UNIVERSIDADE DE LISBOA
FACULDADE DE CIÊNCIAS
DEPARTAMENTO DE INFORMÁTICA

Subtyping Context-Free Session Types

Gil Afonso Tavares Pereira da Silva

Mestrado em Informática

Dissertação orientada por:
Prof. Doutor Vasco Manuel Thudichum de Serpa Vasconcelos
Prof.ª Doutora Andreia Filipa Torcato Mordido Rodrigues

2024

Acknowledgments

I thank my advisors, Vasco T. Vasconcelos and Andreia Mordido, for their kindness, guidance and patience. Without them, this work would not be possible. I give my thanks also to Luca Padovani, Diana Costa and Diogo Poças for their insightful comments on the draft that eventually became this thesis, and Henry DeYoung for accepting to review my intermediate report.

I thank also my parents and my sister for their continuous and unconditional support during my studies, my good friend Isabel for encouraging me to pursue this project, and my past professors for sharing the knowledge and skills that enabled me to complete this work.

Finally, I must thank all of my friends and my grandmother for both motivating and distracting me, as necessary.

Funding This work was supported by FCT through project SafeSessions, ref. PTDC/CCI-CIF/6453/2020 (https://doi.org/10.5449/PTDC/CCI-COM/6453/2020) and the LASIGE Research Unit, ref. UIDB/00408/2020 (https://doi.org/10.5449/UIDB/00408/2020) and ref. UIDP/00408/2020 (https://doi.org/10.5449/UIDP/00408/2020).

Resumo

Os tipos de sessão [51, 52, 94] permitem a especificação de protocolos de comunicação em canais bidirecionais e heterogéneos. Tipicamente, estas especificações incluem o tipo, direção e ordem das mensagens, bem como pontos de ramificação onde os participantes podem escolher como deve prosseguir a comunicação.

Ao suportar tipos de sessão, as linguagens de programação conseguem oferecer mais segurança na programação concorrente, permitindo ao seu verificador de tipos assegurar a *fidelidade das sessões* (i.e., que a comunicação procede de acordo com o protocolo), a *privacidade* (i.e., que os canais de comunicação são conhecidos apenas pelos participantes) e a *integridade da comunicação* (i.e., que não há incompatibilidades resultantes da ordem e tipos das mensagens). Em certas formulações, também é possível assegurar que a comunicação nunca chega a um impasse [53].

Até recentemente, os tipos de sessão apenas permitiam a recursão terminal, e estavam, portanto, limitados à especificação de protocolos correspondentes a linguagens regulares (em particular, à união das linguagens regulares e ω-regulares). Esta classe de linguagens exclui muitos protocolos de interesse prático, sendo o exemplo mais típico a serialização de dados com estrutura de árvore num só canal. Os tipos de sessão livres de contexto, propostos por Thiemann e Vasconcelos [96], libertam os tipos de sessão desta restrição, permitindo a recursão não-terminal através de um operador de composição sequencial que forma um monóide com o elemento neutro Skip (um tipo que representa o protocolo vazio, i.e., sem ação). Tal como o nome indica, os tipos de sessão livres de contexto conseguem especificar protocolos correspondentes a linguagens livres de contexto (determinísticas e simples [62]), e são portanto consideravelmente mais expressivos que os tipos de sessão convencionais.

Na sua busca pela fiabilidade, os tipos de sessão podem tornar a programação demasiado rígida para os programadores. A subtipagem, um aspeto central de muitos sistemas de tipos, procura aliviar a tensão entre a fiabilidade e a flexibilidade. É tipicamente justificada por um apelo ao *princípio da substituição segura* de Liskov [65]: um tipo T é considerado subtipo de U se um valor do tipo T puder ser usado no lugar de um valor do tipo U em qualquer contexto, sem violar as propriedades desejáveis do sistema de tipos em questão.

O que significa dizer que um tipo de sessão livre de contexto é subtipo de outro? Uma possível resposta, seguindo o princípio de Liskov, pode ser encontrada no influente trabalho de Gay e Hole [42] sobre os tipos de sessão regulares: um tipo de sessão S é subtipo de R se um canal governado pelo tipo S puder ser usado em qualquer contexto onde é esperado um canal governado pelo tipo

R. De forma mais concreta, a subtipagem para tipos de sessão permite uma maior flexibilidade nos tipos das mensagens trocadas e nas escolhas oferecidas aos participantes, sem comprometer as garantias mencionadas acima. Na prática, esta flexibilidade promove a modularidade no desenvolvimento de software concorrente: o protocolo de um participante pode ser refinado, mantendo intacto o do seu correspondente.

Enquanto que propriedades algorítmicas da subtipagem para os tipos de sessão regulares são bem conhecidas (graças também ao trabalho de Gay e Hole), o mesmo não pode ser dito sobre os os tipos de sessão livres de contexto. Os maiores desafios neste sentido são as suas propriedades algébricas (identidade, associatividade, distributividade e absorção), a sua interpretação equirecursiva e a possibilidade de recursão não-terminal. A investigação de Padovani [82] sobre este tópico mostrou que o problema da subtipagem para estes tipos, quando formulado segundo os princípios clássicos de Gay e Hole, é indecidível—o preço habitual a pagar por mais expressividade.

Apesar destes desafios, propomos duas abordagens à subtipagem para os tipos de sessão livres de contexto: uma sintática e baseada em regras de inferência, e outra semântica e apoiada numa nova noção de pré-ordem observacional a que chamamos de *simulação-$\mathcal{XYZW}$*, uma generalização da noção de *simulação-$\mathcal{XY}$* proposta por Aarts e Vaandrager [1]. Esta relação permite uma combinação seletiva dos requisitos da simulação simples, simulação inversa e de uma certa forma de contra-simulação, o que lhe permite suportar a habitual *covariância* e *contravariância* dos construtores de tipos de sessão na sua plenitude (em contraste, a relação proposta por Padovani não permite qualquer variância nos tipos das mensagens).

A equivalência dos tipos de sessão livres de contexto é baseada na noção de *bissimulação* [83], a qual a simulação-$\mathcal{XYZW}$ generaliza. Tirando partido deste facto, derivamos um algoritmo de subtipagem a partir de um algoritmo de equivalência já existente [5]. O nosso algoritmo é correto (i.e., as suas respostas afirmativas são verdadeiras) mas, devido à indecidibilidade do problema, necessariamente incompleto.

Este algoritmo foi implementado no verificador de tipos da linguagem FREEST [2, 4, 95], uma linguagem de programação funcional com suporte para tipos de sessão livres de contexto. De forma a avaliar a sua performance, construímos um conjunto de 4000 testes, gerados automaticamente com a ajuda da biblioteca Quickcheck para a linguagem Haskell [22]. Obtivémos uma performance satisfatória, observando apenas 200 timeouts. Para uma avaliação mais realista, e de forma a comparar a nossa adaptação com o algoritmo original, corremos ambos os algoritmos lado-a-lado num conjunto de 288 programas FREEST sem subtipagem. Nesta avaliação não observamos diferenças significativas de performance, o que sugere que a aplicação prática do nosso algoritmo é viável.

O trabalho desenvolvido no âmbito desta tese foi apresentado na 34ª Conferência Internacional de Teoria da Concorrência (CONCUR 2023) e no INForum 2023, Simpósio de Informática, e publicado nas atas correspondentes sob a forma de artigo [90] e resumo [92], respetivamente.

Palavras-chave: Tipos de sessão, Subtipagem, Simulação, Recursão não-terminal, Gramáticas simples

Abstract

Session types allow the specification of structured communication protocols on bidirectional, heterogeneously typed channels. Typically, these specifications include the type, direction and order of messages, as well as branching points where participants can choose how communication should proceed. By supporting session types, programming languages are able to offer safer concurrency, ensuring session fidelity, privacy and communication safety. Until recently, session types were restricted to the description of regular, tail-recursive protocols, considerably limiting their expressiveness. Context-free session types liberate themselves from this restriction by introducing a monoidal type operator to express the sequential composition of any two protocols, making them significantly more expressive.

However, in their quest for safe concurrency, session types can make programming too rigid. Subtyping, an important feature of many type systems, is meant to alleviate this tension between safety and flexibility. While the algorithmic properties of subtyping for regular session types are by now well known, the same cannot be said for their context-free counterparts. The main challenges in this respect are the algebraic properties they exhibit, along with their equirecursive interpretation. A previous investigation into this topic by Padovani has shown this problem to be undecidable—the usual, unfortunate price to pay for expressive power.

Despite these challenges, we propose two novel approaches to subtyping for context-free session types: one syntactic and based on inference rules, and another semantic and based on a novel kind of observational preorder we call $\mathcal{XYZW}$-simulation, which generalizes $\mathcal{XY}$-simulation and therefore also bisimulation, on which type equivalence for context-free session types is often based. We take advantage of this fact to derive a sound (but, due to the undecidability of the problem, necessarily incomplete) subtyping algorithm from an existing type equivalence algorithm. We then present an empirical evaluation of its performance in the context of a programming language compiler.

Keywords: Session types, Subtyping, Simulation, Non-tail recursion, Simple grammars

Contents

List of Figures

List of Listings

Chapter 1

Introduction

Communication and concurrency are increasingly important aspects of software systems. Whereas in the past computers, heavy and expensive, acted as centralized units for storing, processing and retrieving data, they have now miniaturized, multiplied and arranged themselves into vast distributed networks, forming the backbone of transportation, banking and telecommunications systems. Their performance has also increased significantly, in no small part due to the parallel processing techniques enabled by the advent of the multi-core processor.

Building safe, high-quality software in this scenario is no easy task, and failure can be costly. On the one hand, designers must carefully consider how system components should coordinate, identifying communication structures and formalizing them in protocols. On the other hand, programmers must also take care to implement these protocols correctly.

Types

As systems grow in size and complexity, programs become harder to reason about, verify and maintain. We cannot rely on developers' judgments alone—automatic and reliable software verification tools are necessary. Of all software verification tools, *type systems* are by far the most accessible. Being an integral part of most programming language compilers, they ensure programs are (to some extent) correct even before they are executed. The information they provide can also be used by code editors to guide the development process with immediate feedback and suggestions about programs as they are being written.

A *type* can be understood as the interface that a software component (e.g. a function, a record, an object, etc.) presents to the outside, an approximation of its run-time behavior and constraints. By assigning types to software components, type systems are able to verify their correct composition by identifying and rejecting invalid operations on these interfaces (e.g. supplying an integer to a function expecting a string, invoking a method on an object does not offer it, etc.). In short, type systems ensure programs have computational *meaning*, or, as famously put by Robin Milner, that they do not "go wrong"[70]. When made explicit through type signatures, types are also a useful tool for abstraction and documentation: developers can gather hints about the behavior and constraints of a software component by simply looking at its declared type.

Subtyping

To be usable, type systems must be flexible. When piecing together software components, programmers do not need their interfaces to match exactly—they only need them to be compatible. In other words, a component should be able to take on a type simpler than its own if the context so requires, provided that this substitution does not break the guarantees offered by the type system. This is the guiding principle of subtyping, a standard feature of type systems that can be found in many modern typed programming languages. Subtyping is typically demonstrated using records (in functional languages) or objects (in object-oriented programming). Consider, for example, types $\mathsf{Person} = \{\mathsf{name}\colon \mathsf{String}, \mathsf{age}\colon \mathsf{Int}\}$ and $\mathsf{Student} = \{\mathsf{name}\colon \mathsf{String}, \mathsf{age}\colon \mathsf{Int}, \mathsf{gpa}\colon \mathsf{Float}\}$.[1] If we ignore the gpa field, any value of type Student can be used as if it had type Person. In this case, Student is said to be a subtype of Person, and Person a supertype of Student[2]. The inverse is not true, however: accessing the gpa field on Person would result in a run-time error.

Session types

Traditional type systems are concerned with data, its structure, and the valid operations that may be performed on it. Most modern typed programming languages provide some mechanism to describe structured data and enforce its correct manipulation using types (e.g. *structs* in C and Go, *enums* in Rust and Scala, *data types* in Haskell, etc.). But as computation grows increasingly concurrent, programs must do more than just process data—they must exchange it among independent, concurrent processes, each with its own concerns. Programmers, then, need an analogous mechanism to describe and enforce structured communication patterns.

Session types, introduced by Honda et al. [51, 52, 94], allow just this. They enhance traditional type systems with the ability to specify and enforce communication protocols on bidirectional, heterogeneously typed channels. Typically, protocol specifications include the type, direction (input or output) and order of the messages, as well as branching points where participants can choose how communication should proceed. These specifications may also be recursive, allowing for repeated, potentially infinite communication behavior (typically seen in servers, streams, etc.).

Context-free session types

Classical session types are somewhat restricted in terms of the protocols they allow. They are limited to what is known as *tail recursion*: protocols may recur, but only in their last step. In formal language terms, classical session types are restrained to the description of protocols that correspond to the class of *regular languages* (earning them the *regular* epithet). As it turns out, this restraint excludes many protocols of practical interest, with the quintessential example being the serialization of tree-structured data (e.g. binary trees, JSON, etc.) on a single channel.

[1] These types could be interpreted as either records (e.g., as in OCaml) or objects (e.g., as in TypeScript). We are deliberately ambiguous here, as the point applies to both.

[2] Notice the parallel between subtyping and *hyponymy*: the noun "student" is a hyponym of "person".

Context-free session types, proposed by Thiemann and Vasconcelos [96], liberate protocols from tail recursion by allowing them to be sequentially composed in an arbitrary manner and enabling them to recur at any point. As their name hints, they can express protocols that correspond to the class of (*simple, deterministic*) *context-free languages*, which properly contains regular languages. They are thus considerably more expressive than their classical, regular counterparts.

Subtyping context-free session types

When applied to session types, subtyping allows increased flexibility in the interactions between participants, namely on the type of the messages and on the choices available at branching points, allowing communication channels to be governed by simpler session types if their context so requires. Subtyping for regular session types has been formalized, shown decidable and given an algorithm by Gay and Hole [42].

While not fundamentally different, subtyping for context-free session types has until now only been considered briefly, and in a limited form, by Padovani [82]. This existing notion allows variance in choices, but not in the type of messages—a crucial feature of Gay and Hole's notion (and indeed of subtyping for channel types in general [85]). Even if limited, this formulation suffices to for Padovani to show that subtyping for context-free session types is undecidable—the usual, unfortunate price to pay for expressive power. Despite this negative result, our goal for this thesis is to propose and implement a more expressive notion of subtyping, in which both choices and input/output may vary according to the principles put forth by Gay and Hole.

While initially formulated in the context of the π-calculus, considerable work has been done to integrate session types in more standard settings, such as functional languages based on the linear polymorphic λ-calculus [2, 23, 86]. With this in mind, we develop our theory in a linear functional setting, showing how subtyping for records, variants and (linear and unrestricted [39]) functions can be seamlessly integrated with subtyping for context-free session types.

As customary, we begin by defining our notion of subtyping by means of a preorder relation, defined by a set of inference rules that analyze and decompose the syntactic structure of types to determine the relationship between them. Finding this relation to be unsuitable for algorithmic treatment, we turn our attention away from the structure of types to focus on the communication behaviors they describe, which are easily modeled by *labeled transition systems* and compared using an appropriate notion of simulation preorder. Here we find that, while subtyping in choices can be captured by the $\mathcal{XY}$*-simulation* of Aarts and Vaandrager [1], no known notion is able to express subtyping in both choices and message types. As such, we introduce a novel notion of simulation preorder called $\mathcal{XYZW}$*-simulation*, which generalizes $\mathcal{XY}$*-simulation* [1] and, as a corollary, conventional notions like (plain) simulation and bisimulation.

Taking advantage of this corollary, we present a sound algorithm for our notion of subtyping, based on the type bisimilarity algorithm of Almeida et al. [5]. This algorithm works by first encoding the types as words from a simple grammar and then deciding their $\mathcal{XYZW}$-similarity. Being grammar-based and, at its core, agnostic to types, this algorithm may also find applications

in other domains where non-regularity and contravariance play a part.

Finally, we describe the implementation of this algorithm in the type checker for FreeST [2, 4, 95], a functional programming language with support for context-free session types. In order to evaluate the performance of our implementation, we designed a suite of 4000 tests, automatically generated with the aid of the QuickCheck [22] testing framework for the Haskell programming language, in which the FreeST compiler is written. We obtained satisfactory results, observing only 200 timeouts under a limit of 30s. For a more realistic evaluation, and in order to compare our adaptation with the original, we ran both algorithms side by side on a suite of 288 FreeST programs exhibiting no subtyping. We did not observe significant differences in performance, which suggests that our algorithm is viable for practical application.

Contributions We address the subtyping problem for context-free session types, proposing:

- A syntactic, rule-based definition of subtyping for higher-order context-free session types;
- A novel notion of observational preorder, $\mathcal{XYZW}$-simulation;
- Based on this notion, a semantic subtyping relation for higher-order context-free session types that also encompasses functional types;
- A sound subtyping algorithm based on the semantic subtyping relation;
- An implementation of the algorithm in the freely available FreeST language compiler[95];
- An empirical evaluation of the performance of the algorithm, based on a custom generative testing framework developed with the aid of the QuickCheck library [22]
- A performance comparison with the equivalence algorithm of Almeida et al. [5].

Overview The first four chapter of this thesis set the stage for our contributions: in Chapter 2 we introduce the notion of types and subtyping in a functional setting; in Chapter 3 we introduce session types in their classical (regular) form, as well as Gay and Hole's notion of subtyping; in Chapter 4 we introduce context-free session types and review previous work on their equivalence and subtyping. In the remaining chapters we present our own work: in Chapter 5 we introduce two novel, coinciding notions of syntactic and semantic subtyping for context-free session types; in Chapter 6 we describe a sound algorithm based on our semantic notion of subtyping; in Chapter 7, after a brief introduction to the FreeST programming language [2, 4, 95], we present some FreeST programs that take advantage of subtyping and describe the changes that were necessary to include this feature in the language compiler; in Chapter 8 we describe our automatic testing framework and evaluate the performance of our implementation; finally, in Chapter 9 we conclude the thesis and trace a path for the work to follow.

Related publications Besides this thesis, our work has resulted in two publications: a conference paper [90] (accompanied by a technical report [91]), presented at CONCUR 2023, the 34th International Conference on Concurrency Theory in Antwerp, Belgium, and an extended abstract [92], presented at INForum 2023, Simpósio de Informática in Porto, Portugal.

Chapter 2

Typing and subtyping

In the introduction, we discussed the usefulness of type systems, session types and subtyping in abstract terms. To make matters more concrete, and to set the stage for our contributions, we begin to introduce these topics more formally, in a functional setting. This presentation owes much to Pierce [84] and Sangiorgi [88]. Readers familiar with type systems, induction and coinduction may safely skip this chapter.

2.1 A simple functional language

Programming languages express computations, or how to produce a certain outcome (e.g. the solution to a problem, the desired behavior of a system) from an initial set of conditions, following well-defined rules.

Functions are mathematical objects that formalize these intuitive notions of input and output, and are thus quite apt to model computations in a way that is amenable to formal analysis. They are the basis of the λ-calculus, a universal model of computation capable of expressing the steps necessary to obtain the solution to any theoretically solvable problem [21, 97]. Developed by Alonzo Church in the 1930s, λ-calculus is only one such model: Turing machines [98], combinatory logic [25] and the π-calculus [73, 72] are examples of alternate formalisms with equivalent computational power.

Programming languages typically take these computational models as the basis for their syntax and semantics. The λ-calculus is the direct inspiration for many of the so-called functional languages (e.g. Scheme [79], ML [75], Haskell [60], etc.), and its concepts have also found their way to popular imperative languages such as Python and Java. Functional programming languages typically favor a small but powerful set of constructs, and as such are easy to learn and analyze. Thus, in order to understand the importance of typing and subtyping at the level of structured data, we introduce a simple functional language with integers, booleans, records and variants (which correspond roughly to *data types* in Haskell, OCaml, and other ML-like languages).

expressions

$$e ::= \lambda x \to e \mid e\, e \mid x \mid i \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{if}\, e\, \mathsf{then}\, e\, \mathsf{else}\, e \mid \mathsf{isZero}\, e \mid \{\ell = e_\ell\}_{\ell \in L} \mid e.\ell$$

Figure 2.1: Syntax for a simple functional language.

Expressions

Our language is based on *expressions*, syntactic structures that are either atomic or composed from other expressions of arbitrary complexity, and that reduce to simpler expressions according to well-defined criteria, much like elementary algebra. And just like in elementary algebra, some expressions do not need to be reduced: these are called *values*. The meaning (or result) of an expression is the value it reduces to after finitely many steps, and reducing an expression to a value is known as *evaluation*.

Each sort of expression (denoted by *metavariable* e) represents a different computational construct. According to the syntax in Fig. 2.1, our language includes functional abstractions ($\lambda x \to e$, corresponding to "anonymous functions" or "lambda expressions" in mainstream programming languages), function application ($e_1\, e_2$, which applies function e_1 to argument e_2), variables (x, y, z, f, etc.), integers (e.g., 0, 2, etc., denoted by metavariable i), booleans and conditional expressions (true, false and $\mathsf{if}\, e_1\, \mathsf{then}\, e_2\, \mathsf{else}\, e_3$) and zero testing ($\mathsf{isZero}\, e$). To provide for some form of data aggregation, the language also allows building records ($\{\ell = e_\ell\}_{\ell \in L}$) and accessing their fields ($e.\ell$). The values in our language are abstractions, integers, booleans and records in which all fields are also values.

The meaning of expressions

The meaning of our expressions is the value they eventually reduce to, after finitely many reduction steps. How do we know an expression reduces to another, formally? The answer is given by a *reduction relation*, a binary relation (a set of pairs), denoted by $\hookrightarrow$, that associates an expression e_1 the one it reduces to e_2. We write $e_1 \hookrightarrow e_2$ to mean $(e_1, e_2) \in \hookrightarrow$. Reduction is, of course, not arbitrary—it follows certain well-defined criteria, which the relation must respect. These criteria can be expressed in the form of inference rules: assuming a (possibly empty) set of statements (called *premises*) to be true, we can infer another statement (called the *conclusion*). An inference rule with no premises and conclusion c is typically written as simply c, and is called an *axiom*. An inference rule with premises $p_1 \ldots p_n$ and conclusion c is typically written as follows.

$$\frac{p_1 \qquad \cdots \qquad p_n}{c}$$

Inference rules allow us prove (or *derive*) statements systematically, as a finite series of inferences: to prove that a certain statement holds, we look for an inference rule for which it is a valid conclusion, and then prove that its premises hold. We recursively apply the same reasoning to the premises, until reaching a rule with no premises (called an *axiom*, which expresses that a statement

$$\textsc{R-App1}\ \frac{e_1 \hookrightarrow e_1'}{e_1\, e_2 \hookrightarrow e_1'\, e_2} \qquad \textsc{R-App2}\ \frac{e_1 \text{ is a value} \qquad e_2 \hookrightarrow e_2'}{e_1\, e_2 \hookrightarrow e_1\, e_2'} \qquad \textsc{R-AppAbs}\ \frac{e_2 \text{ is a value}}{(\lambda x \to e_1)\, e_2 \hookrightarrow [e_2/x]e_1}$$

$$\textsc{R-If}\ \frac{e_1 \hookrightarrow e_1'}{\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \hookrightarrow \mathsf{if}\ e_1'\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3} \qquad \textsc{R-IfTrue}\ \mathsf{if\ true\ then}\ e_2\ \mathsf{else}\ e_3 \hookrightarrow e_2$$

$$\textsc{R-IfFalse}\ \mathsf{if\ false\ then}\ e_2\ \mathsf{else}\ e_3 \hookrightarrow e_3 \qquad \textsc{R-IsZero}\ \frac{e \hookrightarrow e'}{\mathsf{isZero}\ e \hookrightarrow \mathsf{isZero}\ e'} \qquad \textsc{R-IsZeroTrue}\ \mathsf{isZero}\ 0 \hookrightarrow \mathsf{true} \qquad \textsc{R-IsZeroFalse}\ \frac{i \neq 0}{\mathsf{isZero}\ i \hookrightarrow \mathsf{false}}$$

$$\textsc{R-Rcd}\ \frac{k \in L \qquad e_k \hookrightarrow e_k' \qquad L' = L \setminus \{k\}}{\{\ell = e_\ell\}_{\ell \in L} \hookrightarrow \{\ell = e_\ell, k = e_k'\}_{\ell \in L'}} \qquad \textsc{R-Acc1}\ \frac{e \hookrightarrow e'}{e.\ell \hookrightarrow e'.\ell} \qquad \textsc{R-Acc2}\ \frac{e_\ell \text{ is a value } (\forall \ell \in L) \qquad k \in L}{\{\ell = e_\ell\}_{\ell \in L}.k \hookrightarrow e_k}$$

(no rule for x)

Figure 2.2: Reduction rules for the simple functional language.

trivially holds). This is called an *proof by rule induction*. A derivation is typically expressed using a similar notation to that used for inference rules.

$$\frac{\begin{array}{ccc}\vdots & & \vdots \\ \overline{p_1} & \cdots & \overline{p_n}\end{array}}{c}$$

The inference rules governing the reduction relation for our functional language are given in Fig. 2.2. Naturally, they all have a statement (more precisely, what we call a *judgment*) of the form $e_1 \hookrightarrow e_2$ as the conclusion. With such rules, we can define our reduction relation as follows: for some expressions e_1 and e_2, the statement $e_1 \hookrightarrow e_2$ holds if it can be derived from the inference rules in Fig. 2.2 in a finite number of steps.[1] To actually construct the relation $\hookrightarrow$ (which is a set of pairs of expressions), we proceed iteratively: starting with the empty set $\emptyset$, we add to it every pair (e_1, e_2) that matches the rules without premises of the form $e \hookrightarrow e$ (R-AppAbs, R-IfTrue, R-IfFalse, R-IsZeroTrue, R-IsZeroFalse and R-Acc2). Then, we repeatedly add every pair that matches the remaining rules if premises of the form $e \hookrightarrow e$ are satisfied by the pairs already in the set. This is called an *inductive definition*. We now give an intuitive justification for each of the rules, explaining how they can be used to produce a sequence of reduction steps from an expression to a value. To reduce an application, we first reduce the applicand to a value (R-App1), then reduce the argument (R-App2), and finally, if the applicand is an abstraction, we substitute the argument for the bound variable in the body of the abstraction (R-AppAbs, where substituting expression e_2 for variable x in expression e_1 is denoted by $[e2/x]e1$). To reduce a

[1] Symbols like e or e_1 are *metavariables*: they are implicitly universally quantified ("for any expression e..."), and meant to be instantiated by concrete expressions like true, 84 or $(\lambda x \to \mathsf{if}\ x\ \mathsf{then}\ 1\ \mathsf{else}\ 0)$.

conditional expression, we reduce the condition to a value (R-IF), and if it is true, we reduce to the expression following then (R-IFTRUE), and if it is false we reduce to the expression following else (R-IFFALSE). To reduce an isZero expression, we must first reduce the enclosed expression to a value. If this value is 0, the expression reduces to true (R-ISZEROTRUE), if it is any other integer, then the expression reduces to false (R-ISZEROFALSE). Records are reduced by successively reducing each of their fields (R-RCD). Field access is reduced by first reducing the accessed expression to a record (R-ACC1) and then to the expression corresponding to the selected field (R-ACC2). Finally, notice that variables by themselves do not reduce; they make no sense unless bound by an abstraction.

Note 2.1.1. Unbound variables are also said to be *free*. Expressions containing free variables are said to be *open*, and otherwise *closed*. Bound variables can be renamed at will: we consider expressions $(\lambda x \to \mathsf{isZero}\, x)$ and $(\lambda y \to \mathsf{isZero}\, y)$, for example, to be equivalent and interchangeable (more precisely, they are said to be α-equivalent).

Note 2.1.2. Our syntax does not include common language constructs like local variable definitions ($\mathsf{let}\ x = e\ \mathsf{in}\ e$) or sequential composition ($e; e$). These constructs can, however, be derived from those we already have. Notice that, according to our reduction rules, writing $\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$ amounts to writing $(\lambda x \to e_2)\, e_1$, while writing $e_1; e_2$ amounts to writing $(\lambda x \to e_2)\, e_1$ for some x not occurring in e_2.

Example 2.1.1. According to our syntax, the following are all valid expressions: the integer 84, the record $\mathsf{p} = \{\mathsf{age} = 84, \mathsf{married} = \mathsf{false}\}$, the abstraction $\mathsf{isNewborn} = (\lambda x \to \mathsf{isZero}\, x.\mathsf{age})$ or the the application $\mathsf{isNewborn}\, \mathsf{p}$, i.e., $(\lambda x \to \mathsf{isZero}\, x.\mathsf{age})\, \{\mathsf{age} = 84, \mathsf{married} = \mathsf{false}\}$. But what do these expressions mean, or, in other words, what is the value they reduce to? For the first three, reduction makes no sense, since they are already values (appropriately, no rule applies to them). The last expression is more complex. To uncover its meaning, we follow a series of reduction steps that culminate in a value, as follows:

$$\mathsf{isNewborn}\, \mathsf{p} \hookrightarrow \mathsf{isZero}\, \mathsf{p.age} \hookrightarrow \mathsf{isZero}\, 84 \hookrightarrow \mathsf{false}$$

Each step is, of course, established by a derivation. We give as an example one for the second step.

$$\dfrac{\dfrac{84, \mathsf{false\ are\ values} \qquad \mathsf{age} \in \{\mathsf{age}, \mathsf{married}\}}{\mathsf{p.age} \hookrightarrow 84}\ \text{R-ACC2}}{\mathsf{isZero}\, \mathsf{p.age} \hookrightarrow \mathsf{isZero}\, 84}\ \text{R-ISZERO}$$

Meaningless expressions

The expressions in our previous example are instances of meaningful expressions, i.e., expressions that result in successful computations. But what about strange expressions like isZero true and if $(\lambda x \to x.\mathsf{age})$ then 1 else 0? Even though we cannot reduce them, we do not consider them values either: they are *stuck*. Other expressions, like $\mathsf{isNewborn}\, \mathsf{p}'$ where $\mathsf{p}' = \{\mathsf{age} = \mathsf{false}, \mathsf{married} = 84\}$, do reduce some steps, but eventually get stuck as well. While syntactically

correct, these expressions are semantically invalid, i.e., meaningless. In an implementation of this language, they would trigger a run-time error signaling an unsuccessful computation. How can we prevent such expressions from being written in the first place? Can they be detected statically, i.e., without trying evaluate them beforehand? These are some of the questions that *type systems* are intended to answer.

2.2 A simple functional type system

How do we know if expressions can be evaluated? For abstractions, integers and booleans this is immediate—they are values. For other compound expressions, evaluation depends on their sub-expressions. We know that function applications $(e_1\, e_2)$ evaluate if e_1 evaluates to an abstraction $(\lambda x \to e_0)$, if e_2 evaluates to some value v, and if e_0 with v substituted for all occurrences of x also evaluates. Conditional expressions $(\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3)$ evaluate if either e_1 evaluates to true and e_2 also evaluates, or if e_1 evaluates to false and e_3 also evaluates. $\mathsf{isZero}\ e$ expressions evaluate to either true or false if e evaluates to an integer. Record expressions $\{\ell = e_\ell\}_{\ell \in L}$ evaluate if each e_k for $k \in L$ also evaluates (if they are already values, then $\{\ell = e_\ell\}_{\ell \in L}$ is also a value). Finally, field access expressions $(e.\ell)$ evaluate if e evaluates to a record containing a field with label ℓ.

Clearly, some expressions (namely applications, conditionals, zero testing and field access) have constraints on the sort of values their constituents should reduce to. Others (namely integers, booleans, abstractions, zero testing and record creation) clearly let us know what sort of values to expect from their evaluation. What if we could use this information to attribute an interface to expressions, a property that approximates the result of their evaluation? This would allow us to know if expressions are well-formed, i.e., that all their sub-expressions satisfy their constraints. To this interface or property we call a *type*, and to the rules that allow us to attribute it we call a *type system*.

There are four sorts of values in our system: abstractions, integers, booleans and records. Thus we should be able to construct four sorts of types and attribute them to expressions, depending on the value they reduce to: function types $(T \to U)$ to those that evaluate to abstractions taking in values of type T and returning values of type U, an integer type (Int) to those that evaluate to integers, a boolean type (Bool) to those that evaluate to booleans, and record types $(\{\ell\colon T_\ell\}_{\ell \in L})$ to those that evaluate to records containing fields $k\colon T_k$ for each $k \in L$. Formally, our types are constructed according to the grammar in Fig. 2.3.

Types are formally attributed through a judgment of the form $\Gamma \vdash e : T$, read "expression e has type T under context Γ". A context Γ is a possibly empty mapping between variables and types, of the form $x_1 : T_1, \ldots, x_n : T_n$. Contexts keep track of which variables are bound in the scope of the expression to which we are trying to attribute a type (for example, the context for sub-expression $\mathsf{isZero}\, x$ in $\lambda x \to \mathsf{isZero}\, x$ should include $x : \mathsf{Int}$). Typing judgments are usually defined through inference rules, since they succinctly capture the recursive reasoning we outline above and, in simple cases such as this, allow us derive a straightforward typing algorithm. Much like the reduction relation in Section 2.1, we can interpret the typing judgment as a ternary

Syntax

$$\begin{array}{rl} \textbf{expressions} & e ::= \ldots \mid \lambda x : T \to e \\ \textbf{types} & T, U ::= T \to U \mid \mathsf{Int} \mid \mathsf{Bool} \mid \{\ell\colon T_\ell\}_{\ell \in L} \end{array}$$

Typing $\boxed{\Gamma \vdash e : T}$

$$\text{T-Abs}\ \frac{\Gamma, x : T \vdash e : U}{\Gamma \vdash (\lambda x : T \to e) : T \to U} \qquad \text{T-App}\ \frac{\Gamma \vdash e_1 : U \to T \qquad \Gamma \vdash e_2 : U}{\Gamma \vdash e_1\, e_2 : T} \qquad \text{T-Var}\ \frac{x : T \in \Gamma}{\Gamma \vdash x : T} \qquad \text{T-Int}\ \Gamma \vdash i : \mathsf{Int}$$

$$\text{T-True}\ \Gamma \vdash \mathsf{true} : \mathsf{Bool} \qquad \text{T-False}\ \Gamma \vdash \mathsf{false} : \mathsf{Bool} \qquad \text{T-If}\ \frac{\Gamma \vdash e_1 : \mathsf{Bool} \qquad \Gamma \vdash e_2 : T \qquad \Gamma \vdash e_3 : T}{\Gamma \vdash \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : T}$$

$$\text{T-IsZero}\ \frac{\Gamma \vdash e : \mathsf{Int}}{\Gamma \vdash \mathsf{isZero}\, e : \mathsf{Bool}} \qquad \text{T-Rcd}\ \frac{\Gamma \vdash e_k : T_k \quad (\forall k \in L)}{\Gamma \vdash \{\ell = e_\ell\}_{\ell \in L} : \{\ell\colon T_\ell\}_{\ell \in L}} \qquad \text{T-Acc}\ \frac{\Gamma \vdash e : \{\ell\colon T_\ell\}_{\ell \in L} \qquad k \in L}{\Gamma \vdash e.k : T_k}$$

Figure 2.3: Type system for the simple functional language

relation (a pair of triples) between contexts, expressions and types, inductively defined by the rules in Fig. 2.3.

One thing to note is that we need to modify the original language of expressions: a type annotation is needed in abstractions, since otherwise we would not know how to validate the use of bound variables within the expressions they enclose (in practical languages, sophisticated techniques like type inference allow programmers to omit these annotations).

Example 2.2.1. We show above that expression isNewborn p, short for

$$(\lambda x \to \mathsf{isZero}\, x.\mathsf{age})\, \{\mathsf{age} = 84, \mathsf{married} = \mathsf{false}\},$$

reduces to the boolean value false. Let Person stand for type $\{\mathsf{age}\colon \mathsf{Int}, \mathsf{married}\colon \mathsf{Bool}\}$. Using our typing rules, and adding a type annotation to the abstraction isNewborn, writing it as

$$\lambda x : \mathsf{Person} \to \mathsf{isZero}\, x.\mathsf{age},$$

we can easily infer that this expression has type Bool. The reasoning is illustrated by the following derivation:

$$\dfrac{\dfrac{\dfrac{\mathcal{D}}{\vdash \mathsf{isNewborn} : \mathsf{Person} \to \mathsf{Bool}}\ \text{T-Abs} \qquad \dfrac{\text{T-Int}\ \vdash 84 : \mathsf{Int} \qquad \text{T-Bool}\ \vdash \mathsf{false} : \mathsf{Bool}}{\vdash \mathsf{p} : \mathsf{Person}}\ \text{T-Rcd}}{\vdash \mathsf{isNewborn}\, \mathsf{r} : \mathsf{Bool}}\ \text{T-App}$$

where $\mathcal{D}$ stands for the following sub-derivation:

$$\dfrac{\dfrac{\dfrac{x : \mathsf{Person} \in x : \mathsf{Person}}{x : \mathsf{Person} \vdash x : \mathsf{Person}}\,\text{T-VAR} \qquad \mathsf{age} \in \{\mathsf{age}, \mathsf{married}\}}{x : \mathsf{Person} \vdash x.\mathsf{age} : \mathsf{Int}}\,\text{T-ACC}}{x : \mathsf{Person} \vdash \mathsf{isZero}\, x.\mathsf{age} : \mathsf{Bool}}\,\text{T-ISZERO}$$

Example 2.2.2. We also note above that expression $\mathsf{isNewborn}\ \mathsf{p}'$, short for

$$(\lambda x : \mathsf{Person} \to \mathsf{isZero}\, x.\mathsf{age})\, \{\mathsf{age} = \mathsf{false}, \mathsf{married} = 84\},$$

cannot be evaluated. Accordingly, our type system does not allow us to derive a type for it. The reason for this lies in the mismatch between the type expected by f and the type of r' when applying the T-APP rule, the only typing rule that matches the expression:

$$\dfrac{\dfrac{\vdots}{\vdash \mathsf{isNewborn} : \mathsf{Person} \to \mathsf{Bool}}\,\text{T-ABS} \qquad \dfrac{\vdots}{\vdash \mathsf{p}' : \{\mathsf{age}\text{: }\mathsf{Bool}, \mathsf{married}\text{: }\mathsf{Int}\}}\,\text{T-RCD}}{\vdash \mathsf{isNewborn}\ \mathsf{p}' : ??}\,\text{T-APP}\ \text{✗}$$

The invalid expression in this example is not the only one for which we cannot derive a type. In fact, we can easily show that our type system does not allow us to derive a type for *any* expression that cannot evaluated. This important property is known as *type safety*, and is what gives our type system the power to reject invalid expressions statically, i.e., without trying to evaluate them. Most type systems in practical use nowadays, whether in functional or imperative languages, offer type safety to some extent.

Note 2.2.1. The history of types and type systems can be traced back to the efforts of early 20th century mathematicians to resolve Russell's paradox ("does the set of all sets that do not contain themselves contain itself?"), which threatened the role of set theory as a foundation of mathematics. Later, types were incorporated in the λ-calculus by Church as a response to the Kleene-Rosser paradox, which showed the untyped system to be logically inconsistent.

2.3 Subtyping functional types

Our notion of type safety gives our type system a sort of *soundness* with respect to evaluation: if we are able to derive the type of an expression, we can be sure its evaluation does not get stuck. But in our quest for safety we have made our system too rigid, and the complementary property, *completeness*, is not ensured: we cannot attribute a type to *every* expression that evaluates. A simple example suffices to show this.

Example 2.3.1. Suppose we wish to remove the married field from the input type annotation in isNewborn, writing it as:

$$\lambda x : \{\mathsf{age}\text{: }\mathsf{Int}\} \to \mathsf{isZero}\, x.\mathsf{age}$$

Observe that we can still successfully evaluate expression isNewborn p:

$$\mathsf{f\ r} \hookrightarrow \mathsf{isZero\ r.age} \hookrightarrow \mathsf{isZero\ 84} \hookrightarrow \mathsf{false}$$

Notice, however, that we cannot derive a type for it, since the type of p, $\{\mathsf{age}\colon \mathsf{Int}, \mathsf{married}\colon \mathsf{Bool}\}$, does not have exactly the same fields as $\{\mathsf{age}\colon \mathsf{Int}\}$, the type accepted by abstraction isNewborn:

$$\dfrac{\dfrac{\vdots}{\vdash \mathsf{isNewborn} : \{\mathsf{age}\colon \mathsf{Int}\} \to \mathsf{Bool}}\ \text{T-Abs} \qquad \dfrac{\vdots}{\vdash \mathsf{p} : \{\mathsf{age}\colon \mathsf{Int}, \mathsf{married}\colon \mathsf{Bool}\}}\ \text{T-Rcd}}{\vdash \mathsf{isNewborn\ p} : ??}\ \text{T-App}\ \text{✗}$$

It is plain to see that this kind of mismatch poses no danger to type safety: a function expecting a record with less fields than the one we give it can only access fields that this record contains, provided the type of these fields match. It is as if expressions of a certain type (e.g. $\{\mathsf{age}\colon \mathsf{Int}, \mathsf{married}\colon \mathsf{Bool}\}$) can safely take on simpler types (e.g. $\{\mathsf{age}\colon \mathsf{Int}\}$, $\{\mathsf{married}\colon \mathsf{Bool}\}$, $\{\}$) if their context so requires. This is the reasoning behind *subtyping*.

Subtyping in general is often justified by appealing to *Liskov's principle of safe substitution*: a type T is said to be a subtype of U if a value of type T can be used wherever one of type U is expected without violating any of the desirable properties (e.g. type safety) of type system in question. We can formalize this notion in our type system by including an additional typing rule, known as *subsumption*, that accounts for this principle:

$$\text{T-Sub}\ \dfrac{\Gamma \vdash e : U \qquad U \leq T}{\Gamma \vdash e : T}$$

This rule allows us to derive a type T for an expression e if we can derive type U for it and if U is a subtype of T. The notion that a type is subtype of another is formalized through a binary relation on types, denoted here by $\leq$. The definition of this relation, which must preserve the desirable properties of the type system, becomes the central question whenever we wish to formalize subtyping for a programming language.

Two properties that this relation should satisfy are immediately apparent from the principle of safe substitution: reflexivity (any type is a subtype of itself) and transitivity (if type T is a subtype of U and U is a subtype of V, then T is a subtype of V). In other words, a subtyping relation should be a *preorder* on types.

Inference rules are also a useful device to define a subtyping relation. Thus, we define ours inductively by the rules in Fig. 2.4. We include a rule for each sort of type: two reflexive rules for types Int and Bool, a rule for records and a rule for functions. The reflexive rules simply state that atomic types are subtypes of themselves. The rule for records follows the reasoning outlined above: a record type is subtype of another record type if it contains a subset of the labels of the other, and each corresponding field is also in the subtyping relation. Thus we can easily derive the desired result of $\{\mathsf{age}\colon \mathsf{Int}, \mathsf{married}\colon \mathsf{Bool}\} \leq \{\mathsf{age}\colon \mathsf{Int}\}$ for our motivating example.

$$\text{S-INT}\quad \mathsf{Int} \leq \mathsf{Int} \qquad \text{S-BOOL}\quad \mathsf{Bool} \leq \mathsf{Bool}$$

$$\text{S-RCD}\quad \frac{K \subseteq L \qquad T_k \leq U_k \quad (\forall k \in K)}{\{\ell\colon T_\ell\}_{\ell \in L} \leq \{k\colon U_k\}_{k \in K}} \qquad \text{S-ARROW}\quad \frac{U_1 \leq T_1 \qquad T_2 \leq T_2}{T_1 \to T_2 \leq U_1 \to U_2}$$

Figure 2.4: Subtyping rules for the simple functional language

Note 2.3.1. Whenever the subtyping relation allows a type constructor to vary in the number of fields, we can say we are in the presence of *width subtyping*.

Note 2.3.2. Every record type is a subtype of $\{\}$, the type of records with no fields.

Subtyping for function types is somewhat less intuitive. Consider once again abstraction $(\lambda x : \{\mathsf{age}\colon \mathsf{Int}\} \to \mathsf{isZero}\, x.\mathsf{age})$. The immediate type for this expression is, as the derivation in Example 2.3.1 shows, $\{\mathsf{age}\colon \mathsf{Int}\} \to \mathsf{Bool}$. But what other types should we allow it to take on? Since we know that $\{\mathsf{age}\colon \mathsf{Int}\} \leq \{\}$, should we let it take on $\{\} \to \mathsf{Bool}$, i.e., allow $\{\mathsf{age}\colon \mathsf{Int}\} \to \mathsf{Bool} \leq \{\} \to \mathsf{Bool}$? Evidently, this cannot happen: if we allow this function to receive an empty record, it will surely access a field it does not contain. What about $\{\mathsf{age}\colon \mathsf{Int}\} \to \mathsf{Bool} \leq \{\mathsf{age}\colon \mathsf{Int}, \mathsf{married}\colon \mathsf{Bool}\} \to \mathsf{Bool}$? This is counterintuitive at first glance, but is readily seen to be safe: supplying a record with more fields than required simply results in these fields being ignored. These examples demonstrate the reasoning for the first premise of the subtyping rule for function types: it is safe for a function type to be considered a subtype of another if their respective argument types are also in the subtyping relation, but in the reverse direction.

And what about the return types? How are they related? Suppose a function with type $\mathsf{Bool} \to \{\mathsf{age}\colon \mathsf{Int}\}$. Is it safe to use it where a function with type $\mathsf{Bool} \to \{\mathsf{age}\colon \mathsf{Int}, \mathsf{married}\colon \mathsf{Bool}\}$ is expected? The answer is no, since the context may try to access the married field of the record returned by the function. But using it in a context where $\mathsf{Bool} \to \{\}$ is expected is fine, since it contains all the fields known by the context (in this case, none). Thus we can see why the direction of the return types is preserved in the second premise of the rule for functions.

Note 2.3.3. Some terminology is useful to succinctly describe the influence type constructors have on the direction of the subtyping relation between their fields. When the direction of the relation is maintained, they are said to be *covariant*. When the direction is reversed, they are said to be *contravariant*. More uncommonly, when subtyping is not allowed, they are said to be *invariant*. Thus we can describe the function type constructor as being contravariant on the argument type and covariant on the return type.

