UNIVERSIDADE DE LISBOA
FACULDADE DE CIÊNCIAS
DEPARTAMENTO INFORMÁTICA

ALGORITHMS FOR INFINITE SESSION TYPES

Inês Maria Caldeira Sardinha

Mestrado em Informática

Dissertação orientada por:
Prof. Doutor Diogo Miguel Ferreira Poças
e co-orientado pelo Prof. Doutor Vasco Thudichum Vasconcelos

2022

Agradecimentos

Esta tese é o culminar de um percurso académico, longo e árduo, que não poderia ser feito sem o apoio de quem escolho aqui agradecer.

Em primeiro lugar, aos meus orientadores, professores Diogo Poças e Vasco Vasconcelos, pelo constante apoio e disponibilidade, por terem sempre mostrado um espírito crítico que incentivou a um trabalho cada vez melhor. Em especial ao professor Diogo, foi um ano de aprendizagem em que grande parte se deve a si. Muito obrigado!

À minha família, que esteve sempre presente e que mostrou um apoio incondicional. Aos meus pais, António e Georgina, e irmão, Jorge, que desde o início me apoiaram em todas as minhas decições mesmo quando pareciam erradas. Ao significado que deram a todas as conquistas, que no fim são tão minhas quanto vossas. Agradeço o incentivo e a educação por parte de todos, por terem celebrado as vitórias comigo e segurado nas derrotas!

Aos meus amigos, dentro e fora da faculdade, desde Campo Maior até Lisboa, passando nas Caldas da Rainha e em Alcochete, que me acompanharam ao longo deste percurso. Aos que nunca me deixaram sozinha e que tiveram sempre uma palavra de apoio e motivação para mostrar. Às amizades que são de sempre e às que mudaram o meu ano. A todos os que tornaram tudo mais fácil desde os primeiros dias de faculdade, obrigado!

Aos meus pais.

Resumo

Interações concorrentes complexas geram frequentemente um grande número de trocas de mensagens entre processos. A sua implementação, tanto na ordem como nos tipos de mensagens, pode levar regularmente a erros por parte dos programadores. Para simplificar as interações e reduzir a probabilidade destes erros, os tipos de sessão foram criados. Estes tipos definem uma estrutura para comunicação entre protocolos de trocas de mensagens. Os canais de comunicação podem ser lineares ou partilhados. Nas interações concorrentes existe um canal partilhado por dois, ou mais, processos e é necessário garantir que se um envia uma mensagem, o outro estará preparado para receber essa mesma mensagem. Do mesmo modo, se um dos processos apresenta um leque de opções que se podem escolher, o processo no lado oposto do canal estará pronto para selecionar uma das opções apresentadas, de forma a garantir a complementaridade das operações.

Existem tipos finitos de mensagem, que repetem as mesmas operações continuamente, onde se pode realizar um número finito de operações de receção/envio ou oferta/selecção, como `!int.end`, que envia um número inteiro e termina a interação. Contudo, os tipos finitos não abrangem todas as interações possíveis, o que levou à criação dos tipos infinitos. Este trabalho apresenta-se como uma extensão dos tipos de sessão, já que é necessários estudar tipos com maior expressividade que os tipos finitos para que se possam incluir operações mais complexas, que necessitem a constante troca de mensagens.

Os tipos infinitos podem ser representados através de árvores infinitas, que vão sendo percorridas à medida que as operações vão sendo realizadas. Usando a recursividade para criar estes tipos infinitos, podemos ter `X=!int.X`, que se traduz no protocolo que envia números inteiros repetidamente, `X` volta a ser chamado sempre depois de se enviar o inteiro. Na derivação dos tipos recursivos existem os tipos 1-counter, onde é associado um contador a cada variável cujo valor define que operações vão sendo realizadas. Existem então duas equações para cada variável, uma que se segue quando o contador chega a zero e outra que é seguida quando o contador é maior que zero. Assim, as operações são realizadas variando entre uma destas duas equações, consoante o valor do contador.

Os tipos recursivos e os tipos 1-counter são o principal foco desta tese, já que são duas abordagens diferentes de tipos infinitos de sessão. Este trabalho é baseado na verificação de sistemas de equações para os tipos recursivos, os mais simples, e os tipos 1-counter, uma abordagem pouco estudada.

Foram criadas gramáticas para que se pudessem testar sistemas de equações para cada um destes tipos. Entre as regras das gramáticas é possível enviar (`!`) e receber (`?`) mensagens, bem como selecionar ($\oplus$) ou oferecer (`&`) opções. Além disto, existem identificadores de tipos e ainda tipos `end`, que terminam as operações. Todas estas regras foram implementadas na linguagem SePi, uma linguagem de programação concorrente baseada no cálculo-pi que define a semântica das mensagens. Foram também analisadas outras abordagens que usam tipos de sessão, como os tipos nested, que conseguem definir protocolos dentro de protocolos, independentemente do protocolo pai, e os context-free, onde é introduzida composição sequencial.

Todos os sistemas, quer apresentem um comportamento finito ou infinito, devem ser testados, e para isso foram contruídos algoritmos que possam verificar várias componentes dos sistemas. Os algoritmos construídos dividem-se em três tipos: formação, contratividade e equivalência de tipos. A verificação da formação analisa se os sistemas de equações apresentados seguem as regras das gramáticas construídas, se todas as variáveis do sistema são declaradas e se apresentam nomes únicos. Além disto, é também examinada a contratividade, que pretende não deixar que um sistema recursivo fique preso sem realizar ações. Assim, um sistema que apresente um comportamento cíclico sem realizar operações de enviar/receber mensagens ou de selecionar/escolher uma opção, será considerado não contrativo. Estes sistemas não são desejados quando se comparam tipos, logo esta verificação é realizada mesmo antes de se analisar o comportamento infinito de ambos. Finalmente, a última verificação baseia-se na comparação dos comportamentos de dois tipos que podem assumir um comportamento infinito e concluir se são, ou não, equivalentes.

A equivalência é um dos problemas fundamentais na teoria da computação, que analiza se dois sistemas se comportam de maneira equivalente. Aqui falamos em equivalência de tipos infinitos, construindo dois algoritmos de forma a decidir se o seu comportamento se assume como igual ou não.

A nossa implementação utilizou o ANTLR, um gerador de interpretadores para ler, processar e analisar ficheiros. Usado para construir linguagens a partir de gramáticas, realiza um parser que pode construir e visitar árvores de parser e assim realizar as verificações que eram pretendidas. Deste modo, as verificações de declaração de variáveis e de nomes únicos para cada uma delas são realizadas visitando o nó da árvore em que se encontram e fazendo essa avaliação.

O principal objetivo deste projeto é implementar estes algoritmos de verificação, bem como testá-los. Deste modo, podemos ter a certeza que tipos com este grau de expressividade estão corretamente definidos e que podem ser comparados.

Palavras-chave: tipos de sessão, tipos infinitos, algoritmos, formação de tipos, equivalência de tipos

Abstract

In concurrent interactions there are a large number of messages exchanged between two or more processes that often lead to coding errors. To simplify these interactions and reduce the coding errors, session types were created. Session types are an approach for structuring interaction protocols between multiple parties. When a channel is shared between two processes it is necessary to ensure that if one is sending a message, the other is prepared to receive it. In the same way, if a process offers some options, the complementary is prepared to select one of the options.

There are finite types of messages, that perform an operation and stop, where you can perform send/receive or offer/select a message and terminate the interaction. However, not all of the interactions are possible just with finite types. This work presents an extension of session types into infinity, since it is necessary to study different classes of types with greater expressive power than finite types.

Recursive and 1-counter types are the main focus of this thesis. We start by designing grammars so we can test equation systems based on those types. Defining rules in which the systems of these types can be written is the purpose of the grammars. These rules include the possibility of sending or receiving messages and selecting or offering a set of options. The grammars are implemented based on SePi, a concurrent programming language based on pi-calculus.

All the systems that present a finite and infinite behavior should be tested. Constructing algorithms for type formation and type equivalence of these systems as well as testing those algorithms is the main goal of this project, so that we can be sure that infinite types, specifically, with different degrees of expressivity are correctly defined and able to be compared.

Keywords: session types, recursive types, algorithms, type formation, type equivalence

Contents

List of Figures **xiii**

List of Tables **xv**

1 Introduction **1**

1.1 Motivation 1

1.2 Contributions 2

1.3 Confrontation with the initial work plan 2

1.4 Structure of the document 2

2 Session Types and the SePi language **5**

2.1 Finite session types 5

2.2 Infinite session types 6

2.3 SePi language 9

3 Grammars and algorithms **11**

3.1 ANTLR 11

3.2 Recursive types 11

3.2.1 Grammar 13

3.2.2 Type formation 13

3.2.3 Contractivity 15

3.2.4 Type equivalence 15

3.3 1-Counter types 18

3.3.1 Grammar 19

3.3.2 Type formation 20

3.3.3 Contractivity 21

3.3.4 Type equivalence 24

4 Tests **31**

5 Conclusion **35**

Bibliography **40**

List of Figures

2.1 Infinite tree type 7
2.2 Offering and selecting options SePi example 9

3.1 Rules for Recursive types 12
3.2 Pairs from 3.8 system 17
3.3 Pairs received from 3.9 system 18
3.4 Rules for 1-Counter types 19
3.5 Tree representation of the second approach of 3.19 29

4.1 Test classes summary 32
4.2 Algorithms line and branch coverage 33

List of Tables

3.1 *EqZeroContext* cases replacements(contractivity algorithm) 22
3.2 *EqOtherContext* cases replacements(contractivity algorithm) 23
3.3 *IdContext* cases replacements(equivalence algorithm) 25

4.1 Number of tests for recursive types algorithms 31
4.2 Number of tests for 1-counter types algorithms 31

Chapter 1

Introduction

1.1 Motivation

In concurrent interactions there are a large number of message exchanges between two or more processes that are often the target of coding errors. To make these interactions simpler and less likely to cause execution errors the session types were built. Session types for concurrent programming languages were first introduced by Honda et al. [8, 9]. Session types are a framework for communication protocols associated with the channels that exchange messages and that can be statically checked. The channels can be linear or shared. In linear channels we have just two threads communicating, however in shared channels we can have zero or more threads communicating between them, making it much more realistic for the computation world.

We can consider different protocols that extend finite types to infinite types by means of systems of equations. The simplest protocols define finite types: a channel can, for example, send an integer, receive a boolean and end `(!int.?bool.end)`. However, not all the interactions are that simple and can include recursion or more expressive types. So, different symbols for representing different operations were thought out. The `!` symbol is used for representing the send operation, the `?` symbol the reception, the `&` symbol offers choices from a finite number of possibilities and finally the $\oplus$ symbol is used for selecting one of the possibilities.

