UNIVERSITÉ SAVOIE MONT BLANC

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ SAVOIE MONT BLANC

Spécialité: Mathématiques et Informatique
Arrêté ministériel: 25 Mai 2016

Présentée par

Peio BORTHELLE

Thèse dirigée par **M. Tom HIRSCHOWITZ**
préparée au sein du **LAMA**
dans l'École Doctorale **MSTII**.

Sémantique des Jeux Operationelle en Théorie des Types

Thèse soutenue publiquement le **12 Mars 2025**
devant le jury composé de :

M. David MONNIAUX
Directeur de Recherche, CNRS, Université Grenoble Alpes, Président
Mme. Stephanie WEIRICH
Distinguished Professor, University of Pennsylvania, Rapportrice
M. Damien POUS
Directeur de Recherche, CNRS, ENS de Lyon, Rapporteur
M. Pierre-Évariste DAGAND
Chargé de Recherche, CNRS, Université Paris Cité, Examinateur
M. Hugo HERBELIN
Directeur de Recherche, INRIA, Université Paris Cité, Examinateur
M. Paul Blain LEVY
Senior Lecturer, University of Birmingham, Examinateur
Mme. Kathrin STARK
Assistant Professor, Heriot-Watt University, Examinatrice
M. Tom HIRSCHOWITZ
Directeur de Recherche, CNRS, Université Savoie Mont Blanc, Directeur de thèse
M. Guilhem JABER
Maître de Conférences, Nantes Université, Invité, Co-encadrant de thèse
M. Yannick ZAKOWSKI
Chargé de Recherche, INRIA, ENS de Lyon, Invité, Co-encadrant de thèse

Operational Game Semantics in Type Theory

Peio Borthelle

Sémantique des jeux operationelle en théorie des types

Cette thèse a été composée en Typst avec les polices EB Garamond, New Computer Modern Math, Latin Modern Sans et Fira Code.

Abstract

In this thesis we construct an operational game semantics (Ogs) model entirely formally in the language of type theory, and prove it correct w.r.t. observational equivalence. These results are implemented in a Rocq code artifact. The Ogs model construction and correctness proof are generic over an axiomatized programming language and its evaluator. The axiomatization in the style of abstract machines handles simply-typed languages with arbitrary control-flow effects (non-termination, call/cc), of which we provide three examples: Jump-with-Argument, polarized $\mu\tilde{\mu}$-calculus with recursive types, and pure untyped call-by-name λ-calculus under weak head reduction. The Ogs model construction builds upon a notion of game by Levy and Staton, and strategies are represented coinductively by generalizing Xia *et al.*'s interaction tree from containers to indexed containers. We further introduce a novel unique fixed point construction for eventually guarded equation systems on (indexed) interaction trees, as well as a generic normal form bisimulation model construction and its correctness proof.

Résumé

Cette thèse construit un modèle de sémantique des jeux opérationelle (Ogs) de manière entièrement formelle dans le langage de la théorie des types, et prouve sa correction vis-à-vis de l'équivalence observationelle. Ces résultats sont mécanisés avec l'assistant de preuve Rocq. La construction du modèle d'Ogs et sa correction sont génériques par rapport à un langage de programmation axiomatisé et à son évaluateur. L'axiomatisation dans le style des machines abstraites capture les langages simplement typés avec effets de contrôle arbitraires (non-terminaison, call/cc), et nous en présentons trois exemple: Jump-with-Argument, $\mu\tilde{\mu}$-calcul polarisé avec types récursifs, et λ-calcul pur non-typé en réduction de tête faible. La construction du modèle d'Ogs se base sur une notion de jeu de Levy et Staton, et les stratégies sont représentées coinductivement en généralisant la définition d'arbre d'intéraction de Xia *et al.* aux conteneurs indexés. Nous introduisons également une nouvelle construction de point fixe pour les systèmes d'équations ultimement gardés sur les arbres d'intéraction (indexés), ainsi qu'une construction générique d'un modèle de bisimulation de forme normale et sa preuve de correction.

Acknowledgments

First of all I would like to thank Damien, David, Hugo, Kathrin, Paul, Pierre-Évariste and Stephanie for accepting to be part of my jury. It is an honor to be judged by so many people I deeply respect. I am particularly grateful to Stephanie and Damien, for their thorough reviews and proof reading, as well as their insightful suggestions.

Cette thèse n'aurait bien sûr pas pu se dérouler sans la supervision attentive de mon *triumvirat* personel composé de Tom, Guilhem et Yannick. Que ce soit sur le terrain scientifique ou dans d'autres domaines, vous m'avez à la fois donné une confiance et un enthousiasme sans limite, et su me tempérer et me remettre dans le droit chemin avec patience chaque fois qu'il le fallait. Ces trois et quelques années riches en enseignements ont été un réel plaisir. Pour cela vous avez toute ma gratitude, je n'aurais rien pu espérer de mieux.

Je remercie toute l'équipe du Lama pour tout les bons moments en salle café et ailleurs. Je pense entre autres aux habitués du Sco, Pierre, Jacques-O, Sébastien, Valentin, mais aussi à Laure, pour l'efficacité surnaturelle (et le canapé). Je pense évidemment aussi aux occupant·e·s des bureaux 20(bis), à Colin, Yen-Chung et à tou·te·s les autres, à Yoann, mon jumeau de thèse et de toit, pour toutes ces pizzas catégorico-philosophiques!

Merci également à toute l'équipe de Gallinette pour les accueils très chaleureux à Nantes, c'était toujours un plaisir de discuter de philo ou des dernières trouvailles en formalisation. À Lyon, merci à toute l'équipe de Cash qui m'a soutenue dans la dernière ligne droite, et plus largement à l'organisation et aux habitué·e·s des rencontres Chocola, qui m'ont fourni un régulier bol d'air frais. Merci Daniel pour ta bonne hum{e,o}ur contagieuse et ta simplicité perspicace, et pour Fouine aussi!

A big shout out to all the type theory enthusiasts I have met here and there. Thanks to the Dutch and the Chilean crews at Oplss'22 for all the fun. Cheers to the Scots who warmly welcomed me in 2018, Conor you have provided me with some long-lasting inspiration. Cheers also to the Agda community for all the hacking, at Tu Delft and online, thank you Miëtek for bringing people together on Irc.

Mais le boulot ça fait pas tout, donc à toutes les personnes, famille ou ami·e·s plus ou moins proches avec qui j'ai pu ces dernières années passer des supers moments de jeu, de politique, de musique, de peine, de réflexion, de baignades, de blagues: merci d'exister! Évidemment le gang des sbires, marmite à projets farfelus et soupape de décompression, Romain, Lescro, Vreb, Gliboc, Calvin, Pierre, Jdc. La team Rk, toujours là pour refaire le monde autour d'un baby-foot endiablé ou d'une partie de cartes, Léna, Clair, Samuel, Quentin, Joséphine, Rémy, Ciara, Antoine et tou·te·s les autres... Les invocateurs fous de la Martin', Paul, Victor, Alexis, Brice, Corentin, Manu. Joseph et Loïs pour toutes les découvertes musicales et politiques, Léo, quand même, pour tous ces super moments passés à regarder des classiques. La grande famille de Die, celle de Soleihas, d'Artignosc et de S^{t} André, pour votre bienveillance et votre curiosité, Véro, Tif, Nico, Max, Damien, Clémence $\times$ 2, Marie, Pauline, Constance, Thomas $\times$ 2, Marine... Mention spéciale à Clément et Tristan pour tous les projets à la maison, les rigolades, la cuisine, les idées, c'est tellement agréable de vivre avec vous. Et Méli, que dire... t'avoir à mes côtés me rempli de tellement d'énergie. Merci à toute ma famille, papa, maman, pour tout le soutient, l'éveil à la créativité et à la curiosité, depuis si longtemps, vous êtes formidables.

J'espère ne pas avoir oublié trop de gens, vous vous reconnaîtrez. Par ailleurs, si cette page vous intéresse plus que les suivantes et que vous en voulez encore, je vous renvoie au travail de Tabitha Carvan[1].

À Gribouille, Farine, M^{lle} Jeanne, Gravillon, Looping, Gaston, Roc, Ratatouille et Albus, pour tous vos câlins.

Domène, 22 février 2025, P.B.

[1] https://science.anu.edu.au/news-events/news/unexpected-poetry-phd-acknowledgements

Contents

1 Introduction

1.1 Interactive Semantics: Context and Motivations

This thesis sets itself in the field of *programming language semantics*, whose central question, "What is the meaning of this program?", is an essential premise to any mathematical study of programs and programming languages. As it happens, most mainstream programming languages are large and complex beasts, comprising a number of intertwined features and subtle interactions with their underlying runtime system. Programs are thus far removed from the more neatly described traditional objects of mathematical interest such as say numbers or vector spaces. Paradoxically, although programs are purely formal constructs, this complexity gives to the study of their semantics the feel of a natural science, where mathematical models are built to capture an ever increasing level of detail. For sure, a handful of languages do admit truly exhaustive semantic descriptions, but this arguably only ever happens when they are designed with this intent (e.g., Standard ML [84] or WebAssembly [41]). Even for programming languages strongly rooted in the theoretical computer science community and routinely used by semanticists, such as proof assistants like Agda [9] or Coq [28], a perfect description is elusive*! Yet, this predicament never stopped anyone from programming, nor simplified semantics from being useful. The value of a mathematical model does not reside in its faithfulness to reality with all its gruesome details, but in its ability to ease reasoning about a particular aspect of interest.

In this thesis, the focus of attention begins with the following question: When can two *program fragments* be considered equivalent? The motivation for studying program fragments—i.e., code snippets, modules, individual functions, etc.—is of practical nature. It can be abstractly argued that approaching large programs is most easily done by cutting them into smaller parts to be studied independently. But to a greater degree, programs are *written* modularly in the first place. Programming languages live and die by the means they provide to organize abstractions and interfaces. Organizing bits and pieces is perhaps the primary task of the programmer. As such, studying equivalence will be our first step towards giving a mathematical meaning to program fragments. This proves to be sufficient for a variety of tasks such as justifying correctness by comparison to a reference implementation, or verifying optimizations and simplifications, in particular in the context of compilation—the art of translating programs.

[84] David MacQueen, Mads Tofte Robin Milner Robert Harper, *The Definition of Standard ML*, 1997.

[41] WebAssembly Working Group, "WebAssembly Specification."

[9] Agda Developers, "Agda."

[28] The Coq Development Team, "The Coq Proof Assistant," 2024.

* To be completely honest, in the case of Coq *kernel*, the MetaCoq [72] project is increasingly close to the ground truth.

[74] James H. Morris, "Lambda-calculus models of programming languages," 1968.

The most important definition for equivalence of program fragments is due to Morris [74]: *observational* (or *contextual*) equivalence. It builds upon the notion of *context*, that is, an incomplete program with a *hole*, a missing part clearly marked as such. Primarily, a context can be combined with a program fragment by replacing the marked hole with the fitting fragment, yielding a complete program. Two program fragments a and b are then considered *observationally equivalent*, whenever for any context C they might be combined with, the final results of the combinations $C[a]$ and $C[b]$ will satisfy exactly the same basic observations. This assumes given some sensible notion of "basic observation" on program results. As the name suggests, observational equivalence characterizes program fragments which cannot be distinguished in any way by programatically interacting with them. It is in a precise sense the "best" (logically weakest) such notion. Observational equivalence is the gold standard, but also notoriously hard to work with directly. Indeed, it is as much of a property on two program fragments, as a statement on *all* the contexts they could fit in. As simple as the programs may be, the intricacy of the possible contexts is without limit. In consequence, a fruitful area of research has been to develop alternative characterizations of observational equivalence, more intrinsically related to the programs at hand, with the hope of being easier to establish in concrete cases. Among such efforts, *game semantics* is one particular strand of prime importance to us.

A core idea of game semantics is to abstract away the concrete nature of the observing contexts and instead put the focus on the ways they would *interact* with a given program fragment. By keeping the inner working of the context opaque, we can concentrate on what actually matters, that is, how it would affect our program fragment. To set things straight, let us illustrate this idea with a short example: assume our program fragment is the following function `twice`.

$$\texttt{twice}(\texttt{f}, \texttt{x}) \coloneqq \texttt{f}(\texttt{f}(\texttt{x}))$$

A sample (polite) interaction with an otherwise unknown context might look like this.

Program — You can ask me about `twice`.
Context — What is the output of $\texttt{twice}(\texttt{a}, 10)$? You can ask me about `a`.
Program — Well, first what is the output of $\texttt{a}(10)$?
Context — The output is 5.
Program — Still, what is the output of $\texttt{a}(5)$?
Context — It is 3.
Program — Then the output you asked for is 3.

Different concrete contexts might generate this interaction with `twice`. For example, several implementations of `a` would fit the bill of mapping 10 to 5

and 5 to 3. However, any such context would only learn the same information about our candidate fragment, namely that when called with such a function `a` and the number 10, it will output 3. For the purpose of testing the behavior of `twice`, we do not loose anything to conflate all these contexts. The precise rules of such dialogues will be elucidated in due time, but for now let us return to our exposition.

Under the light of interactions, a program fragment can be understood on its own without any mention of contexts, as a blueprint or *strategy*, describing how to react to any possible action by the other party. Equivalence of such strategies —reacting the same way to any action—can then serve as replacement for observational equivalence. Put more succinctly, game semantics is a family of models in which program fragments are interpreted as strategies for restricted forms of dialogues between them and the rest of the program. Although the rules of these dialogues are called "games", bear in mind that they are only tangentially related to the games studied in game *theory*, e.g., they are devoid of any notion of reward. This basic idea is quite flexible and suitable to model a wide range of programming language features. In fact it originates as part of a wider interpretation not of programs but of *proofs*, i.e., evidence in logical systems. Girard's geometry of interaction [38] has been particularly influential in this respect, but we will not try to trace the game interpretation of logic down to its origins, since the connection between logical proofs and argumentative dialogues is most likely as old as logic itself.

[38] Jean-Yves Girard, "Geometry of Interaction 1: Interpretation of System F," 1989.

Although it has been motivated as a practical tool for reasoning with observational equivalence, and although its flexibility has been demonstrated to apply to a number of advanced programming language features, game semantics has not yet truly gone out of the hands of the game semanticists and into the everyday toolkit of the generalist programming language researcher. This can be contrasted, e.g., with the framework of *logical relations*, also known as Tait's method of computability [90], which has overlapping use cases [34] and which enjoys a very large number of introductory material, tutorials and other proof walk-through. While game semantics is relatively lively as a research area, it has seen comparably little activity in digesting existing methods, streamlining proofs and definitions, and making them technically approachable to a wider community. This thesis is no tutorial, but we will keep this motivation in the back of the mind.

[90] William W. Tait, "Intensional Interpretations of Functionals of Finite Type I," 1967.

[34] Derek Dreyer, Georg Neis, and Lars Birkedal, "The impact of higher-order state and control effects on local relational reasoning," 2012.

More concretely, the goal of this thesis is to reconstruct from the ground up a particular flavor of dialog models, *operational game semantics* (Ogs), and prove that equivalence of strategies entails observational equivalence. We do not build it for a precise programming language, but instead target a generic construction, readily applicable to several languages. This generality is not for the sake of it, but practically guided by getting to the core of the construction and separating it from

the technicalities pertaining to the given language. We do so fully formally, in the mathematical language of *type theory* common in computer-assisted proving. Before jumping to the content we will provide a bit more details on operational game semantics, but for now let us discuss some of the methodological underpinnings of computer-assisted proofs.

1.2 Programming Mathematics: How and Why?

For most people, the phrase "computer-assisted proof" usually evokes the idea of a computer program, checking if some formula or some other kind of mathematical problem is true or false. This kind of algorithm, known as *solvers* are indeed useful in a number of situations, but several landmark results early in the 20th century have shown that this can very quickly become unfeasible. First, Gödel [42] showed that for any logical system, as soon as some moderate level of expressivity is attained, there are some statements which are neither provable nor disprovable. Quickly thereafter, Church [25] and Turing [92] independently showed that there are some tasks which no program can solve, i.e., functions that we can specify but not *compute*. Hence, the "computer assistance" we make use of in this thesis is of another kind. Instead of asking for a program to come up with the proofs, we write them ourselves in a purpose designed language. A program understanding this language then helps us both during the writing, e.g., by providing information about the ongoing proof, and at the end, when it checks that the proofs we have written are correct and missing no argument.

[42] Kurt Gödel, "Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I," 1931.
[25] Alonzo Church, "An Unsolvable Problem of Elementary Number Theory," 1936.
[92] Alan M. Turing, "On Computable Numbers, with an Application to the Entscheidungsproblem," 1937.

There are a number of such systems, but the one we have used in the code artifact accompanying this thesis is Coq [28] (henceforth the Rocq Prover, as it is in the process of being renamed). It is part of a wider family of systems which we will simply call *type theories* and whose distinguishing characteristic is that they are in fact programming languages, although perhaps of a slightly peculiar kind. To explain this fact and its importance in our approach to proving mathematical theorems, we need to take a slight step back.

[28] The Coq Development Team, "The Coq Proof Assistant," 2024.

The beginning of this happy coincidence joining logic and programming started when, still in the early 20th century, some mathematicians started insisting that to consider proven the existence of some mathematical object satisfying some property, one must actually pinpoint it precisely. In other words, we need to provide a concrete way to construct it, so that it is not sufficient to merely show that it is impossible for it not to exist, without saying anything about its shape. In this *intuitionistic* or *constructive* school of thought, the idea emerged that to any such intuitionistic proof can be associated a concrete witness or *realizer*, the so-called Brouwer-Heyting-Kolmogorov interpretation. Programs turned out to be quite suitable for expressing such realizers, in particular the λ-calculus

put forward by Church [51][52], establishing a link between proofs and programs. The bridge between these two worlds is however quite deeper, as around that time and onward, a string of observations have been made. They essentially understood, that with a very moderate amount of squinting, the linguistic rules of existing formal logical systems were actually the same as the rules of existing programming languages. Although not an actual theorem, this has been extrapolated to a guiding principle, or methodological posture, the Curry-Howard correspondence, asserting that logical systems and proofs on one hand, and programming languages and programs on the other, are the two sides of the same coin.

[51] Stephen C. Kleene, "On the interpretation of intuitionistic number theory," 1945.
[52] Georg Kreisel, "Interpretation of Analysis by Means of Constructive Functionals of Finite Types," 1959.

This correspondence sparked a particularly fruitful activity in logic and theoretical computer science, namely taking a concept from one side and looking at it from the other side. We can now study the behavior of a proof when executed, compare the computational complexity of two proofs of the same statement, search for the statement(s) that a given algorithm proves, and much more! Type theories such as Rocq embody this correspondence, so that we are truly both programming and proving at the same time. I personally believe that this opens up a radically new perspective both on programming and on mathematical practice of which I outline three important aspects.

A first aspect, relevant to the programmer, is that of *correct-by-construction* programming, or type-driven development [16][22], whereby a program is its own proof of correctness. *Types*, akin to syntactic categories, classify programs and they are the counterpart of the logical *propositions*, or statements, in the programming world. Most programmers are probably familiar with some types, such as booleans, byte arrays, functions from some type A to some type B, records, etc. In type theories such as Rocq, these types are now able to express a number of powerful logical constructions. Instead of writing a program and then tediously proving that it is correct w.r.t. some specification, we can thus write that specification as a type, write the program, and then simply typecheck it, i.e., ask the system to check that our program has the given type, in other words that it verifies the corresponding specification. This is no magic bullet: if there are non-trivial arguments to be had as to why the program corresponds to the specification, they will now be required to be put alongside it. Still, a number of invariants can be pushed into programs and data structures in this way, so that a host of basic properties are tracked and enforced effortlessly by the language itself. We use this idiom extensively throughout our Rocq development.

[16] Thorsten Altenkirch, Conor McBride, and James McKinna, "Why Dependent Types Matter," 2005.
[22] Edwin Brady, *Type-Driven Development with Idris*, 2017.

A second aspect is relevant to the mathematician. As we stated earlier, programmers, with the support of computer science, have become quite good at organizing code modularly, a necessity to ensure ease of use, maintainability, extensibility, etc. This experience learned the hard way by managing large bodies of formal

constructs can now be pulled across the Curry-Howard correspondence and leveraged to organize mathematical concepts. In some respect, mathematicians have also been preoccupied by properly organizing concepts, however, we dare to say that this has been a rather organic task, mostly left to aesthetic judgment and to time. By inflicting upon oneself the rigor of entirely formal reasoning as is done when using proof assistants, there is no other way than to rationalize the concepts down to their core. When programming, nothing can be "left for the reader", which is the gentleman's agreement providing an escape hatch for uninteresting and tedious mathematical writing. Such untold details are usually self-evident for trained mathematicians with a good *mental* representation of the objects in question, as indeed, the problem is only *formal*. When programming, the similar phenomenon of "boilerplate" or "glue code", is usually caused by improper data representations or missing abstractions and it can be resolved by refactoring. As such, formalization can encourage mathematicians to question the presentational status quo, and provide sound criteria for judging the suitability of abstractions.

A third aspect concerns *accessibility*. Programming is quite notable for its number of self-taught practitioners, sometimes of very young age or otherwise well outside of the intended public. While we do not pretend to explain this fact, we believe that two points must be part of the picture. First, explicitness. Although as semanticists we can regret some imperfections, programming languages are usually thoroughly documented in plain words, with large manuals describing every element of the syntax and their meaning. As each and every program is entirely described by this syntax, one does not need to know the theoretical background of some algorithm to decipher its atomic operations. Instead programs can be reverse-engineered, starting from a purely superficial formal understanding. Although the learning curve may be steep, extremely little background knowledge is required: most things are built from the ground up and can be inspected. Second, interactivity. Computers turn programs into a reality which can be poked at and experimented with simply by running them and testing their behavior. The process of programming itself is interactive: at the very least one can always try to execute the program, it will either run or produce an error, reporting what went wrong. In practice, advances in code editors and other tooling have made the process of writing code vastly more supportive. This interactivity enables ingenuous trial-and-error, and removes the requirement of having a knowledgeable person around for pointing out errors. All in all, there is no reason to believe that mathematics and computer science are better or worse than any other academic discipline with respect to gatekeeping. Yet a large amount of crucial knowledge is hidden behind unspoken conventions, behind a number of little ambiguities that the reader is assumed to resolve, or inside a sea of publications that requires a full time job and a dedicated mentor to navigate. By building upon

programming languages, which are executable and clearly specified, the bar to start understanding and producing meaningful mathematics can be enormously lowered. In fact, we can already observe several substantial contributions by students and non-professional mathematicians, in particular using type theory implementations such as Agda [9] or Lean [75] whose syntax is quite close to mainstream programming languages.

[9] Agda Developers, "Agda."
[75] Leonardo de Moura and Sebastian Ullrich, "The Lean 4 Theorem Prover and Programming Language," 2021.

With the frame of discourse now set, let us jump back to operational game semantics and provide some technical grounding for the rest of this thesis. From this point on, we will start assuming some familiarity with the λ-calculus.

1.3 A Primer to Operational Game Semantics

To set some intuitions about operational game semantics and to introduce some design choices that will follow us throughout the thesis, let us start by describing its rules and the obtained strategies in the case of a very simple programming language: simply-typed λ-calculus with recursive functions and booleans. We recall its syntax, typing, and operational semantics in Figure 1.1.

1.3.1 First Steps

Our presentation will more or less follow Laird [54], keeping only what is required for our simple example language. The game goes as follows: the two players, unoriginally named "client" and "server", exchange function *symbols* such as `twice` and `a` from the previous example, but whose associated definition they do not disclose to each other. These symbols are introduced in two possible ways: by *calling* a previously introduced symbol or by *returning* a value following a previous call. When calling, if the argument is a boolean it is passed as-is, but in case it is a function, a fresh symbol is introduced in its place. Similarly when returning, booleans are given explicitly but functions are again hidden and shared as a new symbol. This syntactic category consisting of true, false and fresh function variables can be recognized as *patterns*. Together with moves they are defined as follows.

[54] James Laird, "A Fully Abstract Trace Semantics for General References," 2007.

$$\begin{aligned} \text{Pattern} \ni Z &::= x \mid \texttt{tt} \mid \texttt{ff} \\ \text{Move} \ni M &::= \texttt{ret}(Z) \mid \texttt{call}(x, Z) \end{aligned}$$

Remark 1.1: Assuming some existing symbol $x : (A \to B) \to C$, the move $\texttt{call}(x, y)$ should really be understood as *binding* the symbol $y : A \to B$ for the other player for the rest of the play, which is moreover asserted fresh. It is a *location*, while x on the other hand is a *pointer*.

Note that what we call "symbols" in the context of the game are technically plain and simple variables. The name is a reference to the "program symbols" as appear in shared library object files.

The interpretation of terms as strategies is conceptually very simple. First, we evaluate the term to its normal form. If this does not terminate, the strategy never plays. If we find a normal form, it can be of two shapes: either a value V or a term $E[x\ V]$ stuck on a symbol x introduced by the server. In the first case we play $\mathtt{ret}(Z)$ and in the second $\mathtt{call}(x, Z)$, where Z is, depending on the type of V, either a fresh function symbol or the boolean V. Upon playing this move, we remember everything that we have not put into the move, that is, possibly the evaluation context E and the function associated to the fresh symbol. Then, to react to moves, if the server plays $\mathtt{ret}(Z)$, we find our last stored evaluation context E and restart our strategy as above with $E[Z]$. If the server instead plays $\mathtt{call}(x, Z)$, we look up the value V associated to x and restart with $V\,Z$.

To make this more precise we need to know how to represent strategies, and this is an important specificity distinguishing OGS from a lot of other game semantical models. In OGS, these are described quite concretely by mean of an *automaton*, or *transition system*, that is, by giving a set of states and a transition relation. More precisely, because a strategy needs to both play moves and respond to server

Syntax

$$\begin{array}{lll} \text{Value} & \ni V, W & ::= x \mid \mathtt{tt} \mid \mathtt{ff} \mid \lambda^{\text{rec}} f, x.P \\ \text{Term} & \ni P, Q & ::= V \mid PQ \\ \text{Eval. Cont.} & \ni E, F & ::= \Box \mid EV \mid PE \end{array}$$

Reduction

$$\begin{array}{rcll} (\lambda^{\text{rec}} f, x.P)V & \rightsquigarrow & P[f \mapsto (\lambda^{\text{rec}} f, x.P), x \mapsto V] & \\ E[P] & \rightsquigarrow & E[Q] & \text{whenever } P \rightsquigarrow Q \end{array}$$

Typing

$$\begin{array}{lll} \text{Type} & \ni A, B & ::= A \to B \mid \mathtt{bool} \\ \text{Scope} & \ni \Gamma, \Delta & ::= \varepsilon \mid \Gamma, x : A \end{array}$$

$$\frac{\Gamma \ni x : A}{\Gamma \vdash x : A} \qquad \frac{}{\Gamma \vdash \mathtt{tt} : \mathtt{bool}} \qquad \frac{}{\Gamma \vdash \mathtt{ff} : \mathtt{bool}}$$

$$\frac{\Gamma, f : A \to B, x : A \vdash P : B}{\Gamma \vdash \lambda^{\text{rec}} f, x.P : A \to B} \qquad \frac{\Gamma \vdash P : A \to B \quad \Gamma \vdash Q : A}{\Gamma \vdash PQ : B}$$

Figure 1.1 – STLC Syntax and Semantics

moves, there will be two sets of states and two transition relations, respectively for *active positions* and *passive positions*. We adopt the point of view of the client, so that active positions are the ones where we need to play a move, while passive positions are the ones in which we are waiting for the server to play.

For our OGS strategy, an active state consists of a term P being evaluated, and the data that we need to remember: a stack of evaluation contexts S and an environment of function values γ. Passive states only contain the evaluation stack and the environment. We write them respectively as $\mathtt{act}(P, S, \gamma)$ and $\mathtt{pas}(S, \gamma)$. Before giving the transitions, let us make precise the *abstraction* procedure, which hides function expressions from values. It takes a type and a value of that type and returns a pattern, together with a *filling*, i.e., an environment containing the value which may have been abstracted.

$$\begin{aligned}\mathrm{abstr}_{S\to T}(V) &:= (x, \{x \mapsto V\}) \quad \text{with } x \text{ a fresh symbol}\\ \mathrm{abstr}_{\mathtt{bool}}(V) &:= (V, \{\})\end{aligned}$$

The transitions are given by relations between states, labeled by the move M which is played or received. The active and passive transitions are given as follows.

$$\begin{array}{lcll}
\mathtt{act}(P,S,\gamma) & \xrightarrow{\mathtt{ret}(Z)} & \mathtt{pas}(S,\gamma \uplus \delta) & \text{whenever } P \rightsquigarrow^* V \text{ and } (Z,\delta) := \mathrm{abstr}(V)\\
\mathtt{act}(P,S,\gamma) & \xrightarrow{\mathtt{call}(x,Z)} & \mathtt{pas}(S::E,\gamma \uplus \delta) & \text{whenever } P \rightsquigarrow^* E[xV] \text{ and } (Z,\delta) := \mathrm{abstr}(V)\\
\mathtt{pas}(S::E,\gamma) & \xrightarrow{\mathtt{ret}(Z)} & \mathtt{act}(E[Z],S,\gamma) & \\
\mathtt{pas}(S,\gamma) & \xrightarrow{\mathtt{call}(x,Z)} & \mathtt{act}(VZ,S,\gamma) & \text{when } (x \mapsto V) \in \gamma
\end{array}$$

A term P can now be interpreted as a strategy by embedding it as the initial active state $\mathtt{act}(P, \varepsilon, \{\})$. Then, strategies are considered equivalent when they are *bisimilar*. As the transition is deterministic, this essentially means that they have the same set of *traces*, that is, the same infinite sequences of moves obtained by unfolding any possible transition starting from the initial state. The primary task to make sure the model is sensible is to prove that for the above given strategy, when two terms are bisimilar, then they are observationally equivalent—a statement called *correctness*, or *soundness* of OGS w.r.t. observational equivalence.

Although this game seems a priori relatively reasonable, before starting our formal treatment in this thesis we will make a slight change of perspective. As a hint, it is slightly bothering that our strategy requires two devices instead of one for remembering what it needs to: a stack and an environment. The blocker for putting evaluation contexts into the environment is that they are not named by a symbol. Instead, they are always referred to implicitly, as e.g. in $\mathtt{ret}(Z)$ which

can be read "return Z to the context *at the top of the stack*", much like $\texttt{call}(x, Z)$ reads "send Z to the function *pointed to by* x". This change of perspective thus involves naming contexts with symbols, as functions are, and unify the return and call moves into one: symbol *observation*.

1.3.2 More Symbols and Fewer Moves

We amend the game as follows. First, the exchanged symbols may now refer either to functions or to evaluation contexts, which we will sometimes (but not always) distinguish by writing them α, β, etc. Next, as now both move kinds explicit the symbol they are targeting, we will separate it more clearly from the arguments, writing $\alpha \cdot \texttt{ret}(Z)$ and $x \cdot \texttt{call}(Z, \alpha)$. Note that function calling now has a second argument α, binding the continuation symbol on which this call expects a return. They are precisely defined as follows.

The Agda programmer or $\mu\tilde{\mu}$-calculus aficionado will surely be happy to recognize "observations" as copatterns and "moves" as postfix copattern applications.

$$\begin{aligned} \text{Observation} \ni O &::= \texttt{ret}(Z) \mid \texttt{call}(Z, \alpha) \\ \text{Move} \qquad \ni M &::= x \cdot O \end{aligned}$$

To adapt the Ogs strategy to this new game, we do away with the context stack, as was our motivation. In compensation, we now need to track explicitly the "current context symbol". The active states thus comprise a *named* program $\langle P \parallel \alpha \rangle$, i.e. a pair of a program and a continuation symbol, and an environment γ mapping symbols to generalized values, that is, function symbols to function values and context symbols to *named contexts* $\langle E \parallel \alpha \rangle$. The passive states now simply consist of an environment. The transition system can be rewritten as follows.

$$\begin{array}{lcll} \texttt{act}(\langle P \parallel \alpha \rangle, \gamma) & \xrightarrow{\alpha \cdot \texttt{ret}(Z)} & \texttt{pas}(\gamma \uplus \delta) & \text{whenever } P \rightsquigarrow^* V \text{ and } (Z, \delta) \coloneqq \text{abstr}(V) \\ \texttt{act}(\langle P \parallel \alpha \rangle, \gamma) & \xrightarrow{x \cdot \texttt{call}(Z,\beta)} & \texttt{pas}(\gamma \uplus \delta') & \text{whenever } P \rightsquigarrow^* E[xV], (Z, \delta) \coloneqq \text{abstr}(V) \text{ and } \delta' \coloneqq \delta \uplus \{\beta \mapsto \langle E \parallel \alpha \rangle\} \\ \texttt{pas}(\gamma) & \xrightarrow{\alpha \cdot \texttt{ret}(Z)} & \texttt{act}(\langle E[Z] \parallel \beta \rangle, \gamma) & \text{when } (\alpha \mapsto \langle E \parallel \beta \rangle) \in \gamma \\ \texttt{pas}(\gamma) & \xrightarrow{x \cdot \texttt{call}(Z,\beta)} & \texttt{act}(\langle V Z \parallel \beta \rangle, \gamma) & \text{when } (x \mapsto V) \in \gamma \end{array}$$

To put the final blow, let us fuse the definition of the return and call moves, for active transitions and passive transitions. For the active transitions, notice that in essence, what is happening is that the named program is reduced to a named normal form, which is subsequently split into three parts: the head symbol, an observation on it with more or less opaque arguments, and a small environment

storing the value of anything that has been hidden. Let us define this splitting as follows, where as usual we make use of $(Z, \gamma) \coloneqq \mathrm{abstr}(V)$.

$$\begin{aligned}\text{split } \langle V \parallel \alpha\rangle &\coloneqq (\alpha \cdot \mathtt{ret}(Z), \gamma)\\ \text{split } \langle E[xV] \parallel \alpha\rangle &\coloneqq (x \cdot \mathtt{call}(Z, \beta), \gamma \uplus \{\beta \mapsto \langle E \parallel \alpha\rangle\})\end{aligned}$$

Our active transition now satisfyingly reads as follows.

$$\mathtt{act}(\langle P \parallel \alpha\rangle, \gamma) \xrightarrow{x \cdot O} \mathtt{pas}(\gamma \uplus \delta) \quad \begin{array}{l}\text{whenever } P \rightsquigarrow^* N \text{ and}\\ (x \cdot O, \delta) \coloneqq \text{split } \langle N \parallel \alpha\rangle\end{array}$$

To unify the two passive transitions, what we are missing is a generalized "observation application" operator $\odot$, which would subsume both context filling and function application. Define it as follows.

$$\begin{aligned}\langle E \parallel \alpha\rangle \odot \mathtt{ret}(Z) &\coloneqq \langle E[Z] \parallel \alpha\rangle\\ V \odot \mathtt{call}(Z, \alpha) &\coloneqq \langle VZ \parallel \alpha\rangle\end{aligned}$$

The passive transition can now be recovered quite simply, and we reproduce the whole transition system one last time in its final form.

$$\begin{array}{lcll}\mathtt{act}(\langle P \parallel \alpha\rangle, \gamma) & \xrightarrow{x \cdot O} & \mathtt{pas}(\gamma \uplus \delta) & \begin{array}{l}\text{when } P \rightsquigarrow^* N \text{ and}\\ (x \cdot O, \delta) \coloneqq \text{split } \langle N \parallel \alpha\rangle\end{array}\\ \mathtt{pas}(\gamma) & \xrightarrow{x \cdot O} & \mathtt{act}(X \odot O, \gamma) & \text{when } (x \mapsto X) \in \gamma\end{array}$$

The path to generalize the Ogs construction to other languages is now ready: we will abstract over the notions of named term, generalized values, observations, normal form splitting and observation application. Notice how both the function call with explicit continuation as well as the named term construction $\langle P \parallel \alpha\rangle$ are reminiscent of abstract machines based presentations of a language's operational semantics. In this sense, in our personal opinion, Ogs is first and foremost a construction on an abstract machine, and not on a "programming language". This point of view will guide us during the axiomatization, as we will indeed entirely forget about bare terms and evaluation contexts, keeping only *configurations* (e.g. named terms) and generalized values. On top of streamlining the Ogs, letting go of contexts will also greatly simplify the necessary syntactic metatheory, as these "programs with a hole" are perhaps *fun*, but certainly not *easy* [1][67][45]. Yet as we have just seen, this does not preclude to treating languages given by more traditional small- or big-step operational semantics.

As it happens, this new game with explicit return pointers is common in Ogs constructions for languages with first-class continuations [58][48]. However, we stress that even for languages without such control operators, as in our λ-calculus, it is an important tool to streamline the system.

[1] Michael G. Abbott, Thorsten Altenkirch, Neil Ghani, and Conor McBride, "Derivatives of Containers," 2003.

[67] Conor McBride, "Clowns to the Left of me, Jokers to the Right," 2008.

[45] Tom Hirschowitz and Ambroise Lafont, "A unified treatment of structural definitions on syntax for capture-avoiding substitution, context application, named substitution, partial differentiation, and so on," 2022.

[58] Søren B. Lassen and Paul Blain Levy, "Typed Normal Form Bisimulation," 2007.

[48] Guilhem Jaber and Andrzej S. Murawski, "Compositional relational reasoning via operational game semantics," 2021.

Let us now expose how this thesis will unfold.

1.4 Contributions

The broad goal of this thesis is to formally implement the Ogs model in type theory, and prove it correct w.r.t. observational equivalence. We do not do so for a particular concrete language, but instead formalize it on top of an axiomatization of pure, simply-typed programming languages, for which we provide several instances. This result, as well as most other constructions in this thesis have been entirely proven in Rocq.

Games and Strategies in Type Theory In order to represent dialogue games and game strategies in type theory, we start off in Ch. 2 by reviewing a notion of game by Levy and Staton [62]. We then generalize upon the construction of *interaction trees* [93] and introduce *indexed interaction trees*, to represent game strategies as finely typed coinductive automata. In order to reason on such strategies, we show powerful reasoning principles for their strong and weak bisimilarity, inside a framework for coinduction based on complete lattices [81][86]. Further, we introduce a new notion of *eventually guarded* systems of recursive equations on indexed interaction trees, which we prove to have existence and uniqueness of fixed points w.r.t. strong bisimilarity.

[62] Paul Blain Levy and Sam Staton, "Transition systems over games," 2014.

[93] Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic, "Interaction trees: representing recursive and impure programs in Coq," 2020.

[81] Damien Pous, "Coinduction All the Way Up," 2016.

[86] Steven Schäfer and Gert Smolka, "Tower Induction and Up-to Techniques for CCS with Fixed Points," 2017.

Theory of Substitution In Ch. 3, we lightly review the standard tools for modeling intrinsically typed and scoped syntaxes with substitution [37][13]. To fit our needs, we present the lesser known notion of *substitution module* [44] over a substitution monoid, generalizing upon the theory of renaming. We further introduce a novel notion of *scope structures*, generalizing upon the traditional DeBruijn indices. Although relatively superficial, scope structures provide us with much appreciated flexibility in choosing how variables should look like.

[37] Marcelo Fiore and Dmitrij Szamozvancev, "Formal metatheory of second-order abstract syntax," 2022.

[13] Guillaume Allais, Robert Atkey, James Chapman, Conor McBride, and James McKinna, "A type- and scope-safe universe of syntaxes with binding: their semantics and proofs," 2021.

[44] André Hirschowitz and Marco Maggesi, "Modules over monads and initial semantics," 2010.

Ogs Construction With all the necessary scaffolding in place, in Ch. 4 we define a generic Ogs game, parametrized only by a notion of *observation*, inspired by copatterns [3]. We then introduce *language machines*, axiomatizing languages with open evaluators and derive from them a strategy for the Ogs game, constructing the Ogs model. We then state and discuss the hypotheses for correctness of equivalence in the model w.r.t. a variant of observational equivalence and state the correctness theorem. A notable finding is the appearance of a hypothesis which was never isolated in previous Ogs correctness proofs, as for most concrete languages it is a simple annoyance to deal with. The language-generic setting, however, makes the requirement and its reason clear. Our variant of observational equivalence, *substitution equivalence*, is an analogue of Ciu equivalence [66] tailored to our axiomatization of language machines.

[3] Andreas Abel, Brigitte Pientka, David Thibodeau, and Anton Setzer, "Copatterns: programming infinite structures by observations," 2013.

[66] Ian A. Mason and Carolyn L. Talcott, "Equivalence in Functional Languages with Effects," 1991.

Ogs Correctness We prove the correctness theorem in Ch. 5. After a standard detour on game strategy composition, we provide a novel decomposition of the correctness proof into two main high-level components, isolating the purely technical argument from the rest. First, a core semantical argument, showing essentially that substitution is a fixed point of the recursive equations defining composition. This part is relatively straightforward as it does not involve coinduction but only a series a basic rewriting steps. Second, we show that the composition equation is eventually guarded. This second part is the most technical, but provides an isolated justification for concluding by uniqueness of fixed points.

Normal Form Bisimulations Normal form (Nf) bisimulations are a notion of program equivalence closely related to the Ogs model. Their game is much more restricted, so that it is typically easier to prove two concrete programs normal form bisimilar than Ogs model equivalent. In Ch. 6, we construct the Nf game and the Nf model parametrized, as Ogs, by observations and a language machine. As a first application of the Ogs correctness theorem, we prove that Nf bisimilarity is correct by showing that the Ogs interpretation factors through the Nf interpretation.

Language Machine Instances To demonstrate the viability of our axiomatization of language machines, in Ch. 7 we provide three examples forming a cross section of what can be expressed. For each of them, we define a language machine and prove the correctness theorem hypotheses. First, we study Jump-with-Argument, a minimalistic continuation passing style calculus [60]. Then, we study $\mu\tilde{\mu}$-calculus with recursive types [30][33], a very expressive language with explicit control flow. Similarly to Call-by-Push-Value [60], it has been exhibited as a fine compilation target, so that this instance also implicitly captures all the calculi which can be compiled to $\mu\tilde{\mu}$-calculus. Finally, we study a more traditional calculus, untyped λ-calculus under weak head reduction. In this case, it is known that Nf bisimulation specializes to Levy-Longo tree equivalence [55], thus providing a new proof of their soundness w.r.t. observational equivalence.

[60] Paul Blain Levy, *Call-By-Push-Value: A Functional/Imperative Synthesis*, 2004.

[30] Pierre-Louis Curien and Hugo Herbelin, "The duality of computation," 2000.

[33] Paul Downen and Zena M. Ariola, "Compiling With Classical Connectives," 2020.

[55] Søren B. Lassen, "Bisimulation in Untyped Lambda Calculus: Böhm Trees and Bisimulation up to Context," 1999.

Rocq Artifact The source repository of the resulting code artifact is hosted at https://github.com/lapin0t/ogs. As part of a publication of the main results of this thesis, it has been archived as `doi:10.5281/zenodo.14697618`. This publication "An Abstract, Certified Account of Operational Game Semantics", to appear in Esop'25, is co-authored by my PhD advisors Tom Hirschowitz, Guilhem Jaber and Yannick Zakowski, and covers most of the material from Ch. 2, Ch. 4 and Ch. 5, although with far less details in the various proofs.

1.5 Metatheory

Before sailing off, there is one last thing we need to do: review the metatheory in which we will work. As we said earlier, our concrete code artifact is written using the Rocq proof assistant. However, in order to make this thesis accessible to a wider community, we have chosen to keep the present text self-contained and without any Rocq snippet. Our construction are written quite explicitly in a dependently typed programming style, but using an idealized type theory. From this point on, we will assume a good understanding of dependent type theory in general, and familiarity at least with *reading* code in either Agda, Rocq or some other type theory (Lean, Idris, ...).

This type theory can be quite succinctly described as an idealized Rocq kernel with an Agda syntax. More explicitly, it is an *intensional* type theory, with a predicative hierarchy of types Type_i, an impredicative universe of propositions Prop and *strict** propositions SProp. We rely on typical ambiguity and cumulativity to entirely disregard the universe levels of types. We moreover assume propositional unicity of identity proofs in the form of Streicher's axiom K for pattern matching [26], as well as definitional η-law on records and functions (in particular for the empty record type $\mathbf{1}$). We further assume the ability to define inductive data types and coinductive record types.

* Our use of strict propositions is anecdotic, so do not worry if your favorite proof assistant does not support it. On the other hand, having an impredicative sort is crucial for our lattice theoretic treatment of coinduction.

[26] Jesper Cockx, "Dependent Pattern Matching and Proof-Relevant Unification," 2017.

More superficially, we adopt Agda like syntax. Let us go through all the constructions. Keywords are highlighted in orange, definitions in blue, data type constructors in green, record projections in pink and identifier and some primitive type formers in black.

Function Types Given $A : \mathrm{Type}$ and $B : A \to \mathrm{Type}$, the dependent function type is written $(a : A) \to B\ a$, or possibly $\forall\ a \to B\ a$, when A can be inferred from the context. We additionally make use of implicit argument, written in braces like $\{a : A\} \to B\ a$ or $\forall\ \{a\} \to B\ a$. We adopt the Rocq convention of writing some argument binders left of the typing colon, simply separated by spaces. For example, we may declare the polymorphic identity function as $\mathrm{id}\ \{A\} : A \to A$, instead of $\mathrm{id} : \forall\ \{A\} \to A \to A$. When readability is at stake, we will even entirely drop implicit binders, but we will try to use this sparingly. Any dangling seemingly free identifier should be considered implicitly universally quantified.

Remark 1.2: For Rocq programmers confused by the Agda-like $\forall$ symbol: just parse the rest of the type as you do in Rocq when left of the typing colon. Any appearance of $\to$ switches back the rest to the usual parsing.

As we will use type families quite heavily, we introduce the notation $\mathrm{Type}^{X_1,\dots,X_n}$ to denote $X_1 \to \dots \to X_n \to \mathrm{Type}$.

(Inductive) Data Types Data types identifier are declared preceded by the keyword data and we give their constructors as inference rules. When the rules are self-referential, the type is always an inductive type. For extremely simple finite types, such as booleans, or the empty type, we write the following.

```
data bool : Type := true | false        data 0 : Type :=
```

We declare so called *mixfix* identifiers with the symbol ␣ as a placeholder for arguments in the identifier. As such, the disjoint sum $A + B$ is declared as data ␣+␣ : Type → Type → Type. Its constructors are given as follows.

$$\frac{a : A}{\mathsf{inl}\ a : A + B} \qquad \frac{b : B}{\mathsf{inr}\ b : A + B}$$

To avoid heavy notations, we may sometimes simply write +, or (+), when referring to infix combinators such the disjoint sum which should normally be identified by ␣+␣. The propositional equality type is declared as data ␣=␣ {A} : A → A → Prop, with the following constructor.

$$\frac{}{\mathsf{refl} : x = x}$$

Pattern matching functions are written by listing their *clauses*. Pattern matching is dependent and we do not write absurd clauses, as the inr case in foo below.

```
foo {A} : A + 0 → A
foo (inl a) := a
```

```
swap {A B} : A + B → B + A
swap (inl a) := inr a
swap (inr b) := inl b
```

Sometimes, we use the case keyword, to pattern match on an expression. For example, we could have alternatively defined swap as follows.

```
swap {A B} : A + B → B + A
swap x := case x
[ inl a := inr a
| inr b := inl b
```

The leftist programmer [71] unsettled by this case construction should note that we use it very sparingly. In fact, we will only use it in head position, as a with construction in disguise.

[71] Conor McBride and James McKinna, "The view from the left," 2004.

(Coinductive) Record Types Record types are introduced by the keyword record and by listing the type of their projections. When the declaration is self-referential, record types are always coinductive. The sigma type is technically declared as below, but we write it $(a : A) \times B\ a$, mimicking the dependent function type.

```
record sigma (A : Type) (B : A → Type) : Type :=
[ fst : A
, snd : B fst
```

We write projections in postfix notation and define record elements by so-called record expressions, giving the value of each projection, such as follows.

```
flip {A B} : A × B → B × A
flip p := [ fst := p.snd
          , snd := p.fst
```

We sometimes introduce constructors for record elements. In particular, for the sigma type we define the constructor $\_,\_$ allowing us to write pairs as the usual (a, b), and destruct them by pattern matching. We additionally define the unit type as the empty record `record 1 : Type := []`, with the constructor $\star$.

Induction and Coinduction We have not described which kind of inductive or coinductive functions we should be allowed to write. This is quite a tricky subject, as we use self-referential definitions instead of eliminators (and coeliminators). As such we will here only mostly say that "it works like in Rocq". Slightly more precisely, we will allow ourselves mutually recursive definitions with calls on structurally smaller arguments, and similarly only corecursive calls below record projections.

Typeclasses To structure our development, we will make use of typeclasses, which are simply records introduced by the keyword class and whose projections are written free standing, i.e., with the record element left implicit. We use the keyword extends, to denote the fact that a given typeclass declaration depends an instance of a previously declared one. As an example, we could define magmas and unital magmas as follows.

```
class Magma (X : Type) :=
[ _•_ : X → X → X
```

```
class UnitalMagma (X : Type) :=
[ extends Magma X
, id : X
, id-• {x} : id • x = x
, •-id {x} : x • id = x
```

Extensional Equality Although we pride ourselves in being very precise and explicit in all of our constructions, there will be a small technical informality (see Ch. 8 for a mea culpa). Because we work with coinductive objects in intensional type theory, propositional equality is too strong for several statements. As such, we use *extensional* equality in several places, written $a \approx b$. The problem is that the definition of extensional equality depends on the type we are considering, so that this should be essentially considered as a kind of typeclass giving the

extensional equality equivalence relation at any given type. Note that this does not mean that we use any extensionality axiom, only that we use a slightly sloppy overloading of the $\approx$ notation, as it cannot easily be given a single definition. We mostly make use of it at function types (denoting pointwise equality) and coinductive types (denoting strong bisimilarity) as well as on compound structures containing such objects. In any case, we try to be as explicit as possible around its use.

2 Coinductive Game Strategies

As we have seen, operational game semantics, and more generally interactive semantics, rest upon a dialogue following particular rules, a so-called two player game. The main task in this chapter is to properly define what is meant by "game", "strategy", and "transition system", and to provide basic building blocks for manipulating them. This chapter thus takes a step back and temporarily puts on hold our concerns about programming language semantics, in order to introduce the tools required to concretely represent games and strategies in type theory. These tools are in part novel, but consist mostly of natural extensions of preexisting devices.

2.1 A Matter of Computation

At heart, a strategy for a given two player game is an automaton of some kind, in the loose sense that it has some internal states tracking information required to choose moves, and alternates between two kinds of transitions. Whenever it is its turn, i.e., in an active state, a strategy must choose a move to play and transition to a passive state. And in a passive state, the strategy must accept any possible move made by a hypothetical opponent and transition to an active state.

In the classical literature on automata, these transitions would typically be represented by a *relation* between input states, moves and output states. On the other hand, in game semantics, the traditional approach is more extensional. There, a strategy is represented by a subset of traces (finite or infinite sequences of moves), i.e., by a formal language, subject to additional conditions. While perfectly valid in a classical logic or set-theoretic metatheory, when translated directly to type theory, both of these representations eschew the computational content of strategies.

Our basis for an idiomatic type theoretical encoding of automata follows the notion of *interaction tree* introduced by Xia et al. [93], originally motivated by representing executable denotational semantics of programs with general recursion. Interaction trees are a coinductive data structure encoding possibly non-terminating computations, interacting with their environment by means of uninterpreted events. Recognizing "programs" as Player strategies, "environments" as yet unknown Opponent strategies and "uninterpreted events" as move exchanges, we are quite close to our setting of alternating two player games.

[93] Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic, "Interaction trees: representing recursive and impure programs in Coq," 2020.

However, there are two remaining obstacles in order to apply interaction trees to our use case.

- **Duality.** We would like strategies and counter-strategies to have similar representations, intuitively simply by swapping the sets of moves allowed for each player. This is not directly possible with interaction trees, as the two kinds of moves do not have the same status. In interaction trees, the events are organized into a set of *queries* $Q : \mathrm{Type}$, and for each query a set of *responses* $R : Q \to \mathrm{Type}$. As such one cannot just swap queries and responses as they are not of the same sort.
- **Indexing.** In an interaction tree, while the set of allowed responses depends on the previous query, queries themselves do not depend on anything. As such, all queries are allowed at any point where it is Player's turn to play. In the context of two player games, this is a strong restriction on expressivity, which forbids us to represent games where some Player moves are allowed at certain points of the game but not at others, depending on what has been played before.

Luckily, both of these points can be resolved by swapping the notion of event from interaction trees, with the notion of game introduced by Levy & Staton [62]. The rest of the chapter is organized as follows.

[62] Paul Blain Levy and Sam Staton, "Transition systems over games," 2014.

- In §2.2, we reconstruct Levy & Staton's notion of game and of coalgebraic transition system.
- In §2.3, we introduce *indexed interaction trees*, a novel generalization of interaction trees adapted to the above notion of games.
- In §2.4, we define their bisimilarity together with powerful reasoning principles based on a lattice-theoretic fixed point construction.
- In §2.5, we give a little bit of structure to indexed interaction tree, mostly lifted from the non-indexed setting.
- In §2.6 we develop upon the theory of iteration operators, providing a novel *eventually guarded iteration*, applicable to indexed interaction trees but also to the delay monad [23] and to non-indexed interaction trees.

[23] Venanzio Capretta, "General recursion via coinductive types," 2005.

2.2 Levy & Staton Games

2.2.1 An Intuitive Reconstruction

[62] Paul Blain Levy and Sam Staton, "Transition systems over games," 2014.

The definition of game obtained by Levy & Staton in [62] arises quite naturally from what is intuitively understood by a "game". Let's build it up first hand.

In the common sense of the word, a game is described by the moves allowed at any point of the play, together with winning conditions and their associated rewards. As we are here only interested in games insofar as they provide a frame-

work for structured interactions, usual notions from classical game theory such as "winning", "reward" or "optimal play" will be completely absent. Moreover, we will restrict our attention to games where two agents play in alternating turns. Thus, for our purpose, games will just consist of the description of allowed moves.

Games in such a restricted view—two-player, alternating, no notion of winning—are similar to combinatorial games and might perhaps be more appropriately named *protocols*, as typically arises in the field of computer networks.

Starting from the idea that a game is described by the moves allowed for each player, arguably the simplest formalization is to say that a game consists of a pair (M^+, M^-) of sets, encoding the set of moves allowed for each player. For example, taking M^+ and M^- to be both equal to the set of UTF-8 character strings, we can think of this as the "game of chatting" where the two players are to alternatively exchange text messages. This definition readily encodes simple kinds of interactions: at a coarse level we could argue that a lot of low-level protocols consist in two players alternatively exchanging byte sequences. However, games-as-set-pairs are very restrictive in the sense that any move from, say, M^+ is valid at any point where it is the first player's turn. Thus, games-as-set-pairs are missing a shared game state, a game *position*, something enabling the set of allowed moves to evolve over the course of the dialogue. In particular, our game of interest in OGS, makes use of such evolution of moves: since players introduce variables while playing, moves mentioning some variable x should only be allowed after x has been introduced.

Still, this definition has the advantage of being quite symmetric: swapping the two sets, we get an involution $(M^+, M^-) \mapsto (M^-, M^+)$ exchanging the roles of both players. There are two lessons to be learnt from this naive definition:

- A game should be described by a pair of two objects of the same sort, each describing what moves one player can do.
- For describing moves, mere sets can be a first approximation, but are a bit too coarse for our purpose.

Back to the drawing board, let's refine this notion of games-as-set-pairs. As we were missing game *positions*, on which moves could then depend, it is but natural to assume a set of such positions. More precisely, we will assume two sets of game positions I^+ and I^-, where it is respectively the first player and the second player's turn to play. Then, instead of describing moves by mere sets, we can describe them by two families $M^+ : I^+ \to \mathrm{Type}$ and $M^- : I^- \to \mathrm{Type}$, mapping to each position the set of currently allowed moves. Finally, we must describe how the position evolves after a move has been played. This can be encoded by two maps $\mathrm{next}^+ : \forall\ \{i^+\} \to M^+\ i^+ \to I^-$ and $\mathrm{next}^- : \forall\ \{i^-\} \to M^-\ i^- \to I^+$. This leads us to the following definitions.

Definition 2.1 (Half-Game):
Given $I, J : \mathrm{Type}$ a *half-game with input positions I and output positions J* is given by a record of the following type.

```
record HGame I J :=
  [ Move : I → Type
  | next {i} : Move i → J
```

Definition 2.2 (Game):
Given $I^+, I^- : \mathrm{Type}$ a *game with active positions* I^+ *and passive positions* I^- is given by a record of the following type.

This is called *discrete game* by Levy & Staton [62].

```
record Game I⁺ I⁻ :=
  [ client : HGame I⁺ I⁻
  | server : HGame I⁻ I⁺
```

2.2.2 Categorical Structure

In order to make (half-)games into a proper category, we will define their morphisms. As games are parametrized over sets of positions, game morphisms could be naturally defined as parametrized over position morphisms, in the displayed style of Ahrens and Lumsdaine [10]. Yet we will resist the urge to dive too deeply into the structure of games and leave most of it for further work to expose. Indeed, we will require none of it for our main goal of proving correctness of Ogs. Moreover, as already noted by Dagand and McBride [32] in the similar setting of indexed containers, describing the extremely rich structures at play requires advanced concepts, such as framed bicategories and two-sided fibrations.

[10] Benedikt Ahrens and Peter LeFanu Lumsdaine, "Displayed Categories," 2019.

[32] Pierre-Évariste Dagand and Conor McBride, "A Categorical Treatment of Ornaments," 2013, Sec. 1.3.