Example 2.3.2. Consider expression isNewborn p from Example 2.3.1, short for

$$(\lambda x : \{\mathsf{age}\colon \mathsf{Int}\} \to \mathsf{isZero}\, x.\mathsf{age})\, \{\mathsf{age} = 84, \mathsf{married} = \mathsf{false}\}.$$

The following typing derivation shows how rule T-SUB now allows us to give it type Bool.

$$\dfrac{\dfrac{\dfrac{\vdots}{\vdash \mathsf{isNewborn} : \{\mathsf{age}\colon \mathsf{Int}\} \to \mathsf{Bool}}\,\text{T-Abs} \qquad \mathcal{D}}{\vdash \mathsf{isNewborn} : \mathsf{Person} \to \mathsf{Bool}}\,\text{T-Sub} \qquad \dfrac{\vdots}{\vdash \mathsf{p} : \mathsf{Person}}\,\text{T-Rcd}}{\vdash \mathsf{isNewborn}\ \mathsf{p} : \mathsf{Bool}}\,\text{T-App}$$

Where $\mathcal{D}$ stands for the following sub-derivation:

$$\dfrac{\dfrac{\{\mathsf{age}\} \subseteq \{\mathsf{age}, \mathsf{married}\} \qquad \text{S-Int}\ \ \mathsf{Int} \leq \mathsf{Int}}{\mathsf{Person} \leq \{\mathsf{age}\colon \mathsf{Int}\}}\,\text{S-Rcd} \qquad \text{S-Bool}\ \ \mathsf{Bool} \leq \mathsf{Bool}}{\{\mathsf{age}\colon \mathsf{Int}\} \to \mathsf{Bool} \leq \mathsf{Person} \to \mathsf{Bool}}\,\text{S-Arrow}$$

Although we do not show it, we could easily demonstrate that the inclusion of the subsumption rule in our system does not allow it to give a type to terms that do not evaluate. As such, we have increased the flexibility of our system (i.e., the set of valid, meaningful expressions it is able to type) without compromising its safety.

Note 2.3.4. Type safety is not the only property offered by type systems, nor is it the only one that subtyping should preserve (in fact, many type systems only offer it to some extent). As stated in the introduction, session type systems offer some guarantees related to communication; these are examples of properties that should also be preserved by subtyping.

Note 2.3.5. While subtyping is meant to combat the *incompleteness* of a type system with respect to evaluation, it is typically not enough to eliminate it altogether. In our case, an expression like $(\mathsf{if}\ \mathsf{true}\ \mathsf{then}\ 84\ \mathsf{else}\ \mathsf{true})$ can be fully evaluated to 84, but remains untypable even with subtyping.

How does subtyping work in practice? How can we incorporate it in a programming language compiler? In a simple type system such as the one explored in this section, a subtyping algorithm can be obtained directly from reading the subtyping rules upward, building a derivation rooted at the subtyping statement we want to check. If we want to check that $T_1 \to U_1 \leq T_2 \to U_2$, we recursively check that $T_2 \leq T_1$ and $U_1 \leq U_2$, and so on. This procedure is guaranteed to terminate because each recursive call we check successively "smaller" types (culminating in atomic types like Int or Bool). Furthermore, there is no ambiguity in which rule to apply, because the subtyping rules are *syntax directed*: given any statement $T \leq U$, there is at most one rule for which this statement matches the conclusion.

Note 2.3.6. An algorithm for the original typing relation $\Gamma \vdash e : T$ (without subtyping) can also be obtained directly from the rules in Fig. 2.3. However, this is not the case if we include the subsumption rule to account for subtyping, because this rule makes the system not syntax directed: since it matches any context, expression and type, we can apply T-Sub instead of any other rule at any point in a derivation. Still, this rule is useful to explain subtyping at a purely theoretical level. Pierce [84] shows how typing rules can be rewritten to accommodate subtyping without T-Sub.

Syntax (continued)

$$\textbf{expressions} \quad e ::= \ldots \mid \ell\, e \text{ as } T \mid \mathsf{case}\ e\ \mathsf{of}\ \{\ell\, x_\ell \to e_\ell\}_{\ell \in L}$$

Reduction (continued) $\boxed{e \hookrightarrow e}$

$$\text{R-TAG}\ \frac{e \hookrightarrow e'}{\ell\, e \text{ as } T \hookrightarrow \ell\, e' \text{ as } T} \qquad \text{R-CASE1}\ \frac{e \hookrightarrow e'}{\mathsf{case}\ e\ \mathsf{of}\ \{\ell x_\ell \to e_\ell\} \hookrightarrow \mathsf{case}\ e'\ \mathsf{of}\ \{\ell x_\ell \to e_\ell\}}$$

$$\text{R-CASE2}\ \mathsf{case}\ (k\, e \text{ as } T)\ \mathsf{of}\ \{\ell\, x_\ell \to e_\ell\} \hookrightarrow [e/x_k]e_k$$

Typing (continued) $\boxed{\Gamma \vdash e : T}$

$$\text{T-TAG}\ \frac{k \in L \qquad \Gamma \vdash e : T_k}{\Gamma \vdash k\, e \text{ as } \langle \ell\colon T_\ell \rangle_{\ell \in L} : \langle \ell\colon T_\ell \rangle_{\ell \in L}} \qquad \text{T-CASE}\ \frac{\Gamma \vdash e : \langle \ell\colon T_\ell \rangle_{\ell \in L} \qquad \Gamma, x_k : T_k \vdash e_k : T \quad (\forall k \in L)}{\Gamma \vdash \mathsf{case}\ e\ \mathsf{of}\ \{\ell\, x_\ell \to e_\ell\}_{\ell \in L} : T}$$

Figure 2.5: Extending the simple functional language with variant types.

A simple functional language with just functions, records and basic types is enough to set up and resolve the conflict between safety and flexibility from which subtyping arises. But from the point of view of structured data, this language and its type system are not very interesting: for example, they do not allow us to straightforwardly describe and use common data structures like lists or trees. In the rest of this section we briefly introduce some constructs that will allow us to do this, and explain how subtyping can be applied to them as well.

Note 2.3.7. Subtyping is a standard feature of many type systems, and the literature on the topic is vast [7, 13, 18, 19, 26, 33, 63]. Its conventional interpretation, based on the notion of substitutability, originates from the work of Liskov [65].

2.4 Variant types

Variants or *tagged unions* roughly correspond to *datatypes* in ML-like languages or *enums* in Rust. They can be succintly described by contrast with records: while records allow us to group together multiple pieces of data with possibly different types, variants allow us to say that a piece of data belongs to one of several possible kinds of data. Each piece is tagged with a *label* ℓ and the variant type T it belongs to ($\ell\, e$ as T), so that it can later be analyzed and dealt with accordingly (using a case analysis construct of the form $\mathsf{case}\ e\ \mathsf{of}\ \{\ell \to e\}$). The modifications made to accommodate variants in the language and type system are shown in Fig. 2.5

Example 2.4.1. Assume we introduce floating-point numbers (expressions like 1.23 of type Float) and arithmetic operators in our language, and suppose we want to use them to represent geometric figures on a plane. We represent circles by values of type $\mathsf{Circle} = \{\mathsf{x}\colon \mathsf{Float}, \mathsf{y}\colon \mathsf{Float}, \mathsf{r}\colon \mathsf{Float}\}$, squares by values of type $\mathsf{Square} = \{\mathsf{x}\colon \mathsf{Float}, \mathsf{y}\colon \mathsf{Float}, \mathsf{s}\colon \mathsf{Float}\}$, etc. We know that these types

have something in common—they all represent a geometric figure—but as it stands, our type system is not capable of expressing this commonality. Observe that we cannot write a well-typed function area that takes the area of an arbitrary shape: it would need to have, simultaneously, types $\mathsf{Circle} \to \mathsf{Float}$ and $\mathsf{Square} \to \mathsf{Float}$, etc. The best we can do is write a specialized function for each kind of shape (areaCirc, areaSqr, etc.), which is quite cumbersome:

$$\mathsf{areaCirc} = \lambda x : \mathsf{Circle} \to \pi \times x.\mathsf{rad\hat{}2}$$

$$\mathsf{areaSqr} = \lambda x : \mathsf{Square} \to x.\mathsf{len\hat{}2}$$

Variants provide exactly what we need to group together these types in a single type Fig and write a function area that will work on any of them:

$$\begin{aligned} & & \mathsf{area} = \lambda x : \mathsf{Fig} \to \mathsf{case}\ x\ \mathsf{of} \\ \mathsf{Fig} = & \langle \mathsf{Circ}\colon \mathsf{Circle} & \{\mathsf{Circ}\ c \to \pi \times c.\mathsf{rad\hat{}2}, \\ & , \mathsf{Sqr}\colon \mathsf{Square} \rangle & , \mathsf{Sqr}\ s \to s.\mathsf{len\hat{}2}\} \end{aligned}$$

Of course, we now need to appropriately tag the records representing circles or squares. Whereas without variants we would simply write $\mathsf{uc} = \{\mathsf{x} = 0, \mathsf{y} = 0, \mathsf{rad} = 1\}$ to represent the unit circle, we now need to write $\mathsf{uc} = (\mathsf{Circ}\,\{\mathsf{x} = 0, \mathsf{y} = 0, \mathsf{rad} = 1\}\ \mathsf{as}\ \mathsf{Fig})$. The Circ tag is necessary to let case expressions know which branch to follow, while the Fig tag is necessary to ensure that case expressions support every possible branch in a given variant type.

As an example of how case expressions reduce, consider the following evaluation:

$$\begin{aligned} & \mathsf{area\ uc} \\ & \hookrightarrow \mathsf{case\ uc\ of}\ \{\mathsf{Circle}\ c \to \pi \times c.\mathsf{rad\hat{}2}, \mathsf{Square}\ s \to s.\mathsf{len\hat{}2}\} \\ & \hookrightarrow \pi \times \{\mathsf{x} = 0.0, \mathsf{y} = 0.0, \mathsf{rad} = 1.0\}.\mathsf{rad\hat{}2} \\ & \hookrightarrow \pi \times 1\hat{}2 \hookrightarrow \pi \times 1 \hookrightarrow \pi \end{aligned}$$

And observe how we can derive a type for expression area uc by the following derivation:

$$\frac{\mathcal{D}_1 \qquad \mathcal{D}_2}{\vdash \mathsf{area\ uc} : \mathsf{Float}}\ \text{T-App}$$

Where $\mathcal{D}_1$ stands for the following sub-derivation:

$$\frac{\dfrac{\dfrac{\vdots}{x : \mathsf{Fig}, c : \mathsf{Circle} \vdash \pi \times c.\mathsf{rad\hat{}2} : \mathsf{Float}} \quad \dfrac{\vdots}{x : \mathsf{Fig}, s : \mathsf{Square} \vdash s.\mathsf{len\hat{}2} : \mathsf{Float}}}{x : \mathsf{Fig} \vdash \mathsf{case}\ x\ \mathsf{of}\ \{\mathsf{Circ}\ c \to \pi \times c.\mathsf{rad\hat{}2}, \mathsf{Sqr}\ s \to s.\mathsf{len\hat{}2}\} : \mathsf{Float}}\ \text{T-Case}}{\vdash \mathsf{area} : \mathsf{Fig} \to \mathsf{Float}}\ \text{T-Abs}$$

And $\mathcal{D}_2$ for the following sub-derivation:

$$\frac{\mathsf{Circ} \in \{\mathsf{Circ}, \mathsf{Sqr}\} \qquad \dfrac{\vdots}{\vdash \{\mathsf{x} = 0.0, \mathsf{y} = 0.0, \mathsf{rad} = 1.0\} : \mathsf{Circle}}}{\vdash \mathsf{uc} : \mathsf{Fig}}\ \text{T-Tag}$$

Notice, however, that our type system is once again too rigid, since it does not let us type some expressions envolving variants. Take, for example, expression $\mathsf{area}\,\mathsf{uc}'$ where uc' the unit circle tagged with a variant type with a single branch, i.e.:

$$\mathsf{uc}' = \mathsf{Circ}\,\{\mathsf{x} = 0.0, \mathsf{y} = 0.0, \mathsf{rad} = 1.0\}\ \mathsf{as}\ \langle \mathsf{Circ}\colon \mathsf{Circle}\rangle$$

Although it can be easily shown that this expression evaluates to π, the type system requires the argument to function area to be of type Fig. But it is clear to see that any value with type $\langle \mathsf{Circ}\colon \mathsf{Circle}\rangle$ is safe to use where one of type Fig is required, since any case expression that analyzes a value of type Fig will be prepared to analyze one of type $\langle \mathsf{Circ}\colon \mathsf{Circle}\rangle$. More generally, a value of any variant type $\langle \ell\colon T_\ell\rangle_{\ell\in L}$ can be used in place of a value of type $\langle k\colon U_k\rangle_{k\in K}$ if L is a subset of K and for each label in common, the field type in the former can also be used in place of the corresponding field type in the latter. Then, by the principle of safe substitution, we arrive at the following subtyping rule for variants:

$$\text{S-VRT}\quad \frac{L \subseteq K \qquad T_\ell \leq U_\ell \quad (\forall \ell \in L)}{\langle \ell\colon T_\ell\rangle_{\ell\in L} \leq \langle k\colon U_k\rangle_{k\in K}}$$

We introduced variants by contrasting them with records. It is also interesting to compare them with respect to subtyping: the rule for variants is nearly identical to that for records. The only difference lies in the direction of the subset relation $\subseteq$, which is reversed.

Note 2.4.1. Both records and variants exhibit width-subtyping, but they differ on how the fields can vary between subtype and supertype. We can extend the covariant/contravariant terminology introduced in Note 2.3.3 to also describe the relationship between the direction of the subset relation $\subseteq$ on the premises and the direction of the subtyping relation $\leq$ on the conclusion of subtyping rules. When the direction of $\subseteq$ is the same as $\leq$ (as in variant types), type constructors are said to be *covariant on width*, and when the direction is reversed (as in record types), types are said to be *contravariant on width*.

2.5 Recursive types

Another application of variant types is in the description of data structures of variable size, such as lists and binary trees. Consider the typical inductive definition for lists of integers: "a list of integers is either an empty list, or a pair containing an integer and a list of integers". How can we define this type? If we model the empty list as a trivial value such as the empty record $\{\}$, and a pair as a record with two fields labelled value and rest, we can define the type of lists of integers as the infinite type that satisfies the following equation:

$$\mathsf{List} = \langle \mathsf{nil}\colon \{\}, \mathsf{cons}\colon \{\mathsf{val}\colon \mathsf{Int}, \mathsf{rest}\colon \mathsf{List}\}\rangle^{2}$$

Unlike all definitions we have seen thus far, this one is recursive: it refers to the very thing it is defining. However, it is rather inconvenient to rely on an equation every time we want to write

[2]The labels nil (latin for "nothing") and cons (short for "construct") are conventional, and originate from the Lisp language.

a recursive type. In the literature, there is a more convenient notation to express recursion. This notation relies on the introduction of two constructs in the type language: type references t, u, symbols that stand for types, and the recursion operator μ, which binds the occurrences of a given type reference in a type to that type itself. With this notation, we can represent the type of lists of integers "anonymously", as follows.

$$\mu t.\langle \mathsf{nil}\colon \{\}, \mathsf{cons}\colon \{\mathsf{val}\colon \mathsf{Int}, \mathsf{rest}\colon t\}\rangle$$

Example 2.5.1. Similarly to lists, we can write the type of binary trees of integers as:

$$\mu t.\langle \mathsf{empty}\colon \{\}, \mathsf{node}\colon \{\mathsf{left}\colon t, \mathsf{val}\colon \mathsf{Int}, \mathsf{right}\colon t\}\rangle$$

Example 2.5.2. JSON a simple data exchange scheme, composed of atomic values like null, true, false, numbers and strings, as well as compound values like arrays (sequences of JSON values) and key-value maps (structures that associate strings to JSON values) [34]. Using recursion, we can write the type of an abstract representation of JSON data:

$$\begin{aligned}
&\mu t.\langle \mathsf{null}\colon \{\} \\
&\quad , \mathsf{bool}\colon \mathsf{Bool} \\
&\quad , \mathsf{number}\colon \mathsf{Float} \\
&\quad , \mathsf{string}\colon \mathsf{String} \\
&\quad , \mathsf{array}\colon \mu u.\langle \mathsf{nil}\colon \{\}, \mathsf{cons}\colon \{\mathsf{val}\colon t, \mathsf{rest}\colon u\}\rangle \\
&\quad , \mathsf{object}\colon \mu u.\langle \mathsf{nil}\colon \{\}, \mathsf{cons}\colon \{\mathsf{key}\colon \mathsf{String}, \mathsf{val}\colon t, \mathsf{rest}\colon u\}\rangle\rangle
\end{aligned}$$

Equirecursive types

While recursive types afford more expressivity for a type system, they also raise some complications. For example, if we substitute the self-references in a recursive type by the type in its entirety (also known as "unfolding" the type) we can write the type of lists in an infinite number of ways:

$$\mu t.\langle \mathsf{nil}\colon \{\}, \mathsf{cons}\colon \{\mathsf{val}\colon \mathsf{Int}, \mathsf{rest}\colon t\}\rangle$$

$$\langle \mathsf{nil}\colon \{\}, \mathsf{cons}\colon \{\mathsf{val}\colon \mathsf{Int}, \mathsf{rest}\colon \mu t.\langle \mathsf{nil}\colon \{\}, \mathsf{cons}\colon \{\mathsf{val}\colon \mathsf{Int}, \mathsf{rest}\colon t\}\}\rangle$$

$$\langle \mathsf{nil}\colon \{\}, \mathsf{cons}\colon \{\mathsf{val}\colon \mathsf{Int}, \mathsf{rest}\colon \langle \mathsf{nil}\colon \{\}, \mathsf{cons}\colon \{\mathsf{val}\colon \mathsf{Int}, \mathsf{rest}\colon \mu t.\langle \mathsf{nil}\colon \{\}, \mathsf{cons}\colon \{\mathsf{val}\colon \mathsf{Int}, \mathsf{rest}\colon t\}\}\}\rangle$$

$$\ldots$$

Clearly, all of these unfoldings represent the same infinite type, the solution to the equation $\mathsf{List} = \langle \mathsf{Nil}\colon \{\}, \mathsf{Cons}\colon \{\mathsf{val}\colon \mathsf{Int}, \mathsf{rest}\colon \mathsf{List}\}\rangle$. Thus we can consider all of them as equivalent and interchangeable—this is known as the *equirecursive interpretation*. Subtyping follows the same reasoning: we can consider type $\mathsf{InfList} = \mu t.\langle \mathsf{cons}\colon \{\mathsf{val}\colon \mathsf{Int}, \mathsf{rest}\colon t\}\rangle$ (the type of infinite lists) as a subtype of any of the three types above—all we need is to unfold it to the limit. This reasoning is formalized in the following subtyping rules.

$$\text{S-RecL}\ \frac{[\mu t.T/t]T \leq U}{\mu t.T \leq U} \qquad \text{S-RecR}\ \frac{T \leq [\mu u.U/u]U}{T \leq \mu u.U}$$

Notice, however, that these rules leave us with a cyclical, infinite subtyping derivation:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\infty}{\mathsf{InfList} \leq \mathsf{List}}}{\{\mathsf{val}\colon \mathsf{Int}, \mathsf{rest}\colon \mathsf{InfList}\} \leq \{\mathsf{val}\colon \mathsf{Int}, \mathsf{rest}\colon \mathsf{List}\}}\,\text{S-RCD}}{\langle \mathsf{cons}\colon \{\mathsf{val}\colon \mathsf{Int}, \mathsf{rest}\colon \mathsf{InfList}\}\rangle \leq \langle \mathsf{nil}\colon \{\}, \mathsf{cons}\colon \{\mathsf{val}\colon \mathsf{Int}, \mathsf{rest}\colon \mathsf{List}\}\rangle}\,\text{S-VRT}}{\langle \mathsf{cons}\colon \{\mathsf{val}\colon \mathsf{Int}, \mathsf{rest}\colon \mathsf{InfList}\}\rangle \leq \mathsf{List}}\,\text{S-RECR}}{\mathsf{InfList} \leq \mathsf{List}}\,\text{S-RECL}$$

To accept such infinite derivations, we must adopt a *coinductive* (rather than *inductive*) interpretation of the subtyping rules. This is achieved by reading the rules "backwards": each pair of types in the subtyping relation $\leq$ must match the conclusion of a rule in such a way that the premises are also satisfied by $\leq$. In this reading, we try to see just how far we can go without reaching a contradiction—thus a derivation is valid if it stops at an axiom or if it extends infinitely. Contrast this with its original inductive interpretation: if the premises of a rule are satisfied by $\leq$, then so is its conclusion. In this reading, the conclusion must follow from base cases in a finite number of steps.

Note 2.5.1. The difference between induction and coinduction can be intuitively understood through a comparison to the legal principles of the presumption of guilt and innocence: induction corresponds to "guilty until proven innocent", while coinduction corresponds to "innocent until proven guilty". A deeper understanding would require a lengthy detour through the theory of monotone functions and fixed points. These matters are explored at length by Sangiorgi [88].

The difference between the inductive and coinductive definitions of the subtyping relation is even clearer when we compare how they can be iteratively constructed. The inductive construction is a growing sequence of sets: starting with the empty set, we add all pairs matching a rule without premises of the form $T \leq T$, and then repeatedly add every pair matching a rule with its premises satisfied by the last set. The coinductive construction, in contrast, is a decreasing sequence of sets: starting from the set of all pairs of types, we repeatedly remove all pairs that match the conclusion of a rule for which at least one of the premises is not satisfied by the pairs in the set.

Note 2.5.2. To ensure that the equations introduced by equirecursive types have a unique solution, types must be what is called *contractive*: every self-reference must appear under a type constructor. In other words, types may not contain subterms of the form $\mu x.\mu x_1.\ \ldots\ \mu x_n.x$ [84].

While accepting infinite derivations is fine in theory, it places a burden on practical implementations: to support equirecursive types, a type checker must somehow be able to represent infinite types and derivations finitely. These issues, along with possible solution, are discussed by Pierce [84] The algorithm we propose in Chapter 6, however, is also capable of handling equirecursive subtyping for functional types.

Note 2.5.3. The algorithmic properties of subtyping for recursive types under an equirecursive interpretation were first explored by Amadio and Cardelli [8].

Isorecursive types

Alternatively, we can view each of these unfoldings not as equivalent, but isomorphic, requiring expressions of one type to be converted to its unfolding, and vice-versa—this is known as the *isorecursive interpretation*. While this allows us to stay in the conventional inductive interpretation, it requires that we include explicit "fold" and "unfold" operations in the expression language, to convert each expression the correct type, as needed. While this leads to simpler subtyping algorithms (there is no need to deal with infinite structures), conversion between unfoldings introduces a burden on the theoretical side, leading to complex inductive proofs for important properties of these algorithms.

The literature on session types tends to follow the equirecursive approach, and this work is no exception. As such, all instances of recursive types in the following chapters are interpreted equirecursively, and the subtyping relations (and algorithm) we propose in Chapter 6 is designed to follow this equirecursive interpretation as well.

Chapter 3

Regular session types

The previous chapter shows how type systems with support for structured data types like records and variants are an accessible and lightweight tool to enforce safe, disciplined data processing by statically detecting and rejecting invalid data operations. It also shows how subtyping makes type systems more flexible and usable without compromising their safety guarantees.

However, as stated in Chapter 1, programs nowadays do more than process data—they exchange it, and these exchanges usually follow structured patterns of communication, i.e., protocols. Can type systems also enforce safe and disciplined data exchange? And can they do this while remaining flexible and usable? These are some of the questions we explore in this chapter.

3.1 Communication and concurrency

In a software system, programs may exchange data at multiple levels: at the macro-level, the system may be composed of several geographically distributed machines communicating over a network; at the micro-level, a single program running on one of these machines may be organized as multiple threads of execution working interleavedly or in parallel, using different resources, and exchanging data for processing or synchronization purposes. It is useful, then, to think about these concepts in more general terms: any form of communication presupposes some form of *concurrency*, i.e., the organization of the system in multiple *processes* acting independently yet cohesively. [1]

How do processes communicate? Concurrent systems typically follow one of two common approaches: *shared memory* and *message-passing*. In the shared memory approach, processes read and write data directly to an area of memory or data structure they all have access to. While efficient, this approach is not very safe: since processes act independently, the order and timing of their memory accesses cannot generally be predicted, leading to unpredictable behavior and memory states (also known as *race conditions*) and, consequently, to synchronization problems.

Over the years, many solutions have been proposed to make shared memory concurrency safer,

[1] Concurrency should not be confused with paralellism. The former is concerned with the organization of computation as a multitude of processes that act independently yet cohesively, but not necessarily simultaneously (processes can be interleaved on the same processor). Simultaneous action between independent processes is the concern of parallelism, which naturally presupposes some form of concurrency.

mainly in the form of *synchronization primitives*, i.e., constructs that programmers can use to enforce disciplined access to shared resources. Examples of such primitives include *locks* [31] and *semaphores* [32]. Despite being widely used, these solutions can be cumbersome to implement, and their effectiveness relies on their correct application by programmers (who tend to be unreliable).

An alternative way for processes to communicate is by exchanging messages between themselves through private communication channels—the message-passing approach. While the management of channels and messages incurs some overhead, this approach is arguably safer: the exchanged data is only known by the sending and receiving processes, who use it to update their own private state, thus avoiding some race conditions and the associated synchronization problems. As a bonus, this approach is also more practical, since it generally spares programmers from the intricacies of implementing synchronization primitives.

3.2 From channel types to session types

Despite the relative simplicity of channels, their effective use also requires some discipline on the part of programmers. For instance, They should ensure that processes only send messages that receivers are equipped to handle (e.g. if the receiver expects an integer, the sender should not send a boolean).

This seems like a case where type systems would be useful—as we have seen, they are valuable tools to enforce the disciplined use of software components, and channels are no exception. The simplest way to ensure that messages contents always have the expected type is to restrict them to a single type, making channels *homogeneous*. Thus we attribute type $\mathsf{Channel\ Int}$ to integer-only channels, $\mathsf{Channel\ Bool}$ to boolean-only channels, etc. We can then use these types to ensure that sending and receiving operations on channels match the types of the messages they are restricted to carry. Extending the functional language of Chapter 2, we could use channel types to ensure that expressions $(\mathsf{send}\,84\,c)$ or $(\mathsf{isZero}\,(\mathsf{receive}\,c))$ are only considered valid if c is a channel of type $\mathsf{Channel\ Int}$.

While this restriction provides some form of type safety to message-passing programs, it still leaves a margin for synchronization problems. For example, two communicating processes may try to send a message at the same time, causing a communication error. Alternatively, they both may try to receive a message from each other, resulting in a *deadlock* (a situation where multiple processes depend on each other to advance, but none can do so). A simple solution, then, is to make channels *unidirectional*, ensuring messages flow only in one direction. To do this, we split a channel in two endpoints: the *sending* endpoint and the *receiving* endpoint, and let types control the sending and receiving capabilities of each, attributing $\mathsf{Receive\ Int}$ to the receiving ends of integer channels, $\mathsf{Send\ Bool}$ to the sending ends of boolean channels, etc. This way, the type system can reject operations like $(\mathsf{send}\,5\,r)$ where r has type $\mathsf{Receive\ Int}$, since according to its type, r can only receive data.

Besides being safer, unidirectional channels also allow us to introduce subtyping for endpoint

types. Consider a channel end of type Receive $\{\mathsf{age}\colon \mathsf{Int}, \mathsf{married}\colon \mathsf{Bool}\}$. Any process with access to this channel end is prepared to receive records with fields $\mathsf{age}\colon \mathsf{Int}$ and $\mathsf{married}\colon \mathsf{Bool}$, but it should face no problems if it receives a record of type $\{\mathsf{age}\colon \mathsf{Int}, \mathsf{married}\colon \mathsf{Bool}, \mathsf{gpa}\colon \mathsf{Float}\}$, since the gpa field will simply be ignored. It should not, however, be able receive a record of type $\{\mathsf{age}\colon \mathsf{Int}\}$, since it may try to access the missing married field. More generally, any process expecting a message of type U can receive a value of type T provided that T is a subtype of U. Thus, from the principle of safe substitution, it follows that we can use a receiving end for values of type T wherever one for values of type U is expected, arriving at the following covariant subtyping rule:

$$\text{S-RECEIVE}\quad \frac{T \leq U}{\mathsf{Receive}\, T \leq \mathsf{Receive}\, U}$$

What about sending endpoints? Is is safe to replace one of type $\mathsf{Send}\,\{\mathsf{age}\colon \mathsf{Int}, \mathsf{married}\colon \mathsf{Bool}\}$ with another of type $\mathsf{Send}\,\{\mathsf{age}\colon \mathsf{Int}, \mathsf{married}\colon \mathsf{Bool}, \mathsf{gpa}\colon \mathsf{Float}\}$? The answer is no, since a process expecting the former will send records without a $\mathsf{gpa}\colon \mathsf{Float}$ field, which the process on the receiving endpoint may try to access. The opposite, however, is safe: we can replace a sending endpoint of type $\mathsf{Send}\,\{\mathsf{age}\colon \mathsf{Int}, \mathsf{married}\colon \mathsf{Bool}, \mathsf{gpa}\colon \mathsf{Float}\}$ with one of type $\mathsf{Send}\,\{\mathsf{age}\colon \mathsf{Int}, \mathsf{married}\colon \mathsf{Bool}\}$, since the extra $\mathsf{gpa}\colon \mathsf{Float}$ field can be safely ignored on the receiving endpoint. We thus arrive at the following contravariant subtyping rule:

$$\text{S-SEND}\quad \frac{U \leq T}{\mathsf{Send}\, T \leq \mathsf{Send}\, U}$$

Note 3.2.1. Since plain channel with types of the form $\mathsf{Channel}\, T$ combine both sending and receiving capabilities, they can only be substituted by other channels of type $\mathsf{Channel}\, U$ with $T \leq U$ and $U \leq T$. In other words, T and U must be equivalent. The plain Channel type constructor is said to be invariant (cf. Note 2.3.3).

Note 3.2.2. The subtyping properties of channel types were first explored by Pierce and Sangiorgi [85] in the context of the π-calculus, a computational framework that takes process composition and message-passing (rather than functional application and abstraction) as its building blocks [73, 74].

3.3 Regular session types

While making channels both homogeneous and unidirectional avoids some of their indisciplined uses and the associated communication errors, it also makes them unsuitable to comfortably model many structured communication patterns where processes exchange multiple types of data back and forth.

Can type systems maintain communication safety in the presence of heterogeneous, bidirectional channels? *Session types*, introduced by Honda et al. [51, 52, 94], provide an affirmative

answer to this question. Instead of representing the static capability to either send or receive messages (such as $\mathsf{Send}\,\mathsf{Int}$ or $\mathsf{Receive}\,\mathsf{Bool}$), session types represent communication protocols (the exact order, direction and type of the messages that should be sent on a channel endpoint) allowing type systems to verify that programs really communicate as intended.

In this section, we present the classical (also known as, for reasons we explain below, *regular*) version of session types, and in the next we explore their subtyping properties. This will make clear how context-free session types, presented in the next chapter, innovate, and how their subtyping properties—the focus of our contributions—arise.

Sending and receiving

Session types represent communication protocols through the composition of individual communication actions, i.e., sending and receiving operations. This is best shown by example. The following session type S represents a simple protocol to be carried on an heterogeneous, bidirectional channel: an integer is sent, then a boolean is received. We can imagine, for instance, that a process is using an endpoint governed by this type to ask another process (a "smarter" one) whether an integer is prime:

$$\mathsf{S} = !\mathsf{Int}.?\mathsf{Bool}.\mathsf{End}$$

We use $!T$ and $?T$ to denote, respectively, the sending and receiving of a message of type T, the dot "." to denote sequential composition (first this, then that), and End to represent the end of a communication session.

There's two sides to every protocol: when one process sends, the other receives, and vice-versa. Thus the "smarter" process in this example should use a channel governed by a session type like S but with the direction of the messages reversed—what we call a *dual* of S.

$$?\mathsf{Int}.!\mathsf{Bool}.\mathsf{End}$$

Linearity

At this point the reader may wonder how this protocol is enforced. What prevents the enquiring process from sending multiple integers on a channel of type S? Or from not engaging with the other process at all? This can be achieved by using a *linear* type system, i.e., a type system where some variables (in our case, those bound to channels) are *ephemeral* and require *exactly* one use. After such a variable is used, it is considered out of scope and any further use of it is invalid. Conversely, failure to use the variable also results in an error. For example, if c is bound to a channel endpoint of type $!\mathsf{Int}.?\mathsf{Bool}.\mathsf{End}$, the linear type system would reject any process that attempts to use c twice (as in $\mathsf{send}\,84\,c; \mathsf{send}\,119\,c$), as well as any process that does not use it at all.

If c is consumed by being passed to send, how can the inquiring process receive a response from the smarter process? The solution is for send to give back the channel endpoint, but now

with type ?Bool.End. The same should happen, in turn, with receive: given an endpoint of type ?Bool.End, it should return not only the incoming boolean but also an endpoint of type End. Faced with an endpoint of this type, the inquiring process has no choice but to terminate the connection with close, which should return an unrestricted and trivial value like $\{\}$.

Note 3.3.1. Linear type systems, derived from the *linear logic* of Yves Girard [45], let us view variables as resources that allow exactly one use. Beyond message-passing communication (where the resources are channel endpoints), linear type systems are also useful to extend type safety to other language features such as low-level memory management (where the resources are pointers) and file system interfaces (where the resources are file handles).

Linear functions

Using a linear type system in a functional setting requires us to be careful about how variables are handled by functional abstraction and application.[2]

Suppose channels are introduced by a constructor ($\mathsf{channel}\, S$) that returns a pair of channel endpoints, one of session type S and another of a session type dual to S (i.e., inverting its sending and receiving actions). Consider then the following expression:

$$
\begin{aligned}
&\mathsf{let}\ (s, r) = \mathsf{channel}\ !\mathsf{Int}.\mathsf{End}\ \mathsf{in}\\
&\mathsf{let}\ f = \lambda x : \mathsf{Int} \to \mathsf{close}\,(\mathsf{send}\, x\, s)\ \mathsf{in}\\
&f\ 84; f\ 119;\\
&\mathsf{close}\,(\mathsf{receive}\, r)
\end{aligned}
$$

Clearly, the linear restrictions on variables bound to channel endpoints are respected: both s and r are use exactly once. Yet, the protocol specified during the construction of the channel (!Int.End, send a single integer and close the channel) is not respected: evaluating both $f\ 84$ and $f\ 119$ causes two integers (84 and 119) to be sent. The problem lies in the fact that s is enclosed in an abstraction that is bound to variable f, and f variable is used twice.

One solution to this problem is to also place linear restrictions on variables bound to abstractions. This will prevent us from using f twice, and ensure the protocol is followed. This is, however, too restrictive—most functions are meant to be used multiple times. A more sensible approach is to include linear functions alongside the usual (unrestricted) ones. We do this by introducing, at the level of expressions, *linear abstractions* ($\lambda x : T \xrightarrow{1} e$), and, at the level of types, the corresponding *linear function types* ($T \xrightarrow{1} U$). We then extend the linear restrictions to variables of linear function types as well, thus ensuring that linear abstractions are used exactly once. Finally, we introduce another constraint in the system: linear variables may not appear in the body of unrestricted abstractions. With these new constructs and constraints, the type system would not

[2] It also requires us to revise field access on records ($e.\ell$), since we cannot discard fields holding channel endpoints. The solution is to generalize field access to *record elimination*, $\mathsf{let}\ \{x_\ell = e_\ell\}_{\ell \in L} = e_1\ \mathsf{in}\ e_2\}$. Since this change is only felt at the level of expressions, we will not consider it further.

allow us to write the previous example. We can write a similar, well-behaved expression:

$$
\begin{array}{l}
\mathsf{let}\ (s, r) = \mathsf{channel}\ !\mathsf{Int}.\mathsf{End}\ \mathsf{in} \\
\mathsf{let}\ f = \lambda x : \mathsf{Int} \xrightarrow{1} \mathsf{close}\,(\mathsf{send}\, x\, s)\ \mathsf{in} \\
f\ 84; \\
\mathsf{close}\,(\mathsf{receive}\, r)
\end{array}
$$

Choice

Many communication protocols feature branching points where one process can choose how to proceed with communication from a set of options offered by another. Imagine, for instance, that the smarter process in our initial example is an efficient mathematical server that offers multiple services related to prime numbers: checking whether an integer is prime, checking whether two integers are coprime[3], or calculating the n^{th} prime number. Individually, and from the point of view of both client and server, each service could be represented as one of the following session types.

	client endpoint	**server endpoint**
primality check	!Int.?Bool.End	?Int.!Bool.End
coprimality check	!Int.!Int.?Bool.End	?Int.?Int.!Bool.End
n^{th} prime	!Int.?Int.End	?Int.!Int.End

But how can we express this as a single type for each participant? In order to support such branching protocols, session types provide *alternative composition* operators $\oplus$ (internal choice) and $\&$ (external choice) that allow us to group together the types corresponding to each service as a set of options, one of which must be selected by the client (using a select operation) and *all* of which must be supported by the server (using a match operation).

client endpoint	**server endpoint**
$\oplus\{$IsPrime: !Int.?Bool.End	$\&\{$IsPrime: ?Int.!Bool.End
, AreCoprimes: !Int.!Int.?Bool.End	, AreCoprimes: ?Int.?Int.!Bool.End
, NthPrime: !Int.?Int.End$\}$	, NthPrime: ?Int.!Int.End$\}$

Now, if the client wants to check if an integer is prime, it must first perform select isPrime. This operations returns a continuation endpoint of type !Int.?Bool.End, on which the client can continue communication (if the client had selected AreCoprimes instead, this endpoint would have type !Int.!Int.?Bool.End, and so on).

The server, however, has quite a different task: it must be ready to communicate according to any of the options it offers (IsPrime, AreCoprimes or NthPrime). As such, the match operation takes the form of a branching construct (not unlike the case expressions in Section 2.4) where each branch binds the continuation endpoint to a variable and performs the necessary operations

[3]Two integers are *coprimes* if their greatest common divisor is 1.

to fulfill its corresponding part of the protocol. Let s be an endpoint with the external choice type above. Then, the match operation should look something like the following.

$$\begin{aligned}\mathsf{match}\ s\ \mathsf{with}\ \{\mathsf{IsPrime}\, s &\to \mathsf{let}\ (n, s) = \mathsf{receive}\, s\ \mathsf{in}\\ &\quad\ \mathsf{let}\ s = \mathsf{send}\,(\mathsf{prime}\, n)\, s\ \mathsf{in}\\ &\quad\ \mathsf{close}\, s\\ , \mathsf{AreCoprimes}\, s &\to \ldots\\ , \mathsf{NthPrime}\, s &\to \ldots\}\end{aligned}$$

Recursion

The prime number server protocol described by the session types above is quite limited: the server can only provide one service per channel before closing the connection. This is not very sensible, since a client may wish to use multiple services, or the same service multiple times (e.g., to check multiple numbers for primality). How can we do this without establishing multiple connections between the client and the server?

Using the recursion operator introduced in Section 2.5, we can express a potentially infinite protocol where the client may request an arbitrary number of services from the server on a single connection. To do this, we can replace End by a self-reference at the end of each option. This means that, after a service is completed, the client may request a new one (conversely, after providing a service, the server must be ready to handle any of the options again). We also want to allow the client to terminate the connection; to do this, we can introduce a Finish option that simply leads to End. We are thus left with the following types for the client and server endpoints.

client endpoint	**server endpoint**
$\mu s.\, \oplus\{\mathsf{IsPrime}\colon \mathsf{!Int.?Bool}.s$	$\mu s.\, \&\{\mathsf{IsPrime}\colon \mathsf{?Int.!Bool}.s$
$\quad , \mathsf{AreCoprimes}\colon \mathsf{!Int.!Int.?Bool}.s$	$\quad , \mathsf{AreCoprimes}\colon \mathsf{?Int.?Int.!Bool}.s$
$\quad , \mathsf{NthPrime}\colon \mathsf{!Int.?Int}.s$	$\quad , \mathsf{NthPrime}\colon \mathsf{?Int.!Int}.s$
$\quad , \mathsf{Finish}\colon \mathsf{End}\}$	$\quad , \mathsf{Finish}\colon \mathsf{End}\}$

In the rest of this section, we write PrimeClient to refer to the session type on the left, and PrimeServer to refer to the one on the right.

Syntax

As this point we are ready to introduce the syntax of session types more formally. It is given by the two mutually recursive grammars in Fig. 3.1. The first grammar (T, U) defines the syntax of types in general, while the second (S, R) defines the syntax of session types exclusively (this separation is necessary to rule out nonsense types like $?\mathsf{Int}.(\mathsf{Int} \xrightarrow{1} \mathsf{Int})$).

On the side of types in general, we have type Int (which stands also for other basic types like Bool, String, etc.), unrestricted functions $T \xrightarrow{*} U$ (now annotated with $*$), linear functions $T \xrightarrow{1} U$, records $\{\ell\colon T_\ell\}_{\ell \in L}$ and variants $\langle \ell\colon T_\ell \rangle_{\ell \in L}$, general type references t, the recursion operator for types in general $\mu t.T$ and, finally, session types S.

$$
\begin{aligned}
\textbf{types} \quad & T, U ::= \mathsf{Int} \mid T \xrightarrow{m} U \mid (\!|\ell\colon T_\ell|\!)_{\ell \in L} \mid t \mid \mu t.T \mid S \\
\textbf{session types} \quad & S, R ::= \sharp T.S \mid \odot\{\ell\colon S_\ell\}_{\ell \in L} \mid \mathsf{End} \mid s \mid \mu s.S \\
\textbf{multiplicities} \quad & m, n ::= * \mid 1 \\
\textbf{records/variants} \quad & (\!|\cdot|\!) ::= \{\cdot\} \mid \langle\cdot\rangle \\
\textbf{polarities} \quad & \sharp ::= ? \mid ! \\
\textbf{choices} \quad & \odot ::= \oplus \mid \&
\end{aligned}
$$

Figure 3.1: The syntax of regular session types

Session types, in turn, can be constructed using input and continuation $?T.S$ (receive a message of type T and continue according to S), output and continuation $!T.S$ (send a message of type T and continue according to S), internal choice $\oplus\{\ell\colon S_\ell\}_{\ell \in L}$ (choose a label k in L to continue according to S_k), external choice $\&\{\ell\colon S_\ell\}_{\ell \in L}$ (offer all labels in L as a choice, handling each possible continuation S_k for all $k \in L$) and the End type (close the channel). Finally, we include session type references s and the recursion operator for session types $\mu s.S$. While this may seem redundant, it is actually necessary—including recursion only at the level of types in general would rule out types like $!\mathsf{Bool}.(\mu s.?\mathsf{Int}.s)$ (send a boolean and then receive integers forever). For convenience, we let x range over both t and s (thus $\mu x.T$ may represent a recursive functional type as well as a recursive session type).

Example 3.3.1. Our syntax is quite flexible. On the functional side, we can have channel endpoints as function arguments (e.g., $(?\mathsf{String}.\mathsf{End}) \xrightarrow{*} \mathsf{Bool}$), return values (e.g., $\mathsf{Int} \xrightarrow{1} \oplus\{\mathsf{A}\colon \mathsf{End}\}$), record fields (e.g., $\{\mathsf{a}\colon \mathsf{String}, \mathsf{b}\colon ?\mathsf{Int}.\mathsf{End}\}$), etc. On the side of session types, we can send and receive functions (e.g., $!(\mathsf{Int} \xrightarrow{*} \mathsf{Bool}).?\mathsf{Bool}.\mathsf{End}$), complex data types (e.g., $?\mathsf{List}.\mathsf{End}$) or even other channel endpoints (e.g., $!(?\mathsf{Int}.\mathsf{End}).\mathsf{End}$). The ability to express the passing of endpoints along endpoints allows us to characterize our session types as *higher-order* (otherwise, we would characterize them as *first-order*).

Note 3.3.2. The syntax of regular session types dictates that all sending and receiving actions be followed by a continuation, ruling out types like $?\mathsf{Int}$ or $?\mathsf{Int}.!\mathsf{Bool}$. As we show below, this limits the protocols they can express, and it is precisely this limitation that context-free session types address.

3.4 Subtyping regular session types

We now arrive at the subtyping problem for regular session types: when can we consider a session type S to be a subtype of R? In other words, when is it safe to treat a channel endpoint of type S as if it had a simpler type R?

Since we have already explored subtyping for unidirectional channel types ($\mathsf{Receive}\,T$ and $\mathsf{Send}\,T$) in Section 3.2, we begin by examining the session type constructors that most closely

resemble them: input and continuation $?T.S$, and output and continuation $!T.S$. This resemblance is, of course, located in the input and output part of the session type constructors. The difference, however, is twofold: (1) the $?T.S$ and $!T.S$ constructors characterize a *single*, rather than *every* intput/output action to take place on a certain channel endpoint, and, as a consequence, (2) they contain also a continuation part where the remaining actions are described by another session type.

It turns out that difference (1) is immaterial: the same reasoning outlined in Section 3.2 applies whether we are characterizing a single or every input/output action—it remains true that we can use a receiving endpoint for values of type T wherever one for values of type U is expected if T is a subtype of U ($?T.S$ is covariant on T), and, conversely, it also remains true that we can use a sending endpoint for values of type T wherever one for values of type U is expected if U is a subtype of T ($!T.S$ is contravariant on T). As such, we need only to account for difference (2), and here the reasoning is straightforward: if subtyping holds for the first action, it must hold for the remaining actions too. As such, we check subtyping on the continuation types recursively, in the same direction (in other words, $?T.S$ and $!T.S$ are both covariant on S). Of course, if we reach End on one type, we also expect to reach it on the other, since the only applicable action to a channel of type End is close. Thus we arrive at the following subtyping rules:

$$\text{S-InCont}\quad \frac{T \leq U \qquad S \leq R}{?T.S \leq ?U.R} \qquad \text{S-OutCont}\quad \frac{U \leq T \qquad S \leq R}{!T.S \leq !U.R} \qquad \text{S-End}\quad \mathsf{End} \leq \mathsf{End}$$

The reasoning for external choice $\&\{\ell\colon S_\ell\}_{\ell \in L}$ and internal choice $\oplus\{\ell\colon S_\ell\}_{\ell \in L}$ types follows directly from this. If we interpret external choices as the input of a label k and the continuation as S_k, we can characterize them as covariant on the set of labels and covariant on the continuations. Conversely, if we interpret internal choices as the output of a label k and continuation as S_k, we can characterize them as contravariant on the set of labels and covariant on the continuations.