For an example of a possible infinite type specifying a choice, the type

```
T = &{start:!int.?bool.end, go:?string.T}
```

offers a choice between two different protocols, *start* and *go*. If we choose *start* then we are facing a finite type, but if we choose *go*, the channel receives a string and repeats T again, a potentially endless loop if we keep choosing *go*.

Systems of equations give rise to the class of recursive types (the above example) but also other extensions of which we focus on two: context-free types and 1-counter types. In context-free types we can have a sequential composition of types making it possible to represent a stack as a type. In 1-counter types there is a counter for every type that can be used for the equations that represent that same type.

Session types with different expressivity levels were first studied by Gay et al. [7]. It is necessary to study fragments of infinite session types such as recursive or 1-counter, in order to go beyond finite session types. To check these systems we want to implement type formation, type contractivity and type checking algorithms. Comparing types and testing if they describe equivalent computations are decidability problems for which we want an answer in finite time, that is, for which we desire a practical algorithm. These problems were thought of when constructing the algorithms. The main goal of this thesis is to develop, implement and test those algorithms for a programming language.

All the grammars created were based in the SePi language, a concurrent programming language based on the monadic pi-calculus [11], where communication among processes is governed by linearly refined session types [5].

1.2 Contributions

The main contributions of this work can be resumed as follows:

- Develop algorithms for type formation, type contractivity and type equivalence for recursive and 1-counter types.

- Implement and test those algorithms for a programming language based on Session types.

1.3 Confrontation with the initial work plan

This section explains the differences between the original plan thesis and what we actually did. The first plan was to develop and test the algorithms and implement these algorithms in a programming language. All the type formation, type contractivity and type equivalence algorithms were made as well as the tests for them. However, we did not implement our algorithms in a programming language. We estimate that this additional work would entail some extra months.

1.4 Structure of the document

The present chapter summarizes our work and presents the motivations and contributions. The next chapters are organized as follow:

- Chapter 2 - reviews the notion of session types and infinite session types followed by the SePi language.

- Chapter 3 – introduces the notions of recursive and 1-counter types and describes the grammars created for them. Describes all the algorithms made.
- Chapter 4 – explains how the tests were made and how the algorithms were tested.
- Chapter 5 – presents the conclusions and the plan for future work.

Chapter 2

Session Types and the SePi language

In this chapter we review some work related to finite session types (Section 2.1) and how infinite session types work (Section 2.2). We also present the base language used for this work, the SePi language (Section 2.3).

2.1 Finite session types

The notion of session types was first introduced to formalize the interactions between partners running in parallel that communicate by passing messages [14], as a variant of the pi-calculus [11]. Usually, only two partners are communicating (two threads), being a linear interaction. However, linearity does not have expressiveness to compute more complex interactions, so shared objects should be used for a multiple number of threads. Consider having a partner A, Alice, that wants to process a text sent from partner B, Bob. In order to do that, Bob should send the text to Alice using a communication channel defined in the form: `!string.end`. This way, Alice can read (`?string.end`) the text from Bob and print it, for example. The `!` operator represents an output and the `?` represents an input. The `end` type represents the end of the interaction in this channel. There exists two more forms of interaction, represented by `&`, for offering a choice, and $\oplus$, for selecting a choice. The existing complementarity between `!` / `?` and `&` / $\oplus$ guarantees good communications when multiple partners are involved and it is called duality. Automatically we can infer that the dual of input is output and the dual of output is input. In the same way, the dual of select is offer and the dual of offer is select, finally the dual of type `end` is itself. Duality is only defined for session types.

Another feature often found in object-oriented languages is subtyping, being that if T is a subtype of U, written $T \leq U$, then a channel of type T can safely be used wherever a channel of type U is expected. Gay and Hole [6] define a notion of subtyping for session types and especially recursive session types. Subtyping allows a channel that accepts a decimal number can accept an integer too, since the integer can be seen as a subtype of the decimal. For the output the relation works reversely. In internal and external choices something different happens. If a channel offers a choice with a set of labels then it can be used in a process that selects a choice from a subset of those same labels. A similar behaviour happens when the channel selects a choice. This concept increases the flexibility of a system since different protocols can be followed if a relation of subtyping is defined.

2.2 Infinite session types

We can have different data structures defined by the processes, but for a repetitive session behavior we have to use recursive types. They perform a behavior where the process is infinite and is represented as a regular infinite tree. As Pierce [12] explained we can add a recursive operator so that our operation continues repeatedly. This kind of types can be used, for example, for reading a stream of strings:

```
X = &{stop:end, continue:?string.X}
```

where in label `continue` we can consume (read) a string and go to `X` again, continuing to read other elements. Intuitively this loop inputs strings continuously if the choice is `continue` successively. When the stream has no more elements left to read, the label `stop` is chosen, which ends the interaction. The representation of this infinite tree is presented in figure 2.1. We start in the top node of the tree, that is `X`, and can choose between option stop or continue. If we choose `stop` then we arrive at the node `end` and we have no more nodes left to choose, the interaction stops. If we choose `continue` we arrive to the node `?string.X`. This node separates in two other nodes, `?string`, where a string is consumed, and `X` where we have again the option `stop` and `continue`, and so on.

To represent an infinite behaviour it is common to introduce a fixed point operator μ, where in $\mu X.Z$, X and Z are two different session types and the procedure is repeated. When defining a language we can achieve the recursion introducing the channel in the end of the operations. The μ operator expresses the recursion, meaning that

$$\mu\texttt{C.}\oplus\texttt{\{get:?int.C, sum:!int.!int.?int.C\}} \tag{2.1}$$

denotes the infinite type C of selecting a choice. In this example, every time `get` or `sum` is chosen, it returns to C adding a new part to the structure, not being stuck in the same operation since integers are send or received. As an alternative to the μ representation for the infinite behaviour, we can use equation systems, the approach used in this project.

For recursive types and its extensions, contractivity is one of the most important concepts to take into account. Contractivity ensures that the unfolding process stops, excluding undesirable cycles that don't perform any operation. For example, a system where

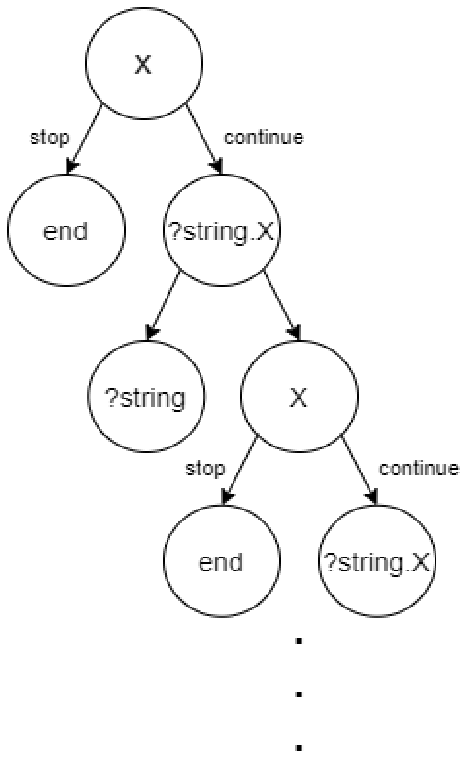


Figure 2.1: Infinite tree type

two types call each other without any other operations (`X=Y Y=X`) is considered a non-contractive system. Contractivity ensures that types can be interpreted as regular infinite trees [15].

When comparing two infinite types we have to look at their infinite unfolding, the equality is proved by coinduction.

In this work, based on systems of equations, two approaches are used for an infinite behaviour. The simplest approach, which we named recursive types and can be represented as the example in 2.1, and 1-counter types, that are a recent approach. In 1-counter types, a counter is introduced associated to each type variable. All type equations of a type depend on this counter. For example, consider the equation system representing a binary tree with nodes and leaves:

$$
\begin{aligned}
&\texttt{X <0> = end}\\
&\texttt{X <n+1> =}\oplus\texttt{\{leaf: !int.X<n>,}\\
&\qquad\qquad\qquad\texttt{node: !int.X<n+2>\}}
\end{aligned}
\tag{2.2}
$$

starting from the type `X<1>`. To send a tree, one can either select that this tree is a leaf, sending its value, or select that the tree has a node, sending its value, and then sending its two sub-trees. We could, following the code below, select `node` and send the number 5, select `leaf` and send the number 2, select `leaf` and send the number 3, and as we arrive at `X<0>` the interaction terminates.

```
select node -> !5.X<2>
select leaf -> !2.X<1>
select leaf -> !3.X<0>
end
```

Context-free types are another representation that goes beyond finite session types. In these types sequential composition is introduced. Thiemann and Vasconcelos [13] have proposed these session types, studying their metatheory. Sequential composition is represented by dropping the restriction of tail recursivity. They removed the continuation from send and receive types and adopted a general form of sequential composition `T;U` with unit `skip` instead of `end` making it clear that does not necessarily ends a session type. For example, the following equation represents a stack:

$$\begin{gathered}\texttt{Stack=}\oplus\{\texttt{push:?int;Stack;Stack,}\\ \texttt{pop:!int;skip}\}\end{gathered} \tag{2.3}$$

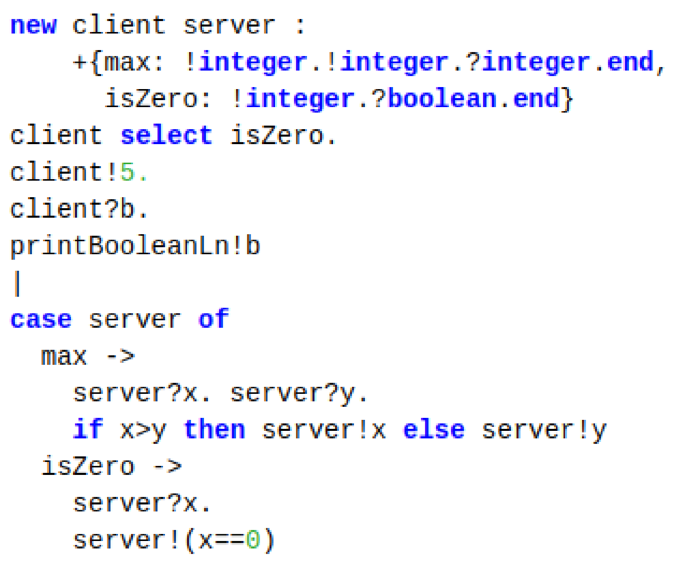
where we can push or pop integer numbers. After pushing a number we call the Stack protocol again. In the pop option an integer is sent followed by the termination of the reception. This way, what was `!int.end` now is just `!int` and what was `!int.!int.end` now is `!int;!int`. We could also create a communication channel to send this stack and wait for their response.