Definition 2.3 (Half-Game Simulation):
Given two half-games $A, B : \mathrm{HGame}\ I\ J$, a *half-game simulation from* A *to* B is given by a record of the following type.

```
record HSim {I J} (A B : HGame I J) :=
  [ hs-move {i} : A.Move i → B.Move i
  | hs-next {i} (m : A.Move i) : B.next (hs-move m) = A.next m
```

Definition 2.4 (Simulation):
Given two games $A, B : \mathrm{Game}\ I^+\ I^-$, a *game simulation from* A *to* B is given by a record of the following type.

```
record Sim {I⁺ I⁻} (A B : Game I⁺ I⁻) :=
  [ s-client : HSim A.client B.client
  | s-server : HSim B.server A.server
```

Remark 2.5: The identity and composition of half-game simulations are given as follows.

$$\mathrm{id}\ \{I\ J\}\ (A : \mathrm{HGame}\ I\ J) : \mathrm{HSim}\ A\ A$$

$$\mathrm{id} := \begin{bmatrix} \text{hs-move}\ m := m \\ \text{hs-next}\ m\ := \mathrm{refl} \end{bmatrix}$$

$$\_ \circ \_\ \{I\ J\}\ \{A\ B\ C : \mathrm{HGame}\ I\ J\} : \mathrm{HSim}\ B\ C \to \mathrm{HSim}\ A\ B \to \mathrm{HSim}\ A\ C$$

$$F \circ G := \begin{bmatrix} \text{hs-move}\ m := F.\text{hs-move}\ (G.\text{hs-move}\ m) \\ \text{hs-next}\ m\ := \ldots \end{bmatrix}$$

The identity and composition of game simulations are then easily derived.

After defining the proper extensional equality on simulations (namely pointwise equality of the hs-move projection), we could prove that the above structure on half-games verifies the laws of a category, and easily deduce the same fact on games. For the reasons explained above, we leave this sketching as it is.

Remark 2.6: HGame extends to a strict functor $\mathrm{Set}^{\mathrm{op}} \times \mathrm{Set} \to \mathrm{Cat}$ as witnessed by the following action on morphisms, which we write curried and in infix style.

$$\_ \gg \_ \ll \_ : (I_2 \to I_1) \to \mathrm{HGame}\ I_1\ J_1 \to (J_1 \to J_2) \to \mathrm{HGame}\ I_2\ J_2$$

$$f \gg A \ll g := \begin{bmatrix} \text{Move}\ i := A.\text{Move}\ (f\ i) \\ \text{next}\ m := g\ (A.\text{next}\ m) \end{bmatrix}$$

The identity and composition laws of this functor hold *definitionally*.

2.2.3 Example Games

Let us introduce a couple example games, to get a feel for their expressivity.

Dual Game Perhaps the simplest combinator on games that we can devise is *dualization*. This allows us to swap the roles of the client and the server. It is defined as follows.

$$\_^{\dagger}\ \{I^+\ I^-\} : \mathrm{Game}\ I^+\ I^- \to \mathrm{Game}\ I^-\ I^+$$

$$G^{\dagger} := \begin{bmatrix} \text{client} := G.\text{server} \\ \text{server} := G.\text{client} \end{bmatrix}$$

Remark that dualization is *strictly* involutive, i.e., $\_^{\dagger} \circ \_^{\dagger}$ is definitionally equal to the identity function.

Nim The game of Nim is typically played with matchsticks. A number of matchsticks are on the playing board, grouped in several *heaps*. The two players must in turn take at least one matchstick away from one heap. They might take

as many matchsticks as they wish, so as long as they all belong to the same heap. Implicitly, one looses when it is not possible to play, i.e., when there are no matchsticks left.

Let us first encode the Nim game with a single heap. It is a symmetric game, where both active and passive positions are given by a natural number n, the number of available matchsticks in the only heap. A move is then an natural number no smaller than one and no greater than n. Equivalently, we can say that it is $1 + i$, where i is any natural number strictly smaller than n. Conveniently, the encoding in type theory of naturals strictly smaller than a given one are a well-known inductive family, the canonical *finite sets* $\mathsf{data}\ \mathsf{Fin} : \mathbb{N} \to \mathsf{Type}$ with constructors given as follows.

$$\frac{}{\mathsf{ze}_f : \mathsf{Fin}\ (\mathsf{su}\ n)} \qquad \frac{i : \mathsf{Fin}\ n}{\mathsf{su}_f\ i : \mathsf{Fin}\ (\mathsf{su}\ n)}$$

Given $n : \mathbb{N}$ and $i : \mathsf{Fin}\ n$, the substraction $n - (1 + i)$ is easily expressed.

$$\begin{aligned}
&_-_f_ : (n : \mathbb{N}) \to \mathsf{Fin}\ n \to \mathbb{N} \\
&\mathsf{su}\ n -_f \mathsf{ze}_f \;\; := n \\
&\mathsf{su}\ n -_f \mathsf{su}_f\ i := n -_f i
\end{aligned}$$

We thus obtain the single-heap Nim game as follows.

$$\mathsf{Nim}_1^{\mathsf{half}} : \mathsf{HGame}\ \mathbb{N}\ \mathbb{N} \qquad \mathsf{Nim}_1^{\mathsf{half}} := \begin{bmatrix} \mathsf{Move}\ n & := \mathsf{Fin}\ n \\ \mathsf{next}\ \{n\}\ i & := n -_f i \end{bmatrix}$$

$$\mathsf{Nim}_1 : \mathsf{Game}\ \mathbb{N}\ \mathbb{N} \qquad \mathsf{Nim}_1 := \begin{bmatrix} \mathsf{client} := \mathsf{Nim}_1^{\mathsf{half}} \\ \mathsf{server} := \mathsf{Nim}_1^{\mathsf{half}} \end{bmatrix}$$

We could obtain the many-heap Nim game by a similar construction. We would define the positions as lists of natural numbers, the moves as first selecting a heap and then choosing a number of matchsticks to take away, etc. Let us seek a more structured approach. Intuitively, the multi-heap Nim game is just some fixed number of copies of the single-heap game played simultaneously, in *parallel*. In case of two copies, the game positions consist of pairs of single-heap Nim positions. A move is then defined as choosing either a single-heap move on the first position or on the second. Let us define this as a generic binary combinator on games.

$$\_+^{\mathsf{half}}\_\ \{I\ J\} : \mathsf{HGame}\ I\ I \to \mathsf{HGame}\ J\ J \to \mathsf{HGame}\ (I \times J)\ (I \times J)$$

$$A +^{\mathsf{half}} B := \begin{bmatrix} \mathsf{Move}\ (i, j) & := A.\mathsf{Move}\ i + B.\mathsf{Move}\ j \\ \mathsf{next}\ (i, j)\ (\mathsf{inl}\ m) & := (A.\mathsf{next}\ m, j) \\ \mathsf{next}\ (i, j)\ (\mathsf{inr}\ m) & := (i, B.\mathsf{next}\ m) \end{bmatrix}$$

$$\llcorner +^{\mathrm{G}} \lrcorner \ \{I\ J\} : \mathrm{Game}\ I\ I \to \mathrm{Game}\ J\ J \to \mathrm{Game}\ (I \times J)\ (I \times J)$$

$$A +^{\mathrm{G}} B := \begin{bmatrix} \mathrm{client} := A.\mathrm{client} +^{\mathrm{half}} B.\mathrm{client} \\ \mathrm{server} := A.\mathrm{server} +^{\mathrm{half}} B.\mathrm{server} \end{bmatrix}$$

A many-heap Nim game, say with three heaps can then be simply be given as the following sum.

$$\mathrm{Nim}_3 : \mathrm{Game}\ (\mathbb{N} \times \mathbb{N} \times \mathbb{N})\ (\mathbb{N} \times \mathbb{N} \times \mathbb{N})$$

$$\mathrm{Nim}_3 := \mathrm{Nim}_1 +^{\mathrm{G}} \mathrm{Nim}_1 +^{\mathrm{G}} \mathrm{Nim}_1$$

Remark 2.7: Note that in the above binary sum of games, we constrained the games to have only a single set of positions. Indeed, it is crucial in Nim that one can play two times in a row in the same heap, while the opponent played in an other one. This only makes sense when the active and passive positions are the same.

For the curious reader, in case the active and passive positions differ, we can devise the following "parallel" sum $⅋^{\mathrm{G}}$, where the server is restricted to respond in the game that the client chose.

$$\llcorner ⅋^{\mathrm{h}} \lrcorner : \mathrm{HGame}\ I_1\ J_1 \to \mathrm{HGame}\ I_2\ J_2 \to \mathrm{HGame}\ (I_1 \times I_2)\ ((J_1 \times I_2) + (I_1 \times J_2))$$

$$A ⅋^{\mathrm{h}} B := \begin{bmatrix} \mathrm{Move}\ (i_1, i_2) & := A.\mathrm{Move}\ i_1 + B.\mathrm{Move}\ i_2 \\ \mathrm{next}\ \{i_1, i_2\}\ (\mathrm{inl}\ m) & := \mathrm{inl}\ (A.\mathrm{next}\ m, i_2) \\ \mathrm{next}\ \{i_1, i_2\}\ (\mathrm{inr}\ m) & := \mathrm{inr}\ (i_1, B.\mathrm{next}\ m) \end{bmatrix}$$

$$\llcorner \otimes^{\mathrm{h}} \lrcorner : \mathrm{HGame}\ I_1\ J_1 \to \mathrm{HGame}\ I_2\ J_2 \to \mathrm{HGame}\ ((I_1 \times J_2) + (J_1 \times I_2))\ (J_1 \times J_2)$$

$$A \otimes^{\mathrm{h}} B := \begin{bmatrix} \mathrm{Move}\ (\mathrm{inl}\ (i_1, j_2)) & := A.\mathrm{Move}\ i_1 \\ \mathrm{Move}\ (\mathrm{inr}\ (j_1, i_2)) & := B.\mathrm{Move}\ i_2 \\ \mathrm{next}\ \{\mathrm{inl}\ (i_1, j_2)\}\ m & := (A.\mathrm{next}\ m, j_2) \\ \mathrm{next}\ \{\mathrm{inr}\ (j_1, i_2)\}\ m & := (j_1, B.\mathrm{next}\ m) \end{bmatrix}$$

$$\llcorner ⅋^{\mathrm{G}} \lrcorner \ \{I\ J\} : \mathrm{Game}\ I\ I \to \mathrm{Game}\ J\ J \to \mathrm{Game}\ (I_1 \times I_2)\ ((J_1 \times I_2) + (I_1 \times J_2))$$

$$A ⅋^{\mathrm{G}} B := \begin{bmatrix} \mathrm{client} := A.\mathrm{client} ⅋^{\mathrm{h}} B.\mathrm{client} \\ \mathrm{server} := A.\mathrm{server} \otimes^{\mathrm{h}} B.\mathrm{server} \end{bmatrix}$$

This game, or rather its De Morgan dual $A \otimes^{\mathrm{G}} B := \left(A^{\dagger} ⅋^{\mathrm{G}} B^{\dagger}\right)^{\dagger}$ has been used by Levy & Staton [62] in their study of game strategy composition.

[62] Paul Blain Levy and Sam Staton, "Transition systems over games," 2014.

Conway Games Conway games are an important tool in the study of *combinatorial games* [27] and may in fact be considered their prime definition. We sketch here how they are an instance of our notion. They admit the following exceedingly simple definition: a Conway game $G : \text{Conway}$ is given by two subsets of Conway games $G_L, G_R \subseteq \text{Conway}$. The left subset G_L is to be thought of as the set of games reachable by the left player in one move, and symmetrically for G_R. Depending on whether this self-referential definition is interpreted inductively or coinductively, one obtains respectively the usual finite Conway games, or their infinite variant, sometimes called *hypergame*. For more background, see Honsell & Lenisa [46], as well as Joyal [50].

[27] John H. Conway, *On Numbers and Games*, 1976.

[46] Furio Honsell and Marina Lenisa, "Conway games, algebraically and coalgebraically," 2011.

[50] André Joyal, "Remarques sur la théorie des jeux à deux personnes," 1977.

In order to translate this definition into type theory, the only question is how to represent subsets. The most familiar representation is the powerset construction, adopting the point of view of subsets as (proof-relevant) predicates:

$$\text{Pow} : \text{Type} \to \text{Type}$$
$$\text{Pow}\ X := X \to \text{Type}$$

However there is a different, more intensional one, viewing subsets as families*. In this view, a subset is given by a type which encodes its *domain*, or *support*, or *total space*, and by a decoding function from this domain to the original set.

* For a more in-depth discussion of the two notions of subsets in type theory, see [43] Peter Hancock and Pierre Hyvernat, "Programming interfaces and basic topology," 2006, pp. 194–198.

```
record Fam (X : Type) : Type :=
  [ supp : Type
  | index : supp → X
```

We can go back and forth between the two representations, but only at the cost of a bump in universe levels. As such, to avoid universe issues and keep the manipulation tractable, it is important to choose the side which is most practical for the task at hand. Because we want to easily manipulate *elements* of the two subsets G_L and G_R, i.e., in this context the actual left moves and right moves, it is best to have we have them readily available by adopting the second representation. More pragmatically, the following definition would not go through when using Pow, as it is not a *strictly positive* operator on types.

Definition 2.8 (Conway Game):
The set of potentially infinite *Conway games* is given by elements of the following coinductive record.

```
record Conway : Type :=
  [ left  : Fam Conway
  | right : Fam Conway
```

We can now give a Levy & Staton game of Conway games. As in a Conway game it is ambiguous whose turn it is to play, the sets of active and passive positions will be the same. Moreover, the current position is in fact given by the current Conway game.

Example 2.9 (Game of Conway Games):
We start by noticing that $I \to \mathrm{Fam}\ J$ is just a shuffling of $\mathrm{HGame}\ I\ J$:

$$\mathrm{fam\text{-}to\text{-}hg}\ \{I\ J\} : (I \to \mathrm{Fam}\ J) \to \mathrm{HGame}\ I\ J$$

$$\mathrm{fam\text{-}to\text{-}hg}\ F := \begin{bmatrix} \mathrm{Move}\ i & := (F\ i).\mathrm{supp} \\ \mathrm{next}\ \{i\}\ m & := (F\ i).\mathrm{index}\ m \end{bmatrix}$$

Then, the *game of Conway games* can be given as follows.

$$\text{G-Conway} : \mathrm{Game}\ \mathrm{Conway}\ \mathrm{Conway}$$

$$\text{G-Conway} := \begin{bmatrix} \mathrm{client} := \mathrm{fam\text{-}to\text{-}hg}\ \mathrm{left} \\ \mathrm{server} := \mathrm{fam\text{-}to\text{-}hg}\ \mathrm{right} \end{bmatrix}$$

To make this example more solid, we should relate the notion of strategy in the sense of Conway to the strategies on G-Conway in the sense of Levy & Staton. But let's not get ahead of ourselves, as the latter are only introduced in the next section. We will leave this little sketch as it is.

2.2.4 Strategies as Transition Systems

Following Levy & Staton [62], we now define client strategies as transition systems over a given game. We will only define *client* strategies, since *server* strategies can be simply derived from client strategies on the dual game—the prime benefit of our symmetric notion of game. We first need to define two interpretations of half-games as functors.

Definition 2.10 (Half-Game Functors):
Given a half-game $G : \mathrm{HGame}\ I\ J$, we define the *active interpretation* and *passive interpretation of* G as functors $\mathrm{Type}^J \to \mathrm{Type}^I$, written $G \circledast \_$ and $G \Rrightarrow \_$ and defined as follows.

$$(G \circledast X)\ i := (m : G.\mathrm{Move}\ i) \times X\ (G.\mathrm{next}\ m)$$

$$(G \Rrightarrow X)\ i := (m : G.\mathrm{Move}\ i) \to X\ (G.\mathrm{next}\ m)$$

Definition 2.11 (Transition System):
Given a game $G : \mathrm{Game}\ I^+\ I^-$ and a family $X : \mathrm{Type}^{I^+}$, a *transition system over* G *with output* X is given by records of the following type.

In Levy & Staton [62], the output parameter X is not present and this is called a *small-step system over* G. We can recover their definition by setting $X\ i := \bot$.

```
record System_G X :=
  [ state⁺ : I⁺ → Type
  | state⁻ : I⁻ → Type
  | play : state⁺ ↛ X + state⁺ + G.client ⊛ state⁻
  | coplay : state⁻ ↛ G.server ⇛ state⁺
```

Remark 2.12: In the above, $X \nrightarrow Y := \forall\ \{i\} \to X\ i \to Y\ i$ denotes a morphism of families. Moreover, we have slightly abused the $+$ notation, silently using its pointwise lifting to families $(X + Y)\ i := X\ i + Y\ i$.

More generally, given n-ary families $X, Y : \mathrm{Type}^{I_1,\dots,I_n}$, the notation $X \nrightarrow Y$ will mean the following implicitly n-ary indexed function space.

$$\forall\ \{i_1\ ..\ i_n\} \to X\ i_1\ ..\ i_n \to Y\ i_1\ ..\ i_n$$

We will regularly implicitly lift constructions pointwise to families, although we try to say it every time we encounter a new one.

There is a lot to unpack here. First the states: instead of a mere set, as is usual in a classical transition system, they here consist of two families state^+, state^- *over* respectively the active and passive game positions. It is important not to confuse *positions* and *states*. The former consists of the information used to determine which moves are allowed to be played. The latter consists of the information used by a given strategy to determine how to play. Their relationship is similar to that of types to terms.

The play function takes as inputs an active position $i : I^+$, an active state $s : \mathrm{state}^+\ i$ over i and returns one of three things:

$X\ i$ *"return move"*

This case was not present in Levy & Staton [62], but it allows a strategy to end the game, provided it exhibits an output. As we will see with interaction trees in §2.3, this allows us to equip transition systems with a monad structure, an important tool for compositional manipulation.

$\mathrm{state}^+\ i$ *"silent move"*

In this case, the strategy postpones progression in the game. This case allows for strategies to be *partial* in the same sense as Capretta's Delay monad [23]. *Total strategies* without this case would make perfect sense, but we are interested in arbitrary, hence partial, computations.

[23] Venanzio Capretta, "General recursion via coinductive types," 2005.

$(G.\mathrm{client} \circledast \mathrm{state}^-)\ i$ *"client move"*

By Definition 2.10, this data consists of a client move valid at the current position i

$m : G.\text{client.Move}\ i,$

together with a passive state over the next position

$x : \text{state}^{-}\ (G.\text{client.next}\ m).$

This case is the one which actually *chooses* a move and sends it to a hypothetical opponent.

The coplay function is simpler. By Definition 2.10, it takes a passive position, a passive state over it, and a currently valid *"server move"*, and must then return an active state over the next position.

Remark 2.13: It might seem as if the hypothetical opponent must be pure, as return and silent moves appear in play but not in coplay, but this is not the case. Recall that we are working with an alternating game. The intent is that the transition system specifies the behavior of a strategy when the client is in control of the game. When a hypothetical opponent plays a return move or silent move, they do not give the control back to the client. As such the client does not have anything to do in these cases, and is in fact unaware of these kinds of moves played by the server.

2.3 Strategies as Indexed Interaction Trees

In Definition 2.11, we have defined strategies similarly to Levy and Staton [62], that is, by a state-and-transition-function presentation of automata. This representation is theoretically satisfying, however most of the time it is painful to work with formally. As an example, let's assume we want to define a binary combinator, taking two transition systems as arguments and returning a third one. Each of the two inputs is a dependent record with four fields, so that we have to work with eight input components to define the resulting transition systems, itself consisting of two families of states, and then, depending on these new states, two suitable transition functions. This is a lot to manage!

[62] Paul Blain Levy and Sam Staton, "Transition systems over games," 2014.

This unwieldiness is well known: while useful, writing state-machine-like code explicitly is closely linked to the dreaded *spaghetti code* and *callback hell*. It is perhaps the prime reason why widely used programming languages have started organizing it more soundly with syntactic facilities like the `yield` keyword of python's *generators* or the `await` keyword for sequencing asynchronous *promises* (or *awaitables*), now common in event-driven programming. Both of these concepts are automata in disguise. Their associated syntactic constructs allow one to write automata featuring bespoke state transitions (producing a sequence element, sleeping in wait of a network response) as if they were normal code.

For enlightening background on Python's generator syntax, see for example the Motivation section of the PEP 255.

As the precise definition of the state space is left implicit and for the language implementation to work out, it can no longer be manipulated by the programmer. What is left is an *opaque* notion of automaton (e.g., generators or awaitable objects), for which the only possible operation is *stepping*, i.e., running until the next transition.

These syntactic features have two benefits. First, we can program automata mostly as we do for normal functions, only sprinkling some automata-specific primitives where required. Second, automata are now black boxes, in a sense much like functions. This makes them quite a bit easier to pass around as their internal implementation details are tightly sealed away. In this section we will apply the same methodology to the definition of transition systems over games and this will bring us to our first contribution: *indexed interaction trees*.

2.3.1 From Games to Containers

Notice that given G and R, Definition 2.11 is exactly the definition of a coalgebra for the following endofunctor on $\mathrm{Type}^{I^+} \times \mathrm{Type}^{I^-}$.

$$(A^+, A^-) \mapsto \left(X + A^+ + G.\mathrm{client} \circledast A^- \;,\; G.\mathrm{server} \Rrightarrow A^+ \right)$$

Then, as by definition any coalgebra maps uniquely into the final coalgebra, it is sufficient to work with this final coalgebra, whose states intuitively consist of infinite coinductive trees, extensionally describing the traces of any possible transition system over G. This "universal" state space—the state space of the final coalgebra—will be our core notion of automata.

However, we will not construct this final coalgebra directly, but start by simplifying the setting slightly. Our motivation is the following. We insisted on having a clearly symmetric notion of game, in order to easily swap the point of view from client to server, a crucial concept in two player games. Strategies however, are inherently biased to one side. There is "our" side, by convention the side of the client, on which we *emit* moves, and the side of the server, on which we *receive* moves. Moreover, as we explained, a transition system only specifies what can happen when the client is in control of the game, i.e., in active positions. As such it is more annoying than anything else to have our constructions like the above endofunctor take place in a product category, forcing upon us *two* kinds of positions, *two* kinds of states and *two* kinds of transitions.

The trick to make the whole passive side disappear is to group in one atomic unit the act of sending and then receiving a move. In other words, we can consider a compound transition, where, starting from an active position, a strategy emits a move, waits for an opponent move, and attains a new active position. This exhibits strategies as coalgebras for the following functor.

$$A \mapsto X + A + G.\text{client} \circledast (G.\text{server} \Rrightarrow A)$$

We can apply this "fusing" trick not only to strategies but also to games. Quite satisfyingly, when forgetting about the passive positions, games will morph into the well-known notion of indexed polynomial functors, or more precisely their type-theoretic incarnation as *indexed containers* [15]. As such, our notion of strategy will not directly be parametrized by a game, but more generally by a biased representation derived from it: an indexed container. Let us introduce indexed containers and their relationship to games.

[15] Thorsten Altenkirch, Neil Ghani, Peter G. Hancock, Conor McBride, and Peter Morris, "Indexed containers," 2015.

Definition 2.14 (Indexed Container):
Given $I : \text{Type}$, an *indexed container with positions* I is given by records of the following type.

$$\begin{aligned}&\text{record Container } I : \text{Type} :=\\ &\left[\begin{array}{l}\text{Query} : I \to \text{Type}\\ \text{Reply } \{i\} : \text{Query } i \to \text{Type}\\ \text{next } \{i\}\ \{q : \text{Query } i\} : \text{Reply } q \to I\end{array}\right.\end{aligned}$$

Definition 2.15 (Game to Container):
There is a functor from games to containers defined on object as follows.

$$\begin{aligned}&\lfloor \_ \rfloor : \text{Game } I^+\ I^- \to \text{Container } I^+\\ &\lfloor A \rfloor := \left[\begin{array}{ll}\text{Query } i &:= A.\text{client.Move } i\\ \text{Reply } q &:= A.\text{server.Move } (A.\text{client.next } q)\\ \text{next } r &:= A.\text{server.next } r\end{array}\right.\end{aligned}$$

Like games, containers can be interpreted as functors.

Definition 2.16 (Extension of a Container):
Given an indexed container $\Sigma : \text{Container } I$, we define its *extension* $[\![\Sigma]\!] : \text{Type}^I \to \text{Type}^I$ as the following functor.

$$[\![\Sigma]\!]\ X\ i := (q : \Sigma.\text{Query } i) \times ((r : \Sigma.\text{Reply } q) \to X\ (\Sigma.\text{next } r))$$

Notice that the mapping from games to containers preserves the functor interpretation, in the sense that for all $A : \text{Game } I^+\ I^-$, the functor $A.\text{client} \circledast (A.\text{server} \Rrightarrow \_)$ is definitionally equal to $[\![\lfloor A \rfloor]\!]$. As such, our compound transition function can be recast as a coalgebra for the following functor.

$$A \mapsto X + A + [\![\lfloor G \rfloor]\!]\ A$$

Remark 2.17: As a matter of fact, as hinted at the beginning of this chapter, I went through this journey backwards. Starting from notions of strategies such as interaction trees [93] or interaction structures [43] where "games" are

[93] Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic, "Interaction trees: representing recursive and impure programs in Coq," 2020.
[43] Peter Hancock and Pierre Hyvernat, "Programming interfaces and basic topology," 2006.

understood as polynomial functors, I had trouble obtaining a sensible notion of dual game. With the realization that such polynomials can be "cut in halves", I arrived at Levy & Staton's more symmetric notion of game.

Remark 2.18: Although games include information about passive positions which containers do not, we can guess an over approximation of this information and embed containers into games as follows.

$$\lceil \_ \rceil : (\Sigma : \mathsf{Container}\ I) \to \mathsf{Game}\ I\ ((i : I) \times \Sigma.\mathsf{Query}\ i)$$

$$\lceil \Sigma \rceil := \begin{cases} \mathsf{client} := \begin{cases} \mathsf{Move}\ i & := \Sigma.\mathsf{Query}\ i \\ \mathsf{next}\ \{i\}\ m & := (i, m) \end{cases} \\ \mathsf{server} := \begin{cases} \mathsf{Move}\ (i,m) & := \Sigma.\mathsf{Reply}\ m \\ \mathsf{next}\ m & := \Sigma.\mathsf{next}\ m \end{cases} \end{cases}$$

We observe in passing that $\lfloor \_ \rfloor \circ \lceil \_ \rceil$ is *definitionally* equal to the identity function on containers.

This embedding suggests that although our symmetric notion of game is more *precise* than indexed containers, it is equally *expressive*. Indeed, we conjecture that $\lceil \_ \rceil$ and $\lfloor \_ \rfloor$ exhibit indexed containers as a reflective subcategory of games, but we have not introduced enough categorical structure to make this statement precise.

2.3.2 Indexed Interaction Trees

Now that have seen how indexed containers provide us with a biased, but more succinct description of games, we can temporarily forget about games to focus on indexed containers. Recall that we observed transition systems as coalgebras and that our goal was to define opaque *strategies* as elements of the final coalgebra. In our simplified setting, given a container Σ and an output family X, this amounts to constructing the following coinductive family, of *indexed interaction trees*.

$$\nu A.\ X + A + [\![\Sigma]\!]\ A$$

Our definition of this family proceeds in two steps. First we define the *action* functor

$$F\ X := X + A + [\![\Sigma]\!]\ A,$$

and only then, we form the coinductive fixed point νF. Although seemingly innocuous, this separation has its importance. Because the head type former of F is the disjoint union $+$, it would seem natural to implement this definition as a coinductive *data* type with three constructors. However, for metatheoretical reasons, it is largely proscribed to pattern-match on coinductive objects*. As such,

* Although Rocq allows it, it is folklore knowledge that this breaks subject reduction. See [68] Conor McBride, "Let's See How Things Unfold," 2009.

we first define the action functor as a data type, and then define its final coalgebra as a coinductive *record* type with a single projection.

Definition 2.19 (Action Functor):
Given a signature $\Sigma : \mathrm{Container}\ I$ and an output $X : \mathrm{Type}^I$, the *action on* Σ *with output* X, $\mathrm{data}\ \mathrm{Action}_\Sigma\ X : \mathrm{Type}^I \to \mathrm{Type}^I$ is given by the following constructors.

$$\frac{x : X\ i}{\text{"ret}\ r : \mathrm{Action}_\Sigma\ X\ A\ i} \qquad \frac{t : A\ i}{\text{"tau}\ t : \mathrm{Action}_\Sigma\ X\ A\ i}$$

$$\frac{q : \Sigma.\mathrm{Query}\ i \quad k : (r : \Sigma.\mathrm{Reply}\ q) \to A\ (\Sigma.\mathrm{next}\ r)}{\text{"vis}\ q\ k : \mathrm{Action}_\Sigma\ X\ A\ i}$$

Action's action on morphisms is without surprise, in fact it is functorial in both arguments. We give it by overloading the name Action_Σ.

$$\begin{aligned}
&\mathrm{Action}_\Sigma : (X_1 \to X_2) \to (A_1 \to A_2) \\
&\qquad \to \mathrm{Action}_\Sigma\ X_1\ A_1 \to \mathrm{Action}_\Sigma\ X_2\ A_2 \\
&\mathrm{Action}_\Sigma\ f\ g\ (\text{"ret}\ x) \ := \text{"ret}\ (f\ x) \\
&\mathrm{Action}_\Sigma\ f\ g\ (\text{"tau}\ t) \ := \text{"ret}\ (g\ t) \\
&\mathrm{Action}_\Sigma\ f\ g\ (\text{"vis}\ q\ k) := \text{"vis}\ q\ (\lambda\ r \mapsto g\ (k\ r))
\end{aligned}$$

Being itself an indexed polynomial functor, $\mathrm{Action}_\Sigma\ R$ has a thoroughly understood theory of fixed points [15] and we can form its final coalgebra as a coinductive family which is accepted by most proof assistants.

[15] Thorsten Altenkirch, Neil Ghani, Peter G. Hancock, Conor McBride, and Peter Morris, "Indexed containers," 2015.

Definition 2.20 (Indexed Interaction Tree):
Given a signature $\Sigma : \mathrm{Container}\ I$ and an output $X : \mathrm{Type}^I$, the family of *indexed interaction trees on* Σ *with output* X, denoted by $\mathrm{ITree}_\Sigma\ X : \mathrm{Type}^I$, is given by coinductive records of the following type.

$$\begin{aligned}
&\mathrm{record}\ \mathrm{ITree}_\Sigma\ X\ i : \mathrm{Type} := \\
&\ [\ \mathrm{out} : \mathrm{Action}_\Sigma\ X\ (\mathrm{ITree}_\Sigma\ X)\ i
\end{aligned}$$

Furthermore, define the following shorthands:

$$\begin{aligned}
\mathrm{ret}\ x \ &:= [\ \mathrm{out} := \text{"ret}\ x \\
\mathrm{tau}\ t \ &:= [\ \mathrm{out} := \text{"tau}\ t \\
\mathrm{vis}\ q\ k &:= [\ \mathrm{out} := \text{"vis}\ q\ k
\end{aligned}$$

Remark 2.21: Note that in accordance with Agda and Rocq, our type theory does not assume the η-law on coinductive records. As such, the above definition technically only constructs a *weakly* final coalgebra. Doing otherwise

would require switching to a slightly more extensional type theory, such as observational type theory [68].

The above definition is to interaction trees [93] what inductive families are to inductive data types, in other words, the same construction but taking place in the category Set^I instead of Set. As we will discover in the remainder of this chapter, all of the monadic theory of interaction trees lifts neatly to this newly indexed setting.

Before moving on to define bisimilarity and other useful structure, let us first link this definition to transition systems over games. First, as indexed interaction trees are parametrized over containers, let us start by defining *game* strategies.

Definition 2.22 (Strategies):
Given a game $G : \mathrm{Game}\ I^+\ I^-$ and output $X : \mathrm{Type}^{I^+}$, the active and passive *strategies over G with output X* are defined as follows.

$$\mathrm{Strat}^+{}_G\ X : I^+ \to \mathrm{Type} \qquad \mathrm{Strat}^-{}_G\ X : I^- \to \mathrm{Type}$$
$$\mathrm{Strat}^+{}_G\ X := \mathrm{ITree}_{\lfloor G \rfloor}\ X \qquad \mathrm{Strat}^-{}_G\ X := G.\mathrm{server} \Rrightarrow \mathrm{Strat}^+{}_G\ X$$

Definition 2.23 (Strategy System):
Given a game $G : \mathrm{Game}\ I^+\ I^-$ and output $X : \mathrm{Type}^{I^+}$, active and passive strategies are the states of a small step system over G with output X defined as follows.

$$\mathrm{Strat}_G\ X : \mathrm{System}_G\ X$$
$$\mathrm{Strat}_G\ X :=$$
$$\begin{bmatrix} \mathrm{state}^+ := \mathrm{Strat}^+{}_G\ X \\ \mathrm{state}^- := \mathrm{Strat}^-{}_G\ X \\ \mathrm{play}\ s := \mathrm{case}\ s.\mathrm{out} \\ \quad \begin{bmatrix} \text{`ret}\ x & := \mathrm{inj}_1\ x \\ \text{`tau}\ t & := \mathrm{inj}_2\ t \\ \text{`vis}\ q\ k & := \mathrm{inj}_3\ (q, k) \end{bmatrix} \\ \mathrm{coplay}\ k\ m := k\ m \end{bmatrix}$$

inj_1, inj_2 and inj_3 denote the obvious injections into the ternary disjoint union.

Definition 2.24 (System Unrolling):
For any system $S : \mathrm{System}_G\ X$, the active and passive states can be respectively *unrolled* to active and passive strategies. The mapping is given by the following two mutually coinductive definitions.

[68] Conor McBride, "Let's See How Things Unfold," 2009.
[93] Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic, "Interaction trees: representing recursive and impure programs in Coq," 2020.

```
unroll⁺ : S.state⁺ ↛ Strat⁺_G X
unroll⁺ s :=
  out := case S.play s
  ⌈ inj₁ x       := ˝ret x
  | inj₂ s       := ˝tau (unroll⁺ s)
  ⌊ inj₃ (m, s) := ˝vis m (unroll⁻ s)

unroll⁻ : S.state⁻ ↛ Strat⁻_G X
unroll⁻ s m := unroll⁺ (S.coplay s m)
```

Remark 2.25: The above unrolling functions can be shown to be the computational part of the unique coalgebra morphism between a transition system S and the strategy system $\mathrm{Strat}_G\ X$. We do not prove it formally, as it would require properly defining transition system morphisms and their extensional equivalence.

Remark 2.26: The above notion of strategy is quite different to the one given by Levy & Staton [62] (Def. 2). At a high level it serves a similar purpose, namely obtaining an opaque representation for transition systems, which forgets about implementation details. In line with the traditional set-theoretic presentation of game semantics, they define strategies as subsets of finite *traces*. These intuitively consist in the set of all the finite approximations of all the possible paths through a strategy tree (discounting silent steps). Technically, such as set must verify some conditions to be considered well-formed, mainly a prefix closure condition (to be considered a valid set of finite approximations), as well as a determinism condition on client moves. We provide a couple comments but leave a more formal comparison for future work.

[62] Paul Blain Levy and Sam Staton, "Transition systems over games," 2014.

By virtue of representing a strategy as a set of valid traces, their notion of strategy equivalence is a *trace equivalence*. Yet because the considered strategies are deterministic, bisimulation and trace equivalence are known to coincide. We conjecture that this can be shown in type theory, in other words, constructively, the final coalgebra of $\mathrm{Action}_\Sigma\ X$ *embeds* into strategies defined as some predicates on traces.

Although Levy & Staton do not prove it, we conjecture that in classical set theory, one should be able to show that our notion and theirs are isomorphic, in other words, that strategies as traces are indeed a final coalgebra. There is however no hope of such result in type theory without further axioms, for two reasons. First, the determinism condition restricts valid plays to be extended by a unique client move. We conjecture that deducing a constructive function computing the next move would amount to a form of unique choice principle. Second, partiality of strategies is handled simply by not requiring that every

finite valid trace in the strategy is extended by a client move. Instead, our representation uses concrete silent "tau steps which might be played indefinitely. We conjecture that going from the former to the latter would amount to a form of excluded middle.

All in all, our coinductive encoding of strategies as a final coalgebra can be seen as slightly more intensional than Levy & Staton's. We believe that it provides a more idiomatic computational account of strategies.

2.3.3 Delay and Big-Step Transition Systems

Before heading to definition of bisimilarity of strategies, we introduce one last notion, half-way between transition systems and strategies: *big-step transition systems*. It is sometimes more convenient to work with a transition system, but where the silent steps have been "grouped together". In other words, we wish to remove the silent steps from the transition *function*, and instead work with a *partial* transition function returning either an output step or a visible step.

To do so, the prime choice is to use Capretta's delay monad [23]. Recall that it is defined abstractly as the following final coalgebra.

[23] Venanzio Capretta, "General recursion via coinductive types," 2005.

$$\text{Delay}\ X := \nu A.\ X + A$$

Instead of directly constructing this coinductive type, it can be observed that it is readily an instance of our interaction trees. Defining it as an interaction tree means that all of the forthcoming theory on interaction trees will effortlessly be available on the delay monad.

Definition 2.27 (Delay Monad):
Define the trivially indexed *void container* as follows.

$$0_P : \text{Container}\ \mathbf{1}$$

$$0_P := \begin{bmatrix} \text{Query}\ i := \mathbf{0} \\ \text{Reply}\ i\ () \\ \text{next}\ \ i\ () \end{bmatrix}$$

Then, define the delay monad as follows.

$$\text{Delay} : \text{Type} \to \text{Type}$$

$$\text{Delay}\ X := \text{ITree}_{0_P\ \mathbf{1}}\ (\lambda\ i \mapsto X)\ \star$$

Remark 2.28: Because the delay monad is not indexed, it is given by a *trivially* indexed interaction tree. This choice is rather debatable in our concrete code artifact, as Rocq's inability to provide an η-law on the empty record $\mathbf{1}$ makes it sometimes painful to work with trivially indexed interaction trees. Indeed,

when exhibiting a trivially indexed family $(i : \mathbf{1}) \to X\ i$ for some given X, we have the choice of pattern matching, or not, on the given index i. It is usually best not to, so that the family is definitionally constant, but this unknown i can sometimes clash when a concrete $\star$ is expected. We gloss over these issues and assume the η-law on $\mathbf{1}$. This makes every element of $\mathbf{1}$ *definitionally* equal to $\star$, and more importantly, every function $\mathbf{1} \to X$ definitionally constant.

Definition 2.29 (Big-Step Transition System):
Given a game $G : \mathrm{Game}\ I^+\ I^-$ and a family $X : \mathrm{Type}^{I^+}$, a *big-step transition system over G with output X* is given by records of the following type.

$$
\begin{aligned}
&\mathrm{record}\ \text{Big-Step-System}_G\ X :=\\
&\quad \left[\begin{array}{ll}
\mathrm{state}^+ & : I^+ \to \mathrm{Type}\\
\mathrm{state}^- & : I^- \to \mathrm{Type}\\
\mathrm{play} & : \mathrm{state}^+ \to \mathrm{Delay}\ (X + G.\mathrm{client} \circledast \mathrm{state}^-)\\
\mathrm{coplay} & : \mathrm{state}^- \to G.\mathrm{server} \Rrightarrow \mathrm{state}^+
\end{array}\right.
\end{aligned}
$$

Note that we have implicitly lifted the delay monad pointwise to families by $\mathrm{Delay}\ X\ i := \mathrm{Delay}\ (X\ i)$.

Remark 2.30: Levy & Staton define a similar notion of big-step transition system [62] (Def. 4). The sole difference is that they model partiality using the $\mathrm{Option}\ X := \mathbf{1} + X$ monad. Once again, in a classical metatheory, this is isomorphic to our definition (when the Delay monad is quotiented by weak bisimilarity). Constructively however, Option is quite far from a suitable model of partiality in the sense of Turing-complete computation, as it trivially allows one to *compute* whether the "partial computation" is undefined or returns an output.

[62] Paul Blain Levy and Sam Staton, "Transition systems over games," 2014.

Like transition systems, big-step transition systems can be unrolled into strategies. This should not come as a surprise as a standard calculation on fixed points shows the following isomorphism.

$$
\begin{aligned}
& \nu A.\ X + A + G.\mathrm{client} \circledast G.\mathrm{server} \Rrightarrow A\\
\approx\ & \nu A.\ \nu B.\ (X + G.\mathrm{client} \circledast G.\mathrm{server} \Rrightarrow A) + B
\end{aligned}
$$

Definition 2.31 (Big-Step System Unrolling):
For any big-step system $S : \text{Big-Step-System}_G\ X$, the active and passive states can be respectively *unrolled* to active and passive strategies. The mapping is given by the following three mutually coinductive definitions.

```
unroll-aux : Delay (X + G.client ⊛ state⁻) → Strat⁺_G X
unroll-aux t :=
  [ out := case t.out
    | [ ᵀret (inl x)        := ᵀret x
      | ᵀret (inr (m , s)) := ᵀvis m (unroll⁻ s)
      | ᵀtau t              := ᵀtau (unroll-aux t)

unroll⁺ : S.state⁺ → Strat⁺_G X
unroll⁺ s := unroll-aux (S.play s)

unroll⁻ : S.state⁻ → Strat⁻_G X
unroll⁻ s m := unroll⁺ (S.coplay s m)
```

Note that we have overloaded unroll⁺ and unroll⁻ for both small- and big-step transition systems. In fact for the rest of this thesis we will be entirely concerned with either strategies or big-step transition systems. With all representation variants of strategies now defined, let us now define their notions of equivalence.

2.4 Bisimilarity

The natural notion of equality on automata is the notion of bisimilarity. Intuitively, a bisimulation between two automata consists of a relation between their respective states, which is preserved by the transition functions. Two automata are then said to be *bisimilar* when one can exhibit a bisimulation relation between them. Another way to phrase this is that two automata are bisimilar whenever they are related by the greatest bisimulation relation, *bisimilarity*, again a coinductive notion. As our strategies feature *silent moves* (the ᵀtau nodes of the action functor), we will need to consider two variants, *strong* and *weak* bisimilarity. Strong bisimilarity requires that both strategy trees match at each step, fully synchronized. Weak bisimilarity, on the other hand, allows both strategies to differ by a finite amount of ᵀtau nodes in between any two synchronization points.

Before translating these ideas into type theory, we will need a bit of preliminary tools. Most implementations of type theory provide some form of support for coinductive records (such as Definition 2.20) and for coinductive definitions (such as Definition 2.24). However, these features—in particular coinductive definitions—are at times brittle, because type theory typically relies on a syntactic *guardedness* criterion to decide whether a given definition should be accepted. For simple definitions—in fact more precisely for computationally relevant definitions—I will indulge the whims of syntactic guardedness. But for complex

bisimilarity proofs such as those which will appear later in this thesis, being at the mercy of a syntactic implementation detail is a recipe for failure.

To tackle this problem, Agda provides more robust capabilities in the form of *sized types*, for which the well-formedness criterion is based on typing. However they are not available in Rocq, the language in which this thesis has been formalized. Moreover, in Agda's experimental tradition, while sized types are of practical help when used as intended, their precise semantics are still not fully clear [2].

[2] Andreas Abel and Brigitte Pientka, "Well-founded recursion with copatterns and sized types," 2016.

We will take an entirely different route, building coinduction for ourselves, *inside* type theory. Indeed, as demonstrated by Pous's `coq-coinduction` software library [81] on which our artifact is based, powerful coinductive constructions and reasoning principles are derivable in the presence of an impredicative sort of propositions.

[81] Damien Pous, "Coinduction All the Way Up," 2016, https://github.com/damien-pous/coinduction.

2.4.1 Coinduction with Complete Lattices

The basis of `coq-coinduction` is the observation that with impredicativity, Prop forms a complete lattice ordered by implication. In fact, not only Prop, but also predicates $X \to \mathrm{Prop}$ or relations $X \to Y \to \mathrm{Prop}$, our case of interest for bisimilarity. By the Knaster-Tarski theorem [91] one can obtain the greatest fixed point $\nu f \coloneqq \bigvee \{x \mid x \lesssim f\ x\}$ of any monotone endo-map f on a complete lattice. Henceforth, we will use $\lesssim$ for the ordering relation of any lattice, and $\approx$ for the derived equivalence relation.

[91] Alfred Tarski, "A lattice-theoretical fixpoint theorem and its applications," 1955.

This is only the first part of the story. Indeed this will provide us with the greatest fixed point νf, in our case, bisimilarity, but the reasoning principles will be cumbersome. At first sight, the only principle available is the following one.

$$\frac{x \lesssim y \quad y \lesssim f\ y}{x \lesssim \nu f}$$

Programming solely with this principle is painful, much in the same way as manipulating inductive types solely using eliminators, instead of using pattern-matching and recursive functions. Thankfully, in the context of bisimulations, a line of work has been devoted to a theory of *enhanced* bisimulations, in which the premise is weakened to $x \lesssim f\ (g\ x)$— bisimulation *up-to* g—for some other monotone map g. We say that g is a *valid up-to principle* for f whenever this enhanced property is derivable.

$$\frac{x \lesssim y \quad y \lesssim f\ (g\ y)}{x \lesssim \nu f}$$

The map g will typically enlarge its argument y, or otherwise tweak it, making g-enhanced bisimulations y easier to exhibit than proper bisimulations. As an example on relations, reasoning up-to transitivity means working with such a principle for $g\ R := R; R$. Because valid up-to principles are not closed under composition, several well-behaved sufficient conditions have been studied, such as *compatibility* where g must verify $g \circ f \lesssim f \circ g$ w.r.t. the pointwise order on monotone maps. Satisfyingly, the least upper bound of all compatible maps is still compatible. It is called the *companion* [81] of f written t_f. This enables working with the following *up-to companion* generalized principle.

[81] Damien Pous, "Coinduction All the Way Up," 2016.

$$\frac{x \lesssim f\ (t_f\ x)}{x \lesssim \nu f}$$

Thanks to the fact that $x \lesssim t_f\ x$ and $t_f\ (t_f\ x) \lesssim t_f\ x$, one can delay until actually required in the proof the choice and use of any particular valid up-to principle $g \lesssim t_f$. This theory based on the companion is the one used in the Rocq formalization of this thesis. However, since I started writing the formalization, an even more practical solution, Schäfer and Smolka's *tower induction* [86], has been merged into `coq-coinduction`. I did not have the time to port my Rocq development to the new version of `coq-coinduction`, but I will nonetheless present it here and use tower induction as the basis for defining bisimilarity. We thus leave both the Knaster-Tarski construction as well as the companion behind and now focus solely on tower induction.

[86] Steven Schäfer and Gert Smolka, "Tower Induction and Up-to Techniques for CCS with Fixed Points," 2017.

Tower induction rests upon the inductive definition of the *tower predicate* Tower_f, whose elements can be understood as the transfinite iterations of f starting from the greatest element $\top$. In other words, Tower_f characterizes the transfinite approximants of the greatest fixed point of f. We will refer to these elements of the tower as *candidates*.

For more easily working with predicates, we introduce some notations. For any predicate $P : \mathrm{Prop}^X$ we write $x \in P$ instead of $P\ x$ and $\forall\ x \in P \to ..$ instead of $\forall\ \{x\} \to P\ x \to ...$ Moreover, given two predicates $P, Q : \mathrm{Prop}^X$, we write $P \subseteq Q$ to denote $\forall\ x \in P \to x \in Q$.

Definition 2.32 (Tower):
Given a complete lattice X and a monotone endo-map $f : X \to X$, the *f-Tower* is an inductive predicate $\mathrm{data}\ \mathrm{Tower}_f : \mathrm{Prop}^X$ defined by the following constructors.

$$\frac{t : x \in \mathrm{Tower}_f}{\text{T-step}\ t : f\ x \in \mathrm{Tower}_f} \qquad \frac{s : F \subseteq \mathrm{Tower}_f}{\text{T-inf}\ s : \bigwedge F \in \mathrm{Tower}_f}$$

Theorem 2.33 (Tower Induction):
Given a complete lattice X, a monotone endo-map $f : X \to X$ and an inf-closed predicate $P : \mathrm{Prop}^X$, the following *tower induction principle* is true.

$$\frac{\forall\, x \in \mathrm{Tower}_f \to x \in P \to f\, x \in P}{\mathrm{Tower}_f \subseteq P}\ \mathrm{tower}$$

Proof: Assuming that P is inf-closed i.e.,

$$K : \forall\, \{F\} \to F \subseteq P \to \bigwedge F \in P,$$

and assuming the hypothesis

$$H : \forall\, x \in \mathrm{Tower}_f \to x \in P \to f\, x \in P,$$

define $\mathrm{tower} : \mathrm{Tower}_f \subseteq P$ by induction as follows.

$$\mathrm{tower}\ \{x\} : \mathrm{Tower}_f\ x \to P\ x$$
$$\mathrm{tower}\ (\text{T-step}\ t) := H\ t\ (\mathrm{tower}\ t)$$
$$\mathrm{tower}\ (\text{T-inf}\ s) \ := K\ (\lambda\, p \mapsto \mathrm{tower}\ (s\ p))$$

■

Remark 2.34: Note that tower induction is only a small rewording of a non-dependent induction principle (called *minimality* in Rocq) on membership proofs of the tower predicate. The sole difference is that the minimality principle does not require full inf-closedness, but only inf-closedness for families below the tower, i.e., the weaker $F \subseteq \mathrm{Tower}_f \to F \subseteq P \to \bigwedge F \in P$. However, inf-closedness is quite a common property, so that it can be automatically derived in full for every properties of interest.

Lemma 2.35 (Tower Properties):
Given a complete lattice X and a monotone endo-map $f : X \to X$, for all candidate $x \in \mathrm{Tower}_f$ the following statements are true.

(1) $f\, x \lesssim x$
(2) $\forall\, y \to y \lesssim f\, y \to y \lesssim x$

Proof: Both statements are proven by direct tower induction on the statement, with simple calculations proving the hypotheses. Let us detail the first statement to showcase the proof method.

Pose $P\, x := f\, x \lesssim x$. Let us show the tower induction hypotheses.

- Assume $F \subseteq P$. Let us show that $\bigwedge F \in P$. By definition of the infimum, it suffices to show that for any $x \in F$ we have $f\ (\bigwedge F) \lesssim x$. By definition of the infimum and monotonicity $f\ (\bigwedge F) \lesssim f\, x$. Conclude using the assumption $F \subseteq P$.

- Assume $x \in \mathrm{Tower}_f$ such that $x \in P$, let us show that $f\,x \in P$. By assumption we know that $f\,x \lesssim x$, thus by monotonicity, $f\,(f\,x) \lesssim f\,x$, which concludes. ■

Theorem 2.36 (Greatest Fixed Point):
Given a complete lattice X and a monotone endo-map $f : X \to X$, pose

$$\nu f \coloneqq \bigwedge \mathrm{Tower}_f\,.$$

The following statements are true.
(1) $\nu f \in \mathrm{Tower}_f$
(2) $\nu f \approx f\,(\nu f)$
(3) $\forall\, x \to x \lesssim f\,x \to x \lesssim \nu f$

Proof:
(1) By T-inf ($\lambda\, t \mapsto t$).
(2) By antisymmetry.
- $\nu f \lesssim f\,(\nu f)$ By T-step and (1), we have $f\,(\nu f) \in \mathrm{Tower}_f$. Conclude by definition of ν as an infimum.
- $f\,(\nu f) \lesssim \nu f$ By (1) we and Lemma 2.35 (1).

(3) By (1) and Lemma 2.35 (2). ■

In Theorem 2.36, (2) and (3) are pretty clear: together they prove that νf is indeed the greatest fixed point of f. On the other hand, (1) might seem like a technical lemma but it is just as important, if not more. Indeed, knowing that νf is part of the tower—i.e., that it is itself a transfinite approximation of the greatest fixed point—enables to directly apply tower induction (Theorem 2.33) to prove properties about νf. Although tower induction also proves properties about any candidate (elements of the tower), it will be our prime reasoning principle for proving properties and the greatest fixed point. As we will see shortly, lemmas about arbitrary candidates (also proven by tower induction) will provide us with a practical alternative to more usual valid up-to principles.

And this is it! I really want to stress the fact that this is the entirety of the core mathematical content of this theory of coinduction, and yet it provides an exceedingly versatile and easy-to-use theorem. It is easily shown to subsume the principles provided by the companion construction and by other frameworks such as *parametrized coinduction* [47]. Despite its great utility, the `coq-coinduction` library is extremely small. Besides some applications and generic theorems, its only additional content consist in several helpers e.g., for deriving inf-closedness of predicates, the definition of the most useful instances of complete lattices and some generic duality and symmetry arguments. The article by Schäfer and Smolka [86] is similarly short, providing the whole theory in merely three pages, including the proofs and the derivation of the companion. As

[47] Chung-Kil Hur, Georg Neis, Derek Dreyer, and Viktor Vafeiadis, "The power of parameterization in coinductive proof," 2013.

[86] Steven Schäfer and Gert Smolka, "Tower Induction and Up-to Techniques for CCS with Fixed Points," 2017.

such, if you need to work with coinduction and are happy with an impredicative universe of propositions, I heartily recommend you to try out tower induction.

2.4.2 Strong Bisimilarity

Equipped with this new construction for greatest fixed points we are ready to define bisimilarity. As our strategies feature silent transitions, there are two variants of interest: *strong bisimilarity* which considers silent transitions as any other "normal" transition, and *weak bisimilarity*, which allows to skip over any finite number of silent steps. We start by describing the strong bisimilarity, which embodies the natural notion of extensional equality for strategies, but the construction of weak bisimilarity will proceed very similarly.

Bisimulations and simulations for coalgebraic presentations of automata have been well studied, with the general blueprint outlined by Levy [61]. Succinctly, bisimulations for F-coalgebras can be expressed given an analogue to the functor F, but operating on *relations* instead of sets. More precisely, this operator is called a *relator* [61][53] and takes relations $X \nrightarrow Y$ to relations $FX \nrightarrow FY$, subject to several conditions. Then, given a relator Γ and two coalgebras $\varphi : X \to FX$ and $\psi : Y \to FY$, we can derive a monotone endo-map which takes a relation $R : X \nrightarrow Y$ to the re-indexed relation $\varphi\rangle\Gamma R\langle\psi : X \nrightarrow Y$. This resulting relation can be understood in plain words: after one unfolding on both sides, the coalgebra states are related by R under Γ. In a sense, Γ decides when two steps should be considered matching or *synchronized*, provided we give it a relation on states. Bisimilarity is then obtained by the greatest fixed point of this monotone map, while bisimulations are its pre-fixed points.

[61] Paul Blain Levy, "Similarity Quotients as Final Coalgebras," 2011.

[53] Ugo Dal Lago, Francesco Gavazzo, and Paul Blain Levy, "Effectful applicative bisimilarity: Monads, relators, and Howe's method," 2017.

We will largely follow this blueprint, only concerned with the final coalgebras given by interaction trees, with perhaps the slight technicality that we are working not with sets and relations, but (indexed) set families and relation families. But to better fit our setting, we adopt a small twist. Recall that Action_Σ is functorial with respect to both the output parameter X as well as the second "coalgebra" parameter. Indeed we intend to show that strategies form a monad, so that in particular they are functorial with respect to this output X. Hence, our goal is not to construct a single relation between strategies, but rather devise an operator taking a relation family on two output families

$$X^r : \forall\ \{i\} \to X^1\ i \to X^2\ i \to \mathrm{Prop},$$

to the strong bisimilarity on strategies with the respective outputs

$$\llcorner \cong[\ X^r\ ]\ \lrcorner : \forall\ \{i\} \to \mathrm{ITree}_\Sigma\ X^1\ i \to \mathrm{ITree}_\Sigma\ X^2\ i \to \mathrm{Prop}\,.$$

This is quite reminiscent of a relator on ITree_Σ! And indeed, pushing the output parameter inside our monotone map will not fundamentally change the construction. Instead of working in the complete lattice of relation families, we will adopt the complete lattice of ITree_Σ relators.

Let us start with some preliminary notation for our indexed relations.

Definition 2.37 (Family Relation):
Given $I : \mathrm{Type}$ and two families $X, Y : \mathrm{Type}^I$, the type of *family relations between X and Y* is defined as follows.

$$\mathrm{IRel}\ X\ Y \coloneqq \forall\ \{i\} \to X\ i \to Y\ i \to \mathrm{Prop}$$

We denote by $\lesssim$ (in infix notation) the standard ordering on family relations, defined by

$$R \lesssim S \coloneqq \forall\ \{i\}\ (x : X\ i)\ (y : Y\ i) \to R\ x\ y \to S\ x\ y,$$

for any $R, S : \mathrm{IRel}\ X\ Y$.

We define the standard operators of diagonal, converse, and sequential composition and re-indexing on family relations, as follows.

$$\Delta^r : \mathrm{IRel}\ X\ X \qquad \Delta^r\ a\ b \coloneqq a = b$$

$$\_^\mathsf{T} : \mathrm{IRel}\ X\ Y \to \mathrm{IRel}\ Y\ X \qquad R^\mathsf{T}\ a\ b \coloneqq R\ b\ a$$

$$\_\overset{\circ}{;}\_ : \mathrm{IRel}\ X\ Y \to \mathrm{IRel}\ Y\ Z \to \mathrm{IRel}\ X\ Z$$
$$(R \overset{\circ}{;} S)\ a\ c \coloneqq \exists\ b, R\ a\ b \wedge S\ b\ c$$

$$\_\rangle\_\langle\_ : (X_2 \twoheadrightarrow X_1) \to \mathrm{IRel}\ X_1\ Y_1 \to (Y_2 \twoheadrightarrow Y_1) \to \mathrm{IRel}\ X_2\ Y_2$$
$$(f \rangle R \langle g)\ a\ b \coloneqq R\ (f\ a)\ (g\ b)$$

Definition 2.38 (Family Equivalence):
Given $R : \mathrm{IRel}\ X\ X$, we say that

- R *is reflexive* whenever $\Delta^r \lesssim R$;
- R *is symmetric* whenever $R^\mathsf{T} \lesssim R$;
- R *is transitive* whenever $R \overset{\circ}{;} R \lesssim R$; and
- R *is an equivalence* whenever it is reflexive, symmetric, and transitive.

We can now define the relational counterpart to Action_Σ.