However, this is better illustrated with an example. Suppose we'd like to encapsulate the behavior of a client who wants to know whether a certain integer is prime. To do this, we can design a function $\mathsf{isPrimeClient}$ that, given an integer n and a $\mathsf{PrimeClient}$ endpoint c, selects $\mathsf{IsPrime}$ on c, sends n and gets back a boolean, which it returns after selecting Finish and closing the channel:

$$\begin{aligned}
&\mathsf{isPrimeClient} = \lambda n : \mathsf{Int} \to \lambda c : \mathsf{PrimeClient} \to \\
&\quad \mathsf{let}\ s = \mathsf{send}\, n\, s\ \mathsf{in} \\
&\quad \mathsf{let}\ (b, s) = \mathsf{receive}\, s\ \mathsf{in} \\
&\quad \mathsf{close}\, s; \\
&\quad b
\end{aligned}$$

The type of this function should then be $\mathsf{Int} \xrightarrow{*} \mathsf{PrimeClient} \xrightarrow{*} \mathsf{Bool}$. Suppose, however, that we

decide to enrich the protocol with a new option to calculate the divisors of an integer:

$$\begin{aligned}\mathsf{PrimeClient2} = \mu s.\, \oplus\{&\mathsf{IsPrime}\colon \mathsf{!Int.?Bool}.s\\ ,\, &\mathsf{AreCoprimes}\colon \mathsf{!Int.!Int.?Bool}.s\\ ,\, &\mathsf{NthPrime}\colon \mathsf{!Int.?Int}.s\\ ,\, &\mathsf{Divisors}\colon \mathsf{!Int.?List}.s\\ ,\, &\mathsf{Finish}\colon \mathsf{End}\}\end{aligned}$$

Can we give function isPrimeClient an endpoint of type PrimeClient2? The naive answer would be no: type PrimeClient2 is clearly different from type PrimeClient. Yet it is easy to see that no communication errors can arise, since PrimeClient2 retains the options IsPrime and Finish used by function isPrimeClient. According to Liskov's principle of safe substitution, we should recognize PrimeClient2 as a subtype of PrimeClient, allowing a value of the former type to be used wherever a value of the latter type is required.

This reasoning is analogous to the one we used for record types in Section 2.3 to recognize that $\{\mathsf{name}\colon \mathsf{String}, \mathsf{age}\colon \mathsf{Int}, \mathsf{gpa}\colon \mathsf{Float}\}$ is a subtype of $\{\mathsf{name}\colon \mathsf{String}, \mathsf{age}\colon \mathsf{Int}\}$. We can thus generalize it in a similar fashion, arriving at the following subtyping rule for internal choices.

$$\text{S-IntChoice}\quad \frac{K \subseteq L \qquad S_j \leq R_j \ (\forall j \in K)}{\oplus\{\ell\colon S_\ell\}_{\ell \in L} \leq \oplus\{k\colon R_k\}_{k \in K}}$$

The covariance of the continuations, described by premise $S_j \leq R_j \ (\forall j \in K)$, is justified by the same reasoning we outlined for the $?T.S$ and $!T.S$ constructors.

We can now be more precise when assigning a type to function isPrimeClient, specifying only the options it really needs to select: $\mathsf{Int} \xrightarrow{*} \mu s.\oplus\{\mathsf{IsPrime}\colon \mathsf{!Int.?Bool}.s, \mathsf{Finish}\colon \mathsf{End}\} \xrightarrow{*} \mathsf{Bool}$. Since $\mu s.\oplus\{\mathsf{IsPrime}\colon \mathsf{!Int.?Bool}.s, \mathsf{Finish}\colon \mathsf{End}\}$ is a subtype of both PrimeClient and PrimeClient2, the type system will accept an endpoint of any of the two types as a valid argument to the function, excluding the need to declare identical functions with different types.

Let us now give an example for external choices. Suppose we want to encapsulate the behavior of a prime number server offering the services described by PrimeServer. We can do this with a recursive function primeServer of type $\mathsf{PrimeServer} \xrightarrow{*} \{\}$ (where the empty record $\{\}$ denotes the trivial value returned by function close):

$$\begin{aligned}\mathsf{primeServer} = \lambda s : \mathsf{PrimeServer} \xrightarrow{*} \mathsf{match}\ s\ \mathsf{with}\ \{&\mathsf{IsPrime}\, s \to \mathsf{let}\ (n, s) = \mathsf{receive}\, s\ \mathsf{in}\\ &\qquad\qquad \mathsf{let}\ s = \mathsf{send}\, (\mathsf{prime}\, n)\, s\ \mathsf{in}\\ &\qquad\qquad \mathsf{primeServer}\, s\\ ,\, &\mathsf{AreCoprimes}\, s \to \ldots\\ ,\, &\mathsf{NthPrime}\, s \to \ldots\\ ,\, &\mathsf{Finish}\, s \to \mathsf{close}\, s\}\end{aligned}$$

Suppose we again enrich the protocol with a new option to calculate the divisors of an integer.

$$\begin{aligned}\mathsf{PrimeClient2} = \mu s.\, \&\{&\mathsf{IsPrime}\colon ?\mathsf{Int}.!\mathsf{Bool}.s\\ ,\, &\mathsf{AreCoprimes}\colon ?\mathsf{Int}.?\mathsf{Int}.!\mathsf{Bool}.s\\ ,\, &\mathsf{NthPrime}\colon ?\mathsf{Int}.!\mathsf{Int}.s\\ ,\, &\mathsf{Divisors}\colon ?\mathsf{Int}.!\mathsf{List}.s\\ ,\, &\mathsf{Finish}\colon \mathsf{End}\}\end{aligned}$$

Then, we must update function primeServer to account for option Divisors, and update also its type to PrimeServer2 $\xrightarrow{*} \{\}$. Can we still pass it an endpoint of the original type PrimeServer? The naive answer would be no, for PrimeServer is clearly different from PrimeServer2. Yet again it is easy to see that no communication errors can arise if we do, since the function can still handle all the branches of type PrimeServer. We should therefore consider PrimeServer as a subtype of PrimeServer2.

Here too the reasoning is similar to another previous construct we have seen, namely the variant types of Section 2.4. We can thus generalize it accordingly, arriving at the following rule for external choices:

$$\text{S-ExtChoice}\quad \frac{L \subseteq K \qquad S_j \leq R_j \ (\forall j \in L)}{\&\{\ell\colon S_\ell\}_{\ell \in L} \leq \&\{k\colon R_k\}_{k \in K}}$$

Note 3.4.1. The subtyping properties of regular session types were first explored by Gay and Hole in the context of the π-calculus [42].

In the examples given thus far we gloss over subtyping for the recursive session type constructor $\mu s.S$, which we use to build types like PrimeClient and PrimeServer. We can deal with in the same manner we did for recursive functional types in Section 2.5: by switching to a coinductive interpretation of the subtyping rules, and including rules to unfold the recursive types at the left and right, as needed.[4]:

$$\text{S-RecL}\quad \frac{[\mu s.S/s]S \leq R}{\mu s.S \leq R} \qquad\qquad \text{S-RecR}\quad \frac{S \leq [\mu r.R/r]R}{S \leq \mu r.R}$$

Finally, since the integration of session types in a functional setting requires introducing a distinction between linear function types $T \xrightarrow{1} U$ (which must be used exactly one time) and unrestricted function types $T \xrightarrow{*} U$ (which may be used any number of types), one might wonder how they

[4] In their seminal work on this topic Gay and Hole [42] give an alternative, equivalent solution. Rather than being defined by a set of subtyping rules (as we have seen so far), their subtyping relation for recursive session types is based on the notion of *type simulation*, a relation coinductively defined by means of clauses of the form "if (T, U) is in the type simulation relation, then T and U are built using the same type constructor, and the types of certain fields of the constructors are also in that same type simulation relation" (in the same direction for covariant fields, and in the opposite direction for contravariant fields). Gay and Hole consider a type T to be a subtype of U if there is a type simulation containing (T, U).

should figure in a subtyping relation. The reasoning is quite simple: since unrestricted functions may be used any number of times, it is safe to use them exactly one time. In other words, it is safe to treat a function $T \xrightarrow{*} U$ as if it had type $T \xrightarrow{1} U$. To express this succintly, we can establish a preorder on the restrictions—also known as *multiplicities*—of functions, established by axioms $1 \sqsubseteq 1$, $* \sqsubseteq *$ and $* \sqsubseteq 1$, allowing us to say that $T \xrightarrow{m} U$ is a subtype of $T \xrightarrow{n} U$ if $m \sqsubseteq n$.

But this is not all. In fact, the classical subtyping properties of argument and return types explored in Section 2.3 also apply here: it is safe to treat a function $T_1 \xrightarrow{*} U_1$ as if it had type $T_2 \xrightarrow{1} U_2$ if T_2 is a subtype of T_1 and if U_1 is a subtype of U_2. Thus we arrive at the following subtyping rule for function types in general:

$$\text{S-ARROW}\quad \frac{T_2 \leq T_1 \qquad U_1 \leq U_2 \qquad m \sqsubseteq n}{T_1 \xrightarrow{m} U_1 \leq T_2 \xrightarrow{n} U_2}$$

Note 3.4.2. Multiple approaches to subtyping for regular session types have been proposed, and they can be classified according to the objects they consider substitutable: channels *versus* processes (the difference being most notable in the variance of type constructors). The earliest approach, subscribing to the substitutability of channels, is that of Gay and Hole [42]. It is also the one we follow in our contributions. A later formulation, proposed by Carbone et al. [16], subscribes to the substitutability of processes. A survey of both interpretations is given by Gay [41].

3.5 A subtyping algorithm for regular session types

To implement subtyping in the compiler for a programming language featuring regular session types, a subtyping algorithm is necessary. In Section 2.3 above, we discuss how a subtyping algorithm for non-recursive functional types can be derived from the subtyping rules for integers, booleans, records and functions by reading them bottom-up. Then, in Section 2.5, we discuss how the introduction of equirecursive types and the S-RECL and S-RECR rules requires a coinductive interpretation to allow infinite derivations, which renders the bottom-up approach moot (because it does not terminate for recursive types).

Since recursion is an integral part of session types, we cannot simply follow the bottom up approach here. Instead, we must devise a way to keep derivations finite. The solution Gay and Hole propose is to devise a different, inductive subtyping relation that is *sound* and *complete* with respect to the coinductively defined one [42]. By *sound* we mean that if (T, U) is in the inductive relation, then it is also in the coinductive one. By *complete* we mean that if it is in the coinductive relation, then it is also in the inductive one.

The key to eliminating infinite derivations is to understand them: in this case, they essentially arise from the unlimited unfolding of recursive types by the S-RECL and S-RECR rules, and follow a regular pattern in the sense that they are rooted at a conclusion that has already appeared somewhere down the derivation (see, for example, the derivation for $\mathsf{InfList} \leq \mathsf{List}$ in Section 2.5). What if we "stored" these conclusions in a set of subtyping assumptions Σ (much like we store

$$\text{AS-Assump}\ \frac{T \leq U \in \Sigma}{\Sigma \vdash T \leq U} \qquad \text{AS-RecL}\ \frac{\Sigma, \mu x.T \leq U \vdash [\mu x.T/x]T \leq U}{\Sigma \vdash \mu x.T \leq U} \qquad \text{AS-RecR}\ \frac{\Sigma, T \leq \mu x.U \vdash T \leq [\mu x.U/x]U}{\Sigma \vdash T \leq \mu x.U}$$

$$\text{AS-Int}\ \Sigma \vdash \mathsf{Int} \leq \mathsf{Int} \qquad \text{AS-Rcd}\ \frac{K \subseteq L \qquad \Sigma \vdash S_j \leq R_j\ (\forall j \in K)}{\Sigma \vdash \{\ell\colon S_\ell\}_{\ell \in L} \leq \{k\colon R_k\}_{k \in K}} \qquad \text{AS-Vrt}\ \frac{L \subseteq K \qquad \Sigma \vdash S_j \leq R_j\ (\forall j \in L)}{\Sigma \vdash \langle \ell\colon S_\ell \rangle_{\ell \in L} \leq \langle k\colon R_k \rangle_{k \in K}}$$

$$\text{AS-Arrow}\ \frac{T_2 \leq T_1 \qquad U_1 \leq U_2 \qquad m \sqsubseteq n}{T_1 \xrightarrow{m} U_1 \leq T_2 \xrightarrow{n} U_2} \qquad \text{AS-End}\ \Sigma \vdash \mathsf{End} \leq \mathsf{End} \qquad \text{AS-InCont}\ \frac{\Sigma \vdash T \leq U \qquad \Sigma \vdash S \leq R}{\Sigma \vdash ?T.S \leq ?U.R}$$

$$\text{AS-OutCont}\ \frac{\Sigma \vdash U \leq T \qquad \Sigma \vdash S \leq R}{\Sigma \vdash !T.S \leq !U.R} \qquad \text{AS-IntChoice}\ \frac{K \subseteq L \qquad \Sigma \vdash S_j \leq R_j\ (\forall j \in K)}{\Sigma \vdash \oplus\{\ell\colon S_\ell\}_{\ell \in L} \leq \oplus\{k\colon R_k\}_{k \in K}}$$

$$\text{AS-ExtChoice}\ \frac{L \subseteq K \qquad \Sigma \vdash S_j \leq R_j\ (\forall j \in L)}{\Sigma \vdash \&\{\ell\colon S_\ell\}_{\ell \in L} \leq \&\{k\colon R_k\}_{k \in K}}$$

Figure 3.2: Algorithmic subtyping for regular session types

typing assumptions in context Γ of the typing relation $\Gamma \vdash e : T$), allowing them to be proven immediately if they appear again somewhere up the derivation? This is the basis of Gay and Hole's algorithmic subtyping relation, inductively defined by the rules for judgment $\Sigma \vdash T \leq U$ in Fig. 3.2.

With these rules, we can finally obtain an algorithm by following the usual bottom-up approach, starting at goal $\vdash T \leq U$ (with an empty set of assumptions). However, since the rules are not quite syntax directed, two additional specifications are necessary: (1) AS-Assump should always be used if applicable (this guarantees termination), and (2) AS-RecL should be used in preference to AS-RecR if both are applicable (the preference is arbitrary and guarantees determinism).

Example 3.5.1. Using the newly defined algorithmic subtyping relation, the infinite derivation for $\mathsf{InfList} \leq \mathsf{List}$ in Section 2.5 can be avoided by using the AS-Assump rule:

$$\dfrac{\dfrac{\dots \qquad \dfrac{\mathsf{InfList} \leq \mathsf{List} \in \mathsf{InfList} \leq \mathsf{List}, \dots}{\mathsf{InfList} \leq \mathsf{List}, \dots \vdash \mathsf{InfList} \leq \mathsf{List}}\ \text{AS-Assump}}{\vdots}\ \text{AS-Rcd}}{\mathsf{InfList} \leq \mathsf{List} \vdash \langle \mathsf{cons}\colon \{\mathsf{val}\colon \mathsf{Int}, \mathsf{rest}\colon \mathsf{InfList}\}\rangle \leq \mathsf{List}}\ \text{AS-RecR}}{\vdash \mathsf{InfList} \leq \mathsf{List}}\ \text{AS-RecL}$$

Note 3.5.1. In Section 2.5 we mention how, under an equirecursive interpretation, and even without subtyping, a type checker must recognize a recursive type and its unfoldings as equivalent and interchangeable. Any valid equivalence relation is, naturally, reflexive, transitive and symmetric,

i.e., a symmetric preorder. It is quite straightforward to derive an equivalence relation from an existing preorder such as the subtyping relation: all we need to do is observe that two equivalent types are simultaneously subtypes and supertypes of each other. In other words, we can derive an equivalence relation from the more general subtyping relation, defined by $T \leqq\!\!\geqq U$ whenever $T \leq U$ and $U \leq T$. Then, even if we do not wish to support subtyping in a practical implementation, we can still use the algorithm outlined in this section to ensure that the equirecursive interpretation is respected.

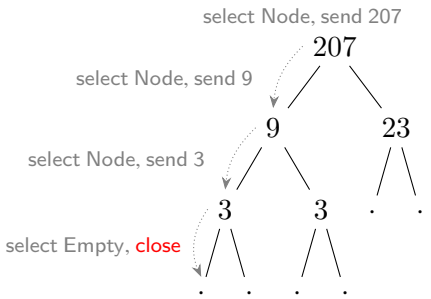
3.6 Limitations

Regular session types are limited in the protocols they can express. Suppose that we want to specify a protocol for sending a tree of integers along a channel. The most obvious solution is to express this using the session type !IntTree.End, where IntTree stands for the recursive variant type $\mu t.\langle \mathsf{Empty}\colon \{\}, \mathsf{Node}\colon \{\mathsf{left}\colon t, \mathsf{value}\colon \mathsf{Int}, \mathsf{right}\colon t\}\rangle$.

The protocol specified by !IntTree.End exchanges a single message containing the entire tree, which may be of any size. This, of course, is not always feasible. There are many practical scenarios where the communication medium requires that messages have a limited size. In such cases, a more sensible alternative is to break down the tree into a sequence of messages (a process known as *serialization*) and send them along the channel, using choices to let the receiver know how to put it back together. We adopt a recursive approach: to serialize an empty tree we simply select Empty; to serialize a non-empty tree, we send the value at the root and then recursively serialize the left and the right sub-trees (effectively performing what is known as a *depth-first traversal*). We can tentatively specify this protocol using a recursive session type like the following.

$$\mathsf{SerializeTree} = \mu s.\oplus\{\mathsf{Empty}\colon \mathsf{End}, \mathsf{Node}\colon \mathsf{!Int}.s.s\}$$

However, as illustrated below, we soon run into difficulties: serialization stops midway whenever we get to a node with no child at the left, since upon reaching End we must close the channel.



Furthermore, SerializeTree is actually malformed, since a sequential composition of the form $s.s$ is not syntactically valid (according to Fig. 3.1, we should have $?T$ or $!T$ before ".", not a type reference).

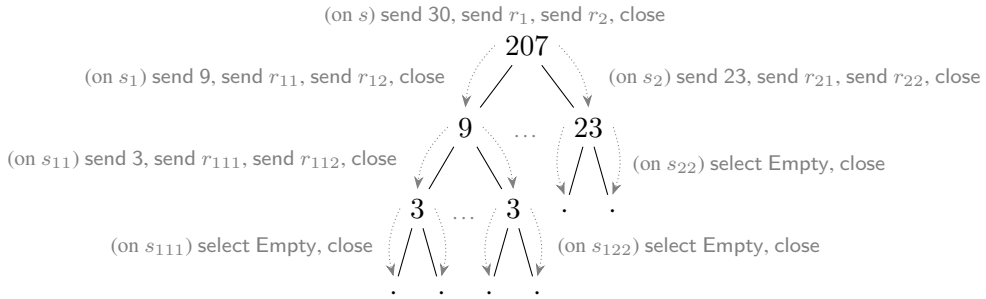
As it turns out, this constraint on the form of sequential composition is the main limitation to the expressivity of regular session types: types may refer to themselves, but only at the last step of

a sequential composition. In other words, they are restricted to *tail recursion*.

Note 3.6.1. An alternative to serializing a tree on a single channel is to use an inefficient delegation technique, where, for each sub-tree, we create a new channel, send its receiving endpoint, and serialize the sub-tree on the corresponding sending endpoint. To describe this protocol, we can use the following session type.

$$\mathsf{SerializeTreeDeleg} = \mu s.\oplus\{\mathsf{Empty}\colon \mathsf{End}, \mathsf{Node}\colon !\mathsf{Int}.!s.!s.\mathsf{End}\}$$

Governed by SerializeTreeDeleg, the serialization of our example tree could then proceed as illustrated below (where s_n and r_n denote the sending and receiving endpoints of the same channel).



3.7 Regularity

We can be more precise about the class of protocols that regular session types are restricted to: those corresponding to the union of *regular* and *ω-regular languages* (this is origin of their "regular" epithet). These notions originate from formal language theory, a field of linguistics and computer science that studies languages from an abstract, formal standpoind that focuses on the rules and patterns that describe the structure of languages, rather than their meaning.

In this field, a *language* can be described as a set of *strings*, which are sequences of *symbols* (like a and b) taken from a certain set called an *alphabet*. A string may contain no symbols: ε denotes the empty string. Sets $\{\mathsf{a}, \mathsf{b}\}$ $\{\mathsf{ab}, \mathsf{ba}\}$ and $\{\mathsf{aa}, \mathsf{bb}\}$ are examples of languages with alphabet $\{\mathsf{a}, \mathsf{b}\}$.

A language is said to be *regular* if it can be described by a *regular expression* [61, 93][5], a finite combination of alphabet symbols and three operations that combine sub-expressions: concatenation, alternation and iteration. Symbols are the simplest form of regular expressions. They denote singleton languages (e.g., expression a describes language $\{\mathsf{a}\}$). We can describe larger languages by combining these expressions using the operations mentioned above.

- *concatenation*, written EF, describes the language obtained by joining together the strings described by E to the strings described by F (e.g., expression ab describes language $\{\mathsf{ab}\}$);

[5] Alternatively and equivalently, a language is also said to be regular if all of its words can be recognized by a *finite state automaton*, an abstract machine consisting of a finite set of states and transitions, that successively changes its current state based on a string of input symbols. If at the end of the input the automaton is in an *acceptance state*, the string is said to be accepted.

- *alternation*, written $E \mid F$, describes the union of the languages described by E and F (e.g., expression $a|ab$ describes language $\{\mathsf{a}, \mathsf{ab}\}$);
- *iteration*, written E^*, describes the (infinite) union of $\{\varepsilon\}$ with the languages described by expressions E, EE, EEE... For example, the regular expression $\mathsf{a}(\mathsf{b} \mid \mathsf{c})^*$ describes language $\{\mathsf{a}, \mathsf{ab}, \mathsf{ac}, \mathsf{abb}, \mathsf{abc}, \mathsf{acb}, \mathsf{acc}, \ldots\}$.

Note 3.7.1. Regular expressions are widely used in programming to describe text patterns, which can be used to search for their ocurrences in strings or to validate text input. Despite maintaining the "regular" name, modern regular expression engines typically include advanced features that allow them to describe languages that are not strictly regular.

The resemblance between regular expressions and regular session types is readily seen. Intuitively, concatenation roughly corresponds to the $?T.S$ and $!T.S$ constructors, alternation to the $\oplus\{\ell\colon S_\ell\}_{\ell\in L}$ and $\&\{\ell\colon S_\ell\}_{\ell\in L}$ constructors, and iteration to the $\mu s.S$ constructor, in which self-reference s can only appear as the last step in in a type S that reaches an End (because language strings are finite). Thus, the finite communication sequences allowed by session types correspond to regular languages.

Yet some session types like $\mu s.!\mathsf{Int}.s$ and $\mu s.\oplus\{\mathsf{Nil}\colon \mathsf{End}, \mathsf{Cons}\colon !\mathsf{Int}.s\}$ may allow infinite communication sequences that do not reach an End. Hence we must also include *ω-regular languages* [14], which include only infinite strings. These linguistic structures are described by *ω-regular expressions*, the infinite counterpart of regular expressions that can be composed through three operations:

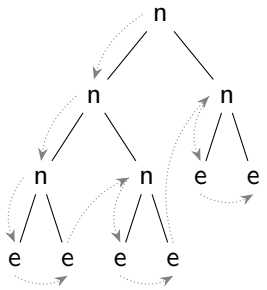
- *infinite iteration*, written E^ω where E is a regular expression, which describes ω-language $EEEEE\ldots$ (e.g., $(\mathsf{a}(\mathsf{b} \mid \mathsf{c})^*)^\omega$ describes $\{\mathsf{aaa}\ldots, \mathsf{abab}\ldots, \mathsf{acac}\ldots, \mathsf{abbabb}\ldots, \ldots\}$).
- *left-concatenation*, written EO, which prefixes the finite strings described by regular expression E to the infinite strings described by ω-regular expression O (e.g., $(\mathsf{a} \mid \mathsf{b})\mathsf{c}^\omega$ describes $\{\mathsf{accccc}\ldots, \mathsf{bccccc}\ldots\}$);[6]
- *alternation*, written $O \mid P$, which describes the union of the ω-languages described by ω-regular expressions O and P (e.g., $(\mathsf{a} \mid \mathsf{b})^\omega \mid (\mathsf{ab})^\omega$ describes $\{\mathsf{aaa}\ldots, \mathsf{bbb}\ldots, \mathsf{abab}\ldots\}$)

Regular and ω-regular expressions are somewhat limited in the kinds of languages they can express. In the classical formulation of the *Chomsky hierarchy*, which classifies language classes according to their complexity, regular languages stand as the least complex. For example, it is not possible to write regular expressions that specify languages as simple as "a certain number of as followed by the same number of bs", i.e., $\{\varepsilon, \mathsf{ab}, \mathsf{aabb}, \mathsf{aaabbb}, \mathsf{aaaabbbb}, \ldots\}$, languages of balanced parenthesis of arbitrary depth, e.g., $\{\varepsilon, (), (()), ()(), (()()), ()(()()), \ldots\}$, or a language of strings that describe depth-first traversals of binary trees, e.g.,

$$\{\mathsf{e}, \mathsf{nee}, \mathsf{nneee}, \mathsf{nneenee}, \mathsf{nnneeneenee}, \mathsf{nnneeneennee}, \mathsf{nnneeneennenee}, \ldots\}$$

[6]The reverse operation, OE, is not well defined and therefore not addressed by ω-regular expressions.

where, for instance, nnneeneenee describes the following traversal.[7]



Given the correspondence between regular session types and regular languages, it is no wonder that we cannot write a type like SerializeTree to specify the tree serialization protocol we described above—it is an inherently non-regular communication pattern. Following the example of language $\{\varepsilon, \mathsf{ab}, \mathsf{aabb}, \ldots\}$, another such pattern is the sending of a certain number of integers followed by the reception of the same number of booleans. To express these communication patterns, we need a new formulation of session types, one that corresponds to a class of languages that stands higher in the Chomsky hierarchy. This is where *context-free session types* come in.

[7] The underlying problem in all of these examples is, roughly stated, that regular expressions and the corresponding automata "lack memory". Thus, in our examples, they cannot keep track of the number of a's, or open parenthesis, or untraversed nodes.

Chapter 4

Context-free session types

We now turn to *context-free session types* [2, 23, 86, 96], a more recent and expressive formulation of session types that overcomes the limitations of regular session types identified in Section 3.6 by allowing the specification of protocols that recur at any point, or, in other words, recursive protocols that are not limited to tail recursion.

The enhanced expressive power of context-free session types does not come without its challenges: it complicates their subtyping problem to the point of *undecidability*[1], making it impossible to design an algorithm that, given any two context-free session types S and R, is able to correctly decide whether S is a subtype of R in a finite amount of time [82]

Despite these challenges, we were able to formulate an intuive notion of subtyping and design a sound subtyping algorithm for it. These contributions are presented in the chapters that follow. For now, we introduce the syntax of context-free session types, identify the larger class of protocols they allow, and, finally, explore the solutions already proposed in the literature for their equivalence and subtyping problems.

4.1 Syntax and examples

Context-free session types overcome the limitations of regular session types by untangling sequential composition from input and output: instead of the $?T.S$ and $!T.S$ constructors, we have the standalone input $?T$ and output $!T$ constructors, a general sequential composition operator $S;R$ (in which S and R may be arbitrary session types) and, finally, session type Skip, which represents no action—the empty protocol. The syntax is shown in Fig. 4.1.

Together, these changes enable us to finally write a valid session type to express tree serialization on a single channel:

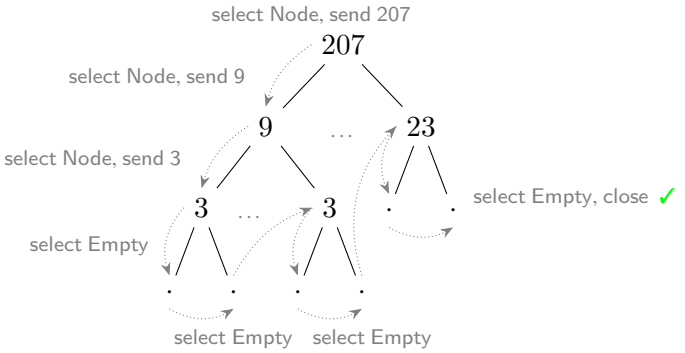
$$\mathsf{SerializeTree} = \mu s.\oplus\{\mathsf{Empty}\colon \mathsf{Skip}, \mathsf{Node}\colon !\mathsf{Int};s;s\}$$

[1] Undecidability is predicated on how the subtyping problem is posed, i.e., on the features expected from the subtyping relation. Here we refer to a comprehensive subtyping relation, analogous to that outlined in Section 3.4, accounting for subtyping in messages, choices and recursive types. It may be possible to design a restricted relation that remains decidable.

$$
\begin{array}{rl}
\textbf{types} & T, U, V, W ::= \mathsf{Unit} \mid T \xrightarrow{m} U \mid (\!|\ell\colon T|\!)_{\ell \in L} \mid S \mid t \mid \mu t.T \\
\textbf{session types} & S, R ::= \sharp T \mid \odot\{\ell\colon S_\ell\}_{\ell \in L} \mid \mathsf{Skip} \mid \mathsf{End} \mid S;R \mid s \mid \mu s.S \\
\textbf{restrictions} & m, n ::= 1 \mid * \\
\textbf{records/variants} & (\!|\cdot|\!) ::= \{\cdot\} \mid \langle\cdot\rangle \\
\textbf{directions} & \sharp ::= ? \mid ! \\
\textbf{choices} & \odot ::= \oplus \mid \&
\end{array}
$$

Figure 4.1: Syntax of context-free session types.

Example 4.1.1. The following tree traversal illustrates a serialization governed by the context-free session type SerializeTree;End.



Example 4.1.2. A similar, more complex example that better demonstrates how expressive and useful context-free session types can be is the serialization of data in a JSON format. The possibility of aggregating arbitrary values using objects and arrays makes JSON effectively non-regular. We can safely serialize data in this format using an endpoint governed by the following SerializeJSON type (which, mirroring the syntax of JSON, features two levels of recursion to allow the list-like serialization of arrays and objects):

$$
\begin{aligned}
\mathsf{SerializeJSON} = \mu s.\, \oplus\{&\mathsf{Null}\colon \mathsf{Skip} \\
&, \mathsf{Boolean}\colon \mathsf{!Bool} \\
&, \mathsf{String}\colon \mathsf{!String} \\
&, \mathsf{Number}\colon \mathsf{!Float} \\
&, \mathsf{Array}\colon \mu r.\oplus\{\mathsf{Nil}\colon \mathsf{Skip}, \mathsf{Cons}\colon s;r\} \\
&, \mathsf{Object}\colon \mu r.\oplus\{\mathsf{Nil}\colon \mathsf{Skip}, \mathsf{Cons}\colon \mathsf{!String};s;r\}\}
\end{aligned}
$$

Example 4.1.3. Since the $(;)$ operator allows us to sequentially compose any two session types, we can also easily express protocols to serialize two trees (SerializeTree;SerializeTree), three trees (SerializeTree;SerializeTree;SerializeTree), an infinite sequence of trees (μs.SerializeTree;s), a tree of trees ($\mathsf{SerializeTreeTree} = \mu s.\oplus\{\mathsf{Empty}\colon \mathsf{Skip}, \mathsf{Node}\colon \mathsf{SerializeTree};s;s\}$), or even an infinite sequence of trees of trees (μs.SerializeTreeTree;s).

Example 4.1.4. Naturally, context-free session types are also "backwards compatible", in the sense that any protocol that can be expressed through a regular session type can also be expressed with a context-free session type. All we need to do is rewrite $\sharp T.S$ as $\sharp T;S$ (where, according to Fig. 4.1, $\sharp$ stands for either ! or ?). We illustrate this by rewriting regular session types from the previous chapter as context-free session types.

$$\begin{array}{lll} \mathsf{!Int;?Bool;End} & \mathsf{!Bool;}(\mu s.\mathsf{?Int;}s) & \mu s.\oplus\{\mathsf{IsPrime:\,!Int;?Bool;}s \\ & & \quad,\mathsf{NthPrime:\,!Int;?Int;}s \\ & & \quad,\mathsf{NthPrime:\,!Int;?Int;}s \\ & & \quad,\mathsf{Finish:\,End}\} \end{array}$$

As Section 3.3 shows, in the regular setting the send and receive operations consume an endpoint and return another endpoint with an updated type, where communication may proceed. The same happens with context-free session types: performing send on an endpoint of type $!T;S$ returns an endpoint of type S, and performing receive on an endpoint of type $?T;S$ returns a value of type T and an endpoint of type S. However, since input and output are no longer syntactically tied to a continuation, the reader may wonder how types like !Int and ?Int "evolve". The answer is simple: since no further action is specified after sending or receiving, the continuation endpoints have type Skip, the empty protocol that specifies no action. In fact, we can view Skip as being the implicit continuation of any session type—after all, ?Int;Skip represents the same protocol as ?Int (or ?Int;Skip;Skip, or even Skip;?Int;Skip, for that matter).

Type Skip possesses two more interesting properties: it is dual to itself (if one participant does nothing, so does the other), and it is unrestricted (i.e., not linear). The latter property means that an endpoint of this type may be discarded at will—after all, no operation applies to it.

Note 4.1.1. Our syntax for context-free session types does not require including End as the final action in a protocol—meaning that endpoints may simply reach type Skip and remain unused. Since unused channels may still consume computing resources, in practical settings it may be wise to enforce the presence of an End for types that allow finite communication sessions. This will ensure that their corresponding channels are eventually closed and wiped from memory.

4.2 Well-formedness

We do not consider all types generated by the grammar in Fig. 4.1 to be *well-formed*. Consider session type $\mu r.r;$!Unit. No matter how many times we unfold it, we cannot resolve its first communication action. The same could be said of $\mu r.$Skip$;r;$!Unit. We must therefore ensure that any self-reference in a sequential composition is preceded by a type constructor representing some meaningful action, i.e., not equivalent to Skip. This is achieved by adapting the conventional notion of contractivity (no subterms of the form $\mu x.\mu x_1.\ \ldots\ \mu x_n.x$, cf. Note 2.5.2) to account for Skip as the identity of sequential composition.[2]

[2]This corresponds to the notion of *guardedness* in the theory of process algebra [47, 71].

Type formation (*inductive*) $\boxed{\Delta \vdash T}$

$$\textsc{TF-Axiom}\ \frac{T = \mathsf{Unit}, \mathsf{End}, \mathsf{Skip}}{\Delta \vdash T} \qquad \textsc{TF-Arrow}\ \frac{\Delta \vdash T \qquad \Delta \vdash U}{\Delta \vdash T \xrightarrow{m} U} \qquad \textsc{TF-RcdVrt}\ \frac{\Delta \vdash T_\ell \qquad (\forall \ell \in L)}{\Delta \vdash (\!|\ell\colon T_\ell|\!)_{\ell \in L}} \qquad \textsc{TF-Msg}\ \frac{\Delta \vdash T}{\Delta \vdash \sharp T}$$

$$\textsc{TF-Choice}\ \frac{\Delta \vdash S_\ell \qquad (\forall \ell \in L)}{\Delta \vdash \odot\{\ell\colon S_\ell\}_{\ell \in L}} \qquad \textsc{TF-Seq}\ \frac{\Delta \vdash S \qquad \Delta \vdash R}{\Delta \vdash S;R} \qquad \textsc{TF-Var}\ \frac{X \in \Delta}{\Delta \vdash X}$$

$$\textsc{TF-Rec}\ \frac{T\not\checkmark \qquad T\ \mathsf{contr}\ X \qquad \Delta, X \vdash T}{\Delta \vdash \mu X.T}$$

Contractivity (*inductive*) $\boxed{T\ \mathsf{contr}\ x}$

$$\textsc{C-Axiom}\ \frac{T = \mathsf{Unit}, U \xrightarrow{m} V, (\!|\ell : T_\ell|\!)_{\ell \in L}, \mathsf{End}, \sharp T, \odot\{\ell\colon S_\ell\}_{\ell \in L}, \mathsf{Skip}}{T\ \mathsf{contr}\ x}$$

$$\textsc{C-Seq1}\ \frac{S\checkmark \qquad R\ \mathsf{contr}\ x}{S;R\ \mathsf{contr}\ x} \qquad \textsc{C-Seq2}\ \frac{S\not\checkmark \qquad S\ \mathsf{contr}\ x}{S;R\ \mathsf{contr}\ x} \qquad \textsc{C-Var}\ \frac{y \neq x}{y\ \mathsf{contr}\ x} \qquad \textsc{C-Rec}\ \frac{T\ \mathsf{contr}\ x}{\mu y.T\ \mathsf{contr}\ x}$$

Is-terminated (*inductive*) $\boxed{T\checkmark}$

$$\checkmark\textsc{-Skip}\ \mathsf{Skip}\checkmark \qquad \checkmark\textsc{-Seq}\ \frac{S\checkmark \qquad R\checkmark}{S;R\checkmark} \qquad \checkmark\textsc{-SRec}\ \frac{S\checkmark}{\mu s.S\checkmark}$$

Figure 4.2: Type formation for context-free session types.

In addition to contractivity, we must ensure that well-formed types contain no free references. The type formation judgement $\Delta \vdash T$, where Δ is a set of references, combines these requirements. It is inductively defined by the rules in Fig. 4.2. Notation Δ, x should be understood as requiring $x \notin \Delta$.

4.3 Context-freedom

Context-free session types are more expressive than their regular variants, since they can express every protocol that a regular session type can, and some more. We can make this statement more precise by looking once again at formal language theory and observing that the protocols that can be expressed by context-free session types correspond to a class of languages that includes regular languages—the so-called *context-free languages*.

Just as regular languages can described by regular expressions, context-free language can be described by a more powerful device: *context-free grammars* [93]. A context-free grammar can

be summarized as a 4-tuple $(\mathcal{V}, \Sigma, \mathcal{P}, \mathcal{S})$: a set $\mathcal{V}$ of symbols we call *variables*; a set Σ of symbols called an alphabet (as in Section 3.7); a set $\mathcal{P}$ of substitution rules of the form $X \to \alpha$, which we call *productions*, that map variables to possibly empty sequences α comprised of variables and alphabet symbols; and a *starting variable* $\mathcal{S}$. The following are three examples of context-free grammars (over alphabets $\{\mathsf{a}, \mathsf{b}\}$, $\{(,)\}$ and $\{\mathsf{e}, \mathsf{n}\}$, respectively):

$$\begin{array}{lll} S \to \mathsf{a}S\mathsf{b} & S \to (S)S & S \to N \\ S \to \varepsilon & S \to \varepsilon & S \to \mathsf{e} \\ & & N \to \mathsf{n}SS \end{array}$$

Such grammars can be used to generate strings of alphabet symbols and the set of all strings generated by a grammar is said to be the language it describes. To generate a string, we begin by writing down the starting variable S. Then, we look for a production mapping S to some α. If α contains no variables, then it is ε or contains only alphabet symbols, and hence we consider it a valid string. If, however, α contains variables, then for each instance of a variable X we look for a corresponding production $X \to \beta$ and replace that instance of X in α with β. We do this repeatedly until we obtain a valid string of alphabet symbols.

Note 4.3.1. Since every occurrence of a variable must be substituted until only alphabet symbols remain, variables are also called *non-terminal symbols*. By the same token, alphabet symbols are also called *terminal symbols*. We adopt this terminology to avoid confusion with the variables x of the functional language in Section 2.1.

Following this procedure on the first grammar above, we can obtain string ε if we immediately choose production $S \to \varepsilon$ on the second step. If we choose production $S \to \mathsf{a}S\mathsf{b}$ instead, we are left with $\mathsf{a}S\mathsf{b}$, and must again look for a production to replace S. We can choose $S \to \varepsilon$, obtaining the string ab, or $S \to \mathsf{a}S\mathsf{b}$, obtaining $\mathsf{aa}S\mathsf{bb}$, in which case we must replace S again. Thus it is easy to see that the set of all strings that the first grammar can generate, i.e., the language it describes, is $\{\varepsilon, \mathsf{ab}, \mathsf{aabb}, \mathsf{aaabbb}, \ldots\}$. Likewise, the languages described by the second and third grammars are, respectively, $\{\varepsilon, (), (()), ()(), \ldots\}$ and $\{\varepsilon, \mathsf{e}, \mathsf{nee}, \mathsf{nneeee}, \mathsf{nneenee}, \ldots\}$.

These three languages are precisely the examples of non-regular languages shown in Section 3.7. But now, with the power of context-free session types, we can describe protocols analogous to them. For language $\{\varepsilon, \mathsf{ab}, \mathsf{aabb}, \mathsf{aaabbb}, \ldots\}$, consider a protocol for sending a certain number of integers and then receiving the same number of booleans: while it cannot be described by any regular session type, the context-free session type $\mu s.\oplus\{\mathsf{More}\colon !\mathsf{Int}; s; ?\mathsf{Bool}, \mathsf{Stop}\colon \mathsf{Skip}\}$ works just fine. An example for $\{\varepsilon, (), (()), ()(), \ldots\}$ can be constructed analogously, while an example for $\{\varepsilon, \mathsf{e}, \mathsf{nee}, \mathsf{nneeee}, \mathsf{nneenee}, \ldots\}$ was already shown above in Section 4.1.

Naturally, context-free grammars can also describe regular languages: observe that any regular expression can easily be rewritten as a context-free grammar of equivalent power. For example, the following grammar describes exactly the same language as regular expression $\mathsf{a}(\mathsf{b} \mid \mathsf{c})*$.

$$S \to \mathsf{a}X \qquad X \to \mathsf{b}X \qquad X \to \mathsf{c}X \qquad X \to \varepsilon$$

Context-free languages in general can be quite complex, to the point of some very relevant problems being undecidable. For example, it is undecidable whether two context-free grammars generate the same language (the *language equivalence* problem), or if the language generated by one is contained in the language generated by another (the *language inclusion* problem) [93]. If context-free session types corresponded to context-free languages in general, this would mean that their equivalence, i.e., whether two types represent the same protocol, was undecidable. Fortunately, this is not the case.

It is a well known result in formal language theory that any context-free grammar can be transformed into an equivalent grammar in which every production has the form $X \to a\vec{Z}$, where a is a terminal symbol and $\vec{Z}$ a sequence of non-terminal symbols—this is known as a grammar in *Greibach Normal Form* (GNF) [46]. Furthermore, context-free session types are deterministic: for every communication action, there is only one possible continuation. This makes them coincide with GNF grammars where, for each non-terminal symbol X and terminal symbol a, there is at most one production of the form $X \to a\vec{Z}$. When a context-free grammar in GNF obeys this condition, it is said to be *simple* [62]

Simple context-free grammars, unlike their unqualified counterparts, exhibit a very useful property: they have a decidable *language equivalence* problem—in other words, it is possible to design an algorithm that, given two simple grammars, is always able to correctly answer, in a finite amount of time, whether they generate the same language [62]. The more general *language inclusion* problem, however, remains undecidable—there is no algorithm that, given two simple grammars, can correctly answer, in a finite amount of time, wether the language generated by one of them is contained in the language generated by the other [38]. These two properties of simple grammars are, as it turns out, crucial for the theory of context-free session types. Their relevance is made clearer in the following sections.

4.4 Algebraic properties

Can two syntactically different context-free session types represent the same communication pattern? Section 2.5 mentions how equirecursion complicates the type equivalence problem by making recursive types and their unfoldings semantically equivalent. Sections 3.4 and 3.5 further show how this issue generalizes to subtyping in the regular setting, and how it can be dealt with using a sound and complete algorithm based on modified subtyping rules.

Context-free session types, by virtue of their generalized sequencing operator $(;)$, complicate this further: even without recursion, they make it possible to represent the same sequential composition of actions in a myriad of ways. To be more specific, this operator exhibits four important algebraic properties: *identity*, *associativity*, *distributivity* and *absorption*. Since any sensible subtyping (or, for that matter, equivalence) relation should respect these properties, it may be useful to briefly survey them.

Identity As Section 4.1 exemplifies, types $\mathsf{Skip};S$, $S;\mathsf{Skip}$ and S are all represent the same com-

munication pattern. In algebraic terms, we can say Skip is the *neutral* or *identity element* of sequential composition.[3]

Associativity With the regular session type syntax, there is no ambiguity about the grouping of session type constructors in a type without parenthesis such as !Int.?Bool.End—the only possibility is !Int.(?Bool.End). In the context-free setting, this is no longer the case: we could parse type !Int;?Bool;End as either !Int;(?Bool;End) or (?Int;?Bool);End. By convention, we default to the former interpretation, but we observe that it should not really matter: both readings denote exactly the same protocol (send an integer, receive a boolean, and close the channel). We further observe that this is true in general: for any session types S_1, S_2, and S_3, type $S_1;(S_2;S_3)$ denotes the same protocol as $(S_1;S_2);S_3$. In other words, we may rearrange the order in which we apply sequential composition, as long as the order of the composed types appear does not change. In algebraic terms, we say that the sequential composition operator is *associative*.[4]

Distributivity In the regular setting, internal and external choice types admit no continuation; all possible remaining actions in the protocol are contained in their fields. This is no longer true for context-free session types, as their general sequencing operator enables any type to have a continuation. For choice types, the continuation is reached after the actions specified in each field are complete. For example, in type $\oplus\{\mathsf{A}\colon \mathsf{!Int}, \mathsf{B}\colon \mathsf{?Bool}\};\mathsf{End}$, we reach End after selecting A and sending an integer, or after selecting B and receiving a boolean. It would make no difference, then, to rewrite this type as $\oplus\{\mathsf{A}\colon \mathsf{!Int;End}, \mathsf{B}\colon \mathsf{?Bool;End}\}$. More generally, any type of the form $\odot\{\ell\colon S_\ell\}_{\ell\in L};R$ denotes exactly the same protocol as $\odot\{\ell\colon S_\ell;R\}_{\ell\in L}$. Note, however, that this is not true of we reverse the order of operations: $R;\odot\{\ell\colon S_\ell\}_{\ell\in L}$ is quite different from $\odot\{\ell\colon R;S_\ell\}_{\ell\in L}$, since in the latter a choice must be selected before reaching R. For this reason, we say that sequential composition *distributes to the right* over internal and external choice.[5]

Absorption Type End represents the closing of a channel, which prevents any further communication from taking place. What should we make of a type like End;!Int, then? From the point of view of a process, this type means exactly the same thing as End: a channel must be closed and not used further. This is, of course, true in general: End nullifies its continuation, whatever it may be. For any type S, in End;S is equivalent to End. In algebraic terms, we say that End is the *left-absorbing element* of sequential composition.[6]

[3] For comparison: in arithmetic, 1 is the identity element of multiplication ($1 \times x = x \times 1 = x$), and 0 is the identity element of addition ($0 + x = x + 0 = x$).