Introduced by Demangeon and Honda [4], nested protocols define a subprotocol independently of its parent protocol, which calls the subprotocol explicitly. This way, it is allowed to pass value arguments, roles or other protocols. Besides that, a protocol can call multiple copies of the same protocol one or more times, given different arguments. These subprotocols are treated as subsessions, where an agent can create a new private session inviting roles of the parent session or other agents from the network. The authors provide an example showing that one does not need to update the specifications of applications of a login protocol in order to enforce the security. Besides that, another advantage of nested types is the separation of the different branches by inviting participants only when necessary, which reduces the complexity and resources [4]. Also, Das et al. [3] develop the metatheory of nested types proving that type equivalence in nested types can be translated to the trace equivalence problem for deterministic first-order grammars.

Also, Gay et al. [7] studied the spectrum of general infinite types: finite-state, 1-counter, pushdown and 2-counter. The inference rules of this work are based in this study. They form a strict hierarchy for this types allowing them to establish decidability and undecidability for type formation and type equivalence. They prove that equivalence, duality and formation relations are all decidable and we make use of that fact.

2.3 SePi language

SePi [5] is a concurrent programming language based on pi-calculus. Pi-calculus is a theory of mobile systems introduced by Milner [10] for defining the semantics of message-based concurrent languages. This language creates a syntax of core processes and values, types and formulae that builds the core language. The processes include channel creation, parallel composition, conditionals, assume, assert and most significant for our work, input and output. In the same way, the pretypes used are very important, including `send`, `receive`, `select` and `branch`. Consider the image 2.2 below:



```
new client server :
    +{max: !integer.!integer.?integer.end,
      isZero: !integer.?boolean.end}
client select isZero.
client!5.
client?b.
printBooleanLn!b
|
case server of
  max ->
    server?x. server?y.
    if x>y then server!x else server!y
  isZero ->
    server?x.
    server!(x==0)
```

Figure 2.2: Offering and selecting options SePi example

Two new channels are created, called client and server. The process client follows the selecting choices from lines 2 and 3. The server is prepared to receive from the client two options, `max`, where the client sends two integers and receives the biggest, and `isZero`, where the client sends an integer and receives a boolean confirming if it is the number 0 or not. The client chooses the option `isZero` and sends the number `5`. After that, the client is expecting to receive a boolean and print it.

Since each process communicates via bidirectional channels, they synchronously use the important concept of duality for ensuring the complementarity between two processes that share a channel. The duality ensures that if one side of the channel sends a message, the other must receive it. The `dualof` operator is the abbreviation representing this in SePi. Besides that, they include a series of other abbreviations such as * , that represent a common class of shared types.

SePi provide a tool where type development may be tested. It is a language that combines session types with linear refinement types, types qualified by a logical constraint.

Chapter 3

Grammars and algorithms

This section explains the grammars used for the recursive and 1-counter types definitions.

For writing the type formation and type equivalence algorithms a grammar for recursive types and a grammar for 1-counter types, were developed using the ANTLR plugin for Eclipse (section 3.1). In section 3.2 type formation, contractivity and type equivalence algorithms are explained in detail for the recursive types. In section 3.3 the same algorithms are explained for 1-counter types.

3.1 ANTLR

ANTLR [1] (ANother Tool for Language Recognition) is a parser generator for reading, processing, executing, or translating structured text or binary files. Given a grammar, ANTLR creates a parser and this parser can build and visit a parse tree, also created. This tool was used to generate a parse tree visitor for each of our grammars. Giving a name to each rule, ANTLR creates a class that provides an empty implementation of visitors, which can be extended to create a visitor that only needs to handle a subset of the available methods. This was the process used in this project. When it was necessary to make some operation in a specific rule, the visitor was extended with a new operation using the *Override* annotation. There are two main visitors in this project, one for the recursive types grammar and one for the 1-counter types grammar. Then, for each class that uses the visitor, the parse tree created was passed and the tree was visited itself.

In order to use ANTLR in Eclipse, the plugin was installed from the marketplace and the grammars were written in a g4 file.

3.2 Recursive types

In figure 3.1 we can see the inference rules of a type. The formation of types is based on four rules. Rule T-end represents a type message `end`, used here as types that could otherwise be `int`, `string`, or `bool`. Rule T-Choice includes the external choice $\&\{l : T_l\}_{l \in L}$, that receives a label $l \in L$ and continues as T_l, and the internal choice $\oplus\{l : T_l\}_{l\in L}$, that selects a label $l \in L$ and continues as T_l. T-Msg rule includes the input type $?T.U$, that inputs a type T and continues to U, and the output type $!T.U$, that outputs a type T and continues to U. To avoid the repetition of the T-Choice and the T-Msg rules the # and $\star$ symbols are used. Finally, rule T-Id implies that T must be contractive and a valid type. For the contractivity rules, C-End represents a contractive type `end`. Rule C-Choice and rule C-Msg are always contractive since they always operate an action. Rule C-Id requires that T must be contractive. Finally, rules E-End, E-Choice and E-Msg are essentially syntactic equality. Rule E-EquiL requires that the two types must be contractive and right side equivalent to the left side, and rule E-EquiR requires that the two types must be contractive and left side equivalent to the right side.

Polarity and view

$\# ::= \,?\mid\, !$ $\qquad \star ::= \& \mid \oplus$

Type formation $\boxed{T \text{ type}}$

$$\text{end type} \qquad \text{T-End}$$

$$\frac{T_l \text{ type } (\forall l \in L)}{\star\{l\colon T_l\}_{l\in L} \text{ type}} \qquad \text{T-Choice}$$

$$\frac{T \text{ type} \quad U \text{ type}}{\#T.U \text{ type}} \qquad \text{T-Msg}$$

$$\frac{X \doteq T \quad T \text{ contr} \quad T \text{ type}}{X \text{ type}} \qquad \text{T-Id}$$

Type contractivity $\boxed{T \text{ contr}}$

$$\text{end contr} \qquad \text{C-End}$$

$$\star\{l\colon T_l\}_{l\in L} \text{ contr} \qquad \text{C-Choice}$$

$$\#T.U \text{ contr} \qquad \text{C-Msg}$$

$$\frac{X \doteq T \quad T \text{ contr}}{X \text{ contr}} \qquad \text{C-Id}$$

Type equivalence $\boxed{T \simeq T}$

$$\text{end} \simeq \text{end} \qquad \text{E-end}$$

$$\frac{T_l \simeq U_l \ (\forall l \in L)}{\star\{l\colon T_l\}_{l\in L} \simeq \star\{l\colon U_l\}_{l\in L}} \qquad \text{E-Choice}$$

$$\frac{T \simeq U \quad \mathrm{V} \simeq W}{\#T.V \simeq \#U.W} \qquad \text{E-Msg}$$

$$\frac{X \doteq U \quad U \text{ contr} \quad U \simeq T}{X \simeq T} \qquad \text{E-EquiL}$$

$$\frac{X \doteq U \quad U \text{ contr} \quad U \simeq T}{T \simeq X} \qquad \text{E-EquiR}$$

Figure 3.1: Rules for Recursive types

3.2.1 Grammar

We defined a grammar to represent the input of the algorithms following the previous rules. The *non-terminal symbols* (`I,E,T,F`) represent a language, each of them can be substituted by its definitions. The *terminal* symbols (end, &, $\oplus$, `!, ?,`) form the definition of the language. The symbol * represents the repetition of the preceding token zero or more times. The terminal symbol `x` can be defined as a regular expression defined by `[A-Z][a-zA-Z0-9_]*`, representing strings starting with one uppercase letter followed by any combination of letters, numbers or underscore character, zero or more times. Finally, the terminal symbol `f` also can be defined as a regular expression, `[a-z][a-zA-Z0-9_]*` that is very similar to `x` with the difference that the initial letter must be a lowercase letter.
Below we can find the rules used for recursive types. Rule *I* receives the two comparative types and goes to the equations that define the identifiers, if they exist. Rule *E* is the definition of identifiers. Rule *T* may indicate a select (&) or offer ($\oplus$) labelled choice defined by *F*, a receive (?) or send (!) value of type *T* continuing with *T*, an *end* that terminates a session, a type variable *x*, or a type *T* between parentheses. Rule *F* defines the composition of labels where *f* selects the type *F* from a finite number of options.

```
I::= ( T , T ) E*                        InitContext
E::= x = T                               EquationContext
T::= &{F(,F)*}                           ChoiceContext
     |⊕{F(,F)*}                          ChoiceContext
     | ?T.T                              MessageContext
     | !T.T                              MessageContext
     | end                               EndContext
     | x                                 IdContext
     | (T)                               ParContext
F::= f:T                                 FieldContext
x::= [A-Z][a-zA-Z0-9_]*
f::= [a-z][a-zA-Z0-9_]*
```

Each rule has a name annotated to the right. The first rule is called *InitContext*, the second rule is *EquationContext*, and so on.

3.2.2 Type formation

For checking the formation it is verified if the types are all defined, are represented with an equation, and the names of the identifiers are unique. For example, we need to check that each type identifier that appears has its definition. If the equation `X = P` appears, where *P* is a type identifier, then there must be an equation for *P*, and so on. Besides that, we also verify if all the labels in the choices are unique in each type.

The first thing checked is the unique name of the type identifiers that appear in the *EquationContext* rule on the left side, followed by the two comparative types, that appear in the *InitContext*, and then if there are any equations they are next. All the definitions of the equations are checked, the right side of the *EquationContext* rule, and if it is a choice we check if the name of the labels is unique, on the *FieldContext* rule. In this order, when one of the errors is found the algorithm stops.

In the example (3.1), the definition of type *X*, which is *M*, is not defined, an exception is thrown (*BadFormation* exception).

```
(X, Y)
X = M                                  (3.1)
Y = !end.!end.end
```

In example (3.2), although all the type identifiers are defined and none of them has a repeated name, the labels have the same name (`one`), so an exception of bad formation is thrown too.

```
(X, Y)
X = M
M = ?end.!end.end                      (3.2)
Y = &{one:!(!end.end), one:end }
```

In the example (3.3), there are two identifiers called *Y* which is incorrect too. If all these three errors were in the same system, then the two `Y` equations would raise the exception, that is the exception of the two `Y` equations have higher priority than the identifier that is not defined (example (3.1)), followed by the same name in labels (example (3.2)).

```
(X, Y)
X = end
Y = !end.!end.end                      (3.3)
Y = &{one:end, two:end }
```

Finally, in (3.4) no exception is thrown, since the two comparative types are well defined, even the *Y* used in the receiving message has a definition.

```
(!end.end, ?Y.!end.end)                (3.4)
Y = end
```

3.2.3 Contractivity

In the algorithm for contractivity we search whether a cycle is found. A cycle is formed by type identifiers that do not realize operations. For this algorithm, only the types that are *IdContext* and *ParContext* are checked, since they are the only ones that can create a cycle without performing any operation. In the *ParContext* the parenthesis are taken off until a different type is found, if they are *IdContext* then the algorithm continues. For example, in the below systems we follow type identifiers that are defined as other type identifiers, saving them. We check if some of them go back to the ones that were already checked, meaning that they were saved.