Definition 2.39 (Action Relator):
Given $\Sigma : \mathrm{Container}\ I$, an output relation $X^r : \mathrm{IRel}\ X^1\ X^2$, and a parameter relation $A^r : \mathrm{IRel}\ A^1\ A^2$, the *action relator over* Σ of signature

$$\mathrm{data}\ \mathrm{Action}^r_\Sigma\ X^r\ A^r : \mathrm{IRel}\ (\mathrm{Action}_\Sigma\ X^1\ A^1)\ (\mathrm{Action}_\Sigma\ X^1\ A^2)$$

is defined by the following constructors.

$$\frac{x^r : X^r\ x^1\ x^2}{\text{"ret}^r\ x^r : \text{Action}^r_\Sigma\ X^r\ A^r\ (\text{"ret}\ x^1)\ (\text{"ret}\ x^2)}$$

$$\frac{t^r : A^r\ t^1\ t^2}{\text{"tau}^r\ t^r : \text{Action}^r_\Sigma\ X^r\ A^r\ (\text{"tau}\ t^1)\ (\text{"tau}\ t^2)}$$

$$\frac{q : \Sigma.\text{Query}\ i \quad k^r : (r : \Sigma.\text{Reply}\ q) \to A^r\ (k^1\ r)\ (k^2\ r)}{\text{"vis}^r\ q\ k^r : \text{Action}^r_\Sigma\ X^r\ A^r\ (\text{"vis}\ q\ k^1)\ (\text{"vis}\ q\ k^2)}$$

Remark 2.40: The above definition of Action^r_Σ is quite a mouthful, yet it should be noted that its derivation is entirely straightforward. Categorically, it is the canonical lifting of Action_Σ to relations, which can be obtained in type theory by a relational, or *parametricity* interpretation [19].

[19] Jean-Philippe Bernardy, Patrik Jansson, and Ross Paterson, "Proofs for free - Parametricity for dependent types," 2012.

Lemma 2.41:
Action^r_Σ is monotone in both arguments, i.e.

$$X^r_1 \lesssim X^r_2 \to A^r_1 \lesssim A^r_2 \to \text{Action}^r_\Sigma\ X^r_1\ A^r_1 \lesssim \text{Action}^r_\Sigma\ X^r_2\ A^r_2$$

Moreover the following statements hold (understood as universally quantified).

$$\Delta^r \lesssim \text{Action}^r_\Sigma\ \Delta^r\ \Delta^r$$
$$(\text{Action}^r_\Sigma\ X^r\ A^r)^\top \lesssim \text{Action}^r_\Sigma\ X^{r\top}\ A^{r\top}$$
$$\text{Action}^r_\Sigma\ X^r_1\ A^r_1\ ;\ \text{Action}^r_\Sigma\ X^r_2\ A^r_2 \lesssim \text{Action}^r_\Sigma\ (X^r_1\ ;\ X^r_2)\ (A^r_1\ ;\ A^r_2)$$
$$\text{too-long} \lesssim \text{Action}^r_\Sigma\ (f_1 \rangle X^r \langle f_2)\ (g_1 \rangle A^r \langle f_2)$$

with $\text{too-long} := \text{Action}_\Sigma\ f_1\ g_1 \rangle \text{Action}^r_\Sigma\ X^r\ A^r \langle \text{Action}_\Sigma\ f_2\ g_2$.

Proof: All by direct case analysis. ∎

Remark 2.42: Although we never formally provide a definition of relators, their list of conditions can be inferred from the above definition, which exhibits $\text{Action}^r_{\Sigma\Sigma}$ as a relator in two arguments. More precisely, the statement related to the converse operator is not always required and defines a lax *conversive* relator [61]. Note that although all the reverse inequalities also holds, we will not make use of them.

[61] Paul Blain Levy, "Similarity Quotients as Final Coalgebras," 2011.

Intuitively, the relator conditions are heterogeneous generalizations (strengthenings) of the facts that a relator sends reflexive relations to reflexive relations, symmetric relations to symmetric relations, etc.

Definition 2.43 (Pre-Relator Lattice):
Given $\Sigma : \mathrm{Container}\ I$, we define the interaction *pre-relator lattice over* Σ as follows.

$$\mathfrak{L}_\Sigma := \forall\ \{X^1\ X^2\} \to \mathrm{IRel}\ X^1\ X^2 \to \mathrm{IRel}\ (\mathrm{ITree}_\Sigma\ X^1)\ (\mathrm{ITree}_\Sigma\ X^2)$$

It is ultimately a set of dependent functions into Prop, as such it forms a complete lattice by pointwise lifting of the structure on Prop.

Remark 2.44: The name "pre-relator" comes from the fact that the elements of this lattice have the same type as ITree_Σ relators, but they are not constrained by any condition.

Definition 2.45 (Strong Bisimilarity):
Given $\Sigma : \mathrm{Container}\ I$, we define the *strong bisimulation map over* Σ as the following monotone endo-map on the pre-relator lattice over Σ.

$$\mathrm{sbisim}_\Sigma : \mathfrak{L}_\Sigma \to \mathfrak{L}_\Sigma$$
$$\mathrm{sbisim}_\Sigma\ \mathcal{F}\ X^r := \mathrm{out} \rangle \mathrm{Action}^r_\Sigma\ X^r\ (\mathcal{F}\ X^r) \langle\ \mathrm{out}$$

For any given family relation $X^r : \mathrm{IRel}\ X^1\ X^2$, we define heterogeneous and homogeneous *strong bisimilarity* over X^r, denoted by $\cong[\ X^r\ ]$, as follows.

$$a \cong[\ X^r\ ]\ b := \nu\ \mathrm{sbisim}_\Sigma\ X^r\ a\ b$$
$$a \cong b \qquad := a \cong[\ \Delta^r\ ]\ b$$

Lemma 2.46:
Given $\Sigma : \mathrm{Container}\ I$, for all strong bisimulation candidates $\mathcal{F} \in \mathrm{Tower}_{\mathrm{sbisim}_\Sigma}$, the following statements are true.

$$X^r_1 \lesssim X^r_2 \to \mathcal{F}\ X^r_1 \lesssim \mathcal{F}\ X^r_2$$
$$\Delta^r \lesssim \mathcal{F}\ \Delta^r$$
$$(\mathcal{F}\ X^r)^\top \lesssim \mathcal{F}\ X^{r\top}$$
$$\mathcal{F}\ X^r_1\ ;\ \mathcal{F}\ X^r_2 \lesssim \mathcal{F}\ (X^r_1\ ;\ X^r_2)$$

As a consequence, when $X^r : \mathrm{IRel}\ X\ X$ is an equivalence relation, $\mathcal{F}\ X^r$ is an equivalence relation. As a particularly important consequence, recalling that $\nu\ \mathrm{sbisim}_\Sigma \in \mathrm{Tower}_{\mathrm{sbisim}_\Sigma}$, all the above statements are true for strong bisimilarity, and thus $\cong$ is an equivalence relation.

Proof: All statements are proven by direct tower induction, applying the corresponding statement from Lemma 2.41.

For example for the first one, pose $P\ x$ to be the goal, i.e.,

$$P\ \mathcal{F} := \forall\ \{X^1\ X^2\}\ \{X_1^r\ X_2^r : \mathrm{IRel}\ X^1\ X^2\}$$
$$\to X_1^r \lesssim X_2^r \to \mathcal{F}\ X_1^r \lesssim \mathcal{F}\ X_2^r.$$

P is inf-closed. Moreover, the premise of tower induction requires that

$$P\ \mathcal{F} \to P\ (\mathrm{sbisim}_\Sigma\ \mathcal{F}),$$

i.e., introducing all arguments of the implication we need to prove

$$\frac{\forall\ \{X^1\ X^2\}\ \{X_1^r\ X_2^r : \mathrm{IRel}\ ..\} \to X_1^r \lesssim X_2^r \to \mathcal{F}\ X_1^r \lesssim \mathcal{F}\ X_2^r \quad X_1^r \lesssim X_2^r \quad \mathrm{Action}_\Sigma^r\ X_1^r\ (\mathcal{F}\ X_1^r)\ (t_1.\mathrm{out})\ (t_2.\mathrm{out})}{\mathrm{Action}_\Sigma^r\ X_2^r\ (\mathcal{F}\ X_2^r)\ (t_1.\mathrm{out})\ (t_2.\mathrm{out})},$$

which follows by direct application of the fact that Action_Σ^r is monotone in both arguments (Lemma 2.41). ■

Remark 2.47: In Lemma 2.46, a statement similar to the one concerned with the functorial re-indexing which we had in Lemma 2.41 is missing. It is simply because we have not yet defined the action of ITree_Σ on morphisms, but we will prove it in due time.

This completes the basic theory of strong bisimilarity: we have defined it and proved its most important properties in Lemma 2.46, namely that when the relation on outputs is well-behaved, not only strong bisimilarity is an equivalence relation, but *every bisimulation candidate* is an equivalence relation. As a consequence, during any tower induction proof, by definition featuring such a bisimulation candidate, one can use these properties. In a sense, strong bisimulation proofs can work up-to reflexivity, symmetry and transitivity.

2.4.3 Weak Bisimilarity

As previously hinted at, we wish to characterize a second notion of bisimilarity, which would gloss over the precise number of silent "tau moves of the two considered interaction trees. While strong bisimilarity will play the role of (extensional) equality between trees, that is, a technical tool, weak bisimilarity will play the role of a semantic equivalence.

To define weak bisimilarity, we follow a similar route to strong bisimilarity, reusing the action relator but defining a new monotone endo-map on the pre-relator lattice. To build this monotone map, we will sequence the action relator on the left and on the right with a small gadget, allowing to skip over a finite number of silent moves before landing in the action relator. This gadget, the *eating* relation $\searrow$, can be understood as a form of reduction relation on interac-

tion trees, with $t \searrow t'$ stating that t starts with some amount of silent steps and finally arrives at t'.

For readability, let us start by defining a shorthand for trees where the top layer of actions has been exposed.

Definition 2.48 (Exposed Interaction trees):
Given $\Sigma : \mathrm{Container}\ I$ and $X : \mathrm{Type}^I$, define *exposed* interaction trees with output X as the following shorthand.

$$\text{"ITree}_\Sigma\ X := \mathrm{Action}_\Sigma\ X\ (\mathrm{ITree}_\Sigma\ X)$$

Definition 2.49 (Eating Relation):
Given $\Sigma : \mathrm{Container}\ I$ and $X : \mathrm{Type}^I$, define the inductive *eating relation* data $\mathrm{Eat}^X_\Sigma : \mathrm{IRel}\ (\text{"ITree}_\Sigma\ X)\ (\text{"ITree}_\Sigma\ X)$ by the following constructors.

$$\frac{}{\text{eat-refl} : \mathrm{Eat}^X_\Sigma\ t\ t} \qquad \frac{e : \mathrm{Eat}^X_\Sigma\ (t_1.\mathrm{out})\ t_2}{\text{eat-step}\ e : \mathrm{Eat}^X_\Sigma\ (\text{"tau}\ t_1)\ t_2}$$

We define the following shorthands:

$$\llcorner \searrow \lrcorner := \mathrm{Eat}^X_\Sigma \qquad \llcorner \swarrow \lrcorner := \mathrm{Eat}^{X^\top}_\Sigma$$

Lemma 2.50:
For all Σ and R, the eating relation Eat^R_Σ is reflexive and transitive.

Proof: By direct induction. ∎

Definition 2.51 (Weak Bisimilarity):
Given $\Sigma : \mathrm{Container}\ I$, we define the *weak bisimulation map over* Σ as the following monotone endo-map on the pre-relator lattice over Σ (Definition 2.43).

$$\mathrm{wbisim}_\Sigma : \mathfrak{L}_\Sigma \to \mathfrak{L}_\Sigma$$
$$\mathrm{wbisim}_\Sigma\ \mathcal{F}\ X^r := \mathrm{out} \rangle \searrow ;\ \mathrm{Action}^r_\Sigma\ X^r\ (\mathcal{F}\ X^r)\ ;\ \swarrow \langle\ \mathrm{out}$$

We define heterogeneous and homogeneous *weak bisimilarity* as follows.

$$a \approx[\ X^r\ ]\ b := \nu\ \mathrm{wbisim}_\Sigma\ X^r\ a\ b$$
$$a \approx b \qquad := a \approx[\ \Delta^r\ ]\ b$$

Remark 2.52: The weak bisimulation map can be understood quite simply. Given a pre-relator $\mathcal{F}$ over Σ and a relation on outputs, it relates interaction trees which both "reduce" to some smaller trees, peeling away some number of silent steps, then both emit "synchronized" moves as constrained by Action^r_Σ, i.e., both return, silent or visible moves, and finally the resulting subtrees are related by $\mathcal{F}$.

We can now give a first batch of easy properties on weak bisimilarity and weak bisimulation candidates.

Lemma 2.53:
Given $\Sigma : \mathrm{Container}\ I$, for all candidate weak bisimulations $\mathcal{F} \in \mathrm{Tower}_{\mathrm{wbisim}_\Sigma}$, the following statements hold.

$$X_1^r \lesssim X_2^r \to \mathcal{F}\ X_1^r \lesssim \mathcal{F}\ X_2^r$$
$$\Delta^r \lesssim \mathcal{F}\ \Delta^r$$
$$(\mathcal{F}\ X^r)^\top \lesssim \mathcal{F}\ X^{r\top}$$

Recalling again that $\nu\ \mathrm{wbisim}_\Sigma \in \mathrm{Tower}_{\mathrm{wbisim}_\Sigma}$, these statements apply to weak bisimilarity, and in particular the homogeneous weak bisimilarity $\approx$ is reflexive and symmetric.

Proof: All by direct tower induction, as for Lemma 2.46. ∎

Notice that in the above lemma, compared to Lemma 2.46, we have left out the statement regarding sequential composition of relations which generalizes the fact that any weak bisimulation candidates sends transitive relations to transitive relations. Indeed it is well-known that weak bisimulation up-to transitivity is not a valid proof technique [82]. As such, there is no hope that the corresponding statement on weak bisimulations candidates holds. However, we would still like to prove that weak bisimilarity is transitive! The proof is slightly more involved and will involve a lot of shuffling around the eating gadget.

[82] Damien Pous and Davide Sangiorgi, "Enhancements of the bisimulation proof method," 2011.

Once again, to make the notations more compact and less overwhelming, we need to introduce several helpers for working with one-step unfolded bisimulation relations.

Definition 2.54 (Exposed Bisimulations):
Given $\Sigma : \mathrm{Container}\ I$, $\mathcal{F} : \mathfrak{L}_\Sigma$ and $X^r : \mathrm{IRel}\ X^1\ X^2$, define the following shorthands for *strong bisimulation unfolding*

$${}^{\mathrm{s}}\mathcal{F} : \mathrm{IRel}\ (\text{"}\mathrm{ITree}_\Sigma\ X^1)\ (\text{"}\mathrm{ITree}_\Sigma\ X^2)$$
$${}^{\mathrm{s}}\mathcal{F} := \mathrm{Action}^r_\Sigma\ X^r\ (\mathcal{F}\ X^r)$$

and *weak bisimulation unfolding*

$${}^{\mathrm{w}}\mathcal{F} : \mathrm{IRel}\ (\text{"}\mathrm{ITree}_\Sigma\ X^1)\ (\text{"}\mathrm{ITree}_\Sigma\ X^2)$$
$${}^{\mathrm{w}}\mathcal{F} := \searrow ; \mathrm{Action}^r_\Sigma\ X^r\ (\mathcal{F}\ X^r) ; \nearrow$$

We can now prove generalized transitivity of weak bisimilarity.

Lemma 2.55:
Given $\Sigma : \mathrm{Container}\ I$, $X_1^r : \mathrm{IRel}\ X_1\ X_2$, and $X_2^r : \mathrm{IRel}\ X_2\ X_3$, the following holds.

$$\approx[\ X_1^r\ ]\fatsemi\approx[\ X_2^r\ ] \quad \lesssim \quad \approx[\ X_1^r \fatsemi X_2^r\ ]$$

Proof: Pose the following shorthands, respectively for "one step synchronization then weak bisimilarity" and for one-step unfolding of weak bisimilarity.

Let us recall our new notations as applied to weak bisimilarity. Note that the output relation X^r will be hidden away, implicitly instantiated either as X_1^r, X_2^r or $X_1^r \fatsemi X_2^r$.

$$^{s}\approx \coloneqq \mathrm{Action}^r_\Sigma\ X^r\ (\approx[\ X^r\ ])$$
$$^{w}\approx \coloneqq \searrow \fatsemi\ ^{s}\approx \fatsemi \swarrow$$

We prove the following statements by direct induction on the eating relation for all a, b, c.

(1) $a \searrow\ ^{\tau}\mathrm{tau}\ b\ ^{s}\approx c \to a\ ^{s}\approx c$
(2) $a\ ^{s}\approx\ ^{\tau}\mathrm{tau}\ b \swarrow c \to a\ ^{s}\approx c$

We then observe that the following statements are true by case analysis.

(3) $^{\tau}\mathrm{tau}\ a\ ^{w}\approx b \to a.\mathrm{out}\ ^{w}\approx b$
(4) $a\ ^{w}\approx\ ^{\tau}\mathrm{tau}\ b \to a\ ^{w}\approx b.\mathrm{out}$

Using 3. and 4. and transitivity of the eating relation, prove the following statements by induction.

(5) $a\ (^{w}\approx \fatsemi \searrow)\ ^{\tau}\mathrm{ret}\ r \to a\ (\searrow \fatsemi\ ^{s}\approx)\ ^{\tau}\mathrm{ret}\ r$
(6) $a\ (^{w}\approx \fatsemi \searrow)\ ^{\tau}\mathrm{vis}\ q\ k \to a\ (\searrow \fatsemi\ ^{s}\approx)\ ^{\tau}\mathrm{vis}\ q\ k$
(7) $^{\tau}\mathrm{ret}\ r\ (\swarrow \fatsemi\ ^{w}\approx)\ b \to\ ^{\tau}\mathrm{ret}\ r\ (^{s}\approx \fatsemi \swarrow)\ b$
(8) $^{\tau}\mathrm{vis}\ q\ k\ (\swarrow \fatsemi\ ^{w}\approx)\ b \to\ ^{\tau}\mathrm{vis}\ q\ k\ (^{s}\approx \fatsemi \swarrow)\ b$

Finally, note that the following is true by (nested) induction.

(9) $a\ (\swarrow \fatsemi \searrow)\ b \to a \searrow b \vee a \swarrow b$

Finally, we prove the theorem statement by tower induction on

$$P\ \mathcal{F} \coloneqq \approx[\ X_1^r\ ] \fatsemi \approx[\ X_2^r\ ] \lesssim \mathcal{F}\ (X_1^r \fatsemi X_2^r).$$

P is inf-closed. Assuming $P\ \mathcal{F}$, let us prove $P\ (\mathrm{wbisim}_\Sigma\ \mathcal{F})$, i.e.,

$$\approx[\ X_1^r\ ] \fatsemi \approx[\ X_2^r\ ] \lesssim \mathrm{wbisim}_\Sigma\ \mathcal{F}\ (X_1^r \fatsemi X_2^r)$$

By one step unfolding, it suffices to prove the following.

$$ {}^{w}\!\approx ; {}^{w}\!\approx \;\lesssim\; {}^{w}\mathcal{F} $$

Introducing and decomposing the hypotheses, we obtain the following:

$$ a \searrow x_1 \; {}^{s}\!\approx x_2 \swarrow b \searrow y_1 \; {}^{s}\!\approx y_2 \swarrow c $$

Apply 9. between x_2 and y_1. Assume w.l.o.g. that the left case is true i.e., $x_2 \searrow y_1$ (the right case is symmetric). By case on y_1.

- When $y_1 := \text{"ret } r$,

  $$\begin{array}{llll} a \searrow x_1 \; {}^{s}\!\approx x_2 & & \searrow \text{"ret } r \; {}^{s}\!\approx y_2 \swarrow c & \\ a \searrow x_1 \; {}^{s}\!\approx x_2 \swarrow x_2 & \searrow \text{"ret } r \; {}^{s}\!\approx y_2 \swarrow c & & \text{by refl.} \\ a & {}^{w}\!\approx & x_2 \searrow \text{"ret } r \; {}^{s}\!\approx y_2 \swarrow c & \text{by def.} \\ a & \searrow & x_3 \; {}^{s}\!\approx \text{"ret } r \; {}^{s}\!\approx y_2 \swarrow c & \text{by 5.} \end{array}$$

  By concatenation (Lemma 2.41) between x_3 and y_2, we obtain

  $$ \text{Action}^{r}_{\Sigma}\ (X_1^r \,;\, X_2^r)\ (\approx[\ X_1^r\ ] \,;\, \approx[\ X_2^r\ ])\ x_3\ y_2. $$

  By coinduction hypothesis $\approx[\ X_1^r\ ] \,;\, \approx[\ X_1^r\ ] \lesssim \mathcal{F}\ (X_1^r \,;\, X_2^r)$. As such, by monotonicity (Lemma 2.41) we obtain

  $$ \text{Action}^{r}_{\Sigma}\ (X_1^r \,;\, X_2^r)\ (\mathcal{F}\ (X_1^r \,;\, X_2^r))\ x_3\ y_2 $$

  and finally conclude ${}^{w}\mathcal{F}\ a\ b$.

- When $y_1 := \text{"vis } q\ k$, the reasoning is the same as for $y_1 := \text{"ret } r$, swapping lemma 5. with lemma 6.

- When $y_1 := \text{"tau } t$,

  $$\begin{array}{llll} a \searrow x_1 \; {}^{s}\!\approx x_2 \searrow \text{"tau } t \; {}^{s}\!\approx y_2 \swarrow c & & & \\ a \searrow x_1 \; {}^{s}\!\approx x_2 & & {}^{s}\!\approx y_2 \swarrow c & \text{by 1.} \end{array}$$

  By Lemma 2.41, using concatenation between x_2 and y_2, we obtain

  $$ \text{Action}^{r}_{\Sigma}\ (X_1^r \,;\, X_2^r)\ (\approx[\ X_1^r\ ] \,;\, \approx[\ X_2^r\ ])\ x_3\ y_2, $$

  we then conclude as before by monotonicity applied to the coinduction hypotheses.

This concludes our tower induction, proving that for all weak bisimulation candidate $\mathcal{F} \in \text{ITree}_{\text{wbisim}_{\Sigma}}$, we have $\approx[\ X_1^r\ ] \,;\, \approx[\ X_2^r\ ] \lesssim \mathcal{F}\ (X_1^r \,;\, X_2^r)$. As in particular weak bisimilarity is a bisimulation candidate, we finally conclude our generalized transitivity property

$$ \approx[\ X_1^r\ ] \,;\, \approx[\ X_2^r\ ] \quad\lesssim\quad \approx[\ X_1^r \,;\, X_2^r\ ]. $$

■

After transitivity, we would like another reasoning principle such that during any weak bisimulation proof, we can freely rewrite by any strong bisimilarity proof.

Lemma 2.56 (Up-to Strong Bisimilarity):
Given $\Sigma : \mathrm{Container}\ I$, $X_1^r : \mathrm{IRel}\ X_1\ X_2$, $X_2^r : \mathrm{IRel}\ X_2\ X_3$ and $X_3^r : \mathrm{IRel}\ X_3\ X_4$, the following holds for any weak bisimulation candidates $\mathcal{F} \in \mathrm{Tower}_{\mathrm{wbisim}_\Sigma}$,

$$\cong[\ X_1^r\ ] \fatsemi \mathcal{F}\ X_2^r \fatsemi \cong[\ X_3^r\ ] \lesssim \mathcal{F}\ (X_1^r \fatsemi X_2^r \fatsemi X_3^r).$$

Proof: Let us define the following shorthand.

Recall again the definition of our compact notations. Note that we will use both ${}^{s}\mathcal{F}$ and ${}^{w}\mathcal{F}$ in infix style. As before, the output relation X^r will be hidden and instantiated variously.

$$\begin{aligned} {}^{s}\cong &:= \mathrm{Action}_\Sigma^r\ X^r \cong[\ X^r\ ] \\ {}^{s}\mathcal{F} &:= \mathrm{Action}_\Sigma^r\ X^r\ (\mathcal{F}\ X^r) \\ {}^{w}\mathcal{F} &:= \searrow \fatsemi {}^{s}\mathcal{F} \fatsemi \swarrow \end{aligned}$$

Prove the following statements by direct induction.

(1) $a\ ({}^{s}\cong \fatsemi \searrow)\ b \to a\ (\searrow \fatsemi {}^{s}\cong)\ b$
(2) $a\ (\swarrow \fatsemi {}^{s}\cong)\ b \to a\ ({}^{s}\cong \fatsemi \swarrow)\ b$

Then, let us prove the goal by tower induction on

$$P\ \mathcal{F} := \cong[\ X_1^r\ ] \fatsemi \mathcal{F}\ X_2^r \fatsemi \cong[\ X_3^r\ ] \lesssim \mathcal{F}\ (X_1^r \fatsemi X_2^r \fatsemi X_3^r).$$

P is inf-closed. Assuming $P\ \mathcal{F}$, let us prove $P\ (\mathrm{wbisim}_\Sigma\ \mathcal{F})$, i.e.,

$$\cong[\ X_1^r\ ] \fatsemi \mathrm{wbisim}_\Sigma\ \mathcal{F}\ X_2^r \fatsemi \cong[\ X_3^r\ ] \lesssim \mathrm{wbisim}_\Sigma\ \mathcal{F}\ (X_1^r \fatsemi X_2^r \fatsemi X_3^r).$$

By one-step unfolding it suffices to prove the following.

$${}^{s}\cong \fatsemi {}^{w}\mathcal{F} \fatsemi {}^{s}\cong \lesssim {}^{w}\mathcal{F}$$

Introducing and destructing the hypotheses we proceed as follows.

$$a\ {}^{s}\cong\ b\ \searrow x_1\ {}^{s}\mathcal{F}\ x_2\ \swarrow\ c\ {}^{s}\cong\ d$$
$$a\ \searrow\ y_1\ {}^{s}\cong\ x_1\ {}^{s}\mathcal{F}\ x_2\ {}^{s}\cong\ y_2\ \swarrow\ d \quad \text{by 1. and 2.}$$

By concatenation (Lemma 2.41) between y_1 and y_2, we obtain

$$\mathrm{Action}_\Sigma^r\ (X_1^r \fatsemi X_2^r \fatsemi X_3^r)\ (\cong[\ X_1^r\ ] \fatsemi \mathcal{F}\ X_2^r \fatsemi \cong[\ X_3^r\ ])\ y_1\ y_2.$$

By coinduction hypothesis $P\ \mathcal{F}$ and monotonicity (Lemma 2.41) we deduce $y_1\ {}^{s}\mathcal{F}\ y_2$ and finally conclude $a\ {}^{w}\mathcal{F}\ b$ ∎

Let us reap some benefits from the very general lemmas we have proven until now and deduce that strong bisimilarity is included in weak bisimilarity. Technically the proof is so simple that we would never really use it, as we would instead "prove" it on-the-fly by applying one of the more powerful principles. This is particularly true in Rocq, where the `setoid_rewrite` mechanism [88] can be hooked up to all of our monotonicity statements and relation inclusions. This is quite appreciable as there is a myriad of precise cases where we would like to "rewrite" by a known bisimilarity proof. Justifying them requires tedious chains of applications of structural lemmas regarding reflexivity, symmetry and transitivity such as we just proved. In the following chapters we will not justify them as thoroughly, but for now let us see how this inclusion property goes.

[88] Matthieu Sozeau, "A New Look at Generalized Rewriting in Type Theory," 2009.

Lemma 2.57 (Strong To Weak):
Given $\Sigma : \mathrm{Container}\ I$ and $X^r : \mathrm{IRel}\ X^1\ X^2$, the following inclusion holds.

$$\approx[\ X^r\ ] \lesssim \approx[\ X^r\ ].$$

Proof: Of course the direct proof by tower induction is quite trivial, but as motivated above, let us prove it solely by using already proven properties.

$$\begin{aligned}
\approx[\ X^r\ ] &\lesssim \approx[\ X^r\ ] \fatsemi \Delta^r \fatsemi \Delta^r && \text{diagonal concatenation} \\
&\lesssim \approx[\ X^r\ ] \fatsemi \approx[\ \Delta^r\ ] \fatsemi \approx[\ \Delta^r\ ] && \text{Lemma 2.53 and 2.46} \\
&\lesssim \approx[\ X^r \fatsemi \Delta^r \fatsemi \Delta^r\ ] && \text{Lemma 2.56} \\
&\lesssim \approx[\ X^r\ ] && \text{Lemma 2.53}
\end{aligned}$$

The only important step is the call to Lemma 2.56, all the rest is simply a matter of filling "missing arguments" by reflexivity and refolding them in the conclusion by monotonicity. ■

This sequence of juggling concludes our core properties for weak bisimilarity: we know that for well-behaved X^r it is an equivalence relation and that it supports coinductive proofs up-to reflexivity, up-to symmetry and up-to strong bisimilarity.

2.5 Monad Structure

An important structure available on interaction trees is that they form a monad. Indeed, as they are parametrized by an *output* family X, a strategy with output X can be considered as an impure computation returning some X. Its *effects* will be to perform game moves and wait for an answer. While at first sight—considering only the goal of representing game strategies—such an output might seem unnecessary, the compositionality offered by monads, that is, sequential composition, is tremendously useful to construct and reason on strategies piece wise.

The monad structure on interaction trees takes place in the family category Type^I and its laws hold both w.r.t. strong bisimilarity and weak bisimilarity. One way to view this is to say that we will define *two* monads, respectively defined by quotienting the resulting trees by strong or weak bisimilarity. However, in line with our choice of using intensional type theory, we will first define a *pre-monad* structure, containing only the computationally relevant operation and then provide two sets of laws.

In fact, in Definition 2.20, we have already defined the "return" operator, ret, which can be typed as follows.

$$\mathrm{ret}\ \{X\} : X \to \mathrm{ITree}_\Sigma\ X$$

Let us define the *fmap* operator and the *bind* operator, which works by tree grafting.

Definition 2.58 (Interaction Tree Fmap):
For any $\Sigma : \mathrm{Container}\ I$, *interaction tree fmap* operator is given by the following coinductive definition.

$$\_\langle\$\rangle\_\ \{X\ Y\} : (X \to Y) \to \mathrm{ITree}_\Sigma\ X \to \mathrm{ITree}_\Sigma\ Y$$

$$f\ \langle\$\rangle\ t := \left[\begin{array}{l} \mathrm{out} := \mathrm{case}\ t.\mathrm{out} \\ \left[\begin{array}{ll} \text{'ret}\ x & := \text{'ret}\ (f\ x) \\ \text{'tau}\ t & := \text{'tau}\ (f\ \langle\$\rangle\ t) \\ \text{'vis}\ q\ k & := \text{'vis}\ q\ (\lambda\ r \mapsto f\ \langle\$\rangle\ k\ r) \end{array}\right] \end{array}\right.$$

Definition 2.59 (Interaction Tree Bind):
Let $\Sigma : \mathrm{Container}\ I$. For any given $X, Y : \mathrm{Type}^I$ and $f : X \to \mathrm{ITree}_\Sigma\ Y$, first define *interaction tree substitution* as follows.

$$\mathrm{subst}_f : \mathrm{ITree}_\Sigma\ X \to \mathrm{ITree}_\Sigma\ Y$$

$$\mathrm{subst}_f\ t := \left[\begin{array}{l} \mathrm{out} := \mathrm{case}\ t.\mathrm{out} \\ \left[\begin{array}{ll} \text{'ret}\ x & := (f\ x).\mathrm{out} \\ \text{'tau}\ t & := \text{'tau}\ \left(\mathrm{subst}_f\ t\right) \\ \text{'vis}\ q\ k & := \text{'vis}\ q\ \left(\lambda\ r \mapsto \mathrm{subst}_f\ (k\ r)\right) \end{array}\right] \end{array}\right.$$

Note that defining subst_f *with* f *fixed* is not a mere stylistic consideration. Indeed, what it achieves, is to pull the binder for f out of the coinductive definition. This enables the syntactic guardedness checker to more easily understand subsequent coinductive definitions making use of the bind operator. To the best of my knowledge, this trick was first used in the InteractionTree library [93]. In general, it is always fruitful to take as many binders as possible out of a coinductive definition.

Then, define the *interaction tree bind* operator as

$$\_ \gg\!= \_\ \{X\ Y\ i\} : \mathrm{ITree}_\Sigma\ X\ i \to (X \to \mathrm{ITree}_\Sigma\ Y) \to \mathrm{ITree}_\Sigma\ Y\ i$$
$$t \gg\!= f := \mathrm{subst}_f\ t$$

and the *kleisli composition* as

$$\_ \gg\!= \_ \{X\ Y\ Z\} : (X \to \mathrm{ITree}_\Sigma\ Y) \to (Y \to \mathrm{ITree}_\Sigma\ Z) \to (X \to \mathrm{ITree}_\Sigma\ Z)$$
$$(f \gg\!= g)\ x := f\ x \gg\!= g.$$

Remark 2.60: The above presentation of the bind operator can be recognized as the *daemonic bind* of McBride [69], who studied monads on type families of the exact same kind as we do. The indexing provides us with the *starting* position of the computation (in our case, strategy), but we do not know the *ending* position at which a ret might appear. In face of this uncertainty, which they called *daemonic*, the continuation must be able to treat *any* possible position.

[69] Conor McBride, "Kleisli Arrows of Outrageous Fortune," 2011.

McBride further shows how we can generically derive a doubly-indexed *parametrized monad* [17] this time operating on sets and not families, which can be represented as follows.

[17] Robert Atkey, "Parameterised notions of computation," 2009.

$$M : \mathrm{Type} \to I \to I \to \mathrm{Type}$$

This second representation supports a corresponding *angelic* bind operator, where the end position of the head computation is known, such that the continuation need only to handle that one. It is typed as follows.

$$\_ \gg\!= \_ : M\ X\ i\ j \to (X \to M\ Y\ j\ k) \to M\ X\ i\ k$$

The trick is to define the following predicate *at-key* essentially encoding the fibers of a constant function $(\lambda\ x \mapsto i) : X \to I$.

$$\mathrm{data}\ \_@\_ : \mathrm{Type} \to I \to I \to \mathrm{Type} \qquad \frac{x : X}{\mathrm{ok}\ x : (X\ @\ i)\ i}$$

This enables to type a hybrid *angelic* bind on interaction trees as follows.

$$\_ \gg\!= \_ : \mathrm{ITree}_\Sigma\ (X\ @\ j)\ i \to (X \to \mathrm{ITree}_\Sigma\ Y\ j) \to \mathrm{ITree}_\Sigma\ Y\ i$$

Sadly this thesis we make very little use of such new tricks that can be pulled in indexed settings. We will only need to study the operators and properties lifted from the non-indexed setting, which are thus always quite unoriginal w.r.t. indexing.

Before proving the monad laws, we will first prove that our operators respect both strong and weak bisimilarity, in other words that they are monotone. For strong bisimilarity and ret, the statement is the following.

$$\forall\ \{X^r : \mathrm{IRel}\ X^1\ X^2\}\ \{i : I\}\ \{x_1 : X^1\ i\}\ \{x_2 : X^1\ i\} \to X^r\ x_1\ x_2 \to \mathrm{ret}\ x_1 \cong[\ X^r\ ]\ \mathrm{ret}\ x_2$$

This is quite heavy, and many more complex monotonicity statements will appear in the thesis. Thus, from now on, we will extensively use relational combinators. To simplify reading such complex relations, we will write $a \langle\!\langle R \rangle\!\rangle b$ for $R\ a\ b$. Our final goal is to replace the above verbose statement with the following.

$$\forall\ \{X^r\} \to \mathrm{ret} \langle\!\langle X^r \to^r \cong[\ X^r\ ] \rangle\!\rangle \mathrm{ret}$$

To achieve this, we define the following combinators.

$$\_ \to^r \_ : \mathrm{Rel}\ X_1\ X_2 \to \mathrm{Rel}\ Y_1\ Y_2 \to \mathrm{Rel}\ (X_1 \to Y_1)\ (X_2 \to Y_2)$$
$$(R \to^r S)\ f\ g := \forall\ \{x_1 x_2\} \to R\ x_1\ x_2 \to S\ (f\ x_1)\ (g\ x_2)$$

$$\_ \to\!\!\!\!\cdot^r \_ : \mathrm{IRel}\ X_1\ X_2 \to \mathrm{IRel}\ Y_1\ Y_2 \to \mathrm{IRel}\ (X_1 \to Y_1)\ (X_2 \to Y_2)$$
$$(R \to\!\!\!\!\cdot^r S)\ f\ g := \forall\ \{i\ x_1\ x_2\} \to R\ \{i\}\ x_1\ x_2 \to S\ \{i\}\ (f\ x_1)\ (g\ x_2)$$

$$\forall^r : \mathrm{IRel}\ X_1\ X_2 \to \mathrm{Rel}\ (\forall\ \{i\} \to X_1\ i)\ (\forall\ \{i\} \to X_2\ i)$$
$$\forall^r R\ f\ g := \forall\ \{i\} \to R\ (f\ \{i\})\ (g\ \{i\})$$

Moreover, we will write $\forall^r A$ for $\forall^r (\lambda\ \{i\} \mapsto A)$, with a very weak parsing precedence.

Lemma 2.61 (ITree Monad Monotonicity):
Given $\Sigma : \mathrm{Container}\ I$, for any X^r and Y^r and for any strong or weak bisimulation candidate $\mathcal{F} \in \mathrm{sbisim}_\Sigma$ or $\mathcal{F} \in \mathrm{wbisim}_\Sigma$, the following holds.

(1) $\mathrm{ret} \langle\!\langle \forall^r X^r \to^r \mathcal{F}\ X^r \rangle\!\rangle \mathrm{ret}$
(2) $(\_ \langle \$ \rangle \_) \langle\!\langle (\forall^r X^r \to^r Y^r) \to^r (\forall^r \mathcal{F}\ X^r \to^r \mathcal{F}\ Y^r) \rangle\!\rangle (\_ \langle \$ \rangle \_)$
(3) $(\_ \gg\!= \_) \langle\!\langle \forall^r \mathcal{F}\ X^r \to^r (\forall^r X^r \to^r \mathcal{F}\ Y^r) \to^r \mathcal{F}\ Y^r \rangle\!\rangle (\_ \gg\!= \_)$

As direct consequences, return, fmap, and bind respect both strong and weak bisimilarity:
(4) $\mathrm{ret} \langle\!\langle \forall^r X^r \to^r \cong[\ X^r\ ] \rangle\!\rangle \mathrm{ret}$
(5) $\mathrm{ret} \langle\!\langle \forall^r X^r \to^r \approx[\ X^r\ ] \rangle\!\rangle \mathrm{ret}$
(6) $(\_ \langle \$ \rangle \_) \langle\!\langle (\forall^r X^r \to^r Y^r) \to^r (\forall^r \cong[\ X^r\ ] \to^r \cong[\ Y^r\ ]) \rangle\!\rangle (\_ \langle \$ \rangle \_)$
(7) $(\_ \langle \$ \rangle \_) \langle\!\langle (\forall^r X^r \to^r Y^r) \to^r (\forall^r \approx[\ X^r\ ] \to^r \approx[\ Y^r\ ]) \rangle\!\rangle (\_ \langle \$ \rangle \_)$
(8) $(\_ \gg\!= \_) \langle\!\langle \forall^r \cong[\ X^r\ ] \to^r (\forall^r X^r \to^r \cong[\ Y^r\ ]) \to^r \cong[\ Y^r\ ] \rangle\!\rangle (\_ \gg\!= \_)$
(9) $(\_ \gg\!= \_) \langle\!\langle \forall^r \approx[\ X^r\ ] \to^r (\forall^r X^r \to^r \approx[\ Y^r\ ]) \to^r \approx[\ Y^r\ ] \rangle\!\rangle (\_ \gg\!= \_)$

Proof:

(1) For $\mathcal{F} \in \mathrm{sbisim}_\Sigma$, assuming $X^r\ x_1\ x_2$, $\backslash\!\!\backslash \mathrm{ret}^r$ proves

$$\mathrm{sbisim}_\Sigma\ \mathcal{F}\ X^r\ (\mathrm{ret}\ x_1)\ (\mathrm{ret}\ x_2),$$

which by Lemma 2.35 entails $\mathcal{F}\ X^r\ (\mathrm{ret}\ x_1)\ (\mathrm{ret}\ x_2)$. The proof for $\mathcal{F} \in \mathrm{wbisim}_\Sigma$ is similar, using reflexivity of Eat_Σ.

(2) By tower induction on the statement. For $\mathcal{F} \in \mathrm{sbisim}_\Sigma$ it is direct by case analysis. For $\mathcal{F} \in \mathrm{wbisim}_\Sigma$, use the fact that

$$t_1 \Downarrow t_2 \to (f\ ^\backprime\langle\$\rangle\ t_1) \Downarrow (f\ ^\backprime\langle\$\rangle\ t_2)$$

where $^\backprime\langle\$\rangle$ is the one step unfolding of $\langle\$\rangle$ operating on an exposed interaction tree.

(3) By tower induction on the statement. For $\mathcal{F} \in \mathrm{sbisim}_\Sigma$ it is direct by case analysis. For $\mathcal{F} \in \mathrm{wbisim}_\Sigma$, use the following fact.

$$t_1 \Downarrow t_2 \to (t_1\ ^\backprime\gg\!=\ f) \Downarrow (t_2\ ^\backprime\gg\!=\ f)$$

(4–9) By direct application of (1–3), using Theorem 2.36. ∎

While perhaps not very impressive, the above lemma is very important. Points (4–9) prove that return, fmap and bind are well-defined as operators on the quotients of strategies respectively by strong and weak bisimilarity. But more importantly, points (1–3) prove that one can reason compositionally *during* any bisimulation proof. To relate two returns under any bisimulation candidate, simply relate their output. To relate two sequential compositions (bind) under any bisimulation candidate, it suffices to first relate the two head computations and then, pointwise, the continuations.

Remark 2.62: This last fact (3) is sometimes called "bisimulation up-to bind". We can now see why it was important to construct bisimilarity not as fixed points on a lattice of relations but on a lattice of pre-relators. Indeed, the statement (3) makes use of the bisimulation candidate at two different output relations X^r and Y^r. If we were to use the lattice of family relations, the output relation parameter of a bisimulation candidate would be fixed, making the statement impossible to write. As it is done in the InteractionTree library [93], only a weaker statement can be made, in which the head computations must be related by bisimilarity, instead of the bisimulation candidate.

[93] Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic, "Interaction trees: representing recursive and impure programs in Coq," 2020.

Before turning to the monad and functor laws, let us revisit the property of relators regarding relation re-indexing that we were missing in Lemma 2.46 and Lemma 2.53. As we now know the action $\langle\$\rangle$ of ITree on morphisms, we can give its statement and proof. Intuitively, it states that whenever two "fmap-ed" computations are related by some bisimulation candidate, then their inside computation is actually directly related by the candidate, with a re-indexed output relation.

Lemma 2.63 (Bisimulation Re-Indexing):
Given $\Sigma : \mathrm{Container}\ I$, for any $X^r : \mathrm{IRel}\ X_1\ X_2$, $f_1 : Y_1 \to X_1$ and

$f_2 : Y_2 \nrightarrow X_2$ and for any strong or weak bisimulation candidate $\mathcal{F} \in \mathrm{sbisim}_\Sigma$ or $\mathcal{F} \in \mathrm{wbisim}_\Sigma$, the following holds.

$$(f_1 \mathbin{\langle\$\rangle} \_) \rangle \mathcal{F}\ X^r \langle (f_2 \mathbin{\langle\$\rangle} \_) \quad \lesssim \quad \mathcal{F}\ (f_1 \rangle X^r \langle f_2)$$

Proof: By tower induction on the statement. For $\mathcal{F} \in \mathrm{sbisim}_\Sigma$ it is direct by case analysis. For $\mathcal{F} \in \mathrm{wbisim}_\Sigma$, use the following fact.

$$(f_1 \mathbin{\backprime\langle\$\rangle} t_1) \searrow t_2 \to \exists\, t_3, (t_1 \searrow t_3) \wedge t_2 = (f \mathbin{\backprime\langle\$\rangle} t_3)$$

■

Remark 2.64: Together with Lemma 2.46, the above lemma verifies that any strong bisimulation candidate is a relator for ITree_Σ. Moreover, Lemma 2.61 verifies that any strong bisimulation candidate is a monad relator for ITree_Σ, in the sense of Dal Lago, Gavazzo and Levy [53]. As such, we can in a sense say that "up-to relator principles" are valid for strong bisimilarity, which is just a precise way to say that standard compositional reasoning is allowed during strong bisimilarity proofs.

[53] Ugo Dal Lago, Francesco Gavazzo, and Paul Blain Levy, "Effectful applicative bisimilarity: Monads, relators, and Howe's method," 2017.

The same is almost entirely true for weak bisimilarity, with the sole exception of transitivity. As such, although weak *bisimilarity* is indeed a monad relator, weak bisimulation *candidates* fail the generalized transitivity requirement.

We can finally prove the monad laws as well as coherence of fmap. We only prove them w.r.t. strong bisimilarity, as this implies weak bisimilarity.

Lemma 2.65 (ITree Monad Laws):
Given $\Sigma : \mathrm{ITree}_\Sigma$, for all $x : X\ i$, $t : \mathrm{ITree}_\Sigma\ X\ i$, $f : X \nrightarrow \mathrm{ITree}_\Sigma\ Y$, $g : Y \nrightarrow \mathrm{ITree}_\Sigma\ Z$ and $h : X \nrightarrow Y$ the following statements are true.

(1) $(\mathrm{ret}\ x \gg\!= f) \cong f\ x$
(2) $(t \gg\!= \mathrm{ret}) \cong t$
(3) $(t \gg\!= f) \gg\!= g \cong t \gg\!= (f \gg\!> g)$
(4) $(t \gg\!= (\mathrm{ret} \circ h)) \cong (h \mathbin{\langle\$\rangle} t)$

Proof:
(1) By one-step unfolding.
(2) By direct tower induction.
(3) By direct tower induction.
(4) By direct tower induction.

■

Remark 2.66: Note that as a direct consequence of the above lemma, we can derive the usual properties of fmap, exhibiting ITree_Σ as a functor.

This concludes the monadic theory of interaction trees. We put a finishing touch by introducing the so-called "do notation". We write, e.g.,

$$\mathrm{do}\ x \leftarrow t; y \leftarrow f\ x; g\ y$$

instead of

$$t \gg\!= (\lambda\, x \mapsto f\ x \gg\!= (\lambda\, y \mapsto g\ y)).$$

2.6 Iteration Operators

Interaction trees [93] were originally introduced to encode arbitrary—i.e., possibly non-terminating—computation. As such, apart from monadic operators, they support *iteration operators* which intuitively allow one to write arbitrary "while" loops. Pioneered by Elgot in the setting of fixed points in algebraic theories [36], iteration in monadic computations enjoys a vast literature. Recalling that a monadic term $a : M\ X$ can be seen intuitively as an "M-term" with variables in X, the idea is to define systems of recursive equations as morphisms

$$f : X \to M\ (Y + X).$$

Assuming $X := \{x_i\}_{1 \le i \le n}$ and writing f_i for $f\ x_i$, this system is intuitively

$$\begin{aligned} s_1 &\approx f_1[x_1 \mapsto s_1, x_2 \mapsto s_2, \ldots, x_n \mapsto s_n] \\ s_2 &\approx f_2[x_1 \mapsto s_1, x_2 \mapsto s_2, \ldots, x_n \mapsto s_n] \\ &\vdots \\ s_n &\approx f_n[x_1 \mapsto s_1, x_2 \mapsto s_2, \ldots, x_n \mapsto s_n], \end{aligned}$$

where each f_i is an M-term mentioning any recursive variables $x_j : X$ or any parameter $y_j : Y$. A *solution* is then a mapping $s : X \to M\ Y$ assigning to each "unknown" in X an M-term mentioning only parameters in Y. A solution must obviously satisfy the original equation system, which in the monadic language may be stated as follows.

$$s\ x \quad \approx \quad f\ x \gg\!= \lambda \begin{bmatrix} \mathrm{inl}\ y \mapsto \mathrm{ret}\ y \\ \mathrm{inr}\ x \mapsto s\ x \end{bmatrix}$$

While the basic idea is simple, a number of subtle questions arise quite quickly during axiomatization. Should all equation systems have solutions? Should the solution be unique? If not, when some solution can be selected by an iteration operator, what coherence properties should this operator satisfy? In fact, almost all imaginable points in the design space have been explored, in an explosion of competing definitions. The concepts have historically been organized roughly as follows.

[93] Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic, "Interaction trees: representing recursive and impure programs in Coq," 2020.

[36] Calvin C. Elgot, "Monadic Computation And Iterative Algebraic Theories," 1975.

Iterative Things Every *guarded* equation system, i.e., eliminating problematic equations where some $x_i \approx x_j$, has a *unique* solution. The following variants have been defined (not all of comparable generality):

- *iterative theories*, for terms in finitary algebraic theories [36],
- *iterative algebras*, for algebras associated to such theories [76],
- *completely iterative monads*, for *ideal* monads, where there is a way to make sense of guardedness [5].
- *completely iterative algebras*, for functor algebras, with an adapted notion of *flat* equation system [6],

Absence of the prefix "completely" denotes the fact that only finitary equation systems are solved.

[36] Calvin C. Elgot, "Monadic Computation And Iterative Algebraic Theories," 1975.

[76] Evelyn Nelson, "Iterative Algebras," 1983.

[5] Peter Aczel, Jirí Adámek, Stefan Milius, and Jirí Velebil, "Infinite trees and completely iterative theories: a coalgebraic view," 2003.

[6] Jirí Adámek, Stefan Milius, and Jirí Velebil, "From Iterative Algebras to Iterative Theories," 2004.

Iteration Things and Elgot Things Every equation system has a *choice* of solution, subject to coherence conditions. The following notions have been defined:

- *iteration theories*, for terms in finitary algebraic theories [21],
- *Elgot algebras*, for finitary functor algebras [7],
- *Elgot monads*, for finitary monads [8],
- *complete Elgot monads*, for any monad [39].

The older "iteration" prefix requires only the four so-called Conway axioms on the iteration operator, while the more recent "Elgot" prefix denotes the addition of the "uniformity" axiom. The prefix "complete" has the same meaning as for iterative things.

[21] Stephen L. Bloom and Zoltán Ésik, *Iteration Theories: The Equational Logic of Iterative Processes*, 1993.

[7] Jirí Adámek, Stefan Milius, and Jirí Velebil, "Elgot Algebras," 2006.

[8] Jirí Adámek, Stefan Milius, and Jirí Velebil, "Equational properties of iterative monads," 2010.

[39] Sergey Goncharov, Christoph Rauch, and Lutz Schröder, "Unguarded Recursion on Coinductive Resumptions," 2015.

More recently, several works have tried to unify the above two families, by axiomatizing abstract *guardedness criteria*, for which guarded equations have a coherent choice of solution [40][73]. This criterion may be syntactic as in the first family, or vacuous (every equation is considered guarded) as in the second family. The iteration operator may then be axiomatized to be coherent with gradually more constraining notions of coherence. These coherences can be in the style of iteration or Elgot monads, and culminate in the strongest coherence, the *unicity* condition. For the type theory practitioner seeking a modern account, I recommend in particular Goncharov et al. [40], which also features much appreciated graphical depictions of all kinds of coherence laws.

[40] Sergey Goncharov, Lutz Schröder, Christoph Rauch, and Maciej Piróg, "Unifying Guarded and Unguarded Iteration," 2017.

[73] Stefan Milius and Tadeusz Litak, "Guard Your Daggers and Traces: Properties of Guarded (Co-)recursion," 2017.

The original InteractionTree library [93] has for now only be concerned with an iteration operator solving arbitrary systems, which can be shown to verify the (complete) Elgot monad laws w.r.t. *weak* bisimilarity. As we will show, this operator lifts without surprises to our indexed setting. However, unicity of

solution is quite a tempting principle even if it is not available for every equation system, and our OGS correctness proof will crucially depend it. As such, we will also present the iterative structure of indexed interaction trees, w.r.t. to two notions of guardedness. First, the usual simple guardedness of ideal monads [40] and second, a finer notion of *eventual* guardedness. It is to the best of our knowledge the first time that this *eventual* guardedness condition is presented, although a similar idea is present in several proofs by ADÁMEK, MILIUS and VELEBIL [8].

[40] Sergey Goncharov, Lutz Schröder, Christoph Rauch, and Maciej Piróg, "Unifying Guarded and Unguarded Iteration," 2017.

[8] Jirí Adámek, Stefan Milius, and Jirí Velebil, "Equational properties of iterative monads," 2010.

2.6.1 Unguarded Iteration

Let us start by lifting the standard unguarded iteration of interaction trees to our indexed setting.

By implementing iteration operators we are tickling the limits of what can be expressed in our metatheory using coinductive definitions. As such this section and the following will contain a couple tricky functions, which have been tailored to work well with ROCQ's syntactic guardedness checker. We try to comment on each one, but some will surely remain mysterious. Let us start with such a function.

Definition 2.67 (Exposed Copairing):
Given $\Sigma : \mathrm{Container}\ I$ define the following *exposed copairing* function.

$$\begin{aligned} &\text{"}[\_,\_]\ \{X\ Y\ Z\} : (X \nrightarrow \text{"ITree}_\Sigma\ Z) \to (Y \nrightarrow \text{"ITree}_\Sigma\ Z) \\ &\qquad\qquad \to (X + Y) \nrightarrow \mathrm{ITree}_\Sigma\ Z \\ &\text{"}[f,g]\ r := \left[\begin{array}{l} \mathrm{out} := \mathrm{case}\ r \\ \left[\begin{array}{l} \mathrm{inl}\ x := f\ x \\ \mathrm{inr}\ y := g\ y \end{array}\right] \end{array}\right] \end{aligned}$$

Remark 2.68: Note the exposed interaction trees in the codomain of f and g above! Moreover, we do not directly case-split on r which would define the function with two clauses. Instead, the idea is to be *lazy* on the argument r and only inspect it when the tree is observed, i.e., below out. Indeed, a general trick to help satisfy guardedness is to copattern-match on out as early as possible, i.e., use maximally lazy definitions.

This helper is enough to provide a clean definition of unguarded iteration.

Definition 2.69 (Interaction Tree Iteration):
Let $\Sigma : \mathrm{Container}\ I$. Given an equation $f : X \nrightarrow \mathrm{ITree}_\Sigma\ (Y + X)$, define its *iteration* coinductively as follows.

$$\mathrm{iter}_f : X \to \mathrm{ITree}_\Sigma\ Y$$
$$\mathrm{iter}_f\ x := f\ x \gg\!= \text{"}[\text{"ret}, \text{"tau} \circ \mathrm{iter}_f]$$

Remark 2.70: It is quite a miracle that this coinductive definition is accepted. In Rocq, it depends on two crucial tricks we have seen: the lazy copairing and the bind operator defined with the continuation bound outside of the coinductive definition. If either is skipped, the above definition is rejected. The more robust way to define unguarded iteration would be to specialize the bind operator, generalizing the above definition to

$$\text{iter-aux}_f : \mathrm{ITree}_\Sigma\ (Y + X) \to \mathrm{ITree}_\Sigma\ Y$$

and then defining $\mathrm{iter}_f\ x := \text{iter-aux}_f\ (f\ x)$. This would however require proving more properties, relating this specialized bind operator to the usual one.

Lemma 2.71 (Iter Fixed Point):
Given $\Sigma : \mathrm{Container}\ I$, for all $f : X \to \mathrm{ITree}_\Sigma\ (Y + X)$, iter_f is a weak fixed point of f, i.e., the following holds.

$$\mathrm{iter}_f\ x \approx f\ x \gg\!= \lambda \begin{bmatrix} \mathrm{inl}\ y \mapsto \mathrm{ret}\ y \\ \mathrm{inr}\ x \mapsto \mathrm{iter}_f\ x \end{bmatrix}$$

Proof: By definition $\mathrm{iter}_f\ x := f\ x \gg\!= \text{"}[\text{"ret}, \text{"tau} \circ \mathrm{iter}_f]$. As such, both sides start with the same computation $f\ x$. By Lemma 2.61 it suffices to prove that the continuations are pointwise weakly bisimilar. Introduce $r : Y + X$ and proceed by one-step unfolding (Lemma 2.35 (1)).

- For $\mathrm{inl}\ y$, conclude by "ret^r.
- For $\mathrm{inr}\ x$, eat the "tau on the left on conclude by reflexivity. ■

Furthermore, we prove the following monotonicity statement for iteration.

Lemma 2.72 (Iter Monotonicity):
Given $\Sigma : \mathrm{Container}\ I$, for all $X^r : \mathrm{IRel}\ X^1\ X^2$ and $Y^r : \mathrm{IRel}\ Y^1\ Y^2$, the following statements holds.

(1) $\mathrm{iter}\ \langle\!\langle (\forall^r X^r \to^r \cong[\ Y^r +^r\ X^r\ ]) \to^r (\forall^r X^r \to^r \cong[\ Y^r\ ]) \rangle\!\rangle\ \mathrm{iter}$

(2) $\mathrm{iter}\ \langle\!\langle (\forall^r X^r \to^r \approx[\ Y^r +^r\ X^r\ ]) \to^r (\forall^r X^r \to^r \approx[\ Y^r\ ]) \rangle\!\rangle\ \mathrm{iter}$

Proof: The proof is by straightforward tower induction. Let us detail it for strong bisimilarity. By tower induction, let us prove that for all $\mathcal{F} \in \mathrm{Tower}_{\mathrm{sbisim}_\Sigma}$ the following holds.

$$\mathrm{iter}\ \langle\!\langle (\forall^r X^r \to^r \mathcal{F}\ (Y^r +^r X^r)) \to^r (\forall^r X^r \to^r \mathcal{F}\ Y^r) \rangle\!\rangle\ \mathrm{iter}$$

The statement is inf-closed. Assuming the statement for $\mathcal{F}$, let us prove it for $\mathrm{sbisim}_\Sigma\ \mathcal{F}$. Introduce the hypotheses

$$f^r : f^1\ \langle\!\langle \forall^r X^r \to^r \mathrm{sbisim}_\Sigma\ \mathcal{F}\ (Y^r +^r X^r) \rangle\!\rangle\ f^2$$
$$x^r : X^r\ x^1\ x^2.$$

We need to prove

$$\mathrm{iter}\ f^1\ x^1\ \langle\!\langle \mathrm{sbisim}_\Sigma\ \mathcal{F}\ Y^r \rangle\!\rangle\ \mathrm{iter}\ f^2\ x^2$$

By definition, both sides are given by binds, respectively starting by $f^1\ x^1$ and $f^2\ x^2$. By x^r applied to f^r, these are related suitably, so that by Lemma 2.61 it now suffices to relate the continutations pointwise.