[4] For comparison: in real number arithmetic, both addition and multiplication are associative operations, since both $a+(b+c) = (a+b)+c$ and $a \times (b \times c) = (a \times b) \times c$ hold for any a,b and c. The same cannot be said of subtraction, for example, since $a - (b - c) = (a - b) - c$ is not generally true.

[5] For comparison: in real number arithmetic, division distributes to the right over addition, since $(a + b) \div c = (a \div c) + (b \div c)$ holds for any a,b and c with $c \neq 0$. Like sequential composition, it also does not distribute to the left, since $a \div (b + c) = (a \div b) + (a \div c)$ is not generally true.

[6] For comparison: in real number arithmetic, 0 is a left-absorbing element of multiplication, since, for any a, it holds that $0 \times a = 0$.

4.5 Equivalence

Before looking at the subtyping problem in its full generality, it might be useful to understand a more specific (and tractable) instance of it—type equivalence—as its solution contains the tools we will need in the following chapters.

Two session types are considered equivalent if they describe *exactly* the same communication actions, even if they are not identical syntactically. The previous section provides multiple examples of this: for instance, $\mathsf{Skip};S$ and S for any S. How can we capture this notion more precisely? One possible answer, given by Costa et al. [23], is to define an equivalence relation $\simeq$ between types by means of inference rules, much like we have done thus far for subtyping. The rules definining this relation, adapted to our formulation of context-free session types, are given in Fig. 4.3. Our adaptations, highlighted in the figure, concern the recursion operator μ, linear functions, and the End type, which are not present in the session type language of Costa et al.[7]

Summarizing the figure, a rule is included for each type constructor. For nullary constructors, this takes the form of a reflexive axiom (E-UNIT, E-SKIP and E-END), while for constructors of greater arity the rules include premises requiring equivalence at every field of the constructor (E-ARROW, E-RCDVRT, E-MSG, and E-CHOICE). The exceptions to this general outline are the μ and $(;)$ constructors. We treat the former as we have done in previous chapters, i.e., by including two rules to unfold the type at the premises, E-RECL and E-RECR (which require a coinductive interpretation). The $(;)$ constructor, however, merits special treatment: its rules should account for identity, associativity, distributivity and absorption. Thus we include, for each session type constructor S, a left-hand rule with a conclusion of the form $S;R \leq S'$ (E-MSGSEQ1L, E-CHOICESEQL, E-SKIPSEQL, E-ENDSEQ1L, E-SEQSEQL and , E-RECSEQL) and a right-hand rule with a conclusion of the form $S' \leq S;R$ (E-MSGSEQ1R, E-CHOICESEQR, E-SKIPSEQR, E-ENDSEQ1R, E-SEQSEQR and , E-RECSEQR). An additional rule is necessary for each constructor over which sequential composition does not distribute or identify (S-MSGSEQ2 and S-ENDSEQ2). It can be shown that the relation defined by these rules satisfies the properties of reflexivity, transitivity and symmetry expected from an equivalence relation [23].

Despite providing a clear formulation of equivalence, these rules do not suggest an immediate algorithm. The reason for this is, much like in the regular setting (Section 3.4), the inclusion of the E-RECL, E-RECR, E-RECSEQL and E-RECSEQR rules, which force us to adopt a coinductive interpretation to accept infinite derivations as valid. And while in the regular setting the tail-recursive structure of session types ensures that the infinite branches of a derivation follow a cyclic pattern, in the context-free setting this is no longer the case: non-tail recursion allows types to grow along the infinite branches of a derivation. As such, these infinite derivations cannot be captured by enriching the equivalence judgment with a context of equivalence assumptions, as we did for algorithmic subtyping. Take, as an example, the derivation

[7] Instead of the usual μ representation for recursive types, the original system relies on type variables defined by possibly recursive systems of equations; we opted for the μ operator approach because it is more closely aligned with programming language literature and implementation.

Equivalence (*coinductive*) $\boxed{T \simeq T}$

E-UNIT
$\mathsf{Unit} \simeq \mathsf{Unit}$

E-ARROW
$$\frac{U_1 \simeq T_1 \qquad T_2 \simeq U_2}{T_1 \xrightarrow{m} T_2 \simeq U_1 \xrightarrow{m} U_2}$$

E-RCDVRT
$$\frac{T_k \simeq U_k \ (\forall k \in L)}{(\!|\ell\colon T_\ell|\!)_{\ell\in L} \simeq (\!|\ell\colon U_\ell|\!)_{\ell\in L}}$$

E-RECL
$$\frac{[\mu x.T/x]T \simeq U}{\mu x.T \simeq U}$$

E-RECR
$$\frac{T \simeq [\mu x.U/x]U}{T \simeq \mu x.U}$$

E-MSG
$$\frac{T \simeq U}{\sharp T \simeq \sharp U}$$

E-CHOICE
$$\frac{S_k \simeq R_k \ (\forall k \in L)}{\odot\{\ell\colon T_\ell\}_{\ell\in L} \simeq \odot\{\ell\colon U_\ell\}_{\ell\in L}}$$

E-SKIP
$\mathsf{Skip} \simeq \mathsf{Skip}$

E-END
$\mathsf{End} \simeq \mathsf{End}$

E-MSGSEQ1L
$$\frac{T \simeq U \qquad S \simeq \mathsf{Skip}}{\sharp T;S \simeq \sharp U}$$

E-MSGSEQ1R
$$\frac{T \simeq U \qquad S \simeq \mathsf{Skip}}{\sharp T \simeq \sharp U;S}$$

E-MSGSEQ2
$$\frac{T \simeq U \qquad S \simeq R}{\sharp T;S \simeq \sharp U;R}$$

E-CHOICESEQL
$$\frac{\odot\{\ell\colon S_\ell;S\}_{\ell\in L} \simeq R}{\odot\{\ell\colon S_\ell\}_{\ell\in L};S \simeq R}$$

E-CHOICESEQR
$$\frac{S \simeq \odot\{\ell\colon R_\ell;R\}_{\ell\in L}}{S \simeq \odot\{\ell\colon R_\ell\}_{\ell\in L};R}$$

E-SKIPSEQL
$$\frac{S \simeq R}{\mathsf{Skip};S \simeq R}$$

E-SKIPSEQR
$$\frac{S \simeq R}{S \simeq \mathsf{Skip};R}$$

E-ENDSEQ1L
$\mathsf{End};S \simeq \mathsf{End}$

E-ENDSEQ1R
$\mathsf{End} \simeq \mathsf{End};R$

E-ENDSEQ2
$\mathsf{End};S \simeq \mathsf{End};R$

E-SEQSEQL
$$\frac{S_1;(S_2;S_3) \simeq R}{(S_1;S_2);S_3 \simeq R}$$

S-SEQSEQR
$$\frac{S \simeq R_1;(R_2;R_3)}{S \simeq (R_1;R_2);R_3}$$

S-RECSEQL
$$\frac{([\mu s.S_1/s]S_1);S_2 \simeq R}{(\mu s.S_1);S_2 \simeq R}$$

S-RECSEQR
$$\frac{S \simeq ([\mu s.R_1/s]R_1);R_2}{S \simeq (\mu s.R_1);R_2}$$

Figure 4.3: Equivalence for context-free session types.

for $\mathsf{SerializeTree} \simeq \mathsf{SerializeTree}'$, where $\mathsf{SerializeTree} = \mu s.\oplus\{\mathsf{Empty}\colon \mathsf{Skip}, \mathsf{Node}\colon \mathsf{!Int};(s;s)\}$ and $\mathsf{SerializeTree}' = \mu r.\oplus\{\mathsf{Empty}\colon \mathsf{Skip}, \mathsf{Node}\colon (\mathsf{!Int};r);r\}$: by applying the rules for recursion, we must eventually derive $\mathsf{SerializeTree};\mathsf{SerializeTree} \simeq \mathsf{SerializeTree}';\mathsf{SerializeTree}$ for which having $\mathsf{SerializeTree} \simeq \mathsf{SerializeTree}'$ as an assumption in the context would be of no avail.

How, then, can we decide whether two types are equivalent? The solution, as initially proposed by Thiemann and Vasconcelos [96] and given an algorithm by Almeida et al. [5], is to look at the *semantics* of session types, rather than their *syntax*. In other words, we must look at the *behaviors* they represent—the interactions they allow—rather than at how they are constructed.

Labelled transition systems (LTSs) are a suitable formalism for this purpose: they succinctly represent discrete systems as a set of states $\mathcal{S}$, together with the set of actions allowed in the system $\mathcal{A}$, and a transition relation $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{A} \times \mathcal{S}$ that maps a state to another through some action. If we interpret session types S as the states of an LTS, then the interactions they permit (e.g., selecting a choice ℓ) correspond the actions a allowed by the system, which can be represented with dedicated symbols (e.g., $\oplus_\ell$). Fig. 4.4 shows the grammar of actions we consider[8]. To define the transition

[8]Letters d, r, p, c in actions stand for "domain", "range", "payload" and "continuation".

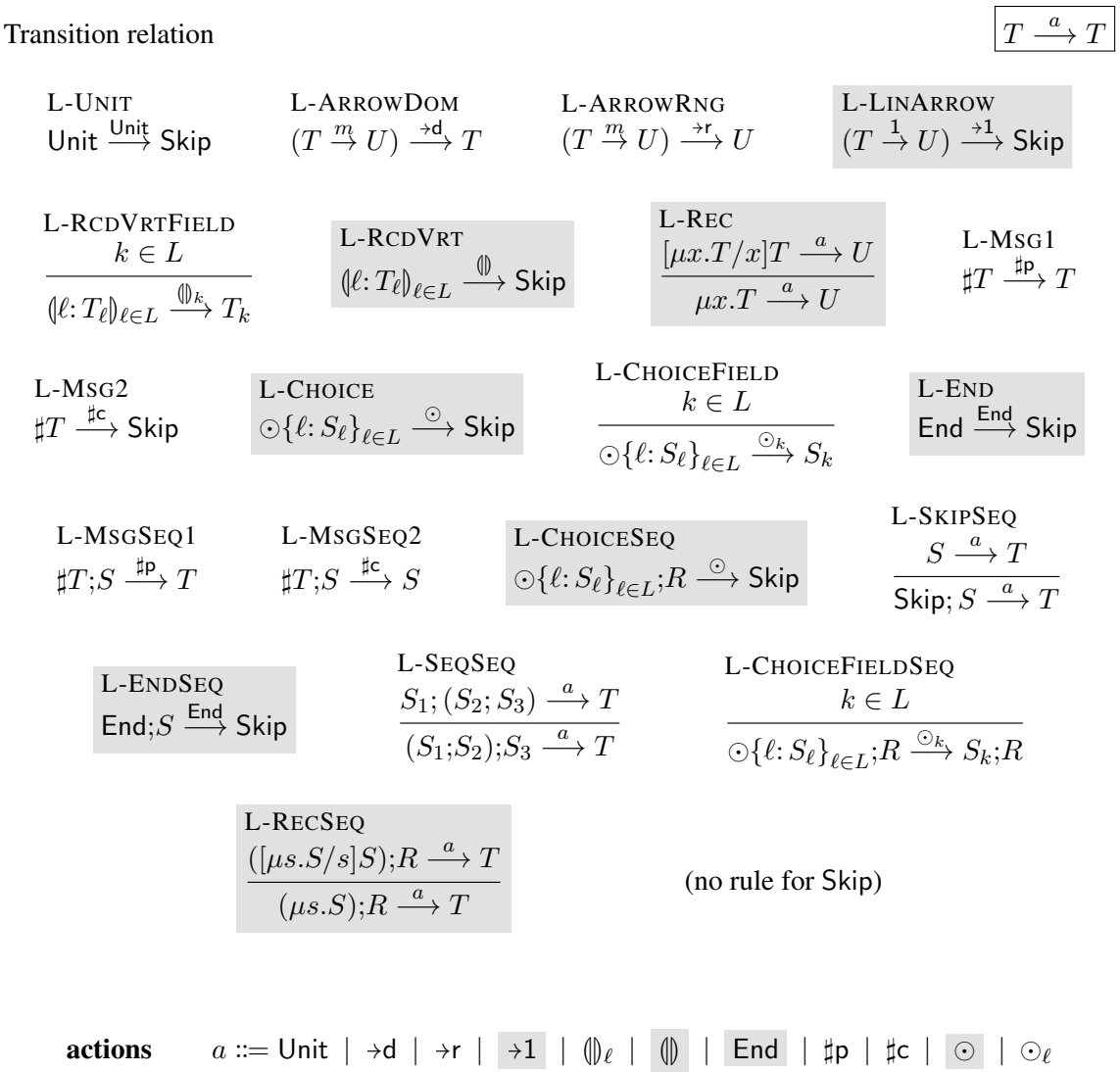


Figure 4.4: Labelled transition system for context-free session types.

relation, which we write as $T \xrightarrow{a} U$, we resort once again to inference rules, also shown in Fig. 4.4. By also attributing "behaviour" to functional types, we allow them to be seamlessly integrated in the semantic equivalence relation.

Note 4.5.1. Besides the adaptations required by our type language (L-LINARROW, L-RCDVRT, L-REC, L-END, L-ENDSEQ and L-RECSEQ, highlighted), which are analogous to those made to the rules in Fig. 4.3, the LTS of Costa et al. requires one correction. By making the behavior of records, variants and choices entirely dependent on their labels, the LTS attributes no behavior to $\{\}$ and $\langle\rangle$, leaving these types undistinguishable from each other and from Skip. To solve this problem, we modify the our LTS to include a default transition for every labelled type (L-RCDVRT, L-CHOICE and L-CHOICESEQ, highlighted).

With the behavior of types established, it becomes straightforward to define an equivalence

relation on them: two types are equivalent if their behavior—the sequences of actions they allow, and their branching points—cannot be distinguished. In the literature, this is known as *bisimilarity* [88], and can be defined (coinductively) as follows.

Definition 4.5.1 (Bisimulation and bisimilarity)**.** A binary relation on types $\mathcal{R}$ is said to be a bisimulation if, whenever $T\ \mathcal{R}\ U$, we have:

1. for each a and T' with $T \xrightarrow{a} T'$, there is U' such that $U \xrightarrow{a} U'$ with $T'\ \mathcal{R}\ U'$;
2. for each a and U' with $U \xrightarrow{a} U'$, there is T' such that $T \xrightarrow{a} T'$ with $T'\ \mathcal{R}\ U'$.

Bisimilarity, written $\sim$, is the union of all bisimulations. We say that a type T is bisimilar to type U if $T \sim U$.

It is straightforward to show that bisimilarity satisfies the properties of reflexivity, symmetry and transitivity expected from an equivalence relation [88]. Furthermore, it can be shown that $\sim$ coincides with $\simeq$, i.e., that $T \sim U \iff T \simeq U$.

This definition tells us that to determine whether types T and U are equivalent, it suffices to show that there exists a bisimulation $\mathcal{R}$ such that $T\ \mathcal{R}\ U$. For example, we can show that $\oplus\{\mathsf{A}\colon \mathsf{Skip}, \mathsf{B}\colon ?\mathsf{Bool}\};\mathsf{End} \simeq \oplus\{\mathsf{A}\colon \mathsf{End}, \mathsf{B}\colon ?\mathsf{Bool};\mathsf{End}\}$ by constructing the set

$$\{(\oplus\{\mathsf{A}\colon \mathsf{Skip}, \mathsf{B}\colon ?\mathsf{Bool}\};\mathsf{End}, \oplus\{\mathsf{A}\colon \mathsf{End}, \mathsf{B}\colon ?\mathsf{Bool};\mathsf{End}\})$$
$$, (\mathsf{Skip};\mathsf{End}, \mathsf{End}), (?\mathsf{Bool};\mathsf{End}, ?\mathsf{Bool};\mathsf{End}), (\mathsf{Bool}, \mathsf{Bool}), (\mathsf{End}, \mathsf{End})\},$$

which can be easily shown to be a bisimulation. Finding such bisimulations algorithmically is easy enough for non-recursive types. However, we once again run into problems when dealing with recursive types: instead of the infinite derivations required to prove the judgements of the syntactic relation, a type equivalence algorithm now needs to find infinite bisimulations, as is the case of that which is needed to show that $\mathsf{SerializeTree} \sim \mathsf{SerializeTree}'$.

How, then, can an algorithm decide whether any two types are bisimilar with a finite amount of time and memory? This is where the simple grammars introduced in the Section 4.3 come in: it is well-known that the bisimilarity is decidable for simple grammars [20] and for the corresponding class of *context-free processes* [20]. These results hinge on the fact that it is possible to represent an infinite bisimulation finitely, by *decomposing* its pairs into "smaller" pairs such that only finitely many indecomposible pairs remain. Exploiting this property of bisimilarity, and building on the work of Jančar and Moller [58], Almeida et al. [5] developed a sound and complete algorithm to decide the bisimilarity of context-free session types. Succinctly, it works by first translating the types to a simple grammar, then pruning it by removing unnecessary words and productions, and finally attempting to build a finite representation of a bisimulation by exploring the transitions of the starting words through an *expansion tree*.

The type bisimilarity algorithm of Almeida et al. is the basis for our subtyping algorithm, which we develop in Chapter 6. Since both algorithms share the same basic structure, we refrain going into details here; we point out the differences between them when introducing ours in Chapter 6.

4.6 Undecidability of subtyping

We are finally ready to tackle our main problem, that of subtyping for context-free session types. We are not, however, the first to approach it: in his work on *type inference* in the presence of context-free session types, Padovani also proposed a subtyping relation, which he used to induce an equivalence relation (cf. Note 3.5.1). This relation is, however, somewhat limited, not accounting for subtyping in messages or, for that matter, functional types. Despite its limitations, this relation was enough for Padovani to obtain a crucial result, one that places a fundamental limitation on the applicability of our work: the undecidability of subtyping, in the classical style of Gay and Hole, for context-free session types. We finish this chapter by briefly reviewing Padovani's subtyping relation and the proof of its undecidability.

Before addressing subtyping, we must point out that Padovani's system does not use explicit syntax (like the μ constructor) for recursive types. Instead, the metavariables for types range over the possibly infinite regular trees generated by the type constructors, and recursive types are introduced as the solutions of finite systems of equations like

$$S_{\mathsf{tree}} = \oplus\{\mathsf{Empty}\colon \mathsf{Skip}, \mathsf{Node}\colon !\mathsf{Int};S_{\mathsf{tree}};S_{\mathsf{tree}}\},$$

where S_{tree} appears unguarded on the left-hand side and guarded by at least one constructor on the right-hand side, guaranteeing exactly one solution. We adopt this method in this section to reproduce the proof of undecidability more faithfully.

Like the semantic equivalence relation in the previous section, Padovani's subtyping relation is also based on an LTS, albeit a different one. As shown in Fig. 4.5, this LTS only distinguishes between four kinds of actions: sending a message of type T ($!T$), receiving a message of type T ($?T$), sending a label ($!k$) and receiving a label ($?k$). Furthermore, in the transitions for message types (LP-MSG), the payload type is part of the transition label itself, and, as such, its behavior is not accounted for in the transition relation. While this greatly simplifies the LTS, it also means that the resulting subtyping relation cannot express the costumary co/contravariance in messages types (cf. Section 3.4)—in fact, this requires the corresponding message types in the subtype and supertype to be not merely equivalent, but *syntactically identical*.[9]

Definition 4.6.1 (Padovani's subtyping relation)**.** Padovani's subtyping relation, written $\lesssim_{\mathrm{P}}$ is defined as the largest binary relation on types such that $S \lesssim_{\mathrm{P}} R$ implies one of the following:

- $S\checkmark$ and $R\checkmark$;
- $S\not\checkmark$, $R\not\checkmark$ and there are no a, S', b, R' such that $S \xrightarrow{a} S'$ and $R \xrightarrow{b} R'$;
- for each ω and each S' with $S \xrightarrow{?\omega} S'$, there is R' such that $R \xrightarrow{?\omega} R'$ and $S' \lesssim_{\mathrm{P}} R'$;
- for each ω and each R' with $R \xrightarrow{!\omega} R'$, there is S' such that $S \xrightarrow{!\omega} S'$ and $S' \lesssim_{\mathrm{P}} R'$.

While $\lesssim_{\mathrm{P}}$ does not allow co/contravariance in messages, it does allow it in internal and external choices. Thus, $S_{\mathsf{tree}} \lesssim_{\mathrm{P}} \oplus\{\mathsf{Empty}\colon \mathsf{Skip}\}$ holds, while $?(S_{\mathsf{tree}}) \lesssim_{\mathrm{P}} ?(\oplus\{\mathsf{Empty}\colon \mathsf{Skip}\})$ does not.

[9] Padovani's work builds on Thiemann and Vasconcelos' original, first-order presentation of context-free session types [96], whose LTS also does not inspect the structure of message types. The later work by Costa et al. on higher-order context-free session types enables this inspection [23].

Transition relation $\boxed{T \xrightarrow{a} T}$

$$\text{LP-Msg}\quad \sharp T \xrightarrow{\sharp T} \mathsf{Skip}$$

$$\text{LP-ExtChoice}\quad \frac{k \in L}{\&\{\ell\colon S_\ell\}_{\ell\in L} \xrightarrow{?k} S_k}$$

$$\text{LP-IntChoice}\quad \frac{k \in L}{\oplus\{\ell\colon S_\ell\}_{\ell\in L} \xrightarrow{!k} S_k}$$

$$\text{LP-Seq1}\quad \frac{S \xrightarrow{a} S'}{S;R \xrightarrow{a} S';R}$$

$$\text{LP-Seq2}\quad \frac{S\checkmark \qquad R \xrightarrow{a} R'}{S;R \xrightarrow{a} R'}$$

(no rule for Skip or End)

$$\begin{array}{rrl} \textbf{message types} & \omega & ::= T \mid \ell \\ \textbf{actions} & a & ::= \sharp\omega \end{array}$$

Figure 4.5: Padovani's labelled transition system for context-free session types.

Despite this limitation (for which we propose a solution in the following chapter), relation $\lesssim_{\mathrm{P}}$ is expressive enough to allow us to formulate the inclusion problem for simple languages in its terms. In other words, we can show that if we are able to decide whether any two session types are related by $\lesssim_{\mathrm{P}}$, then we are also able to decide whether a simple language includes another (i.e., $L(\mathcal{G}_1) \subseteq L(\mathcal{G}_2)$ for simple grammars $\mathcal{G}_1$ and $\mathcal{G}_2$) [10]. Given that, as mentioned above, the inclusion problem for simple languages is undecidable, it follows that relation $\lesssim_{\mathrm{P}}$ (and any relation that includes it) must also be undecidable.

To demonstrate this, Padovani devised a way to encode any simple grammar $\mathcal{G} = (\mathcal{V}, \Sigma, \mathcal{P}, \mathcal{S})$ into a corresponding session type $\mathsf{st}(\mathcal{G})$, such that the language generated by $\mathcal{G}$ coincides with the complete *traces*, or sequences of actions, allowed by the LTS starting at $\mathsf{st}(\mathcal{G})$. Assuming, without loss of generality, that $\mathcal{V} = \{X_1, \ldots, X_n\}$, $\Sigma = \{\ell_1, \ldots, \ell_m\}$, $\mathcal{S} = X_1$, it is possible to define the following finite system of equations:

$$X_i = \&\{\ell\colon Y_1;\ldots;Y_k\}_{X_i \to \ell Y_1 \ldots Y_k \in \mathcal{P}} \qquad (1 \le i \le n)$$

By appealing to Courcelle [24], Padovani deduces that this system must have a unique solution $\{X_1 \to S_1, \ldots, X_n \to S_n\}$ and identifies $\mathsf{st}(\mathcal{G})$ with S_1, the type corresponding to the starting non-terminal of $\mathcal{G}$. From this, it is simple to show that

$$\mathsf{st}(\mathcal{G}) \xrightarrow{?\ell_1} \ldots \xrightarrow{?\ell_k} \mathsf{Skip} \iff \ell_1 \ldots \ell_k \in L(\mathcal{G}).$$

Suppose now that $\mathsf{st}(\mathcal{G}_1) \lesssim_{\mathrm{P}} \mathsf{st}(\mathcal{G}_2)$: since the external choices that form these types are covariant on width, the traces of $\mathsf{st}(\mathcal{G}_1)$ may omit entire branches from those of $\mathsf{st}(\mathcal{G}_2)$. Thus, given the correspondence between session type traces and language words, it is easy to demonstrate that

$$\mathsf{st}(\mathcal{G}_1) \lesssim_{\mathrm{P}} \mathsf{st}(\mathcal{G}_2) \iff L(\mathcal{G}_1) \subseteq L(\mathcal{G}_2),$$

[10] In computability terms, we say that inclusion between simple languages is *reducible* to subtyping for context-free session types.

which completes the proof.

This unfortunate result places a fundamental restriction on any implementation of subtyping (in the classical style of Gay and Hole) for context-free session types, leaving language designers with two options: either restrict the the subtyping relation to a decidable but less expressive approximation, or design an incomplete subtyping algorithm.

In his work, Padovani took the first route, leveraging the subtyping features of the implementation language (OCaml) to provide this approximation. Our goal, however, is to achieve an even more expressive notion of subtyping than Padovani's, one that features co/contravariance in message types while also accommodating functional types. Our only choice, then, is to follow alternate path. Our journey begins in the next chapter, in which we develop our subtyping relation.

4.7 Related work

Context-free session types have seen considerable development since their introduction, most notably their integration in System F [2, 86], an higher-order formulation [23], as well as proposals for kind and type inference [3, 82]. To the best of our knowledge, the only work on subtyping for these types before our work is that of Padovani [82].

Much of the work on equivalence for these types rests on results from the theories of formal languages, automata and process algebra. On these topics, several algorithms have been proposed to check the equivalence of their central objects of study. On the topic of automata, Henry and Sénizergues [48] proposed an algorithm to decide the language equivalence problem on deterministic pushdown automata (which recognize deterministic context-free languages). On the related topic of basic process algebra (BPA), BPA processes have been shown to be equivalent to grammars in GNF [9], of which simple grammars are a particular case. This makes results and algorithms for BPA processes applicable to grammars in GNF, and vice-versa. A bisimilarity algorithm for general BPA processes, of doubly-exponential complexity, has been proposed by Burkart et al. [15], while an analogous polynomial-time algorithm for the special case of normed BPA processes has been proposed by Hirschfield et al. [49].

Context-free session types are not the only formulation of session types that go beyond the regular realm. Das et al. [27] introduce nested session types, which extend regular session types with parameterized type definitions, resulting in a system that is *strictly* more expressive than context-free session types. Like their context-free counterparts, nested session types exhibit a decidable equivalence problem and an undecidable subtyping problem, for which there is a sound but incomplete algorithm [28].

Chapter 5

Subtyping context-free session types

Equipped with an understanding of typing and subtyping in general, as well as context-free session types in particular, we are now ready to present our contributions, which will be the focus of the remaining chapters.

Besides this thesis, our work has resulted in two publications: a conference paper [90] (accompanied by a technical report [91]), presented at CONCUR 2023, the 34th International Conference on Concurrency Theory in Antwerp, Belgium, and an extended abstract [92], presented at INForum 2023, Simpósio de Informática in Porto, Portugal.

In this chapter, we begin by introducing our two coinciding notions of syntactic and semantic subtyping for context-free session types, the latter of which we use as a stepping stone to develop a subtyping algorithm in the next chapter.

5.1 A syntactic subtyping relation

Following the previous chapter, our first attempt at formalizing a notion of subtyping for context-free session types takes a syntactic approach. We characterize syntactic subtyping as a relation defined by a collection of inference rules with a coinductive interpretation, shown in Figure 5.1. We obtain these rules by modifying the equivalence rules of Costa et al. [23], previously shown in Fig. 4.3 and already adapted to account for multiplicity-annotated functions, the explicit recursion operator μ and the left-absorbing End type that were not present in their type language.

More concretely, besides renaming the rules and denoting the new relation with the $\leq$ symbol, we replace E-Arrow with S-Arrow (allowing multiplicity subtyping), E-RcdVrt with S-Rcd and S-Vrt, E-Msg with S-In and S-Out, and E-Choice with S-ExtChoice and S-IntChoice, establishing the classical subtyping properties associated with both functional types and session types we saw throughout Sections 2.3 and 3.4. Additionally, we replace E-MsgSeq1L with S-InSeq1L and S-OutSeq1L, E-MsgSeq1R with S-InSeq1R and S-OutSeq1R, and finally E-MsgSeq2 with S-InSeq2 and S-OutSeq2, thus ensuring that subtyping is compatible the identity, associativity, distributivity and absorption properties of the ; operator.

It is easy to see how these changes do not preserve the symmetry of the original relation, but a coinductive argument shows that both reflexivity and transitivity are maintained.

Syntactic subtyping (*coinductive*) $\boxed{T \leq T}$

$$\textsc{S-Unit}\ \ \mathsf{Unit} \leq \mathsf{Unit} \qquad \textsc{S-Arrow}\ \ \frac{U_1 \leq T_1 \qquad T_2 \leq U_2 \qquad m \sqsubseteq n}{T_1 \xrightarrow{m} T_2 \leq U_1 \xrightarrow{n} U_2} \qquad \textsc{S-Rcd}\ \ \frac{K \subseteq L \qquad T_j \leq U_j\ (\forall j \in K)}{\{\ell\colon T_\ell\}_{\ell \in L} \leq \{k\colon U_k\}_{k \in K}}$$

$$\textsc{S-Vrt}\ \ \frac{L \subseteq K \qquad T_j \leq U_j\ (\forall j \in L)}{\langle \ell\colon T_\ell \rangle_{\ell \in L} \leq \langle k\colon U_k \rangle_{k \in K}} \qquad \textsc{S-RecL}\ \ \frac{[\mu x.T/x]T \leq U}{\mu x.T \leq U} \qquad \textsc{S-RecR}\ \ \frac{T \leq [\mu x.U/x]U}{T \leq \mu x.U} \qquad \textsc{S-In}\ \ \frac{T \leq U}{?T \leq ?U}$$

$$\textsc{S-Out}\ \ \frac{U \leq T}{!T \leq !U} \qquad \textsc{S-ExtChoice}\ \ \frac{L \subseteq K \qquad S_j \leq R_j\ (\forall j \in L)}{\&\{\ell\colon S_\ell\}_{\ell \in L} \leq \&\{k\colon R_k\}_{k \in K}} \qquad \textsc{S-IntChoice}\ \ \frac{K \subseteq L \qquad S_j \leq R_j\ (\forall j \in K)}{\oplus\{\ell\colon S_\ell\}_{\ell \in L} \leq \oplus\{k\colon R_k\}_{k \in K}}$$

$$\textsc{S-Skip}\ \ \mathsf{Skip} \leq \mathsf{Skip} \qquad \textsc{S-End}\ \ \mathsf{End} \leq \mathsf{End} \qquad \textsc{S-InSeq1L}\ \ \frac{T \leq U \qquad S \leq \mathsf{Skip}}{?T;S \leq ?U} \qquad \textsc{S-InSeq1R}\ \ \frac{T \leq U \qquad S \leq \mathsf{Skip}}{?T \leq ?U;S}$$

$$\textsc{S-InSeq2}\ \ \frac{T \leq U \qquad S \leq R}{?T;S \leq ?U;R} \qquad \textsc{S-OutSeq1L}\ \ \frac{U \leq T \qquad S \leq \mathsf{Skip}}{!T;S \leq !U} \qquad \textsc{S-OutSeq1R}\ \ \frac{U \leq T \qquad S \leq \mathsf{Skip}}{!T \leq !U;S}$$

$$\textsc{S-OutSeq2}\ \ \frac{U \leq T \qquad S \leq R}{!T;S \leq !U;R} \qquad \textsc{S-ChoiceSeqL}\ \ \frac{\odot\{\ell\colon S_\ell;S\}_{\ell \in L} \leq R}{\odot\{\ell\colon S_\ell\}_{\ell \in L};S \leq R} \qquad \textsc{S-ChoiceSeqR}\ \ \frac{S \leq \odot\{\ell\colon R_\ell;R\}_{\ell \in L}}{S \leq \odot\{\ell\colon R_\ell\}_{\ell \in L};R} \qquad \textsc{S-SkipSeqL}\ \ \frac{S \leq R}{\mathsf{Skip};S \leq R}$$

$$\textsc{S-SkipSeqR}\ \ \frac{S \leq R}{S \leq \mathsf{Skip};R} \qquad \textsc{S-EndSeq1L}\ \ \mathsf{End};S \leq \mathsf{End} \qquad \textsc{S-EndSeq1R}\ \ \mathsf{End} \leq \mathsf{End};R \qquad \textsc{S-EndSeq2}\ \ \mathsf{End};S \leq \mathsf{End};R \qquad \textsc{S-SeqSeqL}\ \ \frac{S_1;(S_2;S_3) \leq R}{(S_1;S_2);S_3 \leq R}$$

$$\textsc{S-SeqSeqR}\ \ \frac{S \leq R_1;(R_2;R_3)}{S \leq (R_1;R_2);R_3} \qquad \textsc{S-RecSeqL}\ \ \frac{([\mu s.S_1/s]S_1);S_2 \leq R}{(\mu s.S_1);S_2 \leq R} \qquad \textsc{S-RecSeqR}\ \ \frac{S \leq ([\mu s.R_1/s]R_1);R_2}{S \leq (\mu s.R_1);R_2}$$

Preorder on multiplicities $\boxed{m \sqsubseteq m}$

$$m \sqsubseteq m \qquad * \sqsubseteq 1$$

Figure 5.1: Syntactic subtyping for context-free session types.

Theorem 5.1.1. *The syntactic subtyping relation $\leq$ is a preorder on types.*

The details of the proof can be consulted in the appendices: Appendix A lays the groundwork (for this, as well as most subsequent proofs), while Appendix B presents the argument itself.

Example 5.1.1. Recall the type $\mathsf{SerializeTree} = \mu s.\oplus\{\mathsf{Empty}\colon \mathsf{Skip}, \mathsf{Node}\colon !\mathsf{Int};s;s\}$, which describes the serialization of an arbitrary binary tree. Suppose now the following types, which

describe the serialization of full trees of successively greater heights:

$$\begin{aligned}
\mathsf{SerializeEmpty} &= \oplus\{\mathsf{Nil}\colon \mathsf{Skip}\} \\
\mathsf{SerializeFullTree}_0 &= \oplus\{\mathsf{Node}\colon \mathsf{SerializeEmpty};!\mathsf{Int};\mathsf{SerializeEmpty}\} \\
\mathsf{SerializeFullTree}_1 &= \oplus\{\mathsf{Node}\colon \mathsf{SerializeFullTree}_0;!\mathsf{Int};\mathsf{SerializeFullTree}_0\} \\
&\vdots \\
\mathsf{SerializeFullTree}_n &= \oplus\{\mathsf{Node}\colon \mathsf{SerializeFullTree}_{n-1};!\mathsf{Int};\mathsf{SerializeFullTree}_{n-1}\}
\end{aligned}$$

By the principle of safe substitution, SerializeTree should be a subtype of SerializeEmpty and $\mathsf{SerializeTree}_n$ for any n—after all, the ability to serialize an arbitrary tree presupposes the ability to serialize the empty tree, as well as full trees of any height. To confirm that $\mathsf{SerializeFullTree} \leq \mathsf{SerializeFullTree}_n$ holds, for any n, we unfold the left-hand side using S-RECL and apply S-INTCHOICE. Then we apply the distributivity rules S-CHOICESEQL and S-CHOICESEQR as necessary, until reaching a judgement of the form $\oplus\{\ell\colon S_\ell\}_{\ell\in L} \leq \oplus\{k\colon R_k\}_{k\in K}$, at which point we can apply S-INTCHOICE again, or until reaching a type with !Int at the head, at which point we apply S-INSEQ2. We repeat this process until reaching $\mathsf{SerializeTree} \leq \mathsf{SerializeFullTree}_{n-1}$, and proceed similarly until reaching $\mathsf{SerializeTree} \leq \mathsf{SerializeEmpty}$, which follows from S-INTCHOICE and S-SKIP.

Despite offering a clear presentation of the features of the subtyping relation, our syntactic rules, like those of the equivalence relation upon which they are based, suggest no obvious algorithmic intepretation: on the one hand, the bare metavariables in rules S-RECL, S-CHOICESEQL, S-RECSEQL and their right-hand counterparts makes the system not syntax-directed; on the other hand, rules S-RECL, S-RECSEQL and their right-hand counterparts lead to infinite derivations which are not solvable by conventional means. Following the work on type equivalence, the next section explores an alternative semantic approach, which we then use as a stepping stone to develop our subtyping algorithm.

5.2 A semantic subtyping relation

As mentioned in the previous chapter, semantic equivalence for context-free session types is usually based on *observational equivalence* or *bisimilarity*, meaning that two session types are considered equivalent if they exhibit exactly the same communication behaviour [96]. An analogous notion of semantic subtyping should therefore rely on an *observational preorder*, where subtypes may omit certain behaviors or exhibit additional ones when compared to the supertype. We now look for such a preorder, taking our adaptation of the LTS of Costa et al., previously shown in Fig. 4.4, as the basis for the behavior of types.

Several notions of observational preorder have been explored in the literature, with studies dating back from the 70s (Sangiorgi provides a gentle introduction to the topic [88]). The simplest of these notions is, arguably, *similarity* [69, 83]. Its definition resembles that of bisimilarity, but with a single clause.

Definition 5.2.1. A type relation $\mathcal{R}$ is said to be a simulation if, whenever $T\mathcal{R}U$, for all a and T' with $T \xrightarrow{a} T'$, there is U' such that $U \xrightarrow{a} U'$ and $T'\mathcal{R}U'$

Similarity, written $\preceq$, is the union of all simulation relations. We say that a type U simulates type T if $T \preceq U$.

Unfortunately, plain similarity cannot be identified with semantic subtyping. A small example demonstrates why: type $\oplus\{\mathsf{A}\colon \mathsf{End}, \mathsf{B}\colon \mathsf{End}\}$ both simulates and is a subtype of $\oplus\{\mathsf{A}\colon \mathsf{End}\}$, while type $\&\{\mathsf{A}\colon \mathsf{End}\}$ does not simulate yet is a subtype of $\&\{\mathsf{A}\colon \mathsf{End}, \mathsf{B}\colon \mathsf{End}\}$. An "anti-simulation" relation, based on the converse clause, would be of no use either, as it would simply leave us with the converse problem.

It is apparent that a more refined notion of simulation is necessary, where the direction of the implication depends on the transition labels. Fortunately, Aarts and Vaandrager provide just such a notion in the form of $\mathcal{XY}$*-simulation* [1], a simulation relation parameterized by two subsets of actions, $\mathcal{X}$ and $\mathcal{Y}$, such that actions in $\mathcal{X}$ are simulated from left to right and those in $\mathcal{Y}$ are simulated from right to left, selectively combining the requirements of simulation and reverse simulation.

Definition 5.2.2. Let $\mathcal{X}, \mathcal{Y} \subseteq \mathcal{A}$. A type relation $\mathcal{R}$ is said to be an $\mathcal{XY}$-simulation if, whenever $T\mathcal{R}U$, we have:

1. for each $a \in \mathcal{X}$ and each T' with $T \xrightarrow{a} T'$, there is U' such that $U \xrightarrow{a} U'$ with $T'\mathcal{R}U'$;
2. for each $a \in \mathcal{Y}$ and each U' with $U \xrightarrow{a} U'$, there is T' such that $T \xrightarrow{a} T'$ with $T'\mathcal{R}U'$.

$\mathcal{XY}$-similarity, written $\preceq^{\mathcal{XY}}$, is the union of all $\mathcal{XY}$-simulation relations. We say that a type T is $\mathcal{XY}$-similar to type U if $T \preceq^{\mathcal{XY}} U$.

Similar or equivalent notions have appeared throughout the literature: *modal refinement* [64], *alternating simulation* [6] and, perhaps more appropriately named (for our purposes), *covariant-contravariant simulation* [35]. Padovani's original subtyping relation for first-order context-free session types [82], seen in the previous chapter, can also be understood as a refined form of $\mathcal{XY}$-simulation.

We can tentatively define a semantic subtyping relation $\lesssim'$ as $\mathcal{XY}$-similarity, where $\mathcal{X}$ and $\mathcal{Y}$ are the label sets generated by the following grammars for $a_{\mathcal{X}}$ and $a_{\mathcal{Y}}$, respectively.

$$a_{\mathcal{X}} ::= a_{\mathcal{XY}} \mid \langle\rangle_\ell \mid \&_\ell \qquad a_{\mathcal{XY}} ::= \mathsf{Unit} \mid \to\mathsf{d} \mid \to\mathsf{r} \mid (\!|\,|\!) \mid \sharp\mathsf{p} \mid \sharp\mathsf{c} \mid \odot \mid \mathsf{End}$$
$$a_{\mathcal{Y}} ::= a_{\mathcal{XY}} \mid \to\mathsf{1} \mid \{\}_\ell \mid \oplus_\ell$$

This would indeed give us the desired result for our previous example, but we still cannot account for the contravariance of output and function types: we want $T = !\{\mathsf{A}\colon \mathsf{Int}\}$ to be a subtype of $U = !\{\mathsf{A}\colon \mathsf{Int}, \mathsf{B}\colon \mathsf{Bool}\}$, yet $T \lesssim' U$ does not hold (in fact, we have $U \lesssim' T$, a clear violation of run-time safety). The same could be said for types $\{\mathsf{A}\colon \mathsf{Int}\} \xrightarrow{*} \mathsf{Int}$ and $\{\mathsf{A}\colon \mathsf{Int}, \mathsf{B}\colon \mathsf{Bool}\} \xrightarrow{*} \mathsf{Int}$. In short, our simulation needs the $!\mathsf{p}$ and $\to\mathsf{d}$-derivatives to be related in the direction opposite to that of the types they derive from. Thus we need to selectively apply a *strong* form of *contrasimulation* as well [88, 99] (the original notion is defined with *weak transitions*, which are not present in our LTS).

To allow this inversion, we generalize the definition of $\mathcal{XY}$-simulation by parameterizing it on two further subsets of actions, and including two more clauses where the direction of the relation between the derivatives is reversed. By analogy with $\mathcal{XY}$-simulation, we call the resulting notion $\mathcal{XYZW}$-simulation.

Definition 5.2.3. Let $\mathcal{X}, \mathcal{Y}, \mathcal{Z}, \mathcal{W} \subseteq \mathcal{A}$. A type relation $\mathcal{R}$ is a $\mathcal{XYZW}$-simulation if, whenever $T\mathcal{R}U$, we have:

1. *for each $a \in \mathcal{X}$ and each T' with $T \xrightarrow{a} T'$, there is U' such that $U \xrightarrow{a} U'$ with $T'\mathcal{R}U'$;*
2. *for each $a \in \mathcal{Y}$ and each U' with $U \xrightarrow{a} U'$, there is T' such that $T \xrightarrow{a} T'$ with $T'\mathcal{R}U'$;*
3. *for each $a \in \mathcal{Z}$ and each T' with $T \xrightarrow{a} T'$, there is U' such that $U \xrightarrow{a} U'$ with $U'\mathcal{R}T'$;*
4. *for each $a \in \mathcal{W}$ and each U' with $U \xrightarrow{a} U'$, there is T' such that $T \xrightarrow{a} T'$ with $U'\mathcal{R}T'$.*

$\mathcal{XYZW}$-similarity, written $\preceq^{\mathcal{XYZW}}$, is the union of all $\mathcal{XYZW}$-simulation relations. We say that a type T is $\mathcal{XYZW}$-similar to type U if $T \preceq^{\mathcal{XYZW}} U$.

Note that since $\mathcal{XYZW}$-simulation is a generalization of $\mathcal{XY}$-simulation, it is also a generalization of bisimulation and plain simulation: $\mathcal{XY}$-simulation can be seen as an $\mathcal{XY}\emptyset\emptyset$-simulation, bisimulation as an $\mathcal{A}\mathcal{A}\emptyset\emptyset$-simulation (alternatively, $\emptyset\emptyset\mathcal{A}\mathcal{A}$-simulation or $\mathcal{A}\mathcal{A}\mathcal{A}\mathcal{A}$-simulation), and plain simulation as an $\mathcal{A}\emptyset\emptyset\emptyset$-simulation.

Our semantic subtyping relation must be a preorder. The following theorem states that $\mathcal{XYZW}$-similarity satisfies this property regardless of its parameters. Its proof can be found in Appendix C.

Theorem 5.2.1. *For any $\mathcal{X}, \mathcal{Y}, \mathcal{Z}, \mathcal{W}$, $\preceq^{\mathcal{XYZW}}$ is a preorder relation on types.*

Equipped with the notion of $\mathcal{XYZW}$-similarity, we are ready to define our semantic subtyping relation for functional and higher-order context-free session types.