In (3.5), `X` is defined by other type, `M`, so `X` is saved and the equation of `M` is followed. As `M` is defined by `Y` (`M` is saved and `Y` is followed) and `Y` by `X`, a type already checked (saved), we are facing an infinite loop and an exception of contractivity is thrown. If we are facing an example as (3.6), as we follow the type identifiers we arrive at `Y` and its definition does not allow the existence of an infinite loop since it as a send operation of `end`, so it is considered an non-contractive system and no exception is thrown.

$$\begin{array}{l}\texttt{(X, Y)}\\ \texttt{X = M}\\ \texttt{M = Y}\\ \texttt{Y = X}\end{array} \tag{3.5}$$

$$\begin{array}{l}\texttt{(X, Y)}\\ \texttt{X = M}\\ \texttt{M = Y}\\ \texttt{Y = !end.end}\end{array} \tag{3.6}$$

In the example (3.7) the `P` identifier is defined by a *ParContext*, however inside of it exist a *IdContext* that creates an non-contractive system. So a side method is created to remove the parenthesis until the *IdContext* is found and we can check the variable `M`.

$$\begin{array}{l}\texttt{(X, Y)}\\ \texttt{X = end}\\ \texttt{Y = W}\\ \texttt{W = P}\\ \texttt{P =((M))}\\ \texttt{M = W}\end{array} \tag{3.7}$$

3.2.4 Type equivalence

The type equivalence for recursive types is defined as the equality of the infinite regular trees created when unfolding the types, if necessary. The comparison of infinite types is proved by coinduction. For example, `X=!int.X` and `Y=!int.!int.Y` can be considered equivalent since they realize the same operations in a infinite perspective (output a integer continuously) but are defined in different ways.

So, two types are received by a recursive Java method. The context of the two types is compared with the existing contexts formed by naming the rules in the grammar (*MessageContext*, *ChoiceContext*, *ParContext*, *IdContext*, *EndContext*).

If a type identifier (*IdContext*) is given, then it is substituted by its definition, and the method is called again. If the two types are messages (*MessageContext*), then the two symbols (send/receive) are compared; if they are equal, then it is checked the first part of the operation (in type *!T.U* the first part of the operation is considered the *T* type and the second part the *U* type). After this, the second part is compared if no exception was thrown. On the other hand, if the context is a choice (*ChoiceContext*), the choose/select symbols are also compared and if they match, then the comparison of the fields starts. The number and the field names must be equal. If they are, they are treated as a normal type, so a call to the method is realized for each of the fields. If no exception appears, the following field is checked, and so on. If the context of a type is the parenthesis (*ParContext*) then the method is called again with the inside of the parentheses. Finally, if the two types are end (*EndContext*), which means we arrived at the end of the definitions so we can assume equality in that branch, the algorithm stops the cycle that is being checked. There is one more way for the algorithm to stop the cycles. All the pairs of types compared are saved, and if they are both called again, we know this branch can be considered equal.

Following the system 3.8 in figure 3.2 are all the pairs that the algorithm receives to compare:

$$
\begin{aligned}
&\texttt{(X, Y)}\\
&\texttt{X = !(!end.end).end}\\
&\texttt{Y = M}\\
&\texttt{M = !T.end}\\
&\texttt{T = !end.end}
\end{aligned}
\tag{3.8}
$$

On the right side of figure 3.2, in the first three lines there are just substitutions of the identifiers when they are received as a type. When line 4 is received, the send (`!`) symbol was compared (in 3.2 represented with `(!,!)` between pairs) so the first argument of each operation can be checked (i.e the message being send), and line 5 is received. In line 6 it is necessary to replace `T` by its definition. In line 7 parentheses were taken off and the send symbol is compared again and in line 8 we compare the first part of the message, two ends. As these two are the same type, in line 9 the second part of the message checks that two ends are equal again. Finally, in line 10, the second part of the message, in line 4, is compared, two ends again. The algorithm stops and they can be considered equivalent.

$$\begin{tikzcd}
 & & \text{(X, Y)} \arrow[d] & \\
 & & \text{(!(!end.end).end, Y)} \arrow[d] & \\
 & & \text{(!(!end.end).end, M)} \arrow[d] & \\
 & & \text{(!(!end.end).end, !T.end)} \arrow[dl, "\text{(!,!)}"] \arrow[dr] & \\
 & \text{((!end.end), T)} \arrow[d] & & \text{(end, end)} \\
 & \text{((!end.end), !end.end)} \arrow[d] & & \\
 & \text{(!end.end, !end.end)} \arrow[dl, "\text{(!,!)}"] \arrow[dr] & & \\
\text{(end, end)} & & \text{(end, end)} &
\end{tikzcd}$$

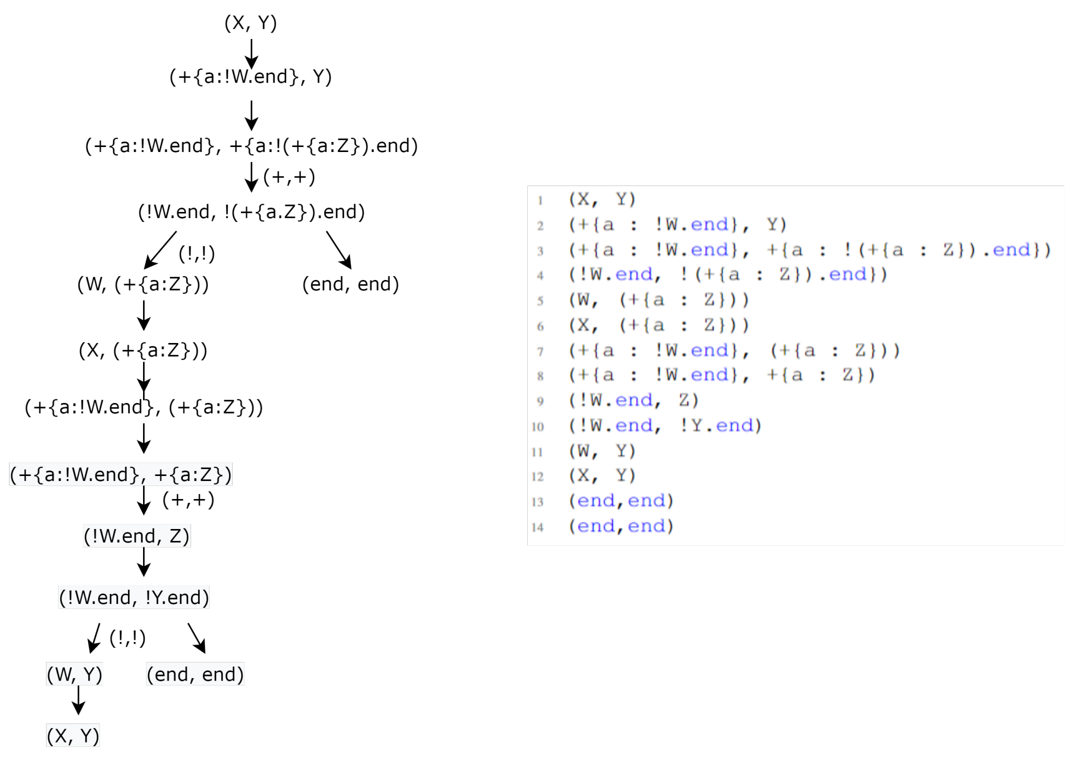
```
(X,Y)
(!(!end.end).end,Y)
(!(!end.end).end,M)
(!(!end.end).end,!T.end)
((!end.end),T)
((!end.end),!end.end)
(!end.end,!end.end)
(end,end)
(end,end)
(end,end)
```

Figure 3.2: Pairs from 3.8 system

In the example 3.9 something different happens. The algorithm stops because we find a pair that was already checked.

$$\begin{aligned}
&\texttt{(X, Y)}\\
&\texttt{X = }\oplus\{\texttt{a : !W.end}\}\\
&\texttt{Y = }\oplus\{\texttt{a : !(}\oplus\{\texttt{a : Z}\}\texttt{).end}\}\\
&\texttt{W = X}\\
&\texttt{Z = !Y.end}
\end{aligned} \qquad (3.9)$$

Following the figure 3.3 and the entries beside, we can see how the types of this example are treated. After the first substitutions, we find, in line 3, two select choices (represented in 3.3 with `(+,+)`) with the same label name (`a`), so in line 4, as we can see a send symbol in both sides, in line 5 the comparison of the messages being sent starts. Afterwards, more substitutions have to be done as well as taking the parentheses off. A new pair of choices are presented in line 8, and in line 9 a new type identifier has to be substituted (`Z`). After that, in line 12 we find a pair already saved, appearing in line 1, as a verified pair, the algorithm breaks and the other branch from line 10 is compared (line 13). As they are equal types (`end,end`) the second branch from line 3 is compared and they are also equal, so both types are considered equivalent, since no more branches are left to compare.



```
(X, Y)
(+{a : !W.end}, Y)
(+{a : !W.end}, +{a : !(+{a : Z}).end})
(!W.end, !(+{a : Z}).end})
(W, (+{a : Z}))
(X, (+{a : Z}))
(+{a : !W.end}, (+{a : Z}))
(+{a : !W.end}, +{a : Z})
(!W.end, Z)
(!W.end, !Y.end)
(W, Y)
(X, Y)
(end,end)
(end,end)
```

Figure 3.3: Pairs received from 3.9 system

Now, we can easily see that 3.10 will not be considered equivalent. After `X` is substituted, the pair received is `(!end.end, ?end.end)`. Although they are both *MessageContext* types, one of them has a send symbol (`!`), and the other has a receive (`?`) symbol, so the algorithm stops immediately with an exception.

```
(X, ?end.end)
X=!end.end
```
(3.10)

3.3 1-Counter types

In figure 3.4 we can see the new inference rules of a type for 1-counter types, extending the inference rules for recursive types. The type formation is based on five different rules. The T-end, T-Choice and T-Msg are the same as in recursive types. The T-z and T-s introduce natural numbers where all type identifiers X must be defined with the two of them. The same happens in the rules for contractivity, C-End, C-Choice and C-Msg are the same as in recursive types and C-z and C-s introduce natural numbers. The two type equivalence rules are adapted for 1-counter types. In order not to repeat similar rules we just show the rules for $X\langle z\rangle$ and $X\langle sn\rangle$ on the left-hand side.