- For $\mathrm{inl}\ y$, conclude by $^\backprime\mathrm{ret}^r$ and reflexivity on y.
- For $\mathrm{inr}\ x$, conclude by $^\backprime\mathrm{tau}^r$ and coinduction hypothesis. ■

We will not prove here that this iteration operator satisfies the requirements of complete ELGOT monads. These properties could be useful for reasoning about interaction trees constructed by iteration, but they are quite limited compared to something such as uniqueness of solutions. The prime shortcoming of these coherence properties, is that they are limited to rearranging equation systems. As such, they are hardly useful to establish bisimilarity between an interaction tree constructed by iteration and another one, constructed entirely differently. Because such a bisimilarity proof will be at the cornerstone of our OGS correctness proof, we need to look further into guardedness, the key to uniqueness of solutions.

2.6.2 Guarded Iteration

A general trend in the research on iteration operators is the observation that, very often, the unguarded iteration operator of, e.g., an ELGOT monad, may be shown to somehow derive from an underlying *guarded* iteration operator enjoying unique fixed points, with the former ELGOT monad typically being a quotient of the latter iterative monad. With interaction trees, we find ourselves exactly in this situation. Indeed, as we will see, every equation system is weakly bisimilar to a guarded equation system. Our previous unguarded iteration operator can then be recast as constructing the unique fixed point of this new guarded equation system, up to strong bisimilarity*. Without further ado, let us define this guarded iteration operator.

Definition 2.73 (Guardedness):
Let $\Sigma : \mathrm{Container}\ I$. An action is *guarded in* X if it satisfies the predicate

$$\mathrm{data}\ ^\backprime\mathrm{guarded}\ \{X\ Y\ A\ i\} : \mathrm{Action}_\Sigma\ (Y + X)\ A\ i \to \mathrm{Prop}$$

* In hindsight, it should not come as a surprise that primitively only guarded and unique fixed points exist. This is what coinduction in our metatheory provides us with. Because we work in a constructive metatheory there is no place for doubt, and to be accepted, any definition must precisely, i.e., *uniquely*, pinpoint some semantical object. It is rather tautological that any definition must indeed define something!

defined by the following constructors.

$$\frac{}{\text{"ret}^g : \text{"guarded ("ret (inl } y))}$$

$$\frac{}{\text{"tau}^g : \text{"guarded ("tau } t)} \qquad \frac{}{\text{"vis}^g : \text{"guarded ("vis } q\ k)}$$

Furthermore, an interaction tree is *guarded in* X if its observation is:

$$\text{guarded } \{X\ Y\ i\} : \text{ITree}_\Sigma\ (Y + X)\ i \to \text{Prop}$$
$$\text{guarded } t := \text{"guarded } t.\text{out} \ .$$

And, finally, guardedness of equations is defined pointwise:

$$\text{eqn-guarded } \{X\ Y\} : (X \to \text{ITree}_\Sigma\ (Y + X)) \to \text{Prop}$$
$$\text{eqn-guarded } f := \forall\ \{i\}\ (x : X\ i) \to \text{guarded } (f\ x).$$

Before even constructing them, we can prove that fixed points of guarded equations w.r.t. strong bisimilarity are unique.

Lemma 2.74 (Unique Guarded Fixed Points):
Given $\Sigma : \text{Container}\ I$ and a guarded equation $f : X \to \text{ITree}_\Sigma\ (Y + X)$, for any two fixed points s_1, s_2 of f w.r.t. strong bisimilarity, i.e., such that for all x and $i = 1, 2$ we have

$$s_i\ x \cong f\ x \gg\!= \lambda \begin{bmatrix} \text{inl } y \mapsto \text{ret } y \\ \text{inr } x \mapsto s_i\ x \end{bmatrix},$$

then for all x, $s_1\ x \cong s_2\ x$.

Proof: By tower induction, assume a strong bisimulation candidate $\mathcal{F}$ such that

$$\forall\ x \to \mathcal{F}\ \Delta^r\ (s_1\ x)\ (s_2\ x)$$

and introduce the argument x. After rewriting (using up-to transitivity, Lemma 2.46) by both fixed point hypotheses, the goal is now to prove

$$\left(f\ x \gg\!= \lambda \begin{bmatrix} \text{inl } y \mapsto \text{ret } y \\ \text{inr } x \mapsto s_1\ x \end{bmatrix}\right)$$
$$\langle\!\langle \text{sbisim}_\Sigma\ \mathcal{F}\ \Delta^r \rangle\!\rangle$$
$$\left(f\ x \gg\!= \lambda \begin{bmatrix} \text{inl } y \mapsto \text{ret } y \\ \text{inr } x \mapsto s_2\ x \end{bmatrix}\right)$$

By inspecting the first step of $f\ x$, by guardedness we obtain a synchronization and it now suffices to prove that for some tree t the following holds.

$$\left(t \gg\!= \lambda\left[\begin{array}{l}\text{inl } y \mapsto \text{ret } y\\ \text{inr } x \mapsto s_1\ x\end{array}\right]\right) \langle\!\langle \mathcal{F}\ \Delta^r \rangle\!\rangle \left(t \gg\!= \lambda\left[\begin{array}{l}\text{inl } y \mapsto \text{ret } y\\ \text{inr } x \mapsto s_2\ x\end{array}\right]\right)$$

Conclude by up-to-bind (Lemma 2.61) and case analysis, with the first case proven by reflexivity and the second by coinduction hypothesis. ■

Let us now construct this unique fixed point.

Definition 2.75 (Guarded Iteration):
Let $\Sigma : \text{Container}\ I$. Given an equation $f : X \to \text{ITree}_\Sigma\ (Y + X)$ with guardedness witness $H : \text{eqn-guarded}\ f$, first define the following *guarded unfolding function*.

$$\text{g-step}_{f,H} : (\text{ITree}_\Sigma\ (Y + X) \to \text{ITree}_\Sigma\ Y) \to (X \to {}^{\prime}\text{ITree}_\Sigma\ Y)$$

$$\text{g-step}_{f,H}\ g\ x := \text{case}\ (f\ x).\text{out} \mid H\ x$$

$$\left[\begin{array}{lll}({}^{\prime}\text{ret}\ (\text{inl}\ y)) & p & := {}^{\prime}\text{ret}\ y\\ ({}^{\prime}\text{ret}\ (\text{inr}\ x)) & (!) & \\ ({}^{\prime}\text{tau}\ t) & p & := {}^{\prime}\text{tau}\ (g\ t)\\ ({}^{\prime}\text{vis}\ q\ k) & p & := {}^{\prime}\text{vis}\ q\ (\lambda\ r \mapsto g\ (k\ r))\end{array}\right]$$

We then define the following coinductive auxiliary function.

$$\text{g-iter}^{\text{aux}}_{f,H} : \text{ITree}_\Sigma\ (Y + X) \to \text{ITree}_\Sigma\ Y$$

$$\text{g-iter}^{\text{aux}}_{f,H}\ t := t \gg\!= {}^{\prime}[{}^{\prime}\text{ret}, \text{g-step}_{f,H}\ \text{g-iter}^{\text{aux}}_{f,H}]$$

Finally, we define the *guarded iteration* as follows.

$$\text{g-iter}_{f,H} : X \to \text{ITree}_\Sigma\ Y$$

$$\text{g-iter}_{f,H}\ x := \text{g-iter}^{\text{aux}}_{f,H}\ (f\ x)$$

We will omit the guardedness witness H when clear from context.

Remark 2.76: While a bit scary, the above definition of g-step is simply mimicking the first step of a "bind" on $f\ x$, delegating the rest of the work to its argument g. Thanks to the added information from the guardedness witness, it is able to only trigger subsequent computation in a guarded fashion. This can be seen from its type, as it returns an exposed interaction tree. Recall that for unguarded iteration, this same guardedness was achieved artificially, by wrapping the whole call in a silent step.

Because of this one step of observing on $f\ x$, the subsequent computation will be triggered on an arbitrary $t : \text{ITree}_\Sigma\ (Y + X)\ i$, instead of something of the form $f\ x$. Hence the coinductive knot must be tied with such a generalized argument, as is done in $\text{g-iter}^{\text{aux}}$.

Theorem 2.77 (Guarded Fixed Point):
Let $\Sigma : \mathrm{Container}\ I$. For any guarded equation $f : X \to \mathrm{ITree}_\Sigma\ (Y + X)$, g-iter_f is the unique fixed point of f w.r.t. strong bisimilarity.

Proof: Since by Lemma 2.74 guarded fixed points are unique w.r.t. strong bisimilarity, it suffices to show that it is indeed a fixed point, i.e.,

$$\text{g-iter}_f\ x \cong f\ x \gg\!= \lambda \begin{bmatrix} \text{inl}\ y \mapsto \text{ret}\ y \\ \text{inr}\ x \mapsto \text{g-iter}_f\ x \end{bmatrix}$$

As both sides start by binding $f\ x$, by Lemma 2.61 it suffices to relate the continuations pointwise. Introduce the argument r and proceed by one-step unfolding (Lemma 2.35 (1)), then splitting on r. For $r := \text{inl}\ y$, conclude by "ret^r. For $r := \text{inr}\ x$, we need to prove the following.

$$\left(\text{g-step}_f\ \text{g-iter}_f^{\text{aux}}\ x\right).\text{out} \overset{s}{\cong} \left(\text{g-iter}_f\ x\right).\text{out}$$

Both sides can be seen to start by observing $(f\ x).\text{out}$ so let us case split.

- $\text{"ret}\ (\text{inl}\ y)$ Conclude by "ret^r and reflexivity.
- $\text{"ret}\ (\text{inr}\ x)$ Absurd by guardedness hypothesis $H\ x$.
- $\text{"tau}\ t$ Conclude by "tau^r and reflexivity.
- $\text{"vis}\ q\ k$ Conclude by "vis^r and pointwise reflexivity. ■

We have thus exhibited interaction trees (considered up to strong bisimilarity) as a completely iterative monad. Let us now link this to unguarded iteration.

Lemma 2.78:
Given $\Sigma : \mathrm{Container}\ I$ and an equation $f : X \to \mathrm{ITree}_\Sigma\ (Y + X)$

(1) Assuming f is guarded, for all x, $\text{g-iter}_f\ x \approx \text{iter}_f\ x$.
(2) Let $f'\ x := f \gg\!= (\text{"tau} \boxplus \text{"ret})$. Then, f' is guarded and for all x,

$$\text{iter}_f\ x \cong \text{g-iter}_{f'}\ x.$$

Proof:
(1) By tower induction, assume a weak bisimulation candidate $\mathcal{F}$ such that the statement holds and let us prove it for $\text{wbisim}_\Sigma\ \mathcal{F}$. Introduce x and rewrite the LHS by the strong fixed point property so that the goal is now

$$\left(f\ x \gg\!= \lambda \begin{bmatrix} \text{inl}\ y \mapsto \text{ret}\ y \\ \text{inr}\ x \mapsto \text{g-iter}_f\ x \end{bmatrix}\right)$$
$$\langle\!\langle \text{wbisim}_\Sigma\ \mathcal{F}\ \Delta^r \rangle\!\rangle$$
$$\left(f\ x \gg\!= \text{"}[\text{"ret}, \text{"tau} \circ \text{iter}_f]\right)$$

Both sides start by observing $f\ x$. By case analysis, refute the problematic case using the guardedness hypothesis on f and exhibit a synchronization guard ("ret^r, "tau^r or "vis^r) in the other cases. Both sides still continue to start by the same computation, so that by Lemma 2.61 it suffices to relate the continuations pointwise w.r.t. $\mathcal{F}$. For inl y conclude by reflexivity. For inr x, eat the "tau on the right and conclude by coinduction hypothesis.

(2) For any x, by case analysis on $(f\ x).\text{out}$ we can show that $f'\ x$ is guarded. Then, by direct tower induction, following approximately the same proof pattern as (1), without eating the "tau at the end. ■

2.6.3 Eventually Guarded Iteration

Equipped with this new guarded iteration, we finally obtain our powerful uniqueness of fixed points. This principle will provide us with a big hammer, very useful for hitting nails looking like $\text{g-iter}_f\ x \approx t\ x$. However, being guarded is quite a strong requirement! Notably, our equation of interest in this thesis, the one defining the composition of OGS strategies and counter-strategies, has no hope of being guarded. However, observe that if the equation contains a finite chain

$$\begin{aligned} x_1 &\mapsto \text{ret}\ (\text{inr}\ x_2) \\ x_2 &\mapsto \text{ret}\ (\text{inr}\ x_3) \\ &\vdots \\ x_n &\mapsto t, \end{aligned}$$

such that t is guarded, then after unfolding the equation n times, x_1 will be mapped to a guarded tree t. The iteration starting from x_1 is then still uniquely defined. This was already noted by ADÁMEK, MILIUS and VELEBIL [8] with their notion of *grounded* variables. However, a clear definition and study of equations containing only grounded variables, or *eventually guarded equations* as we call them, is still novel to the best of our knowledge. In fact, in future work it might be fruitful to consider this in the setting of GONCHAROV et al. [40], as a generic relaxation of any abstract guardedness criterion.

[8] Jirí Adámek, Stefan Milius, and Jirí Velebil, "Equational properties of iterative monads," 2010.

[40] Sergey Goncharov, Lutz Schröder, Christoph Rauch, and Maciej Piróg, "Unifying Guarded and Unguarded Iteration," 2017.

Definition 2.79 (Eventual Guardedness):
Let $\Sigma : \text{Container}\ I$ and $f : X \to \text{ITree}_\Sigma\ (Y + X)$. An interaction tree is *eventually guarded w.r.t.* f if it verifies the inductive predicate

```
data ev-guarded_f {i} : "ITree_Σ (Y + X) i → Prop
```

defined by mutually with the following shorthand

```
ev-guarded_f {i} : ITree_Σ (Y + X) i → Prop
ev-guarded_f t := ~ev-guarded_f t.out
```

and whose constructors are given as follows.

$$\frac{p : \text{~guarded } t}{\text{~ev-guard } p : \text{~ev-guarded}_f\ t} \qquad \frac{p : \text{ev-guarded}_f\ (f\ x)}{\text{~ev-step } p : \text{~ev-guarded}_f\ (\text{~ret } (\text{inr } x))}$$

An equation is *eventually guarded* if it is pointwise eventually guarded w.r.t. itself.

```
eqn-ev-guarded {X Y} : (X ↛ ITree_Σ (Y + X)) → Prop
eqn-ev-guarded f := ∀ {i} (x : X i) → ev-guarded_f (f x)
```

As for guardedness, we can show unicity of fixed points w.r.t. strong bisimilarity of eventually guarded equations.

Lemma 2.80 (Uniqueness of Eventually Guarded Fixed Points):
Given $\Sigma : \text{Container } I$ and $f : X \nrightarrow \text{ITree}_\Sigma\ (Y + X)$ such that f is eventually guarded, for any fixed points g and h of f w.r.t. strong bisimilarity, for all x, we have $g\ x \approx h\ x$.

Proof: By tower induction, then by induction on the eventual guardedness proof, repeatedly rewriting both sides by the fixed point equation to exhibit a synchronization point. ■

To construct an eventually guarded fixed point, we reduce the problem to computing a guarded fixed point. Indeed, any eventually guarded equation can be pointwise *unrolled* into a guarded one.

Definition 2.81 (Unrolling):
Let $\Sigma : \text{Container } I$. Given $f : X \nrightarrow \text{ITree}_\Sigma\ (Y + X)$ and eventually guarded t w.r.t. f, define the *unrolling of* t as the following inductive definition.

```
unroll_f {i} (t : ~ITree_Σ (X + Y) i) : ~ev-guarded_f t → ~ITree_Σ (X + Y) i
unroll_f (~ret (inl x)) (~ev-step p) := unroll_f (f x).out p
unroll_f (~ret (inr y)) p            := ~ret (inr y)
unroll_f (~tau t)       p            := ~tau t
unroll_f (~vis q k)     p            := ~vis q k
```

Moreover, define the following *pointwise unrolling* of f.

$$\text{eq-unroll}_f : \text{eqn-ev-guarded}\ f \to (X \to \text{ITree}_\Sigma\ (Y + X))$$
$$\text{eq-unroll}_f\ H\ x \coloneqq \left[\ \text{out} \coloneqq \text{unroll}_f\ (f\ x).\text{out}\ (H\ x)\right.$$

Lemma 2.82 (Unroll Guarded):
Given $\Sigma : \text{Container}\ I$ and $f : X \to \text{ITree}_\Sigma\ (Y + X)$ such that $H : \text{eqn-ev-guarded}\ f$, then $\text{eq-unroll}_f\ H$ is guarded.

Proof: By direct induction. ■

We can now define our candidate fixed point by using the guarded iteration.

Definition 2.83 (Eventually Guarded Iteration):
Given $\Sigma : \text{Container}\ I$ and $f : X \to \text{ITree}_\Sigma\ (Y + X)$ such that $H : \text{eqn-ev-guarded}\ f$, define the *eventually guarded iteration of* f as follows.

$$\text{ev-iter}_{f,H} : X \Rightarrow \text{ITree}_\Sigma\ Y$$
$$\text{ev-iter}_{f,H} \coloneqq \text{g-iter}_{\text{eq-unroll}\ f\ H}$$

It now remains to verify that this construction is indeed a fixed point of f, as for now we only know that it is a fixed point of the unrolled equation.

Theorem 2.84 (Eventually Guarded Fixed Point):
Given $\Sigma : \text{Container}\ I$ and $f : X \to \text{ITree}_\Sigma\ (Y + X)$ such that f is eventually guarded, then ev-iter_f is the unique fixed point of f w.r.t. strong bisimilarity.

Proof: By Lemma 2.80, eventually guarded fixed points are unique w.r.t. pointwise strong bisimilarity, so it suffices to prove that ev-iter_f is a fixed point of f. This is proven by induction on the eventual guardedness witness and repeated one-step unfolding on both sides, appealing to the guarded fixed point property on the base case. ■

Once again, we further relate the eventually guarded iteration of an equation with its unguarded iteration.

Lemma 2.85:
Given $\Sigma : \text{Container}\ I$ and an eventually guarded equation $f : X \to \text{ITree}_\Sigma\ (Y + X)$, then for all x,

$$\text{ev-iter}_f\ x \approx \text{iter}_f\ x$$

Proof: By direct tower induction, then by induction on the eventually guardedness witness. For the step case, eat the ˜tau on the right and conclude by induction hypothesis. For the base case proceed by analysis of the guardedness witness, exhibiting a synchronization point, then further eat the ˜tau on the right and conclude by coinduction hypothesis. ■

This concludes not only our study of iteration operators, but this whole chapter. We now have enough base theory on games and strategies, both represented as transition systems or as indexed interaction trees. Our last theorem, unicity of eventually guarded equations, will provide us with the core coinduction proof technique to obtain Ogs correctness. Although we will stop referring to them at every single step, the bunch of compositional reasoning principles proven throughout the last few section will also be instrumental in achieving flexible reasoning on complex game strategies.

3 A Categorical Treatment of Substitution

Our generic Ogs construction depends mainly on two broad technical fields: games and programming language syntax. Since we clarified games in the previous chapter, it is now left to provide the same technical grounding for syntax. While syntax might seem like already well-known, the details of working formally with syntactic objects are more subtle than they seem.

Being close topics, programming languages and type systems are routinely studied by type theory practitioners. As such, the concrete matter of how to encode and work with syntactic terms of an object language inside type theory has been widely researched, for example in the various submissions to the first PoplMark challenge [18]. There are two main design points: how to represent variables and bindings, and how to enforce typing. My inclination towards correct-by-construction programming, that is, enforcing as much invariants as possible inside data structures using dependent typing, makes it a natural choice to use the *type- and scope-safe*, or *intrinsically typed and scoped* representation of syntax. Quoting Fiore and Szamozvancev [37],

[18] Brian E. Aydemir *et al.*, "Mechanized Metatheory for the Masses: The PoplMark Challenge," 2005.

[37] Marcelo Fiore and Dmitrij Szamozvancev, "Formal metatheory of second-order abstract syntax," 2022.

> "We believe that the nameless, intrinsic representation is hard to surpass in dependently-typed proof assistants thanks to its static guarantees on the typing and scoping of terms."

In this setting, the sort of terms is indexed both by a scope (or typing context) and by a type, so as to form a family $\mathsf{Term} : \mathsf{Scope} \to \mathsf{Ty} \to \mathsf{Type}$. This indexing may then be used to enforce that only well-typed terms are represented and that all the mentioned variables are in scope.

An important specificity of the point of view we will adopt in this chapter is to be completely silent on the actual construction of the syntax. Indeed, as our goal is to formalize a (reasonably) language-generic Ogs construction, we will only be interested in *specifying* what operations a syntax should support and leave open the choice for actual *instantiation*. Crucially, we will not assume any kind of induction principle and keep the syntax opaque. Surely it can be debated whether or not something which is not inductively defined deserves to be called a syntax, and indeed our generic Ogs construction could very well be instantiated not by syntax but by some other denotational model of a language. However, for clarity, we will keep using syntactic terminology.

We start in §3.1 with a short informal overview of the core points of the intrinsic representation and the axiomatization of substitution. This overview largely fol-

lows two comprehensive papers with beautifully crafted Agda implementations which I heartily recommend reading [37][13]. The main point is the introduction of *substitution monoids*, which axiomatize type- and scope-indexed families supporting variables and substitutions. In §3.2, we motivate and introduce a small contribution. In typical type theory formalizations, the notion of scope is fixed in the abstract theory of substitution and defined as lists of types, but this rigidity can be cumbersome for some languages. This is remedied by providing a simple abstraction for scopes. Finally, in §3.3, we adapt the definition of substitution monoid to this new abstraction. We also present *substitution modules*, a notion required to deal with syntactic objects which have a substitution operation, but whose arguments may be of a different kind, such as evaluation contexts, whose variables can be substituted by values. Substitution modules have already been studied [44], but they are rarely presented even though they become necessary quite quickly in lots of concrete examples.

[37] Marcelo Fiore and Dmitrij Szamozvancev, "Formal metatheory of second-order abstract syntax," 2022.
[13] Guillaume Allais, Robert Atkey, James Chapman, Conor McBride, and James McKinna, "A type- and scope-safe universe of syntaxes with binding: their semantics and proofs," 2021.

[44] André Hirschowitz and Marco Maggesi, "Modules over monads and initial semantics," 2010.

3.1 The Theory of Intrinsically-Typed Substitution

Remark 3.1: The following section will consist in an informal overview. As such every introduced notion will be properly defined later on.

Contexts The type- and scope-safe approach starts by defining typing contexts as lists of types (written backwards, for consistency with the traditional notation of sequents) and variables as dependently typed De Bruijn indices, in other words, proof-relevant membership witnesses:

$$\begin{aligned}
&\mathsf{data}\ \mathsf{Ctx}\ (T : \mathsf{Type}) : \mathsf{Type} :=\\
&\quad \left[\begin{array}{l} \varepsilon : \mathsf{Ctx}\ T \\ _ \blacktriangleright _ : \mathsf{Ctx}\ T \to T \to \mathsf{Ctx}\ T \end{array}\right.\\
&\mathsf{data}\ _ \ni _\ \{T\} : \mathsf{Ctx}\ T \to T \to \mathsf{Type} :=\\
&\quad \left[\begin{array}{l} \mathsf{top}\ \{\Gamma\ \alpha\} : (\Gamma \blacktriangleright \alpha) \ni \alpha \\ \mathsf{pop}\ \{\Gamma\ \alpha\ \beta\} : \Gamma \ni \alpha \to (\Gamma \blacktriangleright \beta) \ni \alpha \end{array}\right.
\end{aligned}$$

The main category of interest for syntactic objects is given by *scoped-and-typed families* $\mathsf{Ctx}\ T \to T \to \mathsf{Type}$. Variables given by $\_ \ni \_$ are already such a family, but so too will be terms, and more generally "things by which variables can be substituted".

Next, renamings are defined as type-preserving mappings from variables in one context to variables in another, turning the set $\mathsf{Ctx}\ T$ into the category $\mathcal{C}\ T$. Instead of just variables, the codomain of these renamings can be generalized to any scoped-and-typed family X, yielding the notion of assignment. More

precisely, given two contexts $\Gamma, \Delta : \mathrm{Ctx}\ T$, an *$X$-assignment from Γ to Δ* is a type-preserving mapping from variables over Γ to X-terms over Δ.

$$\Gamma \dashv[X]\mapsto \Delta := \forall\ \{\alpha\} \to \Gamma \ni \alpha \to X\ \Delta\ \alpha$$
$$\Gamma \subseteq \Delta := \Gamma \dashv[\ni]\mapsto \Delta$$

Remark 3.2: As contexts are finite, assignments are maps with finite domain, and may thus be tabulated and represented as tuples. The tuple representation makes intensional equality of assignments better behaved. However as other parts of my Rocq development already depend on extensional equality, I will not shy away from the pointwise extensional equality of functions.

Although it does have this issue, the representation of assignments as functions has other benefits. Thanks to the η-law renamings as defined above construct a *strict* category, where all the laws hold w.r.t. *definitional* equality. This would be lost when working with tabulated representations.

Remark 3.3: When computational efficiency is a concern, another typical choice is to define a more economic subcategory of contexts, whose renamings consist only of order-preserving embeddings (Ope). An Ope can be computationally modeled by a bitvector, where a 0 at position i means that the i-th variable is dropped while 1 means that it is kept. However as computational efficiency is not our prime concern, we will not go down this route. Still, we keep this idea around, borrowing and slightly abusing the notation $\subseteq$ for renamings, which is traditionally associated with embeddings.

Substitution Monoids It is now direct to express that a given scoped-and-typed family X has variables and substitution. First, variables must map into X. Second, given an X-term over Γ and an X-assignment from Γ to Δ, substitution should return some X-term over Δ. In other words, X must admit maps of the following types.

$$\mathrm{var}\ \{\Gamma\ \alpha\} : \Gamma \ni \alpha \to X\ \Gamma\ \alpha$$
$$\mathrm{sub}\ \{\Gamma\ \Delta\ \alpha\} : X\ \Gamma\ \alpha \to (\Gamma \dashv[X]\mapsto \Delta) \to X\ \Delta\ \alpha$$

This structure is dubbed a *substitution monoid* [37] and is further subject to the usual associativity and left and right identity laws.

[37] Marcelo Fiore and Dmitrij Szamozvancev, "Formal metatheory of second-order abstract syntax," 2022.

$$\mathrm{sub}\ (\mathrm{var}\ i)\ \gamma = \gamma\ i$$
$$\mathrm{sub}\ x\ \mathrm{var} = x$$
$$\mathrm{sub}\ (\mathrm{sub}\ x\ \gamma)\ \delta = \mathrm{sub}\ x\ (\lambda\ i \mapsto \mathrm{sub}\ (\gamma\ i)\ \delta)$$

To explain how these two maps can be seen as the unit and multiplication maps of a monoid, notice that their types may be refactored as

$$\mathsf{var} : \_ \ni \_ \to X$$
$$\mathsf{sub} : X \to [\![X,X]\!]$$

where $[\![\_,\_]\!]$ is the following *internal substitution hom* functor [37].

[37] Marcelo Fiore and Dmitrij Szamozvancev, "Formal metatheory of second-order abstract syntax," 2022.

$$[\![X,Y]\!]\ \Gamma\ \alpha := \forall\ \{\Delta\} \to \Gamma \dashv [X] \mapsto \Delta \to Y\ \Delta\ \alpha$$

Further note that for any scoped-and-typed family X, the functor $[\![X,\_]\!]$ has a left adjoint $\_ \odot X$, the *substitution tensor product* [37], given by

$$(X \odot Y)\ \Gamma\ \alpha := (\Delta : \mathsf{Ctx}\ T) \times X\ \Delta\ \alpha \times \Delta \dashv [Y] \mapsto \Gamma.$$

[14] Thorsten Altenkirch, James Chapman, and Tarmo Uustalu, "Monads Need Not Be Endofunctors," 2010.
[89] Kornel Szlachányi, "Skew-monoidal categories and bialgebroids," 2012.

This substitution tensor exhibits scoped-and-typed families as a *skew monoidal category* [14][89] with unit $\ni$. Being *skew* monoidal is slightly weaker than monoidal, as the left and right unit laws as well as associativity laws on the tensor product are not true w.r.t. isomorphisms, but w.r.t. morphisms in one direction. By adjointness, the substitution map could be alternatively written with the isomorphic type

$$\mathsf{sub} : X \odot X \to X.$$

Thus, although we prefer using the internal substitution hom presentation which gives a much more easily manipulated *curried* function type to substitution, from a mathematical point of view, substitution monoids are precisely monoid objects in the skew monoidal category $(\mathsf{Ctx}\ T \to T \to \mathsf{Type}, \ni, \odot)$.

Renamings Let us finish this overview of the state of the art with a notable recent insight on the type-theoretical presentation of the operation of *renaming*.

In the categorical approach, it seems particularly obvious to formalize that a family $X : \mathsf{Ctx}\ T \to T \to \mathsf{Type}$ supports renamings if it is functorial in the first argument, i.e., if it extends to a functor $\mathcal{C}\ T \to T \to \mathsf{Type}$. In fact, as is customary in category-theoretic presentations, all of the above theory can be recast in the functor category, entirely eliminating families *not* supporting renamings. However, as shown by folklore practice in the dependently-typed community, and stressed by Fiore and Szamozvancev [37], working solely in the functor category is problematic as it crucially requires to work with quotients. Essentially, the reason is that when constructing the syntax, one can *implement* renamings by induction on terms. This implementation does not appeal to the automatic renaming operation that exists by virtue of working only with functors. As such, we are left with two renaming operations, and quotients are necessary to make sure they coincide.

[37] Marcelo Fiore and Dmitrij Szamozvancev, "Formal metatheory of second-order abstract syntax," 2022.

The trick to provide a theoretical account of the renaming operation while avoiding functors is to notice that the faithful functor

$$(\mathcal{C}\ T \to T \to \mathrm{Type}) \to (\mathrm{Ctx}\ T \to T \to \mathrm{Type})$$

is comonadic, with associated comonad given by $\Box X := [\![\ni, X]\!]$, i.e.,

$$\Box X\ \Gamma\ \alpha := \forall\{\Delta\} \to (\Gamma \subseteq \Delta) \to X\ \Delta\ \alpha.$$

In plain words, families supporting renamings, i.e., functors $\mathcal{C}\ T \to T \to \mathrm{Type}$ can be equivalently seen as families with a $\Box$-coalgebra structure [13][37]. This coalgebra structure exactly gives the renaming map, expressing it as

$$\mathrm{ren} : X \to \Box X.$$

This is obviously an after-the-fact theoretical reconstruction of the familiar operation of renaming and indeed matches the obvious type

$$\mathrm{ren}\ \{\Gamma\ \Delta\ \alpha\} : X\ \Gamma\ \alpha \to \Gamma \subseteq \Delta \to X\ \Delta\ \alpha.$$

Every substitution monoid induces such a renaming coalgebra structure since renamings can be implemented by substitution with variables. However, the typical implementation of substitution on a syntax with binders requires readily available ren and var operations to allow substitution to go under binders. This package of renamings and variables can be formalized as a *pointed coalgebra structure* [37] and its compatibility conditions with a substitution monoid structure are straightforward.

[13] Guillaume Allais, Robert Atkey, James Chapman, Conor McBride, and James McKinna, "A type- and scope-safe universe of syntaxes with binding: their semantics and proofs," 2021.

[37] Marcelo Fiore and Dmitrij Szamozvancev, "Formal metatheory of second-order abstract syntax," 2022.

3.2 What is a Variable? Abstracting De Bruijn Indices

While theoretically a sound choice, defining contexts as lists of types and variables as De Bruijn indices is practically unsatisfactory. Perhaps the most convincing reason is that storing sequences as singly-linked lists and membership proofs as unary numbers is not computationally efficient. When efficient execution is a concern, one typically chooses an off-the-shelf finite map data structure such as binary trees, which enjoy logarithmic time lookup and logarithmic size membership proofs.

Although I like to imagine that my Rocq development makes sound computational choices, I must admit that I have not yet been truly serious about efficiency. But there is a more type-theoretical objection to lists and De Bruijn indices: while all free monoid constructions are isomorphic (extensionally equal) to lists, there are situations where some are much more practical to manipulate than others.

The prime example is the following setting. We have a set of types T and we construct some syntax $\mathrm{Term} : \mathrm{Ctx}\ T \to T \to \mathrm{Type}$. Now for some reason, we

have a subset ✨ : $T \to$ Prop of let's say, *nice* types, and we need to work with the sub-syntax of terms in *nice* contexts, that is in contexts containing only nice types. Assuming we have worked out the theory of substitution for bare terms, we want to lift it to the nice setting*.

* This situation is not entirely artificial and does in fact appear routinely in Ogs instances. Indeed, the scopes tracking the shared variables of both players are usually restricted to contain only the types of some kind of non-transmitted values, typically called *negative types*.

In the framework of lists and De Bruijn indices, we must define nice contexts as lists of pairs of a type together with a proof that it is nice:

$$T^✨ := (\alpha : T) \times ✨\ \alpha$$
$$\text{Ctx}^✨\ T := \text{Ctx}\ T^✨$$

To lift the syntax into a *nice* syntax $\text{Term}^✨ : \text{Ctx}^✨\ T \to T^✨ \to$ Type, we set

$$\text{Term}^✨\ \Gamma\ \alpha := \text{Term}\ \downarrow\Gamma\ \downarrow\alpha$$

where $\downarrow$ is overloaded both as $\text{Ctx}^✨\ T \to \text{Ctx}\ T$ and as $T^✨ \to T$, *downgrading* nice things to their underlying bare object.

Assuming the bare syntax supports variables with the operator

$$\text{var} : \Gamma \ni \alpha \to \text{Term}\ \Gamma\ \alpha,$$

we can lift it to the nice syntax as follows using a suitable downgrade on variables, of type $\Gamma \ni \alpha \to \downarrow\Gamma \ni \downarrow\alpha$.

$$\text{var}^✨ : \Gamma \ni \alpha \to \text{Term}^✨\ \Gamma\ \alpha$$
$$\text{var}^✨\ i := \text{var}\ \downarrow i$$

Now to lift substitution our goal is to define

$$\text{sub}^✨ : \text{Term}^✨\ \Gamma\ \alpha \to \Gamma \dashv[\ \text{Term}^✨\ ]\mapsto \Delta \to \text{Term}^✨\ \Delta\ \alpha.$$

This is almost but not quite the following instance of our already defined bare substitution

$$\text{sub} : \text{Term}\ \downarrow\Gamma\ \downarrow\alpha \to \downarrow\Gamma \dashv[\ \text{Term}\ ]\mapsto \downarrow\Delta \to \text{Term}\ \downarrow\Delta\ \downarrow\alpha.$$

The culprit is the assignment argument. Spelling out the two assignment types completely, we have respectively nice assignments of sort

$$\{\alpha : T^✨\} \to \Gamma \ni \alpha \to \text{Term}\ \downarrow\Delta\ \downarrow\alpha$$

and bare assignments at downgraded contexts of sort

$$\{\alpha : T\} \to \downarrow\Gamma \ni \alpha \to \text{Term}\ \downarrow\Delta\ \alpha.$$

One can already feel that things are going south: to downgrade the former into the latter, we are given a bare type α and a membership proof in a downgraded nice context $i : \downarrow\Gamma \ni \alpha$, and to apply it to the nice assignment we need to *upgrade*

this into a niceness witness $p : ✨\ \alpha$ and a membership proof in the original nice context $\Gamma \ni (\alpha, p)$. This is perfectly doable as indeed the following isomorphism holds

$$\downarrow\Gamma \ni \alpha \approx (p : ✨\ \alpha) \times \Gamma \ni (\alpha, p),$$

but I will stop at this point. It is in some way satisfying, but quite exhausting, to play the upgrade-downgrade yoga on variables which is required to finish the definition of sub✨, and then to prove the substitution monoid laws (which I have for now only alluded to).

The way out of all this administrative type shuffling is to notice that our definition of Ctx✨ T completely misses the point that nice contexts are a subset of contexts. Indeed a more practical definition in this situation would have been as pairs of a context together with a proof that it contains only nice types.

$$\text{Ctx✨}\ T := (\Gamma : \text{Ctx}\ T) \times \text{All}\ ✨\ \Gamma,$$

where the All predicate lifting can be defined as

$$\begin{aligned}&\text{All} : (T \to \text{Prop}) \to (\text{Ctx}\ T \to \text{Prop})\\&\text{All}\ P\ \Gamma := \forall\ \{\alpha\} \to \Gamma \ni \alpha \to P\ \alpha.\end{aligned}$$

This makes downgrading nice contexts easier, but the prime benefit of this change is in the definition variables.

$$\begin{aligned}&\_ \ni✨ \_ : \text{Ctx✨}\ T \to T✨ \to \text{Type}\\&\Gamma \ni✨ \alpha := \Gamma.\text{fst} \ni \alpha.\text{fst}\end{aligned}$$

As variables now disregard niceness, all of the upgrade-downgrade yoga vanishes. In fact lifting the substitution monoid structure to the nice terms is now mostly a matter of η-expanding all the fields, and the hard work is taken care of by unification.

$$\begin{aligned}\text{var✨}\ i &:= \text{var}\ i\\\text{sub✨}\ \{\Gamma\}\ x\ \gamma &:= \text{sub}\ x\ (\lambda\ \{\alpha\}\ i \mapsto \gamma\ \{\alpha, \Gamma.\text{snd}\ i\}\ i)\end{aligned}$$

The conclusion of this small case study is that although our two definitions of nice contexts are *isomorphic*, they are by no means equivalent in term of ease of use. Because I believe it is important to build abstractions that people will actually willingly instantiate in their particular case of interest, it becomes necessary to provide some breathing room in the concrete definition of contexts and crucially in their notion of variables.

3.2.1 Abstract Scopes

So what is a scope, if not a list? For our purpose very little is needed. We will only need to know about an empty scope, a concatenation operation on scopes and a definition for variables. More precisely, given a set of object language types $T : \mathrm{Type}$, a scope structure on a set $S : \mathrm{Type}$ consists of

Notice that we do not ask for a singleton scope $[\_] : T \to S$ which would embed types into scopes. This operation is not part of the core theory, but may be easily added in applications other than OGS for which it is required.

(1) a distinguished *empty scope* $\varnothing : S$,
(2) a binary *concatenation* operation $\_ +\!\!+ \_ : S \to S \to S$,
(3) and a family of *variables* $\_ \ni \_ : S \to T \to \mathrm{Type}$,
(4) such that the empty scope has no variable: $\varnothing \ni t \approx \bot$,
(5) and such that the set of variables of a concatenation is the coproduct of the sets of variables: $(\Gamma +\!\!+ \Delta) \ni t \approx (\Gamma \ni t) + (\Delta \ni t)$.

To formalize the two isomorphisms, we will not take the route of axiomatizing two maps, *forward* and *backward*, which compose to the identity. First remark that by initiality of $\bot$, it suffices to have the forward direction $\varnothing \ni t \to \bot$ to obtain the first isomorphism (4) in full.

For the second isomorphism (5), taking hints both from Homotopy Type Theory [83] and from the *view* methodology [71][12], we will axiomatize only the backward map and ask that its fibers (sets of preimages) are *contractible*, i.e., inhabited by exactly one element. This will make the isomorphism much easier to use, enabling inversions by a simple dependent pattern matching instead of tedious equational rewriting.

[83] The Univalent Foundations Program, *Homotopy Type Theory: Univalent Foundations of Mathematics*, 2013, §4.4, pp. 136–137.
[71] Conor McBride and James McKinna, "The view from the left," 2004, §6, pp. 98–102.
[12] Guillaume Allais, "Builtin Types Viewed as Inductive Families," 2023.

Remark 3.4: Let us quickly see why contractible fibers are of practical interest. The fibers of a function f can be encoded in type theory by the following family.

$$\mathrm{data}\ \mathrm{Fiber}\ \{A\ B\}\ (f : A \to B) : \mathrm{Type}^B \qquad \frac{a : A}{\mathrm{fib}\ a : \mathrm{Fiber}\ f\ (f\ a)}$$

Then, given a function $f : A \to B$ and a proof that its fibers are inhabited

$$\mathrm{inv} : (b : B) \to \mathrm{Fiber}\ f\ b,$$

in any proof with a variable $b : B$ in scope, we can do a dependent elimination on inv b. This will introduce an $a : A$ and magically unify b with $f\ a$, clearing b from the scope. It is for example trivial to obtain an left-inverse to f, given by $\mathrm{get} \circ \mathrm{inv}$ as follows.

```
                                  left-inv (b : B) : f (get (inv b)) = b
get {b} : Fiber f b → A           left-inv := case p b
get (fib a) := a                  [ fib a := refl
```

From an additional proof that every element of the fiber $x : \mathsf{Fiber}\ f\ b$ is equal to $\mathsf{inv}\ b$

```
H : ∀ {b} (x : Fiber f b) → x = inv b,
```

it is again not much work to obtain the right-inverse property, recognizing that by definition $\mathsf{get}\ (\mathsf{fib}\ a) = a$

```
right-inv (a : A) : get (inv (f a)) = a
right-inv a := congr get (H (fib a))
```

As the domain of the backward map of the isomorphism in (5) has as domain a sum type, I will axiomatize it implicitly as the copairing of two simpler maps:

```
∀ {t} → Γ ∋ t → (Γ ∔ Δ) ∋ t
∀ {t} → Δ ∋ t → (Γ ∔ Δ) ∋ t,
```

which are respectively definitionally equal to more concise notations

```
Γ ⊆ (Γ ∔ Δ)
Δ ⊆ (Γ ∔ Δ).
```

The fibers of the copairing of two maps can be more directly characterized by the following data type.

```
data SumView (f : A → C) (g : B → C) : C → Type

      i : A                              j : B
-------------------------   -------------------------
v-left i : SumView f g (f i)   v-right j : SumView f g (g j)
```

We are now ready to give the definition of abstract scope structures.

Definition 3.5 (Abstract Scope Structure):
Given $S, T : \mathsf{Type}$, an *abstract scope structure on* S *over* T is given by the following typeclass, mutually defined with a notation for renamings.

```
class Scope_T S :=
  [ ∅ : S
  ; _ ⧺ _ : S → S → S
  ; _ ∋ _ : S → T → Type
  ; view-emp {t} (i : ∅ ∋ t) : ⊥
  ; r-cat_l {Γ Δ} : Γ ⊆ (Γ ⧺ Δ)
  ; r-cat_r {Γ Δ} : Δ ⊆ (Γ ⧺ Δ)
  ; view-cat {Γ Δ t} (i : (Γ ⧺ Δ) ∋ t) : SumView r-cat_l r-cat_r i
  ; view-cat-eq {Γ Δ t} {i : (Γ ⧺ Δ) ∋ t} v : view-cat i = v
  ]

_ ⊆ _ : S → S → Type
Γ ⊆ Δ := ∀ {t} → Γ ∋ t → Δ ∋ t
```

Note that the mnemonic "cat" in the above stands for con*cat*enation (and not for *cat*egory).

Definition 3.6 (Scope Category):
A scope structure $\mathrm{Scope}_T\ S$ defines a *category of scopes* $\mathcal{C}_S$ whose objects are given by S and whose morphisms are given by renamings $\Gamma \subseteq \Delta$. In other words, $\mathcal{C}_S$ is the *full image* of $\_ \ni \_$.

Remark 3.7: Note that in the definition of abstract scope structures, the set T plays almost no role, being only used to form the family category $T \to \mathrm{Type}$ in the sort of $\ni$. In future work I believe to be particularly fruitful to replace $T \to \mathrm{Type}$ with an arbitrary suitably well-behaved category $\mathcal{A}$, i.e. axiomatizing variables as $\_ \ni : S \to \mathcal{A}$.

In particular $\mathcal{A} := \mathrm{Type}$ provides a more satisfying account of untyped calculi than setting $T := \mathbf{1}$, i.e. $\mathcal{A} := \mathbf{1} \to \mathrm{Type}$ (as is currently required). In general, it would allow much more flexibility in choosing the sort of term families.

Remark 3.8: Our definition of abstract scope structure is quite close in spirit to the the *nameless, painless* (NaPa) abstraction of Pouillard [80]. Their notion is only concerned with untyped scopes and variables, but this is only a superficial difference as their theory could certainly be lifted to indexed settings, or ours lowered as sketched in the previous remark. Apart from this, the actual difference is twofold.

[80] Nicolas Pouillard, "Nameless, painless," 2011.

First, they focus on extending scopes by one variable on the right, whereas we axiomatize arbitrary concatenation. We believe that such single variable extension is an accident of the typical lists and De-Bruijn indices, and that it is more practical to abstract over a more symmetric core operation, namely binary concatenation. This leads to a rather more concise axiomatization of the

laws, and to an easier instantiation of the structure in cases where extending on the right is not the natural primitive operation.

Second, they further *axiomatize* a notion of scope inclusions, whereas we *derived* $\subseteq$ as functions from variables to variables. Again, this leads to their addition of several more laws and coherence conditions. These essentially state that scopes and inclusions form a category and that $\ni$ is functorial. Because we decreed that the category of scopes is given by the full image of $\ni$, every single such law is true *definitionally*.

This axiomatization of scopes is enough to derive the two isomorphisms describing the variables of our scope operations:

$$\varnothing \ni t \approx \bot$$
$$(\Gamma \mathbin{+\!\!+} \Delta) \ni t \approx (\Gamma \ni t) + (\Delta \ni t)$$

In particular, this entails that r-cat_l and r-cat_r are both injective and have disjoint images. In fact, assuming an abstract scope structure $\text{Scope}_S\ T$, the category $\mathcal{C}_S$ is cocartesian, with the initial object $\varnothing$ and the coproduct given by $\mathbin{+\!\!+}$.

Before moving on to the theory of scoped-and-typed families and substitution monoids, let us reap the benefits of this new abstraction and conclude with some instances of abstract scopes.

3.2.2 Instances

Concrete Scopes Lists and De Bruijn indices are the obvious first instance, which we call *concrete scopes*. Concatenation is computed by induction on the second (right-hand) context:

$$\_ \blacktriangleright\!\!\blacktriangleright \_ : \text{Ctx}\ T \to \text{Ctx}\ T \to \text{Ctx}\ T$$
$$\Gamma \blacktriangleright\!\!\blacktriangleright \varepsilon \quad := \Gamma$$
$$\Gamma \blacktriangleright\!\!\blacktriangleright (\Delta \blacktriangleright \alpha) := (\Gamma \blacktriangleright\!\!\blacktriangleright \Delta) \blacktriangleright \alpha$$

I will not provide the full instantiation of the scope structure, suffice it to say that statements about concatenation are proven by induction on the second context argument. Notably, I believe that proving the contractibility property of the fibers of the coproduct injections (view-cat-eq) requires the use of Streicher's axiom K, although I am not entirely sure about this.

Definition 3.9 (Concrete Scopes):
Given $T : \text{Type}$, *concrete scopes* $\text{Ctx}\ T$ have an abstract scope structure with types T given by the following (incomplete) definition.

$$\mathrm{CtxScope}_T := \begin{bmatrix} \varnothing & := \varepsilon \\ \Gamma +\!\!+ \Delta & := \Gamma \blacktriangleright\!\blacktriangleright \Delta \\ \Gamma \ni \alpha & := \Gamma \ni \alpha \\ \dots \end{bmatrix}$$

Pay attention to the difference between $\ni$ and $\exists$! The former denotes the family of concrete De-Bruijn indices, while the latter denotes the abstract variables of a scope structure instance. I apologize for this abuse of notation to the color blind reader. In any case the symbol $\exists$ can without any loss be considered to always denote the abstract notion of variable. In the concrete case it will be definitionally equal to De-Bruijn indices. Further note that this symbol is entirely different to $\in$, which we used to denote predicate membership proofs, i.e., a specialized notation for reverse function application. The latter will be used very rarely from this point on.

Subset Scopes We can now revisit our introductory example motivating the notion of abstract scope structures: *subset scopes*. Given an abstract scope structure $C : \mathrm{Scope}_T\ S$, define the following (strict) predicate lifting.

$$\mathrm{All}_S : (T \to \mathrm{SProp}) \to (S \to \mathrm{SProp})$$
$$\mathrm{All}_S\ P\ \Gamma := \forall\ \{\alpha\} \to \Gamma \ni \alpha \to P\ \alpha$$

We define the subset type of elements of x satisfying $P\ x$, i.e., the *total space* of the predicate P as follows.

$$\textstyle\int \{X : \mathrm{Type}\} : (X \to \mathrm{SProp}) \to \mathrm{Type}$$
$$\textstyle\int P := (x : X) \times P\ x$$

Definition 3.10 (Subset Scopes):
Given an abstract scope instance $\mathrm{Scope}_T\ S$ and a predicate $P : T \to \mathrm{Prop}$, the type $\int(\mathrm{All}_S\ P)$ of *subset scopes* bears an abstract scope structure on types $\int P$, given by the following (incomplete) definition.

```
SubScope : Scope_{∫P} ∫(All_S P)
SubScope :=
  ∅       := [ fst    := ∅
             | snd i := case (view-emp i) [
  Γ ∔ Δ := [ fst := Γ.fst ∔ Δ.fst
             | snd i := case (view-cat i)
               [ v-left i  := Γ.snd i
               | v-right j := Δ.snd i
  Γ ∋ α   := Γ.fst ∋ α.fst
  ...
```

Direct Sum of Scopes Languages exist in various shapes and forms, and sometimes the designers deem it useful to have *two* kinds of variables, stored in *two* different scopes. We can capture this pattern as the direct sum of abstract scope structures.

Definition 3.11 (Direct Sum Scopes):
Given two abstract scope instances $\mathrm{Scope}_{T_1}\ S_1$ and $\mathrm{Scope}_{T_2}\ S_2$, the type $S_1 \times S_2$ of *direct sum scopes* bears an abstract scope structure on types $T_1 + T_2$ given by the following (incomplete) definition.

```
SumScope : Scope_{T₁+T₂} (S₁ × S₂)
SumScope :=
  ∅          := (∅, ∅)
  Γ ∔ Δ      := (Γ.fst ∔ Δ.fst, Γ.snd ∔ Δ.snd)
  Γ ∋ inl α := Γ.fst ∋ α
  Γ ∋ inr α := Γ.snd ∋ α
  ...
```

Untyped Scopes An untyped syntax can always be made to fit into a typed setting by seeing it as uni**t**yped, i.e., where the set of types is given by the singleton $\mathbf{1}$, but it is not *that* simple. Setting $T := \mathbf{1}$ and going on working with e.g., concrete scopes $\mathrm{Ctx}\ \mathbf{1}$ and De-Bruijn indices is slightly unsatisfying. First of all, $\mathrm{Ctx}\ \mathbf{1}$ is isomorphic to the more idiomatic $\mathbb{N}$ and likewise, the corresponding De-Bruijn indices $\_ \ni \star : \mathrm{Type}^{\mathrm{Ctx}\ \mathbf{1}}$ are isomorphic to $\mathrm{Fin} : \mathrm{Type}^{\mathbb{N}}$, the finite sets. Apart from these esthetical considerations, a more worrying technicality arises when your chosen type theory does *not* support the η-rule on $\mathbf{1}$. This law is quite important as it makes all inhabitants of $\mathbf{1}$ definitionally equal, and, more importantly, all function $f : \mathbf{1} \to X$ definitionally constant. In the idealized type theory chosen for this thesis we do assume this η-rule, but our concrete code artifact is stuck with a theory which does not (Rocq!).

Recall the definition of *finite sets* data Fin : $\mathrm{Type}^{\mathbb{N}}$.

$$\frac{}{\mathrm{ze} : \mathrm{Fin}\ (\mathrm{su}\ n)} \qquad \frac{i : \mathrm{Fin}\ n}{\mathrm{su}\ i : \mathrm{Fin}\ (\mathrm{su}\ n)}$$

Further define the following helpers.

$$\begin{aligned}
&\mathrm{fin\text{-}weaken}\ \{m\ n\} : \mathrm{Fin}\ m \to \mathrm{Fin}\ (m + n)\\
&\mathrm{fin\text{-}weaken}\ \mathrm{ze} \quad := \mathrm{ze}\\
&\mathrm{fin\text{-}weaken}\ (\mathrm{su}\ i) := \mathrm{su}\ (\mathrm{fin\text{-}weaken}\ i)
\end{aligned}$$

$$\begin{aligned}
&\mathrm{fin\text{-}shift}\ \{m\ n\} : \mathrm{Fin}\ n \to \mathrm{Fin}\ (m + n)\\
&\mathrm{fin\text{-}shift}\ \{\mathrm{ze}\} \quad i := i\\
&\mathrm{fin\text{-}shift}\ \{\mathrm{su}\ m\}\ i := \mathrm{su}\ (\mathrm{fin\text{-}shift}\ \{m\}\ i)
\end{aligned}$$

Finally define *untyped scopes* as the following instance of scope structure.

$$\begin{aligned}
&\mathrm{UntypedScope} : \mathrm{Scope}_1\ \mathbb{N}\\
&\mathrm{UntypedScope} :=\\
&\left[\begin{aligned}
&\varnothing \quad := \mathrm{ze}\\
&m +\!\!+ n := n + m\\
&n \ni x \quad := \mathrm{Fin}\ n\\
&\text{r-cat}_l\ i \ := \mathrm{fin\text{-}shift}\ i\\
&\text{r-cat}_r\ i \ := \mathrm{fin\text{-}weaken}\ i\\
&\ldots
\end{aligned}\right.
\end{aligned}$$

Note that we swap m and n in the definition of $m +\!\!+ n$. The reason for this is that scopes are traditionally taken to grow towards the right, while unary natural numbers grow towards the left, i.e., addition is defined by recursion on the first argument. This is technically unnecessary but helps avoid unpleasant surprises during index juggling.

We will use concrete scopes in Ch. 4, and subset scopes will make an appearance in Ch. 7 but the other instances are mostly here for illustration.

3.3 Substitution Monoids and Modules

Equipped with this new abstraction for scopes, we are ready to continue the theory of substitution. This will largely follow the now standard approach outlined in §3.1. We will however introduce one novel contribution: substitution modules. Let us start with scoped families and assignments.

Definition 3.12 (Scoped-and-Typed Family):
Given S, T : Type, the set of *scoped-and-typed families* is given by the following sort.

$$\mathrm{SFam}_T\ S := S \to T \to \mathrm{Type}$$

Scoped-and-typed families form a category with arrows $X \twoheadrightarrow Y$ lifted pointwise from Type.

Definition 3.13 (Assignments):
Assuming a scope structure $\mathrm{Scope}_T\ S$, given a scoped-and-typed family $X : \mathrm{Type}^{S,T}$ and $\Gamma, \Delta : S$, the set of *X-assignments from Γ to Δ* is defined as follows.

$$\_ \dashv[X]\mapsto \_ : S \to S \to \mathrm{Type}$$
$$\Gamma \dashv[X]\mapsto \Delta := \forall\ \{\alpha\} \to \Gamma \ni \alpha \to X\ \Delta\ \alpha$$

As seen in § 3.1, because assignments are represented as functions, we will use of extensional equality on assignments at several places. Given $\gamma, \delta : \Gamma \dashv[X]\mapsto \Delta$, it is expressed as follows.

$$\gamma \approx \delta := \forall\ \{\alpha\}\ (i : \Gamma \ni \alpha) \to \gamma\ i \approx \delta\ i$$

Remark 3.14: By definition, renamings $\Gamma \subseteq \Delta$ are exactly given by $\ni$-assignments $\Gamma \dashv[\ni]\mapsto \Delta$.

Definition 3.15 (Copairing):
Given a scope structure $\mathrm{Scope}_T\ S$ and a family $X : \mathrm{Type}^{S,T}$, we extend the initial renaming $\varnothing \subseteq \Gamma$ and the renaming copairing derived from the cocartesian structure of $\mathcal{C}_S$ to arbitrary X-assignments as follows.

$$[]\ \{\Gamma\} : \varnothing \dashv[X]\mapsto \Gamma$$
$$[]\ i := \mathrm{case}\ \text{view-emp}\ i\ []$$

$$[\_,\_]\ \{\Gamma_1\ \Gamma_2\ \Delta\} : (\Gamma_1 \dashv[X]\mapsto \Delta) \to (\Gamma_2 \dashv[X]\mapsto \Delta) \to (\Gamma_1 +\!\!+ \Gamma_2) \dashv[X]\mapsto \Delta$$
$$[f,g]\ i := \mathrm{case}\ \text{view-cat}\ i\ \begin{bmatrix} \text{v-left}\ i & := f\ i \\ \text{v-right}\ j & := g\ i \end{bmatrix}$$

3.3.1 Substitution Monoids

We now define the internal substitution hom and subsequently substitution monoids.