Definition 5.2.4. The semantic subtyping relation for functional and higher-order context-free session types $\lesssim$ is defined by $T \lesssim U$ when $T \preceq^{\mathcal{XYZW}} U$ such that $\mathcal{X}$, $\mathcal{Y}$, $\mathcal{Z}$ and $\mathcal{W}$ are defined as the label sets generated by the following grammars for $a_{\mathcal{X}}$, $a_{\mathcal{Y}}$, $a_{\mathcal{Z}}$ and $a_{\mathcal{W}}$, respectively.

$$a_{\mathcal{X}} ::= a_{\mathcal{XY}} \mid \twoheadrightarrow\mathsf{1} \mid \langle\rangle_\ell \mid \&_\ell \qquad a_{\mathcal{Z}}, a_{\mathcal{W}} ::= !\mathsf{p} \mid \twoheadrightarrow\mathsf{d}$$

$$a_{\mathcal{Y}} ::= a_{\mathcal{XY}} \mid \{\}_\ell \mid \oplus_\ell \qquad a_{\mathcal{XY}} ::= \mathsf{Unit} \mid \twoheadrightarrow\mathsf{r} \mid (\!|) \mid ?\mathsf{p} \mid \sharp\mathsf{c} \mid \odot \mid \mathsf{End}$$

Notice the correspondence between the placement of the labels and the variance of their respective type constructors. Labels arising from covariant positions of the arrow and input type constructors are placed in both the $\mathcal{X}$ and $\mathcal{Y}$ sets, while those arising from the contravariant positions of the arrow and output type constructors are placed in both the $\mathcal{Z}$ and $\mathcal{W}$ sets. Labels arising from the fields of constructors exhibiting width subtyping are placed in a single set, depending on the variance of the constructor on the label set: $\mathcal{X}$ for covariance (external choice and variant constructors), $\mathcal{Y}$ for contravariance (internal choice and record constructors). The function type constructor is covariant on its multiplicity, thus the linear arrow label is placed in $\mathcal{X}$. Finally, checkmark labels and those arising from nullary constructors are placed in $\mathcal{X}$ and $\mathcal{Y}$, but they

could alternatively be placed in $\mathcal{Z}$ and $\mathcal{W}$ or in all four sets (notice the parallel with bisimulation, that can be defined as $\mathcal{AA}\emptyset\emptyset$-simulation, $\emptyset\emptyset\mathcal{AA}$-simulation, or $\mathcal{AAAA}$-simulation).

Example 5.2.1. Recall the tree serialization types from Example 5.1.1. Here it is also easy to see that $\mathsf{SerializeTree} \lesssim \mathsf{SerializeFullTree}_n$ for any n. Observe that, on the side of SerializeTree, transitions by $\oplus_{\mathsf{Nil}}$ and $\oplus_{\mathsf{Node}}$ always appear together, while, on the side of $\mathsf{SerializeTree}_n$, may transition by $\oplus_{\mathsf{Node}}$ or $\oplus_{\mathsf{Nil}}$, but never both simultaneously. Since $\oplus_{\mathsf{Node}}$ and $\oplus_{\mathsf{Nil}}$ belong exclusively to $\mathcal{Y}$, the left-hand side (beginning with SerializeTree) is always able to match the right-hand side (beginning with $\mathsf{SerializeFullTree}_n$) on these labels (as well as on all the others in $\mathcal{Y} \cup \mathcal{W}$, and vice-versa for $\mathcal{X} \cup \mathcal{Z}$).

Before moving on to our algorithm, we present a result that states that the syntactic and the semantic subtyping relation coincide. In other words, any result about one relation can be transferred to the other. As hinted above, the semantic relation is more amenable to an algorithmic treatment than the syntactic one, which, in turn, is easier to grasp. If our algorithm turns out to be sound with respect to $\lesssim$, Theorem 5.2.2 will assure us that the definition of $\leq$ is still an accurate specification.

Theorem 5.2.2 (Soundness and completeness for subtyping relations)**.** *Let* $\vdash T$ *and* $\vdash U$*. Then* $T \leq U$ *iff* $T \lesssim U$*.*

The details of the proof can be found in Appendix D.

Chapter 6

A subtyping algorithm for context-free session types

The notion of subtyping we have outlined is undecidable. This follows from the fact that our type language, albeit slightly different, contains all the features necessary to reconstruct Padovani's proof of undecidability [82] which we replicated in Section 4.6.

Despite this negative result, we are still able to devise a sound (but necessarily incomplete) algorithm for it. In other words, we are able to guarantee that it returns no false positives: if the algorithm returns **True** on input (T, U), then $T \lesssim U$ really holds.

The algorithm is an adaptation of the equivalence algorithm of Almeida et al. [5]. At its core, it determines the $\mathcal{XYZW}$-similarity of simple grammars. Its application to context-free session types is facilitated by a translation function to properly encode types as grammars. By providing an appropriate encoding for other kinds of objects, the algorithm may likewise be adapted to other domains.

Much like the original, our algorithm can be succinctly described in three distinct phases:

1. translate the given types to a simple grammar and two starting words;
2. prune unreachable symbols from productions;
3. explore an expansion tree rooted at a node containing the initial words, alternating between expansion and simplification operations until either an empty node is found (decide **True**) or all nodes fail to expand (decide **False**).

6.1 Translating types to grammars

As we have seen, context-free session types enjoy a tight correspondence with simple grammars, i.e., deterministic grammars in Greibach normal form (GNF). By their more abstract and uniform nature, grammars are more amenable to an algorithmic treatment than context-free session types. The first step in our algorithm, then, is to convert a pair of types into a simple grammar—more concretely, into set of productions $\mathcal{P}$ and two *words* $(\vec{X}, \vec{Y})$, which we define as sequences of non-terminal symbols (each non-terminal symbol corresponds roughly to a type in a sequential composition).

We can check the bisimilarity and $\mathcal{XYZW}$-similarity of words in a GNF grammar with productions $\mathcal{P}$ because it naturally induces a labelled transition system, where states are words $\vec{X}$, actions are terminal symbols a and the transition relation is defined as $X\vec{Y} \xrightarrow{a}_{\mathcal{P}} \vec{Z}\vec{Y}$ when $X \to a\vec{Z} \in \mathcal{P}$. We denote the bisimilarity and $\mathcal{XYZW}$-similarity of grammars by, respectively, $\sim_{\mathcal{P}}$ and $\preceq_{\mathcal{P}}^{\mathcal{XYZW}}$, where $\mathcal{P}$ is the set of productions. We also let $\lesssim_{\mathcal{P}}$ denote grammar $\mathcal{XYZW}$-similarity with label sets as in Definition 5.2.4.

Grammar translation is done by procedure grm (Definition 6.1.3), which remains unchanged from the original algorithm [5], but relies on two auxiliary definitions which must be adapted: the unr function (Definition 6.1.1), which normalizes the head of session types and unravels recursive types until reaching a type constructor, and the word procedure (Definition 6.1.2), which builds a word from a session type while updating a set $\mathcal{P}$ of productions. Self-references are used as non-terminal symbols, and we assume a non-terminal symbol $\bot$ with no productions [5]. Our adaptations to both functions are highlighted, and concern the inclusion of the End type and the default LTS transitions for records/variants and choices (originating from rules L-RCDVRT and L-CHOICE in Fig. 4.4), which are necessary to distinguish their empty forms from each other and from Skip.

Definition 6.1.1. The *unraveling* of a type T is defined by induction on the structure of T:

$$\mathsf{unr}(\mu x.T) = \mathsf{unr}([\mu x.T/x]T) \qquad \mathsf{unr}(\mathsf{Skip};S) = \mathsf{unr}(S)$$
$$\mathsf{unr}(\mathsf{End};S) = \mathsf{End} \qquad \mathsf{unr}((\mu s.S);R) = \mathsf{unr}(([\mu s.S/s]S);R)$$
$$\mathsf{unr}(\odot\{\ell\colon S_\ell\}_{\ell\in L};R) = \odot\{\ell\colon S_\ell; R\}_{\ell\in L} \qquad \mathsf{unr}((S_1;S_2);S_3) = \mathsf{unr}(S_1;(S_2;S_3))$$

and in all other cases by $\mathsf{unr}(T) = T$.

Definition 6.1.2. The word corresponding to a well-formed type T, $\mathsf{word}(T)$, is built by descending on the structure of T while updating a set $\mathcal{P}$ of productions:

$$\begin{aligned}
\mathsf{word}(\mathsf{Unit}) &= Y, \text{ setting } \mathcal{P} \coloneqq \mathcal{P} \cup \{Y \to \mathsf{Unit}\}\\
\mathsf{word}(U \xrightarrow{1} V) &= Y, \text{ setting } \mathcal{P} \coloneqq \mathcal{P} \cup \{Y \to \to\mathsf{d}\,\mathsf{word}(U), Y \to \to\mathsf{r}\,\mathsf{word}(V), Y \to \to\mathsf{1}\}\\
\mathsf{word}(U \xrightarrow{*} V) &= Y, \text{ setting } \mathcal{P} \coloneqq \mathcal{P} \cup \{Y \to \to\mathsf{d}\,\mathsf{word}(U), Y \to \to\mathsf{r}\,\mathsf{word}(V)\}\\
\mathsf{word}(\langle\!|\ell\colon T_\ell|\!\rangle_{\ell\in L}) &= Y, \text{ setting } \mathcal{P} \coloneqq \mathcal{P} \cup \{Y \to \langle\!|\bot|\!\rangle\} \cup \{Y \to \langle\!|_k\,\mathsf{word}(T_k) \mid k \in L\}\\
\mathsf{word}(\mathsf{Skip}) &= \varepsilon\\
\mathsf{word}(\mathsf{End}) &= Y, \text{ setting } \mathcal{P} \coloneqq \mathcal{P} \cup \{Y \to \mathsf{End}\bot\}\\
\mathsf{word}(\sharp U) &= Y, \text{ setting } \mathcal{P} \coloneqq \mathcal{P} \cup \{Y \to \sharp\mathsf{p}\,\mathsf{word}(U)\bot, Y \to \sharp\mathsf{c}\}\\
\mathsf{word}(\odot\{\ell\colon S_\ell\}_{\ell\in L}) &= Y, \text{ setting } \mathcal{P} \coloneqq \mathcal{P} \cup \{Y \to \odot\bot\} \cup \{Y \to \odot_k\mathsf{word}(S_k) \mid k \in L\}\\
\mathsf{word}(S_1;S_2) &= \mathsf{word}(S_1)\,\mathsf{word}(S_2)\\
\mathsf{word}(\mu x.U) &= X
\end{aligned}$$

where, in each equation, Y is understood as a fresh non-terminal symbol, X as the non-terminal symbol corresponding to type reference x, and $\bot$ as a non-terminal symbol without productions.

In addition, we must identify the unrolled versions of all the μ-subterms in the type T we want to translate to a grammar. Assume $\{\mu X_1.T_1, ..., \mu X_n.T_n\}$ is the set of such subterms, and that they are topologically sorted with respect to their lexical nesting, innermost first (that is, $i < j$ whenever $X_j \in \mathit{free}(\mu X_i.T_i)$). The unrolled versions of such subterms are as follows.

$$\begin{aligned} T_1' &= [\mu x_n.T_n/x_n]...[\mu x_2.T_2/x_2][\mu x_1.T_1/x_1]T_1 \\ T_2' &= [\mu x_n.T_n/x_n]...[\mu x_2.T_2/x_2]T_2 \\ &\vdots \\ T_n' &= [\mu x_n.T_n/x_n]T_n \end{aligned}$$

We are now able to define the type-to-grammar translation function grm as follows.

Definition 6.1.3. Given an initial set of productions $\mathcal{P}_0$, the grammar corresponding to a type T is given by function grm, defined as:

$$\mathsf{grm}(T, \mathcal{P}_0) = (\mathsf{word}(T), \mathcal{P}_n)$$

where each $\mathcal{P}_i$ is computed from $\mathcal{P}_{i-1}$ by the following recurrence.

$$\mathcal{P}_i' \cup \{X_i \to a_j \vec{Y_j} \vec{Z} \mid (Z \to a_j \vec{Y_j}) \in \mathcal{P}_i', \text{where } (Z\vec{Z}, \mathcal{P}_i') = \mathsf{grm}(\mathit{unr}(T_i'), \mathcal{P}_{i-1})\}$$

where each T_i' is the unrolled version of the ith μ-subterm in T, topologically sorted with respect to lexical nesting (innermost first).

To obtain two initial words $\vec{X}$ and $\vec{Y}$ and the set of productions $\mathcal{P}$ corresponding to two well-formed types T and U, proceed as follows: rename U so that its bound type references do not overlap with those of T, run $\mathsf{grm}(T, \emptyset)$ to obtain $(\vec{X}, \mathcal{P}')$ and, finally, run $\mathsf{grm}(U, \mathcal{P}')$ to obtain $(\vec{Y}, \mathcal{P})$.

Example 6.1.1. Consider again the types for tree serialization introduced in Example 5.1.1. Suppose we want to know whether $\mathsf{SerializeFullTree}_0 \xrightarrow{*} \mathsf{Unit} \lesssim \mathsf{STree} \xrightarrow{1} \mathsf{Unit}$. Applying the procedure we just described, the productions generated for these types are as follows, with X_0 and Y_0 as the starting words.

$X_0 \to \to\mathsf{d}X_1$	$X_2 \to \oplus_{\mathsf{Empty}}$	$X_4 \to \mathsf{Int}$	$Y_0 \to \to\mathsf{d}Y_1$	$Y_1 \to \oplus\bot$
$X_0 \to \to\mathsf{r}X_5$	$X_2 \to \oplus\bot$	$X_5 \to \mathsf{Unit}$	$Y_0 \to \to\mathsf{r}X_5$	$Y_1 \to \oplus_{\mathsf{Empty}}$
$X_1 \to \oplus_{\mathsf{Node}}X_2X_3X_2$	$X_3 \to \mathsf{!p}X_4\bot$		$Y_0 \to \to\mathsf{1}$	$Y_1 \to \oplus_{\mathsf{Node}}Y_1X_3Y_1$
$X_1 \to \oplus\bot$	$X_3 \to \mathsf{!c}$			

The following result states that procedure grm preserves the semantic subtyping relation, which establishes the soundness of the first step of algorithm. Its proof can be found in Appendix E.

Theorem 6.1.1 (Soundness for grammars)**.** *Let* $\vdash T$, $\vdash U$, $(\vec{X}, \mathcal{P}') = \mathsf{grm}(T, \emptyset)$ *and* $(\vec{Y}, \mathcal{P}) = \mathsf{grm}(U, \mathcal{P}')$. *If* $\vec{X} \lesssim_{\mathcal{P}} \vec{Y}$, *then* $T \lesssim U$.

6.2 Pruning unreachable symbols

The grammars generated by procedure grm may contain unreachable words, which can be ignored by the algorithm. Intuitively, these words correspond to communication actions that cannot be fulfilled, such as subterm ?Bool in type $(\mu s.!\mathsf{Int};s);?\mathsf{Bool}$. Formally, these words appear in productions following what are known as *unnormed words*.

Definition 6.2.1. Let $\vec{a}$ be a non-empty sequence of non-terminal symbols $a_1, \ldots, a_n$. Write $\vec{Y} \xrightarrow{\vec{a}}_{\mathcal{P}} \vec{Z}$ when $\vec{Y} \xrightarrow{a_1}_{\mathcal{P}} \ldots \xrightarrow{a_n}_{\mathcal{P}} \vec{Z}$. We say that a word $\vec{Y}$ is *normed* if for some $\vec{a}$ we have $\vec{Y} \xrightarrow{\vec{a}}_{\mathcal{P}} \varepsilon$, and *unnormed* otherwise. If $\vec{Y}$ is normed and $\vec{a}$ is the shortest path such that $\vec{Y} \xrightarrow{\vec{a}}_{\mathcal{P}} \vec{Y'}$ for some $\vec{Y'} \not\longrightarrow_{\mathcal{P}}$, then $\vec{a}$ is called the *minimal path* of $\vec{Y}$, and its length is the *norm* of $\vec{Y}$, denoted $\mathsf{norm}(\vec{Y})$.

It is known that any unnormed word $\vec{Y}$ is bisimilar to its concatenation with any other word, i.e., if $\vec{Y}$ is unnormed, then $\vec{Y} \sim_{\mathcal{P}} \vec{Y}\vec{X}$ [20]. It is also easy to show that $\sim_{\mathcal{P}} \subseteq \lesssim_{\mathcal{P}}$, and hence that $\vec{Y} \lesssim_{\mathcal{P}} \vec{Y}\vec{X}$. In this case, $\vec{X}$ is said to be unreachable and can be safely removed from the grammar. We call the procedure of removing all unreachable symbols from a grammar *pruning*, and denote the pruned version of a grammar $\mathcal{P}$ by $\mathsf{prune}(\mathcal{P})$.

Lemma 6.2.1 (Soundness and completeness for pruning)**.** *$\vec{X} \preceq_{\mathcal{P}}^{\mathcal{XYZW}} \vec{Y}$ iff $\vec{X} \preceq_{\mathsf{prune}(\mathcal{P})}^{\mathcal{XYZW}} \vec{Y}$*

6.3 Exploring an expansion tree

In its third and final phase, the algorithm explores an *expansion tree*, alternating between expansion and simplification steps. An expansion tree is a tree whose nodes are sets of pairs of words, whose root is the singleton set containing the pair of starting words under test, and where every child is an *expansion* of its parent. A branch is deemed *successful* if it is infinite or has an empty leaf, and deemed *unsuccessful* otherwise. The original definition of expansion ensures that the union of all nodes along a successful branch (without simplifications) constitutes a bisimulation [58]. We adapt the definition of expansion to ensure that such a union yields an $\mathcal{XYZW}$-simulation instead, obtaining the following definition and result (cf. Appendix D for the proof).

Definition 6.3.1. The *$\mathcal{XYZW}$-expansion* of a node N is defined as the minimal set N' such that, for every pair $(\vec{X}, \vec{Y})$ in N, it holds that:

1. if $\vec{X} \to a\vec{X'}$ and $a \in \mathcal{X}$ then $\vec{Y} \to a\vec{Y'}$ with $(\vec{X'}, \vec{Y'}) \in N'$
2. if $\vec{Y} \to a\vec{Y'}$ and $a \in \mathcal{Y}$ then $\vec{X} \to a\vec{X'}$ with $(\vec{X'}, \vec{Y'}) \in N'$
3. if $\vec{X} \to a\vec{X'}$ and $a \in \mathcal{Z}$ then $\vec{Y} \to a\vec{Y'}$ with $(\vec{Y'}, \vec{X'}) \in N'$
4. if $\vec{Y} \to a\vec{Y'}$ and $a \in \mathcal{W}$ then $\vec{X} \to a\vec{X'}$ with $(\vec{Y'}, \vec{X'}) \in N'$

Lemma 6.3.1 (Safeness property for $\mathcal{XYZW}$-simulation)**.** *Given a set of productions $\mathcal{P}$, it holds that $\vec{X} \preceq_{\mathcal{P}}^{\mathcal{XYZW}} \vec{Y}$ iff the expansion tree rooted at $\{(\vec{X}, \vec{Y})\}$ has a successful branch.*

The simplification steps we mentioned consist of applying rules that safely modify the expansion tree during its construction, in an attempt to keep some branches finite. The rules are iteratively applied to each node until a fixed point is reached, at which point we can proceed with expansion. To each node N we apply three simplification rules, adapted from the equivalence algorithm [5]:

1. REFLEXIVITY: omit pairs of the form $(\vec{X}, \vec{X})$;
2. PREORDER: omit pairs belonging to the least preorder containing the ancestors of N;
3. SPLIT: if $(X_0\vec{X}, Y_0\vec{Y}) \in N$ and X_0 and Y_0 are normed, then:
   - Case $|X_0| \leq |Y_0|$: Let $\vec{a}$ be a minimal path for X_0 and $\vec{Z}$ the word such that $Y_0 \xrightarrow{\vec{a}}_{\mathcal{P}} \vec{Z}$. Add a sibling node for N including pairs $(X_0\vec{Z}, Y_0)$ and $(\vec{X}, \vec{Z}\vec{Y})$ in place of $(X_0\vec{X}, Y_0\vec{Y})$;
   - Otherwise: Let $\vec{a}$ be a minimal path for Y_0 and $\vec{Z}$ the word such that $X_0 \xrightarrow{\vec{a}}_{\mathcal{P}} \vec{Z}$. Add a sibling node for N including pairs $(X_0, Y_0\vec{Z})$ and $(\vec{Z}\vec{X}, \vec{Y})$ in place of $(X_0\vec{X}, Y_0\vec{Y})$.

When a node is simplified, we keep track of the original node in a sibling, thus ensuring that along the tree we keep an "expansion-only" branch.

The algorithm explores the tree by breadth-first search using a queue of node-ancestors pairs, thus avoiding getting stuck in infinite branches, and alternates between expansion and simplification steps until it terminates with **False** if all nodes fail to expand or with **True** if an empty node is reached. The following pseudo-code illustrates the procedure.

$\mathsf{subG}(\vec{X}, \vec{Y}, \mathcal{P}) = \mathsf{explore}(\mathsf{singletonQueue}((\{(\vec{X}, \vec{Y})\}, \emptyset), \mathcal{P})$
where $\mathsf{explore}(q, \mathcal{P}) =$
if $\mathsf{empty}(q)$ **then False** % all nodes failed to expand, no successful branch found
else let $(n, a) = \mathsf{head}(q)$ **in**
if $\mathsf{empty}(n)$ **then True** % empty node reached, successful branch found
else if $\mathsf{hasExpansion}(n, \mathcal{P})$ % if possible, expand node, simplify and recur
then $\mathsf{explore}(\mathsf{simplify}(\mathsf{expand}(n, \mathcal{P}), a \cup n, \mathsf{dequeue}(q)), \mathcal{P})$
else $\mathsf{explore}(\mathsf{dequeue}(q), \mathcal{P})$ % otherwise, discard node

Example 6.3.1. The $\mathcal{XYZW}$-expansion tree for Example 6.1.1 is illustrated in Fig. 6.1.

Finally, function subT puts all the pieces of the algorithm together:

$$\mathsf{subT}(T, U) = \textbf{let } (\vec{X}, \mathcal{P}') = \mathsf{grm}(T, \emptyset), (\vec{Y}, \mathcal{P}) = \mathsf{grm}(U, \mathcal{P}') \textbf{ in } \mathsf{subG}(\vec{X}, \vec{Y}, \mathsf{prune}(\mathcal{P}))$$

It receives two well-formed types T and U, computes their grammar and respective starting words $\vec{X}$ and $\vec{Y}$, prunes the productions of the grammar and, lastly, uses function subG to determine whether $\vec{X} \lesssim_{\mathcal{P}} \vec{Y}$.

We now state our main result concerning the soundness of subT, which guarantees that its implementation in a type checker cannot compromise any safety properties that depend on $\lesssim$. Its proof can be found in Appendix D.

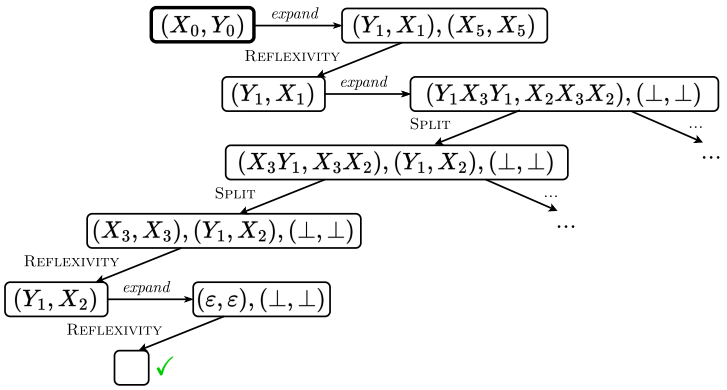


Figure 6.1: An $\mathcal{XYZW}$-expansion tree for Example 6.1.1, exhibiting a finite successful branch.

Theorem 6.3.1 (Soundness)**.** *If* $\mathsf{subT}(T, U)$ *returns* **True***, then* $T \lesssim U$.

6.4 Optimizations

Almeida et al. [5] present three possible optimizations for their equivalence algorithm:

1. *Eliminating redundant productions in the grammar.* When adding a new production $X \to \vec{Y}$ to the grammar, check if a syntactically equal production (up to renaming of non-terminal symbols) $Z \to \vec{W}$ already exists. If it does, abandon $Y \to \vec{X}$ and return Z.
2. *Filtering nodes with hopeless pairs.* Remove from the tree any node composed of pairs of words with different norms (observe that words with different norms cannot be bisimilar).
3. *Using a double-ended queue to prepend promising children.* Prepend (rather than append) empty nodes or nodes with pairs $(\vec{X}, \vec{Y})$ such that $\mathsf{norm}(\vec{X}) \leq 1$ and $\mathsf{norm}(\vec{Y}) \leq 1$.

The first and third optimizations are still applicable in the subtyping setting, since they do not deal directly with the properties of bisimilarity. The same, however, cannot be said for the second optimization. By generalizing the algorithm for $\mathcal{XYZW}$-similarity, the norm of a word loses its heuristic power: words with different norms may still be $\mathcal{XYZW}$-similar, normed words may be $\mathcal{XYZW}$-similar to unnormed words, and *vice-versa*.

Example 6.4.1. Consider streams of integers. We can use session types to describe:

- finite output streams, $\mathsf{FiniteOStream} = \mu s.\oplus\{\mathsf{Next}\colon !\mathsf{Int}; s, \mathsf{Stop}\colon \mathsf{Skip}\}$,
- infinite output streams, $\mathsf{InfiniteOStream} = \mu s.\oplus\{\mathsf{Next}\colon !\mathsf{Int}; s\}$, and
- singleton output streams, $\mathsf{SingletonOStream} = \oplus\{\mathsf{Next}\colon !\mathsf{Int}; \oplus\{\mathsf{Stop}\colon \mathsf{Skip}\}\}$,

and with corresponding words $\vec{X}_{\mathsf{F}}$, $\vec{X}_{\mathsf{I}}$ and $\vec{X}_{\mathsf{S}}$, respectively. Observe that we have $\vec{X}_{\mathsf{F}} \lesssim_{\mathcal{P}} \vec{X}_{\mathsf{S}}$ ($\mathsf{FiniteOStream} \lesssim \mathsf{SingletonOStream}$), yet $\mathsf{norm}(\vec{X}_{\mathsf{F}}) = 1$ and $\mathsf{norm}(\vec{X}_{\mathsf{S}}) = 2$. Similarly, $\vec{X}_{\mathsf{F}} \lesssim_{\mathcal{P}} \vec{X}_{\mathsf{I}}$ ($\mathsf{FiniteOStream} \lesssim \mathsf{InfiniteOStream}$), yet $\vec{X}_{\mathsf{F}}$ is normed and $\vec{X}_{\mathsf{I}}$ is unnormed.

It should be noted, however, that we suffer no great loss without optimization 2: in the empirical evaluation of Almeida et al., it actually worsened the performance of the algorithm. In fact, of the three optimizations, only the first brought general performance gains. As such, it is the only one we decided to include in our implementation.

Chapter 7

Subtyping the FREEST programming language

Establishing the soundness of our algorithm gives us the green light to implement it in a type checker. After all, if all positives it returns are *true positives*, the properties guaranteed by the type checker cannot be compromised. However, given the necessary incompleteness of the algorithm, we should not be surprised if we find cases where it does not return at all (in which case it should be interrupted and return **False**, whether true or not). Fortunately, as we demonstrate in the next chapter, this (safe) inaccuracy seems to be a small price to pay for the increase in expressive power afforded by subtyping.

We begin this chapter by briefly describing the FREEST programming language [95, 4, 2], version 3.3.0, which provided the testing grounds for the implementation and empirical evaluation of our algorithm. We then present a non-trivial program that highlights the benefits of adding subtyping to the language. We end the chapter by briefly describing the changes that were made to implement this feature.

7.1 The FREEST programming language

Programming language support for basic concurrency has been around for decades, and is by now expected from any practical programming language. Most mainstream general-purpose programming languages offer primitives for shared memory concurrency, and support for message-passing concurrency is already widespread.

Despite not enjoying mainstream presence yet, regular session types have, over their 30-year history, seen many proposals for their embedding in established languages like C [77], Java [57, 55], Python [76, 56], Rust [59], OCaml [81] and Haskell [80]. Several experimental languages, such as ATS [100], Links [36]and SePi [37] also offer native support for some variation of them. FREEST, however, is among the very few languages that can claim to support non-regular session types (to our knowledge, the only other is Rast [29]).

Spurred by the development of the type equivalence algorithm of Almeida et al., FREEST is currently the paramount implementation of context-free session types in a functional setting.

It's core language can be described as a *polymorphic*[1] *call-by-value*[2] λ-calculus enriched with primitives for spawning processes, creating communication channels and exchanging messages on them. Firmly grounded in the ML tradition, FREEST features a static typing discipline, favors currying and higher-order functions, and borrows much of its syntax and idioms from Haskell (and occasionally from OCaml and F#).

A FREEST program consists of an optional `module` header, where the current module is named and other modules may be imported, followed by a sequence of declarations: `type`/`data` declarations, type signatures and variable/function definitions. After parsing a program, the FREEST compiler uses the information provided by these declarations to check each of the definitions for type errors. If no errors are found, each of the definitions is then evaluated in order, and the output of the program is the value of the `main` variable, if defined.

Listing 7.1.1 shows an example of a FREEST program implementing a simple math server, governed by session types similar to those found in Chapter 3. It begins with two `type` declarations, which allow it to refer to the client and server channel end types by the names `NegClient` and `MathServer`. These declarations highlight one difference between our type language and FREEST: the `Wait` and `Close` types. FREEST uses asynchronous communication semantics, which means that sending processes don't wait for messages to be received in order to proceed. Having a single `End` type presupposes a single `close` operation, which forces processes to wait for each other to terminate the connection. By splitting `End` into `Wait` and `Close`, we can distinguish between two terminal operations, `wait` and `close`, and achieve asynchronicity by making the former blocking and the latter non-blocking. Following the `type` declarations, a type signature specifies that the type of `mathServer`, the function that performs the duties of the server, is `MathServer -> ()`, i.e., a function taking in a channel end of type `MathServer` and returning `()`. The definition for this function follows, in the form of two equations. These equations showcase FREEST's pattern matching feature, which allow us to emulate common mathematical notation and perform case analysis directly on the left-hand side of equations, avoiding the need to write `case` and `match` expressions [3]. The right-hand side of these equations is similar to our examples from Chapter 3, and, apart from operator `|>`, should not be hard to read. [4]

The signature and definition of `negClient` follow, and the right-hand side of its equation should also be clear (`@Int` is a type checking annotation and can be ignored.[5]).

[1] I.e., featuring generic types, like $\forall\alpha.\alpha \to \alpha$ for a function that takes and returns values of the same type.

[2] *Call-by-value* refers to an evaluation strategy in which arguments are evaluated before being substituted in the body of abstractions (this was our strategy in Chapter 2). It stands in contrast with *call-by-need* or *lazy* evaluation (used by e.g. Haskell), where arguments are passed as suspended computations (*thunks*) to be evaluated only when needed.

[3] Alternatively, we could write this definition as we have done in Chapter 3, using a single equation of the form `mathServer c = match c with {...}` (in fact, this is what the definition above gets translated to by the compiler).

[4] This operator, inspired by F#, simply denotes reverse function application: `(x |> f)` stands for `(f x)`. By associating to the right, `|>` helps programmers chain sequences of operations on the same channel end in the order they are intended, and is therefore widely used in FREEST programs.

[5] As stated above, FREEST supports generic functions. The self explanatory `receiveAndClose` function, having type `forall a. ?a;End -> a`, is one such function. In order to apply it to a channel end of type `?Int;End`, we must instantiate the type variable `a` with type `Int`, and this is the purpose of the `@Int` annotation. Sparing programmers the burden of writing and reading such annotations is one of the goals of future work on the language.

```
type MathServer = &{Neg: ?Int;!Int, Add: ?Int;?Int;!Int};Wait
type NegClient  = +{Neg: !Int;?Int                      };Close

mathServer : MathServer -> ()
mathServer (Neg s) = let (n, s) = receive s in
                     send (-n) s |> wait
mathServer (Add s) = let (n1, s) = receive c in
                     let (n2, s) = receive c in
                     send (n1 + n2) s |> wait

negClient : NegClient -> Int
negClient c = c |> select Neg |> send 5 |> receiveAndClose @Int

main : Int
main = let (c,s) = new @NegClient () in
       fork (\_:() 1-> mathServer s);
       negClient c
```

Listing 7.1.1: A FREEST program implementing a simple math server.

Finally, the definition of variable `main`, of type `Int`, sets everything in motion. It begins by creating a channel for the client and server to communicate, using `new` primitive. Annotating `new` with `@NegClient` results in a channel end of type `NegClient`, `c`, and another of a type dual to `NegClient`, `s`, which together will ensure that the intended protocol is followed. Then, applying the `fork` primitive to what we call a *thunk* (a function with a trivial argument of type `()`, meant to simply suspend the evaluation of its body), a new process is spawned to evaluate the expression `mathServer r`, effectively launching the server. Finally, by calling `negClient` with `w`, the main thread takes on the role of the client. Since `negClient` requests the negation of 5, we can expect `main` to evaluate to `-5`.

Besides demonstrating the main features of the FREEST language, this small program also enjoys the benefits of subtyping, for which we added support in version 3.3.0. Observe that, without this feature, expression `mathServer s` would raise a type error, for the type of `s` (a dual of `NegClient`) is clearly not equivalent to `MathServer` (but is clearly a subtype of it).

7.2 Example: JSON serialization

The example from the previous section is quite simple, and does not take full advantage of our contributions (observe that `MathServer` and `NegClient` are perfectly regular session types). Recalling the JSON serialization protocol from Example 4.1.2, we now turn our attention to a FREEST program exhibiting subtyping in a more practical and inherently context-free setting.

Representing JSON The context-free nature of the JSON format arises from its recursive, tree-like structure, which allows array and object values to aggregate multiple other arbitrary JSON

```
data JSON = JObj  JObj
          | JArr  JArr
          | JStr  String
          | JNum  Int
          | JBool Bool
          | JNull
data JObj = JKeyVal String JSON JObj | JEmptyO
data JArr = JElem          JSON JArr | JEmptyA

j : JSON
j = JObj (JKeyVal "age" (JNum 84) (JKeyVal "married" (JBool False) JEmptyO))
```

Listing 7.2.1: FREEST representation of JSON data using algebraic datatypes

values. Before serializing such values, we must know how to represent them. For this purpose, FREEST features an algebraic datatype mechanism, providing a simple and convenient syntax to declare and use (possibly recursive) variant types like those introduced in Section 2.4. Our representation of JSON data using this mechanism is given by the `data` declarations in Listing 7.2.1. The definition for `j`, corresponding to the JSON object `{"age": 84, "married": false}`, shows how such values may be created. [6]

Note 7.2.1. In FREEST, `data` declarations are the only way to introduce variant types. Since FREEST does not allow tags to be shared among `data` declarations, it is not possible to have the same tag in two different variant types. As such, it is impossible for these types to benefit from width subtyping as described in previous chapters. This is also the case for record types, which, as of version 3.1.0, have no concrete syntax for their introduction and elimination (their use in FREEST is restricted to the internal core language).

Deserializing JSON With an appropriate representation for JSON values, we can now think about how they should be serialized across a channel. Following the structure of the `data` declarations and (for the sake of our example) taking the point of view of the receiver, we obtain the `type` declaration for `ReceiveJSON` in Listing 7.2.2, which relies on the declarations for `ReceiveKeyVals` and `ReceiveElems` to handle the recursive, list-like deserialization of arrays and objects. The implementation of the protocol follows almost immediately from these declarations, in the form of functions `receiveJSON`, `receiveKeyVals` and `receiveElems` respectively (observe that the type of these functions must be polymorphic on the continuation of the argument type, for otherwise they would not be able to serialize multiple branches of a JSON value).

Serializing an array of numbers Given the implementation of JSON deserialization, implementing its reverse, serialization, is easy: we simply follow `SendJSON`, the dual of `ReceiveJSON`,

[6] To exemplify the convenience of datatypes, in our original language, type `JArr` would be defined as $\mu t.\langle \mathsf{Elem}: (\mathsf{JSON}, t), \mathsf{EmptyArr}: ()\rangle$, and expression `(KeyVal "married" (JBool False) EmptyObj)` would be written as KeyVal ("married", JBool false as JSON, EmptyObj () as JObj) as JObj.

```
type ReceiveJSON    = &{ Object : ReceiveKeyVals
                       , Array  : ReceiveElems
                       , String': ?String
                       , Number : ?Int
                       , Boolean: ?Bool
                       , Null   : Skip }
type ReceiveKeyVals = &{ KeyVal: ?String;ReceiveJSON;ReceiveKeyVals
                       , EmptyO: Skip }
type ReceiveElems   = &{ Elem  : ReceiveJSON;ReceiveElems
                       , EmptyA: Skip }

type SendJSON = dualof ReceiveJSON

receiveJSON : ReceiveJSON;a -> (JSON, a)
receiveJSON (Object  c) = let (o, c) = receiveKeyVals @a c in (JObj o, c)
receiveJSON (Array   c) = let (a, c) = receiveElems   @a c in (JArr a, c)
receiveJSON (String' c) = let (s, c) = receive c in (JStr  s, c)
receiveJSON (Number  c) = let (n, c) = receive c in (JNum  n, c)
receiveJSON (Boolean c) = let (b, c) = receive c in (JBool b, c)
receiveJSON (Null    c) = (JNull, c)

receiveKeyVals : ReceiveKeyVals;a -> (JObj, a)
receiveKeyVals (KeyVal c) = let (k, c) = receive c in
                            let (v, c) = receiveJSON @(ReceiveKeyVals;a) c in
                            let (kvs, c) = receiveKeyVals @a c in
                            (JKeyVal k v kvs, c)
receiveKeyVals (EmptyO c) = (JEmptyO, c)

receiveElems : ReceiveElems;a -> (JArr, a)
receiveElems (Elem   c) = let (j , c) = receiveJSON @(ReceiveElems;a) c in
                          let (js, c) = receiveElems @a c in
                          (JElem j js, c)
receiveElems (EmptyA c) = (JEmptyA, c)
```

Listing 7.2.2: FREEST implementation of a JSON deserialization protocol.

which can be succinctly defined using the `dualof` type operator. Suppose, however, that we would like to have specialized serialization functions that enforce certain restrictions on the structure of the JSON data they send. For example, we may want to have a serialization function `sendNums` that is restricted to sending lists of integers (type `[Int]`) but is still able to communicate with `receiveJSON`, which receives arbitrary JSON values. Subtyping enables us to implement this asymmetry, requiring only that we carefully refine `sendNums`' view of the JSON protocol until it is restricted to arrays of numbers. The resulting type, `SendNums`, is shown in Listing 7.2.3 along with the implementation of the protocol in the form of function `sendNums`.

```
type SendNums     = +{ Array : SendNumElems }
type SendNumElems = +{ Elem: SendNum;SendNumElems
                     , EmptyA: Skip }
type SendNum      = +{ Num: !Int }

sendNums : [Int] -> SendNums;a -> a
sendNums xs c = c |> select Array |> sendNumElems @a xs

sendNumElems : [Int] -> SendNumElems;a -> a
sendNumElems []      c = c |> select EmptyA
sendNumElems (x::xs) c = c |> select Elem |> select Number |> send x
                           |> sendNumElems @a xs

main : JSON
main = let (o,i) = new @(SendJSON;Close) () in
       fork (\_:() 1-> sendNums @Close [1,3,5] o |> close);
       let (j, i) = receiveJSON @Wait i in
       wait i; j
```

Listing 7.2.3: FreeST implementation of a JSON-compatible integer list serialization protocol, governed by context-free session types

Putting it all together Finally, also in Listing 7.2.2, we have the definition of the `main` variable, which sets in motion two threads exemplifying the usage of `sendNums` and `receiveJson`. First, a general JSON serialization channel is created with `new @(SendJSON;Close) ()`, resulting in a pair of channel ends `o`, of type `SendJSON;Close`, and `i`, of type `ReceiveJSON;Wait`. Then, a new process is spawned to evaluate `o |> sendNums @Close [1,2,3,4] |> close`, which will serialize list `[1,3,5]` as a JSON value through our previously created channel. Here we can witness subtyping in action: `sendNums @Close` expects a channel end of type `SendNums;Close`, but is given one of type `SendJSON;Close`. Since our contributions allow FreeST to recognize `SendJSON;Close` as a subtype `SendNums;Close`, the type checker considers this expression valid and no error is raised. With the sending process spawned, the `main` function then proceeds to take the role of the receiver by calling `receiveJSON @Wait` on `i`. After receiving the JSON data and binding it to `j`, it uses `wait` and blocks until the sending process uses `close` to terminate connection. Finally, it returns `j`, which, being the value of the `main` variable, will be output to the command line as `JArr (JElem (JNum 1) (JElem (JNum 3) (JElem (JNum 5) JEmptyA)))`.

7.3 Implementing subtyping in FreeST

The FreeST 3.2.0 compiler is written in Haskell and features a running implementation of the type equivalence algorithm of Almeida et al. [5], on which our subtyping algorithm is based. As such, adding subtyping in FreeST 3.3.0 was mostly (but not only) a matter of adapting this implementation to reflect our changes to the original algorithm. We close this chapter by describing

$$\text{TA-EQ}\quad \frac{\Gamma_1 \vdash e \Rightarrow U \mid \Gamma_2 \qquad T \simeq U}{\Gamma_1 \vdash e : T \Rightarrow \Gamma_2} \qquad \text{TA-SUB}\quad \frac{\Gamma_1 \vdash e \Rightarrow U \mid \Gamma_2 \qquad U \leq T}{\Gamma_1 \vdash e : T \Rightarrow \Gamma_2}$$

Figure 7.1: FREEST's previous check-against rule, TA-EQ, and the new one, TA-SUB.

this process in some detail. The source code for both versions of the compiler can be downloaded from the FREEST language homepage [95]. For the sake of brevity, we assume the reader has some familiarity with the Haskell programming language.

Compiler structure The FREEST compiler can be described as a pipeline that takes a source file containing a FREEST program, parses it to build an abstract representation of that program, prepares said representation for further processing (in a process called *elaboration*), validates it and, finally, executes it with an interpreter. In Haskell, this is implemented as a series of stateful computations encapsuled by a `State` monad, with the state represented as a record bundling together the information needed across the compilation process, e.g., type signatures, variable definitions, errors encountered, as well as bookkeeping information.

Adapting type checking Subtyping is strictly contained in the validation phase of the compiler, which, besides ensuring types are well-formed, checks the definition of every variable against the types declared in its signature using a bidirectional type checking algorithm. In such typing algorithms, a typing derivation is constructed bottom-up two with the aid of two mutually defined operations: *synthesis*, which induces the type of an expression from its structure, and *check-against*, which verifies that an expression matches a certain type.

According to the language specification [2], the original type equivalence algorithm is evoked in the checking phase, which, omitting some details, is described by rule TA-EQ in Fig. 7.1 where judgment $\Gamma_1 \vdash e : T \Rightarrow \Gamma_2$ is read as "under context Γ_1, expression e successfully checks against type T and yields a new context Γ_2" and judgment $\Gamma_1 \vdash e \Rightarrow U \mid \Gamma_2$ can be read as "under context Γ_1, we can determine that e has type U, yielding a new context Γ_2". The Γ_2 context yielded at the end of both judgments contains all variables in Γ_1 except the linear ones consumed during synthesis, ensuring that they cannot be used more than once. In short, this rule states that an expression successfully checks against a given type T if we can find some type U for it such that $T \simeq U$. To enjoy subtyping, all we need to do is replace the call to the equivalence algorithm, $T \simeq U$, with a call to the subtyping algorithm, $U \leq T$, obtaining rule TA-SUB.

Since FREEST is an experimental language and users may want to retain control over whether or not to use subtyping, we include the command line flag `--sub` to enable this feature. The relevant code for these changes can be found in modules `Validation.Typing` and `Util.CmdLine` of the FREEST compiler.

Adapting equivalence The changes necessary to adapt the implementation of the equivalence algorithm directly reflect those mentioned in the previous chapter. Summarily, we:

1. adjusted the grammar translation procedure (to include new productions for linear functions, records, variants and choices, as well as the `Close` and `Wait` types);
2. generalized the definition of expansion to allow for $\mathcal{XYZW}$-expansion;
3. adjusted the simplification rules;
4. removed the optimization that filtered pairs of words based on their norms.

Setting a timeout Since the subtyping algorithm may not terminate, it is wise to set a time limit for it to run before interrupting the computation and signaling a timeout error. In this way, we avoid leaving the user on hold indefinitely in the cases where the algorithm does not halt. Based on our experiments (which are covered in the next chapter), we chose a default limit of 1 minute per call to the algorithm. However, since this may be too low for some complex (e.g., highly recursive) types, we also allow the user to adjust this before compilation using the command line option `--check-timeout TIME`, where argument `TIME` is the desired limit in milliseconds. When this limit is exceeded, the user is presented with an error message like the following.

```
Timeout when matching expected type rec a. &{N:a;a;!Int, L:Skip};!Int
                    with actual type rec a. &{N:a;a;!Int, L:!Int
                      for expression c
Subtyping is enabled, timeout set to 60000ms

FreeST's subtyping relation is undecidable, so it may not be
  possible to determine relation between certain types.
Consider disabling subtyping with the '--no-sub' flag, or
  setting a higher limit with the '--check-timeout TIME' option.
Help us make FreeST better! Report this issue to the
  development team at freest-lang@listas.ciencias.ulisboa.pt
```

Haskell is a mostly pure functional language: except under an `IO` monad, functions may not have side effects, and their return values must depend entirely on their arguments. In the FreeST compiler, before our changes, the use of `IO` was mostly restricted to the parser (to access the file system) and the interpreter (to evaluate FREEST programs, which themselves have side effects). However, setting a timeout during type checking requires us to extend the use of `IO` to this phase as well, which, as mentioned above, was already under a `State` monad. To combine the two monads, we used the *monad transformer* `StateT`, creating a *monad stack* that allows the use of the operations associated with the `State` monad and, through *lifting* (more concretely, through the `liftIO` function), to the effectful operations that need `IO`.

Chapter 8

Evaluating the algorithm

With our contributions, FREEST effectively gains support for subtyping at little to no cost in performance. In this chapter, we present an empirical study to support this claim. Our evaluation is based on two suites of tests: a suite of randomly generated pairs of types, and a suite of handwritten FREEST programs. All data was collected on a machine featuring an Intel Core i5-6300U at 2.4GHz with 16GB of RAM.

8.1 Generative testing

The design and implementation of our algorithm was guided by the principles of test-driven development: even before any code was written, dozens of unit tests were designed. These tests consisted of valid and invalid pairs of types, carefully designed to access whether the implementation correctly reflects the particular features of our notion of subtyping. As development progressed, more tests were written such that, by the end 168 unit tests had been written.