Polarity and view

$\# ::= ? \mid !$ $\quad \star ::= \& \mid \oplus$

Natural numbers

$n ::= \mathrm{z} \mid sn$

Type formation $\boxed{T \text{ type}}$

end type T-End

$$\frac{T_l \text{ type } (\forall l \in L)}{\star\{l\colon T_l\}_{l \in L} \text{ type}}$$ T-Choice

$$\frac{T \text{ type} \quad U \text{ type}}{\#T.U \text{ type}}$$ T-Msg

$$\frac{X\text{<z>} \doteq T \quad T \text{ contr} \quad T \text{ type}}{X\text{<z>} \text{ type}}$$ T-z

$$\frac{X\text{<s N>} \doteq T \quad T[n/\mathrm{N}] \text{ contr} \quad T \text{ type}[n/\mathrm{N}]}{X\text{<}sn\text{>} \text{ type}}$$ T-s

Type contractivity $\boxed{T \text{ contr}}$

end contr C-End

$\star\{l\colon T_l\}_{l \in L}$ contr C-Choice

$\#T.U$ contr C-Msg

$$\frac{X\text{<z>} \doteq T \quad T \text{ contr}}{X\text{<z>} \text{ contr}}$$ C-z

$$\frac{X\text{<s N>} \doteq T \quad T[n/\mathrm{N}] \text{ contr}}{X\text{<}sn\text{>} \text{ contr}}$$ C-s

Type equivalence $\boxed{T \simeq T}$

end $\simeq$ end E-end

$$\frac{T_l \simeq U_l \; (\forall l \in L)}{\star\{l\colon T_l\}_{l \in L} \simeq \star\{l\colon U_l\}_{l \in L}}$$ E-Choice

$$\frac{T \simeq U \quad V \simeq W}{\#T.V \simeq \#U.W}$$ E-Msg

$$\frac{X\text{<z>} \doteq U \quad U \text{ contr} \quad U \simeq T}{X\text{<z>} \simeq T}$$ E-EquiZL

$$\frac{X\text{<s N>} \doteq U \quad U[n/\mathrm{N}] \text{ contr} \quad U[n/\mathrm{N}] \simeq T}{X\text{<}sn\text{>} \simeq T}$$ E-EquiSL

Figure 3.4: Rules for 1-Counter types

3.3.1 Grammar

The grammar for 1-counter types is very similar to the grammar for recursive types, with the difference that a natural number n parameterizes type identifiers (and equations). We introduce the symbol $+$ that represents the repetition of the rule at least once. Besides that, in the left side of equations we can only have `x<0>` and `x<n+1>` expressions, but in the right side we can have any number or expression, for example: `X<0>`, `X<1>`, `X<2>`, `X<n>`, `X<n+1>`, `X<n+2>`. This is defined by rule M, where j is a natural number and n is a specific letter. Also, the comparative types can only be x with a natural number, as opposed to recursive types where just the identifier is used.

```
I::= (x<j> , x<j>) E +                    InitContext
E::= x<0> = T                           EqZeroContext
     | x<n+1> = T                      EqOtherContext
T::= &{F(,F)*}                          ChoiceContext
     |⊕{F(,F)*}                         ChoiceContext
     | ?T.T                            MessageContext
     | !T.T                            MessageContext
     | end                                 EndContext
     | x<M>                                 IdContext
     | (T)                                 ParContext
F::= f:T                                 FieldContext
M::= j                                OpNumberContext
     | n                                   OpNContext
     | n + j                           OpNplusContext
x::= [A-Z][a-zA-Z0-9_]*
f::= [a-z][a-zA-Z0-9_]*
j::= 0|[1-9][0-9]*
```

Each rule has a name annotated to the right. The first rule is called *InitContext*, the second rule is *EqZeroContext*, and so on.

3.3.2 Type formation

The verification of type formation can be divided in three steps: checking the left sides and the right sides of the equations and if the comparative types exists. The left side checks if all the type identifiers have a unique name and exactly one equation for `x<0>` and `x<n+1>`. The next thing to check is if the types under comparison also occur in the left sides of the equations. Then the right side of equations is visited. If a *ChoiceContext* is encountered the field names are verified, if an *IdContext* is encountered more verifications are needed. In addition to verifying if the two mandatory equations exist for that type identifier, the `x<0>` equations can only be defined by other type identifiers if they are defined with natural numbers. So, if a type identifier depending on *n* is found we verify whether the parent is a `x<0>` equation or not.

Three examples of bad definitions are presented in (3.11), (3.12) and (3.13). In (3.11), the equation for `X<0>` is defined twice, and in (3.12) the equation for `Y<0>` is defined with a *OpNplusContext* instead of a *OpNumberContext*. Finally, in (3.13) the `Y<n+1>` equation is missing.

$$\begin{aligned}&\texttt{(X<1>, Y<0>)}\\ &\texttt{X<0> = (\&\{one:end\})}\\ &\texttt{X<0> = end}\\ &\texttt{X<n+1> = Y<n+2>}\\ &\texttt{Y<0> = X<4>}\\ &\texttt{Y<n+1> = ?end.end}\end{aligned} \tag{3.11}$$

$$\begin{aligned}&\texttt{(X<2>, Y<3>)}\\ &\texttt{X<0> = !end.end}\\ &\texttt{X<n+1> = end}\\ &\texttt{Y<0> = X<n+1>}\\ &\texttt{Y<n+1> = !X<n+1>.end}\end{aligned} \tag{3.12}$$

$$\begin{aligned}&\texttt{(X<2>, Y<3>)}\\ &\texttt{X<0> = Y<0>}\\ &\texttt{X<n+1> = Y<n+3>}\\ &\texttt{Y<0> = X<n+1>}\end{aligned} \tag{3.13}$$

3.3.3 Contractivity

The contractivity in 1-counter types is much more complex than in recursive types since we have to choose between the two mandatory equations.

One of the simplest situations is when the two main equations call each other, then we know it is a cycle, for example `{X<0>=Y<0> Y<0>=X<0>}`. However, we might have some more difficult situations like (3.14).

$$\begin{aligned}&\texttt{(X<2>, Y<3>)}\\ &\texttt{X<0> = X<1>}\\ &\texttt{X<n+1> = Y<2>}\\ &\texttt{Y<0> = end}\\ &\texttt{Y<n+1> = X<0>}\end{aligned} \tag{3.14}$$

Thinking about how the flow of the equations work, the `X<0>` is defined by `X<1>`, so the `X<n+1>` equation must be followed with `n` equal to 0, since *n+1=1*. In `X<n+1>` we go to `Y<2>`, so we follow the definition for `Y<n+1>`, which is `X<0>`. At this point we are facing an already checked type, `X<0>`, so we know we entered a cycle with no actual operations, like sending/receiving a message, and can consider this system of equations a non-contractive one.

For constructing this algorithm, an approach with some helpers was built. A map is created with two different objects, the key and the corresponding value. Only the types that are *IdContext* are saved in this map. The key is a pair of type identifier and type of equation, it could be a *(x,z)* if it is an $x\langle 0\rangle$ equation and *(x,s)* if it is an $x\langle n{+}1\rangle$ equation. The value represents the right side of the equations as an object, with three parameters *(id, b, j)*. The first parameter (*id*) is the name of the identifier, the second parameter is a boolean representing whether *n* occurs in the identifier, and the third one is a number *j* (for example, for a type $x\langle n\rangle$, the *j* is 0). When a new *IdContext* type appears, the map is updated. The (3.15) represents how the equations `X<0>=Y<5>` and `X<n+1>=Y<n+2>` are saved.

$$\begin{aligned}(X, z) &: (Y, \text{false}, 5)\\(X, s) &: (Y, \text{true}, 2)\end{aligned} \tag{3.15}$$

We can have different situations when updating the map. First, let us think about the $x\langle 0\rangle$ cases. Its definition can have other $y\langle 0\rangle$, in this case we just have to look for the $y\langle 0\rangle$ equation and replace the value of *(x,z)* in the map for the value of *(y,z)*. However, if its definition is $y\langle j\rangle$, with $j{>}0$, we have to find the $y\langle n{+}1\rangle$ equation, and here two different cases can happen. If the definition of $y\langle n{+}1\rangle$ is just a identifier with number *j*, then the value of *(x,z)* is going to be the value of *(y,s)*. On the other hand, if the definition of $y\langle n{+}1\rangle$ has a variable *n*, then the new value of *(x,z)* is going to be the identifier with the result of adding the *j* and *j'* numbers from *y* and the definition of *w*, minus 1, since the *n+1* is adding 1 that has to be removed. Table 3.1 summarizes these three cases.

	Search for	Case	Replace
$x\langle 0\rangle{=}y\langle 0\rangle$	$y\langle 0\rangle$	$y\langle 0\rangle{=}w\langle j'\rangle$	$x\langle 0\rangle{=}w\langle j'\rangle$
$x\langle 0\rangle{=}y\langle j\rangle$ $j{>}0$	$y\langle n{+}1\rangle$	$y\langle n{+}1\rangle{=}w\langle j'\rangle$	$x\langle 0\rangle{=}w\langle j'\rangle$
		$y\langle n{+}1\rangle{=}w\langle n{+}j'\rangle$	$x\langle 0\rangle{=}w\langle j{+}j'{-}1\rangle$

Table 3.1: *EqZeroContext* cases replacements(contractivity algorithm)

On the other hand we have the $x\langle n{+}1\rangle$ cases, that can be summarized in four different situations. First of all, its definition can be the *z* equation of an identifier *y*, so we just have to replace the value of *(x,s)* with the value of *(y,z)*. If the definition of $x\langle n{+}1\rangle$ is not a $y\langle 0\rangle$ case then we have to search for the $y\langle n{+}1\rangle$ equation. If the definition of $y\langle n{+}1\rangle$ is a *w* identifier with a number *j*, we just have to replace the *(x,s)* with the value of *(y,s)*. However, if the definition of $y\langle n{+}1\rangle$ is a *w* identifier with an *n*, then the new value of *(x,s)* must be the sum of the numbers *j* minus 1. The third situation is the simplest, if $x\langle n{+}1\rangle$ has an identifier *y* with an *n* and no *j*, nothing changes in the map, since this equation cannot be simplified further. Finally, the last situation occurs when the value is a *y* with an *n* and a $j{>}0$. The value of *(y,s)* can be a *w* identifier with an *j* and the value of *(x,s)* is replaced by this value of *(y,s)*, or it can be an *n* with *j'* and the new value of *(x,s)* is *n* with the sum of *j* and *j'* minus 1. Table 3.2 summarizes all these cases.