Definition 3.16 (Internal Substitution Hom):
Assuming a scope structure $\mathrm{Scope}_T\ S$, the *internal substitution hom* is defined as follows.

$$[\![\_ , \_]\!] : \mathrm{Type}^{S,T} \to \mathrm{Type}^{S,T} \to \mathrm{Type}^{S,T}$$
$$[\![X,Y]\!]\ \Gamma\ \alpha := \forall\ \{\Delta\} \to \Gamma \dashv[X]\mapsto \Delta \to Y\ \Delta\ \alpha$$

Definition 3.17 (Substitution Monoids):
Assuming a scope structure $\mathrm{Scope}_T\ S$ and a family $X : \mathrm{Type}^{S,T}$, a *substitution monoid* structure on X is given by the following typeclass.

$$\begin{aligned}
&\mathrm{class}\ \mathrm{SubstMonoid}_S\ (X : \mathrm{Type}^{S,T}) :=\\
&\quad \mathrm{var} : _ \ni _ \to X\\
&\quad \mathrm{sub} : X \to [\![X,X]\!]\\
&\quad \mathrm{sub\text{-}ext} : \mathrm{sub}\ \langle\!\langle \forall^r \approx \to^r [\![\approx , \approx]\!]^r \rangle\!\rangle\ \mathrm{sub}\\
&\quad \mathrm{sub\text{-}id}_l\ \{\Gamma\ \alpha\}\ (x : X\ \Gamma\ \alpha) : \mathrm{sub}\ x\ \mathrm{var} \approx x\\
&\quad \mathrm{sub\text{-}id}_r\ \{\Gamma\ \alpha\}\ (i : \Gamma \ni \alpha)\ (\gamma : \Gamma \dashv[X]\mapsto \Delta) : \mathrm{sub}\ (\mathrm{var}\ i)\ \gamma \approx \gamma\ i\\
&\quad \mathrm{sub\text{-}assoc}\ \{\Gamma_1\ \Gamma_2\ \Gamma_3\ \alpha\}\ (x : X\ \Gamma_1\ \alpha)\\
&\qquad (\gamma : \Gamma_1 \dashv[X]\mapsto \Gamma_2)\ (\delta : \Gamma_2 \dashv[X]\mapsto \Gamma_3)\\
&\qquad : \mathrm{sub}\ (\mathrm{sub}\ x\ \gamma)\ \delta \approx \mathrm{sub}\ x\ (\lambda\ i \mapsto \mathrm{sub}\ (\gamma\ i)\ \delta)
\end{aligned}$$

To make substitution a bit less wordy we will use the notation $v[\gamma] := \mathrm{sub}\ v\ \gamma$. Moreover, we extend substitution pointwise to assignments with the same notation, using the context to disambiguate:

$$\gamma[\delta] := \lambda\ i \mapsto \mathrm{sub}\ (\gamma\ i)\ \delta.$$

For example, using these notations the conclusion of the sub-assoc law can be written $x[\gamma][\delta] = x[\gamma[\delta]]$.

Remark 3.18: Note that the type of var, here written $\_ \ni \_ \to X$, is definitionally equal to the *identity assignment* type $\forall\ \{\Gamma\} \to \Gamma \dashv[X]\mapsto \Gamma$. This coincidence stems from the fact that substitution monoid structures are exactly $\ni$-relative monads [14]. From this perspective, one can construct something similar to a Kleisli category for X, the *X-assignment category* $\mathcal{A}_X$ whose objects are contexts in S and morphisms are given by X-assignments. It is then unsurprising that var—the unit of the relative monad X—is the identity morphism of its Kleisli category.

[14] Thorsten Altenkirch, James Chapman, and Tarmo Uustalu, "Monads Need Not Be Endofunctors," 2010.

Remark 3.19: As stated previously, to avoid functional extensionality, we need to know that every function taking assignments as arguments respects their pointwise equality. This is the case for sub, for which sub-ext is the corresponding "congruence" property (sometimes we say "monotonicity"). As in the previous chapter, we hide the rather large type of sub-ext by liberally using a form of relational translation of type theory [19], denoted by the superscript $\_^r$. Explicitly, given $X^1, X^2, Y^1, Y^2 : \mathrm{Type}^{S,T}$ and

[19] Jean-Philippe Bernardy, Patrik Jansson, and Ross Paterson, "Proofs for free - Parametricity for dependent types," 2012.

$$X^r : \forall\ \{\Gamma\ \alpha\} \to X^1\ \Gamma\ \alpha \to X^2\ \Gamma\ \alpha \to \mathrm{Prop}$$
$$Y^r : \forall\ \{\Gamma\ \alpha\} \to Y^1\ \Gamma\ \alpha \to Y^2\ \Gamma\ \alpha \to \mathrm{Prop},$$

$[\![X^r,Y^r]\!]^r$ is defined as follows.

$$[\![X^r,Y^r]\!]^r\ \{\Gamma\ \alpha\} : [\![X^1,Y^1]\!]\ \Gamma\ \alpha \to [\![X^2,Y^2]\!]\ \Gamma\ \alpha \to \mathrm{Prop}$$
$$[\![X^r,Y^r]\!]^r\ f\ g :=$$
$$\forall\ \{\Delta\}\ (\gamma^1 : \Gamma \dashv[X^1]\mapsto \Delta)\ (\gamma^2 : \Gamma \dashv[X^2]\mapsto \Delta)$$
$$\to (\forall\ \{\alpha\}\ (i : \Gamma \ni \alpha) \to X^r\ (\gamma^1\ i)\ (\gamma^2\ j))$$
$$\to Y^r\ (f\ \gamma^1)\ (g\ \gamma^2)$$

Then, the type of sub-ext can be seen to unfolds as follows.

$$\forall\ \{\Gamma\ \alpha\}\ \{x_1\ x_2 : X\ \Gamma\ \alpha\}\ (x^r : x_1 \approx x_2)$$
$$\{\Delta\}\ \{\gamma_1\ \gamma_2 : \Gamma \dashv[X]\mapsto \Delta\}$$
$$\to (\forall\ \{\beta\}\ (i : \Gamma \ni \beta) \to \gamma_1\ i \approx \gamma_2\ i)$$
$$\to x_1[\gamma_1] \approx x_2[\gamma_2]$$

Remark that with our notation for extensional equality of assignments, the above type is the same as the following one.

$$\forall\ \{\Gamma\ \alpha\}\ \{x_1\ x_2 : X\ \Gamma\ \alpha\}\ (x^r : x_1 \approx x_2)$$
$$\{\Delta\}\ \{\gamma_1\ \gamma_2 : \Gamma \dashv[X]\mapsto \Delta\}$$
$$\to \gamma_1 \approx \gamma_2$$
$$\to x_1[\gamma_1] \approx x_2[\gamma_2]$$

As such, an alternative compact way to write once again the exact same thing would have been the following

$$\mathrm{sub}\ \langle\!\langle \forall^r \approx \to^r \forall^r (\approx \to^r \approx) \rangle\!\rangle\ \mathrm{sub}.$$

The extraordinarily scrupulous reader will have noticed that our use of $\forall^r$ is here slightly inconsistent with its definition (right before Lemma 2.61). Because type-and-scoped families have *two* indices (the scope and the type), we should have written the last expression above as well as our actual type for sub-ext with *two* corresponding $\forall^r$ at the head, i.e., respectively

$$\mathrm{sub}\ \langle\!\langle \forall^r \forall^r \approx \to^r \forall^r (\approx \to^r \approx) \rangle\!\rangle\ \mathrm{sub}$$
$$\mathrm{sub}\ \langle\!\langle \forall^r \forall^r \approx \to^r [\![\approx,\approx]\!]^r \rangle\!\rangle\ \mathrm{sub}.$$

This abuse is "easily" made formal by extending the definition of $\forall^r$ to n-ary type families (and their n-ary relation families) as follows.

$$\forall^r\ \{X^1\ X^2 : \mathrm{Type}^{I_1,..,I_n}\}$$
$$\quad : (\forall\ \{i_1\ ..\ i_n\} \to X^1\ i_1\ ..\ i_n \to X^2\ i_1\ ..\ i_n \to \mathrm{Prop})$$
$$\quad \to (\forall\ \{i_1\ ..\ i_n\} \to X^1\ i_1\ ..\ i_n)$$
$$\quad \to (\forall\ \{i_1\ ..\ i_n\} \to X^2\ i_1\ ..\ i_n)$$
$$\quad \to \mathrm{Prop}$$
$$\forall^r\ X^r\ F^1\ F^2 := \forall\ \{i_1\ ..\ i_n\} \to X^r\ \{i_1\}\ ..\ \{i_n\}\ F^1\ F^2$$

I hope that this glimpse into pure bureaucratic madness makes it clearer why we *need* our terse and perhaps slightly magical relational combinators.

3.3.2 Substitution Modules

Substitution monoids have neatly been generalized to abstract scopes, but for the purpose of modeling Ogs, a part of the theory of substitution is still missing. As explained in our introductory primer (§1.3), in Ogs we will typically refer to various different syntactic constructs such as *values*, *evaluation contexts*, *terms* and evaluator *configurations*.

Values (as well as terms) can be readily represented as a scoped-and-typed family

$$\mathrm{Val} : S \to T \to \mathrm{Type}\,.$$

In contrast, evaluation contexts are better represented as a family

$$\mathrm{ECtx} : S \to T \to T \to \mathrm{Type},$$

where $E : \mathrm{ECtx}\ \Gamma\ \alpha\ \beta$ typically denotes an evaluation context in scope Γ, with a *hole* of type α and an *outer type* β. The family of configurations of an abstract machine has yet a different sort as it is only indexed by a scope:

$$\mathrm{Conf} : S \to \mathrm{Type}\,.$$

We already know how to axiomatize substitution for values: their scoped-and-typed family should form a substitution monoid. But for the other two kinds of families, we would like to axiomatize a substitution operation that allows replacing their variables by values. More explicitly, we want the following substitution operations.

$$\mathrm{sub}\ \{\Gamma\ \alpha\ \beta\} : \mathrm{ECtx}\ \Gamma\ \alpha\ \beta \to \Gamma \to [\!|\ \mathrm{Val}\ |\!] \to \Delta \to \mathrm{ECtx}\ \Delta\ \alpha\ \beta$$
$$\mathrm{sub}\ \{\Gamma\} : \mathrm{Conf}\ \Gamma \to \Gamma \to [\!|\ \mathrm{Val}\ |\!] \to \Delta \to \mathrm{Conf}\ \Delta$$

As we will see, these two maps can be accounted for by constructing a *substitution module structure over* Val for both ECtx and Conf.

To capture the substitution of both kinds of variously indexed families, let us extend the internal substitution hom to n-ary families.

Definition 3.20 (Generalized Substitution Hom):
Given an abstract scope structure $\mathrm{Scope}_T\ S$ and a sequence of indexing types $T_1, .., T_n : \mathrm{Type}$, the *generalized substitution hom* is defined as follows.

$$[\![\_ , \_]\!] : \mathrm{Type}^{S,T} \to \mathrm{Type}^{S,T_1,..,T_n} \to \mathrm{Type}^{S,T_1,..,T_n}$$
$$[\![X , Y]\!]\ \Gamma\ \alpha_1\ ..\ \alpha_n := \forall\ \{\Delta\} \to \Gamma\ \neg[X]\!\!\rightarrow\ \Delta \to Y\ \Delta\ \alpha_1\ ..\ \alpha_n$$

Definition 3.21 (Substitution Module):
Given an abstract scope structure $\mathrm{Scope}_T\ S$ and a sequence of indexing types $T_1, .., T_n : \mathrm{Type}$, a substitution monoid $\mathrm{SubstMonoid}_S\ M$ and a family $X : \mathrm{Type}^{S,T_1,..,T_n}$, a *substitution module over M on X* is given by the following typeclass.

$$\mathrm{class\ SubstModule}_S\ \left(X : \mathrm{Type}^{S,T_1,..,T_n}\right) :=$$
$$\left[\begin{array}{l}
\mathrm{act} : X \to [\![M , X]\!] \\
\mathrm{act\text{-}ext} : \mathrm{act}\ \langle\!\langle \approx \to^r [\![\approx , \approx]\!]^r \rangle\!\rangle\ \mathrm{act} \\
\mathrm{act\text{-}id}\ \{\Gamma\ \alpha_1\ ..\ \alpha_n\}\ (x : X\ \Gamma\ \alpha_1\ ..\ \alpha_n) : \mathrm{act}\ x\ \mathrm{var} \approx x \\
\mathrm{act\text{-}comp}\ \{\Gamma_1\ \Gamma_2\ \Gamma_3\ \alpha_1\ ..\ \alpha_n\}\ (x : X\ \Gamma_1\ \alpha_1\ ..\ \alpha_n) \\
\quad (\gamma : \Gamma_1\ \neg[M]\!\!\rightarrow\ \Gamma_2)\ (\delta : \Gamma_2\ \neg[M]\!\!\rightarrow\ \Gamma_3) \\
\quad : \mathrm{act}\ (\mathrm{act}\ x\ \gamma)\ \delta \approx \mathrm{act}\ x\ \gamma[\delta]
\end{array}\right.$$

I am slightly sloppy around the n-ary binders denoted by "..". In the current Rocq code, I have rather unsatisfyingly special-cased this definition for scoped families indexed by 0, 1 or 2 types, which are sufficient for our purpose. In further work this precise definition could be captured by building upon [11] Guillaume Allais, "Generic level polymorphic n-ary functions," 2019.

Overloading the notation for the ordinary substitution sub, we will use $x[\gamma]$ as shorthand for $\mathrm{act}\ x\ \gamma$.

3.3.3 Renaming Structures

Substitution modules shed a new light on the renaming operation. Indeed, as seen in §3.1 the state of the art is to mechanize a family with renamings as a coalgebra for the $[\![\ni , \_]\!]$ comonad [37][13]. However, a family with renamings can also be characterized as a substitution module over $\ni$ (as $\ni$ trivially forms a substitution monoid).

[37] Marcelo Fiore and Dmitrij Szamozvancev, "Formal metatheory of second-order abstract syntax," 2022.
[13] Guillaume Allais, Robert Atkey, James Chapman, Conor McBride, and James McKinna, "A type- and scope-safe universe of syntaxes with binding: their semantics and proofs," 2021.

Pulling the other way on these two point of views, substitution modules over M could be reframed as coalgebras for the comonad $\square_M X := [\![M , X]\!]$, exhibiting the reindexing functor $(\mathcal{A}_M \to \mathcal{C}) \to (S \to \mathcal{C})$ as comonadic.

Let's give a bit more details. First, the monoid structure on $\ni$.

Lemma 3.22 (Monoid Structure on $\ni$):
Assuming a scope structure $\mathrm{Scope}_T\ S$, the scoped-and-typed family

$\_ \ni \_ : \mathrm{Type}^{S,T}$ can be equipped with a substitution monoid structure as follows.

```
∋-sub-monoid : SubstMonoid (_ ∋ _)
∋-sub-monoid :=
  [ var i := i
  | sub i γ := γ i
  | ...
```

Next, we can define a shorthand for renaming structures.

Definition 3.23 (Renaming Structure):
Assuming a scope structure $\mathrm{Scope}_T\ S$, given a family $X : \mathrm{Type}^{S,T_1,..,T_n}$, a *renaming structure* on X is given by the following typeclass.

```
class RenModule X := SubstModule_∋ X
```

Finally, we define the $\square_M$ comonad and link it with substitution modules.

Definition 3.24 (Internal Substitution Hom Comonad):
Assuming a scope structure $\mathrm{Scope}_T\ S$, given a family $M : \mathrm{Type}^{S,T}$ equipped with a substitution monoid structure SubstMonoid M, define the following functor.

```
□_M : Type^{S,T_1,..,T_n} → Type^{S,T_1,..,T_n}
□_M X := ⟦M,X⟧
```

$\square_M$ has a comonad structure, with counit ε and comultiplication δ given as follows.

```
ε : □_M X → X          δ : □_M X → □_M (□_M X)
ε f := f var           δ f γ1 γ2 := f γ1[γ2]
```

Lemma 3.25 (Substitution Module is Coalgebra):
Assuming a scope structure $\mathrm{Scope}_T\ S$, given a family $M : \mathrm{Type}^{S,T}$ equipped with a substitution monoid structure SubstMonoid M, for any $X : \mathrm{Type}^{S,T_1,..,T_n}$, substitution module structures over M on X coincide with $\square_M$ comonad coalgebra structures on X.

For any X, we directly deduce that $\square_M\ X$ enjoys a substitution module structure over M: the free coalgebra structure on X.

Proof: This lemma is more or less trivial, since our definition of substitution module can be directly read as the definition of $\square_M$ comonad coalgebras. Indeed, act coincides with the coalgebra structure map while act-id and act-comp coincide with the two comonad coalgebra laws. ■

The above lemma exhibits the link between our substitution modules and $\square_M$ coalgebras, extending the previous result on renaming structures [13][37].

Finally, we conclude this chapter by defining one last structure, for families that have both renamings and variables, described by Fiore and Szamozvancev as *pointed coalgebras*.

Definition 3.26 (Pointed Renaming Structure):
Assuming a scope structure $\mathrm{Scope}_T\ S$, given a family $X : \mathrm{Type}^{S,T}$, a *pointed renaming structure* on X is given by the following typeclass.

```
class PointedRenModule X :=
  [ extends RenModule X
  | var : ␣ ∋ ␣ → X
  | act-var {Γ Δ α} i (ρ : Γ ⊆ Δ) : (var i)[ρ] ≈ var (ρ i)
```

With substitution monoids, substitution modules and renaming structures defined, we now have the flexible tools we need in the next chapter to axiomatize the object language of our generic Ogs construction. Although we have only seen a glimpse of what can be done using the intrinsically typed and scoped approach for modeling binders, I hope to have demonstrated the ease with which it can be adapted to specific situations like different indexing (with substitution modules) or new scope representations (with abstract scope structures).

[13] Guillaume Allais, Robert Atkey, James Chapman, Conor McBride, and James McKinna, "A type- and scope-safe universe of syntaxes with binding: their semantics and proofs," 2021.
[37] Marcelo Fiore and Dmitrij Szamozvancev, "Formal metatheory of second-order abstract syntax," 2022.

4 Generic Operational Game Semantics

In Ch. 2 we have seen a general definition of games and the structure of their strategies and in Ch. 3 we have seen an axiomatic framework for intrinsically typed and scoped substitution. Everything is now in place to define the generic operational game semantics construction. Our goal for this chapter in threefold. We need to define the Ogs game, then axiomatize an abstract notion of language with evaluator, and finally construct a model of this language inside Ogs game strategies.

While a naive definition of the Ogs game is not too hard to develop, as we will see this naive construction hits several related roadblocks quite a bit later, when trying to prove the model correct w.r.t. observational equivalence. These roadblocks can be overcome, but at the cost of several small tweaks, distracting us from the core of the construction.

As such, we start in § 4.1 by pushing as far as we can the naive construction, to get a good understanding of the important parts of the model, and to grasp what these roadblocks precisely are. In §4.1.1 we introduce an abstract notion of *observation* and define the naive Ogs game, parametrized by such observations. We then provide in §4.1.2 an informal and progressive construction of the model. In §4.1.3 we introduce our novel axiomatization of languages, and properly define their interpretation into naive Ogs strategies. Finally in § 4.1.4 we discuss the correctness proof and the problematic points of the naive model.

We present the refined model in § 4.2. We start by refining the game (§ 4.2.1), then the strategies and their composition (§ 4.2.2), and finally the Ogs model (§4.2.3). We conclude in §4.2.4 by stating the correctness theorem, making all of its hypotheses explicit.

4.1 A Simple Ogs Model

4.1.1 The Ogs Game

Recall from the informal introductory description (§1.3) that the Ogs model proceeds by computing the normal form of a given configuration (or term) and then splitting it into a *head variable*, an *observation* on it, and a *filling assignment*, associating a value to each *hole* or *argument* of the observation. An Ogs *move* is then obtained by combining the head variable and the observation, while the assignment is kept local to the Ogs strategy and hidden from the opponent.

Symmetrically, the opponent can then *query* these hidden arguments by itself playing a move of the same shape, a variable and an observation, resuming execution on player's side. To make these ideas formal, we first need to properly define what these observations look like.

Intuitively, the set of observations should be indexed by an object type: the type of values they are meant to be observing. Moreover, each observation has a given arity, a list of arguments or holes, i.e., a scope. At first sight it might seem natural to describe observations simply as a scoped-and-typed family with scopes S and types T given by $\mathrm{Obs} : \mathrm{Type}^{S,T}$, where $o : \mathrm{Obs}\ \Gamma\ \alpha$ denotes an observation o on some value of type α with arity Γ. While this representation could work out, it would be quite unnatural to manipulate. To explain this, let us take a step back.

For other scoped syntactic categories like values or terms, the indexing scope constrains what variables the term can *mention*. Because we want to think of variables as pointers into a scope, in terms of conceptual dependency, the scope is preexisting and the term over it comes afterwards. It is then sensible to reflect this dependency formally and use scoped-and-typed families, making the sort of terms *depend* on a scope (and also on an object type). On the other hand, the scope of an observation is not there to constrain anything but to make known the arity of that observation. Here, the more natural information flow is to have the scope come *after* the observation, which would thus only depend on the type. In the language of bidirectional typing [77][35], this amounts to saying that for patterns, the scope is being *inferred* while the type is being *checked*.

[77] Benjamin C. Pierce and David N. Turner, "Local Type Inference," 1998.
[35] Jana Dunfield and Neel Krishnaswami, "Bidirectional Typing," 2022.

These fine encoding considerations might be dismissed as philosophical or even aesthetic. But they do have pragmatic consequences, typically on universe levels and sizes. As a perhaps more tangible argument, there are in typical calculi only a finite number of observations at any given type while the set of scopes is infinite. As such, only a fraction of scopes can be the arity of some observation, wasting the expressivity of scoped-and-typed families since for most Γ, $\mathrm{Obs}\ \Gamma\ \alpha$ would be empty.

This leads us to axiomatize observations as *binding families*, which we now define.

Definition 4.1 (Binding Family):
Given $S, T : \mathrm{Type}$, a *binding family* is given by records of the following type.

$$\begin{aligned}&\mathrm{record}\ \mathrm{Bind}\ S\ T :=\\ &\quad \left[\begin{array}{l}\mathrm{Op} : T \to \mathrm{Type}\\ \mathrm{holes}\ \{\alpha\} : \mathrm{Op}\ \alpha \to S\end{array}\right.\end{aligned}$$

Note that this definition is isomorphic to $T \to \mathrm{Fam}\ S$, the record presentation is simply for clarity. In fact, up-to the name of the projections, it is exactly the same definition as the half-games from Ch. 2, but it is used for a different purpose (or maybe not).

Let us now define Ogs moves, which are made of pairs of a variable and an observation (or in fact *triplets* accounting for the type which is also implicitly there).

Definition 4.2 (Named Observation):
Assuming a scope structure $\mathrm{Scope}_T\ S$ and a binding family $O : \mathrm{Bind}\ S\ T$, the scoped family of *named observations* $O^{\aleph} : \mathrm{Type}^S$ is defined as follows.

$$O^{\aleph}\ \Gamma := (\alpha : T) \times (\Gamma \ni \alpha) \times O.\mathrm{Op}\ \alpha$$

We will write $i \blacksquare o$ as a shorthand for (α, i, o), leaving α implicit. Moreover, we lift holes to named observations with the following definition.

$$\mathrm{holes}^{\aleph}\ \{\Gamma\} : O^{\aleph}\ \Gamma \to S$$
$$\mathrm{holes}^{\aleph}\ (i \blacksquare o) := O.\mathrm{holes}\ o$$

In the Ogs game, both players play named observations and in doing so, they introduce fresh variables corresponding to their holes. These fresh variables can then be further observed by the other player, but not by themselves! The game positions can thus be described as pairs of scopes (Γ, Δ), each tracking the variables introduced by a given player, and thus allowed to be chosen and observed by the other player. We obtain the following game definition.

Definition 4.3 (Ogs Game):
Assuming a scope structure $\mathrm{Scope}_T\ S$, given a binding family $O : \mathrm{Bind}_T\ S$, the *Ogs game on observations* O is defined as follows.

$$\mathrm{Ogs}_O^{\mathrm{HG}} : \mathrm{HGame}\ (S \times S)\ (S \times S) :=$$
$$\begin{bmatrix} \mathrm{Move}\ (\Gamma, \Delta) := O^{\aleph}\ \Gamma \\ \mathrm{next}\ \{\Gamma, \Delta\}\ o := (\Delta +\!\!+ \mathrm{holes}^{\aleph}\ o, \Gamma) \end{bmatrix}$$

$$\mathrm{Ogs}_O^{\mathrm{G}} : \mathrm{Game}\ (S \times S)\ (S \times S) :=$$
$$\begin{bmatrix} \mathrm{client} := \mathrm{Ogs}_O^{\mathrm{HG}} \\ \mathrm{server} := \mathrm{Ogs}_O^{\mathrm{HG}} \end{bmatrix}$$

Remark 4.4: To avoid needlessly duplicating definitions we have preferred a symmetric game, where the client and server half-games are equal. To achieve this, we do not use an *absolute* point of view on the scopes, where some player would always append to the first component Γ while the other player would append to Δ. Instead, we adopt a *relative* point of view, where the first component always tracks the variables introduced by the *currently passive* player, in other words, the one who played last. As such after each move the two components are swapped.

While this trick has bought us some simplicity by obtaining a *strictly* symmetric game, it should be re-evaluated in future work. Indeed, I suspect that it murkens the categorical structure of the Ogs game in contrast to the absolute presentation. Note that the absolute presentation is still symmetric but in a

weaker sense, only up to a function adapting the position (namely swapping the two scopes).

With the Ogs game defined, using the generic description of game strategies as interaction trees developed in Ch. 2, we obtain Ogs strategies easily.

Definition 4.5 (Ogs Strategies):
Assuming an abstract scope structure $\mathrm{Scope}_T\ S$, given a binding family $O : \mathrm{Bind}_T\ S$ *active and passive Ogs strategy over* O are given as follows.

$$\mathrm{Ogs}^+_O := \mathrm{Strat}^+{}_{\mathrm{Ogs}^{\mathrm{c}}_O}\ (\lambda\ (\Gamma, \Delta) \mapsto \mathbf{0})$$
$$\mathrm{Ogs}^-_O := \mathrm{Strat}^-{}_{\mathrm{Ogs}^{\mathrm{c}}_O}\ (\lambda\ (\Gamma, \Delta) \mapsto \mathbf{0})$$

4.1.2 Diving Into the Machine Strategy

For now we know relatively little about the abstract language for which we are constructing an Ogs model: we know about its scope structure, its set of types and a binding family describing its observation. To complete the construction of the Ogs model, that is, to implement a strategy for the Ogs game, we will need to know quite a bit more. Concretely, our goal is to axiomatize something which we call a *language machine*, and to deduce from it the *machine strategy*, implementing the Ogs game. More precisely, we will implement the machine strategy as a big-step system (Definition 2.29) over the Ogs game. As our axiomatization of the language machine is largely guided by what we need to implement the machine strategy, we informally introduce both of them in lock-step, walking gradually through all of their components.

States The starting point to implement a strategy is to choose two families for the active and passive states. Recall that intuitively, during the Ogs game, each player introduces fresh variables that their opponent can subsequently query. As such, the states of the strategy ought to remember what *value* was associated to each introduced variable. Because of our tricky *relative* point of view we have to take some care with the scopes. Recall that in a position (Γ, Δ), if it is *our turn* to play, then Γ lists the opponent-introduced variables, while Δ lists the ones we introduced. As such, the *active* states of the machine strategy should contain an assignment

$$\sigma : \Delta \multimap[\ \mathrm{Val}\ ]\mapsto \Gamma.$$

In contrast, *passive* strategy states should contain an assignment

$$\sigma : \Gamma \multimap[\ \mathrm{Val}\ ]\mapsto \Delta.$$

For this to make sense, the language machine must have a scoped-and-typed family $\mathrm{Val} : \mathrm{Type}^{S,T}$ which we call *values*.

In an active position, we need to decide which move to play, so a bit more data is required besides the assignment σ. Recall that intuitively the Ogs model computes the next move by reducing a program to a normal form. As such active states need to store such a program. As motivated in the introduction, we will entirely forget about terms and instead only work with *configurations*, intuitively the package of a term with its continuation. We thus postulate a scoped family $\mathrm{Conf} : \mathrm{Type}^{S}$ and define the active and passive states of the machine strategy as follows.

$$\mathrm{ogs}^{+}\ (\Gamma, \Delta) \coloneqq \mathrm{Conf}\ \Gamma \times (\Delta \multimap[\ \mathrm{Val}\ ]\mapsto \Gamma)$$
$$\mathrm{ogs}^{-}\ (\Gamma, \Delta) \coloneqq \Gamma \multimap[\ \mathrm{Val}\ ]\mapsto \Delta$$

Next, to implement the machine strategy transitions we need to know two things: how to compute our next move and how to resume when we receive an opponent move. Accordingly this will be reflected in the language machine axiomatization with two maps, evaluation and application. Let us start with the first one.

Play In the Ogs construction the next move is computed by evaluating the term at hand, hence we need to axiomatize an evaluator. Given some family of normal form configurations $\mathrm{Nf} : \mathrm{Type}^{S}$, a possibly non-terminating evaluator for open configurations can be represented as follows.

$$\mathrm{eval}\ \{\Gamma\} : \mathrm{Conf}\ \Gamma \to \mathrm{Delay}\ (\mathrm{Nf}\ \Gamma)$$

Then, to actually compute the next move, the Ogs construction mandates that these normal forms be split into three components: the head variable on which it is stuck, an observation on that variable, and an assignment filling the arguments of the observation (in other words, a named observation and a filling assignment). Instead of axiomatizing a family of normal forms and a splitting map, exploding each normal form into our three components, we will simply *define* normal forms as such triples. Although this might seem overly restrictive, it makes no real difference on the implementation side. These *split normal forms* can be defined as follows.

Definition 4.6 (Split Normal Forms):
Assuming a scope structure $\mathrm{Scope}_T\ S$, given a binding family $O : \mathrm{Bind}\ S\ T$ and a scoped-and-typed family $V : \mathrm{Type}^{S,T}$, *split normal forms* are the scoped family $\mathrm{Nf}_V^O : \mathrm{Type}^{S}$ defined as follows.

$$\mathrm{Nf}_V^O\ \Gamma \coloneqq (o : O^{\mathrm{ix}}\ \Gamma) \times (\mathrm{holes}^{\mathrm{ix}}\ o \multimap[V]\mapsto \Gamma)$$

Extensional equality of normal is given by the data type

```
data _≈_ : IRel Nf^O_V Nf^O_V
```

overloading as usual the symbol $\approx$. It has the following only constructor.

$$\frac{H : \gamma_1 \approx \gamma_2}{\mathrm{refl}_{\mathrm{nf}}\ H : (o \blacksquare \gamma_1) \approx (o \blacksquare \gamma_2)}$$

Using the newly defined split normal forms, the evaluation map of a machine language can now be given its final type.

```
eval {Γ} : Conf Γ → Delay (Nf^O_Val Γ)
```

This evaluator is sufficient to implement the active transition function of the machine strategy as follows.

Here, $\mathbf{0}$ stands for the output family of the machine strategy, which is empty in concordance with Definition 4.5.

```
mstrat-play {Ψ} : ogs⁺ Ψ → Delay (𝟎 + (Ogs^HG_O ⊛ ogs⁻) Ψ)
mstrat-play (c, σ) := do
  (o, γ) ← eval c ;
  ret (inr (o, [σ, γ]))
```

The function starts by computing the normal form (o, γ), where o is a named observation and γ the filling assignment. Then, it plays the move o and stores the new filling γ besides the currently stored assignment σ by copairing of assignments. To detail the typing, assuming $\Psi = (\Gamma, \Delta)$, we have

$$\sigma : \Delta \qquad \dashv[\ \mathrm{Val}\ ]\mapsto \Gamma$$
$$\gamma : \mathrm{holes}^{\aleph}\ o \dashv[\ \mathrm{Val}\ ]\mapsto \Gamma$$

By definition of the Ogs game, after playing o, the next position is given by $(\Delta +\!\!+ \mathrm{holes}^{\aleph}\ o, \Gamma)$, meaning that we must provide a passive state of type

$$(\Delta +\!\!+ \mathrm{holes}^{\aleph} o) \dashv[\ \mathrm{Val}\ ]\mapsto \Gamma$$

which indeed matches the type of the copairing $[\sigma, \gamma]$.

Coplay We now need to define the coplay function, which takes a passive machine strategy state, a move and returns the next active machine strategy state. Active states contain a configuration, but also an assignment quite similar to the passive configuration. This assignment will simply be weakened and passed along: when the opponent plays a move, the player does not share anything new. Hence, the list of values we need to remember does not change. We can already provide a partial definition.

```
mstrat-coplay {Ψ} : ogs⁻ Ψ → (Ogs^HG_O ⇛ ogs⁺) Ψ
mstrat-coplay σ o :=
  [ fst := ...
  , snd := σ[r-cat_l] ]
```

What is left is to compute the configuration part of the next active state. Recall that intuitively, upon receiving a move $(i \blacksquare o)$, the Ogs construction obtains the next configuration by looking up the value v associated to i and *applying* the observation o to v. As such, we need to axiomatize this application operator in the language machine. It could be modeled by a map of the following type.

```
apply {Γ α} : Val Γ α → (o : O.Op α) → Conf (Γ ∔ O.holes o)
```

Note that the scope of the returned configuration is extended by the observation's holes, since the filling was not given. While this is precisely the operator needed for the Ogs construction, we chose to generalize it slightly by adding the filling assignment (i.e., the observation's arguments) as arguments instead of extending the returned configuration's scope. We thus obtain the following type.

```
apply {Γ α} : Val Γ α → (o : O.Op α) → (O.holes o ⊣[ Val ]⊢ Γ) → Conf Γ
```

Remark 4.7: While slightly superfluous for the Ogs construction, in presence of variables and substitution, both variants are mutually expressible. My feeling is that this second variant is more natural to implement in instances as it is the natural encoding of a generalized eliminator. This design choice should probably be revisited in future work, if only to motivate it better.

Using the apply operator we now finish the definition of the coplay transition function.

```
mstrat-coplay {Ψ} : ogs⁻ Ψ → (Ogs^HG_O ⇛ ogs⁺) Ψ
mstrat-coplay σ (i ■ o) :=
  [ fst := apply (σ i)[r-cat_l] o (r-cat_r ∘ var)
  , snd := σ[r-cat_l] ]
```

Note that this definition crucially requires that the syntax of values has a pointed renaming structure. Indeed, both variables and renamings are used to weaken the assignment part of the state and to "synthesize" the filling assignment passed to apply. As noted above, the second apply operator is indeed superfluous. If we had used the first one, the first projection of the strategy state would have simply read

```
apply (σ i) o.
```

4.1.3 Language Machines and Ogs Interpretation

Let us sum up the previous section and properly define the language machines and then the machine strategy. We have seen that language machines consist of values, configurations, an evaluation map and an observation application map.

Definition 4.8 (Language Machine):
Given a scope structure $\mathrm{Scope}_T\ S$, a binding family $O : \mathrm{Bind}_T\ S$ and families $V : \mathrm{Type}^{S,T}$ and $C : \mathrm{Type}^{S}$, a *language machine over O with values V and configurations C* is given by records of the following type.

$$
\begin{aligned}
&\textbf{record}\ \mathrm{LangMachine}\ O\ V\ C :=\\
&\quad \mathrm{eval}\ \{\Gamma\} : C\ \Gamma \to \mathrm{Delay}\ (\mathrm{Nf}_V^O\ \Gamma)\\
&\quad \mathrm{apply}\ \{\Gamma\ \alpha\}\ (v : V\ \Gamma\ \alpha)\ (o : O.\mathrm{Op}\ \alpha)\\
&\qquad : (O.\mathrm{holes}\ o \multimap [V] \to \Gamma) \to C\ \Gamma\\
&\quad \text{eval-ext} : \mathrm{eval}\ \langle\!\langle \forall^r \approx \to^r \approx \rangle\!\rangle\ \mathrm{eval}\\
&\quad \text{apply-ext} : \mathrm{apply}\ \langle\!\langle \forall^r \approx \to^r\ \forall^r \approx \to^r \approx \rangle\!\rangle\ \mathrm{apply}
\end{aligned}
$$

Remark 4.9: Note that we added two congruence properties to the language machine. The first states that eval sends extensionally equal configurations to strongly bisimilar computations of extensionally equal normal forms. Similarly, we require that apply sends extensionally equal values applied to *the same* observation and two extensionally equal assignments to extensionally equal configurations. There is some slight notational abuse in the above type of this last statement so we give it here in full.

$$
\begin{aligned}
&\forall\ \{\Gamma\ \alpha\}\ (v^1\ v^2 : V\ \Gamma\ \alpha)\ (v^r : v^1 \approx v^2)\\
&\quad (o : O.\mathrm{Op}\ \alpha)\ (\gamma^1\ \gamma^2 : O.\mathrm{holes}\ o \multimap [V] \to \Gamma)\ (\gamma^r : \gamma^1 \approx \gamma^2)\\
&\quad \to \mathrm{apply}\ v^1\ o\ \gamma^1 \approx \mathrm{apply}\ v^2\ o\ \gamma^2
\end{aligned}
$$

Remark 4.10: Note that the above definition only gives the computational structure of a language machine, only requiring very lightweight laws concerned with extensional equality. This will of course not be enough to prove the Ogs model correct w.r.t. observational equivalence of configurations, so that we will gradually define more properties on language machines.

Next, assuming a language machine where values and configurations have renamings, we can give the full definition of the machine strategy.

Definition 4.11 (Machine Strategy):
Given a language machine $M : \mathrm{LangMachine}\ O\ V\ C$ such that V and C have renamings, i.e., such that $\mathrm{PointedRenModule}\ V$ and $\mathrm{RenModule}\ C$ hold, then the *machine strategy* is the big-step system $\mathrm{mstrat}\ M : \text{Big-Step-System}_{\mathrm{Ogs}_O^{\mathcal{C}}}\ \mathbf{0}$ defined as follows.

```
mstrat M :=
  state⁺ (Γ, Δ) := C Γ × (Δ ⊣[V]↦ Γ)
  state⁻ (Γ, Δ) := Γ ⊣[V]↦ Δ
  play (c, σ) := do
    (o, γ) ← M.eval c ;
    ret (inr (o, [σ, γ]))
  coplay σ (i ▪ o) :=
    fst  := M.apply (σ i)[r-cat_l] o (r-cat_r · var)
    snd := σ[r-cat_l]
```

We finish up the model construction by embedding the language configurations into active Ogs strategies.

Definition 4.12 (Ogs Interpretation):
Given a language machine M : LangMachine O V C such that V and C have renamings, i.e., such that PointedRenModule V and RenModule C hold, then the *active and passive Ogs interpretations* are defined as follows.

$$[\![\_]\!]^{+}_{M}\ \{\Gamma\} : C\ \Gamma \to \text{Ogs}^{+}_{O}\ (\Gamma, \varnothing)$$
$$[\![c]\!]^{+}_{M} := \text{unroll}^{+}{}_{\text{mstrat}\ M}\ (c, [])$$

$$[\![\_]\!]^{-}_{M}\ \{\Gamma\ \Delta\} : \Gamma \dashv[V]\mapsto \Delta \to \text{Ogs}^{-}_{O}\ (\Gamma, \Delta)$$
$$[\![\gamma]\!]^{-}_{M} := \text{unroll}^{-}{}_{\text{mstrat}\ M}\ \gamma$$

4.1.4 Correctness?

Now that the Ogs interpretation is defined, we can at last state the correctness property. Intuitively, the goal is to say that if two programs have bisimilar Ogs interpretations, then they are observationally equivalent. Traditionally, two programs are deemed observationally equivalent, or more technically *contextually equivalent*, if for any closed context of some given *ground* type, when placed in that context either both diverge, or both reduce to the same value. In our slightly unusual setting which forgoes any notion of context and instead places configurations at the forefront, the natural notion of observational equivalence is not *contextual equivalence* but instead something we call *substitution equivalence*.

Intuitively, substitution equivalence relates two machine configurations c_1 and c_2 whenever for any substitution γ, $M.\text{eval}\ c_1[\gamma] \approx M.\text{eval}\ c_2[\gamma]$. There are however some subtleties which we will discuss after the actual definition.

Definition 4.13 (Substitution Equivalence):
Assume a language machine M : LangMachine O V C such that V forms a substitution monoid SubstMonoid V and that C forms a substitution

module over it SubstModule$_V$ C. Define *evaluation to (named) observation* as follows.

```
eval-to-obs {Γ} : C Γ → Delay (O^ϰ Γ)
eval-to-obs c := fst ⟨$⟩ M.eval c
```

Then, given a scope $\Omega : S$, *substitution equivalence at final scope* Ω is the indexed relation on C defined as follows.

```
_ ≈^Ω_SUB _ {Γ} : C Γ → C Γ → Prop
c₁ ≈^Ω_SUB c₂ := (γ : Γ –[V]→ Ω) → eval-to-obs c₁[γ] ≈ eval-to-obs c₂[γ]
```

The first subtlety is that in the above definition the final *scope* Ω plays the same role as the ground *type* of contextual equivalence. The generalization from a single type to an entire scope is required simply because in the abstract scope structure axiomatization we did not introduce any means to talk about singleton scopes. However, as this scope is freely chosen in the instances, it may very well be instantiated by a singleton scope, which usually exists in concrete cases.

Second, instead of directly comparing two normal forms obtained by evaluation, substitution equivalence first projects them onto their named observation part, disregarding the filling assignments. The reason for this stems from what makes a "good" ground type. For standard calculi, the ground type of contextual equivalence is invariably taken to be a very simple type such as booleans or the unit type. What is important, is that values of this type can be meaningfully compared syntactically, as this is what contextual equivalence does.

To see how things could turn bad, let us look at a pathological example that does not follow this rule. Assume our calculus has a weak reduction that does not reduce function bodies and now set the ground type of contextual equivalence to some function type $A \to B$. Then, given two lambda abstractions $u := \lambda x.\, U$ and $v := \lambda x.\, V$ of type $A \to B$, contextual equivalence of u and v implies that both are syntactically equal. Indeed, under the trivial context both evaluate to themselves. Hence, two merely pointwise equal function may be distinguished and this completely breaks important properties of contextual equivalence, such as characterizing the greatest adequate congruence relation on terms.

While it might not be very easy to give a clear criterion on what makes a good ground type in general, our setting makes it is relatively easy. The part of a normal form which can meaningfully be syntactically compared is exactly its named observation part (obtained by first projection). One approach would be to restrict the types of Ω to be such that no observation has any hole, i.e., such that for any $o : O.\mathrm{Op}\ \alpha$, $O.\mathrm{holes}\ o \approx \varnothing$. This would entail that the projection $\mathrm{fst} : \mathrm{Nf}_V^O\ \Omega \to O^\varkappa\ \Omega$ is in fact an isomorphism. However, we opt

for the arguably simpler approach of leaving Ω unconstrained but dropping the problematic part of the normal forms.

Finally, we can state the much awaited correctness property of the Ogs model.

Definition 4.14 (Ogs Correctness):
Assume a language machine

$$M : \text{LangMachine } O\ V\ C$$

such that V forms a substitution monoid $\text{SubstMonoid } V$ and that C forms a substitution module over it $\text{SubstModule}_V\ C$. Given a scope $\Omega : S$, *Ogs is correct with respect to substitution equivalence at* Ω if weak bisimilarity of Ogs strategy interpretations entails substitution equivalence.

$$\forall\ \{\Gamma\}\ (c_1, c_2 : C\ \Gamma) \to [\![c_1]\!]^+_M \approx [\![c_2]\!]^+_M \to c_1 \approx^{\Omega}_{\text{SUB}} c_2$$

Alas, from now on, things start to break apart. As explained in the introduction of this chapter, we will not be able to directly prove correctness with this simple version of the Ogs model. Without going into too many details, let us see why.

The main tool for proving correctness of Ogs, and in fact arguably the prime reason for introducing game or interactive models in the first hand, is the definition of a *composition* operation, taking a player strategy and an opponent strategy and pitting them to "play" against each other. Indeed, if we manage to define an operator $\_ \parallel \_$ such that the following two properties hold, then we can easily conclude correctness of the Ogs model.

$$\text{eval-to-obs}\ c[\gamma] \approx [\![c]\!]^+_M \parallel [\![\gamma]\!]^-_M \qquad \text{(adequacy)}$$
$$s_1 \approx s_2 \to (s_1 \parallel t) \approx (s_2 \parallel t) \qquad \text{(congruence)}$$

Indeed, given $c_1, c_2 : C\ \Gamma$ and assuming $[\![c_1]\!]^+_M \approx [\![c_2]\!]^+_M$, we need to prove that for any $\gamma : \Gamma \dashv[V]\mapsto \Omega$, $\text{eval-to-obs}\ c_1[\gamma] \approx \text{eval-to-obs}\ c_2[\gamma]$. The proof goes like this.

$$\begin{aligned}
\text{eval-to-obs}\ c_1[\gamma] &\approx [\![c_1]\!]^+_M \parallel [\![\gamma]\!]^-_M && \text{by adequacy}\\
&\approx [\![c_2]\!]^+_M \parallel [\![\gamma]\!]^-_M && \text{by congruence on hyp.}\\
&\approx \text{eval-to-obs}\ c_2[\gamma] && \text{by adequacy}
\end{aligned}$$

Note that for all of this to typecheck, the composition needs to have the following type.

$$\_ \parallel \_ : \text{Ogs}^+_O\ (\Gamma, \varnothing) \to \text{Ogs}^-_O\ (\Gamma, \Omega) \to \text{Delay}\ (O^{\aleph}\ \Omega)$$

So how would we go about to define composition? Although we have not yet talked about it, composition is quite a natural thing to do and makes sense for any

game as introduced in Ch. 2. After all, games exist to be played! Forgetting about Ogs for a second, intuitively, composition works by inspecting the beginning of the active strategy, searching for the first *return* or *visible* move. In case of a `"ret r` move, composition ends, returning the result r. In case of a `"vis` m k move, m is passed to the opponent strategy and a *synchronization* occurs: the active strategy becomes passive, the passive strategy becomes active, the roles are switched and the composition starts again. Assuming for simplicity that both the player and the opponent strategies have a constant output family $R : \mathrm{Type}$, given a game $G : \mathrm{Game}\ I\ J$, these intuitions guide us to the following definition.

```
␣ ∥ ␣ : Strat⁺ G (λ i ↦ R) i → Strat⁻ G† (λ j ↦ R) i → Delay R
s⁺ ∥ s⁻ :=
  out := case s⁺.out
    | "ret r     := "ret r
    | "tau t     := "tau (t ∥ s⁻)
    | "vis m k := "tau (s⁻ m ∥ k)
```

Our first roadblock is now apparent: instantiating this with the Ogs game does not match the type that we wanted. There are two discrepancies. First, for two strategies to compose, they must agree on the current game position i. However, for adequacy to typecheck, we need to compose $[\![c]\!]^+_M$ with $[\![\gamma]\!]^-_M$, respectively at position $(\Gamma, \varnothing)$ and (Γ, Ω). Second, Ogs strategies as defined in Definition 4.5 have an *empty* output family, i.e., $R := \bot$. As such, no `"ret` move will ever be encountered and the composition operator we have defined will output an element of $\mathrm{Delay}\ \bot$, in other words an infinite loop!

The definition of composition definitely works as it intuitively should, so the problem lies in our treatment of Ω in the Ogs strategies. To fix this, the idea is that instead of Ω being part of the game position, it should appear in the output family. We thus update our definition of Ogs strategies, parametrizing it by this *final scope*.

$$\mathrm{Ogs}^+_O\ \Omega := \mathrm{Strat}^+_{\mathrm{Ogs}^{\mathrm{c}}_O}\ (\lambda\ (\Gamma, \Delta) \mapsto O^{\aleph}\ \Omega)$$
$$\mathrm{Ogs}^-_O\ \Omega := \mathrm{Strat}^-_{\mathrm{Ogs}^{\mathrm{c}}_O}\ (\lambda\ (\Gamma, \Delta) \mapsto O^{\aleph}\ \Omega)$$

With this fix, the composition operator can now be specialized to the following type.

$$\_ \parallel \_ : \mathrm{Ogs}^+_O\ \Omega\ (\Gamma, \Delta) \to \mathrm{Ogs}^-_O\ \Omega\ (\Gamma, \Delta) \to \mathrm{Delay}\ (O^{\aleph}\ \Omega)$$

This is already much more satisfying, although we will need to fix the machine strategy and the active and passive Ogs interpretations to take this new parameter Ω and the associated return moves into account.

There is however one more roadblock, much more insidious. To understand it, we need to dive yet deeper into the correctness proof. Proving congruence of composition will be entirely straightforward and the meat of the correctness proof is concentrated in the much trickier adequacy proof. Attacking adequacy directly, by starting to construct a bisimulation, is largely unfeasible because of the complexity of the right hand side. Hence, we again need to cut this statement into smaller pieces and devise a more structured proof strategy. As it happens, composition can be presented as the fixed point of an equation in the Delay monad, in the sense of §2.6. Moreover, without too much work we can show that the left-hand side, eval-to-obs $c[\gamma]$, seen as a function of c and γ, is also a fixed point of the same equation. Then, since both sides are fixed points of the same equation, to conclude adequacy it is sufficient to show that this composition equation has unicity of fixed points (w.r.t. strong bisimilarity). To ensure this, we build upon our new theory of fixed points and prove that the composition equation is *eventually guarded*.

What eventual guardedness means in this case, is that synchronizations (move exchanges) do not happen *too often*. More precisely, in the output of composition, silent moves have two sources: the ones arising from seeking the next non-silent move of the active strategy, and the ones arising from a synchronization. Then, eventual guardedness of composition means that every so many synchronizations we can find a guard, i.e., a "move-seeking" ᵀtau. In other words, "move-seeking" ᵀtau happen infinitely often.

This is no small property. It does not hold for the composition of arbitrary strategies, but only for the composition of strategies verifying a weak form of *visibility*. Essentially, visibility mandates that in a position (Γ, Δ) when a strategy is queried on a given variable in its scope, say $i : \Gamma \ni \alpha$, it must only query variables in Δ that were introduced *before* i during the play. However, and this is the problem, because we have kept the two scopes Γ and Δ separate, it is not evident which variables in Δ were introduced before some given variable in Γ as they are not stored contiguously.

Our solution to this second problem is conceptually quite simple: we will switch to a more informative set of positions for the Ogs game. Instead of using a pair of scopes, we will use a single *sequence* of scopes, containing an alternation of scopes of variables introduced by each player, hence keeping track of their relative order.

Remark 4.15: Note that to avoid getting too deep into the theory of Ogs strategies, we will entirely side-step the definition of the visibility condition and instead only prove eventual guardedness for composition of two instances of the machine strategy, which happens to behave satisfyingly.

The next section is thus devoted to the definition of an Ogs game refined with our two patches: final moves and interlaced positions. We will then fix the machine strategy, properly define composition and state the correctness theorem.

4.2 A Refined Ogs Model

4.2.1 Interlaced Positions

As explained in the previous section, instead of tracking the Ogs position using two scopes, where after each move the fresh increment is concatenated into one of the components, we now keep a common *list* $\Psi : \mathrm{Ctx}\ S$ of these context increments. Such lists $\varepsilon \blacktriangleright \Gamma_0 \blacktriangleright \Delta_0 \blacktriangleright \Gamma_1 \blacktriangleright \Delta_1 \blacktriangleright \ldots$ will thus contain groups of scopes of fresh variables introduced alternatively by each player. Hence, the concatenation of all the *even* elements forms the scope of variables introduced by the currently passive player, while the concatenation of the *odd* elements contains the variables introduced by the currently active player. Let us define the necessary utilities.

Definition 4.16 (Interlaced Ogs Context):
Given a set $S : \mathrm{Type}$, the set of *interlaced Ogs contexts* is given by $\mathrm{Ctx}\ S$.

Assuming a scope structure $\mathrm{Scope}_T\ S$, further define the *even* and *odd concatenation maps* $\downarrow^+, \downarrow^- : \mathrm{Ctx}\ S \to S$ as follows.

$$\begin{aligned}
\downarrow^+ \varepsilon &:= \varnothing \\
\downarrow^+ (\Psi \blacktriangleright \Gamma) &:= \downarrow^- \Psi +\!\!+ \Gamma \\
\\
\downarrow^- \varepsilon &:= \varnothing \\
\downarrow^- (\Psi \blacktriangleright \Gamma) &:= \downarrow^+ \Psi
\end{aligned}$$

We can now give the definition of the refined Ogs game.

Definition 4.17 (Ogs Game):
Assuming a scope structure $\mathrm{Scope}_T\ S$, given a binding family $O : \mathrm{Bind}_T\ S$, the *Ogs game* is defined as follows.

$$\begin{aligned}
&\mathrm{Ogs}^{\mathrm{HG}}_O : \mathrm{HGame}\ (\mathrm{Ctx}\ S)\ (\mathrm{Ctx}\ S) := \\
&\quad \left[\begin{array}{l} \mathrm{Move}\ \Psi := O^{\aleph}\ \downarrow^+ \Psi \\ \mathrm{next}\ \{\Psi\}\ o := \Psi \blacktriangleright \mathrm{holes}^{\aleph}\ o \end{array}\right. \\
\\
&\mathrm{Ogs}^{\mathrm{G}}_O : \mathrm{Game}\ (\mathrm{Ctx}\ S)\ (\mathrm{Ctx}\ S) := \\
&\quad \left[\begin{array}{l} \mathrm{client} := \mathrm{Ogs}^{\mathrm{HG}}_O \\ \mathrm{server} := \mathrm{Ogs}^{\mathrm{HG}}_O \end{array}\right.
\end{aligned}$$

4.2.2 Final Moves and Composition

Besides refining the Ogs positions, our second patch is to add *final moves* to the Ogs strategies. Intuitively, Ogs strategies will now be parametrized by a *final scope* Ω, and will be allowed to use "ret moves to play a named observation on Ω. While these moves are quite similar to the usual ones (also being named observations), they bear no continuation. Instead, they should be thought of as *final moves*, ending the game.

Definition 4.18 (Ogs Strategies):
Assuming an abstract scope structure $\mathrm{Scope}_T\ S$, given a binding family $O : \mathrm{Bind}_T\ S$ and a scope $\Omega : S$, *active and passive Ogs strategy over O with final scope Ω* are given as follows.

$$\mathrm{Ogs}^+_O\ \Omega := \mathrm{Strat}^+{}_{\mathrm{Ogs}^{\mathrm{e}}_O}\ (\lambda\ \Psi \mapsto O^{\mathrm{x}}\ \Omega)$$

$$\mathrm{Ogs}^-_O\ \Omega := \mathrm{Strat}^-{}_{\mathrm{Ogs}^{\mathrm{e}}_O}\ (\lambda\ \Psi \mapsto O^{\mathrm{x}}\ \Omega)$$

As explained before, this is now enough to define a meaningful composition operator. However, instead of the direct construction we have shown earlier, we will construct composition as the fixed point of an equation (see §2.6) in the Delay monad. This construction of composition as the solution of an equation system will allow us to be more precise during its manipulation. As the composition operator has two arguments, to express it as the fixed point of an equation system, we first need to uncurry it. The type of its uncurried argument is a bit of a mouthful, so we express it with a small gadget which will also be useful later on.

Definition 4.19 (Family Join):
Define the *family join* operator as follows.

$$\_ \bowtie \_\ \{I\} : \mathrm{Type}^I \to \mathrm{Type}^I \to \mathrm{Type}$$

$$X \bowtie Y := (i : I) \times X\ i \times Y\ i$$

Borrowing from the similarly structured named observations, we will use the same constructor notation $x \blacksquare y := (i, x, y)$ with the first component i left implicit.

The domain of the Ogs composition function can now be expressed as the family join of active and passive Ogs strategies $(\mathrm{Ogs}^+_O\ \Omega) \bowtie (\mathrm{Ogs}^-_O\ \Omega)$. We follow up with the composition equation.

Definition 4.20 (Composition Equation):
Assuming $\mathrm{Scope}_T\ S$, given $O : \mathrm{Bind}\ S\ T$ and $\Omega : S$, define the *composition equation* coinductively as follows, with $Arg := (\mathrm{Ogs}^+_O\ \Omega) \bowtie (\mathrm{Ogs}^-_O\ \Omega)$.

```
compo-eqn : Arg → Delay (Arg + O^𝕊 Ω)
compo-eqn (s⁺ ▪ s⁻) :=
  out := case s⁺.out
  | 'ret r     := 'ret (inr r)
  | 'tau t     := 'tau (compo-eqn (t ▪ s⁻))
  | 'vis m k := 'ret (inl (s⁻ m ▪ k))
```

Remark 4.21: Note the different treatment of iteration in the 'tau case and in the 'vis case. In the 'tau case, we emit a 'tau move, guarding a *coinductive* self-call, while in the 'vis case, we instead return into the left component of the equation, in a sense of performing a *formal* self-call.

One way to look at it, is that the interaction works by seeking the first visible move (or return move) of the active strategy and that an interaction step (i.e. a formal loop of the equation) should only happens when both strategies truly *interact*.

More pragmatically, much of the work for proving Ogs correctness will be dedicated to showing that this equation admits a *strong* fixed point, i.e., that adding a 'tau node to guard the self-call in the 'vis case is *not* required. On the other hand, adding the 'tau node in the 'tau case is indeed always necessary.

With this equation in hand we can readily obtain a fixed point by iteration, although only w.r.t. weak bisimilarity.

Definition 4.22 (Composition Operator):
Assuming $\mathrm{Scope}_T\ S$, given $O : \mathrm{Bind}\ S\ T$ and $\Omega : S$, define the *composition operator* by iteration (Definition 2.69) of the interaction equation.

```
compo : (Ogs⁺_O Ω) ⋈ (Ogs⁻_O Ω) → Delay (O^𝕊 Ω)
compo := iter_compo-eqn

_ ∥ _ {Ψ} : Ogs⁺_O Ω Ψ → Ogs⁻_O Ω Ψ → Delay (O^𝕊 Ω)
s⁺ ∥ s⁻ := compo (s⁺ ▪ s⁻)
```

This concludes our abstract constructions on the refined Ogs game.

4.2.3 Precise Scopes for the Machine Strategy

Thankfully, the axiomatization of language machines has been left intact by our two patches to the Ogs game. We however need to modify the machine strategy. First, we need to take into account the final scope Ω, and second, we need to take advantage of the new information available in the positions.

To avoid some clutter, in this section we globally set a scope structure $\mathrm{Scope}_T\ S$, a binding family of observations $O : \mathrm{Bind}\ S\ T$, two families $V : \mathrm{Type}^{S,T}$ and $C : \mathrm{Type}^{S}$ such that $\mathrm{PointedRenModule}\ V$ and $\mathrm{RenModule}\ C$, and a language machine $M : \mathrm{LangMachine}\ O\ V\ C$.