Despite their usefulness in guiding develpment, we judged these test cases to be too few and small (each using only a handful of type constructors per type), and therefore not suitable for a robust evaluation of the performance of our implementation. Writing larger types by hand was, however, judged to be too menial and error-prone. As such, we turned our attention to *generative testing*, a testing technique test cases are constructed automatically and at random using dedicated generators. By redirecting most of our efforts to the design and implementation of said generators, this technique makes it easier to produce great numbers of large test cases.

The QuickCheck[22] library is by far the most popular tool for generative testing. Native to the Haskell ecosystem, it has been ported to numerous languages, and its ideas have inspired many other tools [67, 78, 50]. It can be described as a lightweight library that provides a carefully designed set of functions and types that can be combined to define random data generators and specify the properties that the generated data should exhibit. The high-level, combinatorial nature of these functions gives QuickCheck the flavor of an embedded domain-specific language that can be used to tailor the generative testing approach to the particular needs of the software under test.

To test our algorithm using QuickCheck, we decided to implement two generators: one for valid subtyping pairs, and another for invalid ones. Using these generators, we then design a testing

procedure that will produce thousands of test cases and run the algorithm on them, expecting it to return **True** on valid pairs and **False** on invalid ones.

Generating valid pairs

To generate valid pairs of types, i.e., pairs that are in the subtyping relation *by construction*, we follow an algorithm induced from the properties of subtyping, much like the one induced by Almeida et al. [5]. The following theorem enumerates the properties.

Theorem 8.1.1 (Valid subtyping properties)**.**

1. $\mathsf{Unit} \leq \mathsf{Unit}$, $\mathsf{End} \leq \mathsf{End}$ *and* $\mathsf{Skip} \leq \mathsf{Skip}$*;*
2. $T \overset{m}{\to} U \leq V \overset{n}{\to} W$ *if* $V \leq T$, $U \leq W$ *and* $m \sqsubseteq n$*;*
3. $\{\ell\colon T_\ell\}_{\ell \in L} \leq \{k\colon U_k\}_{k \in L}$ *if* $K \subseteq L$ *and* $T_j \leq U_j (\forall j \in K)$*;*
4. $\langle \ell\colon T_\ell \rangle_{\ell \in L} \leq \langle k\colon U_k \rangle_{k \in K}$ *if* $L \subseteq K$ *and* $T_j \leq U_j (\forall j \in L)$*;*
5. $S_1;S_2 \leq R_1;R_2$ *if* $S_1 \leq R_1$ *and* $S_2 \leq R_2$*;*
6. $\mu x.T \leq U$ *if* $T \leq U$ *and* $x \notin \mathsf{free}(T) \cup \mathsf{free}(U)$*;*
7. $T \leq \mu x.U$ *if* $T \leq U$ *and* $x \notin \mathsf{free}(T) \cup \mathsf{free}(U)$*;*
8. $\mu x.T \leq [\mu y.U/y]U$ *if* $\mu x.T \leq \mu y.U$*.*
9. $\mu x.T \leq \mu x.U$ *if* $T \sim U$ *or otherwise* $T \leq U$ *and* $x \notin \mathsf{free}^-(T) \cup \mathsf{free}^-(U)$*;*
10. $\odot\{\ell\colon S_\ell\}_{\ell \in L} \leq \odot\{k\colon R_k\}_{k \in L}$ *if* $K \subseteq L$ *and* $S_j \leq R_j (\forall j \in L)$*;*
11. $\&\{\ell\colon S_\ell\}_{\ell \in L} \leq \&\{k\colon R_k\}_{k \in K}$ *if* $L \subseteq K$ *and* $S_j \leq R_j (\forall j \in L)$*;*
12. $\mathsf{End};S \leq \mathsf{End}$, $\mathsf{End} \leq \mathsf{End};S$ *and* $\mathsf{End};S \leq \mathsf{End};R$ *for any* S, R*;*
13. $S;\mathsf{Skip} \leq R$, $\mathsf{Skip};S \leq R$, $S \leq R;\mathsf{Skip}$ *and* $S \leq \mathsf{Skip};R$ *if* $S \leq R$*;*
14. $\oplus\{\ell\colon S_\ell\}_{\ell \in L};S' \leq \oplus\{k\colon R_k;R'\}_{k \in L}$ *and* $\oplus\{\ell\colon S_\ell;S'\}_{\ell \in L} \leq \oplus\{k\colon R_k\}_{k \in L};R'$ *if* $K \subseteq L$, $S_j \leq R_j (\forall j \in L)$ *and* $S' \leq R'$*;*
15. $\&\{\ell\colon S_\ell\}_{\ell \in L};S' \leq \&\{k\colon R_k;R'\}_{k \in L}$ *and* $\&\{\ell\colon S_\ell;S'\}_{\ell \in L} \leq \&\{k\colon R_k\}_{k \in L};R'$ *if* $L \subseteq K$, $S_j \leq R_j (\forall j \in L)$ *and* $S' \leq R'$*;*
16. $S_1;(S_2;S_3) \leq (R_1;R_2);R_3$ *and* $(S_1;S_2);S_3 \leq R_1;(R_2;R_3)$ *if* $S_1 \leq R_1$, $S_2 \leq R_2$ *and* $S_3 \leq R_3$*;*

Proof. By observation of the syntactic subtyping rules in Fig. 5.1 and Definition 8.1.1. □

From this theorem we can derive generation algorithm for valid subtyping pairs, parameterized on the size i of the pair: *if* $i = 0$*, then generate a type reference* x *and return pair* (x, x)*, or select at random one of the pairs in Item 1 of Theorem 8.1.1; if* $i \geq 1$*, select at random one of the pairs in the remaining items and, for all corresponding pairs of type metavariables, recursively generate a valid pair with size* $i - 1$. (In the actual implementation, the size passed to the recursive call is adjusted on a case-by-case basis to balance the distribution of type constructors.)

We must pay special attention to type references when generating pairs envolving recursive type constructor μ. For example, when generating pair $\mu x.T \leq U$ with $T \leq U$ in Item 6, we must ensure that x is not free in T or U, for otherwise enclosing only T in $\mu x.$ would make the

ocurrences of x in T refer to T itself, while those in U could refer to a larger type, of which U would be a part.

Even where we we have the μ on both sides, in Item 9, we must take special precautions. It is not enough for the types underneath $\mu x.$ to be in the subtype relation: x must not appear in a contravariant position in either type. Observe that, despite looking so, $\mu t.t \xrightarrow{*} t$ is not a subtype of $\mu t.t \xrightarrow{1} t$ (their unfolding makes it clear). Since t appears in both covariant (range) and contravariant (domain) positions, $\mu t.t \xrightarrow{*} t$ must be simultaneously subtype and supertype of $\mu t.t \xrightarrow{1} t$, which cannot be true because of the multiplicities of the arrows. We avoid generating such pairs in Item 9 by ensuring that the reference we bind only appears in covariant positions of T and U, unless they are equivalent, in which case there is no such restriction. The set of free references in contravariant positions of a type T is given by $\mathsf{free}^-(T)$, and is defined as follows.

Definition 8.1.1. The sets of free references in covariant and contravariant positions in a type T, $\mathsf{free}^+(T)$ and $\mathsf{free}^-(T)$, are mutually defined by induction on the structure of T:

$$\begin{aligned}
\mathsf{free}^+(T \xrightarrow{m} U) &= \mathsf{free}^-(T) \cup \mathsf{free}^+(U) & \mathsf{free}^-(T \xrightarrow{m} U) &= \mathsf{free}^+(T) \cup \mathsf{free}^-(U) \\
\mathsf{free}^+(\{\!|\ell\colon T_\ell|\!\}_{\ell\in L}) &= \bigcup_{k\in L} \mathsf{free}^+(T_k) & \mathsf{free}^-(\{\!|\ell\colon T_\ell|\!\}_{\ell\in L}) &= \bigcup_{k\in L} \mathsf{free}^-(T_k) \\
\mathsf{free}^+(?T) &= \mathsf{free}^+(T) & \mathsf{free}^-(?T) &= \mathsf{free}^-(T) \\
\mathsf{free}^+(!T) &= \mathsf{free}^-(T) & \mathsf{free}^-(!T) &= \mathsf{free}^+(T) \\
\mathsf{free}^+(S;R) &= \mathsf{free}^+(S) \cup \mathsf{free}^+(R) & \mathsf{free}^-(S;R) &= \mathsf{free}^-(S) \cup \mathsf{free}^-(R) \\
\mathsf{free}^+(\mu x.T) &= \mathsf{free}^+(T) \setminus \{x\} & \mathsf{free}^-(\mu x.T) &= \mathsf{free}^-(T) \setminus \{x\} \\
\mathsf{free}^+(x) &= \{x\}
\end{aligned}$$

and in all other cases by $\mathsf{free}^+(T) = \emptyset$ and $\mathsf{free}^-(T) = \emptyset$, respectively.

Generating invalid pairs

To generate invalid subtyping pairs, we follow the same algorithm but inject, at random, the invalid pairs that occur in each item of the following theorem.

Theorem 8.1.2 (Invalid subtyping properties)**.**

1. $\mathsf{Int} \not\leq \mathsf{Unit}$ *and* $\mathsf{Unit} \not\leq \mathsf{Int}$*;*
2. $\{\ell\colon T_\ell\}_{\ell\in L} \not\leq \{k\colon U_k\}_{k\in L}$ *if* $L \subsetneq K$ *and* $T_j \leq U_j (\forall j \in L)$*;*
3. $\langle \ell\colon T_\ell\rangle_{\ell\in L} \not\leq \langle k\colon U_k\rangle_{k\in K}$ *if* $K \subsetneq L$ *and* $T_j \leq U_j (\forall j \in K)$*;*
4. $T \xrightarrow{1} U \not\leq V \xrightarrow{*} W$*, with* $V \leq T, U \leq W$*;*
5. $T \xrightarrow{m} U \not\leq V \xrightarrow{n} W$*, with* $T \not\sim V$*,* $T \leq V, m \sqsubseteq n, U \leq W$*;*
6. $T \xrightarrow{m} U \not\leq V \xrightarrow{n} W$*, with* $V \leq T, m \sqsubseteq n, U \not\sim W, W \leq U$*;*
7. $\mathsf{Skip} \not\leq \mathsf{End}$ *and* $\mathsf{End} \not\leq \mathsf{Skip}$*;*
8. $?T \not\leq !T$ *and* $!T \not\leq ?T$*;*
9. $?T \not\leq ?U$*, with* $T \not\sim U, U \leq T$*;*
10. $!T \not\leq !U$*, with* $T \not\sim U, T \leq U$*;*

11. $\oplus\{\ell\colon T_\ell\}_{\ell\in L} \not\leq \oplus\{k\colon T_k\}_{k\in K}$ *if* $L \subsetneq K$ *and* $T_j \leq U_j (\forall j \in L)$*;*
12. $\&\{\ell\colon T_\ell\}_{\ell\in L} \not\leq \&\{k\colon T_k\}_{k\in K}$ *if* $K \subsetneq L$ *and* $T_j \leq U_j (\forall j \in K)$.

Proof. By observation of the syntactic subtyping rules in Fig. 5.1. □

If $i = 0$, we generate one of the pairs in Item 1 of Theorem 8.1.2 or Item 1 of Theorem 8.1.1. Otherwise, if $i \geq 1$, we use one of the items in Theorem 8.1.1 and randomly inject an invalid pair derived from Theorem 8.1.2 where a valid one is supposed to be generated. Naturally, this may result in a valid pair. In that case, when testing, we simply discard the result and try again until an invalid type is found (we can do this since the algorithm is sound and we are simply interested in evaluating its performance).

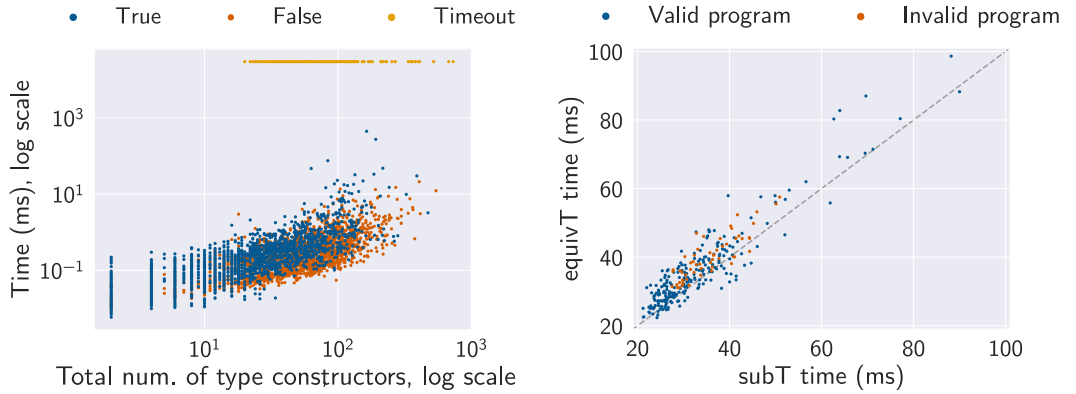
Example 8.1.1. Suppose $i = 4$. We randomly choose Item 5 of Theorem 8.1.1 to generate a pair of sequential compositions $(S_1;R_1, S_2;R_2)$. We proceed in the valid path for the types before the semicolon, obtaining $S_1 = !\mathsf{Int};\mathsf{Skip}$ and $S_2 = !\mathsf{Int}$, but inject an invalid pair in R_1 and R_2. To generate (R_1, R_2), we randomly choose Item 5 of Theorem 8.1.1. Here we generate a valid pair (T, U) using Theorem 8.1.1 and ensuring T and V are not equivalent, then multiplicities m and n such that $m \sqsubseteq n$, and finally another valid pair (V, W). Thus we obtain $R_1 = T \overset{m}{\to} U$ and $R_2 = V \overset{n}{\to} W$, making $(S_1;R_1, S_2;R_2)$ an invalid pair. Observe, however, that if we happened to generate $S_1 = \mathsf{End}$ and $S_2 = \mathsf{End};\mathsf{Skip}$, then $(S_1;R_1, S_2;R_2)$ would be a valid pair, and its test result would be discarded.

Conducting the evaluation

Finally, we conducted our evaluation by taking the running time of the algorithm on 2000 valid pairs and 2000 invalid pairs, ranging from 2 to 730 total type constructors. Since the algorithm may not terminate, we set a timeout of 30s. The results are plotted in Figure 8.1a. Despite the incompleteness of the algorithm, we encountered no failed tests, but obtained 200 timeouts. We found, as expected, that the running time increases considerably with the number of nodes. When a result was produced, valid pairs took generally longer than invalid pairs. On the other hand, we found that pairs originating from the invalid generator produced the largest number of timeouts, with 186 of 200 attributed to them.

8.2 Program testing

Randomly generated types allow for a robust analysis, but they typically do not reflect the types encountered by a subtyping algorithm in its most obvious practical application: a programming language compiler. For this reason, we turn our attention to our suite of FreeST programs, comprised of 286 valid and invalid programs collected throughout the development of the FreeST language. Programs range from small examples demonstrating particular features of the language to concurrent applications simulating, for example, an FTP server.



(a) Performance on valid and invalid subtyping pairs

(b) Performance comparison against the original equivalence algorithm

Figure 8.1: Performance evaluation and comparison

We began by integrating the algorithm in the FreeST compiler, placing next to every call to the original algorithm [5] (henceforth equivT) a call to subT on the same pairs of types, both algorithms equipped with optimization 1. from Section 6.4 (*eliminating redundant productions in the grammar*). We then ran each program in our suite 10 times, collecting and averaging the accumulated running time of both algorithms on the same pairs of types. We then took the difference between the average accumulated running times of subT and equivT, obtaining an average difference of -3.85ms, with a standard deviation of 7.08ms, a minimum difference of -71.29ms and a maximum difference of 8.03ms (subT performed faster, on average). Fig. 8.1b illustrates this comparison by plotting against each other the accumulated running times (for clarity, those in the 20-100ms range) of both algorithms during the typechecking phase of each.

The data collected from the program suite suggests that replacing the original equivalence algorithm [5] with the subtyping algorithm in the FreeST type checker incurs virtually no overhead, and greatly increases the number of programs it accepts. Still, however promising these results, we cannot ignore the inherent incompleteness of our algorithm, nor the timeouts we observed in the random tests. For this reason, and as described in the previous chapter, we included a timeout policy in its implementation, as well as a mechanism to choose whether to use subtyping, or to rely solely on equivalence (for which there is a sound, complete and terminating algorithm).

Chapter 9

Conclusion and future work

Subtyping context-free session types comes with challenges that are not encountered when subtyping their less expressive regular counterparts: we must account for the algebraic properties introduced by the generalized sequencing operator $(;)$, while dealing with the complexities of the non-tail recursion it allows. Together, these challenges make undecidable the problem of checking whether an arbitrary context-free session type is a subtype of another, as previous work by Padovani [82] shows. This is the unfortunate—yet usual—price to pay for expressive power.

Despite these challenges, we propose a syntactic, inference rule-based notion of subtyping for context-free session types. Finding it unsuitable for algorithmic treatment, we then propose a coinciding semantic notion, based on a novel form of observational pre-order we call $\mathcal{XYZW}$-simulation, a generalization of $\mathcal{XY}$-simulation proposed by Aarts and Vaandrager [1]. This notion of simulation allows the selective combination of the requirements of plain simulation and its inverse, along with a strong form of contra-simulation, which enables it to model the full covariance and contravariance expected from the choice and input/output constructors of context-free session types (by contrast, the notion of subtyping proposed by Padovani does not allow variance in the input/output constructors).

Semantic equivalence for context-free session types is usually based on *bisimilarity* [83], which $\mathcal{XYZW}$-similarity generalizes. Taking advantage of this fact, we derive a subtyping algorithm from the existing type bisimilarity algorithm of Almeida et al. [5]. Our algorithm is sound but, due to the undecidability of our notion of subtyping, necessarily incomplete.

We then implemented our algorithm in the freely available compiler for the FreeST programming language [2, 4, 95], which supports context-free session types and features an implementation of the equivalence algorithm of Almeida et al. [5]. In order to evaluate the correctness and performance of our implementation, we employed a suite of 4000 tests, generated automatically with the aid of the QuickCheck library for Haskell [22]. No tests failed, and we obtained satisfactory performance, observing only 200 timeouts. For a more realistic evaluation, and in order to compare our adaptation to the original algorithm it replaced, we also measured the running times of both algorithms on a suite of 286 FreeST programs without subtyping. We did not observe any significant differences in performance, which, together with the previous results, suggests that our algorithm is viable for practical use.

Future work

Partial correctness Despite its unavoidable incompleteness, which stems from the undecidability of our notion of subtyping, our (semi-)algorithm has not yielded any false negatives. Thus, we conjecture that is *partially correct*: it may not halt, but, when it does, the answer is correct. We cannot, however, back this claim without careful analysis, which we leave for future work. We believe such an analysis will advance the understanding of the subtyping problem by clarifying the practical reasons for its undecidability.

Parametric polymorphism As shown by Thiemann and Vasconcelos [96], support for *parametric polymorphism* [44, 87] is paramount in practical applications of context-free session types. This feature allows software components to be given a "generic" type and is particularly useful to avoid code duplication, since it lets functions that don't depend on the type of their argument be applied to a term of any type. It can be achieved by enriching the language of types with the universal quantifier constructor $\forall\alpha.\, T$, which binds occurences of variable α in T and allows them to be substituted by a concrete type when necessary. For example, instead of having an identity function for every type ($\mathsf{Int} \to \mathsf{Int}$, $\mathsf{Bool} \to \mathsf{Bool}$, etc.), programmers can write a single function of type $\forall\alpha.\, \alpha \to \alpha$, and later instantiate it to whatever type is necessary. With context-free session types, this feature is actually *necessary* to be able to type functions that recur on non-regular session types. As Thiemann and Vasconcelos exemplify [96], a recursive function that sends a Tree along a $\mathsf{SerializeTree}$ channel must be polymorphic on both the continuation of the channel and its return type (i.e., have type $\forall\alpha.\, \mathsf{Tree} \to \mathsf{SerializeTree};\alpha \to \alpha$) since, roughly stated, the recursive call of the left branch of the tree needs to return the channel on which the recursive call for the right branch will be made. While in common type systems subtyping and parametric polymorphism can be taken as orthogonal features, it is not clear if it this is also the case in type systems with context-free session types, which rely on *kinds* (the "types of types") to distinguish between functional types and session types, and control the linearity of variables. Clarifying this question is therefore a natural avenue for future work.

Bounded quantification Despite being usually orthogonal, subtyping and parametric polymorphism can be deliberately coupled in the form of *bounded quantification* [17]. This feature allows placing a subtyping constraint on a universally quantified variable (written $\forall\alpha \leq T.\, U$), effectively limiting types with which a polymorphic type can be instantiated (subtypes of T only). This results in significantly more expressive type systems: whereas an unbounded polymorphic function cannot depend on the type of its argument, the type information provided by a subtype constraint allows a bounded polymorphic functions to safely act upon its argument as if it had the type specified by the bound (e.g., access the name field if the bound is Person). Bounded quantification in the presence of regular session types has previously been investigated by Gay [40]. Extending his work to the context-free setting is another natural path to follow.

Bibliography

[1] Fides Aarts and Frits W. Vaandrager. Learning I/O automata. In Paul Gastin and François Laroussinie, editors, *CONCUR 2010 - Concurrency Theory, 21th International Conference, CONCUR 2010, Paris, France, August 31-September 3, 2010. Proceedings*, volume 6269 of *Lecture Notes in Computer Science*, pages 71–85. Springer, 2010.

[2] Bernardo Almeida, Andreia Mordido, Peter Thiemann, and Vasco T. Vasconcelos. Polymorphic lambda calculus with context-free session types. *Inf. Comput.*, 289(Part):104948, 2022.

[3] Bernardo Almeida, Andreia Mordido, and Vasco T. Vasconcelos. Kind inference for the FreeST programming language.

[4] Bernardo Almeida, Andreia Mordido, and Vasco T. Vasconcelos. FreeST: Context-free session types in a functional language. In Francisco Martins and Dominic Orchard, editors, *Proceedings Programming Language Approaches to Concurrency- and Communication-cEntric Software, PLACES@ETAPS 2019, Prague, Czech Republic, 7th April 2019*, volume 291 of *EPTCS*, pages 12–23, 2019.

[5] Bernardo Almeida, Andreia Mordido, and Vasco T. Vasconcelos. Deciding the bisimilarity of context-free session types. In *TACAS@ 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings, Part II*, volume 12079 of *Lecture Notes in Computer Science*, pages 39–56. Springer, 2020.

[6] Rajeev Alur, Thomas A. Henzinger, Orna Kupferman, and Moshe Y. Vardi. Alternating refinement relations. In Davide Sangiorgi and Robert de Simone, editors, *CONCUR '98: Concurrency Theory, 9th International Conference, Nice, France, September 8-11, 1998, Proceedings*, volume 1466 of *Lecture Notes in Computer Science*, pages 163–178. Springer, 1998.

[7] Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4):575–631, 1993.

[8] Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. *ACM Trans. Program. Lang. Syst.*, 15(4):575–631, 1993.

[9] Jos C. M. Baeten, Jan A. Bergstra, and Jan Willem Klop. Decidability of bisimulation equivalence for processes generating context-free languages. *J. ACM*, 40(3):653–682, 1993.

[10] Jos C. M. Baeten, D. A. van Beek, Bas Luttik, Jasen Markovski, and Jacobus E. Rooda. A process-theoretic approach to supervisory control theory. In *American Control Conference, ACC 2011, San Francisco, CA, USA, June 29 - July 1, 2011*, pages 4496–4501. IEEE, 2011.

[11] Diogo Barros, Andreia Mordido, Vasco T. Vasconcelos, and Bernardo Almeida. Sharing the stateful world. In Mário Freire and Matilde Pato, editors, *INForum 2022: Atas do 13º Simpósio de Informática, 8 e 9 de setembro de 2022, Guarda, Portugal*, pages 27–38, 2022.

[12] Diogo Barros, Andreia Mordido, Vasco T. Vasconcelos, and Bernardo Almeida. Sharing the stateful world. In Mário Freire and Matilde Pato, editors, *INForum 2022: Atas do 13º Simpósio de Informática 8 e 9 de setembro de 2022, Guarda, Portugal*, pages 27–38, 2022.

[13] Michael Brandt and Fritz Henglein. Coinductive axiomatization of recursive type equality and subtyping. *Fundamenta Informaticae*, 33(4):309–338, 1998.

[14] J. Richard Büchi. *On a Decision Method in Restricted Second Order Arithmetic*, pages 425–435. Springer New York, New York, NY, 1990.

[15] Olaf Burkart, Didier Caucal, and Bernhard Steffen. An elementary bisimulation decision procedure for arbitrary context-free processes. In Jirí Wiedermann and Petr Hájek, editors, *Mathematical Foundations of Computer Science 1995, 20th International Symposium, MFCS'95, Prague, Czech Republic, August 28 - September 1, 1995, Proceedings*, volume 969 of *Lecture Notes in Computer Science*, pages 423–433. Springer, 1995.

[16] Marco Carbone, Kohei Honda, and Nobuko Yoshida. Structured communication-centred programming for web services. In Rocco De Nicola, editor, *Programming Languages and Systems, 16th European Symposium on Programming, ESOP 2007, Held as Part of the Joint European Conferences on Theory and Practics of Software, ETAPS 2007, Braga, Portugal, March 24 - April 1, 2007, Proceedings*, volume 4421 of *Lecture Notes in Computer Science*, pages 2–17. Springer, 2007.

[17] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Comput. Surv.*, 17(4):471–522, 1985.

[18] Giuseppe Castagna and Alain Frisch. A gentle introduction to semantic subtyping. In *Proceedings of the 7th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, July 11-13 2005, Lisbon, Portugal*, pages 198–199. ACM, 2005.

[19] Giuseppe Castagna, Kim Nguyen, Zhiwu Xu, Hyeonseung Im, Sergueï Lenglet, and Luca Padovani. Polymorphic functions with set-theoretic types: part 1: syntax, semantics, and

evaluation. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, page 5–17, 2014.

[20] Søren Christensen, Hans Hüttel, and Colin Stirling. Bisimulation equivalence is decidable for all context-free processes. *Inf. Comput.*, 121(2):143–148, 1995.

[21] Alonzo Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58(2):345–363, 1936.

[22] Koen Claessen and John Hughes. QuickCheck: a lightweight tool for random testing of haskell programs. In Martin Odersky and Philip Wadler, editors, *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18-21, 2000*, pages 268–279. ACM, 2000.

[23] Diana Costa, Andreia Mordido, Diogo Poças, and Vasco T. Vasconcelos. Higher-order context-free session types in system F. In Marco Carbone and Rumyana Neykova, editors, *Proceedings of the 13th International Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software, PLACES@ETAPS 2022, Munich, Germany, 3rd April 2022*, volume 356 of *EPTCS*, pages 24–35, 2022.

[24] Bruno Courcelle. Fundamental properties of infinite trees. *Theor. Comput. Sci.*, 25:95–169, 1983.

[25] H. B. Curry. Grundlagen der kombinatorischen logik. *American Journal of Mathematics*, 52(3):509–536, 1930.

[26] Nils Anders Danielsson and Thorsten Altenkirch. Subtyping, declaratively. In *10th International Conference on Mathematics of Program Construction (MPC 2010)*, pages 100–118, Québec City, Canada, June 2010. Springer LNCS 6120.

[27] Ankush Das, Henry DeYoung, Andreia Mordido, and Frank Pfenning. Nested session types. In Nobuko Yoshida, editor, *Programming Languages and Systems - 30th European Symposium on Programming, ESOP 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings*, volume 12648 of *Lecture Notes in Computer Science*, pages 178–206. Springer, 2021.

[28] Ankush Das, Henry DeYoung, Andreia Mordido, and Frank Pfenning. Subtyping on nested polymorphic session types. *CoRR*, abs/2103.15193, 2021.

[29] Ankush Das and Frank Pfenning. Rast: A language for resource-aware session types. *Log. Methods Comput. Sci.*, 18(1), 2022.

[30] Romain Demangeon and Kohei Honda. Full abstraction in a subtyped pi-calculus with linear types. In Joost-Pieter Katoen and Barbara König, editors, *CONCUR 2011 - Concurrency*

Theory - 22nd International Conference, CONCUR 2011, Aachen, Germany, September 6-9, 2011. Proceedings, volume 6901 of *Lecture Notes in Computer Science*, pages 280–296. Springer, 2011.

[31] Edsger W. Dijkstra. Solution of a problem in concurrent programming control. *Commun. ACM*, 8(9):569, 1965.

[32] Edsger W. Dijkstra. Over de sequentialiteit van procesbeschrijvingen. circulated privately, n.d.

[33] Stephen Dolan. *Algebraic Subtyping: Distinguished Dissertation 2017*. BCS, Swindon, GBR, 2017.

[34] ECMA. *ECMA-404: The JSON Data Interchange Syntax*. ECMA (European Association for Standardizing Information and Communication Systems), Geneva, Switzerland, second edition, December 2017.

[35] Ignacio Fábregas, David de Frutos-Escrig, and Miguel Palomino. Non-strongly stable orders also define interesting simulation relations. In Alexander Kurz, Marina Lenisa, and Andrzej Tarlecki, editors, *Algebra and Coalgebra in Computer Science, Third International Conference, CALCO 2009, Udine, Italy, September 7-10, 2009. Proceedings*, volume 5728 of *Lecture Notes in Computer Science*, pages 221–235. Springer, 2009.

[36] Simon Fowler, Sam Lindley, J. Garrett Morris, and Sára Decova. Exceptional asynchronous session types: session types without tiers. *Proc. ACM Program. Lang.*, 3(POPL):28:1–28:29, 2019.

[37] 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.

[38] Emily P. Friedman. The inclusion problem for simple languages. *Theor. Comput. Sci.*, 1(4):297–316, 1976.

[39] Simon Gay. Subtyping between standard and linear function types. Technical report, University of Glasgow, 2006.

[40] Simon J. Gay. Bounded polymorphism in session types. *Math. Struct. Comput. Sci.*, 18(5):895–930, 2008.

[41] Simon J. Gay. Subtyping supports safe session substitution. In Sam Lindley, Conor McBride, Philip W. Trinder, and Donald Sannella, editors, *A List of Successes That Can*

Change the World - Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday, volume 9600 of *Lecture Notes in Computer Science*, pages 95–108. Springer, 2016.

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

[43] Simon J. Gay, Peter Thiemann, and Vasco T. Vasconcelos. Duality of session types: The final cut. In Stephanie Balzer and Luca Padovani, editors, *Proceedings of the 12th International Workshop on Programming Language Approaches to Concurrency- and Communication-cEntric Software, PLACES@ETAPS 2020, Dublin, Ireland, 26th April 2020*, volume 314 of *EPTCS*, pages 23–33, 2020.

[44] Jean-Yves Girard. Une extension de l'interpretation de gödel à l'analyse, et son application a l'elimination des coupures dans l'analyse et la theorie des types. In J.E. Fenstad, editor, *Proceedings of the Second Scandinavian Logic Symposium*, volume 63 of *Studies in Logic and the Foundations of Mathematics*, pages 63–92. Elsevier, 1971.

[45] Jean-Yves Girard. Linear logic. *Theor. Comput. Sci.*, 50:1–102, 1987.

[46] Sheila A. Greibach. A new normal-form theorem for context-free phrase structure grammars. *J. ACM*, 12(1):42–52, 1965.

[47] Jan Friso Groote and Hans Hüttel. Undecidable equivalences for basic process algebra. *Inf. Comput.*, 115(2):354–371, 1994.

[48] Patrick Henry and Géraud Sénizergues. Lalblc a program testing the equivalence of dpda's. In Stavros Konstantinidis, editor, *Implementation and Application of Automata*, pages 169–180, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[49] Yoram Hirshfeld, Mark Jerrum, and Faron Moller. A polynomial algorithm for deciding bisimilarity of normed context-free processes. *Theor. Comput. Sci.*, 158(1&2):143–159, 1996.

[50] Paul R. Holser. junit-quickcheck: Property-based testing, JUnit-style.

[51] 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.

[52] 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.

[53] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In George C. Necula and Philip Wadler, editors, *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, pages 273–284. ACM, 2008.

[54] Ross Horne and Luca Padovani. A logical account of subtyping for session types.

[55] Raymond Hu, Dimitrios Kouzapas, Olivier Pernet, Nobuko Yoshida, and Kohei Honda. Type-safe eventful sessions in Java. In Theo D'Hondt, editor, *ECOOP 2010 - Object-Oriented Programming, 24th European Conference, Maribor, Slovenia, June 21-25, 2010. Proceedings*, volume 6183 of *Lecture Notes in Computer Science*, pages 329–353. Springer, 2010.

[56] Raymond Hu, Rumyana Neykova, Nobuko Yoshida, Romain Demangeon, and Kohei Honda. Practical interruptible conversations - distributed dynamic verification with session types and Python. In Axel Legay and Saddek Bensalem, editors, *Runtime Verification - 4th International Conference, RV 2013, Rennes, France, September 24-27, 2013. Proceedings*, volume 8174 of *Lecture Notes in Computer Science*, pages 130–148. Springer, 2013.

[57] Raymond Hu, Nobuko Yoshida, and Kohei Honda. Session-based distributed programming in Java. In Jan Vitek, editor, *ECOOP 2008 - Object-Oriented Programming, 22nd European Conference, Paphos, Cyprus, July 7-11, 2008, Proceedings*, volume 5142 of *Lecture Notes in Computer Science*, pages 516–541. Springer, 2008.

[58] Petr Jancar and Faron Moller. Techniques for decidability and undecidability of bisimilarity. In Jos C. M. Baeten and Sjouke Mauw, editors, *CONCUR '99: Concurrency Theory, 10th International Conference, Eindhoven, The Netherlands, August 24-27, 1999, Proceedings*, volume 1664 of *Lecture Notes in Computer Science*, pages 30–45. Springer, 1999.

[59] Thomas Bracht Laumann Jespersen, Philip Munksgaard, and Ken Friis Larsen. Session types for rust. In Patrick Bahr and Sebastian Erdweg, editors, *Proceedings of the 11th ACM SIGPLAN Workshop on Generic Programming, WGP@ICFP 2015, Vancouver, BC, Canada, August 30, 2015*, pages 13–22. ACM, 2015.

[60] Simon Peyton Jones. *Haskell 98 language and libraries: the revised report*. Cambridge University Press, 2003.

[61] S. C. Kleene. *Representation of Events in Nerve Nets and Finite Automata*. RAND Corporation, Santa Monica, CA, 1951.

[62] A. J. Korenjak and John E. Hopcroft. Simple deterministic languages. In *7th Annual Symposium on Switching and Automata Theory, Berkeley, California, USA, October 23-25, 1966*, pages 36–46. IEEE Computer Society, 1966.

[63] Zeeshan Lakhani, Ankush Das, Henry DeYoung, Andreia Mordido, and Frank Pfenning. Polarized subtyping. In *Programming Languages and Systems: 31st European Symposium on Programming, ESOP 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2–7, 2022, Proceedings*, pages 431–461. Springer International Publishing Cham, 2022.

[64] Kim Guldstrand Larsen and Bent Thomsen. A modal process logic. In *Proceedings of the Third Annual Symposium on Logic in Computer Science (LICS '88), Edinburgh, Scotland, UK, July 5-8, 1988*, pages 203–210. IEEE Computer Society, 1988.

[65] Barbara Liskov. Keynote address - data abstraction and hierarchy. In Leigh R. Power and Zvi Weiss, editors, *Addendum to the Proceedings on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 1987 Addendum, Orlando, Florida, USA, October 4-8, 1987*, pages 17–34. ACM, 1987.

[66] Barbara Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.*, 16(6):1811–1841, 1994.

[67] David Maciver and Zac Hatfield-Dodds. Hypothesis: A new approach to property-based testing. *J. Open Source Softw.*, 4(43):1891, 2019.

[68] Karl Mazurak, Jianzhou Zhao, and Steve Zdancewic. Lightweight linear types in system Fdeg. In Andrew Kennedy and Nick Benton, editors, *Proceedings of TLDI 2010: 2010 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, Madrid, Spain, January 23, 2010*, pages 77–88. ACM, 2010.

[69] Robin Milner. An algebraic definition of simulation between programs. In D. C. Cooper, editor, *Proceedings of the 2nd International Joint Conference on Artificial Intelligence. London, UK, September 1-3, 1971*, pages 481–489. William Kaufmann, 1971.

[70] Robin Milner. A theory of type polymorphism in programming. *J. Comput. Syst. Sci.*, 17(3):348–375, 1978.

[71] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, 1980.

[72] Robin Milner. Functions as processes. *Math. Struct. Comput. Sci.*, 2(2):119–141, 1992.

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

[74] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, II. *Inf. Comput.*, 100(1):41–77, 1992.

[75] Robin Milner, Mads Tofte, and Robert Harper. *Definition of standard ML*. MIT Press, 1990.

[76] Rumyana Neykova. Session types go dynamic or how to verify your python conversations. In Nobuko Yoshida and Wim Vanderbauwhede, editors, *Proceedings 6th Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software, PLACES 2013, Rome, Italy, 23rd March 2013*, volume 137 of *EPTCS*, pages 95–102, 2013.

[77] Nicholas Ng, Nobuko Yoshida, and Kohei Honda. Multiparty session C: safe parallel programming with message optimisation. In Carlo A. Furia and Sebastian Nanz, editors, *Objects, Models, Components, Patterns - 50th International Conference, TOOLS 2012, Prague, Czech Republic, May 29-31, 2012. Proceedings*, volume 7304 of *Lecture Notes in Computer Science*, pages 202–218. Springer, 2012.

[78] R. Nilsson. *ScalaCheck: The Definitive Guide*. IT Pro. Artima Press, 2014.

[79] Massachusetts Institute of Technology. Artificial Intelligence Laboratory, G.J. Sussman, and G.L. Steele. *Scheme: an interpreter for extended lambda calculus*. AI Memo No. Massachusetts Institute of Technology, Artificial Intelligence Laboratory, 1975.

[80] Dominic A. Orchard and Nobuko Yoshida. Effects as sessions, sessions as effects. In Rastislav Bodík and Rupak Majumdar, editors, *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 568–581. ACM, 2016.

[81] Luca Padovani. A simple library implementation of binary sessions. *J. Funct. Program.*, 27:e4, 2017.

[82] Luca Padovani. Context-free session type inference. *ACM Trans. Program. Lang. Syst.*, 41(2):9:1–9:37, 2019.

[83] David Michael Ritchie Park. Concurrency and automata on infinite sequences. In Peter Deussen, editor, *Theoretical Computer Science, 5th GI-Conference, Karlsruhe, Germany, March 23-25, 1981, Proceedings*, volume 104 of *Lecture Notes in Computer Science*, pages 167–183. Springer, 1981.

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

[85] Benjamin C. Pierce and Davide Sangiorgi. Typing and subtyping for mobile processes. *Math. Struct. Comput. Sci.*, 6(5):409–453, 1996.

[86] Diogo Poças, Diana Costa, Andreia Mordido, and Vasco T. Vasconcelos. System F^{μ}_{ω} with context-free session types. In Thomas Wies, editor, *Programming Languages and Systems - 32nd European Symposium on Programming, ESOP 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2023, Paris, France, April 22-27, 2023, Proceedings*, volume 13990 of *Lecture Notes in Computer Science*, pages 392–420. Springer, 2023.

[87] John C. Reynolds. Towards a theory of type structure. In Bernard J. Robinet, editor, *Programming Symposium, Proceedings Colloque sur la Programmation, Paris, France, April 9-11, 1974*, volume 19 of *Lecture Notes in Computer Science*, pages 408–423. Springer, 1974.

[88] Davide Sangiorgi. *Introduction to Bisimulation and Coinduction*. Cambridge University Press, 2011.

[89] Kurt Schelfthout. FsCheck: Random testing for .NET.

[90] Gil Silva, Andreia Mordido, and Vasco T. Vasconcelos. Subtyping context-free session types. In Guillermo A. Pérez and Jean-François Raskin, editors, *34th International Conference on Concurrency Theory, CONCUR 2023, September 18-23, 2023, Antwerp, Belgium*, volume 279 of *LIPIcs*, pages 11:1–11:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023.

[91] Gil Silva, Andreia Mordido, and Vasco T. Vasconcelos. Subtyping context-free session types. Cornell University (ArXiv), 2023.

[92] Gil Silva, Andreia Mordido, and Vasco T. Vasconcelos. Subtyping context-free session types. In João Leitão and Luís Veiga, editors, *Atas do 14º INForum — Simpósio de Informática*, pages 153–155. Cornell University (ArXiv), 2023.

[93] Michael Sipser. *Introduction to the theory of computation*. PWS Publishing Company, 1997.

[94] Kaku Takeuchi, Kohei Honda, and Makoto Kubo. An interaction-based language and its typing system. In Constantine Halatsis, Dimitris G. Maritsas, George Philokyprou, and Sergios Theodoridis, editors, *PARLE '94: Parallel Architectures and Languages Europe, 6th International PARLE Conference, Athens, Greece, July 4-8, 1994, Proceedings*, volume 817 of *Lecture Notes in Computer Science*, pages 398–413. Springer, 1994.

[95] The FreeST Team. FreeST homepage. `https://freest-lang.github.io/`. Accessed: August 2024.

[96] 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.

[97] Alan M. Turing. Computability and λ-definability. *J. Symb. Log.*, 2(4):153–163, 1937.

[98] Alan M. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proc. London Math. Soc.*, s2-42(1):230–265, 1937.

[99] Rob J. van Glabbeek. The linear time - branching time spectrum II. 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 66–81. Springer, 1993.

[100] Hongwei Xi, Zhiqiang Ren, Hanwen Wu, and William Blair. Session types in a linearly typed multi-threaded lambda-calculus. *CoRR*, abs/1603.03727, 2016.

Appendix A

Groundwork for the proofs

A.0.1 Substitution

The following lemma shows that we can discard type references from type formation contexts, under the assumption that they do not occur free in the type in question.

Lemma A.0.1 (Type strengthening)**.** *Let* $x \notin \mathsf{free}(T)$*. If* $\Delta, x \vdash T$*, then* $\Delta \vdash T$*.*

Proof. By rule induction on the hypothesis. □

The following lemma shows that substitution preserves the good properties of types: termination, contractivity and type formation. From it follows that these properties are also preserved by the unr function (Definition 6.1.1).

Lemma A.0.2 (Type substitution)**.** *Suppose that* $\Delta \vdash U$*.*

1. *If* $\Delta, x \vdash T$ *and* $T\checkmark$*, then* $[U/x]T\checkmark$*.*
2. *If* $\Delta, x \vdash T$ *and* $T\not\downarrow$*, then* $[U/x]T\not\downarrow$*.*
3. *If* $\Delta, x, y \vdash T$ *and* T contr y *and* $y \notin \mathsf{free}(U)$*, then* $[U/x]T$ contr y*.*
4. *If* $\Delta, x \vdash T$ *then* $\Delta \vdash [U/x]T$*.*

Proof.

1. By rule induction on $T\checkmark$.
2. By structural induction on T. All cases are either straightforward or follow from the induction hypothesis.
3. By rule induction on T contr y, using Item 1 and Item 2. All cases follow from the induction hypothesis except the case for rule C-VAR, where we have $T = z$ with $z \neq x, y$, and where the result follows from hypothesis z contr y.
4. By rule induction on $\Delta, x \vdash T$. For the case TF-REC we have $T = \mu y.V$. The premises to the rule are $V\not\downarrow$, V contr y and $\Delta, x, y \vdash V$. Induction on the third premise gives $\Delta, y \vdash [U/x]V$. Item 2 gives $[U/x]V\not\downarrow$, while Item 3 gives $[U/x]V$ contr y. Rule TF-REC gives $\Delta \vdash \mu y.([U/x]V)$. Conclude with the definition of substitution. For the case TF-VAR with $T = y \neq x$, we have $x \notin \mathsf{free}(y)$. The result follows from hypothesis $\Delta, x \vdash y$ and strengthening. For TF-VAR with $T = x$ the result follows from the hypothesis $\Delta \vdash U$.

□

A.0.2 Unraveling

By inspecting definition of the unr function (Definition 6.1.1), we get the following immediate result.

Proposition A.0.1. *If* $T = \mathsf{unr}(T)$*, then* T *is one of* Unit*,* $U \xrightarrow{m} V$*,* $(\!|\ell\colon T_\ell|\!)_{\ell\in L}$*,* x*,* $\sharp U$*,* $\odot\{\ell\colon S_\ell\}_{\ell\in L}$*,* $\sharp U;S$*,* Skip *or* End*. If* $T \neq \mathsf{unr}(T)$*, then* T *is one of* $\mu x.U$*,* $\odot\{\ell\colon S_\ell\}_{\ell\in L};S$*,* $(S_1;S_2);S_3$*,* $\mathsf{Skip};S$*,* $\mathsf{End};S$ *or* $(\mu s.S);R$*.*

We can also define the notion of one-step unraveling for our types.

Definition A.0.1. We say that a type T' is a *one-step unraveling* of another type T, denoted $\mathsf{unr1}(T)$, if: T is a direct application of a type constructor, and $T' = T$; or T is not a direct application of a type constructor, and T' is obtained by one recursive call of the unr function, which attempts to bring a type constructor into the front of a type.

One example is $\mathsf{unr1}(\mathsf{Skip};S) = S$; another example is $\mathsf{unr1}((S_1;S_2);S_3) = S_1;(S_2;S_3)$. Notice that T_0 is contractive iff any sequence $T_0, T_1, ...$, where $T_{i+1} = \mathsf{unr1}(T_i)$, eventually stabilises in $\mathsf{unr}(T)$ (after finitely many steps).