	Search for	Case	Replace
$x\langle n+1\rangle = y\langle 0\rangle$	$y\langle 0\rangle$	$y\langle 0\rangle = w\langle j'\rangle$	$x\langle n+1\rangle = w\langle j'\rangle$
$x\langle n+1\rangle = y\langle j\rangle$ $j>0$	$y\langle n+1\rangle$	$y\langle n+1\rangle = w\langle j'\rangle$ $y\langle n+1\rangle = w\langle n+j'\rangle$	$x\langle n+1\rangle = w\langle j'\rangle$ $x\langle n+1\rangle = w\langle j+j'-1\rangle$
$x\langle n+1\rangle = y\langle n\rangle$			
$x\langle n+1\rangle = y\langle n+j\rangle$ $j>0$	$y\langle n+1\rangle$	$y\langle n+1\rangle = w\langle j'\rangle$ $y\langle n+1\rangle = w\langle n+j'\rangle$	$x\langle n+1\rangle = w\langle j'\rangle$ $x\langle n+1\rangle = w\langle n+j+j'-1\rangle$

Table 3.2: *EqOtherContext* cases replacements(contractivity algorithm)

In addition to all these cases in which there may be changes in the map, we still have to think about one last case: when the left part of the equation is the definition (value) of an identifier. Consider the example 3.14. When the first line is read, the map is {*(X,z): (X, false, 1)*}. However, when the second line is visited, although we do not make changes on the right side because there are no *(Y,s)* in the map yet, we must change the first entry since there is an equation that uses *(X,s)* as value. Now the map is updated to {*(X,z): (Y, false, 2), (X,s): (Y, false, 2)*}. When the last equation is visited, we have to find *(X,z)* in the map and replace with its value, following the line 1 in the table 3.2, and then find all the equations that use *(Y,s)* as values. The final map is : {*(X,z): (Y, false, 2), (X,s): (Y, false, 2), (Y,s): (Y, false, 2*}*)*. Based on this map we can conclude that the system is not contractive. The equation represented by the entry *(Y,s): (Y, false, 2)*, uses itself, so creates a infinite loop.

Following a more complex example, (3.16), the first pair inserted on the map is {*(X,z):(Y, false, 2)*}. Next, the pair *((X,s): (Y, false, 0))* is also inserted without any replacements. When `Y<n+1>` is visited, the entry *((Y,s): (Z, true, 0))* is added and the substitutions are made. The pair *(X,z)*, that use *Y* variable with *j>0* is replaced with the new value *(Z, false, 1)*. The map now looks like: {*(X,z):(Z, false, 1), (X,s): (Y, false, 0), (Y,s): (Z, true, 0)*}. Finally, the `Z<n+1>` equation is visited and the last replacements are made, using line 4 of table 3.2. The final map 3.17 is shown below. The system is contractive, since no loop is created.

```
(X<2>, Y<3>)
X<0> = Y<2>
X<n+1> = Y<0>
Y<0> = end                (3.16)
Y<n+1> = Z<n>
Z<0> = end
Z<n+1> = X<n+2>
```

$$\begin{aligned}&(X,z):(X,false,2),\\&(X,s):(Y,false,0),\\&(Y,s):(Z,true,0),\\&(Z,s):(Y,false,1)\end{aligned} \tag{3.17}$$

After the final map is obtained there are three cases that can determine whether a system of equations is non-contractive.

(x,z):(x,false,0)
(x,s):(x,false,j>0)
(x,s):(x,true,j>0)

The first case is when a *z* equation is equal to itself, like `Y<0>=Y<0>`. The second case is when a *s* equation is defined by the same variable and a *j>0* number, like `Y<n+1>=Y<4>`. The last case is when an *s* equation is defined by the same variable and an *n+j* with *j>0* like `Y<n+1>=Y<n+2>`. If any of these cases is found, then the system is non-contractive and an exception is thrown.

3.3.4 Type equivalence

The equivalence in 1-counter types is proved by coinduction, as it happens in the recursive types, unfolding types if necessary. However, 1-counter types work with a counter that represents the type. So, this algorithm always saves the counter of both types, decreasing or increasing, according to the way the type is going.

A method that receives two types and their respective counters was developed to compare them. Depending on the context of the types, different things can happen. The simplest case is when the two types are *EndContext*, it just stops. If one of the types is *ParContext*, then the algorithm is called again with the type inside the parentheses, and all the other parameters are the same. Very similarly to what happens in recursive types when the two types are *MessageContext*, the polarity symbol(*!/?*) is compared and if it is equal, then the first part of the messages is sent to the algorithm again as independent types, with the respective counters. If no exception is thrown, the second part of the message is also compared. Also, the *ChoiceContext* is very similar to the recursive types. When the two types are *ChoiceContext* and the symbols(&/⊕) are equal, then they are treated in a different method. The names of the fields are compared, and if they are equal, then the algorithm is called with both fields and the counters that were when they were compared. The last case is when one of them is *IdContext*, they are sent to a different method. Depending on the context of the interior of the *IdContext* (*OpNumberContext*, *OpNContext*, *OpNplusContext*) the algorithm is called with different parameters.

	Search for	Call
$x\langle j \rangle$, c j=0	$x\langle 0 \rangle = T$	T, 0
$x\langle j \rangle$, c j>0	$x\langle n+1 \rangle = T$	T, j-1
$x\langle n \rangle$, c c=0	$x\langle 0 \rangle = T$	T, 0
$x\langle n \rangle$, c c>0	$x\langle n+1 \rangle = T$	T, c-1
$x\langle n+j \rangle$, c j>0	$x\langle n+1 \rangle = T$	T, c+j-1

Table 3.3: *IdContext* cases replacements(equivalence algorithm)

Following the table 3.3, the first column shows all the different types of *IdContext* cases with the different counters (c). So, if the context is *OpNumberContext* and the j number is 0, then the algorithm is called again with the definition of $x\langle 0 \rangle$ and a counter of 0. If the j number is bigger than 0, we search for the definition of $x\langle n+1 \rangle$ and call the algorithm with it and a counter of j-1. In case the context is *OpNContext* and the counter is 0, then the algorithm is called with the definition of $x\langle 0 \rangle$ and a 0 counter. On the other hand, if the counter is bigger than 0, then is the definition of $x\langle n+1 \rangle$ that is used and a counter of c-1. The last case is when the context is *OpNplusContext* and here we always search for the type of the equation $x\langle n+1 \rangle$ and the algorithm is called with and a counter of c+j-1.

In example 3.18, on the left side we have the equation system, and on the right side we find all the entries that the algorithm takes. The first entries that the algorithm takes is always the first substitution from the comparative types with the counters from them. In this case, the second entries are just the substitution of the identifier `W` with its definition and the respective counter. In line 2 the two types are messages, so in line 3 the first part starts their comparison, and as they are equal types (`end`, `end`) (the pair (`end`, `end`) is the only case that the types are considered equal independently of the counters), in line 5 the second part is compared as well. As they end to be equal types, again (`end`, `end`), we can affirm that `X<1>` and `Y<1>` are equivalent.

```
(X<1>, Y<1>)
X<0>=end
X<n+1>=W<1>
Y<0>=end
Y<n+1>=!end.end
W<0>=end                          (3.18)
W<n+1>=!Z<1>.P<1>
Z<0>=end
Z<n+1>=end
P<0>=end
P<n+1>=end
```

```
(W<1>, 1, !end.end, 1)
(!Z<1>.P<1>, 0, !end.end, 1)
(Z<1>, 0, end, 1)
(end, 0, end, 1)
(P<1>, 0, end, 1)
(end, 0, end, 1)
```

In this example 3.19 something different happens. We know this example may not be considered equal just by looking at the comparative types. To prove this, we can look at the entries below, the counters are decreasing as the algorithm progress. In the end, two different type contexts are given, and we can assume that they are not equivalent.

```
(X<4>, X<3>)
X<0>=end                          (3.19)
X<n+1>=!end.X<n>
```

```
!end.X<n>, 4, !end.X<n>, 3
end, 4, end, 3
X<n>, 4, X<n>, 3
!end.X<n>, 3, X<n>, 3
!end.X<n>, 3, !end.X<n>, 2
end, 3, end, 2
X<n>, 3, X<n>, 2
!end.X<n>, 2, X<n>, 2
!end.X<n>, 2, !end.X<n>, 1
end, 2, end, 1
X<n>, 2, X<n>, 1
!end.X<n>, 1, X<n>, 1
!end.X<n>, 1, end, 0
```

There are two more ways to stop the algorithm's cycle that have nothing to do with the different contexts. Since every pair of types and their counters are saved in a map, the algorithm stops if a new entry in the algorithm is equal to a saved one. Another way is if the two types and their counters are the same, then we know that the operations will always be equal, so they can be considered equivalent and the algorithm can stop too. These two stops make the algorithm more efficient since we can take a conclusion from these branches immediately.

The algorithm stops in one more situation without throwing a *NotEquivalent* exception. As we can see in 3.20 some systems can grow infinitely and never be considered not equivalent. A recursion limit was implemented so that the algorithm can break at some point.

```
(X<2>, Y<3>)
X<0>=end
X<n+1>=!end.Y<n+2>
Y<0>=end
Y<n+1>=!end.Y<n+2>
```

(3.20)

```
!end.Y<n+2>, 2, !end.Y<n+2>, 3
end, 2, end, 3
Y<n+2>, 2, Y<n+2>, 3
!end.Y<n+2>, 3, Y<n+2>, 3
!end.Y<n+2>, 3, !end.Y<n+2>, 4
end, 3, end, 4
Y<n+2>, 3, Y<n+2>, 4
...
```

A recursion limit is necessary since unfolding types can be infinite, so if this number is reached, the algorithm stops, and the types are considered equivalent in that branch. If no exception is thrown below the limit, then there is a good probability that they may be equivalent. Two different implementations of this recursion limit were made. Given the limit number, if any of the counters reaches it, the algorithm stops in that branch, and we record that at least one branch reached the limit. The other approach keeps the depth of the unfolding types and if it reaches the limit number the algorithm breaks and once more we record that the limit was reached in one branch. The algorithm goes back to the previous types before entering that branch and continues the comparison.

```
(X<7>, Y<9>)
X<0>=end
X<n+1>=+{a:!end.Y<n+2>,b:!end.end}
Y<0>=end
Y<n+1>=+{a:!end.Y<n+2>,b:W<1>}
W<0>=end
W<n+1>=!Z<1>.!Z<1>.end
Z<0>=end
Z<n+1>=end
```

(3.21)

We illustrate the two approaches for the recursion limit with example 3.21. Looking at the system, we know the algorithm must throw a *NotEquivalent* exception since the field `b` is not equivalent in the two types, regardless of the counters. Limiting the recursion limit to 10, in the first version of the algorithm the entries will be as follows:

```
+{a:!end.Y<n+2>,b:!end.end}, 7, +{a:!end.Y<n+2>,b:W<1>}, 9
!end.Y<n+2>, 7, !end.Y<n+2>, 9
end, 7, end, 9
```

```
4  Y<n+2>, 7, Y<n+2>, 9
5  +{a:!end.Y<n+2>,b:W<1>}, 8, Y<n+2>, 9
6  +{a:!end.Y<n+2>,b:W<1>}, 8, +{a:!end.Y<n+2>,b:W<1>}, 10
7  !end.end, 7, W<1>, 9
8  !end.end, 7, !Z<1>.!Z<1>.end, 0
9  end, 7, Z<1>, 0
10 end, 7, end, 0
11 end, 7, !Z<1>.end, 0
```

The algorithm is performing normally until line 6. At this point, one of the counters reaches the limit, so the branch that was being checked (field a) stops and the other branch (field b) initiates the comparison. As none of the counters reaches 10 again and two different types are given to be compared (end, !Z<1>.end), the algorithm stops with a *NotEquivalent* exception.