Recall that in the naive Ogs game, machine strategy states were defined as follows.

$$\mathrm{ogs}^+\ (\Gamma, \Delta) := C\ \Gamma \times (\Delta \multimap[V] \Gamma)$$
$$\mathrm{ogs}^-\ (\Gamma, \Delta) := \Gamma \multimap[V] \Delta$$

With the new interlaced game positions we can still recompute the two scopes, so that adding the final scope Ω to the mix, we could be tempted to define the new states as follows.

$$\mathrm{ogs}^+\ \Psi := C\ (\Omega +\!\!+ \downarrow^+\Psi) \times (\downarrow^-\Psi \multimap[V] (\Omega +\!\!+ \downarrow^+\Psi))$$
$$\mathrm{ogs}^-\ \Psi := \downarrow^+\Psi \multimap[V] (\Omega +\!\!+ \downarrow^-\Psi)$$

This would indeed work, but now that we have more information we can be much more precise in tracking the scopes of each value stored in the assignments. This is quite important since every ounce of precise specification we can cram into the typing will be something less to worry about during manipulation and proofs. Taking a step back, let us consider what must actually be stored inside these assignments. Taking the point of view of the machine strategy, at every point of the game where we play a move, we have a normal form, we emit its named observation part and we must remember the filling assignment part. As such, the exact scope used by this filling is the opponent scope *at that point in the game* (and as always the final scope Ω). We concretize this idea with the following definition of *telescopic environment*, defined inductively over the interleaved Ogs position.

Definition 4.23 (Telescopic Environments):
Given a final scope $\Omega : S$, *active and passive telescopic environments* are given by the two mutually defined inductive families

$$\mathrm{data}\ \mathrm{Tele}^+_\Omega : \mathrm{Ctx}\ S \to \mathrm{Type}$$
$$\mathrm{data}\ \mathrm{Tele}^-_\Omega : \mathrm{Ctx}\ S \to \mathrm{Type}$$

given by the following constructors.

$$\frac{}{\varepsilon^+ : \mathrm{Tele}^+_\Omega\ \varepsilon} \qquad \frac{e : \mathrm{Tele}^-_\Omega\ \Psi}{e \blacktriangleright^+ \circ : \mathrm{Tele}^+_\Omega\ (\Psi \blacktriangleright \Gamma)}$$

$$\frac{}{\varepsilon^- : \mathrm{Tele}^-_\Omega\ \varepsilon} \qquad \frac{e : \mathrm{Tele}^+_\Omega\ \Psi \quad \gamma : \Gamma \multimap[\ \mathrm{Val}\ ] (\Omega +\!\!+ \downarrow^+\Psi)}{e \blacktriangleright^- \gamma : \mathrm{Tele}^-_\Omega\ (\Psi \blacktriangleright \Gamma)}$$

Remark 4.24: The notation $e \blacktriangleright^{+} \circ$ has been chosen to evoke a sequence extension on the right by "nothing", in contrast to $e \blacktriangleright^{-} \gamma$ which does add γ to the environment. Indeed, when a player is currently active this means that they did *not* play the last move, so that what they last stored was indeed *nothing*, only recording the fact that the other side played.

This data is enough to recover the original assignments of the simple Ogs model, as witnessed by the following *collapsing functions*.

Definition 4.25 (Telescopic Collapse):
Given a final scope $\Omega : S$, define the following *active and passive telescopic environment collapsing functions*, by mutual induction.

$$\begin{aligned}
&\downarrow^{+} : \mathrm{Tele}^{+}_{\Omega}\ \Psi \to \downarrow^{-}\Psi \multimap [\ \mathrm{Val}\ ] \mapsto (\Omega \mathbin{+\!\!+} \downarrow^{+}\Psi)\\
&\downarrow^{+} \varepsilon^{+} \qquad := []\\
&\downarrow^{+} (e \blacktriangleright^{+} \circ) := (\downarrow^{-} e)[\rho]
\end{aligned}$$

$$\begin{aligned}
&\downarrow^{-} : \mathrm{Tele}^{-}_{\Omega}\ \Psi \to \downarrow^{+}\Psi \multimap [\ \mathrm{Val}\ ] \mapsto (\Omega \mathbin{+\!\!+} \downarrow^{-}\Psi)\\
&\downarrow^{-} \varepsilon^{-} \qquad := []\\
&\downarrow^{-} (e \blacktriangleright^{-} \gamma) := [\downarrow^{+} e, \gamma]
\end{aligned}$$

The shorthand ρ is the obvious renaming of type

$$(\Omega \mathbin{+\!\!+} \downarrow^{-}\Psi) \subseteq (\Omega \mathbin{+\!\!+} (\downarrow^{-}\Psi \mathbin{+\!\!+} \Gamma))$$

given by $\rho := [\text{r-cat}_l, \text{r-cat}_r[\text{r-cat}_l]]$.

The refined machine strategy is now simply a matter of adapting to the new telescopic environment, and routing the moves properly, depending on whether they concern the final scope Ω or "normal" variables.

Definition 4.26 (Machine Strategy):
Given a final scope $\Omega : S$, define the *machine strategy* as the big-step strategy $\text{mstrat}_M : \text{Big-Step-System}_{\mathrm{Ogs}^{\mathrm{c}}_{O}}\ (O^{\aleph}\ \Omega)$ defined as follows.

$$\begin{aligned}
&\text{mstrat}_M :=\\
&\quad \left[\begin{array}{l}
\text{state}^{+}\ \Psi := \mathrm{Conf}\ (\Omega \mathbin{+\!\!+} \downarrow^{+}\Psi) \times \mathrm{Tele}^{+}_{\Omega}\ \Psi\\
\text{state}^{-}\ \Psi := \mathrm{Tele}^{-}_{\Omega}\ \Psi\\
\text{play}\ (c, e) := \text{do}\\
\quad ((i \blacksquare o), \gamma) \leftarrow \text{eval}\ c\ ;\\
\quad \text{case}\ (\text{view-cat}\ i)\\
\quad \left[\begin{array}{l}
\text{v-left}\ i \quad := \text{ret}\ (\text{inl}\ (i \blacksquare o))\\
\text{v-right}\ j := \text{ret}\ (\text{inr}\ ((j \blacksquare o), (e \blacktriangleright^{-} \gamma)))
\end{array}\right.\\
\text{coplay}\ e\ (i \blacksquare o) := (\text{apply}\ (\downarrow^{-} e\ i)[\rho_1]\ o\ \rho_2, e \blacktriangleright^{+} \circ)
\end{array}\right.
\end{aligned}$$

The renamings ρ_1 and ρ_2 are defined as follows.

$$\rho_1 \coloneqq [\text{r-cat}_l, \text{r-cat}_r[\text{r-cat}_l]]$$
$$\rho_2 \coloneqq \text{r-cat}_r[\text{r-cat}_r]$$

Ogs interpretation can now be defined just like before, by unrolling of the big-step system defined by the machine strategy.

Definition 4.27 (Ogs Interpretation):
Given a final scope $\Omega : S$, define the *active and passive Ogs interpretations* as follows.

$$[\![\_]\!]^+_M\ \{\Gamma\} : C\ \Gamma \to \text{Ogs}^+_O\ \Omega\ (\varepsilon \blacktriangleright \Gamma)$$
$$[\![c]\!]^+_M \coloneqq \text{unroll}^+_{\text{mstrat}\ M}\ (c[\text{r-cat}_r], \varepsilon^- \blacktriangleright^+ \circlearrowright)$$

$$[\![\_]\!]^-_M\ \{\Gamma\} : (\Gamma \multimap [V] \to \Omega) \to \text{Ogs}^-_O\ \Omega\ (\varepsilon \blacktriangleright \Gamma)$$
$$[\![\gamma]\!]^-_M \coloneqq \text{unroll}^-_{\text{mstrat}\ M}\ (\varepsilon^+ \blacktriangleright^-\ \gamma[\text{r-cat}_l])$$

4.2.4 Correctness!

Finally we arrive at correctness. The statement is still the same as for the simple Ogs model:

$$\forall\ \{\Gamma\}\ (c_1\ c_2 : C\ \Gamma) \to [\![c_1]\!]^+_M \approx [\![c_2]\!]^+_M \to c_1 \approx^{\Omega}_{\text{sub}} c_2$$

Now, though, we will not stop at the mere statement, but provide the actual theorem. As such, we need to introduce a couple hypotheses on which the theorem depends.

Respecting Substitution Until now, we have required very little on the language machine. For the machine strategy construction, we have required a renaming structure on values and configurations, while for the correctness statement we have required substitution monoid and module structures on values and configurations. In both cases, we did not constrain in any way M.eval and M.apply, this is of course unrealistic! For correctness to work, we will crucially need to know that both maps of the machine respect substitution. Let us introduce these core hypotheses.

Definition 4.28 (Language Machine Respects Substitution):
Assume a scope structure $\text{Scope}_T\ S$, a binding family $O : \text{Bind}\ S\ T$, two families V, C, and a language machine $M : \text{LangMachine}\ O\ V\ C$ such that moreover $\text{SubstMonoid}\ V$ and $\text{SubstModule}_V\ C$. Define the embedding of normal forms into configurations as follows.

It would definitely be interesting to define language machines respecting *renamings*, but as substitutions subsume renamings we will skip over this notion.

```
nf-emb : Nf_V^O → C
nf-emb ((i ▪ o), γ) := M.apply (var i) o γ
```

Then, the language machine M *respects substitution* if there is an instance to the following typeclass.

Note that the laws eval-sub and eval-nf are specified w.r.t. *strong bisimilarity*, with *extensional equality* at the leaves (which in both cases are normal forms).

```
class LangMachineSub M :=
  [ eval-sub {Γ Δ} (c : C Γ) (σ : Γ –[V]→ Δ)
      : M.eval c[σ] ≅ (M.eval c >>= λ n ↦ M.eval (nf-emb n)[σ])
    apply-sub {Γ Δ α} (v : V Γ α) o γ (σ : Γ –[V]→ Δ)
      : M.apply v[σ] o γ[σ] ≈ (M.apply v o γ)[σ]
    eval-nf {Γ} (n : Nf_V^O Γ)
      : M.eval (nf-emb n) ≅ ret n
```

Our statement of the law apply-sub should be relatively straightforward. However, eval-sub is a bit more tricky. Indeed, there is no hope of stating a simple law such as

$$M.\mathrm{eval}\ c[\gamma] \cong (M.\mathrm{eval}\ c)[\gamma]$$

since it is a well-known fact that normal forms are never stable by substitution. Instead, after evaluating c to a normal form n, we need to embed it into configurations, substitute it *and then* evaluate the result again. In other words, we perform a *hereditary substitution*.

Remark 4.29: The last law eval-nf should have a clear meaning: it justifies calling normal forms *normal* as it requires them to be fully evaluated. It might be a bit surprising to find it in this package since it does not seem to have anything to do with substitution. The reason for including it here, is that assuming LangMachineSub M, although there is no hope of defining a substitution operator on normal forms, we can show that the family $\Gamma \mapsto \mathrm{Delay}\ (\mathrm{Nf}_V^O\ \Gamma)$ is a substitution module over values. Its substitution action is a form of hereditary substitution: first evaluate the delayed normal form, embed it into configurations, substitute it using the module structure of configurations, and further evaluate the result.

```
NfSub : SubstModule_V (Delay ∘ Nf_V^O)
NfSub :=
  [ act u σ := u >>= λ n ↦ M.eval (nf-emb n)[σ]
  ..
```

The proof for identity law of this substitution module structure crucially depends on eval-nf. Given $u : \mathrm{Delay}\ (\mathrm{Nf}_V^O\ \Gamma$, its statement is the following.

$$u \gg\!= \lambda\, n \mapsto M.\mathrm{eval}\ (\mathrm{nf\text{-}emb}\ n)[\mathrm{var}] \quad \cong \quad u$$

Then, by monad laws, the above is easily reduced to proving that for any $n : \mathrm{Nf}_V^O\ \Gamma$

$$M.\mathrm{eval}\ (\mathrm{nf\text{-}emb}\ n)[\mathrm{var}] \quad \cong \quad \mathrm{ret}\ n$$

which can be further simplified using the identity substitution law of configurations to

$$M.\mathrm{eval}\ (\mathrm{nf\text{-}emb}\ n) \quad \cong \quad \mathrm{ret}\ n$$

in other words, exactly eval-nf.

We conjecture that this hints at less ad-hoc route for formalizing language machines respecting substitution. Instead of eval-sub and apply-sub, we would require that $\mathrm{Delay} \circ \mathrm{Nf}_V^O$ forms a substitution module over values, and that eval and nf-emb are substitution module morphisms.

We are now done with the core hypotheses, but there are two more technical hypotheses we must require.

Decidable Variables The argument for the eventual guardedness of the composition equation requires us to case-split on whether or not some given value is a variable. "Being a variable" can be neatly expressed as belonging to the image of var, in other words, exhibiting an element of the fiber of var over some value.

$$\mathrm{is\text{-}var}\ \{\Gamma\ \alpha\} : V\ \Gamma\ \alpha \to \mathrm{Type}$$
$$\mathrm{is\text{-}var}\ v := \mathrm{Fiber}\ \mathrm{var}\ v$$

Then, our case-splitting requirement can be formalized by asking that is-var v is decidable for all v. We define the standard decidability data type

$$\mathrm{data}\ \mathrm{Decidable}\ (X : \mathrm{Type}) : \mathrm{Type}$$

by the following constructors.

$$\frac{p : X}{\mathrm{yes}\ p : \mathrm{Decidable}\ X} \qquad \frac{p : X \to \mathbf{0}}{\mathrm{no}\ p : \mathrm{Decidable}\ X}$$

We package this into a typeclass, together with some additional requirements making is-var well-behaved.

Definition 4.30 (Decidable Variables):
Assume a scope structure $\mathrm{Scope}_T\ S$ and a family $V : \mathrm{Type}^{S,T}$ with a pointed renaming structure PointedRenModule V, V *has decidable variables* if there is an instance to the following typeclass.

```
class DecidableVar V :=
  [ is-var? {Γ α} (v : V Γ α) : Decidable (is-var v)
  | is-var-irr {Γ α} {v : V Γ α} (p₁ p₂ : is-var v) : p₁ = p₂
  | is-var-ren {Γ Δ α} (v : V Γ α) (ρ : Γ ⊆ Δ) : is-var v[ρ] → is-var v
```

is-var-irr is quite powerful, it states that there is at most one way to show that a value is a variable. Assuming unicity of identity proofs (axiom K) on values, this is equivalent to the more common fact that var is injective. The second law, is-var-ren, validates the fact that if the renaming of a value is a variable, then it must have been a variable in the first place. Note that we do not need to ask the reverse implication: since $(\text{var } i)[\rho] \approx \text{var } (\rho\ i)$, we can deduce that is-var $v \to$ is-var $v[\rho]$.

Finite Redexes The last technical hypothesis is perhaps the most mysterious. To me it was, at least, and I was quite surprised when I realized I could *still* not conclude the eventual guardedness proof of the composition equation with the two previously presented hypotheses.

In essence, what interlaced positions and telescopic environment buy us, is a bound on the number of what we call *chit-chats*. These chit-chats are synchronizations events during the composition, for which the exchanged move $(i \blacksquare o)$ targets a variable i that is associated in the opponent's environment to a value which happens to be some variable j. As such, the composition continues with some new active configuration $M.\text{apply } (\text{var } j)\ o\ \gamma$, which by eval-nf evaluates instantly to $((j \blacksquare o), \gamma)$. Hence, in case of such a chit-chat, the original move $(i \blacksquare o)$ is immediately "bounced" back to the original player with the move $(j \blacksquare o)$. The observation o is the same, but the new variable j is different, in fact it must have been introduced *before* i, which limits the number of bounces. Recalling that to prove eventual guardedness, the goal is to find a ▷tau move in the reduction of the active configuration, it is quite nice to be able to limit the number of such bounces.

As such, we can assume that after some amount of chit-chat, the active configuration will be of the form $M.\text{apply } v\ o\ \gamma$, where v *is not* a variable. But this is the surprise, at this point we are completely stuck without any further hypothesis. The natural thing to do would be to postulate that such a configuration $M.\text{apply } v\ o\ \gamma$ where v is not a variable has a *redex*, in the sense that its evaluation necessarily starts with a ▷tau move, i.e., a reduction step. This requirement would intuitively make sense: o typically stands for an elimination, so applying an elimination to something which is not a variable should yield a redex. Alas, I realized that this was severely restricting the language machine. Even the simply-typed λ-calculus fails this hypothesis*! We thus need a weaker hypothesis.

* To be completely honest, the particular example of the λ-calculus can be made to verify the redex hypothesis by designing the observations more smartly, but still, we would like to have maximal latitude to design the language machine as we wish.

Concretely the statement of the hypothesis is quite technical, but the idea is relatively simple. We simply ask for a bound on the number of consecutive times that the redex hypothesis can fail. As such, although the configuration may not have a redex *right now*, we know that after some more composition steps we will eventually find one.

Technically, it is formulated as follows. If the redex hypothesis fails, it means that the configuration $M.\mathrm{apply}\ v\ o\ \gamma$ happens to be some normal form which will thus immediately trigger a new message, say (i, o'). This defines a relation $o' \prec o$ on observations. We then require that this relation is well-founded.

Definition 4.31 (Finite Redexes):
Assume a scope structure $\mathrm{Scope}_T\ S$, a binding family $O : \mathrm{Bind}\ S\ T$, two families V, C, and a language machine $M : \mathrm{LangMachine}\ O\ V\ C$. Pose $\tilde{O} := (\alpha : T) \times O.\mathrm{Op}\ \alpha$ and define the *redex failure relation* $\_ \prec \_ : \mathrm{Rel}\ \tilde{O}\ \tilde{O}$ as follows.

$$\frac{\begin{array}{c} i : \Gamma \ni \alpha_1 \quad o_1 : O.\mathrm{Op}\ \alpha_1 \quad \gamma_1 : O.\mathrm{holes}\ o_1 \dashv[V]\mapsto \Gamma \quad v : V\ \Gamma\ \alpha_2 \\ o_2 : O.\mathrm{Op}\ \alpha_2 \quad \gamma_2 : O.\mathrm{holes}\ o_2 \dashv[V]\mapsto \Gamma \quad H_1 : \text{is-var}\ v \to \bot \\ H_2 : M.\mathrm{eval}\ (M.\mathrm{apply}\ v\ o_2\ \gamma_2) \cong \mathrm{ret}\ ((i \blacksquare o_1), \gamma_1) \end{array}}{\mathrm{bad}\ H_1\ H_2 : o_1 \prec o_2}$$

Then, the machine M *has finite redexes* if the redex failure relation is well-founded.

$$\mathrm{FiniteRedexes}\ M := \mathrm{WellFounded} \prec$$

Remark 4.32: Recall that in type theory, well-foundedness is defined in terms of inductive *accessibility*.

$$\mathrm{data}\ \mathrm{Acc}\ \{X\}\ (R : \mathrm{Rel}\ X\ X)\ (a : X) : \mathrm{Prop}$$
$$[\ \mathrm{acc} : (\forall\ \{b\} \to R\ b\ a \to \mathrm{Acc}\ R\ b) \to \mathrm{Acc}\ R\ a$$

$$\mathrm{WellFounded}\ \{X\} : \mathrm{Rel}\ X\ X \to \mathrm{Prop}$$
$$\mathrm{WellFounded}\ R := (a : X) \to \mathrm{Acc}\ R\ a$$

Then, well-founded induction is simply obtained by induction on the accessibility proof.

We are finally in a position to state the correctness theorem.

Theorem 4.33 (Ogs Correctness):
Assuming
- a scope structure $\mathrm{Scope}_T\ S$,
- a binding family $O : \mathrm{Bind}\ S\ T$,
- a substitution monoid of values $V : \mathrm{Type}^{S,T}$ with decidable variables,

- a substitution module over values of configurations $C : \mathrm{Type}^S$,
- a language machine $M :$ LangMachine $O\ V\ C$ which respects substitution and which has finite redexes,

then, for any final scope $\Omega : S$, the OGS interpretation is correct with respect to substitution equivalence at Ω, i.e., the following property holds.

$$\forall\ \{\Gamma\}\ (c_1\ c_2 : C\ \Gamma) \to [\![c_1]\!]^+_M \approx [\![c_2]\!]^+_M \to c_1 \approx^{\Omega}_{\mathrm{SUB}} c_2$$

This correctness theorem concludes our presentation of the OGS game and language machine model. What is left to do is to actually prove it, and this is the subject of the next chapter.

5 OGS Correctness Proof

In Ch. 4 we have constructed the OGS model, interpreting configurations of a language machine into OGS strategies. Then we have given the correctness theorem (Theorem 4.33), stating that when two configurations are equivalent in the OGS model (i.e., when they have weakly bisimilar strategies), then they are *substitution equivalent*, the concretization of observational equivalence for language machines. The current chapter is now devoted to providing the actual proof of this statement.

For the entirety of this chapter, we will globally assume all of the hypotheses of the correctness theorem. Hence we assume given

- a scope structure $\mathrm{Scope}_T\ S$,
- a binding family $O : \mathrm{Bind}\ S\ T$,
- a substitution monoid of values $V : \mathrm{Type}^{S,T}$ with decidable variables
- a substitution module over values of configurations $C : \mathrm{Type}^{S}$
- a language machine $M : \mathrm{LangMachine}\ O\ V\ C$ which respects substitution and which has finite redexes,
- a final scope $\Omega : S$.

Spelled out more formally, we assume given elements of the following typeclasses.

$\mathrm{SubstMonoid}\ V \quad \mathrm{DecidableVar}\ V \quad \mathrm{SubstModule}_V\ C$
$\mathrm{LangMachineSub}\ M \quad \mathrm{FiniteRedexes}\ M$

5.1 Proof Outline

We have already somewhat hinted at the proof strategy and the most important lemmas in various places. Let us recapitulate the blueprint. To prove the correctness, we will do a detour on composition, as indeed correctness follows from two properties of composition, *congruence* and *adequacy*.

Definition 5.1 (Congruence for Composition):
Weak bisimilarity is a *congruence* for composition if for all $\Psi : \mathrm{Ctx}\ S$, $s_1^+, s_2^+ : \mathrm{OGS}_O^+\ \Omega\ \Psi$ and $s^- : \mathrm{OGS}_O^-\ \Omega\ \Psi$, the following property holds.

$$s_1^+ \approx s_2^+ \to (s_1^+ \parallel s^-) \approx (s_2^+ \parallel s^-)$$

Definition 5.2 (Adequacy of Composition):
The composition operator is *adequate* if for all $\Gamma : S$, $c : C\,\Gamma$ and $\gamma : \Gamma \dashv[V]\mapsto \Omega$, the following property holds.

$$\text{eval-to-obs}\ c[\gamma] \approx (\llbracket c \rrbracket^{+}_{M} \parallel \llbracket \gamma \rrbracket^{-}_{M})$$

Congruence is quite straightforward to prove, it is yet another instance of our hard to read and boring relational statements! The most important task is the proof of adequacy. The idea to prove adequacy boils down to two arguments: first we show that $\lambda\ c\ \gamma \mapsto \text{eval-to-obs}\ c[\gamma]$ is a fixed point of the composition equation and second we show that the composition equation is eventually guarded, hence has unique fixed points. At this point however, eval-to-obs being a fixed point of composition does not make much sense, since the composition equation operates on pairs of OGS strategies while eval-to-obs takes a configuration and an assignment. To fix this issue we will make two patches, respectively bringing composition and eval-to-obs to a common meeting point: pairs of machine strategy states.

First, we will define a second composition equation, called *machine composition*, operating not on *opaque* OGS strategies, but specifically on pairs of active and passive states of the machine strategy. The definition will be essentially the same, but instead of peeling off layers of a coinductive tree to obtain the next moves, we will call the play and coplay functions of the big-step system.

Second, notice that in the function $\lambda\ c\ \gamma \mapsto \text{eval-to-obs}\ c[\gamma]$, the two arguments c and γ are special cases respectively of active and passive machine strategy states (more specifically *initial* states). We will thus generalize this function to arbitrary pairs of active and passive machine strategy states. Intuitively, this function will *zip* together the telescopic environments of the two states, then substitute the active state's configuration by the resulting assignment and finally apply eval-to-obs.

More precisely, to conclude adequacy we will show that
- *machine composition* is strongly bisimilar to composition,
- *zip-then-eval-to-obs* is a strong fixed point of *machine composition*,
- *machine composition* is eventually guarded.

5.2 Machine Strategy Composition

Recall that in Definition 4.20 we formalized the composition equation as a map of type

$$\text{Arg} \to \text{Delay}\ (O^{\aleph}\ \Omega + \text{Arg})$$

where the arguments

$$\text{Arg} := \text{Ogs}^+_O\ \Omega \bowtie \text{Ogs}^-_O\ \Omega$$

consisted of pairs of an active strategy $s^+ : \text{Ogs}^+_O\ \Omega\ \Psi$ and a passive strategy $s^- : \text{Ogs}^-_O\ \Omega\ \Psi$, both over the same interleaved scope Ψ. To specialize composition to the machine strategy, we will simply swap out $\text{Ogs}^+_O\ \Omega$ and $\text{Ogs}^-_O\ \Omega$ for active and passive machine strategy states.

Definition 5.3 (Machine Strategy Composition):
First, define the *arguments* of the machine strategy composition as follows.

$$\text{MCArg} := \text{mstrat}_M\ .\text{state}^+ \bowtie \text{mstrat}_M\ .\text{state}^-$$

Then, define the *machine strategy composition equation* as follows.

$$\text{m-compo-eqn} : \text{MCArg} \to \text{Delay}\ (O^{\aleph}\ \Omega + \text{MCArg})$$
$$\text{m-compo-eqn}\ (s^+ \blacksquare s^-) := \text{aux} \langle\$\rangle\ \text{mstrat}_M\ .\text{play}\ s^+$$
$$\text{where}$$
$$\begin{bmatrix} \text{aux}\ (\text{inl}\ r) & := \text{inl}\ r \\ \text{aux}\ (\text{inr}\ (m, k)) & := \text{inr}\ ((\text{mstrat}_M\ .\text{coplay}\ s^-\ m) \blacksquare k) \end{bmatrix}$$

Finally, define the *machine strategy composition* as the following iteration.

$$\text{m-compo} : \text{MCArg} \to \text{Delay}\ (O^{\aleph}\ \Omega)$$
$$\text{m-compo} := \text{iter}_{\text{m-compo-eqn}}$$

We then link this new composition of machine strategies with the more general one, introduced in the previous chapter.

Lemma 5.4 (Machine Composition and Composition):
For any $\Psi : \text{Ctx}\ S$, $s^+ : \text{mstrat}_M\ .\text{state}^+\ \Psi$ and $s^- : \text{mstrat}_M\ .\text{state}^-\ \Psi$ the following property holds.

$$\text{m-compo}\ (s^+ \blacksquare s^-) \cong \left(\text{unroll}^+{}_{\text{mstrat}_M}\ s^+ \parallel \text{unroll}^-{}_{\text{mstrat}_M}\ s^-\right)$$

Proof: As both sides are constructed by (unguarded) iteration, this lemma is proven by an application of Lemma 2.72. In other words, we show that their respective equation systems operate in lockstep, with some relation on their argument being preserved. More precisely, define the following relation between the arguments of compositions and machine composition.

$$R : \text{MCArg} \to \text{Ogs}^+_O\ \Omega \bowtie \text{Ogs}^-_O\ \Omega \to \text{Prop}$$
$$R\ (s_1^+ \blacksquare s_1^-)\ (s_2^+ \blacksquare s_2^-) :=$$
$$\left(\text{unroll}^+{}_{\text{mstrat}_M}\ s_1^+ = s_2^+\right) \times \left(\text{unroll}^-{}_{\text{mstrat}_M}\ s_1^- = s_2^-\right)$$

Then, for any $a : \text{MCArg}$ and $b : \text{Ogs}^+_O\ \Omega \bowtie \text{Ogs}^-_O\ \Omega$, we prove that $R\ a\ b \to (\text{m-compo-eqn}\ a) \approx[= +^r\ R] (\text{compo-eqn}\ b)$.

Succinctly, the above proposition is proven by inspecting the head of the configuration part of the active state a. In case it plays ⁿret, the conclusion is direct, while for ⁿtau the conclusion is by coinduction.

Then, by Lemma 2.72 we have $R\ a\ b \to (\text{m-compo}\ a) \approx (\text{compo}\ b)$, and we obtain the result by instantiating the relation witness $R\ a\ b$ by (refl, refl). ■

Finally, before moving on, we will prove a variant of the eval-sub law of language machines respecting substitution. It is a special case which will be particularly useful at several points.

Lemma 5.5 (Evaluation under Shifted Substitution):
For any $c : C\ (\Omega + \Gamma)$ and $\sigma : \Gamma \multimap[V]\mapsto \Omega$, the following proposition holds.

$$\text{eval}\ c[\text{var}, \sigma] \approx \left[\begin{array}{l} ((i \blacksquare o), \gamma) \leftarrow \text{eval}\ c\ ; \\ \text{case view-cat}\ i \\ \left[\begin{array}{ll} \text{v-left}\ j & := \text{ret}\ ((j \blacksquare o), \gamma[\text{var}, \sigma]) \\ \text{v-right}\ j & := \text{eval}\ (\text{apply}\ (\sigma\ j)\ o\ \gamma[\text{var}, \sigma]) \end{array}\right. \end{array}\right.$$

Proof: By eval-sub, unfolding the definition of nf-emb, we have the following.

$$\text{eval}\ c[\text{var}, \sigma] \approx \left[\begin{array}{l} ((i \blacksquare o), \gamma) \leftarrow \text{eval}\ c\ ; \\ \text{eval}\ (\text{apply}\ ([\text{var}, \sigma]\ i)\ o\ \gamma[\text{var}, \sigma]) \end{array}\right.$$

The right-hand sides of both the lemma and the equivalence above start by binding eval c. Hence by monotonicity of bind (Lemma 2.61) it suffices to relate their continuations. Introduce some normal form $((i \blacksquare o), \gamma)$. By case on view-cat i.

- If view-cat $i :=$ v-left j, then $[\text{var}, \sigma]\ i :=$ var j. Conclude by eval-nf, showing that $\text{eval}\ (\text{apply}\ (\text{var}\ j)\ o\ \gamma[\text{var}, \sigma]) \approx \text{ret}\ ((j \blacksquare o), \gamma[\text{var}, \sigma])$.
- If view-cat $i :=$ v-right j, then $[\text{var}, \sigma]\ i := \sigma\ j$ and we conclude by reflexivity. ■

5.3 Evaluation as a Fixed Point of Machine Composition

After defining the machine strategy composition, the next step is to generalize $\lambda\ c\ \gamma \mapsto$ eval-to-obs $c[\gamma]$ to active and passive machine strategy states and then show that the obtained function is a fixed point of our newly defined composition. Our goal is to define a function of the following type.

$$\text{zip-then-eval} : \text{MCArg} \to \text{Delay}\ (O^{\text{ᵒᵖ}}\ \Omega)$$

We start by defining the following zipping of telescopic environments. Recall how in Definition 4.25 we defined the *collapsing* of telescopic environments into usual assignments. Here the process is relatively similar, but instead of only *one* telescopic environment, we are here given *two*, which nicely mesh into one together.

Definition 5.6 (Zipping Map):
The *left-to-right and right-to-left zipping maps* of type

$$\text{␣} \curvearrowleft \text{␣}\{\Psi\} : \text{Tele}^{+}_{\Omega}\ \Psi \to \text{Tele}^{-}_{\Omega}\ \Psi \to \downarrow^{+}\ \Psi \multimap [V] \mapsto \Omega$$
$$\text{␣} \curvearrowright \text{␣}\{\Psi\} : \text{Tele}^{+}_{\Omega}\ \Psi \to \text{Tele}^{-}_{\Omega}\ \Psi \to \downarrow^{-}\ \Psi \multimap [V] \mapsto \Omega$$

are defined by mutual induction as follows.

$$\varepsilon^{+} \curvearrowleft \varepsilon^{-} := []$$
$$(a \blacktriangleright^{+} \circ) \curvearrowleft (b \blacktriangleright^{-} \gamma) := [b \curvearrowright a, \gamma[\text{var}, b \curvearrowleft a]]$$

$$\varepsilon^{+} \curvearrowright \varepsilon^{-} := []$$
$$(a \blacktriangleright^{+} \circ) \curvearrowright (b \blacktriangleright^{-} \gamma) := b \curvearrowleft a$$

Remark 5.7: Intuitively, $\curvearrowleft$ is concerned with providing hereditarily substituted values for every variable introduced by the currently passive player (i.e., the RHS), while $\curvearrowright$ provides hereditarily substituted values for every variable introduced by the currently active player (i.e., the LHS). It is thus normal that only $\curvearrowleft$ does any real work: since the LHS is always the currently active side, it did not play the last move and thus did not store anything at the top of its environment.

Before going further, let us prove the crucial property relating the zipping of two telescopic environments and their respective collapse. Recall that the collapsing functions have the following types.

$$\downarrow^{+} : \text{Tele}^{+}_{\Omega}\ \Psi \to \downarrow^{-}\Psi \multimap [V] \mapsto (\Omega \mathbin{+\!\!+} \downarrow^{+}\Psi)$$
$$\downarrow^{-} : \text{Tele}^{-}_{\Omega}\ \Psi \to \downarrow^{+}\Psi \multimap [V] \mapsto (\Omega \mathbin{+\!\!+} \downarrow^{-}\Psi)$$

Lemma 5.8 (Zip Fixed Point of Collapse):
Given $\Psi : \mathrm{Ctx}\ S$, for any $a : \mathrm{Tele}^{+}_{\Omega}\ \Psi$ and $b : \mathrm{Tele}^{-}_{\Omega}\ \Psi$, the following two statements hold.

(1) $(\downarrow^{+} a)[\mathrm{var}, a \curvearrowleft b] \approx a \curvearrowright b$
(2) $(\downarrow^{-} b)[\mathrm{var}, a \curvearrowright b] \approx a \curvearrowleft b$

Proof: The two statements are proven simultaneously, by induction on Ψ, by inspecting a and b. While they are slightly tedious, the calculations are purely equational manipulations of assignments, based on substitution monoid laws. ■

Finally, we can define the function we wanted and prove that it is a fixed point of composition.

Definition 5.9 (Zip-then-evaluate):
The *zip-then-evaluate* map is defined as follows.

$$\mathrm{zip\text{-}then\text{-}eval} : \mathrm{MCArg} \to \mathrm{Delay}\ (O^{\aleph}\ \Omega)$$
$$\mathrm{zip\text{-}then\text{-}eval}\ ((c, a) \blacksquare b) := \mathrm{eval\text{-}to\text{-}obs}\ c[\mathrm{var}, a \curvearrowleft b]$$

Theorem 5.10 (Zip-then-evaluate Fixed Point):
Zip-then-evaluate is a fixed point of the machine strategy composition equation w.r.t. strong bisimilarity, i.e., for all $x : \mathrm{MCArg}$, the following property holds.

$$(\mathrm{zip\text{-}then\text{-}eval}\ x) \cong \left(\mathrm{m\text{-}compo\text{-}eqn}\ x \gg\!= \lambda \begin{bmatrix} \mathrm{inl}\ r := \mathrm{ret}\ r \\ \mathrm{inr}\ y := \mathrm{zip\text{-}then\text{-}eval}\ y \end{bmatrix}\right)$$

Proof: Let us consider the composition argument $((c, a) \blacksquare b)$. We will reason equationally, simplifying both sides to the same computation.

Starting from the left-hand side, we rewrite as follows.

$$\begin{aligned}
&\mathrm{zip\text{-}then\text{-}eval}\ ((c, a) \blacksquare b) \\
\cong\ &\mathrm{fst}\ \langle\$\rangle\ \mathrm{eval}\ c[\mathrm{var}, a \curvearrowleft b] && \text{(def.)} \\
\cong\ &\mathrm{fst}\ \langle\$\rangle \begin{bmatrix} ((i \blacksquare o), \gamma) \leftarrow \mathrm{eval}\ c\ ; \\ \mathrm{case}\ \mathrm{view\text{-}cat}\ i \\ \begin{bmatrix} \text{v-left}\ j & := \mathrm{ret}\ ((j \blacksquare o), \gamma[\mathrm{var}, a \curvearrowleft b]) \\ \text{v-right}\ j & := \mathrm{eval}\ (\mathrm{apply}\ (\rho\ j)\ o\ \gamma[\mathrm{var}, a \curvearrowleft b]) \end{bmatrix} \end{bmatrix} && \text{(Lemma 5.5)} \\
\cong\ &\begin{bmatrix} ((i \blacksquare o), \gamma) \leftarrow \mathrm{eval}\ c\ ; \\ \mathrm{case}\ \mathrm{view\text{-}cat}\ i \\ \begin{bmatrix} \text{v-left}\ j & := \mathrm{ret}\ (j \blacksquare o) \\ \text{v-right}\ j & := \mathrm{eval\text{-}to\text{-}obs}\ (\mathrm{apply}\ ((a \curvearrowleft b)\ j)\ o\ \gamma[\mathrm{var}, a \curvearrowleft b]) \end{bmatrix} \end{bmatrix} && \text{(monad)}
\end{aligned}$$

The last computation above is now simple enough. We will remember it and seek to obtain the same starting from the right-hand side of the theorem statement. To ease the unfolding of definitions, first pose the following shorthands.

$$f := \lambda \begin{bmatrix} \mathsf{inl}\ r & := \mathsf{inl}\ r \\ \mathsf{inr}\ (m, k) & := \mathsf{inr}\ ((\mathsf{mstrat}_M\ .\mathsf{coplay}\ b\ m) \blacksquare k) \end{bmatrix}$$

$$\kappa := \lambda \begin{bmatrix} \mathsf{inl}\ r & := \mathsf{ret}\ r \\ \mathsf{inr}\ y & := \mathsf{zip\text{-}then\text{-}eval}\ y \end{bmatrix}$$

Starting from the right-hand side, we rewrite as follows.

$$\begin{aligned}
& \mathsf{m\text{-}compo\text{-}eqn}\ x \gg\!= \begin{bmatrix} \mathsf{inl}\ r & := \mathsf{ret}\ r \\ \mathsf{inr}\ y & := \mathsf{zip\text{-}then\text{-}eval}\ y \end{bmatrix} \\
\cong\ & (f \langle\$\rangle \mathsf{mstrat}_M\ .\mathsf{play}\ (c, a)) \gg\!= \kappa && \text{(def.)} \\
\cong\ & \mathsf{mstrat}_M\ .\mathsf{play}\ (c, a) \gg\!= (\kappa \circ f) && \text{(monad)} \\
\cong\ & \begin{bmatrix} x \leftarrow \begin{bmatrix} ((i \blacksquare o), \gamma) \leftarrow \mathsf{eval}\ c\ ; \\ \mathsf{case}\ (\mathsf{view\text{-}cat}\ i) \\ \begin{bmatrix} \mathsf{v\text{-}left}\ j & := \mathsf{ret}\ (\mathsf{inl}\ (j \blacksquare o)) \\ \mathsf{v\text{-}right}\ j & := \mathsf{ret}\ (\mathsf{inr}\ ((j \blacksquare o), (a \blacktriangleright^{-} \gamma))) \end{bmatrix} \end{bmatrix} \\ \kappa\ (f\ x) \end{bmatrix} && \text{(def.)} \\
\cong\ & \begin{bmatrix} ((i \blacksquare o), \gamma) \leftarrow \mathsf{eval}\ c\ ; \\ \mathsf{case}\ (\mathsf{view\text{-}cat}\ i) \\ \begin{bmatrix} \mathsf{v\text{-}left}\ j & := \kappa\ (f\ (\mathsf{inl}\ (j \blacksquare o))) \\ \mathsf{v\text{-}right}\ j & := \kappa\ (f\ (\mathsf{inr}\ ((j \blacksquare o), (a \blacktriangleright^{-} \gamma)))) \end{bmatrix} \end{bmatrix} && \text{(monad)} \\
\cong\ & \begin{bmatrix} ((i \blacksquare o), \gamma) \leftarrow \mathsf{eval}\ c\ ; \\ \mathsf{case}\ (\mathsf{view\text{-}cat}\ i) \\ \begin{bmatrix} \mathsf{v\text{-}left}\ j & := \mathsf{ret}\ (j \blacksquare o) \\ \mathsf{v\text{-}right}\ j & := \mathsf{zip\text{-}then\text{-}eval}\ (\mathsf{mstrat}_M\ .\mathsf{coplay}\ b\ (j \blacksquare o) \blacksquare (a \blacktriangleright^{-} \gamma)) \end{bmatrix} \end{bmatrix} && \text{(def.)}
\end{aligned}$$

At this point, our two rewriting chains almost match up, with only the second branch of the respective case split differing. More precisely, both computations start by binding eval c, so that by Lemma 2.61 it suffices to prove the continuations pointwise bisimilar. Then, we eliminate view-cat i. In case of v-left j we conclude by reflexivity. We now turn to the v-right j case. What is left is to prove is

$$\begin{aligned}
& \mathsf{eval\text{-}to\text{-}obs}\ (\mathsf{apply}\ ((a \curvearrowleft b)\ j)\ o\ \gamma[\mathsf{var}, a \curvearrowleft b])) \\
\cong\ & \mathsf{zip\text{-}then\text{-}eval}\ (\mathsf{mstrat}_M\ .\mathsf{coplay}\ b\ (j \blacksquare o) \blacksquare (a \blacktriangleright^{-} \gamma)).
\end{aligned}$$

First pose the following two "administrative" renamings (from the definition of the machine strategy coplay function).

$\rho_1 \coloneqq [\text{r-cat}_l, \text{r-cat}_r[\text{r-cat}_l]]$
$\rho_2 \coloneqq \text{r-cat}_r[\text{r-cat}_r]$.

Then, we start rewriting from the right-hand side.

$$\begin{aligned}
&\text{zip-then-eval}\ (\text{mstrat}_M\ .\text{coplay}\ b\ (j \blacksquare o) \blacksquare (a \blacktriangleright^- \gamma)) \\
\cong\ &\text{zip-then-eval}\ ((\text{apply}\ (\downarrow^- b\ j)[\rho_1]\ o\ \rho_2, b \blacktriangleright^+ \circlearrowright) \blacksquare (a \blacktriangleright^- \gamma)) && \text{(def.)} \\
\cong\ &\text{eval-to-obs}\ (\text{apply}\ (\downarrow^- b\ j)[\rho_1]\ o\ \rho_2)[\text{var}, (b \blacktriangleright^+ \circlearrowright) \curvearrowleft (a \blacktriangleright^- \gamma))] && \text{(def.)} \\
\cong\ &\text{eval-to-obs}\ (\text{apply}\ (\downarrow^- b\ j)[\rho_1]\ o\ \rho_2)[\text{var}, a \curvearrowright b, \gamma[\text{var}, a \curvearrowleft b]]) && \text{(def.)}
\end{aligned}$$

Pose $\sigma \coloneqq [\text{var}, a \curvearrowright b, \gamma[\text{var}, a \curvearrowleft b]]$ and continue as follows.

$$\begin{aligned}
&\cong \text{eval-to-obs}\ (\text{apply}\ (\downarrow^- b\ j)[\rho_1][\sigma]\ o\ \rho_2[\sigma])) && \text{(apply-sub)} \\
&\cong \text{eval-to-obs}\ (\text{apply}\ (\downarrow^- b\ j)[\text{var}, a \curvearrowright b]\ o\ \gamma[\text{var}, a \curvearrowleft b])) && \text{(sub laws)} \\
&\cong \text{eval-to-obs}\ (\text{apply}\ ((a \curvearrowleft b)\ j)\ o\ \gamma[\text{var}, a \curvearrowleft b])) && \text{(Lemma 5.8)}
\end{aligned}$$

This concludes our proof: although tedious, it simply rests upon:

- passing a substitution under an evaluation using eval-sub (in Lemma 5.5),
- passing a substitution under an application using apply-sub,
- the use of Lemma 5.8 to fuse collapsing and zipping,
- and a series of basic rewriting steps using the monad laws of Delay and the categorical structure of assignments. ■

5.4 Eventual Guardedness of Machine Composition

Now that we know that zip-then-eval is a fixed point of the machine strategy composition equation, we can conclude adequacy of composition if we know that the machine strategy composition equation has unique fixed points. We will deduce this from the fact that the equation is eventually guarded.

Recall that in this precise case, eventual guardedness means that every so many synchronization events of composition, we will stumble upon a silent step of one of the two strategies (i.e., a reduction step of their configuration). The number of synchronizations necessary to see such a reduction step is bounded by two imbricated arguments. First, by the finite redex property of the language machine, after some amount of synchronizations *where the observed value is not a variable*, we will find a reduction step. Second, by a visibility-like condition, after some amount of synchronizations, we will observe a value that is not a variable.

More precisely, the "visibility-like" argument states that if some variable i is associated in a telescopic environment to some variable j, then the *depth* of j is

strictly smaller than the depth of i, where the depth denotes the number of moves after which the variable was introduced. Let us define this.

Definition 5.11 (Variable Depth):
Define the *positive and negative depth functions* by mutual induction as follows.

$$\mathrm{depth}^+\ \Psi\ \{\alpha\} : \downarrow^+\Psi \ni \alpha \to \mathbb{N}$$

$$\mathrm{depth}^+\ \varepsilon\qquad\quad i := \mathrm{case}\ (\mathrm{view\text{-}emp}\ i)\ []$$

$$\mathrm{depth}^+\ (\Psi \blacktriangleright \Gamma)\ i := \mathrm{case}\ (\mathrm{view\text{-}cat}\ i)$$

$$\begin{bmatrix} \text{v-left}\ i & := \mathrm{depth}^-\ \Psi\ i \\ \text{v-right}\ i & := 1 + \mathrm{length}\ \Psi \end{bmatrix}$$

$$\mathrm{depth}^-\ \Psi\ \{\alpha\} : \downarrow^-\Psi \ni \alpha \to \mathbb{N}$$

$$\mathrm{depth}^-\ \varepsilon\qquad\quad i := \mathrm{case}\ (\mathrm{view\text{-}emp}\ i)\ []$$

$$\mathrm{depth}^-\ (\Psi \blacktriangleright \Gamma)\ i := \mathrm{depth}^+\ \Psi\ i$$

Lemma 5.12 (Depth Decreases):
The depth of a variable stored in a telescopic environment is strictly smaller than its index. More precisely, the following two statements are true.

(1) Given an environment $a : \mathrm{Tele}^+_\Omega\ \Psi$ and variables $i : \downarrow^+\Psi \ni \alpha$ and $j : \downarrow^-\Psi \ni \alpha$ such that $\downarrow^+ a\ i = \mathrm{var}\ (\text{r-cat}_r\ j)$, then

$$\mathrm{depth}^-\ j < \mathrm{depth}^+\ i$$

(2) Given an environment $a : \mathrm{Tele}^-_\Omega\ \Psi$ and variables $i : \downarrow^-\Psi \ni \alpha$ and $j : \downarrow^+\Psi \ni \alpha$ such that $\downarrow^- a\ i = \mathrm{var}\ (\text{r-cat}_r\ j)$, then

$$\mathrm{depth}^+\ j < \mathrm{depth}^-\ i$$

Proof: The two statements are proven simultaneously, by direct induction on the graph of the depth function at i. ■

We can now prove eventual guardedness.

Theorem 5.13 (Composition Eventually Guarded):
The machine strategy composition equation m-compo-eqn is eventually guarded.

Proof: More formally, the goal is to prove eqn-ev-guarded m-compo-eqn, i.e.,

$$(a : \mathrm{MCArg}) \to \mathrm{ev\text{-}guarded}_{\text{m-compo-eqn}}\ (\text{m-compo-eqn}\ a).$$

Introducing and fully destructing the argument a as $((c, \sigma^+) \blacksquare \sigma^-)$, obtain $c : C\ (\Omega +\!\!+ \downarrow^+\Psi)$, $\sigma^+ : \mathrm{Tele}^+_\Omega\ \Psi$ and $\sigma^- : \mathrm{Tele}^-_\Omega\ \Psi$. As m-compo-eqn starts by evaluating c, lets look at the first step of eval c:

- If it is a ᵗtau step, we directly conclude by guardedness (ᵗev-guard).
- If it is a return ᵗret $((i \blacksquare o), \gamma)$ where i belongs to the final scope Ω, the composition returns and we conclude again by guardedness (ᵗev-guard).
- In the last case, where eval c returns as above, with i a non-final variable, we arrive at the core of the proof and continue as follows.

Recapitulating, we can now forget about c, we still have σ^+ and σ^- and we now have $i : \downarrow^+\Psi \ni \alpha$, $o : O.\mathrm{op}\ \alpha$ and $\gamma : O.\mathrm{holes}\ o \multimap (\Omega +\!\!+ \downarrow^+\Psi)$. Our goal is now to prove

$$\mathrm{ev\text{-}guarded}_{\text{m-compo-eqn}}\ (\mathrm{ret}\ (\mathrm{inr}\ ((\mathrm{apply}\ (\downarrow^-\sigma^-\ i)[\rho_1]\ o\ \rho_2),\ \sigma^-) \blacksquare (\sigma^+ \blacktriangleright^- \gamma)))$$

with the usual culprits $\rho_1 := [\text{r-cat}_l, \text{r-cat}_r[\text{r-cat}_l]]$ and $\rho_2 := \text{r-cat}_r[\text{r-cat}_r]$.

Proceed first by well-founded induction on (α, o) (using the finite redex hypothesis FiniteRedexes), and then by well-founded induction on the depth of i. This does not change the proof goal but simply introduces two induction hypotheses.

As obviously the current computation is not guarded, first apply ᵗev-step to unfold one more step of the equation, obtaining the following goal

$$\mathrm{ev\text{-}guarded}_{\text{m-compo-eqn}}\ (\text{m-compo-eqn}\ ((\mathrm{apply}\ (\downarrow^-\sigma^-\ i)[\rho_1]\ o\ \rho_2),\ \sigma^-) \blacksquare (\sigma^+ \blacktriangleright^- \gamma)).$$

Then, by case on whether or not $v := \downarrow^-\sigma^-\ i$ is a variable (is-var?).

- If v is some variable $j : (\Omega +\!\!+ \downarrow^-\Psi) \ni \alpha$, the apply expression is in fact the embedding of the normal form $((\rho_1\ j \blacksquare o), \rho_2)$. Thus, by eval-nf we know its evaluation. Then by case on view-cat j:
  - If $j : \Omega \ni \alpha$, the composition ends with $(j \blacksquare o)$, thus is guarded.
  - If $j : \downarrow^-\Psi \ni \alpha$, then by Lemma 5.12, $\mathrm{depth}^-\ j < \mathrm{depth}^+\ i$. Taking the next context increment into account by moving from Ψ to $\Psi \blacktriangleright O.\mathrm{holes}\ o$, we deduce $\mathrm{depth}^+\ (\text{r-cat}_l\ j) < \mathrm{depth}^-\ i$ and conclude by the innermost induction hypothesis.
- If v is not a variable, pose $c := \mathrm{apply}\ v[\rho_1]\ o\ \rho_2$ and inspect the first step of eval c.
  - If it is a ᵗtau move, then the composition is guarded.

- If it is a ˜ret $((k \blacksquare o'), \delta)$, then, by case on view-cat k. If k is a final move then composition is guarded. Else k is non-final and we conclude by applying the outermost induction hypothesis, as indeed $o' \prec o$. ■

5.5 Conclusion

Now that the core properties are proven (Theorem 5.10 and Theorem 5.13), what is left to do is mostly to combine them in the right way to deduce adequacy. Still, to finish up and deduce correctness we have left out one benign step described in the proof outline: congruence of weak bisimilarity for composition (Definition 5.1). Let us prove it now.

Lemma 5.14 (Congruence for Composition):
For all Ψ : Ctx S, $s_1^+, s_2^+ : \text{Ogs}_O^+\ \Omega\ \Psi$ and $s^- : \text{Ogs}_O^-\ \Omega\ \Psi$, the following property holds.

$$s_1^+ \approx s_2^+ \to (s_1^+ \parallel s^-) \approx (s_2^+ \parallel s^-)$$

Proof: The proof is very similar to Lemma 5.4 and is mostly an application of monotonicity of iteration (Lemma 2.72). Define the following relation on $\text{Arg} := \text{Ogs}_O^+\ \Omega \bowtie \text{Ogs}_O^-\ \Omega$

$$R : \text{Arg} \to \text{Arg} \to \text{Prop}$$
$$R\ (s_1^+ \blacksquare s_1^-)\ (s_2^+ \blacksquare s_2^-) := (s_1^+ \approx s_2^+) \times (\forall\ r \to s_1^-\ r \approx s_2^-\ r).$$

It is easy to show that R is preserved by the composition equation, then, by application of Lemma 2.72 the result follows. ■

Theorem 5.15 (Adequacy of Composition):
For all $\Gamma : S, c : C\ \Gamma$ and $\gamma : \Gamma \multimap [V] \mapsto \Omega$, the following property holds.

$$\text{eval-to-obs}\ c[\gamma] \approx (\llbracket c \rrbracket_M^+ \parallel \llbracket \gamma \rrbracket_M^-)$$

Proof: Embedding c and γ respectively to initial active and initial passive states of the machine strategy, by definition we have

$$\text{eval-to-obs}\ c[\gamma] \cong \text{zip-then-eval}\ ((c[\text{r-cat}_r], (\varepsilon^- \blacktriangleright^+ \circlearrowright)) \blacksquare (\varepsilon^+ \blacktriangleright^- \gamma[\text{r-cat}_l]))$$

By Lemma 2.80 and Theorem 5.13, the machine strategy composition equation has unicity of strong fixed points, hence by Theorem 5.10, zip-then-eval is pointwise strongly bisimilar to its *eventually guarded iteration*. Continuing the above chain we obtain the following.

$$\text{eval-to-obs}\ c[\gamma] \cong \text{ev-iter}_{\text{m-compo-eqn}}\ ((c[\text{r-cat}_r], (\varepsilon^- \blacktriangleright^+ \circlearrowright)) \blacksquare (\varepsilon^+ \blacktriangleright^- \gamma[\text{r-cat}_l]))$$

By Lemma 2.85, the eventually guarded iteration is pointwise weakly bisimilar the unguarded iteration, i.e., m-compo. We obtain the following.

$$\text{eval-to-obs}\ c[\gamma] \approx \text{m-compo}\ ((c[\text{r-cat}_r], (\varepsilon^- \blacktriangleright^+ \circlearrowright)) \blacksquare (\varepsilon^+ \blacktriangleright^- \gamma[\text{r-cat}_l]))$$

Finally we conclude by using Lemma 5.4 to bridge between machine strategy compositions and opaque composition.

$$\text{eval-to-obs}\ c[\gamma] \approx [\![c]\!]^+_M \parallel [\![\gamma]\!]^-_M$$ ■

We finally prove OGS correctness w.r.t. contextual equivalence (Theorem 4.33).

Proof: Given $c_1, c_2 : C\ \Gamma$ such that

$$[\![c_1]\!]^+_M \approx [\![c_2]\!]^+_M,$$

for any $\gamma : \Gamma \multimap[V]\to \Omega$ we have

$$\begin{aligned} \text{eval-to-obs}\ c_1[\gamma] &\approx ([\![c_1]\!]^+_M \parallel [\![\gamma]\!]^-_M) && \text{(Theorem 5.15)} \\ &\approx ([\![c_2]\!]^+_M \parallel [\![\gamma]\!]^-_M) && \text{(Lemma 5.14)} \\ &\approx \text{eval-to-obs}\ c_2[\gamma] && \text{(Theorem 5.15)} \end{aligned}$$ ■

To conclude this chapter, and with it the proof of the central result of this thesis, let me say a couple words about its significance. First of all, OGS models which are sound with respect to observational equivalence have already been published, some in fact for instances which we do not cover here, such as, say, impure languages with references [54][48]. As such, although there is some progress in providing a generic proof for all languages which are expressible as language machines, I believe that the core contribution is in the way we streamline the correctness proof, hopefully making it accessible to a broader community. There are two novelties which contribute to this.

[54] James Laird, "A Fully Abstract Trace Semantics for General References," 2007.
[48] Guilhem Jaber and Andrzej S. Murawski, "Compositional relational reasoning via operational game semantics," 2021.

First, by working with an abstract *language machine*, we can more precisely see and isolate which parts of the proofs are generic and which parts are language specific. This has led us to formalize precise requirements, as well as recognize previously hidden details, such as the finite redex hypothesis.

Second, while the decomposition into an adequacy and congruence proof for composition is common, in published articles, adequacy is typically proven monolithically. The method provided here further decomposes adequacy into two independent arguments, each quite informative on its own.

Theorem 5.10 provides what we can think of as the core semantical argument for adequacy: one step of composition does not change the result obtained by syntactically substituting the two machine strategy states and evaluating the obtained

language configuration. Satisfyingly, the proof is quite direct and relatively free of administrative headaches: it is simply a sequence of rewriting step.

Knowing this, one should not be blamed for thinking it is enough to conclude adequacy. After all, if one composition step leaves the final result unaltered, why should it be any different after an infinite number of steps, i.e., full composition? But arbitrary fixed points of partial computations can behaved surprisingly. Thus, a second argument, Theorem 5.13, concentrates the technical justification of the informal reasoning: the composition equation is "nice" enough and thus enjoys unicity of fixed points.

I hope that this separation between the high-level argument and the tedious technical justification demystifies the adequacy proof, and enables a better understanding of it by non specialists. In particular, it opens up an intermediate level of explanation of the OGS correctness in which Theorem 5.10 is detailed but where unicity is taken for granted, leaving Theorem 5.13 to the specialist.

Normal Form Bisimulations

6

Operational game semantics is intimately linked to another slightly older family of coinductive constructions: *normal form bisimulations*. Building upon Sangiorgi's *open bisimulations* [85], the term was coined by Lassen [56][57] as an umbrella for coinductive operational characterizations of the equivalences induced by several related constructions such as Böhm trees, Levy-Longo trees, Lassen trees and others tailored to a variety of languages. They distinguish themselves from the other big family of *applicative* bisimulations [4] mainly by using *fresh variables* instead of a *closed values* as arguments to observations.

