Finding and understanding bugs in c compilers. In: *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*. [S.l.: s.n.], 2011. p. 283–294.