In the second approach, the entries will be as follows (the last parameter is the depth):

```
1  +{a:!end.Y<n+2>,b:!end.end}, 7, +{a:!end.Y<n+2>,b:W<1>}, 9, depth=0
2  !end.Y<n+2>, 7, !end.Y<n+2>, 9, depth=1
3  end, 7, end, 9, depth=2
4  Y<n+2>, 7, Y<n+2>, 9, depth=2
5  +{a:!end.Y<n+2>,b:W<1>}, 8, Y<n+2>, 9, depth=2
6  +{a:!end.Y<n+2>,b:W<1>}, 8, +{a:!end.Y<n+2>,b:W<1>}, 10, depth=3
7  !end.Y<n+2>, 8, !end.Y<n+2>, 10, depth=4
8  end, 8, end, 10, depth=5
9  Y<n+2>, 8, Y<n+2>, 10, depth=5
10 +{a:!end.Y<n+2>,b:W<1>}, 9, Y<n+2>, 10, depth=5
11 +{a:!end.Y<n+2>,b:W<1>}, 9, +{a:!end.Y<n+2>,b:W<1>}, 11, depth=6
12 !end.Y<n+2>, 9, !end.Y<n+2>, 11, depth=7
13 end, 9, end, 11, depth=8
14 Y<n+2>, 9, Y<n+2>, 11, depth=8
15 +{a:!end.Y<n+2>,b:W<1>}, 10, Y<n+2>, 11, depth=8
16 +{a:!end.Y<n+2>,b:W<1>}, 10, +{a:!end.Y<n+2>,b:W<1>}, 12, depth=9
17 !end.Y<n+2>, 10, !end.Y<n+2>, 12, depth=10
18 W<1>, 10, W<1>, 12, depth=10
19 W<1>, 9, W<1>, 11, depth=7
20 !Z<1>.!Z<1>.end, 0, W<1>, 11, depth=7
21 !Z<1>.!Z<1>.end, 0, !Z<1>.!Z<1>.end, 0, depth=8
22 W<1>, 8, W<1>, 10, depth=4
23 !Z<1>.!Z<1>.end, 0, W<1>, 10, depth=4
24 !Z<1>.!Z<1>.end, 0, !Z<1>.!Z<1>.end, 0, depth=5
25 !end.end, 7, W<1>, 9, depth=1
26 !end.end, 7, !Z<1>.!Z<1>.end, 0, depth=2
27 end, 7, Z<1>, 0, depth=3
28 end, 7, end, 0, depth=4
29 end, 7, !Z<1>.end, 0, depth=3
```

In the first six lines the algorithm behaves as in the first approach. However, instead of stopping there it continues until the intended depth is reached. In line 17 it breaks, and the second branch from line 16 (field b) starts the comparison. The depth is reached right away since we go down a level and we go back to line 11 for the comparison of the next field. In line 21, two equal types are given with the same counters so the algorithm breaks in that branch and go back to the next comparison of line 6 (filed b). In line 24, two equal types and counters are given again and the same happens, the algorithm stops in that branch and the field `b` of line 1 is compared (line 25). Finally, the comparison of the types (`!end.end, !Z<1>.!Z<1>.end`) ends up throwing the *NotEquivalent* exception. The tree representation of these steps is represented in figure 3.5. On the right side of the figure we find the depth and some annotations of the breaks.

As we can see, the two algorithmic approaches lead to different performances. The second approach takes longer to throw the exception.

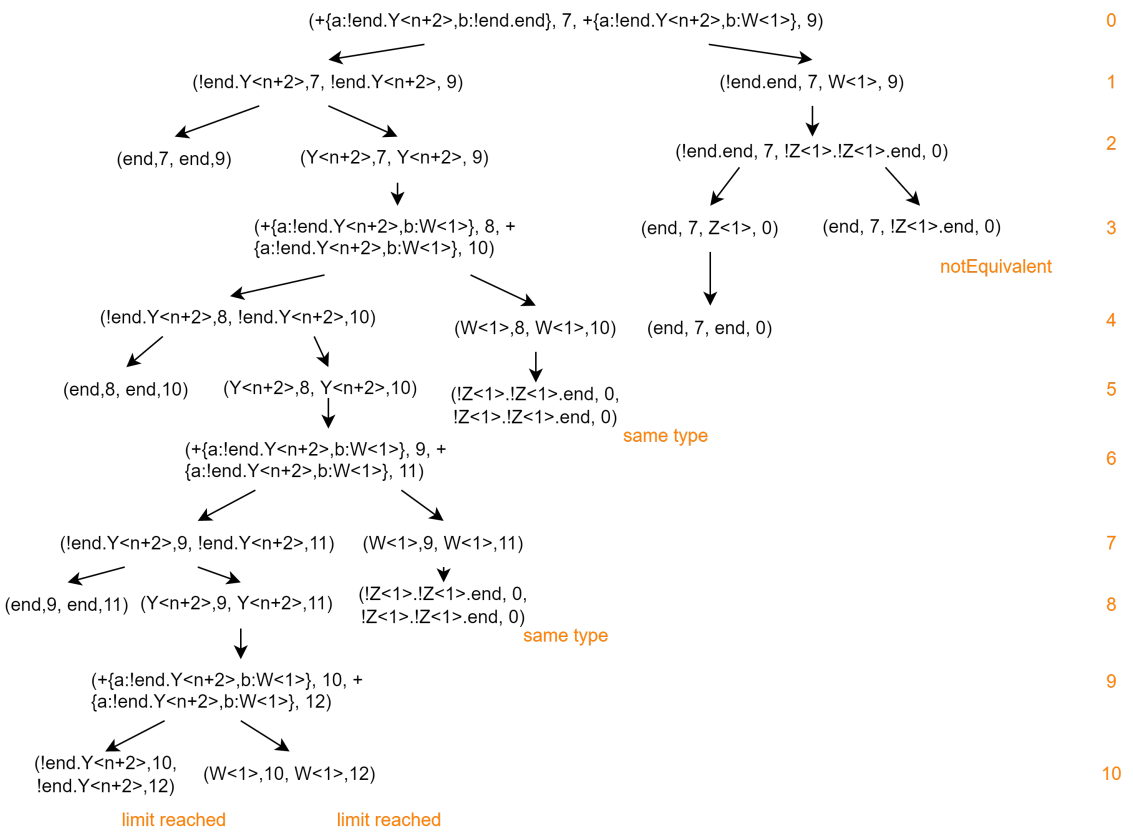

(+{a:!end.Y<n+2>,b:!end.end}, 7, +{a:!end.Y<n+2>,b:W<1>}, 9)
(!end.Y<n+2>,7, !end.Y<n+2>, 9)
(!end.end, 7, W<1>, 9)
(end,7, end,9)
(Y<n+2>,7, Y<n+2>, 9)
(!end.end, 7, !Z<1>.!Z<1>.end, 0)
(+{a:!end.Y<n+2>,b:W<1>}, 8, +{a:!end.Y<n+2>,b:W<1>}, 10)
(end, 7, Z<1>, 0)
(end, 7, !Z<1>.end, 0)
notEquivalent
(!end.Y<n+2>,8, !end.Y<n+2>,10)
(W<1>,8, W<1>,10)
(end, 7, end, 0)
(end,8, end,10)
(Y<n+2>,8, Y<n+2>,10)
(!Z<1>.!Z<1>.end, 0, !Z<1>.!Z<1>.end, 0)
same type
(+{a:!end.Y<n+2>,b:W<1>}, 9, +{a:!end.Y<n+2>,b:W<1>}, 11)
(!end.Y<n+2>,9, !end.Y<n+2>,11)
(W<1>,9, W<1>,11)
(end,9, end,11)
(Y<n+2>,9, Y<n+2>,11)
(!Z<1>.!Z<1>.end, 0, !Z<1>.!Z<1>.end, 0)
same type
(+{a:!end.Y<n+2>,b:W<1>}, 10, +{a:!end.Y<n+2>,b:W<1>}, 12)
(!end.Y<n+2>,10, !end.Y<n+2>,12)
(W<1>,10, W<1>,12)
limit reached
limit reached
0
1
2
3
4
5
6
7
8
9
10

Figure 3.5: Tree representation of the second approach of 3.19

The decision of the best recursion limit was a problem to take into account. Böhm et al. [2] prove that the 1-counter automata equivalence problem is NL-complete. In particular, they show that two deterministic 1-counter automata are not equivalent if and only if there is a distinguishing word of polynomial size. This leads us to the conclusion that two 1-counter types are equivalent if and only if they are equivalent up to n depth, being n polynomial on the input side. We can use this conclusion since Gay et al. [7] show that 1-counter types can be converted to 1-counter automata. We can say that exists a large enough recursion limit n that it will give us the right answer. However in Böhm et al. [2] the proof is very technical and does not immediately give rise to the concrete polynomial n. Using the experience of our tests, a heuristic was made to find the a good choice of recursion limit to be used. The general limit number used is 10, however if there exists a *j* number higher or a higher number of equations, the limit becomes that number. As an example, in 3.22 the recursion limit would be 12.

```
(X<11>, X<12>)
X<0>=end                                   (3.22)
X<n+1>=!end.X<n+1>
```

The decidability of the algorithm is shown in different ways; either an exception is thrown, the limit number is reached, the types are equal, the types were already checked, or it ends in an equal type. If a *NotEquivalent* exception is thrown, then we know for certain that they do not perform the same behaviour. However, if they are considered equivalent two things could have happened. The types could actually be equivalent or, if the roof was reached in one branch, they are considered equal but with a *LimitExceeded* exception since we can not ensure that if the algorithm had continued the types were equivalent.

Chapter 4

Tests

One of the goals of this thesis was to create an extensive pool of tests for our algorithms, making different systems of equations to check a specific possible fail. The test suite was developed incrementally in order to follow the construction of the algorithms, and then more tests were added to the initial set. So, when a new version of an algorithm was made or a new algorithm was made, the old tests were used to validate it and new tests were introduced.

JUnit5 was used to test all the TXT files created that contain the equation systems that we want to test. The tests were divided in four different categories: grammar, construction, contractivity and equivalence. For each category there are valid and invalid tests.