[85] Davide Sangiorgi, "A Theory of Bisimulation for the pi-Calculus," 1993.
[56] Søren B. Lassen, "Eager Normal Form Bisimulation," 2005.
[57] Søren B. Lassen, "Normal Form Simulation for McCarthy's Amb," 2005.
[4] Samson Abramsky, "The lazy lambda calculus," 1990.

In this short chapter, we start by introducing our own variant of normal form bisimulation, for any given language machine (§6.1). Then, in §6.2 we show how the interpretation from language configurations to Ogs strategies can be factored through *normal form strategies*. Thanks to this, we deduce a correctness theorem, stating—as for Ogs—that any two *normal form bisimilar* language configurations are *substitution equivalent*.

Remark 6.1: Be advised that at the time of writing these lines, the constructions and proofs contained in this chapter are only sketched in our Rocq artifact. As we will see the proofs are not particularly challenging, but this part came in quite late during the thesis. I invite you to check the online repository to see if this has been fixed by the time you are reading.

https://github.com/lapin0t/ogs

6.1 Normal Form Bisimulations in a Nutshell

Implicitly or explicitly, the main idea in all normal form (Nf) bisimulation constructions, is to associate to each program a possibly infinite tree intuitively representing its infinitely expanded normal form. Here, we will call these trees *normal form strategies*, that is, strategies for the *normal form game*. This induces a notion of program equivalence which holds whenever two programs have bisimilar associated strategies: *normal form bisimilarity*.

These infinite trees are constructed by reducing the program to a normal form for some given reduction strategy—most usually some kind of head-reduction. The "head" of the normal form gives us the node of the tree, while the children are obtained by coinductively applying the same process to all subterms of the normal form. By now this process of splitting a normal form into a head and a sequence of subterms should ring a bell... Although Ogs and Nf bisimulations have historically been introduced in reverse order, we can use our readily available

knowledge of the Ogs game to provide a very succinct and precise description of the normal form game. The Nf game is nothing but a restriction of the Ogs game, where the server is only allowed to query the variables introduced by the *last* client move.

Reusing our existing infrastructure of binding families and named observations, we express the Nf game quite similarly to the naive Ogs game, slightly changing the game positions and transition functions. Since the allowed server moves are dictated by the *last* client move, only the client scope needs to be threaded throughout the game positions. As such, client positions consist of a single scope Γ, containing the variables the client is allowed to observe, while server positions consist of two scopes (Γ, Δ), containing the variables that respectively the server and the client are allowed to observe.

Definition 6.2 (Nf Game):
Assuming a scope structure $\mathrm{Scope}_T\ S$, given a binding family $O : \mathrm{Bind}_T\ S$, the *normal form (Nf) game* is defined as follows.

$$\mathrm{Nf}^{\mathrm{G}}_O : \mathrm{Game}\ S\ (S \times S) :=$$
$$\begin{bmatrix} \mathrm{client} := \\ \begin{bmatrix} \mathrm{Move}\ \Gamma := O^{\aleph}\ \Gamma \\ \mathrm{next}\ \{\Gamma\}\ o := (\mathrm{holes}^{\aleph}\ o, \Gamma) \end{bmatrix} \\ \mathrm{server} := \\ \begin{bmatrix} \mathrm{Move}\ (\Gamma, \Delta) := O^{\aleph}\ \Gamma \\ \mathrm{next}\ \{\Gamma, \Delta\}\ o := \Delta +\!\!+ \mathrm{holes}^{\aleph}\ o \end{bmatrix} \end{bmatrix}$$

We then define active and passive normal form strategies with respect to a final scope Ω, as for Ogs strategies.

Definition 6.3 (Nf Strategies):
Assuming a scope structure $\mathrm{Scope}_T\ S$, given a binding family $O : \mathrm{Bind}_T\ S$ and a scope $\Omega : S$, *active and passive normal form strategies over O with final scope Ω* are defined as follows.

$$\mathrm{Nf}^{+}_O\ \Omega\ \Gamma := \mathrm{Strat}^{+}{}_{\mathrm{Nf}^{\mathrm{G}}_O}\ (\lambda\ \Gamma \mapsto O^{\aleph}\ \Omega)\ \Gamma$$
$$\mathrm{Nf}^{-}_O\ \Omega\ \Gamma\ \Delta := \mathrm{Strat}^{-}{}_{\mathrm{Nf}^{\mathrm{G}}_O}\ (\lambda\ \Gamma \mapsto O^{\aleph}\ \Omega)\ (\Gamma, \Delta)$$

Remark 6.4: Note that $\mathrm{Nf}^{-}_O\ \Omega$ is isomorphic to an assignment type. Indeed, define "unary" passive strategies as follows.

$$\mathrm{Nf}^{1}_O\ \Omega : \mathrm{Type}^{S,T}$$
$$\mathrm{Nf}^{1}_O\ \Omega\ \Gamma\ \alpha := (o : O.\mathrm{Op}\ \alpha) \to \mathrm{Nf}^{+}_O\ \Omega\ (\Gamma +\!\!+ O.\mathrm{holes}\ o)$$

Then, $\text{NF}^{-}_{O}\ \Omega\ \Gamma\ \Delta$ is isomorphic to $\Gamma \multimap[\ \text{NF}^{1}_{O}\ \Omega\ ]\mapsto \Delta$, as witnessed by the following two conversion functions, definitionally inverse to each other.

$$\text{into}\ s^{-}\ i\ o \quad := s^{-}\ (i \blacksquare o)$$
$$\text{from}\ \sigma\ (i \blacksquare o) := \sigma\ i\ o$$

In light of this, we will extend standard assignment notations to passive strategies, in particular the initial arrow and the copairing.

$$[] : \text{NF}^{-}_{O}\ \Omega\ \varnothing\ \Delta \qquad [k_1, k_2] : \text{NF}^{-}_{O}\ \Omega\ (\Gamma_1 + \Gamma_2)\ \Delta$$
$$[] := \text{from}\ [] \qquad [k_1, k_2] := \text{from}\ [\text{into}\ k_1, \text{into}\ k_2]$$

Given a language machine with renamings, we now construct the strategy associated to any given language configuration. Once again, it is merely a simplified version of the (naive) OGS machine strategy: it proceeds by evaluating the current language configuration to compute the next move, and using the application map to respond to queries.

Definition 6.5 (NF Strategy):
Given a language machine M : LangMachine $O\ V\ C$ with renamings, i.e., such that PointedRenModule V and RenModule C hold, given a final scope $\Omega : S$, the *NF machine strategy* is the big-step system defined as follows.

$$\text{NF-mstrat}\ M : \text{Big-Step-System}_{\text{NF}^{\mathbb{G}}_{O}}\ (\lambda\ \Gamma \mapsto O^{\mathbb{N}}\ \Omega)$$

$$\text{NF-mstrat}\ M :=$$
$$\begin{bmatrix}
\text{state}^{+}\ \Gamma := C\ (\Omega + \Gamma) \\
\text{state}^{-}\ (\Gamma, \Delta) := \Gamma \multimap[V]\mapsto (\Omega + \Delta) \\
\text{play}\ c := \text{do} \\
\quad ((i \blacksquare o), \gamma) \leftarrow \text{eval}\ c\ ; \\
\quad \text{case}\ (\text{view-cat}\ i) \\
\quad \begin{bmatrix} \text{v-left}\ i & := \text{ret}\ (\text{inl}\ (i \blacksquare o)) \\ \text{v-right}\ j & := \text{ret}\ (\text{inr}\ ((j \blacksquare o), \gamma)) \end{bmatrix} \\
\text{coplay}\ \gamma\ (i \blacksquare o) := \text{apply}\ (\gamma\ i)[\rho_1]\ o\ \rho_2
\end{bmatrix}$$

$$\rho_1 := [\text{r-cat}_l, \text{r-cat}_r[\text{r-cat}_l]]$$
$$\rho_2 := \text{r-cat}_r[\text{r-cat}_r]$$

Definition 6.6 (NF Interpretation, NF Bisimulation):
Given a language machine M : LangMachine $O\ V\ C$ with renamings, i.e., such that PointedRenModule V and RenModule C hold, given a final scope $\Omega : S$, the *NF interpretation* is obtained by unrolling the NF machine strategy.

```
⟦␣⟧ᴺᶠ_M {Γ} : C Γ → Nf⁺_O Ω Γ
⟦c⟧ᴺᶠ_M := unroll⁺ Nf-mstrat M c[r-cat_r]
```

Furthermore, two configurations $c_1, c_2 : C\ \Gamma$ are *normal form bisimilar* whenever $[\![c_1]\!]^{\text{Nf}}_M \approx [\![c_2]\!]^{\text{Nf}}_M$.

6.2 Nf Correctness through Ogs

To deduce correctness of Nf bisimulation from our Ogs correctness theorem, we need to relate Nf strategies and Ogs strategies. First, since the Nf game is very close to the Ogs game, simply allowing less server moves, it is easy to restrict an Ogs strategy to an Nf strategy.

Definition 6.7 (Ogs to Nf):
Assuming a scope structure $\text{Scope}_T\ S$, given a binding family $O : \text{Bind}_T\ S$ and a scope $\Omega : S$, define the *restriction from Ogs to Nf strategies* by coinduction as follows.

```
Ogs-to-Nf⁺ {Ψ} : Ogs⁺_O Ω Ψ → Nf⁺_O Ω (↓⁺Ψ)
Ogs-to-Nf⁺ s :=
  out := case s.out
    'ret r   := 'ret r
    'tau t   := 'tau (Ogs-to-Nf⁺ t)
    'vis q k := 'vis q (λ (i ▪ o) ↦ Ogs-to-Nf⁺ (k (r-cat_r i ▪ o)))
```

Yet the most interesting direction is the other one: embedding Nf strategies into Ogs strategies. In the Ogs game, the server is also allowed to query older variables, which, on the face of it, an Nf strategy does not know how to respond to. However, every variable was once new! So if we remember all the continuations of an Nf strategy along the way, we can accumulate enough information to respond to any Ogs server queries, by restarting the relevant old continuation. In order to do so, we will first need a small helper to rename the scope of Nf strategies.

Definition 6.8 (Nf Strategy Renaming):
Assuming a scope structure $\text{Scope}_T\ S$, given a binding family $O : \text{Bind}_T\ S$ and a scope $\Omega : S$, define the *active and passive Nf strategy renamings* by mutual coinduction as follows.

```
Nf-ren⁺ {Ω} : Nf⁺_O Ω → ⟦(∋), Nf⁺_O Ω⟧
Nf-ren⁺ s ρ :=
  [ out := case s.out
    [ 'ret r       := 'ret r
    | 'tau t       := 'tau (Nf-ren⁺ t ρ)
    | 'vis (i ▪ o) k := 'vis (ρ i ▪ o) (Nf-ren⁻ k ρ)

Nf-ren⁻ {Ω Γ} : Nf⁻_O Ω Γ → ⟦(∋), Nf⁻_O Ω Γ⟧
Nf-ren⁻ k ρ m := Nf-ren⁺ (k m) [ρ · r-cat_l, r-cat_r]
```

This is in fact the definition of the action of two renaming structures, on $\mathrm{Nf}^+_O\ \Omega$ and $\mathrm{Nf}^-_O\ \Omega\ \Gamma$.

Remark 6.9: Our definition of the renaming action makes use of the internal substitution hom (Definition 3.20), which we did not see in a long time! The types can be spelled out more explicitly as follows.

```
Nf-ren⁺ {Ω Δ₁ α} (s : Nf⁺_O Ω Δ₁ α) {Δ₂} : Δ₁ ⊆ Δ₂ → Nf⁺_O Ω Δ₂ α
Nf-ren⁻ {Ω Γ Δ₁} (s : Nf⁻_O Ω Γ Δ₁) {Δ₂} : Δ₁ ⊆ Δ₂ → Nf⁻_O Ω Γ Δ₂
```

Definition 6.10 (Nf to Ogs):
Assuming a scope structure $\mathrm{Scope}_T\ S$, given a binding family $O : \mathrm{Bind}_T\ S$ and a scope $\Omega : S$, define as follows the *active and passive embedding from Nf strategies to Ogs strategies*.

```
Nf-to-Ogs⁺ {Ω Ψ} : Nf⁺_O Ω (↓⁺Ψ) → Nf⁻_O Ω (↓⁻Ψ) (↓⁺Ψ) → Ogs⁺_O Ω Ψ
Nf-to-Ogs⁺ s ks :=
  [ out := case s.out
    [ 'ret r     := 'ret r
    | 'tau t     := 'tau (Nf-to-Ogs⁺ t ks)
    | 'vis m k := 'vis m (Nf-to-Ogs⁻ [ks, k])

Nf-to-Ogs⁻ {Ω Ψ} : Nf⁻_O Ω (↓⁺Ψ) (↓⁻Ψ) → Ogs⁻_O Ω Ψ
Nf-to-Ogs⁻ ks m := Nf-to-Ogs⁺ (ks m) (Nf-ren⁻ ks r-cat_r)
```

Finally, define the following shorthand, embedding Nf strategies to Ogs strategies over an initial position.

```
Nf-to-Ogs {Ω Γ} : Nf⁺_O Ω Γ → Ogs⁺_O Ω (ε ▸ Γ)
Nf-to-Ogs s := Nf-to-Ogs⁺ s []
```

We can now show that the Ogs interpretation can be factorized through the Nf interpretation. However, because the coinductive call of Nf-to-Ogs⁻ happens on a renamed strategy, renamings will creep up during the bisimulation proof. For this reason we first need to prove an up-to-renaming reasoning principle for Nf

strategies, essentially stating that any NF bisimulation candidate is closed under renamings.

Lemma 6.11 (NF Bisimulation Up-to Renaming):
Assuming a scope structure $\mathrm{Scope}_T\ S$, a binding family $O : \mathrm{Bind}_T\ S$ and a scope $\Omega : S$, then $\text{NF-ren}^+\ \{\Omega\}$ respects any strong or weak bisimulation candidate, i.e., for any

$$\mathcal{F} \in \mathrm{Tower}_{\mathrm{sbisim}_{\text{NF}^{\mathrm{G}}_O}} \qquad \text{or} \qquad \mathcal{F} \in \mathrm{Tower}_{\mathrm{wbisim}_{\text{NF}^{\mathrm{G}}_O}},$$

the following property holds.

$$\text{NF-ren}^+\ \langle\!\langle \forall^r \mathcal{F}\ (=) \twoheadrightarrow^r [\![(=), \mathcal{F}\ (=)]\!]^r \rangle\!\rangle\ \text{NF-ren}^+$$

Proof: Both statements (weak and strong) are proven by direct tower induction. ■

Remark 6.12: Recalling Theorem 2.36, the above lemma implies in particular that NF-ren^+ respects both weak and strong bisimilarity.

Theorem 6.13 (OGS Through NF Factorization):
Given a language machine $M : \mathrm{LangMachine}\ O\ V\ C$ with renamings and a final scope $\Omega : S$, the OGS interpretation factorizes through the NF interpretation. More precisely, for any $c : C\ \Gamma$, the following property holds.

$$[\![c]\!]^+_M \cong^+ \text{NF-to-OGS}\ ([\![c]\!]^{\text{NF}}_M)$$

Proof: The only trick required to prove this is to generalize the statement to arbitrary OGS game positions and machine strategy states. We prove that for any $\Psi : \mathrm{Ctx}\ S$, $c' : C\ (\Omega +\!\!+ \downarrow^+ \Psi)$ and $e : \mathrm{Tele}^+_\Omega\ \Psi$ the following property holds.

$$\begin{aligned} &\mathrm{unroll}^+{}_{\text{mstrat}\ M}\ (c' \blacksquare e) \\ \approx^+ &\ \text{NF-to-OGS}^+\ (\mathrm{unroll}^+{}_{\text{NF-mstrat}\ M}\ c')\ (\mathrm{unroll}^-{}_{\text{NF-mstrat}\ M}\ (\downarrow^+ e)) \end{aligned}$$

This statement is then proven by direct tower induction, unfolding the definitions and using Lemma 6.11 where required. The theorem follows by direct application, taking $\Psi := \varepsilon \blacktriangleright \Gamma$, $c' := c[\text{r-cat}_r]$ and $e := []$. ■

In order to finally prove NF bisimulation correctness, we still need to show a technicality, namely that NF-to-OGS respects weak bisimilarity.

Lemma 6.14 (NF-to-OGS Respects Weak Bisimilarity):
Assuming a scope structure $\mathrm{Scope}_T\ S$, a binding family $O : \mathrm{Bind}_T\ S$, NF-to-OGS respects weak bisimilarity, i.e., the following property holds.

$$\text{NF-to-OGS}\ \langle\!\langle \forall^r \approx \twoheadrightarrow^r \approx \rangle\!\rangle\ \text{NF-to-OGS}$$

Proof: We generalize the statement and show that NF-to-OGS$^+$ respects weak bisimilarity:

$$\text{NF-to-OGS}^+ \ \langle\!\langle \forall^r \approx \twoheadrightarrow^r \approx^- \twoheadrightarrow^r \approx \rangle\!\rangle \ \text{NF-to-OGS}^+$$

with $\approx^-$ denoting pointwise weak bisimilarity of passive strategies. This statement is proven by straightforward tower induction, using Lemma 6.11 where required. ■

We can now prove the normal form bisimulation correctness theorem.

Theorem 6.15 (NF Correctness):
Given a language machine M : LangMachine $O\ V\ C$ with renamings, i.e., such that PointedRenModule V and RenModule C hold, given a final scope $\Omega : S$, NF bisimulation is correct w.r.t. OGS bisimulation, i.e., for any $\Gamma : S$ and $c_1, c_2 : C\ \Gamma$, the following statement holds.

$$[\![c_1]\!]_M^{\text{NF}} \approx [\![c_2]\!]_M^{\text{NF}} \to [\![c_1]\!]_M^{+} \approx [\![c_2]\!]_M^{+}$$

Proof: Assume c_1 and c_2 such that $[\![c_1]\!]_M^{\text{NF}} \approx [\![c_2]\!]_M^{\text{NF}}$.

$$\begin{aligned}
[\![c_1]\!]_M^{+} &\cong \text{NF-to-OGS}\ ([\![c_1]\!]_M^{\text{NF}}) && \text{(Theorem 6.13)} \\
&\approx \text{NF-to-OGS}\ ([\![c_2]\!]_M^{\text{NF}}) && \text{(Lemma 6.14)} \\
&\cong [\![c_2]\!]_M^{+} && \text{(Theorem 6.13)}
\end{aligned}$$

■

The above correctness theorem concludes the treatment of normal form bisimulations for this thesis. By combining it with Theorem 4.33 proving OGS correctness w.r.t. substitution equivalence, we can directly deduce that NF bisimulations are correct w.r.t. substitution equivalence. Since the server is allowed to play less moves in the NF game than in the OGS game, it is naturally easier to prove that two language configurations are normal form bisimilar than OGS bisimilar. As such, to prove substitution equivalence of two concrete programs, the NF correctness theorem is a more practical entry point than the OGS correctness theorem. In fact in the realm of program equivalence for languages without state or polymorphism, it can be argued that OGS is merely a technical device for proving NF correctness. And indeed, in a line of work by LASSEN and LEVY [58] [59], an early appearance of an OGS-like construction can be seen during the NF correctness proof.

There is definitely a number of side results on NF strategies which we have glossed over. Indeed, much of the above constructions and proofs are still to be written in the ROCQ artifact, and it is quite uncomfortable to program without the safety net of an actual type checker! Among the presumably low hanging fruits, proving the injectivity of NF-to-OGS would give us the reverse of the above implication, in other words that the NF model is correct and complete w.r.t. the OGS model.

[58] Søren B. Lassen and Paul Blain Levy, "Typed Normal Form Bisimulation," 2007.
[59] Søren B. Lassen and Paul Blain Levy, "Typed Normal Form Bisimulation for Parametric Polymorphism," 2008.

There is also much more to say on the relationship between the NF game and the OGS game, but we leave this thorough study for future work.

7 OGS Instances

In Ch. 4 we have seen a language-generic OGS construction, parametrized by an abstract notion of language machine. We have even seen a shiny theorem, correction for this model w.r.t. substitution equivalence, our variant of observational equivalence. Hopefully some intuitions from the introduction helped understand these abstract constructions, but it is now time to look at some concrete examples. In this chapter we try to show a small but representative collection of calculi that are covered by our abstract theorem. We start with two calculi neatly fitting our language machine presentation, finally showing the driving intuitions behind our axiomatization. To warm up we start with perhaps the simplest one: Jump-with-Argument (§7.1). We then follow up with a much more featureful language, polarized $\mu\tilde{\mu}$-calculus (§7.2). Then, in §7.3, we look at a language which for several reasons does not look like the prototypical language machine, but still can be twisted (rather heavily) to fit our axiomatization: pure untyped λ-calculus under weak head reduction.

Remark 7.1: Be advised that at the time of writing these lines, the set of example calculi presented in this chapter is not exactly the same as the one present in our ROCQ artifact. Their respective features are broadly the same, and our present choice is guided by making this collection more comprehensive. I invite you to check the online repository to see if this has been fixed by the time you are reading.

https://github.com/lapin0t/ogs

7.1 Jump-with-Argument

7.1.1 Syntax

Jump-with-Argument (JWA) was introduced by LEVY [60] as a target for his *stack passing style* transform of Call-By-Push-Value (CBPV). As such, it is a minimalistic language with first-class continuations centered around so-called *non-returning commands*: an ideal target for our first language machine. We direct the interested reader to two existing constructions of NF bisimulations and OGS-like model for increasingly featureful variants of JWA by LEVY and LASSEN [58][59]. This is a typed language, so as is usual, there is some leeway regarding which types are included. As we are aiming for simplicity, we will only look at a representative fragment featuring three types of values: booleans $\mathbb{B}$, continuations $\neg A$ and pairs $A \times B$. This language is strongly normalizing, although it is not directly obvious from the evalutator and usually proven by a reducibility argument. As

[60] Paul Blain Levy, *Call-By-Push-Value: A Functional/Imperative Synthesis*, 2004.

[58] Søren B. Lassen and Paul Blain Levy, "Typed Normal Form Bisimulation," 2007.

[59] Søren B. Lassen and Paul Blain Levy, "Typed Normal Form Bisimulation for Parametric Polymorphism," 2008.

such, although we know that the evaluator is semantically quite simple, we will still benefit from our framework allowing for non-termination, by simply side-stepping any proof of totality of said evaluator.

The language consists of two judgments: non-returning commands and values. Commands play the role of the configurations of an abstract machine, and are only parametrized by a scope. They are of three kinds: sending a value to a continuation (the eponymous "jump with argument"), splitting a value of product type into its two components and case splitting on a boolean. Values are parametrized by a scope and a type and can be either a variable, a continuation abstraction which binds its argument and continues as a command, a pair of values or a boolean. Let us present its syntax by an intrinsically typed and scoped representation inside type theory.

Definition 7.2 (Syntax):
Jwa types are given by the following data type.

```
data typ : Type

  𝔹 : typ
  ¬_ : typ → typ
  _×_ : typ → typ → typ
```

Define the two syntactic judgments by the following two mutually inductive data types.

```
data _⊢c : Ctx typ → Type
data _⊢v _ : Ctx typ → typ → Type
```

We will also use the shorthands cmd := _ ⊢c and val := _ ⊢v _ when it is more practical. The constructors are as follows.

$$\frac{x : \Gamma \ni A}{\mathrm{var}\ x : \Gamma \vdash^{v} A} \qquad \frac{}{\mathrm{true} : \Gamma \vdash^{v} \mathbb{B}} \qquad \frac{}{\mathrm{false} : \Gamma \vdash^{v} \mathbb{B}}$$

$$\frac{c : \Gamma \blacktriangleright A \vdash^{c}}{\gamma\, c : \Gamma \vdash^{v} \neg A} \qquad \frac{v : \Gamma \vdash^{v} A \quad w : \Gamma \vdash^{v} B}{\langle v, w \rangle : \Gamma \vdash^{v} A \times B}$$

$$\frac{v : \Gamma \vdash^{v} \mathbb{B} \quad c_1 : \Gamma \vdash^{c} \quad c_2 : \Gamma \vdash^{c}}{\mathrm{case}\ v\ \mathrm{in}\ [c_1, c_2] : \Gamma \vdash^{c}} \qquad \frac{v : \Gamma \vdash^{v} A \quad k : \Gamma \vdash^{v} \neg A}{v \nearrow k : \Gamma \vdash^{c}}$$

$$\frac{v : \Gamma \vdash^{v} A \times B \quad c : \Gamma \blacktriangleright A \blacktriangleright B \vdash^{c}}{\mathrm{split}\ v\ \mathrm{in}\ c : \Gamma \vdash^{c}}$$

Note that we do not include the standard let v in c command, as it can be simply derived as $v \nearrow \gamma\, c$.

Remark 7.3: Among the surprising elements is probably the continuation abstraction $\gamma\, c$. Intuitively it can be understood as a λ-abstraction, but which

never returns. As such, its body is not a term as in λ-calculus but a non-returning command. The "jump-with-argument" $v \nearrow k$ can then be seen as the counter-part to function application, written in reverse order.

Lemma 7.4 (Substitution):
JWA values form a substitution monoid, and JWA commands form a substitution module over it. Moreover, val has decidable variables. In other words, the following typeclasses are inhabited: SubstMonoid val, SubstModule val cmd and DecidableVar val.

Proof: Although quite tedious, these constructions are entirely standard [37] [13]. One essentially starts by mutually defining the renaming operation on commands and values, unlocking the definition of the weakening operation necessary for substitution. Then, on top of this, one mutually defines the substitution operation, giving the monoid and module structures. The proofs of the laws as similarly layered. ■

[37] Marcelo Fiore and Dmitrij Szamozvancev, "Formal metatheory of second-order abstract syntax," 2022.
[13] Guillaume Allais, Robert Atkey, James Chapman, Conor McBride, and James McKinna, "A type- and scope-safe universe of syntaxes with binding: their semantics and proofs," 2021.

7.1.2 Patterns and Negative Types

The next logical step is the definition of the evaluation. Informally, the reduction rules are defined on commands as follows.

$$
\begin{aligned}
v \nearrow \gamma\, c &\rightsquigarrow c[\mathsf{var}, v] \\
\text{split } \langle v,w \rangle \text{ in } c &\rightsquigarrow c[\mathsf{var}, v, w] \\
\text{case true in } [c_1,c_2] &\rightsquigarrow c_1 \\
\text{case false in } [c_1,c_2] &\rightsquigarrow c_2
\end{aligned}
$$

Note the usage of the assignments $[\mathsf{var}, v]$ and $[\mathsf{var}, v, w]$ for substituting only the top variables, leaving the rest unchanged (with the identity assignment var).

However, recall that in a language machine, the evaluator should return a normal form configuration, which is to be expressed using a binding family of *observations*. As the definition of observations is central and essentially decides the shape that the OGS model will take, let us take a small step back.

Recall from the introduction (§1.3) that in OGS models, depending on their type, some values are "given" to the opponent as part of a move, while some other are "hidden" behind fresh variables. We did not, however, worry about this distinction at all during our generic development, as we simply assumed there was a set of types T and worked on top of that. This discrepancy is simply explained: what the generic construction considers as *types* should not be instantiated to all of our language's types, but only the ones that are interacted with, i.e., the "hidden" types.

Most eloquently described in the case of CBPV [60], this split occurs between types that have dynamic behavior, the *computation types* and the ones that have

[60] Paul Blain Levy, *Call-By-Push-Value: A Functional/Imperative Synthesis*, 2004.

static content, the *value types*. For concision we will call them respectively *negative* and *positive* types. Although the concrete assignment of types to either category can be somewhat varied, obtaining models with different properties, for Jwa it is most natural to decide that continuations are negative types while booleans and pairs are positive.

Definition 7.5 (Negative Types):
The definition of Jwa negative types is based on a strict predicate picking out the continuation types.

$$\begin{aligned}
&\text{is-neg} : \text{typ} \to \text{SProp} \\
&\text{is-neg}\ \mathbb{B} &&:= \bot && \text{typ}^{\text{neg}} := \textstyle\int_{\text{typ}} \text{is-neg} \\
&\text{is-neg}\ \neg A &&:= \top && \text{ctx}^{\text{neg}} := \textstyle\int_{\text{Ctx typ}} (\text{All is-neg}) \\
&\text{is-neg}\ (A \times B) &&:= \bot
\end{aligned}$$

Remark 7.6: Note that to manipulate the above contexts and types, we will make great use of the *subset scope* structure introduced in Definition 3.10.

Further, as is-neg is trivially decidable, we will allow ourselves a bit of informality by using $\neg A$ as an element of typ^{neg}, instead of the more correct $(\neg A, \star)$. Moreover, we will do as if elements of typ^{neg} could be pattern-matched on, with the sole pattern being $\neg A$, although technically this has to be justified with a *view* [71][12], adding a small amount of syntactic noise.

[71] Conor McBride and James McKinna, "The view from the left," 2004.
[12] Guillaume Allais, "Builtin Types Viewed as Inductive Families," 2023.

Lemma 7.7 (Restricted Values and Commands):
The family of values and commands restricted to negative scopes and types defined as

$$(\lambda\ \Gamma\ \alpha \mapsto \text{val}\ \Gamma.\text{fst}\ \alpha.\text{fst}) : \text{Type}^{\text{ctx}^{\text{neg}}, \text{typ}^{\text{neg}}}$$
$$(\lambda\ \Gamma \mapsto \text{cmd}\ \Gamma.\text{fst}) : \text{Type}^{\text{ctx}^{\text{neg}}}$$

again form respectively a substitution monoid and a substitution module, with respect to the subset scope structure (Definition 3.10).

We will overload the notations val and cmd for both the unrestricted and restricted families. Similarly we will stop writing the projection fst and implicitly use it as a coercion from negative types and scopes to normal ones.

Proof: By η-expansion of the records and functions witnessing the unrestricted structures. ■

For these negative types, i.e., continuations, we need to devise a notion of observation which will make up the content of the Ogs moves. The only sensible thing to do with a continuation $\neg A$ is to send it something of type A. As our goal is to hide continuations from moves, this something should be void of any

continuation abstraction. This can be recognized as the set of *ultimate patterns* of type A, i.e., maximally deep patterns of type A.

Definition 7.8 (Ultimate Patterns and Observations):
Define the inductive family of ultimate patterns data pat : typ → Type with the following constructors.

$$\frac{}{\mathrm{true}^p : \mathrm{pat}\ \mathbb{B}} \qquad \frac{}{\mathrm{false}^p : \mathrm{pat}\ \mathbb{B}} \qquad \frac{}{\blacksquare_A : \mathrm{pat}\ \neg A} \qquad \frac{p : \mathrm{pat}\ A \quad q : \mathrm{pat}\ B}{\langle p,q\rangle^p : \mathrm{pat}\ (A \times B)}$$

Define their *domain* by the following inductive function.

$$\begin{aligned}
&\mathrm{dom}\ \{A\} : \mathrm{pat}\ A \to \mathrm{Ctx}\ \mathrm{typ}^{\mathrm{neg}} \\
&\mathrm{dom}\ \blacksquare_A \quad := \varepsilon \blacktriangleright \neg A \\
&\mathrm{dom}\ \mathrm{true}^p \ := \varepsilon \\
&\mathrm{dom}\ \mathrm{false}^p \ := \varepsilon \\
&\mathrm{dom}\ \langle p,q\rangle^p := \mathrm{dom}\ p \mathbin{+\!\!+} \mathrm{dom}\ q
\end{aligned}$$

Finally define *observations* as the following binding family.

$$\begin{aligned}
&\mathrm{obs} : \mathrm{Bind}\ \mathrm{ctx}^{\mathrm{neg}}\ \mathrm{typ}^{\mathrm{neg}} \\
&\mathrm{obs} := \begin{bmatrix} \mathrm{Op}\ \neg A := \mathrm{pat}\ A \\ \mathrm{holes}\ p \ := \mathrm{dom}\ p \end{bmatrix}
\end{aligned}$$

Remark 7.9: The continuation pattern $\blacksquare_A$ should be understood intuitively as the pattern consisting of one fresh variable. This is comforted by the fact that its domain is indeed a singleton scope. More generally, the domain of a pattern collects the set of continuations inside a value, when seeing patterns as a subset of values.

The first thing to do with ultimate patterns is to show how to embed them into values.

Definition 7.10 (Ultimate Pattern Embedding):
Ultimate patterns can be embedded into values, as witnessed by the following inductive function.

$$\begin{aligned}
&\mathrm{p\text{-}emb}\ \{A\}\ (p : \mathrm{pat}\ A) : \mathrm{dom}\ p \vdash^{\mathrm{v}} A \\
&\mathrm{p\text{-}emb}\ \blacksquare_A \quad := \mathrm{var}\ \mathrm{top} \\
&\mathrm{p\text{-}emb}\ \mathrm{true}^p \ := \mathrm{true} \\
&\mathrm{p\text{-}emb}\ \mathrm{false}^p \ := \mathrm{true} \\
&\mathrm{p\text{-}emb}\ \langle p,q\rangle^p := \langle (\mathrm{p\text{-}emb}\ p)[\mathrm{r\text{-}cat}_l], (\mathrm{p\text{-}emb}\ q)[\mathrm{r\text{-}cat}_r]\rangle
\end{aligned}$$

The next step is to do the reverse: given a value whose scope is entirely negative (as will happen during the OGS game, since only negative variables are exchanged), split it into an ultimate pattern and a filling assignment.

Remark 7.11: The fact that we only do this in negative scopes is important as we will be able to refute the case of, e.g. a variable of type $\mathbb{B}$. Indeed, assuming a scope structure $\mathrm{Scope}_T\ S$ and a predicate $P : \mathrm{Prop}^T$, define the following function, "upgrading" a type into a proof that it verifies P, provided we know that it is a member of a P-subscope.

$$\mathrm{elt\text{-}upgr}\ \{\Gamma : \textstyle\int_S\ (\mathrm{All}\ P)\}\ \{x\} : \Gamma.\mathrm{fst} \ni x \to P\ x$$
$$\mathrm{elt\text{-}upgr}\ i := \Gamma.\mathrm{snd}\ i$$

Definition 7.12 (Value Splitting):
Define the following functions, splitting values in negative context into a pattern and an assignment over its domain.

$$\mathrm{split\text{-}pat}\ \{\Gamma : \mathrm{ctx}^{\mathrm{neg}}\}\ (A : \mathrm{typ}) : \Gamma \vdash^{\mathrm{v}} A \to \mathrm{pat}\ A$$
$$\mathrm{split\text{-}pat}\ \mathbb{B}\quad (\mathrm{var}\ i) := \mathrm{ex\text{-}falso}\ (\mathrm{elt\text{-}upgr}\ i)$$
$$\mathrm{split\text{-}pat}\ \mathbb{B}\quad \mathrm{true} := \mathrm{true}^p$$
$$\mathrm{split\text{-}pat}\ \mathbb{B}\quad \mathrm{false} := \mathrm{false}^p$$
$$\mathrm{split\text{-}pat}\ \neg A\quad v := \blacksquare_A$$
$$\mathrm{split\text{-}pat}\ (A \times B)\ (\mathrm{var}\ i) := \mathrm{ex\text{-}falso}\ (\mathrm{elt\text{-}upgr}\ i)$$
$$\mathrm{split\text{-}pat}\ (A \times B)\ \langle v,w\rangle := \langle \mathrm{split\text{-}pat}\ A\ v, \mathrm{split\text{-}pat}\ B\ w\rangle^p$$

$$\mathrm{split\text{-}fill}\ \{\Gamma : \mathrm{ctx}^{\mathrm{neg}}\}\ A\ (v : \Gamma \vdash^{\mathrm{v}} A) : \mathrm{dom}\ (\mathrm{split\text{-}pat}\ v) \multimap[\ \mathrm{val}\ ]\mapsto \Gamma$$
$$\mathrm{split\text{-}fill}\ \mathbb{B}\quad (\mathrm{var}\ i) := \mathrm{ex\text{-}falso}\ (\mathrm{elt\text{-}upgr}\ i)$$
$$\mathrm{split\text{-}fill}\ \mathbb{B}\quad \mathrm{true} := []$$
$$\mathrm{split\text{-}fill}\ \mathbb{B}\quad \mathrm{false} := []$$
$$\mathrm{split\text{-}fill}\ \neg A\quad v := [\ v\ ]$$
$$\mathrm{split\text{-}fill}\ (A \times B)\ (\mathrm{var}\ i) := \mathrm{ex\text{-}falso}\ (\mathrm{elt\text{-}upgr}\ i)$$
$$\mathrm{split\text{-}fill}\ (A \times B)\ \langle v,w\rangle := [\ \mathrm{split\text{-}fill}\ A\ v, \mathrm{split\text{-}fill}\ B\ w\ ]$$

Before moving on towards evaluation, we can prove our first interesting lemma, namely that splitting a value and refolding it yields the same value unchanged and that this splitting is unique.

Lemma 7.13 (Refolding):
The following statements holds.

(1) For all $\Gamma : \mathrm{ctx}^{\mathrm{neg}}$, $A : \mathrm{typ}$ and $v : \Gamma \vdash^{\mathrm{v}} A$,

$(\text{p-emb}\ (\text{split-pat}\ v))[\text{split-fill}\ v] = v.$

(2) For all $\Gamma : \text{ctx}^{\text{neg}}$, $A : \text{typ}$, $p : \text{pat}\ A$ and $\gamma : \text{dom}\ p \multimap[\ \text{val}\ ]\mapsto \Gamma$,

$(p, \gamma) \approx (\text{split-pat}\ (\text{p-emb}\ p)[\gamma], \text{split-fill}\ (\text{p-emb}\ p)[\gamma]).$

Note the second equation lives in $(p : \text{pat}\ A) \times (\text{dom}\ p \multimap[\ \text{val}\ ]\mapsto \Gamma)$, with extensional equality $\approx$ here meaning equality on the first projections and pointwise equality on the second.

Proof: Both by direct induction, following the same case pattern as split-pat and split-fill. ■

7.1.3 The Jwa Language Machine

Lets recapitulate where we stand in the instantiation. We have defined the negative types typ^{neg} and scopes ctx^{neg}, as well as the matching observation family $\text{obs} : \text{Bind}\ \text{ctx}^{\text{neg}}\ \text{typ}^{\text{neg}}$. We have defined values val and commands cmd over general types and then restricted them to negative types and scopes. To instantiate a language machine, this leaves us to define the evaluation and application maps, which have the following types.

$$\text{eval}\ \{\Gamma : \text{ctx}^{\text{neg}}\} : \text{cmd}\ \Gamma \to \text{Delay}\ \left(\text{Nf}^{\text{obs}}_{\text{val}}\ \Gamma\right)$$

$$\text{apply}\ \{\Gamma : \text{ctx}^{\text{neg}}\}\ \{A : \text{typ}^{\text{neg}}\}\ (v : \text{val}\ \Gamma\ A)\ (o : \text{obs}\ .\text{Op}\ A)$$
$$: \text{obs}\ .\text{holes}\ o \multimap[\ \text{val}\ ]\mapsto \Gamma \to \text{cmd}\ \Gamma$$

Let us start with the evaluation map. Although it is not really necessary, for clarity we will implement an *evaluation step* function, taking a command to either a normal form or to a new command to be evaluated further. The evaluation map is then simply obtained by unguarded iteration in the Delay monad.

Definition 7.14 (Evaluation):
Define evaluation by iteration of an evaluation step as follows.

$$\text{eval}\ \{\Gamma : \text{ctx}^{\text{neg}}\} : \text{cmd}\ \Gamma \to \text{Delay}\ \left(\text{Nf}^{\text{obs}}_{\text{val}}\ \Gamma\right)$$
$$\text{eval} := \text{iter}\ (\text{ret} \circ \text{eval-step})$$

$$\text{eval-step}\ \{\Gamma : \text{ctx}^{\text{neg}}\} : \text{cmd}\ \Gamma \to \text{Nf}^{\text{obs}}_{\text{val}}\ \Gamma + \text{cmd}\ \Gamma$$

```
eval-step (case (var i) in [c₁,c₂]) := ex-falso (elt-upgr i)
eval-step (case true in [c₁,c₂])    := inr c₁
eval-step (case false in [c₁,c₂])   := inr c₂
eval-step (v ↗ var i)               := inl ((i ▪ split-pat v), split-fill v)
eval-step (v ↗ γ c)                 := inr c[var, v]
eval-step (split (var i) in c)      := ex-falso (elt-upgr i)
eval-step (split ⟨v,w⟩ in c)        := inr c[var, v, w]
```

The next and final step is to define the application map, which is easily done. The target type

```
apply {Γ : ctx^neg} {A : typ^neg} (v : val Γ A) (o : obs .Op A)
  : obs .holes o ⊣[ val ]↦ Γ → cmd Γ
```

is perhaps slightly scary, but this is largely due to the fact that A is quantified over negative types, instead of explicitly asking that it is a continuation type. This is an artifact of the language machine axiomatization and in this case it would be better written with the following isomorphic representation.

```
apply {Γ : ctx^neg} {A : typ} (v : val Γ ¬A) (o : pat A)
  : dom o ⊣[ val ]↦ Γ → cmd Γ
```

What needs to be done is then more clear: embed the pattern, substitute it by the given assignment, and send the whole thing to the continutation value v.

Definition 7.15 (Observation Application):
Define observation application as follows.

```
apply {Γ : ctx^neg} {A : typ^neg} (v : val Γ A) (o : obs .Op A)
  : obs .holes o ⊣[ val ]↦ Γ → cmd Γ
apply {Γ} {¬A} v o γ := (p-emb o)[γ] ↗ v
```

We now have everything to instantiate the Jwa language machine.

Definition 7.16 (Language Machine):
The Jwa language machine is given by the following record.

```
Jwa : LangMachine obs val cmd

Jwa := [ eval       := eval
       | apply      := apply
       | eval-ext   := ...
       | apply-ext  := ... ]
```

Note that apply-ext is proven by direct application of sub-ext from the substitution monoid structure of values. eval-ext is obtained by simple rewriting, as extensional equality of commands is simply propositional equality.

By the above definition we can already instantiate the Ogs model, but to obtain correctness w.r.t. substitution equivalence, we need to verify the hypotheses of Theorem 4.33. We are left with the two interesting hypotheses: the core semantic argument, the Jwa machine respects substitution (Definition 4.28), and the technical side condition, that it has finite redexes (Definition 4.31).

Lemma 7.17 (Jwa Respects Substitution):
The Jwa language machine respects substitution, i.e., the following typeclass is inhabited: LangMachineSub Jwa.

Proof:
eval-sub Given $\Gamma, \Delta : \mathrm{ctx}^{\mathrm{neg}}$, $c : \mathrm{cmd}\ \Gamma$ and $\sigma : \Gamma \dashv[\ \mathrm{val}\ ]\mapsto \Delta$, we need to prove the following.

$$\mathrm{eval}\ c[\sigma] \cong \mathrm{eval}\ c \gg\!= \lambda\, n \mapsto \mathrm{eval}\ (\mathrm{nf\text{-}emb}\ n)[\sigma]$$

Proceed by tower induction and then by cases on c, following the elimination pattern of eval-step.

- case (var i) in $[c_1,c_2]$ Impossible by ex-falso (elt-upgr i).
- case true in $[c_1,c_2]$ Unfold one coinductive layer of the RHS and rewrite as follows.

$$\begin{aligned} & (\mathrm{eval}\ (\mathrm{case\ true\ in}\ [c_1,c_2])[\sigma]).\mathrm{out} \\ = \ & (\mathrm{eval}\ (\mathrm{case\ true\ in}\ [c_1[\sigma],c_2[\sigma]])).\mathrm{out} && \text{by def.} \\ = \ & \text{"tau}\ (\mathrm{eval}\ c_1[\sigma]) && \text{by def.} \end{aligned}$$

Similarly, after unfolding, the RHS reduces to

$$\text{"tau}\ (\mathrm{eval}\ c_1 \gg\!= \lambda\, n \mapsto \mathrm{eval}\ (\mathrm{nf\text{-}emb}\ n)[\sigma]).$$

The two "tau provide a synchronization and we conclude by coinduction hypothesis on c_1.

- case false in $[c_1,c_2]$ Same as previous case, with c_2.
- $v \nearrow \mathrm{var}\ i$ The LHS reduces to $\mathrm{eval}\ (v[\sigma] \nearrow \sigma\ i)$. In the RHS, the first evaluation unfolds and reduces to "ret $((i \blacksquare \mathrm{split\text{-}pat}\ v), \mathrm{split\text{-}fill}\ v)$, so that the RHS as a whole can be rewritten to

$$\mathrm{eval}\ (\mathrm{nf\text{-}emb}\ ((i \blacksquare \mathrm{split\text{-}pat}\ v), \mathrm{split\text{-}fill}\ v))[\sigma]$$

We rewrite and conclude as follows.

$$
\begin{aligned}
& (\mathsf{nf\text{-}emb}\ ((i \blacksquare \mathsf{split\text{-}pat}\ v), \mathsf{split\text{-}fill}\ v))[\sigma] \\
= \ & (\mathsf{apply}\ (\mathsf{var}\ i)\ (\mathsf{split\text{-}pat}\ v)\ (\mathsf{split\text{-}fill}\ v))[\sigma] && \text{by def.} \\
= \ & ((\mathsf{p\text{-}emb}\ (\mathsf{split\text{-}pat}\ v))[\mathsf{split\text{-}fill}\ v] \nearrow (\mathsf{var}\ i))[\sigma] && \text{by def.} \\
= \ & (v \nearrow (\mathsf{var}\ i))[\sigma] && \text{by Lemma 7.13} \\
= \ & v[\sigma] \nearrow (\sigma\ i) && \text{by def.}
\end{aligned}
$$

- $v \nearrow \gamma\ c'$ Unfold the LHS and rewrite as follows.

$$
\begin{aligned}
& (\mathsf{eval}\ (v \nearrow \gamma\ c')[\sigma]).\mathsf{out} \\
= \ & (\mathsf{eval}\ (v[\sigma] \nearrow \gamma\ c'[\sigma[\mathsf{pop}], \mathsf{top}])).\mathsf{out} && \text{by def.} \\
= \ & \text{"}\mathsf{tau}\ (\mathsf{eval}\ c'[\sigma[\mathsf{pop}], \mathsf{top}][\mathsf{var}, v[\sigma]]) && \text{by def.} \\
= \ & \text{"}\mathsf{tau}\ (\mathsf{eval}\ c'[\mathsf{var}, v][\sigma]) && \text{by sub. fusion}
\end{aligned}
$$

  Unfolding the RHS reduces to

$$\text{"}\mathsf{tau}\ (\mathsf{eval}\ c'[\mathsf{var}, v] \gg\!= \lambda\ n \mapsto \mathsf{eval}\ (\mathsf{nf\text{-}emb}\ n)[\sigma]).$$

  The two "tau provide a synchronization and we conclude by coinduction hypothesis on $c'[\mathsf{var}, v]$

- $\mathsf{split}\ (\mathsf{var}\ i)\ \mathsf{in}\ c'$ Impossible by $\mathsf{ex\text{-}falso}\ (\mathsf{elt\text{-}upgr}\ i)$.
- $\mathsf{split}\ \langle v,w \rangle\ \mathsf{in}\ c'$ Unfold the LHS and rewrite as follows.

$$
\begin{aligned}
& (\mathsf{eval}\ (\mathsf{split}\ \langle v,w \rangle\ \mathsf{in}\ c')[\sigma]).\mathsf{out} \\
= \ & (\mathsf{eval}\ (\mathsf{split}\ \langle v[\sigma],w[\sigma] \rangle\ \mathsf{in}\ c'[\sigma[\mathsf{pop} \circ \mathsf{pop}, \mathsf{pop}\ \mathsf{top}, \mathsf{top}]])).\mathsf{out} && \text{by def.} \\
= \ & \text{"}\mathsf{tau}\ (\mathsf{eval}\ c'[\sigma[\mathsf{pop} \circ \mathsf{pop}], \mathsf{pop}\ \mathsf{top}, \mathsf{top}][\mathsf{var}, v[\sigma], w[\sigma]]) && \text{by def.} \\
= \ & \text{"}\mathsf{tau}\ (\mathsf{eval}\ c'[\mathsf{var}, v, w][\sigma]) && \text{by sub. fusion}
\end{aligned}
$$

  Unfolding the RHS reduces to

$$\text{"}\mathsf{tau}\ (\mathsf{eval}\ c'[\mathsf{var}, v, w][\sigma] \gg\!= \lambda\ n \mapsto \mathsf{eval}\ (\mathsf{nf\text{-}emb}\ n)[\sigma]).$$

  The two "tau provide a synchronization and we conclude by coinduction hypothesis on $c'[\mathsf{var}, v, w]$

This concludes eval-sub. Although slightly tedious, it is not hard to prove. As we will see in other instances, the pattern is always the same: analyzing the configuration to make the evaluation reduce, upon hitting a redex, shift substitutions around and conclude by coinduction. Upon hitting a normal form, apply a refolding lemma and conclude by reflexivity.

Thankfully the other two properties are almost trivial. apply-sub is a direct application of substitution fusion, as follows.

$$
\begin{array}{lll}
 & \mathsf{apply}\ v[\sigma]\ o\ \gamma[\sigma] & \\
= & (\mathsf{p\text{-}emb}\ o)[\gamma[\sigma]] \nearrow v[\sigma] & \text{by def.} \\
= & ((\mathsf{p\text{-}emb}\ o)[\gamma] \nearrow v)[\sigma] & \text{by sub. fusion} \\
= & (\mathsf{apply}\ v\ o\ \gamma)[\sigma] & \text{by def.}
\end{array}
$$

eval-nf is proven by unicity of splitting after one-step unfolding, as follows.

$$
\begin{array}{lll}
 & (\mathsf{eval}\ (\mathsf{nf\text{-}emb}\ ((i \blacksquare o), \gamma))).\mathsf{out} & \\
= & (\mathsf{eval}\ (\mathsf{apply}\ (\mathsf{var}\ i)\ o\ \gamma)).\mathsf{out} & \text{by def.} \\
= & (\mathsf{eval}\ ((\mathsf{p\text{-}emb}\ o)[\gamma] \nearrow \mathsf{var}\ i)).\mathsf{out} & \text{by def.} \\
= & \text{"ret}\ ((i \blacksquare \mathsf{split\text{-}pat}\ (\mathsf{p\text{-}emb}\ o)[\gamma]), \mathsf{split\text{-}fill}\ (\mathsf{p\text{-}emb}\ o)[\gamma] & \text{by def.} \\
= & \text{"ret}\ ((i \blacksquare o), \gamma) & \text{by Lemma 7.13}
\end{array}
$$

■

Lemma 7.18 (Jwa Finite Redexes):
The Jwa language machine has finite redexes.

Proof: As explained for Definition 7.15, we silently do a benign preprocessing to express the property using patterns instead of observations, for else the notational weight would be unbearable. We need to prove that the relation defined by the following inference rule is well founded.

$$
\frac{\begin{array}{c} i : \Gamma \ni \alpha_1 \quad o_1 : \mathsf{pat}\ \alpha_1 \quad \gamma_1 : \mathsf{dom}\ o_1 \dashv[\ \mathsf{val}\ ]\mapsto \Gamma \quad v : \mathsf{val}\ \Gamma\ \neg\alpha_2 \\ o_2 : \mathsf{pat}\ \alpha_2 \quad \gamma_2 : \mathsf{dom}\ o_2 \dashv[V]\mapsto \Gamma \\ H_1 : \mathsf{is\text{-}var}\ v \to \bot \quad H_2 : \mathsf{eval}\ (\mathsf{apply}\ v\ o_2\ \gamma_2) \cong \mathsf{ret}\ ((i \blacksquare o_1), \gamma_1) \end{array}}{\mathsf{bad}\ H_1\ H_2 : o_1 \prec o_2}
$$

What we will prove is in fact that this relation is empty, directly implying that it is wellfounded.

Assume $\alpha_1, \alpha_2 : \mathsf{typ}^{\mathsf{neg}}$, $o_1 : \mathsf{pat}\ \alpha_1$ and $o_2 : \mathsf{pat}\ \alpha_2$ such that $o_1 \prec o_2$. Destruct the proof of $o_1 \prec o_2$, introduce all the hypotheses as above and proceed by case on v, as it is a continuation value, it can be either an abstraction or a variable. If $v := \gamma\ c$, $\mathsf{apply}\ v\ o_2\ \gamma$ reduces to $(\mathsf{p\text{-}emb}\ o_2)[\gamma] \nearrow \gamma\ c$. This is a redex, thus its evaluation start with "tau and thus H_2 is absurd as the RHS starts with "ret. If instead $v := \mathsf{var}\ i$, then H_1 is absurd. ■

Remark 7.19: The above proof of finite redex is quite remarkable: as the relation is empty no "redex failure" can happen, i.e., evaluating the application of a non-variable value *always* creates a redex. "Low-level" languages such as Jwa which are designed around first-class continuations usually have this stronger property.

This concludes the proof of the hypotheses for correctness. We may now deduce Ogs correctness for Jwa. We will do it with respect to the standard final scope $\Omega := \varepsilon \blacktriangleright \neg\mathbb{B}$ containing exactly one boolean continuation. To match common practice, we also adapt the definition of substitution equivalence to use a notion of big-step reduction $c \Downarrow b$.

Definition 7.20 (Evaluation Relation):
For $c : \mathrm{cmd}\ (\varepsilon \blacktriangleright \neg\mathbb{B})$ and $b : \mathrm{pat}\ \mathbb{B}$, define the following big step *evaluation relation*.

$$c \Downarrow b := (\mathrm{fst} \langle\$\rangle\ \mathrm{eval}\ c) \approx \mathrm{ret}\ (\mathrm{top} \blacksquare b)$$

Theorem 7.21 (Ogs Correctness):
For all $\Gamma : \mathrm{ctx}^{\mathrm{neg}}$ and $c_1, c_2 : \mathrm{cmd}\ \Gamma$, if $[\![c_1]\!]^+_M \approx [\![c_2]\!]^+_M$, then for all $\gamma : \Gamma \dashv[\ \mathrm{val}\ ]\vdash (\varepsilon \blacktriangleright \neg\mathbb{B})$,

$$c_1[\gamma] \Downarrow \mathrm{true}^p \leftrightarrow c_2[\gamma] \Downarrow \mathrm{true}^p\ .$$

Proof: By Theorem 4.33 applied to Jwa, with hypotheses proven in Lemma 7.4, Lemma 7.17 and Lemma 7.18, obtain

$$\mathrm{fst} \langle\$\rangle\ \mathrm{eval}\ c_1[\gamma] \quad \approx \quad \mathrm{fst} \langle\$\rangle\ \mathrm{eval}\ c_2[\gamma].$$

Conclude by the fact that $\approx$ is an equivalence relation*. ■

* For the beauty of the game, let us say that a tiny generalization of the last proof step, shifting an equivalence relation around a bi-implication, is a very useful fact of any symmetric transitive relation R, namely that $R \langle\!\langle R \to^r R \to^r (\leftrightarrow) \rangle\!\rangle R$.

By Theorem 6.15, we can then deduce Nf model correctness w.r.t. substitution equivalence.

Note that we obtain correctness only for commands in negative scopes. This is easily dealt with, by defining an extended equivalence relation on commands in arbitrary scopes $\Gamma : \mathrm{Ctx}\ \mathrm{typ}$, which first quantifies over an ultimate pattern for each type in Γ and asserts Ogs equivalence of the configurations substituted by the given sequence of patterns. First, let us sketch the pointwise lifting from types to scopes of several constructs related to patterns.

Definition 7.22 (Pointwise Pattern Lifting):
Define the lifting of pat and dom to scopes as follows.

$$\mathrm{pat}^* : \mathrm{Ctx}\ \mathrm{typ} \to \mathrm{Type}$$
$$\mathrm{pat}^*\ \Gamma := \{A : \mathrm{typ}\} \to \Gamma \ni A \to \mathrm{pat}\ A$$

$$\mathrm{dom}^*\ (\Gamma : \mathrm{Ctx}\ \mathrm{typ}) : \mathrm{pat}^*\ \Gamma \to \mathrm{ctx}^{\mathrm{neg}}$$
$$\mathrm{dom}^*\ \varepsilon \qquad \gamma := \varepsilon$$
$$\mathrm{dom}^*\ (\Gamma \blacktriangleright A)\ \gamma := \mathrm{dom}^*\ \Gamma\ (\gamma \circ \mathrm{pop}) \blacktriangleright\!\blacktriangleright \mathrm{dom}\ (\gamma\ \mathrm{top})$$

Further define an embedding of pattern assignments into ordinary assignments as follows.

```
p-emb* {Γ} (γ : pat* Γ) : Γ ─[ val ]↦ dom* γ
p-emb* γ i := (p-emb (γ i))[aux γ i]
```

We only give the type of the required family of renamings aux as it is not a joy to write down.

```
aux {Γ} (γ : pat* Γ) {A} (i : Γ ∋ A) : dom (γ i) ⊆ dom* γ
```

Using these tools we can define our Ogs equivalence relation, extended to general scopes.

Definition 7.23 (Extended Ogs Equivalence):
For all Γ : Ctx typ and c_1, c_2 : cmd Γ define the *extended Jwa Ogs equivalence* as follows.

$$c_1 \underset{\text{ext}}{\approx} c_2 := (\gamma : \text{pat}^* \ \Gamma) \to [\![c_1[\text{p-emb}^* \ \gamma]]\!]^+_M \approx [\![c_2[\text{p-emb}^* \ \gamma]]\!]^+_M$$

And finally we recover correctness.

Theorem 7.24 (Extended Ogs Correctness):
For all Γ : Ctx typ and c_1, c_2 : cmd Γ if $c_1 \underset{\text{ext}}{\approx} c_2$, then for all $\gamma : \Gamma \ ─[\text{val}]↦ (\varepsilon \blacktriangleright \neg \mathbb{B})$,

$$c_1[\gamma] \Downarrow \text{true}^p \leftrightarrow c_2[\gamma] \Downarrow \text{true}^p \ .$$

Proof: By pointwise lifting of split-pat and split-fill applyied to γ, compute α : pat* Γ and β : dom* α $─[\text{val}]↦ (\varepsilon \blacktriangleright \neg \mathbb{B})$. By $c_1 \underset{\text{ext}}{\approx} c_2$, we have

$$[\![c_1[\text{p-emb}^* \ \alpha]]\!]^+_M \approx [\![c_2[\text{p-emb}^* \ \alpha]]\!]^+_M.$$

From Theorem 7.21, deduce

$$c_1[\text{p-emb}^* \ \alpha][\beta] \Downarrow \text{true}^p \leftrightarrow c_2[\text{p-emb}^* \ \alpha][\beta] \Downarrow \text{true}^p \ .$$

Finally by pointwise lifting of Lemma 7.13 obtain $(\text{p-emb}^* \ \alpha)[\beta] \approx \gamma$, which concludes. ■

Remark 7.25: Of course the extended correctness results would hold just as well when considering Nf bisimularity instead of the Ogs strategy bisimilarity, by following exactly the same step, replacing the Ogs interpretation by the Nf strategy interpretation.