Finally, we make some observations about the structure of subtyping derivations. We can classify the syntactic subtyping rules from Fig. 5.1 in three groups: *progressing*, *left-preserving* and *right-preserving*:

- We designate rules S-UNIT, S-ARROW, S-RCD, S-VRT, S-IN, S-OUT, S-INTCHOICE, S-EXTCHOICE, S-END, S-SKIP, S-ENDSEQ1L, S-ENDSEQ1R, S-ENDSEQ2, S-INSEQ1L, S-OUTSEQ1L, S-INSEQ1R, S-OUTSEQ1R, S-INSEQ2 and S-OUTSEQ2 as *progressing*. These rules consume the types on both sides of the relation, i.e., if we apply one of these rules from judgement $T \leq U$, we end up with judgements $T' \leq U'$ where T', U' are both proper subterms of T, U. Moreover, they are applicable iff $T = \mathsf{unr}(T)$ and $U = \mathsf{unr}(U)$.
- We designate rules S-RECL, S-SKIPSEQL, S-CHOICESEQL, S-SEQSEQL, S-RECSEQL as *right-preserving*. These rules change the type on the left-hand side of the relation, but preserve the type on the right-hand side. They are applicable when $T \neq \mathsf{unr}(T)$.
- We designate rules S-RECR, S-SKIPSEQR, S-CHOICESEQR, S-SEQSEQR, S-RECSEQR are *left-preserving*. These rules change the type on the right-hand side of the relation, but preseve the type on the left-hand side. They are applicable when $U \neq \mathsf{unr}(U)$.

Furthermore, by inspecting the rules, we can gather that:

- If we can apply a progressing rule for $T \leq U$, then it is the only applicable rule.
- If we can apply a left-preserving rule for $T \leq U$, then this the only left-preserving rule that can be applied (but a right-preserving rule may also be applicable).
- If we can apply a right-preserving rule for $T \leq U$, then this the only right-preserving rule that can be applied (but a left-preserving rule may also be applicable).
- If we can apply both a left-preserving rule and a right-preserving rule for $T \leq U$, then we can apply them one after the other in any order. Furthermore, both rules must eventually be applied in any successful derivation for $T \leq U$.

From these observations we can immediately derive the following results.

Lemma A.0.3.

1. *Let* $T' = \mathsf{unr1}(T)$ *for some type* T.
   (a) *If* $\Delta \vdash T$, *then* $\Delta \vdash T'$.
   (b) $T \leq U$ *iff* $T' \leq U$.
   (c) $T \xrightarrow{a} U$ *iff* $T' \xrightarrow{a} U$.
2. *Let* $T' = \mathsf{unr}(T)$ *for some type* T.
   (a) *If* $\Delta \vdash T$, *then* $\Delta \vdash T'$.
   (b) $T \leq U$ *iff* $T' \leq U$.
   (c) $T \xrightarrow{a} U$ *iff* $T' \xrightarrow{a} U$.

Proof. Sub-item 1.a is immediate by inspection of the type formation rules. Sub-item 1.b follows from the preceding discussion. Sub-item 1.c is immediate by inspection of the LTS rules (Fig. 4.4). Item 2 follows from Item 1 since $\mathsf{unr}(T)$ is reached in a finite number of steps. □

Appendix B

Proof of Theorem 5.1.1

Theorem 5.1.1. *The syntactic subtyping relation $\leq$ is a preorder on types.*

Proof. We need to prove reflexivity and transitivity.

Reflexivity We prove by coinduction that $T \leq T$ for every type T s.t. $\vdash T$. Consider the following relation.

$$\begin{aligned}\mathcal{R} = &\{(T,T) \mid \vdash T\} \\ &\cup \{(T,T') \mid \vdash T \text{ and } T' = \mathsf{unr1}(T)\}\end{aligned}$$

We shall prove that $\mathcal{R}$ is backward-closed for the rules of syntactic subtyping. This will show that $\mathcal{R} \subseteq \leq$ and, consequently, that $T \leq T$ for every type T.

Let $(T,T) \in \mathcal{R}$. We consider first the cases in which T fits a type constructor, i.e., $T = \mathsf{unr}(T)$. Given that T is well-formed, we have the following case analysis for it:

(**Case** $T = \mathsf{Unit}$): We apply axiom S-UNIT to (T,T).

(**Case** $T = U \xrightarrow{m} V$): We apply rule S-ARROW to (T,T), arriving at goals (U,U) and (V,V). Since the derivation of $\vdash T$ must use rule TF-ARROW, we also have that $\vdash U$ and $\vdash V$, and therefore that $(U,U),(V,V) \in \mathcal{R}$.

(**Case** $T = \{\ell\colon T_\ell\}_{\ell\in L}$): We apply rule (S-RCD and arrive at goals (T_k,T_k) for each $k \in L$. The derivation of $\vdash T$ must use rule TF-RCDVRT, which implies that $\vdash T_k$ for each $k \in L$, which means that $(T_k,T_k) \in \mathcal{R}$ for each $k \in \mathcal{R}$.

(**Case** $T = \langle \ell\colon T_\ell \rangle_{\ell\in L}$): Analogous to case $T = \{\ell\colon T_\ell\}_{\ell\in L}$.

(**Case** $T = x$): Cannot occur, for $\not\vdash x$.

(**Case** $T = \mathsf{End}$): We apply axiom S-END to (T,T).

(**Case** $T = ?U$): We apply rule S-IN to (T,T), arriving at goal (U,U). Since the derivation of $\vdash T$ must use rule TF-MSG, we have that $\vdash U$, and therefore that $(U,U) \in \mathcal{R}$.

(**Case** $T = \oplus\{\ell\colon T_\ell\}_{\ell\in L}$): Analogous to case $T = \{\ell\colon T_\ell\}_{\ell\in L}$.

(**Case** $T = \&\{\ell\colon T_\ell\}_{\ell\in L}$): Analogous to case $T = \{\ell\colon T_\ell\}_{\ell\in L}$.

(**Case** $T = \mathsf{Skip}$): We apply axiom S-SKIP to (T,T).

(**Case** $T = ?U;S$): We apply rule S-INSEQ2, arriving at goals $(U,U),(S,S)$. The derivation of $\vdash T$ must use rule TF-SEQ, implying $\vdash ?U$ and $\Delta \vdash S$. Moreover, the derivation of $\vdash ?U$ must

use rule TF-MSG, implying $\vdash U$. Since $\vdash U$ and $\vdash S$, we have $(U,U),(S,S) \in \mathcal{R}$. The case where $T = !U;S$ is similar.

(**Case** $T = s;S$): Cannot occur, since $\not\vdash T$.

Next, we consider cases in which $T \neq \mathsf{unr}(T)$.

(**Case** $T = \mu x.U$): We apply rule S-RECR to (T,T), arriving at goal (T,T') where $T' = [\mu x.U/x]U$. Since $T' = \mathsf{unr1}(T)$, we have that $(T,T') \in \mathcal{R}$.

(**Case** $T = \mathsf{End};S$): We apply axiom S-ENDSEQ2.

(**Case** $T = \odot\{\ell\colon S_\ell\}_{\ell\in L};R$): We apply rule S-CHOICESEQR to (T,T), arriving at goal (T,T') where $T' = \odot\{\ell\colon S_\ell;R\}_{\ell\in L}$. Since $T' = \mathsf{unr1}(T)$, we have that $(T,T') \in \mathcal{R}$.

(**Case** $T = \mathsf{Skip};S$): We apply rule S-SKIPSEQR, arriving at goal (T,S). Since $S = \mathsf{unr1}(T)$, we obtain that $(T,S) \in \mathcal{R}$.

(**Case** $T = (S_1;S_2);S_3$): We apply rule S-SEQSEQR, arriving at goal (T,T') where $T' = S_1;(S_2;S_3)$. Since $T' = \mathsf{unr1}(T)$, we obtain that $(T,T') \in \mathcal{R}$.

(Case $T = (\mu s.S);R$**):** We apply rule S-RECSEQR to (T,T), arriving at goal (T,T'), where $T' = ([\mu s.S/s]S);R$. Since $T' = \mathsf{unr1}(T)$, we have that $(T,T') \in \mathcal{R}$.

Next, we must consider cases $(T,T') \in \mathcal{R}$ where $T \neq T'$, which means, by definition, that $T' = \mathsf{unr1}(T)$ and therefore that $T \neq \mathsf{unr}(T)$. Given that $\vdash T$, we have the following case analysis for T.

(**Case** $T = \mu x.U$): From Lemma A.0.2 follows that $\vdash [\mu x.U/x]U$. Since $T' = \mathsf{unr1}(T)$, we know that $T' = [\mu x.U/x]U$. We apply rule S-RECL to (T,T'), arriving at goal $(T',T') \in \mathcal{R}$.

(**Case** $T = \mathsf{End};S$): Then $T' = \mathsf{End}$. We apply axiom S-ENDSEQ1L to (T,T').

(**Case** $T = \odot\{\ell\colon S_\ell\}_{\ell\in L};R$): The derivation of $\vdash T$ must use rules TF-SEQ and TF-CHOICE, implying that $\vdash S_k$ for each $k \in L$ and $\vdash U$. Again by rule TF-SEQ, we get that $\vdash S_k;R$ for each $k \in L$ and thus, by rule TF-CHOICE, $\vdash \odot\{\ell\colon S_\ell;R\}_{\ell\in L}$. Since $T' = \mathsf{unr1}(T)$, we know that $T' = \odot\{\ell\colon S_\ell;R\}_{\ell\in L}$. We apply rule S-CHOICESEQL to (T,T'), arriving at goal $(T',T') \in \mathcal{R}$.

(**Case** $T = \mathsf{Skip};S$): The derivation of $\vdash T$ must use rule TF-SEQ, implying that $\vdash S$. Since $T' = \mathsf{unr1}(T)$, we know that $T' = S$. We apply rule E-SKIPSEQL to (T,T'), arriving at goal $(T',T') \in \mathcal{R}$.

(**Case** $T = (S_1;S_2);S_3$): The derivation of $\vdash T$ must use rule TF-SEQ, hence $\vdash S_1$, $\vdash S_2$, $\vdash S_3$. Therefore, by rule TF-SEQ also, we have $\vdash S_1;(S_2;S_3)$. Since $T' = \mathsf{unr1}(T)$, we know that $T' = S_1;(S_2;S_3)$. We apply rule S-SEQSEQL to (T,T'), arriving at goal $(T',T') \in \mathcal{R}$.

(**Case** $T = (\mu s.S);R$): The derivation of $\vdash T$ must use rule TF-SEQ, implying that $\vdash \mu s.S$ and $\vdash R$. From Lemma A.0.2 follows that $\vdash [\mu s.S/s]S$. By rule TF-SEQ we get that $\vdash ([\mu s.S/s]S);R$. Since $T' = \mathsf{unr1}(T)$, we know that $T' = ([\mu s.S/s]S);R$. We apply rule S-RECSEQL to (T,T'), arriving at goal $(T',T') \in \mathcal{R}$.

Transitivity We now prove by coinduction that, for all types T,U,V with $\vdash T, \vdash U, \vdash V$, if $T \leq U$ and $U \leq V$, then $T \leq V$. Consider the following relation.

$$\mathcal{R} = \{(T,V) \mid \vdash T, \vdash V \text{ and there exists } U \text{ s.t.} \vdash U, T \leq U \text{ and } U \leq V\}$$

We prove that $\mathcal{R}$ is backward closed for the rules of the syntactic subtyping relation, showing that $\mathcal{R} \subseteq \leq$. This will give the desired property.

We begin by assuming that no left-preserving rule applies to judgements $T \leq U$. Otherwise, we could apply its symmetric counterpart to judgement $U \leq V$ to get a different U' for which $T \leq U'$ and $U' \leq V$. Without loss of generality, our derivation for $T \leq U$ starts with a finite sequence of left-preserving rules until we reach a type U' that can only be consumed. We could then reach the same type U' by applying a symmetric sequence of right-preserving rules to the derivation for $U \leq V$. We can therefore assume that U is ready to be consumed.

Suppose now a derivation for $T \leq U$ starting with a right-preserving rule, after which we get judgement $T' \leq U$ for some type T'. Here we can apply the corresponding rule to (T, V), arriving at (T', V), which is in $\mathcal{R}$ since $T' \leq U$ and $U \leq V$. The case where $U \leq V$ starts with a left-preserving rule can be handled similarly.

What if both derivations for $T \leq U$ and $U \leq V$ start with a progressing rule? In this case, we need to inspect which rule is at the start of the derivation for $T \leq U$.

(**Case** S-UNIT): Then $T = \mathsf{Unit}$ and $U = \mathsf{Unit}$. The only progressing rule that can be applied at $U \leq V$ is also S-UNIT, implying that $V = \mathsf{Unit}$ as well. Therefore, we can apply axiom S-UNIT to (T, V).

(**Case** S-ARROW): Then $T = T_1 \xrightarrow{m} T_2$ and $U = U_1 \xrightarrow{n} U_2$ for some T_1, T_2, U_1, U_2, m, n. Furthermore, we have $U_1 \leq T_1$ and $T_2 \leq U_2$ and $m \sqsubseteq n$. The only progressing rule that can be applied to $U \leq V$ is also S-ARROW, implying that $V = V_1 \xrightarrow{o} V_2$ for some V_1, V_2, o. Furthermore, we have $V_1 \leq U_1$, $U_2 \leq V_2$ and $n \sqsubseteq o$. By transitivity of $\sqsubseteq$ we obtain $m \sqsubseteq o$. We apply rule S-ARROW to (T, V), arriving at goals $(V_1, T_1), (T_2, V_2) \in \mathcal{R}$.

(**Case** S-RCD): Then $T = \{\ell\colon T_\ell\}_{\ell \in L}$ and $U = \{k\colon U_k\}_{k \in K}$ for some $L, K, T_i, U_j, i \in L, j \in K$. Furthermore, we have $K \subseteq L$, $T_j \leq U_j$ for $j \in K$. The only progressing rule that can be applied to $U \leq V$ is also S-RCD, implying that $V = \{h\colon V_h\}_{h \in H}$ for some $H, V_h, h \in H$. Furthermore, we have $H \subseteq K$ $U_h \leq V_h$ for each $h \in H$. By transitivity of $\subseteq$ we get $H \subseteq L$. We apply rule S-RCD to (T, V), arriving at goals $(T_h, V_h) \in \mathcal{R}$ for each $h \in H$. Case S-INTCHOICE is similar.

(**Case** S-VRT): Then $T = \langle \ell\colon T_\ell \rangle_{\ell \in L}$ and $U = \langle k\colon U_k \rangle_{k \in K}$ for some $L, K, T_i, U_j, i \in L, j \in K$. Furthermore, we have $L \subseteq K$, $T_j \leq U_j$ for $j \in K$. The only progressing rule that can be applied to $U \leq V$ is also S-VRT, implying that $V = \langle h\colon V_h \rangle_{h \in H}$ for some $H, V_h, h \in H$. Furthermore, we have $K \subseteq H, U_h \leq V_h$ for each $h \in H$. By transitivity of $\subseteq$ we get $L \subseteq H$. We apply rule S-VRT to (T, V), arriving at goals $(T_h, V_h) \in \mathcal{R}$ for each $h \in H$. Case S-EXTCHOICE is similar.

(**Case** S-END): Then $T = \mathsf{End}$ and $U = \mathsf{End}$. The two possible progressing rules for $U \leq V$ are S-END and S-ENDSEQ1R. In the first case we have $V = \mathsf{End}$, so we apply S-END to (T, V). In the second case we have $V = \mathsf{End};S$ for some S, so we apply rule S-ENDSEQ1R to (T, V).

(**Case** S-IN): Then $T = ?T'$ and $U = ?U'$ for some T', U'. Furthermore, we have $T' \leq U'$. The two possible progressing rules for $U \leq V$ are S-IN and S-INSEQ1R. In the first case, we

have $V = ?V'$ for some V'. It follows that $U' \leq V'$. We apply rule S-IN to (T, V), arriving at goal $(T', V') \in \mathcal{R}$. In the second case, we have $V = ?V';S$ for some V' and S. It follows that $U' \leq V'$ and $S \leq \mathsf{Skip}$. We apply rule S-INSEQ1R to (T, V), arriving at goal $(T', V') \in \mathcal{R}$. The cases S-INSEQ1L, S-INSEQ1R, S-INSEQ2 are handled similarly.

(**Case** S-OUT): Then $T = !T'$ and $U = !U'$ for some T', U'. Furthermore, we have $U' \leq T'$. The two possible progressing rules for $U \leq V$ are S-OUT and S-OUTSEQ1R. In the first case, we have $V = !V'$ for some V'. It follows that $V' \leq U'$. We apply rule S-OUT to (T, V), arriving at goal $(V', T') \in \mathcal{R}$. In the second case, we have $V = !V';S$ for some V' and S. It follows that $V' \leq U'$ and $S \leq \mathsf{Skip}$. We apply rule S-OUTSEQ1R to (T, V), arriving at goal $(V', T') \in \mathcal{R}$. The cases S-OUTSEQ1L, S-OUTSEQ1R, S-OUTSEQ2 are handled similarly.

(**Case** S-ENDSEQ1L): Then $T = \mathsf{End};S$ and $U = \mathsf{End}$ for some S. The two possible progressing rules for $U \leq V$ are S-END and S-ENDSEQ1R. In the first case we have $V = \mathsf{End}$, so we can apply rule S-ENDSEQ1L to (T, V). In the second case we have $V = \mathsf{End};R$ for some R, so we can apply rule S-ENDSEQ2 to (T, V).

(**Case** S-ENDSEQ1R): Then $T = \mathsf{End}$ and $U = \mathsf{End};S$ for some S. The two possible progressing rules for $U \leq V$ are S-ENDSEQ1L and S-ENDSEQ2. In the first case we have $V = \mathsf{End}$, so we can apply S-END to (T, V). In the second case we have $V = \mathsf{End};R$ for some R, so we can apply S-ENDSEQ1R to (T, V).

(**Case** S-ENDSEQ2): Then $T = \mathsf{End};S$ and $U = \mathsf{End};R$ for some S, R. The two possible progressing rules for $U \leq V$ are S-ENDSEQ1L and S-ENDSEQ2. In the first case we have $V = \mathsf{End}$, so we can apply S-ENDSEQ1L to (T, V). In the second case we have $V = \mathsf{End};S'$ for some S', so we apply S-ENDSEQ2 to (T, V). □

Appendix C

Proof of Theorem 5.2.1

Theorem 5.2.1. *For any $\mathcal{X}, \mathcal{Y}, \mathcal{Z}, \mathcal{W}$, $\preceq^{\mathcal{X}\mathcal{Y}\mathcal{Z}\mathcal{W}}$ is a preorder relation on types.*

Proof. We need to prove reflexivity and transitivity.

To prove reflexivity, we need to show that the identity relation, $\{(T, T) \mid T \text{ is a type}\}$, is a subtype simulation. This can be done by simple case analysis on the form of T.

For transitivity, we need to demonstrate that if $\mathcal{R}_1$ and $\mathcal{R}_2$ are $\mathcal{X}\mathcal{Y}\mathcal{Z}\mathcal{W}$-simulations, then their interleaved concatenation,

$$\mathcal{R} = \{(T, V) \mid\vdash T, \vdash V \text{ and there is } U \text{ s.t. } \vdash U \text{ and } T\mathcal{R}_1U \text{ and } U\mathcal{R}_2V \text{ or } T\mathcal{R}_2U \text{ and } U\mathcal{R}_1V\},$$

is itself an $\mathcal{X}\mathcal{Y}\mathcal{Z}\mathcal{W}$-simulation.

We assume a pair $(T, V) \in \mathcal{R}$ and analyze two cases: $T \xrightarrow{a} T'$ for some a, T' and $V \xrightarrow{a} V'$ for some a, V'.

In case $T \xrightarrow{a} T'$ we need to analyze subcases $a \in \mathcal{X}$ and $a \in \mathcal{Z}$ to ensure, respectively, that Items 1 and 3 of Definition 5.2.3 are met.

(**Case** $a \in \mathcal{X}$): then we need to show that $V \xrightarrow{a} V'$ with $T'\mathcal{R}V'$. By definition of $\mathcal{R}$, we know that there is an U such that $T\mathcal{R}_1U$ and $U\mathcal{R}_2V$ or $T\mathcal{R}_2U$ and $U\mathcal{R}_1V$, and therefore that $U \xrightarrow{a} U'$ for some U' with $T'\mathcal{R}_1U'$ or $T'\mathcal{R}_2U'$. In the first case, $U\mathcal{R}_2V$ gives us $V \xrightarrow{a} V'$ for some V' with $U'\mathcal{R}_2V'$, from which we conclude that $(T', V') \in \mathcal{R}$. In the second case, $U\mathcal{R}_1V$ gives us $V \xrightarrow{a} V'$ for some V' with $U'\mathcal{R}_1V'$, from which we conclude that $T'\mathcal{R}V'$. Hence Item 1 of Definition 5.2.3 is met in both cases.

(**Case** $a \in \mathcal{Z}$): then we need to show that $V \xrightarrow{a} V'$ with $V'\mathcal{R}T'$. By definition of $\mathcal{R}$, we know that there is an U such that $T\mathcal{R}_1U$ and $U\mathcal{R}_2V$ or $T\mathcal{R}_2U$ and $U\mathcal{R}_1V$, and therefore that $U \xrightarrow{a} U'$ for some U' with $U'\mathcal{R}_1T'$ or $U'\mathcal{R}_2T'$. In the first case, $U\mathcal{R}_2V$ gives us $V \xrightarrow{a} V'$ for some V' with $V'\mathcal{R}_2U'$, from which we conclude that $(T', V') \in \mathcal{R}$. In the second case, $U\mathcal{R}_1V$ gives us $V \xrightarrow{a} V'$ for some V' with $V'\mathcal{R}_1U'$, from which we conclude that $V'\mathcal{R}T'$. Hence Item 1 of Definition 5.2.3 is met in both cases.

In case $V \xrightarrow{a} V'$ we need to analyze subcases $a \in \mathcal{Y}$ and $a \in \mathcal{W}$ to ensure, respectively, that Items 2 and 4 of Definition 5.2.3 are met.

(**Case** $a \in \mathcal{Y}$): then we need to show that $T \xrightarrow{a} T'$ with $T'\mathcal{R}V'$. By definition of $\mathcal{R}$, we know that there is an U such that $T\mathcal{R}_1U$ and $U\mathcal{R}_2V$ or $T\mathcal{R}_2U$ and $U\mathcal{R}_1V$, and therefore that

$U \xrightarrow{a} U'$ for some U' with $U' \mathcal{R}_2 V'$ or $U' \mathcal{R}_1 V'$. In the first case, $T \mathcal{R}_1 U$ gives us $T \xrightarrow{a} T'$ for some T' with $T' \mathcal{R}_1 U'$, from which we conclude that $(T', V') \in \mathcal{R}$. In the second case, $T \mathcal{R}_2 U$ gives us $T \xrightarrow{a} T'$ for some T' with $T' \mathcal{R}_2 U'$, from which we conclude that $T' \mathcal{R} V'$. Hence Item 2 of Definition 5.2.3 is met in both cases.

(**Case** $a \in \mathcal{W}$): then we need to show that $T \xrightarrow{a} T'$ with $V' \mathcal{R} T'$. By definition of $\mathcal{R}$, we know that there is an U such that $T \mathcal{R}_1 U$ and $U \mathcal{R}_2 V$ or $T \mathcal{R}_2 U$ and $U \mathcal{R}_1 V$, and therefore that $U \xrightarrow{a} U'$ for some U' with $U' \mathcal{R}_2 V'$ or $U' \mathcal{R}_1 V'$. In the first case, $T \mathcal{R}_1 U$ gives us $T \xrightarrow{a} T'$ for some T' with $U' \mathcal{R}_1 T'$, from which we conclude that $(T', V') \in \mathcal{R}$. In the second case, $T \mathcal{R}_2 U$ gives us $T \xrightarrow{a} T'$ for some T' with $U' \mathcal{R}_2 T'$, from which we conclude that $V' \mathcal{R} T'$. Hence Item 2 of Definition 5.2.3 is met in both cases.

□

Appendix D

Proof of Theorem 5.2.2

Theorem 5.2.2 (Soundness and completeness for subtyping relations)**.** *Let* $\vdash T$ *and* $\vdash U$*. Then* $T \leq U$ *iff* $T \lesssim U$*.*

Proof. We analyse both directions of the biconditional separately.

Direct implication Consider the relation $\mathcal{R} = \{(T, U) \mid \vdash T, \vdash U \text{ and } T \leq U\}$. We must show that $\mathcal{R}$ is an $\mathcal{XYZW}$-simulation with $\mathcal{X}, \mathcal{Y}, \mathcal{Z}, \mathcal{W}$ as defined in Definition 5.2.4. This will show that $\mathcal{R} \subseteq \leq$, and hence that $T \leq U$ implies $T \lesssim U$.

The proof has two parts. First consider cases $(T, U) \in \mathcal{R}$ s.t. $\mathsf{unr}(T) = T$ and $\mathsf{unr}(U) = U$. We proceed by case analysis for the last rule in the derivation of $T \leq U$, which must be a progressing rule.

(**Case** S-UNIT): Then $T = \mathsf{Unit}$ and $U = \mathsf{Unit}$. The unique transition that can be applied to Unit is $\mathsf{Unit} \xrightarrow{\mathsf{Unit}} \mathsf{Skip}$ (L-UNIT). Since $\mathsf{Unit} \in \mathcal{X}, \mathcal{Y}$, we should have that if $T \xrightarrow{\mathsf{Unit}} T'$ for some T', then $U \xrightarrow{\mathsf{Unit}} U'$ for some U' with $(T', U') \in \mathcal{R}$, and also that if $U \xrightarrow{\mathsf{Unit}} U'$ for some U', then $T \xrightarrow{\mathsf{Unit}} T'$ for some T' with $(T', U') \in \mathcal{R}$. It is readily verifiable that the single transitions of both T and U match each other. That $(\mathsf{Skip}, \mathsf{Skip}) \in \mathcal{R}$ follows from S-SKIP and the definition of $\mathcal{R}$.

(**Case** S-ARROW): Then $T = T_1 \xrightarrow{m} T_2$, $U = U_1 \xrightarrow{n} U_2$, $U_1 \leq T_1$, $T_2 \leq U_2$ and $m \sqsubseteq n$. The only transitions that can be applied to T are $T \xrightarrow{\to\mathsf{d}} T_1$, $T \xrightarrow{\to\mathsf{r}} T_2$ and, if $m = 1$, $T \xrightarrow{\to 1} \mathsf{Skip}$ (L-ARROWDOM, L-ARROWRNG and L-LINARROW). Similarly, the only transitions applicable to U are $U \xrightarrow{\to\mathsf{d}} U_1$, $U \xrightarrow{\to\mathsf{r}} U_2$ and, if $n = 1$, $U \xrightarrow{\to 1} \mathsf{Skip}$.

Since $\to\mathsf{d} \in \mathcal{Z}, \mathcal{W}$, we should have that if $T \xrightarrow{\to\mathsf{d}} T'$ for some T', then $U \xrightarrow{\to\mathsf{d}} U'$ for some U' with $(U', T') \in \mathcal{R}$, and also that if $U \xrightarrow{\to\mathsf{d}} U'$ for some U', then $T \xrightarrow{\to\mathsf{d}} T'$ for some T' with $(U', T') \in \mathcal{R}$. That the transitions of T and U by $\to\mathsf{d}$ match is readily verifiable, and that $(U_1, T_1) \in \mathcal{R}$ is given by $U_1 \leq T_1$ and the definition of $\mathcal{R}$.

Similarly, since $\to\mathsf{r} \in \mathcal{X}, \mathcal{Y}$, we should have that if $T \xrightarrow{\to\mathsf{r}} T'$ for some T', then $U \xrightarrow{\to\mathsf{r}} U'$ for some U' with $(T', U') \in \mathcal{R}$, and also that if $U \xrightarrow{\to\mathsf{r}} U'$ for some U', then $T \xrightarrow{\to\mathsf{r}} T'$ for some T' with $(T', U') \in \mathcal{R}$. That the transitions of T and U by $\to\mathsf{r}$ match is readily verifiable, and that $(T_2, U_2) \in \mathcal{R}$ is given by $T_2 \leq U_2$ and the definition of $\mathcal{R}$.

Finally, as $\to 1 \in \mathcal{X}$, we need to show that if $T \xrightarrow{\to 1} T'$ for some T', then $U \xrightarrow{\to 1} U'$ for some U' with $(T', U') \in \mathcal{R}$. As we have seen, T can have at most one such transition, $T \xrightarrow{\to 1} \mathsf{Skip}$,

and only in the case where $m = 1$. From $m \sqsubseteq n$ follows that $n = 1$. From this follows that $U \xrightarrow{\to 1} \mathsf{Skip}$. We arrive at pair $(\mathsf{Skip}, \mathsf{Skip})$, which is obviously in $\mathcal{R}$.

(**Case** S-RCD): Then $T = \{\ell\colon T_\ell\}_{\ell \in L}$, $U = \{k\colon U_k\}_{k \in K}$ and $K \subseteq L$. The only transitions that can be applied to T are $T \xrightarrow{\{\}} \mathsf{Skip}$ and $T \xrightarrow{\{\}_i} T_i$ for each $i \in L$ (L-RCDVRT and L-RCDVRTFIELD). Similarly, the only transitions that can be applied to U are $U \xrightarrow{\{\}} \mathsf{Skip}$ and $U \xrightarrow{\{\}_j} U_j$ for each $j \in K$. It is clear from the rules of type formation that $\vdash T_i$ and $\vdash U_j$ for each $i \in L, j \in K$ and, because S-RCD was used, we know $T_j \leq U_j$ for each $j \in K$.

Since $\{\} \in \mathcal{X}, \mathcal{Y}$, we should have that if $T \xrightarrow{\{\}} T'$ for some T', then $U \xrightarrow{\{\}} U'$ for some U' with $(T', U') \in \mathcal{R}$, and also that if $U \xrightarrow{\{\}} U'$ for some U', then $T \xrightarrow{\{\}} T'$ for some T' with $(T', U') \in \mathcal{R}$. That the transitions of T and U by $\{\}$ match is readily verifiable, and that $(\mathsf{Skip}, \mathsf{Skip}) \in \mathcal{R}$ is also evident.

Finally, since $\{\}_j \in \mathcal{Y}$ for each $j \in K$, we need to show that if $U \xrightarrow{\{\}_j} U'$ for some U', then $T \xrightarrow{\{\}_j} T'$ for some T' with $(T', U') \in \mathcal{R}$. As $K \subseteq L$, it is readily verifiable that T matches every transition of U by $\{\}_j$ for each $j \in K$. That $(T_j, U_j) \in \mathcal{R}$ follows by $T_j \leq U_j$ for each $j \in K$ and the definition of $\mathcal{R}$.

(**Case** S-VRT): Then $T = \langle \ell\colon T_\ell \rangle_{\ell \in L}$, $U = \langle k\colon U_k \rangle_{k \in K}$ and $L \subseteq K$. The only transitions that can be applied to T are $T \xrightarrow{\langle\rangle} \mathsf{Skip}$ and $T \xrightarrow{\langle\rangle i} T_i$ for each $i \in L$ (L-RCDVRT and L-RCDVRTFIELD). Similarly, the only transitions that can be applied to U are $U \xrightarrow{\langle\rangle} \mathsf{Skip}$ and $U \xrightarrow{\langle\rangle_j} U_j$ for each $j \in K$. It is clear from the rules of type formation that $\vdash T_i$ and $\vdash U_j$ for each $i \in L, j \in K$ and, because S-VRT was used, we know $T_i \leq U_i$ for each $i \in L$.

Since $\langle\rangle \in \mathcal{X}, \mathcal{Y}$, we should have that if $T \xrightarrow{\langle\rangle} T'$ for some T', then $U \xrightarrow{\langle\rangle} U'$ for some U' with $(T', U') \in \mathcal{R}$, and also that if $U \xrightarrow{\langle\rangle} U'$ for some U', then $T \xrightarrow{\langle\rangle} T'$ for some T' with $(T', U') \in \mathcal{R}$. That the transitions of T and U by $\langle\rangle$ match is readily verifiable, and that $(\mathsf{Skip}, \mathsf{Skip}) \in \mathcal{R}$ is also evident.

Finally, since $\langle\rangle_i \in \mathcal{X}$ for each $i \in L$, we need to show that if $T \xrightarrow{\langle\rangle_i} T'$ for some T', then $U \xrightarrow{\langle\rangle_i} U'$ for some U' with $(T', U') \in \mathcal{R}$. As $L \subseteq K$, it is readily verifiable that U matches every transition of T by $\langle\rangle_i$ for each $i \in L$. That $(T_i, U_i) \in \mathcal{R}$ follows by $T_i \leq U_i$ for each $i \in L$ and the definition of $\mathcal{R}$.

(**Case** S-END): Analogous to case S-UNIT.

(**Case** S-IN): Then $T = ?T'$ and $U = ?U'$. The only transitions that can be applied to T are $T \xrightarrow{?\mathsf{p}} T'$ and $T \xrightarrow{?\mathsf{c}} \mathsf{Skip}$ (L-MSG1, L-MSG2). Similarly, the only transitions that can be applied to U are $U \xrightarrow{?\mathsf{p}} U'$ and $U \xrightarrow{?\mathsf{c}} \mathsf{Skip}$. It is clear from the rules of type formation that $\vdash T'$ and $\vdash U'$. Furthermore, because S-IN was used, $T' \leq U'$.

Since $?\mathsf{p}, ?\mathsf{c} \in \mathcal{X}, \mathcal{Y}$, we should have that if $T \xrightarrow{?\mathsf{p}} T'$ for some T', then $U \xrightarrow{?\mathsf{p}} U'$ for some U' with $(T', U') \in \mathcal{R}$, and similarly for $?\mathsf{c}$. We must also have that if $U \xrightarrow{?\mathsf{p}} U'$ for some U', then $T \xrightarrow{?\mathsf{p}} T'$ for some T' with $(T', U') \in \mathcal{R}$, and similarly for $?\mathsf{c}$. That the transitions of T and U by $?\mathsf{p}$ and $?\mathsf{c}$ match is readily verifiable, and $(T', U'), (\mathsf{Skip}, \mathsf{Skip}) \in \mathcal{R}$ follows from $T' \leq U'$, S-SKIP and from the definition of $\mathcal{R}$.

(**Case** S-OUT): Then $T = !T'$ and $U = !U'$. The only transitions that can be applied to T

are $T \xrightarrow{!\mathsf{p}} T'$ and $T \xrightarrow{!\mathsf{c}} \mathsf{Skip}$ (L-MSG1, L-MSG2). Similarly, the only transitions that can be applied to U are $U \xrightarrow{!\mathsf{p}} U'$ and $U \xrightarrow{!\mathsf{c}} \mathsf{Skip}$. It is clear from the rules of type formation that $\vdash T'$ and $\vdash U'$. Furthermore, because S-OUT was used, $U' \leq T'$.

Since $!\mathsf{p} \in \mathcal{Z}, \mathcal{W}$, we should have that if $T \xrightarrow{!\mathsf{p}} T''$ for some T'', then $U \xrightarrow{!\mathsf{p}} U''$ for some U'' with $(U'', T'') \in \mathcal{R}$, and also that if $U \xrightarrow{!\mathsf{p}} U''$ for some U'', then $T \xrightarrow{!\mathsf{p}} T''$ for some T'' with $(U'', T'') \in \mathcal{R}$. That the transitions of T and U by $!\mathsf{p}$ match is readily verifiable, and that $(U', T') \in \mathcal{R}$ is given by $U' \leq T'$ and the definition of $\mathcal{R}$.

Finally, since $!\mathsf{c} \in \mathcal{X}, \mathcal{Y}$, we should have that if $T \xrightarrow{!\mathsf{c}} T'$ for some T', then $U \xrightarrow{!\mathsf{c}} U'$ for some U' with $(T', U') \in \mathcal{R}$, and also that if $U \xrightarrow{!\mathsf{c}} U'$ for some U', then $T \xrightarrow{!\mathsf{c}} T'$ for some T' with $(T', U') \in \mathcal{R}$. That the transitions of T and U by $!\mathsf{c}$ match is readily verifiable, and that $(\mathsf{Skip}, \mathsf{Skip}) \in \mathcal{R}$ follows from S-SKIP and the definition of $\mathcal{R}$.

(**Case** S-EXTCHOICE): Analogous to case S-VRT.

(**Case** S-INTCHOICE): Analogous to case S-RCD.

(**Case** S-SKIP): Then $T = \mathsf{Skip}$ and $U = \mathsf{Skip}$. Since no transitions apply to Skip, the conditions for $\mathcal{XYZW}$-simulation trivially hold.

(**Case** S-ENDSEQ1L): Then $T = \mathsf{End};S$ and $U = \mathsf{End}$. The only transition that can be applied to T is $T \xrightarrow{\mathsf{End}} \mathsf{Skip}$ (L-ENDSEQ). Similarly, the only transition that can be applied to U is $U \xrightarrow{\mathsf{End}} \mathsf{Skip}$ (L-END).

Since $\mathsf{End} \in \mathcal{X}, \mathcal{Y}$, we should have that if $T \xrightarrow{\mathsf{End}} T'$ for some T', then $U \xrightarrow{\mathsf{End}} U'$ for some U' with $(T', U') \in \mathcal{R}$, and also that if $U \xrightarrow{\mathsf{End}} U'$ for some U', then $T \xrightarrow{\mathsf{End}} T'$ for some T' with $(T', U') \in \mathcal{R}$. That the transitions of T and U by End match is readily verifiable, and $(\mathsf{Skip}, \mathsf{Skip}) \in \mathcal{R}$ follows from S-SKIP and the definition of $\mathcal{R}$.

(**Case** S-ENDSEQ1R, S-ENDSEQ2): Analogous to case S-ENDSEQ1L.

(**Case** S-INSEQ1L): Then $T = ?T';S$ and $U = ?U'$. The only transitions that can be applied to T are $T \xrightarrow{?\mathsf{p}} T'$ and $T \xrightarrow{?\mathsf{c}} S$ (L-MSGSEQ1, MSGSEQ2). Similarly, the only transitions that can be applied to U are $U \xrightarrow{?\mathsf{p}} U'$ and $U \xrightarrow{?\mathsf{c}} \mathsf{Skip}$. It is clear from the rules of type formation that $\vdash T'$,$\vdash U'$ and $\vdash S$. Furthermore, because S-INSEQ1L was used, $T' \leq U'$ and $S \leq \mathsf{Skip}$.

Since $?\mathsf{p}, ?\mathsf{c} \in \mathcal{X}, \mathcal{Y}$, we should have that if $T \xrightarrow{?\mathsf{p}} T'$ for some T', then $U \xrightarrow{?\mathsf{p}} U'$ for some U' with $(T', U') \in \mathcal{R}$, and similarly for $?\mathsf{c}$. For the same reason, we must also have that if $U \xrightarrow{?\mathsf{p}} U'$ for some U', then $T \xrightarrow{?\mathsf{p}} T'$ for some T' with $(T', U') \in \mathcal{R}$, and similarly for $?\mathsf{c}$. That the transitions of T and U by $?\mathsf{p}$ and $?\mathsf{c}$ match is readily verifiable, and $(T', U'), (S, \mathsf{Skip}) \in \mathcal{R}$ follows from $T' \leq U'$, from $S \leq \mathsf{Skip}$ and from the definition of $\mathcal{R}$.

(**Case** S-INSEQ1R, S-INSEQ2): Analogous to case S-INSEQ1L.

(**Case** S-OUTSEQ1L): Then $T = !T';S$ and $U = !U'$. The only transitions that can be applied to T are $T \xrightarrow{!\mathsf{p}} T'$ and $T \xrightarrow{!\mathsf{c}} S$ (L-MSGSEQ1, MSGSEQ2). Similarly, the only transitions that can be applied to U are $U \xrightarrow{!\mathsf{p}} U'$ and $U \xrightarrow{!\mathsf{c}} \mathsf{Skip}$. It is clear from the rules of type formation that $\vdash T'$, $\vdash U'$ and $\vdash S$. Furthermore, because S-OUTSEQ1L was used, $U' \leq T'$ and $S \leq \mathsf{Skip}$.

Since $!\mathsf{p} \in \mathcal{Z}, \mathcal{W}$, we should have that if $T \xrightarrow{!\mathsf{p}} T''$ for some T'', then $U \xrightarrow{!\mathsf{p}} U''$ for some U'' with $(U'', T'') \in \mathcal{R}$, and also that if $U \xrightarrow{!\mathsf{p}} U''$ for some U'', then $T \xrightarrow{!\mathsf{p}} T''$ for some T''

with $(U'', T'') \in \mathcal{R}$. That the transitions of T and U by $!\mathsf{p}$ match is readily verifiable, and that $(U', T') \in \mathcal{R}$ is given by $U' \leq T'$ and the definition of $\mathcal{R}$.

Finally, since $!\mathsf{c} \in \mathcal{X}, \mathcal{Y}$, we should have that if $T \xrightarrow{!\mathsf{c}} T'$ for some T', then $U \xrightarrow{!\mathsf{c}} U'$ for some U' with $(T', U') \in \mathcal{R}$, and also that if $U \xrightarrow{!\mathsf{c}} U'$ for some U', then $T \xrightarrow{!\mathsf{c}} T'$ for some T' with $(T', U') \in \mathcal{R}$. That the transitions of T and U by $!\mathsf{c}$ match is readily verifiable, and $(S, \mathsf{Skip}) \in \mathcal{R}$ follows from $S \leq \mathsf{Skip}$ and the definition of $\mathcal{R}$.

(**Case** S-OUTSEQ1R, S-OUTSEQ2): Analogous to case S-OUTSEQ1L.

Now consider that $T \neq \mathsf{unr}(T)$. From $T \leq U$ follows that $\mathsf{unr}(T) \leq U$, resulting from the application of right-preserving rules. If $U = \mathsf{unr}(U)$, then the above case analysis shows that the conditions for $\mathcal{XYZW}$-simulation between $\mathsf{unr}(T)$ and U hold. Since T and $\mathsf{unr}(T)$ have the same transitions, i.e., $T \xrightarrow{a} T'$ iff $\mathsf{unr}(T) \xrightarrow{a} T'$ for some T', the conditions for $\mathcal{XYZW}$-simulation between T and U also hold. Otherwise, if $U \neq \mathsf{unr}(U)$, it similarly follows that $\mathsf{unr}(T) \leq \mathsf{unr}(U)$, resulting from the application of left-preserving rules. The previous case analysis shows that the conditions for $\mathcal{XYZW}$-simulation between $\mathsf{unr}(T)$ and $\mathsf{unr}(U)$ hold. Since U and $\mathsf{unr}(U)$ have the same transitions, the same conditions also hold between T and U. The case with $T = \mathsf{unr}(U), U \neq \mathsf{unr}(U)$ is analogous.

Reverse implication Consider the relation $\mathcal{S} = \{(T, U) \mid \vdash T, \vdash U \text{ and } T \lesssim U\}$. We prove that relation $\mathcal{S}$ is backward closed for the rules of the syntactic subtyping relation. This will show that $\mathcal{S} \subseteq \leq$, and hence that $T \lesssim U$ implies $T \leq U$.

The proof has two parts. First, consider the cases where both T and U fit a type constructor, i.e., $T = \mathsf{unr}(T)$ and $U = \mathsf{unr}(U)$. We proceed by case analysis on the structure of T.

(**Case** $T = \mathsf{Unit}$): The only transition that applies to T is $T \xrightarrow{\mathsf{Unit}} \mathsf{Skip}$. Since $T \lesssim U$ and $U = \mathsf{unr}(U)$, then $U = \mathsf{Unit}$. Therefore we can apply S-UNIT.

(**Case** $T = T_1 \xrightarrow{m} T_2$): Two transitions apply to T regardless of m: $T \xrightarrow{\to\mathsf{d}} T_1$ and $T \xrightarrow{\to\mathsf{r}} T_2$. Since $T \lesssim U$ and $U = \mathsf{unr}(U)$, we know that $U = U_1 \xrightarrow{n} U_2$ and that, regardless of n, $U \xrightarrow{\to\mathsf{d}} U_1$ and $U \xrightarrow{\to\mathsf{r}} U_2$. Furthermore, we know that $U_1 \lesssim T_1$ (since $\to\mathsf{d} \in \mathcal{Z}, \mathcal{W}$) and that $T_2 \lesssim U_2$ (since $\to\mathsf{r} \in \mathcal{X}, \mathcal{Y}$).

Before we apply S-ARROW, we need to have $m \sqsubseteq n$. We know that $T \xrightarrow{\to 1} \mathsf{Skip}$ iff $m = 1$ and that $U \xrightarrow{\to 1} \mathsf{Skip}$ iff $n = 1$. Recall that $\to 1 \in \mathcal{X}$, which means that the only case where $n \not\sqsubseteq m$ ($m = 1$ and $n = *$) cannot occur, for it would contradict $T \lesssim U$ (since U could not match a transition of T by a label in $\mathcal{X}$).

We can therefore apply S-ARROW, arriving at $(U_1, T_1), (T_2, U_2) \in \mathcal{S}$.

(**Case** $T = \{\ell\colon T_\ell\}_{\ell \in L}$): The only transitions that can be applied to T are $T \xrightarrow{\{\}} \mathsf{Skip}$ and $T \xrightarrow{\{\}_i} T_i$ for each $i \in L$. Since $T \lesssim U$ and $U = \mathsf{unr}(U)$, we have $U = \{k\colon U_k\}_{k \in K}$ with transitions $U \xrightarrow{\{\}} \mathsf{Skip}$ and $U \xrightarrow{\{\}_j} U_j$ for $j \in K$. Since labels of the form $\{\}_\ell$ belong to $\mathcal{Y}$, we know that $K \subseteq L$, for T must be able to match all transitions of U by $\{\}_j$ for each $j \in K$. From this we obtain $T_j \lesssim U_j$ for each $j \in K$, arriving at $(T_j, U_j) \in \mathcal{S}$ for each $j \in K$.

(**Case** $T = \langle \ell\colon T_\ell \rangle_{\ell \in L}$): The only transitions that can be applied to T are $T \xrightarrow{\langle\rangle} \mathsf{Skip}$ and

$T \xrightarrow{\langle\rangle_i} T_i$ for each $i \in L$. Since $T \lesssim U$ and $U = \mathsf{unr}(U)$, we have $U = \langle k\colon U_k\rangle_{k\in K}$ with transitions $U \xrightarrow{\langle\rangle} \mathsf{Skip}$ and $U \xrightarrow{\langle\rangle_j} U_j$ for $j \in K$. Since labels of the form $\langle\rangle_\ell$ belong to $\mathcal{X}$, we know that $L \subseteq K$, for U must be able to match all transitions of T by $\langle\rangle_i$ for each $i \in L$. From this we obtain $T_i \lesssim U_i$ for each $i \in L$, arriving at $(T_i, U_i) \in \mathcal{S}$ for each $i \in L$.