	Grammar	Construction	Contractivity	Equivalence
Valid	100	20	10	20
Invalid	10	20	10	20

Table 4.1: Number of tests for recursive types algorithms

	Grammar	Construction	Contractivity	Equivalence V1	Equivalence V2
Valid	160	10	20	20	20
Invalid	10	20	30	20	20

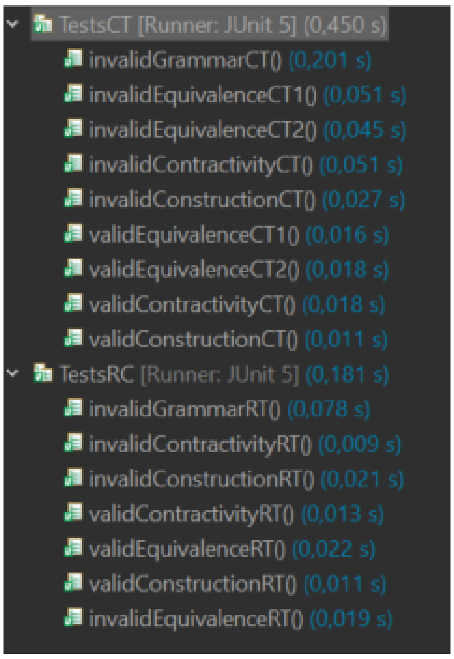
Table 4.2: Number of tests for 1-counter types algorithms

No tests were specifically made for valid grammars since all the other tests require a valid grammar in order to test their main category, so they are the sum of all the other tests. The wrong grammar tests have errors from the ANTLR itself, because a wrong token or a no viable input is given, so we do not advance to our classes.

All invalid tests expect an exception, so if no exception is received, then a warning is shown with each test that has failed. There are four types of exceptions: *BadFormation* exception for construction, *NotContractive* exception for contractivity, *NotEquivalent* exception for equivalence, and there exists one more when the recursion limit is achieved, the *LimitExceeded* exception. For valid tests, if the respective exception appears then it is printed since it should not exist.

The two versions of equivalence algorithm for 1-counter types use the same set of tests. There are a total of 240 distinct tests, divided into 100 valid tests and 140 invalid tests.

Figure 4.1 show all the tests for 1-counter types (TestCT) and recursive types (TestRC). As we can see, the two approaches for the equivalence algorithm in 1-counter types, both in valid and invalid tests, take similar time to execute.

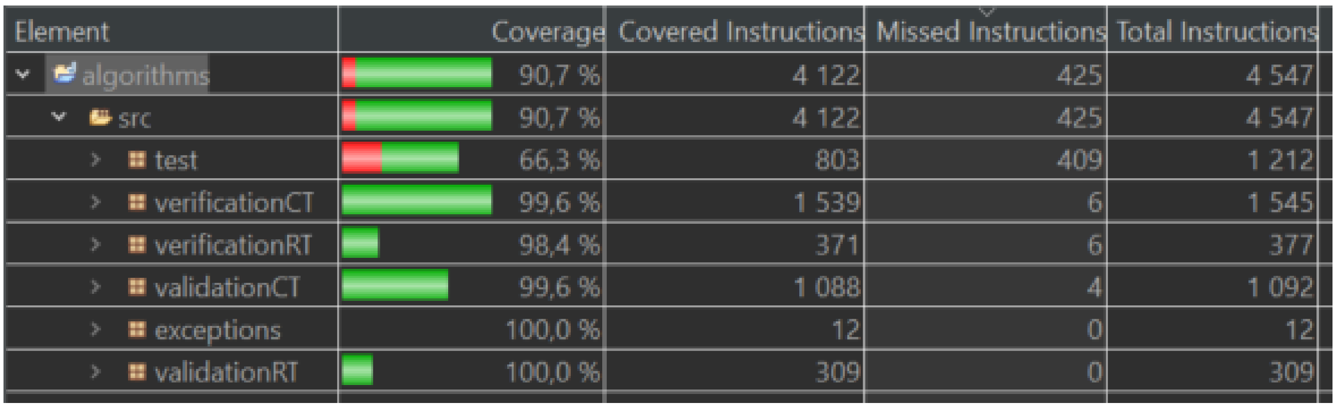


```
TestsCT [Runner: JUnit 5] (0,450 s)
    invalidGrammarCT() (0,201 s)
    invalidEquivalenceCT1() (0,051 s)
    invalidEquivalenceCT2() (0,045 s)
    invalidContractivityCT() (0,051 s)
    invalidConstructionCT() (0,027 s)
    validEquivalenceCT1() (0,016 s)
    validEquivalenceCT2() (0,018 s)
    validContractivityCT() (0,018 s)
    validConstructionCT() (0,011 s)
TestsRC [Runner: JUnit 5] (0,181 s)
    invalidGrammarRT() (0,078 s)
    invalidContractivityRT() (0,009 s)
    invalidConstructionRT() (0,021 s)
    validContractivityRT() (0,013 s)
    validEquivalenceRT() (0,022 s)
    validConstructionRT() (0,011 s)
    invalidEquivalenceRT() (0,019 s)
```

Figure 4.1: Test classes summary

In order to assist verification of the tests, we used a Java code coverage tool for Eclipse called EclEmma. EclEmma is based on the JaCoCo code coverage library that presents the line and branch coverage results immediately summarized and highlighted in the Java source code editors. All the lines in the code are colored with green if they are executed, yellow if the lines are partly covered (not all the instructions associated with this line have been executed), or red if they are not executed. This way, we ensure that all the options in the algorithms are visited. The tool was taken from the Eclipse Marketplace and the test files run with it. The ANTLR classes were excluded of the evaluation, since we are just interested in the algorithms we implemented.

As we can see in figure 4.2 the full coverage of the project is 90.7%, however this counts with the coverage of all the classes, including the test classes (can not be removed from the set) that have multiple try/catch that are not visited unless some test goes wrong, which lowers the percentage. If we calculate the coverage without the test package then we achieve a line and branch coverage of 99.43%.



Element	Coverage	Covered Instructions	Missed Instructions	Total Instructions
algorithms	90,7 %	4 122	425	4 547
src	90,7 %	4 122	425	4 547
test	66,3 %	803	409	1 212
verificationCT	99,6 %	1 539	6	1 545
verificationRT	98,4 %	371	6	377
validationCT	99,6 %	1 088	4	1 092
exceptions	100,0 %	12	0	12
validationRT	100,0 %	309	0	309

Figure 4.2: Algorithms line and branch coverage

Chapter 5

Conclusion

In this thesis we studied two different extensions of session types, recursive and 1-counter types. We presented and implemented type formation, type contractivity and type equivalence algorithms for them. We also developed a battery of tests for our algorithms. Extending session types into infinity and studying different classes of types was necessary.

All the algorithms aim to establish correctness of the finite and infinite session types systems. The grammars created with ANTLR helped defining multiple rules for the input. With this tool we were able to search for a specific rule and make our verifications.

From the simplest to the more complex protocols, it is important to verify equivalence/inequivalence between representation of types. The comparison of the equivalent computations in finite time was accomplished in a practical algorithm, a problem that was thought of at the beginning of this work. We used a tree-like search for each branch that was extended from unfolding the types in the system, and made a recursion limit for the infinite unfolding of them so that a conclusion could be made at some point. The contractivity problem in the systems was also resolved since no system can be compared if they are not contractive, as a necessary intermediate step in solving the equivalence problem. This way, we are not stuck in systems that have no operation to be considered.

The tests created increase the algorithm reliability. Various tests were made trying to catch a specific error or rule. Each algorithm has multiple systems that visit a part of the code. The systems of equations that we tested can be used for future verifications of languages that use session types as their bases such as SePi.

There are some aspects of the algorithms that we leave for future work. For the equivalence of 1-counter types finding the perfect recursion limit for all cases is one of them, since the one written is based in our test experience and a mathematically provably correct limit would make the algorithms formally reliable. Another important aspect is to implement these algorithms for the SePi language, since it was one of the purposes of this thesis and would increase both our algorithms as well as the SePi language itself. A type checking algorithm could also be considered for future work. A type checking algorithm receives a type as well as a program and verifies whether the program is correctly typed. This algorithm would also be valuable for session type languages, verifying the assignment of types to values in various circumstances. Finally, another very useful future work would be a creation of an automatic test generator tool to increase the confidence of our algorithms, by generating a large number of random tests, the probability of discovering code errors would improve.

Bibliography

[1] Antlr. https://www.antlr.org/.

[2] Stanislav Böhm, Stefan Göller, and Petr Jancar. Equivalence of deterministic one-counter automata is NL-complete. *CoRR*, abs/1301.2181, 2013.

[3] Ankush Das, Henry DeYoung, Andreia Mordido, and Frank Pfenning. Nested session types. *CoRR*, abs/2010.06482, 2020.

[4] Romain Demangeon and Kohei Honda. Nested protocols in session types. In Maciej Koutny and Irek Ulidowski, editors, *CONCUR 2012 - Concurrency Theory - 23rd International Conference, CONCUR 2012, Newcastle upon Tyne, UK, September 4-7, 2012. Proceedings*, volume 7454 of *Lecture Notes in Computer Science*, pages 272–286. Springer, 2012.

[5] Juliana Franco and Vasco Thudichum Vasconcelos. A concurrent programming language with refined session types. In Steve Counsell and Manuel Núñez, editors, *Software Engineering and Formal Methods - SEFM 2013 Collocated Workshops: BEAT2, WS-FMDS, FM-RAIL-Bok, MoKMaSD, and OpenCert, Madrid, Spain, September 23-24, 2013, Revised Selected Papers*, volume 8368 of *Lecture Notes in Computer Science*, pages 15–28. Springer, 2013.

[6] Simon J. Gay and Malcolm Hole. Subtyping for session types in the pi calculus. *Acta Informatica*, 42(2-3):191–225, 2005.

[7] Simon J. Gay, Diogo Poças, and Vasco T. Vasconcelos. The different shades of infinite session types. In Patricia Bouyer and Lutz Schröder, editors, *Foundations of Software Science and Computation Structures - 25th International Conference, FOSSACS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings*, volume 13242 of *Lecture Notes in Computer Science*, pages 347–367. Springer, 2022.

[8] Kohei Honda. Types for dyadic interaction. In Eike Best, editor, *CONCUR '93, 4th International Conference on Concurrency Theory, Hildesheim, Germany, August*

23-26, 1993, Proceedings, volume 715 of *Lecture Notes in Computer Science*, pages 509–523. Springer, 1993.

[9] Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In Chris Hankin, editor, *Programming Languages and Systems - ESOP'98, 7th European Symposium on Programming, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'98, Lisbon, Portugal, March 28 - April 4, 1998, Proceedings*, volume 1381 of *Lecture Notes in Computer Science*, pages 122–138. Springer, 1998.

[10] Robin Milner. *Communicating and mobile systems - the Pi-calculus*. Cambridge University Press, 1999.

[11] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, I. *Inf. Comput.*, 100(1):1–40, 1992.

[12] Benjamin C. Pierce. *Types and programming languages*. MIT Press, 2002.

[13] Peter Thiemann and Vasco T. Vasconcelos. Context-free session types. In Jacques Garrigue, Gabriele Keller, and Eijiro Sumii, editors, *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*, pages 462–475. ACM, 2016.

[14] Vasco T. Vasconcelos. Sessions, from types to programming languages. *Bull. EATCS*, 103:53–73, 2011.

[15] Vasco T. Vasconcelos. Fundamentals of session types. *Inf. Comput.*, 217:52–70, 2012.