(**Case** $T = x$): Cannot occur, since $\not\vdash T$.

(**Case** $T = \mathsf{End}$): Analogous to the case where $T = \mathsf{Unit}$.

(**Case** $T = ?T'$): The only transitions applicable to T are $T \xrightarrow{?\mathsf{p}} T'$ and $T \xrightarrow{?\mathsf{c}} \mathsf{Skip}$. Since $T \lesssim U$ and $U = \mathsf{unr}(U)$, then either $U = ?U'$ or $U = ?U';V$ with $V \lesssim \mathsf{Skip}$. In either case, $T' \lesssim U'$. In the first case, we can apply S-IN, arriving at $(T', U') \in \mathcal{S}$. In the second case, we can apply S-INSEQ1R, arriving at $(T', U'), (V, \mathsf{Skip}) \in \mathcal{S}$.

(**Case** $T = !T'$): The only transitions applicable to T are $T \xrightarrow{!\mathsf{p}} T'$ and $T \xrightarrow{!\mathsf{c}} \mathsf{Skip}$. Since $T \lesssim U$ and $U = \mathsf{unr}(U)$, then either $U = !U'$ or $U = !U';V$ with $V \lesssim \mathsf{Skip}$. In either case, $U' \lesssim T'$. In the first case, we can apply S-OUT, arriving at $(U', T') \in \mathcal{S}$. In the second case we can apply S-OUTSEQ1R, arriving at $(T', U'), (V, \mathsf{Skip}) \in \mathcal{S}$.

(**Case** $T = \oplus\{\ell\colon S_\ell\}_{\ell\in L}$): Analogous to case $T = \{\ell\colon T_\ell\}_{\ell\in L}$.

(**Case** $T = \&\{\ell\colon S_\ell\}_{\ell\in L}$): Analogous to case $T = \langle\ell\colon T_\ell\rangle_{\ell\in L}$.

(**Case** $T = \mathsf{Skip}$): No transitions apply to T. Since $T \lesssim U$ and $U = \mathsf{unr}(U)$, then $U = \mathsf{Skip}$. Therefore we can apply S-SKIP.

(**Case** $T = ?T_1;T_2$): The only transitions applicable to T are $T \xrightarrow{?\mathsf{p}} T_1$ and $T \xrightarrow{?\mathsf{c}} T_2$. Since $T \lesssim U$ and $U = \mathsf{unr}(U)$, then either $U = ?U_1$ or $U = ?U_1;U_2$. In the first case, $T_1 \lesssim U_1$ and $T_2 \lesssim \mathsf{Skip}$; we can apply S-INSEQ1L, arriving at $(T_1, U_1) \in \mathcal{S}$. In the second case, $T_1 \lesssim U_1$ and $T_2 \lesssim U_2$; we can apply S-INSEQ2, arriving at $(T_1, U_1), (T_2, U_2) \in \mathcal{S}$.

(**Case** $T = !T_1;T_2$): The only transitions applicable to T are $T \xrightarrow{!\mathsf{p}} T_1$ and $T \xrightarrow{!\mathsf{c}} T_2$. Since $T \lesssim U$ and $U = \mathsf{unr}(U)$, then either $U = !U_1$ or $U = !?U_1U_2$. In the first case, $U_1 \lesssim T_1$ and $T_2 \lesssim \mathsf{Skip}$; we can apply S-OUTSEQ1L, arriving at $(U_1, T_1) \in \mathcal{S}$. In the second case, $U_1 \lesssim T_1$ and $T_2 \lesssim U_2$; we can apply S-OUTSEQ2, arriving at $(U_1, T_1), (T_2, U_2) \in \mathcal{S}$.

Now consider that $T \neq \mathsf{unr}(T)$. From $T \lesssim U$ it follows that $T' \lesssim U$ where $T' = \mathsf{unr1}(T)$, due to the fact that T and T' have the same transitions (Lemma A.0.3). Then we can apply an appropriate right-preserving rule to (T, U), arriving at $(T', U) \in \mathcal{S}$. The case with $T = \mathsf{unr}(T), U \neq \mathsf{unr}(U)$ is analogous. □

Appendix E

Proof of Theorem 6.1.1

We begin by presenting some preliminary results that will be needed for the main proof.

Lemma E.0.1. *Let $\vec{X}$ and $\vec{Y}$ be words from a simple GNF grammar with productions $\mathcal{P}$. If $\vec{X} \sim_{\mathcal{P}} \vec{Y}$, then $\vec{X} \lesssim_{\mathcal{P}} \vec{Y}$ and $\vec{Y} \lesssim_{\mathcal{P}} \vec{X}$.*

Proof. Consider the relation $\mathcal{R} = \{(\vec{X}, \vec{Y}) \mid \vec{X} \sim_{\mathcal{P}} \vec{Y}\}$. We want to show that $\mathcal{R} \subseteq \lesssim_{\mathcal{P}}$ and $\mathcal{R} \subseteq \lesssim_{\mathcal{P}}^{-1}$.

To show that $\mathcal{R} \subseteq \lesssim_{\mathcal{P}}$, we show that for all $(\vec{X}, \vec{Y}) \in \mathcal{R}$: for every label $a \in \mathcal{X} \cup \mathcal{Z}$ and word $\vec{X}$, if $\vec{X} \xrightarrow{a}_{\mathcal{P}} \vec{X}'$, then there exists a word $\vec{Y}'$ s.t. $\vec{Y} \xrightarrow{a}_{\mathcal{P}} \vec{Y}'$ and $(\vec{X}', \vec{Y}') \in \mathcal{R}$ if $a \in \mathcal{X}$, or otherwise $(\vec{Y}', \vec{X}') \in \mathcal{R}$ if $a \in \mathcal{Z}$; and, for every label $a \in \mathcal{Y} \cup \mathcal{W}$ and word $\vec{Y}$, if $\vec{Y} \xrightarrow{a}_{\mathcal{P}} \vec{Y}'$, then there exists a word $\vec{X}'$ s.t. $\vec{X} \xrightarrow{a}_{\mathcal{P}} \vec{X}'$ and $(\vec{X}', \vec{Y}') \in \mathcal{R}$ if $a \in \mathcal{Y}$, or otherwise $(\vec{Y}', \vec{X}') \in \mathcal{R}$ if $a \in \mathcal{W}$.

First, let there be $a \in \mathcal{X}$ and $\vec{X}'$ s.t. $\vec{X} \xrightarrow{a} \vec{X}'$. Since $\vec{X} \sim_{\mathcal{P}} \vec{Y}$, we know there is a $\vec{Y}'$ s.t. $\vec{Y} \xrightarrow{a}_{\mathcal{P}} \vec{Y}'$ and $\vec{X}' \sim_{\mathcal{P}} \vec{Y}'$ and hence that $(\vec{X}', \vec{Y}') \in \mathcal{R}$.

Next, let there be $a \in \mathcal{Y}$ and $\vec{Y}'$ s.t. $\vec{Y} \xrightarrow{a} \vec{Y}'$. Since $\vec{X} \sim_{\mathcal{P}} \vec{Y}$, we know there is a $\vec{X}'$ s.t. $\vec{X} \xrightarrow{a} \vec{X}'$ and $\vec{X}' \sim_{\mathcal{P}} \vec{Y}'$. Hence $(\vec{X}', \vec{Y}') \in \mathcal{R}$.

Now let there be $a \in \mathcal{Z}$ and $\vec{X}'$ s.t. $\vec{X} \xrightarrow{a} \vec{X}'$. Since $\vec{X} \sim_{\mathcal{P}} \vec{Y}$, we know there is a $\vec{Y}'$ s.t. $\vec{Y} \xrightarrow{a} \vec{Y}'$ and $\vec{X}' \sim_{\mathcal{P}} \vec{Y}'$. By the symmetry of $\sim_{\mathcal{P}}$ we get $\vec{Y}' \sim_{\mathcal{P}} \vec{X}'$, and therefore $(\vec{Y}', \vec{X}') \in \mathcal{R}$.

Finally, let there be $a \in \mathcal{W}$ and $\vec{Y}'$ s.t. $\vec{Y} \xrightarrow{a} \vec{Y}'$. Since $\vec{X} \sim_{\mathcal{P}} \vec{Y}$, we know there is a $\vec{X}'$ s.t. $\vec{X} \xrightarrow{a} \vec{X}'$ and $\vec{X}' \sim_{\mathcal{P}} \vec{Y}'$. By the symmetry of $\sim_{\mathcal{P}}$ we get $\vec{Y}' \sim_{\mathcal{P}} \vec{X}'$, and therefore $(\vec{Y}', \vec{X}') \in \mathcal{R}$.

We proceed similarly to show that $\mathcal{R} \subseteq \lesssim_{\mathcal{P}}^{-1}$. For all $(\vec{X}, \vec{Y}) \in \mathcal{R}$: for every label $a \in \mathcal{X} \cup \mathcal{Z}$ and type $\vec{Y}'$, if $\vec{Y} \xrightarrow{a}_{\mathcal{P}} \vec{Y}'$, then there exists a word $\vec{X}'$ s.t. $\vec{X} \xrightarrow{a}_{\mathcal{P}} \vec{X}'$ and $(\vec{X}', \vec{Y}') \in \mathcal{R}$ if $a \in \mathcal{X}$, or otherwise $(\vec{Y}', \vec{X}') \in \mathcal{R}$ if $a \in \mathcal{Z}$; and, for every label $a \in \mathcal{Y} \cup \mathcal{W}$ and word $\vec{X}'$, if $\vec{X} \xrightarrow{a}_{\mathcal{P}} \vec{X}'$, then there exists a word $\vec{Y}'$ s.t. $\vec{Y} \xrightarrow{a}_{\mathcal{P}} \vec{Y}'$ and $(\vec{X}', \vec{Y}') \in \mathcal{R}$ if $a \in \mathcal{Y}$, or otherwise $(\vec{Y}', \vec{X}') \in \mathcal{R}$ if $a \in \mathcal{W}$.

First, let there be $a \in \mathcal{X}$ and $\vec{Y}'$ s.t. $\vec{Y} \xrightarrow{a} \vec{Y}'$. Since $\vec{X} \sim_{\mathcal{P}} \vec{Y}$, we know there is a $\vec{X}'$ s.t. $\vec{X} \xrightarrow{a}_{\mathcal{P}} \vec{X}'$ and $\vec{X}' \sim_{\mathcal{P}} \vec{Y}'$. Hence $(\vec{X}', \vec{Y}') \in \mathcal{R}$.

Next, let there be $a \in \mathcal{Y}$ and $\vec{X}'$ s.t. $\vec{X} \xrightarrow{a} \vec{X}'$. Since $\vec{X} \sim_{\mathcal{P}} U$, we know there is a $\vec{Y}'$ s.t. $\vec{Y} \xrightarrow{a}_{\mathcal{P}} \vec{Y}'$ and $\vec{X}' \sim_{\mathcal{P}} \vec{Y}'$. Hence $(\vec{X}', \vec{Y}') \in \mathcal{R}$.

Now let there be $a \in \mathcal{Z}$ and $\vec{Y}'$ s.t. $\vec{Y} \xrightarrow{a} \vec{Y}'$. Since $\vec{X} \sim \vec{Y}$, we know there is a $\vec{X}'$ s.t. $\vec{X} \xrightarrow{a} \vec{X}'$ and $\vec{X}' \sim_{\mathcal{P}} \vec{Y}'$. By the symmetry of $\sim_{\mathcal{P}}$ we get $\vec{Y}' \sim_{\mathcal{P}} \vec{X}'$, and therefore $(\vec{Y}', \vec{X}') \in \mathcal{R}$.

Finally, let there be $a \in \mathcal{W}$ and $\vec{X}'$ s.t. $\vec{X} \xrightarrow{a} \vec{X}'$. Since $\vec{X} \sim \vec{Y}$ we know there is $\vec{Y}'$ s.t. $\vec{Y} \xrightarrow{a} \vec{Y}'$ and $\vec{X}' \sim \vec{Y}'$. By the symmetry of $\sim_{\mathcal{P}}$ we get $\vec{Y}' \sim_{\mathcal{P}} \vec{X}'$, and therefore $(\vec{Y}', \vec{X}') \in \mathcal{R}$. □

Lemma E.0.2. *Let T be a type and let $\mathcal{N}$ be the set of non-terminal symbols in the simple GNF grammar $\mathsf{grm}(T, \mathcal{P}_0)$ for any $\mathcal{P}_0$. For every $\vec{X}, \vec{Y} \in \mathcal{N}^*$ we have that $\vec{X} \sim_{\mathcal{P}} \vec{X} \bot \vec{Y}$.*

Proof. Immediate from the fact that $\bot$ has no productions. □

Lemma E.0.3 (The behaviours of types and their words coincide)**.** *Let $\vdash T$ and $(\vec{X}_T, \mathcal{P}) = \mathsf{grm}(T, \mathcal{P}_0)$ for some $\mathcal{P}_0$. Then,*

- *If $T \xrightarrow{a} U$ for some U, then there exists $\vec{X}'$ such that $\vec{X}_T \xrightarrow{a}_{\mathcal{P}} \vec{X}'$ and $\vec{X}_U \sim_{\mathcal{P}} \vec{X}'$, where $(\vec{X}_U, \mathcal{P}') = \mathsf{grm}(U, \mathcal{P})$;*
- *If $\vec{X}_T \xrightarrow{a}_{\mathcal{P}} \vec{X}'$ for some $\vec{X}'$, then there exists U such that $T \xrightarrow{a} U$ and $\vec{X}' \sim_{\mathcal{P}} \vec{X}_U$, where $(\vec{X}_U, \mathcal{P}') = \mathsf{grm}(U, \mathcal{P})$.*

Proof. Let us define the relation

$$\mathcal{R} = \{(T, \vec{Y}) \mid \vec{X}_T \sim_{\mathcal{P}} \vec{Y}, \text{where } (\vec{X}_T, \mathcal{P}) = \mathsf{grm}(T, \emptyset)\}.$$

We now prove that $\mathcal{R}$ is backward closed for the transition relations, showing the desired property. We begin by considering the cases in which T fits a type constructor, i.e., $T = \mathsf{unr}(T)$. We have the following case analysis for T.

(**Case** $T = \mathsf{Unit}$): by rule L-UNIT, the LTS at T has the unique transition $T \xrightarrow{\mathsf{Unit}} \mathsf{Skip}$. Similarly, the LTS at $\vec{X}_T$ has the unique transition $\vec{X}_T \xrightarrow{\mathsf{Unit}}_{\mathcal{P}} \varepsilon$. Since $\vec{X}_T \sim_{\mathcal{P}} \vec{Y}$, the LTS at $\vec{Y}$ has the unique transition $\vec{Y} \xrightarrow{\mathsf{Unit}}_{\mathcal{P}} \vec{Y}'$ for some $\vec{Y}$ such that $\vec{Y} \sim_{\mathcal{P}} \varepsilon$. Since $\mathsf{word}(\mathsf{Skip}) = \varepsilon$, we have $\mathsf{word}(\mathsf{Skip}) \sim_{\mathcal{P}} \vec{Y}'$ and therefore $(\mathsf{Skip}, \vec{Y}') \in \mathcal{R}$.

(**Case** $T = U \xrightarrow{*} V$): By rules L-ARROWDOM and L-ARROWRNG, we know that T has exactly two transitions: $T \xrightarrow{\rightarrow \mathsf{d}} U$ and $T \xrightarrow{\rightarrow \mathsf{r}} V$ (rule L-LINARROW does not apply). Similarly, the LTS at $\vec{X}_T$ has exactly two transitions: $\vec{X}_T \xrightarrow{\rightarrow \mathsf{d}}_{\mathcal{P}} \mathsf{word}(U)$ and $\vec{X}_T \xrightarrow{\rightarrow \mathsf{r}}_{\mathcal{P}} \mathsf{word}(V)$. Since $\vec{X}_T \sim_{\mathcal{P}} \vec{Y}$, the LTS at $\vec{Y}$ has exactly two transitions, $\vec{Y} \xrightarrow{\rightarrow \mathsf{d}}_{\mathcal{P}} \vec{Y}_1$ and $\vec{Y} \xrightarrow{\rightarrow \mathsf{d}}_{\mathcal{P}} \vec{Y}_2$ for some $\vec{Y}_1, \vec{Y}_2$ s.t. $\vec{Y}_1 \sim_{\mathcal{P}} \mathsf{word}(U)$ and $\vec{Y}_2 \sim_{\mathcal{P}} \mathsf{word}(V)$. Hence $(U, \vec{Y}_1), (V, \vec{Y}_2) \in \mathcal{R}$.

(**Case** $T = U \xrightarrow{1} V$): Similar to the previous case, but the LTS at T has a single additional transition in $T \xrightarrow{\rightarrow 1} \mathsf{Skip}$, as does the LTS at $\vec{X}_T$ in $\vec{X}_T \xrightarrow{\rightarrow 1}_{\mathcal{P}} \varepsilon$. Since $\vec{X}_T \sim_{\mathcal{P}} \vec{Y}$, we know $\vec{Y}$ also has an additional transition, $\vec{Y} \xrightarrow{\rightarrow 1}_{\mathcal{P}} \vec{Y}'$ for some $\vec{Y}'$ s.t. $\vec{Y}' \sim_{\mathcal{P}} \varepsilon$. Since $\mathsf{word}(\mathsf{Skip}) = \varepsilon$, we have $(\mathsf{Skip}, \vec{Y}') \in \mathcal{R}$.

(**Case** $T = (\!|\ell\colon T|\!)_{\ell \in L}$): By rules L-RCDVRT and L-RCDVRTFIELD, the LTS at T has exactly the transitions $T \xrightarrow{(\!|\checkmark|\!)} \mathsf{Skip}$ and $T \xrightarrow{(\!|k|\!)} T_k$ for each $k \in L$. Similarly, the LTS at $\vec{X}_T$ has exactly the transitions $\vec{X}_T \xrightarrow{(\!|\checkmark|\!)}_{\mathcal{P}} \bot$ and $\vec{X}_T \xrightarrow{(\!|k|\!)}_{\mathcal{P}} \mathsf{word}(T_k)$ for each $k \in L$. Since $\vec{X}_T \sim_{\mathcal{P}} \vec{Y}$, the LTS

at $\vec{Y}$ has exactly the transitions $\vec{Y} \xrightarrow{\oplus_k}_{\mathcal{P}} \vec{Y}_k$ for some $\vec{Y}_k$ s.t. $\mathsf{word}(T_k) \sim_{\mathcal{P}} \vec{Y}$ for each $k \in L$ and additionally $\vec{Y} \xrightarrow{\oplus_\checkmark}_{\mathcal{P}} \vec{Y}_\checkmark$ for some $\vec{Y}_\checkmark$ s.t. $\vec{Y}_\checkmark \sim_{\mathcal{P}} \bot$. It follows that $(T_k, \vec{Y}_k) \in \mathcal{R}$ for each $k \in L$. Observe that, since neither ε or $\bot$ have any transitions, we have $\varepsilon \sim_{\mathcal{P}} \bot$. From this and $\mathsf{word}(\mathsf{Skip}) = \varepsilon$ it follows that $(\mathsf{Skip}, \vec{Y}_\checkmark) \in \mathcal{R}$.

(**Case** $T = x$): Cannot occur, since $\not\vdash x$.

(**Case** $T = \mathsf{End}$): by rule L-END, the LTS at T has the unique transition $T \xrightarrow{\mathsf{End}} \mathsf{Skip}$. Similarly, the LTS at $\vec{X}_T$ has the unique transition $\vec{X}_T \xrightarrow{\mathsf{End}}_{\mathcal{P}} \bot$. Since $\vec{X}_T \sim_{\mathcal{P}} \vec{Y}$, the LTS at $\vec{Y}$ has the unique transition $\vec{Y} \xrightarrow{\mathsf{End}}_{\mathcal{P}} \vec{Y}'$ for some $\vec{Y}$ such that $\vec{Y} \sim_{\mathcal{P}} \bot$. Since $\mathsf{word}(\mathsf{Skip}) = \varepsilon$ and $\varepsilon \sim_{\mathcal{P}} \bot$, we have $\mathsf{word}(\mathsf{Skip}) \sim_{\mathcal{P}} \vec{Y}'$ and therefore $(\mathsf{Skip}, \vec{Y}') \in \mathcal{R}$.

(**Case** $T = \sharp U$): By rule L-MSG, the LTS at T has exactly two transitions $T \xrightarrow{\sharp\mathsf{p}} U$ and $T \xrightarrow{\sharp\mathsf{c}} \mathsf{Skip}$. Similarly, the LTS at $\vec{X}_T$ has exactly two transitions $\vec{X}_T \xrightarrow{\sharp\mathsf{p}}_{\mathcal{P}} \mathsf{word}(U)\bot$ and $\vec{X}_T \xrightarrow{\sharp\mathsf{c}}_{\mathcal{P}} \varepsilon$. Since $\vec{X}_T \sim_{\mathcal{P}} \vec{Y}$, the LTS at $\vec{Y}$ has exactly two transitions $\vec{Y} \xrightarrow{\sharp\mathsf{p}}_{\mathcal{P}} \vec{Y}_1$ and $\vec{Y} \xrightarrow{\sharp\mathsf{c}}_{\mathcal{P}} \vec{Y}_2$ for some $\vec{Y}_1, \vec{Y}_2$ s.t. $\mathsf{word}(U)\bot \sim_{\mathcal{P}} \vec{Y}_1$ and $\varepsilon \sim_{\mathcal{P}} \vec{Y}_2$. By Lemma E.0.2, we have $\mathsf{word}(U) \sim_{\mathcal{P}} \mathsf{word}(U)\bot \sim_{\mathcal{P}} \vec{Y}_1$, and by the definition of grm, $\mathsf{word}(\mathsf{Skip}) \sim_{\mathcal{P}} \varepsilon \sim_{\mathcal{P}} \vec{Y}_2$. Hence $(U, \vec{Y}_1), (\mathsf{Skip}, \vec{Y}_2) \in \mathcal{R}$.

(**Case** $T = \odot\{\ell\colon T\}_{\ell \in L}$): By rules L-RCDVRT and L-RCDVRTFIELD, the LTS at T has exactly the transitions $T \xrightarrow{\odot_\checkmark} \mathsf{Skip}$ and $T \xrightarrow{\odot_k} T_k$ for each $k \in L$. Similarly, the LTS at $\vec{X}_T$ has transitions $\vec{X}_T \xrightarrow{\odot_\checkmark}_{\mathcal{P}} \bot$ and $\vec{X}_T \xrightarrow{\odot_k}_{\mathcal{P}} \mathsf{word}(T_k)$ for each $k \in L$. Since $\vec{X}_T \sim_{\mathcal{P}} \vec{Y}$, the LTS at $\vec{Y}$ has exactly the transitions $\vec{Y} \xrightarrow{\odot_\checkmark}_{\mathcal{P}} \vec{Y}_\checkmark$ for some $\vec{Y}_\checkmark$ s.t. $\vec{Y}_\checkmark \sim_{\mathcal{P}} \bot$ and $\vec{Y} \xrightarrow{\odot_k}_{\mathcal{P}} \vec{Y}_k$ for some $\vec{Y}_k$ s.t. $\mathsf{word}(T_k) \sim_{\mathcal{P}} \vec{Y}_k$ for each $k \in L$. Since $\bot$ has no productions, we have $\bot \sim_{\mathcal{P}} \varepsilon$ and by transitivity $\vec{Y} \sim_{\mathcal{P}} \varepsilon$. Hence $(\mathsf{Skip}, \vec{Y}_\checkmark) \in \mathcal{R}$ and $(T_k, \vec{Y}_k) \in \mathcal{R}$ for each $k \in L$.

(**Case** $T = \sharp U;S$): By rules L-MSGSEQ1 and L-MSGSEQ2, the LTS at T has exactly two transitions, $T \xrightarrow{\sharp\mathsf{p}} U$ and $T \xrightarrow{\sharp\mathsf{c}} S$. Given that $\mathsf{word}(T) = \mathsf{word}(\sharp U) \cdot \mathsf{word}(S)$ and that $\mathsf{word}(\sharp U)$ yields exactly two productions $\mathsf{word}(\sharp U) \to \sharp\mathsf{p}\mathsf{word}(U)\bot$ and $\mathsf{word}(\sharp U) \to \sharp\mathsf{c}\mathsf{word}(S)$, we have for the LTS at $\vec{X}_T$ exactly two transitions $X_T \xrightarrow{\sharp\mathsf{p}}_{\mathcal{P}} \mathsf{word}(U)\bot\mathsf{word}(S)$ and $X_T \xrightarrow{\sharp\mathsf{c}}_{\mathcal{P}} \mathsf{word}(S)$. Since $X_T \sim_{\mathcal{P}} \vec{Y}$, the LTS at $\vec{Y}$ has exactly two transitions $\vec{Y} \xrightarrow{\sharp\mathsf{p}}_{\mathcal{P}} \vec{Y}_1$ and $\vec{Y} \xrightarrow{\sharp\mathsf{c}}_{\mathcal{P}} \vec{Y}_2$ for some $\vec{Y}_1, \vec{Y}_2$ s.t. $\mathsf{word}(U)\bot\mathsf{word}(S) \sim_{\mathcal{P}} \vec{Y}_1$ and $\mathsf{word}(S) \sim_{\mathcal{P}} \vec{Y}_2$. By Lemma E.0.2, we have $\mathsf{word}(U) \sim_{\mathcal{P}} \mathsf{word}(U)\bot\mathsf{word}(S) \sim_{\mathcal{P}} \vec{Y}_1$. Hence $(U, \vec{Y}_1), (S, \vec{Y}_2) \in \mathcal{R}$.

(**Case** $T = x;S$): Cannot occur, since $\not\vdash x;S$.

Now we consider the cases in which $T \neq \mathsf{unr}(T)$. It is straightforward to check that the LTS at T has a transition $T \xrightarrow{a} U$ iff the LTS at $\mathsf{unr}(T)$ has a corresponding transition $\mathsf{unr}(T) \xrightarrow{a} U$ (with the same U). This is a consequence of the fact that the LTS rules for T essentially follow its unfolding, which eventually terminates due to contractivity. On the side of grammars, we have $\mathsf{word}(\mathsf{unr}(T)) \sim_{\mathcal{P}} \mathsf{word}(T) \sim_{\mathcal{P}} \vec{Y}$. Now suppose that $T \xrightarrow{a} U$; then $\mathsf{unr}(T) \xrightarrow{a} U$; since $\mathsf{word}(\mathsf{unr}(T)) \sim_{\mathcal{P}} \vec{Y}$, our previous case analysis yields $\vec{Y} \xrightarrow{a} \vec{Y}'$ for some $\vec{Y}'$ s.t. $\mathsf{word}(U) \sim_{\mathcal{P}} \vec{Y}'$; concluding that $(U, \vec{Y}') \in \mathcal{R}$. Conversely, suppose that $\vec{Y} \xrightarrow{a}_{\mathcal{P}} \vec{Y}'$ for some $\vec{Y}'$. Since $\mathsf{word}(\mathsf{unr}(T)) \sim_{\mathcal{P}} \vec{Y}$, our previous case analysis yields $\mathsf{unr}(T) \xrightarrow{a}_{\mathcal{P}} U$ for some U s.t. $\mathsf{word}(U) \sim_{\mathcal{P}} \vec{Y}'$. We conclude that $T \xrightarrow{a} U$ and therefore that $(U, \vec{Y}') \in \mathcal{R}$. □

We are now able to prove Theorem 6.1.1

Theorem 6.1.1 (Soundness for grammars)**.** *Let* $\vdash T$, $\vdash U$, $(\vec{X}, \mathcal{P}') = \mathsf{grm}(T, \emptyset)$ *and* $(\vec{Y}, \mathcal{P}) = \mathsf{grm}(U, \mathcal{P}')$. *If* $\vec{X} \lesssim_{\mathcal{P}} \vec{Y}$, *then* $T \lesssim U$.

Proof. Consider the following relation on pairs of types

$$\mathcal{R} = \{(V, W) \mid \vec{X}_V \lesssim_{\mathcal{P}} \vec{X}_W\},$$

where $\vdash V$, $\vdash W$, $(\vec{X}_V, \mathcal{P}') = \mathsf{grm}(V, \emptyset)$ and $(\vec{X}_W, \mathcal{P}) = \mathsf{grm}(W, \mathcal{P}')$.

To prove the desired property, we show that $\mathcal{R} \subseteq \lesssim$, in other words, that $\mathcal{R}$ is an $\mathcal{X}\mathcal{Y}\mathcal{Z}\mathcal{W}$-simulation with $\mathcal{X}, \mathcal{Y}, \mathcal{Z}, \mathcal{W}$ defined as the sets generated, respectively, by the grammars for $a_{\mathcal{X}}, a_{\mathcal{Y}}, a_{\mathcal{Z}}, a_{\mathcal{W}}$ in Definition 5.2.4. Take $(V, W) \in \mathcal{R}$ with $(\vec{X}_V, \mathcal{P}') = \mathsf{grm}(V, \emptyset)$ and $(\vec{X}_W, \mathcal{P}) = \mathsf{grm}(W, \mathcal{P}')$.

First, let there be $a \in \mathcal{X}$ and V' s.t. $V \xrightarrow{a} V'$. We want to show that there is a W' s.t. $W \xrightarrow{a} W'$ and $(V', W') \in \mathcal{R}$. By Lemma E.0.3, there exists $\vec{Y}$ such that $\vec{X}_V \xrightarrow{a}_{\mathcal{P}} \vec{Y}$ and $\mathsf{word}(V') \sim_{\mathcal{P}} \vec{Y}$. Since $\vec{X}_V \lesssim_{\mathcal{P}} \vec{X}_W$, we know there exists a matching word $\vec{Z}$ such that $\vec{X}_W \xrightarrow{a}_{\mathcal{P}} \vec{Z}$ and that $\vec{Y} \lesssim_{\mathcal{P}} \vec{Z}$. Again by Lemma E.0.3, there exists W' such that $W \xrightarrow{a} W'$ and $\vec{Z} \sim_{\mathcal{P}} \mathsf{word}(W')$. Let $(\vec{X}_{V'}, \mathcal{P}') = \mathsf{grm}(V', \emptyset)$ and $(\vec{X}_{W'}, \mathcal{P}) = \mathsf{grm}(W', \mathcal{P}')$, and recall that $\vec{X}_{V'} = \mathsf{word}(V')$ and $\vec{X}_{W'} = \mathsf{word}(W')$. Having $\vec{X}_{V'} \sim_{\mathcal{P}} \vec{Y}$ and $\vec{X}_{W'} \sim_{\mathcal{P}} \vec{Z}$, we know by Lemma E.0.1 that $\vec{X}_{V'} \lesssim_{\mathcal{P}} \vec{Y}$ and $\vec{Z} \lesssim_{\mathcal{P}} \vec{X}_{W'}$. Recalling that $\vec{Y} \lesssim_{\mathcal{P}} \vec{Z}$, we establish by transitivity that $\vec{X}_{V'} \lesssim_{\mathcal{P}} \vec{X}_{W'}$ and therefore that $(V', W') \in \mathcal{R}$.

Next, let there be $a \in \mathcal{Y}$ and W' s.t. $W \xrightarrow{a} W'$. We want to show that there is a V' s.t. $V \xrightarrow{a} V'$ and $(V', W') \in \mathcal{R}$. Lemma E.0.3 gives us some $\vec{Z}$ with $\vec{X}_W \xrightarrow{a}_{\mathcal{P}} \vec{Z}$ and $\mathsf{word}(W') \sim_{\mathcal{P}} \vec{Z}$. Since $a \in \mathcal{Y}$, we know from $\vec{X}_V \lesssim_{\mathcal{P}} \vec{X}_W$ that there is a matching word $\vec{Y}$ such that $\vec{X}_V \xrightarrow{a}_{\mathcal{P}} \vec{Y}$ and that $\vec{Y} \lesssim_{\mathcal{P}} \vec{Z}$. Once again by Lemma E.0.3 we know there exists V' such that $V \xrightarrow{a} V'$ and $\vec{Y} \sim_{\mathcal{P}} \mathsf{word}(V')$. Let $(\vec{X}_{V'}, \mathcal{P}') = \mathsf{grm}(V', \emptyset)$ and $(\vec{X}_{W'}, \mathcal{P}) = \mathsf{grm}(W', \mathcal{P}')$. Having $\vec{Y} \sim_{\mathcal{P}} \vec{X}_{V'}$ and $\vec{X}_{W'} \sim_{\mathcal{P}} \vec{Z}$, we know by Lemma E.0.1 that $\vec{X}_{V'} \lesssim_{\mathcal{P}} \vec{Y}$ and $\vec{Z} \lesssim_{\mathcal{P}} \vec{X}_{W'}$. Recalling that $\vec{Y} \lesssim_{\mathcal{P}} \vec{Z}$, we can establish by transitivity that $\vec{X}_{V'} \lesssim_{\mathcal{P}} \vec{X}_{W'}$ and therefore that $(V', W') \in \mathcal{R}$.

Now let there be $a \in \mathcal{Z}$ and V' s.t. $V \xrightarrow{a} V'$. We want to show that there is a W' s.t. $W \xrightarrow{a} W'$ and $(W', V') \in \mathcal{R}$. By Lemma E.0.3, there exists $\vec{Y}$ s.t. $\vec{X}_V \xrightarrow{a} \vec{Y}$ and $\mathsf{word}(V') \sim_{\mathcal{P}} \vec{Y}$. Since $\vec{X}_V \lesssim_{\mathcal{P}} \vec{X}_W$, we know there exists a matching word $\vec{Z}$ s.t. $\vec{X}_W \xrightarrow{a} \vec{Z}$ and that $\vec{Z} \lesssim_{\mathcal{P}} \vec{Y}$. Again by Lemma E.0.3, there exists W' s.t. $W \xrightarrow{a} W'$ and $\vec{Z} \sim_{\mathcal{P}} \mathsf{word}(W')$. Let $(\vec{X}_{V'}, \mathcal{P}') = \mathsf{grm}(V', \emptyset)$ and $(\vec{X}_{W'}, \mathcal{P}) = \mathsf{grm}(W', \mathcal{P}')$, and recall that $\vec{X}_{V'} = \mathsf{word}(V')$ and $\vec{X}_{W'} = \mathsf{word}(W')$. Having $\vec{X}_{W'} \sim_{\mathcal{P}} \vec{Z}$ and $\vec{X}_{V'} \sim_{\mathcal{P}} \vec{Y}$, we know by Lemma E.0.1 that $\vec{X}_{W'} \lesssim_{\mathcal{P}} \vec{Z}$ and $\vec{Y} \lesssim_{\mathcal{P}} \vec{X}_{V'}$. Recalling that $\vec{Z} \lesssim_{\mathcal{P}} \vec{Y}$, we establish by transitivity that $\vec{X}_{W'} \lesssim_{\mathcal{P}} \vec{X}_{V'}$ and therefore that $(W', V') \in \mathcal{R}$.

Finally, let there be $a \in \mathcal{W}$ and W' s.t. $W \xrightarrow{a} W'$. We want to show that there is a V' s.t. $V \xrightarrow{a} V'$ and $(V', W') \in \mathcal{R}$. Lemma E.0.3 gives us some $\vec{Z}$ with $\vec{X}_W \xrightarrow{a}_{\mathcal{P}} \vec{Z}$ and $\mathsf{word}(W') \sim_{\mathcal{P}} \vec{Z}$. Since $a \in \mathcal{Y}$, we know from $\vec{X}_V \lesssim_{\mathcal{P}} \vec{X}_W$ that there is a matching word $\vec{Y}$ such that $\vec{X}_V \xrightarrow{a}_{\mathcal{P}} \vec{Y}$ and that $\vec{Z} \lesssim_{\mathcal{P}} \vec{Y}$. Once again by Lemma E.0.3 we know there exists V' such that $V \xrightarrow{a} V'$ and $\vec{Y} \sim_{\mathcal{P}} \mathsf{word}(V')$. Let $(\vec{X}_{V'}, \mathcal{P}') = \mathsf{grm}(V', \emptyset)$ and $(\vec{X}_{W'}, \mathcal{P}) = \mathsf{grm}(W', \mathcal{P}')$.

Having $\vec{X}_{W'} \sim_{\mathcal{P}} \vec{Z}$ and $\vec{Y} \sim_{\mathcal{P}} \vec{X}_{V'}$, we know by Lemma E.0.1 that $\vec{X}_{W'} \lesssim_{\mathcal{P}} \vec{Z}$ and $\vec{Y} \lesssim_{\mathcal{P}} \vec{X}_{V'}$. Recalling that $\vec{Z} \lesssim_{\mathcal{P}} \vec{Y}$, we can establish by transitivity that $\vec{X}_{V'} \lesssim_{\mathcal{P}} \vec{X}_{W'}$ and therefore that $(V', W') \in \mathcal{R}$. □

Appendix F

Proof of Lemmas 6.2.1 and 6.3.1 and Theorem 6.3.1

To prove the soundness of our algorithm, we need to prove the soundness of its three phases: translation from types to grammars, grammar pruning and exploration of an $\mathcal{XYZW}$-expansion tree. The soundness of the first phase is guaranteed by Theorem 6.1.1 (with its proof given in Appendix E). It remains now to show that the remaining phases are also sound. We begin with the second phase, grammar pruning.

Lemma 6.2.1 (Soundness and completeness for pruning)**.** $\vec{X} \preceq_{\mathcal{P}}^{\mathcal{XYZW}} \vec{Y}$ *iff* $\vec{X} \preceq_{\mathsf{prune}(\mathcal{P})}^{\mathcal{XYZW}} \vec{Y}$

Proof. For the direct implication, the $\mathcal{XYZW}$-simulation for $\vec{X}$ and $\vec{Y}$ over $\mathcal{P}$ is also an $\mathcal{XYZW}$-simulation for $\vec{X}$ and $\vec{Y}$ over $\mathsf{prune}(\mathcal{P})$. For the reverse implication, if $\mathcal{R}'$ is an $\mathcal{XYZW}$-simulation for $\vec{X}$ and $\vec{Y}$ over $\mathsf{prune}(\mathcal{P})$, then relation

$$\mathcal{R} = \mathcal{R}' \cup \{(\vec{Z}Y, \vec{Z}Y\vec{W}) \mid (X \to \vec{Z}Y\vec{W}) \in \mathcal{P}, Y \text{ unnormed}\}$$

is an $\mathcal{XYZW}$-simulation for $\vec{X}$ and $\vec{Y}$ over $\mathcal{P}$. □

We now turn to the soundness of the last phase. The exploration of the $\mathcal{XYZW}$-expansion tree is carried out through expansion and simplification steps. Expansion steps attempt to build an $\mathcal{XYZW}$-simulation from an initial pair of grammar words $(\vec{X}, \vec{Y})$, while simplification steps are used modify the construction of the tree, attempting to keep some branches of the tree finite even when the corresponding $\mathcal{XYZW}$-simulation is not. This procedure returns **True** whenever an empty node is reached (meaning there is a successful branch), or **False** if all nodes fail to expand (meaning there is no successful branch). Given these stopping conditions, the soundness of the exploration procedure relies on the tree it yields having the following property: there is a successful branch iff $\vec{X} \preceq_{\mathcal{P}}^{\mathcal{XYZW}} \vec{Y}$. This is known as the *safeness property*.

The proof of this property is given in two parts: in the first we show the property holds for a tree constructed through expansion steps only; in the second we show that the simplification rules modify the tree safely, i.e., if their application results in a tree with a successful branch, then an $\mathcal{XYZW}$-simulation can actually be constructed.

Lemma 6.3.1 (Safeness property for $\mathcal{XYZW}$-simulation)**.** *Given a set of productions $\mathcal{P}$, it holds that $\vec{X} \preceq_{\mathcal{P}}^{\mathcal{XYZW}} \vec{Y}$ iff the expansion tree rooted at $\{(\vec{X}, \vec{Y})\}$ has a successful branch.*

Proof. In an expansion tree without simplification both directions follow directly from the definition of expansion. Recall that in an $\mathcal{XYZW}$-expansion tree, each child is an $\mathcal{XYZW}$-expansion of its parent. Observe then that in an $\mathcal{XYZW}$-expansion tree rooted at $\{(\vec{X}, \vec{Y})\}$, the union of all nodes along a successful branch (i.e. infinite or containing an empty leaf) constitutes an $\mathcal{XYZW}$-simulation (over productions $\mathcal{P}$) that includes $(\vec{X}, \vec{Y})$. Hence $\vec{X} \lesssim_{\mathcal{P}} Y$.

In an expansion tree with simplifications, the union of all nodes along a successful branch need not be an $\mathcal{XYZW}$-simulation, only a (hopefully finite) representation of it, i.e., a set with which we can reconstruct it by reversing the simplifications. It remains to show how this can be done for each simplification rule:

- REFLEXIVITY: Let N be a node at depth n such that $\{(\vec{X}_i, \vec{X}_i)\}_{i \in 1..j} \subseteq N$ for some j. Applying REFLEXIVITY, its simplification is $N' = N \setminus \{(\vec{X}_i, \vec{X}_i)\}_{i \in 1..j}$. Observe that the reflexive closure of the union of all nodes along the successful branch containing N' is an $\mathcal{XYZW}$-simulation containing N.
- PREORDER: Let $\leq_N$ be the least preorder containing the ancestors of a node N. Applying PREORDER, its simplification is $N' = N \setminus \leq_N$. Observe that the reflexive and transitive closure of the union of all nodes along the successful branch containing N' is an $\mathcal{XYZW}$-simulation containing N.
- SPLIT: Let N be a node containing a pair of the form $(X_0\vec{X}, Y_0\vec{Y})$ with $\mathsf{norm}(X_0) \leq \mathsf{norm}(Y_0)$ (the case where $\mathsf{norm}(X_0) > \mathsf{norm}(Y_0)$ is similar). Let sequence $\vec{a} = a_1, \ldots, a_k$ be a minimal path for X_0, and $\vec{Z}$ be a word such that $Y_0 \xrightarrow{\vec{a}}_{\mathcal{P}} \vec{Z}$. Applying SPLIT to N yields N itself and a sibling N' with pairs $(X_0\vec{Z}, Y_0), (\vec{X}, \vec{Z}\vec{Y})$ in place of $(X_0\vec{X}, Y_0\vec{Y})$. We need to show that, assuming there is an $\mathcal{XYZW}$-simulation over $\mathcal{P}$ containing N', it is possible to obtain an $\mathcal{XYZW}$-simulation over $\mathcal{P}$ containing N.
  Let $\mathcal{R}'$ be an $\mathcal{XYZW}$-simulation over $\mathcal{P}$ containing N' and $\mathcal{S}_1$ be the smallest $\mathcal{XYZW}$-simulation over $\mathcal{P}$ that includes pair $(X_0\vec{Z}, Y_0)$. Then, relation

$$\mathcal{R} = \mathcal{R}' \cup \{(X_0\vec{X}, Y_0\vec{Y})\} \cup \{(\vec{X}_1\vec{X}, \vec{Y}_1\vec{Y}) \mid (\vec{X}_1\vec{Z}, \vec{Y}_1) \in \mathcal{S}_1\}$$

  is an $\mathcal{XYZW}$-simulation over $\mathcal{P}$ containing N.
  Since N' includes all pairs of N except $(X_0\vec{X}, Y_0\vec{Y})$ and $\mathcal{R}'$ is an $\mathcal{XYZW}$-simulation over $\mathcal{P}$ containing N', it follows from the union of $\mathcal{R}'$ with $\{(X_0\vec{X}, Y_0\vec{Y})\}$ that $\mathcal{R}$ contains N.
  All that remains now is to show that $\mathcal{R}$ is an $\mathcal{XYZW}$-simulation over $\mathcal{P}$. Since $\mathcal{R}'$ is already such a relation, we need to show that it remains so after adding to it every pair in $\{(X_0\vec{X}, Y_0\vec{Y})\} \cup \{(\vec{X}_1\vec{X}, \vec{Y}_1\vec{Y}) \mid (\vec{X}_1\vec{Z}, \vec{Y}_1) \in \mathcal{S}_1\}$. For $(X_0\vec{X}, Y_0\vec{Y})$ this is easy: we observe that the pairs containing the derivatives of all matching transitions of $X_0\vec{X}$ and $Y_0\vec{Y}$ are elements of $\{(\vec{X}_1\vec{X}, \vec{Y}_1\vec{Y}) \mid (\vec{X}_1\vec{Z}, \vec{Y}_1) \in \mathcal{S}_1\}$. For the pairs in $\{(\vec{X}_1\vec{X}, \vec{Y}_1\vec{Y}) \mid (\vec{X}_1\vec{Z}, \vec{Y}_1) \in \mathcal{S}_1\}$ we need to distinguish two cases:
  - (**Case** $\vec{X}_1 \neq \varepsilon$): We observe that, from the definition of $\mathcal{S}_1$, it follows that relation

$\{(\vec{X}_1\vec{X}, \vec{Y}_1\vec{Y}) \mid (\vec{X}_1\vec{Z}, \vec{Y}_1) \in \mathcal{S}_1\}$, which is contained in $\mathcal{R}$, contains the paired derivatives of all matching transitions of $X_1\vec{X}$ and $Y_1\vec{Y}$.

– (**Case** $\vec{X}_1 = \varepsilon$): We observe that, since $\mathcal{R}$ contains $\mathcal{R}'$ (an $\mathcal{XYZW}$-simulation containing N'), it must include the paired derivatives of all matching transitions of $\vec{X}$ and $Y_1\vec{Y}$.

□

With the safeness property established, we can finally put together a soundness proof encompassing all of the three phases of the algorithm.

Lemma F.0.1. *If* $\mathsf{subG}(\vec{X}_T, \vec{X}_U, \mathsf{prune}(\mathcal{P}))$ *returns* **True***, then* $\vec{X}_T \lesssim_{\mathsf{prune}(\mathcal{P})} \vec{X}_U$.

Proof. Function subG returns **True** whenever it finds a finite successful branch (i.e., a branch terminating in an empty node) in the expansion tree rooted at $\{(\vec{X}, \vec{Y})\}$. Conclude with the safeness property, Lemma 6.3.1. □

Theorem 6.3.1 (Soundness)**.** *If* $\mathsf{subT}(T, U)$ *returns* **True***, then* $T \lesssim U$.

Proof. From Theorem 6.1.1, Lemma 6.2.1 and Lemma F.0.1. □