This concludes the instantiation of our generic framework for Jwa. Arguably, although defining the actual data takes some getting used to, the proof mostly

amounts to busywork. The only meaningful lemma to be designed and proven is the refolding lemma.

7.2 Polarized μμ̃-calculus

This next instance serves as a demonstration of the limits of what is captured by our axiomatization of language machines. Technically, the structure will not be much different than for Jwa, simply scaled up. As such, we will give a bit less details.

The μμ̃-calculus was introduced by Curien and Herbelin [30] as a term assignment for the classical sequent calculus. Its core idea is to scrupulously follow a discipline of duality. The configurations of its evaluator can be understood as a formal cut, i.e., a pair $\langle t \parallel e \rangle$ of a term t producing something of type A and a *coterm* (i.e., a context) e, consuming something of type A. These producers and consumers are truly on an equal footing and we consolidate both into a single judgment of *generalized terms* indexed by "side annotated types" with ${}^{+}A$ and ${}^{-}A$ respectively denoting the type of A-terms and A-coterms. Likewise, what is usually called "variable" (term name) and "covariable" (context name) are consolidated into a single construction, such that the typing scopes of all judgments also consist of side annotated types.

[30] Pierre-Louis Curien and Hugo Herbelin, "The duality of computation," 2000.

μ and $\tilde{\mu}$ are the prime constructions respectively for terms and coterms, giving their name to the calculus. The first can be understood as a form of call-with-current-continuation, while the second is similar to a let-binding. More precisely, the term $\mu\,\alpha.c$ captures the current coterm it is facing, binding it to the fresh covariable α and continuing the execution as the configuration c. On the other hand, the coterm $\tilde{\mu}\,x.c$ captures the current term, binding it to the fresh variable x and continuing the execution as c. In this form, the calculus in non-confluent as witnessed by the following critical pair

$$c_1[\alpha \mapsto \tilde{\mu}\,x.c_2] \quad \leftsquigarrow \quad \langle \mu\,\alpha.c_1 \parallel \tilde{\mu}\,x.c_2 \rangle \quad \rightsquigarrow \quad c_2[x \mapsto \mu\,\alpha.c_1],$$

depending on which of μ or $\tilde{\mu}$ reduces first. A simple way to overcome this non-determinism is to bias the calculus to either call-by-value, prioritizing μ or call-by-name, prioritizing $\tilde{\mu}$. We adopt the other standard solution, arguably more general, which is to parametrize configurations by a mode, or *polarity*, recording whether they are currently in call-by-value mode (v) or call-by-name mode (n). This *polarized* μμ̃-calculus thus has the ability to express both execution strategies. In effect, each type is assigned a polarity, and the polarity of a configuration is determined by the type on which it is cutting. The type system is entirely symmetric with respect to polarity, so that every type former has a dual of opposite

polarity. Do not confuse the Cbv-Cbn duality of type polarities with the producer-consumer duality of terms and coterms as the two are entirely orthogonal! The distinction between producer and consumers is the one between programs and continuations, while the distinction between Cbv and Cbn is between strict and lazy programs (and between lazy and strict continuations). Because of the profusion of dualities we have deliberately avoided the "positive" and "negative" nomenclature of polarities.

The concrete way by which the priority of the μ or $\tilde{\mu}$ rule is managed is by restricting both of their reduction rules to only apply when the other side of the configuration is a *(co)value*. Now pay attention because the different dualities mingle!

- A **value of Cbv type** is a new syntactic category included in terms, the *weak head normal terms*, consisting of variables and of *constructors* of that type.
- A **covalue of Cbv type** is simply a coterm of that type.
- A **value of Cbn type** is simply a term of that type.
- A **covalue of Cbn type** is a new syntactic category included in coterms, *weak head normal coterms*, consisting of covariables and of *destructors* of that type.

We hope that all the symmetries are enjoyable. The consequence is that at a Cbv type, the μ reduction rule will fire across any coterm, while the $\tilde{\mu}$ rule will only fire across a weak-head normal term (of which μ is not). The opposite happens at Cbn types.

Technically, our polarized presentation approximately follows the one from Downen and Ariola [33], obtaining a middle ground between their System L and System D. For a more general introduction to unpolarized μμ̃-calculus, we can recommend the tutorial by Binder *et. al* [20].

[33] Paul Downen and Zena M. Ariola, "Compiling With Classical Connectives," 2020.

[20] David Binder, Marco Tzschentke, Marius Müller, and Klaus Ostermann, "Grokking the Sequent Calculus," 2024.

Without further ado, let us jump to the formal definition of types and syntax.

Definition 7.26 (Types):
There are two kinds, or polarities, given as follows.

```
data pol : Type := v | n
```

The syntax of *open types* is given by the inductive family

```
data typ^o (Θ : Ctx pol) : pol → Type
```

whose constructors are given below.

$$\frac{}{0 : \mathrm{typ}^o\ \Theta\ \mathsf{v}}$$

$$\frac{}{\top : \mathrm{typ}^o\ \Theta\ \mathsf{n}} \quad \frac{}{1 : \mathrm{typ}^o\ \Theta\ \mathsf{v}} \quad \frac{}{\bot : \mathrm{typ}^o\ \Theta\ \mathsf{n}} \quad \frac{A : \mathrm{typ}^o\ \Theta\ \mathsf{v} \quad B : \mathrm{typ}^o\ \Theta\ \mathsf{v}}{A \otimes B : \mathrm{typ}^o\ \Theta\ \mathsf{v}}$$

$$\frac{A : \mathrm{typ}^o\ \Theta\ \mathsf{n} \quad B : \mathrm{typ}^o\ \Theta\ \mathsf{n}}{A \parr B : \mathrm{typ}^o\ \Theta\ \mathsf{n}} \quad \frac{A : \mathrm{typ}^o\ \Theta\ \mathsf{v} \quad B : \mathrm{typ}^o\ \Theta\ \mathsf{v}}{A \oplus B : \mathrm{typ}^o\ \Theta\ \mathsf{v}}$$

$$\frac{A : \mathrm{typ}^o\ \Theta\ \mathsf{n} \quad B : \mathrm{typ}^o\ \Theta\ \mathsf{n}}{A \,\&\, B : \mathrm{typ}^o\ \Theta\ \mathsf{n}} \quad \frac{A : \mathrm{typ}^o\ \Theta\ \mathsf{n}}{{\downarrow}A : \mathrm{typ}^o\ \Theta\ \mathsf{v}}$$

$$\frac{A : \mathrm{typ}^o\ \Theta\ \mathsf{v}}{{\uparrow}A : \mathrm{typ}^o\ \Theta\ \mathsf{n}} \quad \frac{A : \mathrm{typ}^o\ \Theta\ \mathsf{n}}{\ominus A : \mathrm{typ}^o\ \Theta\ \mathsf{v}} \quad \frac{A : \mathrm{typ}^o\ \Theta\ \mathsf{v}}{\neg A : \mathrm{typ}^o\ \Theta\ \mathsf{n}} \quad \frac{A : \mathrm{typ}^o\ (\Theta \blacktriangleright \mathsf{v})\ \mathsf{v}}{\mu A : \mathrm{typ}^o\ \Theta\ \mathsf{v}}$$

$$\frac{A : \mathrm{typ}^o\ (\Theta \blacktriangleright \mathsf{n})\ \mathsf{n}}{\nu A : \mathrm{typ}^o\ \Theta\ \mathsf{n}} \quad \frac{i : \Theta \ni p}{\mathrm{tvar}\ i : \mathrm{typ}^o\ \Theta\ p}$$

Define (closed) *types* by the shorthand $\mathrm{typ} := \mathrm{typ}^o\ \varepsilon$.

Lemma 7.27 (Type Substitution):
Open types typ^o form a substitution monoid. We will write A/B for the topmost variable substitution $A[\mathrm{var}, B]$.

The types of our language thus comprise the usual bunch of empty, singleton, product and sum types, each in their Cbv (0, 1, $\otimes$, $\oplus$) and Cbn ($\bot$, $\top$, $\&$, $\parr$) variants. We then have the two shifts, $\downarrow$ for *thunks* of a Cbn type and $\uparrow$ for *returners* of a Cbv type, and two negations ($\ominus$, $\neg$), for continuations of the two polarities. Finally, we do not consider existential and universal quantification, but replace them by two fixed point types, μ for inductive-like types and ν for coinductive-like types.

Remark 7.28: The above type variables and type substitution might raise some questions. In the context of a formal development, it is well-known that formalizing polymorphic calculi is quite involved, in particular in this well-typed-and-scoped style [24]. Moreover, our generic Ogs construction only supports simple types. Here though, we only need to mention open types in passing, to express recursive types. The rest of the formalization will be solely concerned with closed types. Indeed, recursive types, in contrast to existential and universal types, do maintain a constant type-variable scope throughout the term syntax. In particular, proving Ogs correctness will *not* require *any* law on substitution or renaming of types.

Remark 7.29: The inclusion of the recursive μ and ν types is mainly motivated by making the language non-terminating, as indeed they should allow us to write a fixed point combinator. A classical case study is then to show that any

[24] James Chapman, Roman Kireev, Chad Nester, and Philip Wadler, "System F in Agda, for Fun and Profit," 2019.

two fixed point combinators are normal form bisimilar, hence substitution equivalent, as e.g. done by Lassen and Levy [58] for a Jwa with recursive types. However, because both normal form bisimulations and the precise language machine instance presented here, have not yet been entirely mechanized in the Rocq artifact, we will not do this case study on fixed point combinators. We nonetheless present the language with recursive types here, to show off how their expressivity still fits into our language machine axiomatization.

[58] Søren B. Lassen and Paul Blain Levy, "Typed Normal Form Bisimulation," 2007.

Definition 7.30 (Side-Annotated Types):
Define *side-annotated types* data typ^s : Type by the following constructors.

$$\frac{A : \mathrm{typ}\ p}{{}^{+}A : \mathrm{typ}^s} \qquad \frac{A : \mathrm{typ}\ p}{{}^{-}A : \mathrm{typ}^s}$$

Note that we will use $^{+}$ and $^{-}$ with very weak parsing precedence, allowing us to write, e.g., ${}^{+}A \parr B$.

Define side-annotated type dualization $A^{\dagger}$ as follows.

$$({}^{+}A)^{\dagger} := {}^{-}A$$
$$({}^{-}A)^{\dagger} := {}^{+}A$$

Definition 7.31 (Syntax):
The μμ̃-calculus syntax is given by the following mutually defined inductive family of judgments, respectively for configurations, generalized terms and generalized weak-head normal forms.

data $\_ \vdash^{c} : \mathrm{ctx} \to \mathrm{Type}$
data $\_ \vdash^{t} \_ : \mathrm{ctx} \to \mathrm{typ}^s \to \mathrm{Type}$
data $\_ \vdash^{w} \_ : \mathrm{ctx} \to \mathrm{typ}^s \to \mathrm{Type}$

The constructors are given in Figure 7.2

Further define the following shorthands.

$\mathrm{conf} := \_ \vdash^{c}$
$\mathrm{term} := \_ \vdash^{t} \_$
$\mathrm{whn} := \_ \vdash^{w} \_$

Define the family of generalized values as follows.

$\mathrm{val} : \mathrm{ctx} \to \mathrm{typ}^s \to \mathrm{Type}$

$$\mathsf{val}\ \Gamma\ ({}^{+}A : \mathsf{typ}\ \mathsf{v}) := \mathsf{whn}\ \Gamma\ {}^{+}A$$
$$\mathsf{val}\ \Gamma\ ({}^{-}A : \mathsf{typ}\ \mathsf{v}) := \mathsf{term}\ \Gamma\ {}^{-}A$$
$$\mathsf{val}\ \Gamma\ ({}^{+}A : \mathsf{typ}\ \mathsf{n}) := \mathsf{term}\ \Gamma\ {}^{+}A$$
$$\mathsf{val}\ \Gamma\ ({}^{-}A : \mathsf{typ}\ \mathsf{n}) := \mathsf{whn}\ \Gamma\ {}^{-}A$$

Lemma 7.32 (Substitution):
μμ̃-calculus values form a substitution monoid, and μμ̃-calculus configurations form a substitution module over it. Moreover, val has decidable variables.

Proof: All by mutual induction on configurations, terms and weak head normal forms. This is not entirely easy, as the statements need to be proven in a particular order, but it is standard metatheory so we will not give further details. ■

$$\frac{A : \mathsf{typ}\ p \quad t : \Gamma \vdash^{\mathsf{t}} {}^{+}A \quad e : \Gamma \vdash^{\mathsf{t}} {}^{-}A}{\langle t \mid p \mid e \rangle : \Gamma \vdash^{\mathsf{c}}} \qquad \frac{c : \Gamma \blacktriangleright {}^{-}A \vdash^{\mathsf{c}}}{\mu\, c : \Gamma \vdash^{\mathsf{t}} {}^{+}A} \qquad \frac{c : \Gamma \blacktriangleright {}^{+}A \vdash^{\mathsf{c}}}{\tilde{\mu}\, c : \Gamma \vdash^{\mathsf{t}} {}^{-}A} \qquad \frac{v : \Gamma \vdash^{\mathsf{w}} \mathcal{A}}{\mathsf{whn}\, v : \Gamma \vdash^{\mathsf{t}} \mathcal{A}}$$

$$\frac{i : \Gamma \ni \mathcal{A}}{\mathsf{var}\, i : \Gamma \vdash^{\mathsf{w}} \mathcal{A}} \qquad \frac{}{\lambda_0\{\} : \Gamma \vdash^{\mathsf{w}} {}^{-}0} \qquad \frac{}{\lambda_\top\{\} : \Gamma \vdash^{\mathsf{w}} {}^{+}\top} \qquad \frac{}{() : \Gamma \vdash^{\mathsf{w}} {}^{+}1} \qquad \frac{c : \Gamma \vdash^{\mathsf{c}}}{\lambda_1\{c\} : \Gamma \vdash^{\mathsf{w}} {}^{-}1}$$

$$\frac{c : \Gamma \vdash^{\mathsf{c}}}{\lambda_\bot\{c\} : \Gamma \vdash^{\mathsf{w}} {}^{+}\bot} \qquad \frac{}{[] : \Gamma \vdash^{\mathsf{w}} {}^{-}\bot} \qquad \frac{w_1 : \Gamma \vdash^{\mathsf{w}} {}^{+}A \quad w_2 : \Gamma \vdash^{\mathsf{w}} {}^{+}B}{(w_1, w_2) : \Gamma \vdash^{\mathsf{w}} {}^{+}A \otimes B} \qquad \frac{c : \Gamma \blacktriangleright {}^{+}A \blacktriangleright {}^{+}B \vdash^{\mathsf{c}}}{\lambda_\otimes\{c\} : \Gamma \vdash^{\mathsf{w}} {}^{-}A \otimes B}$$

$$\frac{c : \Gamma \blacktriangleright {}^{-}A \blacktriangleright {}^{-}B \vdash^{\mathsf{c}}}{\lambda_\parr\{c\} : \Gamma \vdash^{\mathsf{w}} {}^{+}A \parr B} \qquad \frac{w_1 : \Gamma \vdash^{\mathsf{w}} {}^{-}A \quad w_2 : \Gamma \vdash^{\mathsf{w}} {}^{-}B}{[w_1, w_2] : \Gamma \vdash^{\mathsf{w}} {}^{-}A \parr B} \qquad \frac{v : \Gamma \vdash^{\mathsf{w}} {}^{+}A}{\mathsf{inl}\, v : \Gamma \vdash^{\mathsf{w}} {}^{+}A \oplus B}$$

$$\frac{v : \Gamma \vdash^{\mathsf{w}} {}^{+}B}{\mathsf{inr}\, v : \Gamma \vdash^{\mathsf{w}} {}^{+}A \oplus B} \qquad \frac{c_1 : \Gamma \blacktriangleright {}^{+}A \vdash^{\mathsf{c}} \quad c_2 : \Gamma \blacktriangleright {}^{+}B \vdash^{\mathsf{c}}}{\lambda_\oplus\{c_1, c_2\} : \Gamma \vdash^{\mathsf{w}} {}^{-}A \oplus B} \qquad \frac{c_1 : \Gamma \blacktriangleright {}^{-}A \vdash^{\mathsf{c}} \quad c_2 : \Gamma \blacktriangleright {}^{-}B \vdash^{\mathsf{c}}}{\lambda_\& \{c_1, c_2\} : \Gamma \vdash^{\mathsf{w}} {}^{+}A \,\&\, B}$$

$$\frac{v : \Gamma \vdash^{\mathsf{w}} {}^{-}A}{\mathsf{fst}\, v : \Gamma \vdash^{\mathsf{w}} {}^{-}A \,\&\, B} \qquad \frac{v : \Gamma \vdash^{\mathsf{w}} {}^{-}B}{\mathsf{snd}\, v : \Gamma \vdash^{\mathsf{w}} {}^{-}A \,\&\, B}$$

$$\frac{t : \Gamma \vdash^{\mathsf{t}} {}^{+}A}{\downarrow t : \Gamma \vdash^{\mathsf{w}} {}^{+}\downarrow A} \qquad \frac{c : \Gamma \blacktriangleright {}^{+}A \vdash^{\mathsf{c}}}{\lambda_\downarrow\{c\} : \Gamma \vdash^{\mathsf{w}} {}^{-}\downarrow A} \qquad \frac{c : \Gamma \blacktriangleright {}^{-}A \vdash^{\mathsf{c}}}{\lambda_\uparrow\{c\} : \Gamma \vdash^{\mathsf{w}} {}^{+}\uparrow A} \qquad \frac{t : \Gamma \vdash^{\mathsf{t}} {}^{-}A}{\uparrow t : \Gamma \vdash^{\mathsf{w}} {}^{-}\uparrow A} \qquad \frac{v : \Gamma \vdash^{\mathsf{w}} {}^{-}A}{\ominus v : \Gamma \vdash^{\mathsf{w}} {}^{+}\ominus A}$$

$$\frac{c : \Gamma \blacktriangleright {}^{-}A \vdash^{\mathsf{c}}}{\lambda_\ominus\{c\} : \Gamma \vdash^{\mathsf{w}} {}^{-}\ominus A} \qquad \frac{c : \Gamma \blacktriangleright {}^{+}A \vdash^{\mathsf{c}}}{\lambda_\neg\{c\} : \Gamma \vdash^{\mathsf{w}} {}^{+}\neg A} \qquad \frac{v : \Gamma \vdash^{\mathsf{w}} {}^{+}A}{\neg v : \Gamma \vdash^{\mathsf{w}} {}^{-}\neg A} \qquad \frac{v : \Gamma \vdash^{\mathsf{w}} {}^{+}A/\mu A}{\mathsf{con}\, v : \Gamma \vdash^{\mathsf{w}} {}^{+}\mu A}$$

$$\frac{c : \Gamma \blacktriangleright {}^{+}A/\mu A \vdash^{\mathsf{c}}}{\lambda_\mu\{c\} : \Gamma \vdash^{\mathsf{w}} {}^{-}\mu A} \qquad \frac{c : \Gamma \blacktriangleright {}^{-}A/\nu A \vdash^{\mathsf{c}}}{\lambda_\nu\{c\} : \Gamma \vdash^{\mathsf{w}} {}^{+}\nu A} \qquad \frac{v : \Gamma \vdash^{\mathsf{w}} {}^{-}A/\nu A}{\mathsf{out}\, v : \Gamma \vdash^{\mathsf{w}} {}^{-}\nu A}$$

Figure 7.2 – μμ̃-calculus Syntax

7.2.1 Patterns

We now define the infrastructure for patterns: first the *observable* subset of side-annotated types, which will appear in the OGS game, then the patterns, their embedding into values, and finally the splitting of values into a pattern and a filling, together with the associated refolding lemmas. In JWA we called the observable types "negative", but here this word is already quite overloaded so we call them "private" instead. Recall that these are the types that will appear in the OGS construction, as they denote syntactic objects whose sharing between players is mediated by variables. Their definition follows the pattern of values, with only CBV consumers and CBN producers being considered private.

Definition 7.33 (Private Types):
Define private types as a subset of types given by the following predicate.

$$\begin{aligned}
&\text{is-priv} : \text{typ}^s \to \text{SProp}\\
&\text{is-priv}\ ({}^+A : \text{typ v}) := \bot\\
&\text{is-priv}\ ({}^-A : \text{typ v}) := \top \qquad \text{typ}^{\text{priv}} := \textstyle\int_{\text{typ}^s} \text{is-priv}\\
&\text{is-priv}\ ({}^+A : \text{typ n}) := \top \qquad \text{ctx}^{\text{priv}} := \textstyle\int_{\text{ctx}} (\text{All is-priv})\\
&\text{is-priv}\ ({}^-A : \text{typ n}) := \bot
\end{aligned}$$

With syntax, values and private (OGS) types defined we can properly start the language machine instantiation. This starts by defining observations, and as for JWA, we first go through the dual notion of *patterns*.

Definition 7.34 (Patterns):
Define the inductive family of ultimate patterns $\text{data pat} : \text{typ}^s \to \text{Type}$ with the following constructors.

$$\frac{A : \text{typ n}}{\blacksquare^{\mathsf{v}}_A : \text{pat}\ {}^+A}$$

$$\frac{A : \text{typ v}}{\blacksquare^{\mathsf{n}}_A : \text{pat}\ {}^-A} \qquad \frac{}{()^p : \text{pat}\ {}^+1} \qquad \frac{}{[]^p : \text{pat}\ {}^-\bot} \qquad \frac{p_1 : \text{pat}\ {}^+A \quad p_2 : \text{pat}\ {}^+B}{(p_1,p_2)^p : \text{pat}\ {}^+A \otimes B}$$

$$\frac{p_1 : \text{pat}\ {}^-A \quad p_2 : \text{pat}\ {}^-B}{[p_1,p_2]^p : \text{pat}\ {}^-A \parr B} \qquad \frac{p : \text{pat}\ {}^+A}{\text{inl}^p\ p : \text{pat}\ {}^+A \oplus B} \qquad \frac{p : \text{pat}\ {}^+B}{\text{inr}^p\ p : \text{pat}\ {}^+A \oplus B}$$

$$\frac{p : \text{pat}\ {}^-A}{\text{fst}^p\ v : \text{pat}\ {}^-A \,\&\, B} \qquad \frac{p : \text{pat}\ {}^-B}{\text{snd}^p\ v : \text{pat}\ {}^-A \,\&\, B} \qquad \frac{p : \text{pat}\ {}^+A}{\downarrow^p p : \text{pat}\ {}^+\downarrow A}$$

$$\frac{p : \mathrm{pat}\ {}^{-}A}{\uparrow^p p : \mathrm{pat}\ {}^{-}{\uparrow}A} \qquad \frac{p : \mathrm{pat}\ {}^{-}A}{\ominus^p p : \mathrm{pat}\ {}^{+}{\ominus}A} \qquad \frac{p : \mathrm{pat}\ {}^{+}A}{\neg^p p : \mathrm{pat}\ {}^{-}{\neg}A} \qquad \frac{p : \mathrm{pat}\ {}^{+}A/\mu A}{\mathrm{con}^p\ p : \mathrm{pat}\ {}^{+}\mu A}$$

$$\frac{p : \mathrm{pat}\ {}^{-}A/\nu A}{\mathrm{out}^p\ p : \mathrm{pat}\ {}^{-}\nu A}$$

Define their *domain* by the function $\mathrm{dom} : \mathrm{pat}\ A \to \mathrm{ctx}^{\mathrm{priv}}$ as follows.

$$\begin{aligned}
\mathrm{dom}\ \blacksquare^{\mathsf{v}}_A &:= \varepsilon \blacktriangleright {}^{+}A & \mathrm{dom}\ (\mathrm{fst}^p\ p) &:= \mathrm{dom}\ p \\
\mathrm{dom}\ \blacksquare^{\mathsf{n}}_A &:= \varepsilon \blacktriangleright {}^{-}A & \mathrm{dom}\ (\mathrm{snd}^p\ p) &:= \mathrm{dom}\ p \\
\mathrm{dom}\ ()^p &:= \varepsilon & \mathrm{dom}\ {\downarrow}^p p &:= \mathrm{dom}\ p \\
\mathrm{dom}\ []^p &:= \varepsilon & \mathrm{dom}\ {\uparrow}^p p &:= \mathrm{dom}\ p \\
\mathrm{dom}\ (p_1, p_2)^p &:= \mathrm{dom}\ p_1 \mathbin{+\!\!+} \mathrm{dom}\ p_2 & \mathrm{dom}\ ({\ominus}^p p) &:= \mathrm{dom}\ p \\
\mathrm{dom}\ [p_1, p_2]^p &:= \mathrm{dom}\ p_1 \mathbin{+\!\!+} \mathrm{dom}\ p_2 & \mathrm{dom}\ ({\neg}^p p) &:= \mathrm{dom}\ p \\
\mathrm{dom}\ (\mathrm{inl}^p\ p) &:= \mathrm{dom}\ p & \mathrm{dom}\ (\mathrm{con}^p\ p) &:= \mathrm{dom}\ p \\
\mathrm{dom}\ (\mathrm{inr}^p\ p) &:= \mathrm{dom}\ p & \mathrm{dom}\ (\mathrm{out}^p\ p) &:= \mathrm{dom}\ p
\end{aligned}$$

With patterns defined, we introduce the embedding and splitting in one go. We do not spell out their definition but only characterize it by its refolding properties as writing them down would become quite unwieldy.

Definition 7.35 (Value Splitting):
Define the following functions

$$\begin{aligned}
&\text{p-emb}\ \{A\}\ (p : \mathrm{pat}\ A) : \mathrm{val}\ (\mathrm{dom}\ p)\ A \\
&\text{split-pat}\ \{\Gamma : \mathrm{ctx}^{\mathrm{priv}}\}\ \{A\} : \mathrm{val}\ \Gamma\ A \to \mathrm{pat}\ A \\
&\text{split-fill}\ \{\Gamma : \mathrm{ctx}^{\mathrm{priv}}\}\ \{A\}\ (v : \mathrm{val}\ \Gamma\ A) : \mathrm{dom}\ (\text{split-pat}\ v) \multimap [\ \mathrm{val}\ ] \mapsto \Gamma,
\end{aligned}$$

characterized by the following two properties.

(1) For all $\Gamma : \mathrm{ctx}^{\mathrm{priv}}$, $A : \mathrm{typ}$ and $v : \mathrm{val}\ \Gamma\ A$,

$$(\text{p-emb}\ (\text{split-pat}\ v))[\text{split-fill}\ v] = v.$$

(2) For all $\Gamma : \mathrm{ctx}^{\mathrm{priv}}$, $A : \mathrm{typ}$, $p : \mathrm{pat}\ A$ and $\gamma : \mathrm{dom}\ p \multimap [\ \mathrm{val}\ ] \mapsto \Gamma$,

$$(p, \gamma) \approx (\text{split-pat}\ (\text{p-emb}\ p)[\gamma], \text{split-fill}\ (\text{p-emb}\ p)[\gamma]).$$

Proof: p-emb is defined by direct induction on patterns. split-pat is defined through the following two mutually inductive auxiliary functions.

$$\begin{aligned}
&\text{split-pat}^{\mathsf{v}}\ \{\Gamma : \mathrm{ctx}^{\mathrm{priv}}\}\ \{A : \mathrm{typ}\ \mathsf{v}\} : \mathrm{whn}\ \Gamma\ {}^{+}A \to \mathrm{pat}\ {}^{+}A \\
&\text{split-pat}^{\mathsf{n}}\ \{\Gamma : \mathrm{ctx}^{\mathrm{priv}}\}\ \{A : \mathrm{typ}\ \mathsf{n}\} : \mathrm{whn}\ \Gamma\ {}^{-}A \to \mathrm{pat}\ {}^{-}A
\end{aligned}$$

Similarly, split-fill is defined through the following two mutually inductive auxiliary functions.

$$\begin{aligned}&\text{split-fill}^{\mathsf{v}}\ \{\Gamma : \text{ctx}^{\text{priv}}\}\ \{A : \text{typ}\ \mathsf{v}\}\ (v : \text{whn}\ \Gamma\ {}^{+}A)\\&\quad : \text{dom}\ (\text{split-pat}^{\mathsf{v}}\ v) \dashv[\ \text{val}\ ]\!\rightarrow \Gamma\\&\text{split-fill}^{\mathsf{n}}\ \{\Gamma : \text{ctx}^{\text{priv}}\}\ \{A : \text{typ}\ \mathsf{n}\}\ (v : \text{whn}\ \Gamma\ {}^{-}A)\\&\quad : \text{dom}\ (\text{split-pat}^{\mathsf{n}}\ v) \dashv[\ \text{val}\ ]\!\rightarrow \Gamma\end{aligned}$$

The first property (refolding) is then obtained by similar decomposition into two auxiliary properties respectively concerned with CBV weak head normal terms and CBN weak head normal coterms. The second property (unicity of splitting) is proven by direct induction on patterns. ■

We can finally give the $\mu\tilde{\mu}$-calculus *observations*, as patterns at the dual side-annotated type.

Definition 7.36 (Observation):
Define *observations* as the following binding family.

$$\begin{aligned}&\text{obs} : \text{Bind}\ \text{ctx}^{\text{priv}}\ \text{typ}^{\text{priv}}\\&\text{obs} := \begin{bmatrix}\text{Op}\ A & := \text{pat}\ A^{\dagger}\\ \text{holes}\ p & := \text{dom}\ p\end{bmatrix}\end{aligned}$$

7.2.2 $\mu\tilde{\mu}$-calculus Language Machine

We now define the rest of the $\mu\tilde{\mu}$-calculus language machine, namely the evaluation and application maps.

Definition 7.37 (Evaluation):
Define evaluation by iteration of an evaluation step as follows.

$$\begin{aligned}&\text{eval}\ \{\Gamma : \text{ctx}^{\text{priv}}\} : \text{conf}\ \Gamma \to \text{Delay}\ \left(\text{Nf}^{\text{obs}}_{\text{val}}\ \Gamma\right)\\&\text{eval} := \text{iter}\ (\text{ret} \circ \text{eval-step})\\&\\&\text{eval-step}\ \{\Gamma : \text{ctx}^{\text{priv}}\} : \text{conf}\ \Gamma \to \text{Nf}^{\text{obs}}_{\text{val}}\ \Gamma + \text{conf}\ \Gamma\end{aligned}$$

```
eval-step ⟨ μ c |v| e ⟩                 := inr c[var, e]
eval-step ⟨ t |n| μ̃ c ⟩                 := inr c[var, t]
eval-step ⟨ whn v |v| μ̃ c ⟩             := inr c[var, v]
eval-step ⟨ μ c |n| whn k ⟩             := inr c[var, k]
eval-step ⟨ whn v |v| whn (var i) ⟩     := inl ((i ▪ split-pat v), split-fill v)
eval-step ⟨ whn (var i) |n| whn k ⟩     := inl ((i ▪ split-pat k), split-fill k)
eval-step ⟨ whn (var i) |v| whn k ⟩     := ex-falso (elt-upgr i)
eval-step ⟨ whn v |n| whn (var i) ⟩     := ex-falso (elt-upgr i)
eval-step ⟨ whn () |v| whn ↙₁{c} ⟩      := inr c
eval-step ⟨ whn λ⊥{c} |n| whn [] ⟩      := inr c
eval-step ⟨ whn (v₁,v₂) |v| whn ↙⊗{c} ⟩ := inr c[var, v₁, v₂]
eval-step ⟨ whn λ⅋{c} |n| whn [k₁,k₂] ⟩ := inr c[var, k₁, k₂]

eval-step ⟨ whn (inl v) |v| whn ↙⊕{c₁,c₂} ⟩ := inr c₁[var, v]
eval-step ⟨ whn (inr v) |v| whn ↙⊕{c₁,c₂} ⟩ := inr c₂[var, v]
eval-step ⟨ whn λ&{c₁,c₂} |n| whn (fst k) ⟩ := inr c₁[var, k]
eval-step ⟨ whn λ&{c₁,c₂} |n| whn (snd k) ⟩ := inr c₂[var, k]
eval-step ⟨ whn ↓t |v| whn ↙↓{c} ⟩      := inr c[var, t]
eval-step ⟨ whn λ↑{c} |n| whn ↓e ⟩      := inr c[var, e]
eval-step ⟨ whn ⊖k |v| whn ↙⊖{c} ⟩      := inr c[var, k]
eval-step ⟨ whn λ¬{c} |n| whn ¬v ⟩      := inr c[var, v]
eval-step ⟨ whn (con v) |v| whn ↙μ{c} ⟩ := inr c[var, v]
eval-step ⟨ whn λν{c} |n| whn (out k) ⟩ := inr c[var, k]
```

Remark 7.38: It should not be obvious, but it can be checked that all the (co)terms and weak head normal (co)terms by which we are substituting are indeed (co)values, with the type polarity and side annotation properly lining up.

Definition 7.39 (Observation Application):
Define observation application as follows.

```
apply {Γ : ctx^priv} {A : typ^priv} (v : val Γ A) (o : obs .Op A)
    : obs .holes o –[ val ]→ Γ → conf Γ
apply {Γ} {⁺A} v o γ := ⟨v |n| (p-emb o)[γ]⟩
apply {Γ} {⁻A} v o γ := ⟨(p-emb o)[γ] |v| v⟩
```

We can now define the language machine.

Definition 7.40 (Language Machine):
The μμ̃-calculus language machine is given by the following record.

$$\tilde{\mathrm{MM}} : \mathrm{LangMachine}\ \mathrm{obs}\ \mathrm{val}\ \mathrm{conf}$$

$$\tilde{\mathrm{MM}} := \begin{bmatrix} \mathrm{eval} & := \mathrm{eval} \\ \mathrm{apply} & := \mathrm{apply} \\ \text{eval-ext} & := \dots \\ \text{apply-ext} & := \dots \end{bmatrix}$$

Let us now sketch the proof of the correctness hypotheses.

Lemma 7.41 (μμ̃ Respects Substitution):
The μμ̃-calculus language machine respects substitution.

Proof:

- eval-sub Given $\Gamma, \Delta : \mathrm{ctx}^{\mathrm{priv}}$, $c : \mathrm{conf}\ \Gamma$ and $\sigma : \Gamma \dashv[\ \mathrm{val}\ ]\mapsto \Delta$, we need to prove the following.

  $$\mathrm{eval}\ c[\sigma] \cong \mathrm{eval}\ c \gg\!= \lambda\ n \mapsto \mathrm{eval}\ (\text{nf-emb}\ n)[\sigma]$$

  Proceed by tower induction, then pattern match on c, following the case tree of eval-step. In case of a redex, i.e., when eval-step returns $\mathrm{inr}\ c'[\gamma]$ for some c' and γ, commute γ and σ in the LHS and conclude by coinduction hypothesis. In case of a normal form, i.e., when eval-step returns $\mathrm{inl}\ ((i \blacksquare \text{split-pat}\ v), \text{split-fill}\ v)$ for some i and v, apply Definition 7.35(1) to rewrite $(\text{nf-emb}\ ((i \blacksquare \text{split-pat}\ v), \text{split-fill}\ v))[\sigma]$ into either $\langle v[\sigma]\ |\mathrm{n}|\ \sigma\ i\rangle$ or $\langle \sigma\ i\ |\mathrm{v}|\ v[\sigma]\rangle$, depending on the polarity of the type, and conclude by reflexivity.
- apply-sub By direct application of substitution fusion.
- eval-nf By direct application of Definition 7.35(2). ■

Lemma 7.42 (μμ̃ Finite Redexes):
The μμ̃-calculus language machine has finite redexes

Proof: As for Jwa, the μμ̃-calculus verifies a stronger property than well-foundedness of $\prec$, namely that there for any observation o_2, there is no o_1 such that $o_1 \prec o_2$. In other words, applying an observation to a non-variable value necessarily yields a redex. This is proven by direct case-analysis of the value and the observation. ■

We can now conclude correctness of the Ogs interpretation w.r.t. substitution equivalence.

Definition 7.43 (Evaluation Relation):
For $c : \mathsf{conf}\ (\varepsilon \blacktriangleright {}^{-}1)$, define the following big step *evaluation relation*.

$$c \Downarrow ()^p := (\mathsf{fst} \langle\$\rangle\ \mathsf{eval}\ c) \approx \mathsf{ret}\ (\mathsf{top} \cdot ()^p)$$

Theorem 7.44 (Ogs Correctness):
For all $\Gamma : \mathsf{ctx}^{\mathsf{priv}}$ and $c_1, c_2 : \mathsf{conf}\ \Gamma$, if $[\![c_1]\!]^+_M \approx [\![c_2]\!]^+_M$, then for all $\gamma : \Gamma \dashv[\ \mathsf{val}\ ]\vdash (\varepsilon \blacktriangleright {}^{-}1)$,

$$c_1[\gamma] \Downarrow ()^p \quad \leftrightarrow \quad c_2[\gamma] \Downarrow ()^p.$$

Proof: By Theorem 4.33 applied to $\mathrm{M\tilde{M}}$, with hypotheses proven in Lemma 7.32, Lemma 7.41 and Lemma 7.42, obtain

$$\mathsf{fst} \langle\$\rangle\ \mathsf{eval}\ c_1[\gamma] \quad \approx \quad \mathsf{fst} \langle\$\rangle\ \mathsf{eval}\ c_2[\gamma].$$

Conclude by the fact that $\approx$ is an equivalence relation. ■

We will stop here and skip the Nf bisimulation correctness as well as the extended results for configurations in arbitrary (non-private) contexts $\Gamma : \mathsf{ctx}$. If needed, these can be obtained exactly as for Jwa, by patching the Ogs or Nf interpretation equivalence to first quantify over patterns for each type in Γ.

7.3 Untyped Weak Head λ-calculus

Our first two examples were in several aspects quite similar: two simply-typed languages, rather low level and centered around explicit control flow using some form of first-class continuations. Let us now turn to a radically different language: pure untyped λ-calculus with weak head reduction semantics. There will be several hurdles to overcome, so let us give an overview, in increasing order of difficulty.

Typing The language is untyped, but our framework for substitution requires some set of types. This is quite easy to overcome with a benign change of perspective, as we can see any untyped language as an un**i**typed language, that is, a language with a single type.

Configurations The language is presented with natural deduction style judgments, as opposed to language machines and the first two examples, which have sequent calculus style judgments. What we mean by this, is that the λ-calculus is usually presented without any notion which would be similar to a language machine's configurations: the thing on which the evaluator operates and which is only indexed by a scope. The way out is two-fold.

First, we switch to a common, but perhaps not the most standard presentation of weak head reduction: some variant of the Krivine machine [29]. This is a bit of a goalpost moving, as arguably we will not present a λ-calculus language machine but rather a Krivine language machine. However it is best to go with the flow of the axiomatization and adopt the abstract machine mindset. This will pay off when proving the correctness theorem hypotheses.

[29] Pierre-Louis Curien, *Categorical Combinators, Sequential Algorithms, and Functional Programming*, 1993.

Second, since continuations, a.k.a. *stacks*, will start floating around our Krivine machine configurations, we will need to introduce formal continuations, i.e., a new kind of variable denoting continuations. Indeed, during the Ogs game and in the Nf interpretation, we will need to exchange symbolic continuations with our opponent. To distinguish them from the already existing "term" variables there is a simple mean: typing. We had a single type, now we already have two: one for terms and one for continuations. To avoid confusion, we will stop referring to them as types and instead call them *sorts*.

In a sense, our Krivine machine will operate on *radically open* configurations, in the sense that not only terms might contain free variables, but also stacks!

Observations The usual intuition behind the design of the binding family of observations is that they denote some kind of copatterns, or eliminators. This understanding is enough to get a satisfying definition when the language already circles around this concept, like our first two examples. More pragmatically, as our axiomatization derives the normal forms from these observations, we better engineer them to fit the normal forms we intend to have. For call-by-value λ-calculus, as sketched in the introduction (§1.3), it does not take much squinting to see that the two shapes of normal forms—a value and a stuck application in an evaluation context—do indeed look like eliminators. Namely calls (on opponent functions) and returns (on implicit opponent continuations).

For weak head reduction λ-calculus, there are two shapes for normal forms: lambda abstractions $\lambda x.T$ and stuck applications $xT_1 ... T_n$. Applications are not too difficult to see as being eliminators. We deduce that in this world, functions are eliminated by giving not one but a whole spine of arguments. In fact, one can recognize this spine as a stack, which is nice since our earlier choices start to make more sense. However, the lambda abstraction is rather devilish. It does not look like it is stuck on anything, so once again, this must be an elimination on an implicit context/stack variable. But which part of $\lambda x.T$ is the pattern (the static part) and the filling? Lets look at it backwards: what kind of elimination would make sense for a stack? Stacks are sequences of terms, so it would make sense to pattern match on them. And indeed, the $\lambda x.T$ normal form is simply a request to *grab* the next argument from the stack. If the stack is empty, this request cannot be fulfilled. But if it is not, the opponent may answer the request by a "here it is", giving us two new handles, a term variable for the head and a stack variable for

the rest. Another way to look at requests is to understand them as fully evaluated, or *forced* functions.

Putting things back together, what we just did is to introduce a *third* sort, for argument *requests*, which will accordingly need be accompanied by request variables. Eliminating a stack introduces a new *request variable* for the opponent, and eliminating a request introduces a term variable and a stack variable.

Let us formalize that!

7.3.1 Syntax and Semantics

Definition 7.45 (Sorts):
Define *sorts* as the following data type.

$$\mathsf{data\ sort} : \mathsf{Type} := \mathsf{tm} \mid \mathsf{stk} \mid \mathsf{req}$$

Further define *contexts* by the following shorthand $\mathsf{ctx} := \mathsf{Ctx\ sort}$.

Instead of writing three mutually defined judgments for terms, stacks and requests, we will simply use a single syntactic judgment, indexed by sorts. This will have the happy side effect to fuse the three different variables in a single sorted construct. Note that we did not talk a lot about *configurations*, but they unsurprisingly pair a term with a stack.

Definition 7.46 (Syntax):
Define the unified judgment for *syntax* as the data type $\mathsf{data\ syn} : \mathsf{Type}^{\mathsf{ctx},\,\mathsf{sort}}$ with the following constructors.

$$\frac{\Gamma \ni s}{\mathsf{var}\ i : \mathsf{syn}\ \Gamma\ s} \qquad \frac{a : \mathsf{syn}\ \Gamma\ \mathsf{tm} \quad b : \mathsf{syn}\ \Gamma\ \mathsf{tm}}{a \cdot b : \mathsf{syn}\ \Gamma\ \mathsf{tm}} \qquad \frac{r : \mathsf{syn}\ \Gamma\ \mathsf{req}}{\lfloor r \rfloor_r : \mathsf{syn}\ \Gamma\ \mathsf{tm}}$$

$$\frac{a : \mathsf{syn}\ (\Gamma \blacktriangleright \mathsf{tm})\ \mathsf{tm}}{\lambda\, a : \mathsf{syn}\ \Gamma\ \mathsf{req}} \qquad \frac{a : \mathsf{syn}\ \Gamma\ \mathsf{tm} \quad k : \mathsf{syn}\ \Gamma\ \mathsf{stk}}{a :: k : \mathsf{syn}\ \Gamma\ \mathsf{stk}}$$

Further define the judgment for *configurations* as the data type $\mathsf{data\ conf} : \mathsf{Type}^{\mathsf{ctx}}$ with the following constructor.

$$\frac{a : \mathsf{syn}\ \Gamma\ \mathsf{tm} \quad k : \mathsf{syn}\ \Gamma\ \mathsf{stk}}{\langle\, a \parallel k \,\rangle : \mathsf{conf}\ \Gamma}$$

Remark 7.47: One may be surprised by the absence of empty stack: the base case for stacks is provided by the stack variables.

Lemma 7.48 (Substitution):
The family syn forms a substitution monoid with decidable variables, and conf forms a substitution module over it.

Proof: Although the meaning of syn is perhaps still puzzling, it can be opaquely viewed as an arbitrary syntax with bindings. As such, the renaming and substitution operators and their laws can be derived by standard means.■

We can now define the binding family of observations. Because technically there is exactly one observation at each sort, we could take the tricky $\lambda\ s \mapsto \mathbf{1}$ definition. The holes map would be quite puzzling though, so that we prefer defining a special purpose family and give these observations nice names.

Definition 7.49 (Observations):
Define the family of *observations* as the data type data o-fam : $\mathrm{Type}^{\mathrm{sort}}$ with the following constructors.

$$\frac{}{\text{force} : \text{o-fam tm}} \qquad \frac{}{\text{grab} : \text{o-fam stk}} \qquad \frac{}{\text{push} : \text{o-fam req}}$$

Define their *domain* by the following function.

```
o-dom {s} : o-fam s → ctx
o-dom force := ε ▸ stk
o-dom grab  := ε ▸ req
o-dom push  := ε ▸ tm ▸ stk
```

Further define their binding family as follows.

```
obs : Bind S T
  [ Op s   := o-fam s
  | holes o := o-dom o
```

Let us recapitulate. We have a force observation on terms, which has one argument: a stack in which it should be evaluated. We have a grab observation on stacks, which has one argument: a request, that is, a lambda abstraction. And finally we have a push observation on requests, which has two arguments: a head term and a tail stack. We are ready to see the evaluator.

Definition 7.50 (Evaluator):
Define the *evaluation* map by iteration of the following *evaluation step* map.

```
eval {Γ} : conf Γ → Delay (Nf^obs_syn Γ)
eval := iter (ret ∘ eval-step)
```

```
eval-step {Γ} : conf Γ → Nf^obs_syn Γ + conf Γ
eval-step   ⟨ var i ∥ k ⟩     := inl (i ▪ force, [k])
eval-step   ⟨ a · b ∥ k ⟩     := inr ⟨ a ∥ b :: k ⟩
eval-step   ⟨ ⌊r⌋_r ∥ var i ⟩ := inl (i ▪ grab, [r])
eval-step ⟨ ⌊var i⌋_r ∥ b :: k ⟩ := inl (i ▪ push, [b, k])
eval-step   ⟨ ⌊λ a⌋_r ∥ b :: k ⟩ := inr ⟨ a[var, b] ∥ k ⟩
```

Let us rejoice! The journey to obtain this may have been convoluted, but the resulting observations and evaluator are even shorter than for Jwa. But before jumping to the proofs, let us actually define the observation application map and finally the language machine.

Definition 7.51 (Observation Application):
Define the observation application map as follows.

```
apply {Γ s} (x : syn Γ s) (o : o-fam s) : o-dom o ⊣[ syn ]↦ Γ → conf Γ
apply a force γ := ⟨ a ∥ γ top ⟩
apply k grab γ  := ⟨ ⌊γ top⌋_r ∥ k ⟩
apply r push γ := ⟨ ⌊r⌋_r ∥ γ (pop top) :: γ top ⟩
```

Definition 7.52 (Language Machine):
Define the Krivine language machine by the following record.

```
Krivine : LangMachine_obs syn
Krivine :=
  ⌈ eval       := eval
  | apply      := apply
  | eval-ext   := ...
  | apply-ext  := ...
```

Lemma 7.53 (Machine Respects Substitution):
The Krivine language machine respects substitution.

Proof:

- eval-sub Given Γ, Δ : ctx, c : conf Γ and $\sigma : \Gamma \dashv[\ \text{syn}\ ]\mapsto \Delta$, we need to prove the following statement.

  $$\text{eval}\ c[\sigma] \cong \text{eval}\ c \gg\!= \lambda\ n \mapsto \text{eval}\ (\text{nf-emb}\ n)[\sigma]$$

  Proceed by coinduction, then destruct c following the pattern of eval-step.

  - When $c := \langle\ \text{var}\ i \parallel k\ \rangle$, by reflexivity.
  - When $c := \langle\ a \cdot b \parallel k\ \rangle$, by synchronization and coinduction hypothesis.

- When $c := \langle\, \lfloor r \rfloor_r \parallel \mathrm{var}\ i \,\rangle$, by reflexivity.
- When $c := \langle\, \lfloor \mathrm{var}\ i \rfloor_r \parallel b :: k \,\rangle$, by reflexivity.
- When $c := \langle\, \lfloor \lambda\ a \rfloor_r \parallel b :: k \,\rangle$, the LHS is definitionally equal to

  $$\mathrm{tau}\ (\mathrm{eval}\ \langle\, a[\sigma[\mathrm{pop}], \mathrm{top}][\mathrm{var}, b] \parallel k[\sigma] \,\rangle).$$

  Rewrite it to the following and conclude by synchronization and coinduction hypothesis.

  $$\mathrm{tau}\ (\mathrm{eval}\ \langle\, a[\mathrm{var}, b][\sigma] \parallel k[\sigma] \,\rangle)$$

- apply-sub By direct case analysis on the observation.
- eval-nf By direct case analysis on the normal form. ∎

Lemma 7.54 (Finite Redexes):
The Krivine language machine has finite redexes.

Proof: Recall that we need to prove the following relation to be well-founded.

$$\frac{\begin{array}{c} i : \Gamma \ni s_1 \quad o_1 : \mathrm{o\text{-}fam}\ s_1 \quad \gamma_1 : \mathrm{o\text{-}dom}\ o_1 \dashv[\ \mathrm{syn}\ ]\mapsto \Gamma \quad x : \mathrm{syn}\ \Gamma\ s_2 \\ o_2 : \mathrm{o\text{-}fam}\ s_2 \quad \gamma_2 : \mathrm{o\text{-}dom}\ o_2 \dashv[\ \mathrm{syn}\ ]\mapsto \Gamma \quad H_1 : \mathrm{is\text{-}var}\ v \to \bot \\ H_2 : \mathrm{eval}\ (\mathrm{apply}\ x\ o_2\ \gamma) \cong \mathrm{ret}\ ((i \cdot o_1), \gamma_1) \end{array}}{\mathrm{bad}\ H_1\ H_2 : o_1 \prec o_2}$$

We will show that this relation only contains $\mathrm{push} \prec \mathrm{grab}$, $\mathrm{push} \prec \mathrm{force}$ and $\mathrm{grab} \prec \mathrm{force}$. As such it has chains of length at most 3 and is thus accessible.

Assume $s_1, s_2 : \mathrm{sort}$, $o_1 : \mathrm{o\text{-}fam}\ s_1$ and $o_2 : \mathrm{o\text{-}fam}\ s_2$ such that $o_1 \prec o_2$. Destruct the relation witness and introduce all the hypotheses as above. By case on o_2, let us determine o_1.

(1) When $o_2 := \mathrm{push}$, then x must be a request.
- When $x := \mathrm{var}\ i$, then H_1 is absurd.
- When $x := \lambda\ a$, then H_2 is absurd as $\mathrm{apply}\ (\lambda\ a)\ \mathrm{push}\ \gamma$ will perform an evaluation step.

(2) When $o_2 := \mathrm{grab}$, then x must be a stack.
- When $x := \mathrm{var}\ i$, then H_1 is absurd.
- When $x := b :: k$, let $r := \gamma\ \mathrm{top}$.
  - When $r := \mathrm{var}\ i$, by H_2, o_1 must be push.
  - When $r := \lambda\ a$, then H_2 is absurd.

(3) When $o_2 := \mathrm{force}$, then x must be a term. Proceed by case on x.
- When $x := \mathrm{var}\ i$, then H_1 is absurd.
- When $x := a \cdot b$, then H_2 is absurd.
- When $x := \lfloor r \rfloor_r$, pose $k := \gamma\ \mathrm{top}$.
  - When $k := \mathrm{var}\ i$, by H_2, o_1 must be grab.
  - When $k := a :: k'$, by case on r.

- When $r := \mathsf{var}\ i$, by H_2, o_1 must be push.
- When $r := \lambda\ a$, then H_2 is absurd. ∎

This concludes the correctness of the Krivine language machine! Its Nf strategies from Ch. 6 can be recognized as Levy-Longo trees [64][63]. Thus, Theorem 6.15 gives us a soundness proof of Levy-Longo tree w.r.t. weak head observational equivalence, although this fact was already known [85]. This claim should probably be properly justified better, by formalizing a clean representation of these trees and proving them isomorphic to our Nf strategies, but we will not go further than this.

[64] Jean-Jacques Lévy, "An algebraic interpretation of the λβ-calculus and a labeled λ-calculus," 1975.
[63] Giuseppe Longo, "Set-theoretical models of λ-calculus: theories, expansions, isomorphisms," 1983.
[85] Davide Sangiorgi, "A Theory of Bisimulation for the pi-Calculus," 1993.

For the sake of it, let us simply state the normal form bisimulation correctness theorem, for the last time in this thesis!

Theorem 7.55 (Nf Correctness):
Define the weak head normalization predicate as follows.

$$\_{\Downarrow} : \mathsf{conf}\ (\varepsilon \blacktriangleright \mathsf{stk}) \to \mathsf{Prop}$$
$$c{\Downarrow} := (\mathsf{fst}\ \langle\$\rangle\ \mathsf{eval}\ c) \approx \mathsf{ret}\ (\mathsf{top} \blacksquare \mathsf{grab})$$

For all $t_1, t_2 : \mathsf{syn}\ \Gamma\ \mathsf{tm}$ such that

$$[\![\langle\ t_1[\mathsf{pop}] \parallel \mathsf{var}\ \mathsf{top}\ \rangle]\!]^{\mathrm{Nf}}_{\mathrm{Krivine}} \approx [\![\langle\ t_2[\mathsf{pop}] \parallel \mathsf{var}\ \mathsf{top}\ \rangle]\!]^{\mathrm{Nf}}_{\mathrm{Krivine}},$$

then for all $\sigma : \Gamma \dashv [\ \mathsf{syn}\ ] \vdash (\varepsilon \blacktriangleright \mathsf{stk})$ and $k : \mathsf{syn}\ (\varepsilon \blacktriangleright \mathsf{stk})\ \mathsf{stk}$

$$\langle\ t_1[\sigma] \parallel k\ \rangle{\Downarrow} \leftrightarrow \langle\ t_2[\sigma] \parallel k\ \rangle{\Downarrow}.$$

Proof: By application of Theorem 6.15 to Krivine, t_1, t_2 and $[\sigma, k]$, with hypotheses proven in Lemma 7.48, Lemma 7.53 and Lemma 7.54, obtain the following.

$$\begin{aligned} &\mathsf{fst}\ \langle\$\rangle\ \mathsf{eval}\ \langle\ t_1[\mathsf{pop}] \parallel \mathsf{var}\ \mathsf{top}\ \rangle[\sigma, k] \\ \approx\ &\mathsf{fst}\ \langle\$\rangle\ \mathsf{eval}\ \langle\ t_2[\mathsf{pop}] \parallel \mathsf{var}\ \mathsf{top}\ \rangle[\sigma, k] \end{aligned}$$

Conclude by substitution fusion and equivalence. ∎

We could further post-process the above theorem to obtain a statement on the more standard λ-terms and evaluation contexts, as they embed into our Krivine machine syntax. This would clear away our non-standard stack and request variables, but we will stop our case study at this point.

8 Perspectives

I hope that this thesis has somewhat demystified operational game semantics to the type theorist. Let us review some of the most important steps we have taken.

- We started off in Ch. 2 by presenting a new data structure for coinductively representing automata in type theory. This puts a new item in the already large bag of constructions based on polynomial functors. But perhaps most importantly, we demonstrated that with a small twist, namely splitting these polynomials in halves, we can obtain well-behaved game descriptions. Thanks to this, finely typed automata prove quite suitable to represent strategies for dialog games inside type theory.
- Next, in Ch. 3, we introduced a small proof pearl: scope structures. They untie the intrinsically scoped and typed theory of substitution from the (too) concrete De-Bruijn indices. This brings a bit more flexibility in the practical formalization of syntactic objects. Subset scopes in particular were of great use in the Ogs instances we have shown (Ch. 7), although it mostly smoothed the work behind the scene, so you may have to take my word for it*.
- In Ch. 4 we arrived at operational game semantics. We constructed a generic Ogs model, proposing an axiomatization of languages with an evaluator for open programs. This axiomatization is inspired by abstract machines, taking a computational approach to operational semantics. Most notably, it leaves the syntax entirely opaque and devoid of any inductive nature, to be contrasted, e.g., with structural operational semantics [79]. Yet this is enough to construct an Ogs model, and to prove it correct w.r.t. an observational equivalence under suitable hypotheses (Ch. 5). This underlines that much like denotational semantics, operational semantics can also beneficially manage to push the clutter and technicalities of syntax out of sight.
- Finally, we reaped some rewards from all of these constructions. First we generically defined normal form bisimilarity, proving it correct by going through Ogs strategy bisimilarity (Ch. 6). Then, we instantiated our language axiomatization with three standard calculi (Ch. 7).

As always, there is the feeling that we have barely started scratching the surface. Formally proving this correctness theorem turned out to be a long and narrow track. Along the way we zoomed past many opportunities for more in-depth study. Furthermore, although we already capture a number of languages, the level of generality of our Ogs model could be improved. Indeed, we neither handle effectful languages (apart from partiality), nor polymorphic type systems, two

* As I recall, when I stopped working with the "wrong" notion of variable and introduced subset scopes, I cut the size of the Rocq code for the polarized $\mu\tilde{\mu}$-calculus instance by roughly a third.

[79] Gordon D. Plotkin, "A structural approach to operational semantics," 2004.

features for which Ogs models have already been demonstrated [54][59][49]. Let us discuss in more details what we managed to glimpse from a couple of these windows into the unknown.

[54] James Laird, "A Fully Abstract Trace Semantics for General References," 2007.
[59] Søren B. Lassen and Paul Blain Levy, "Typed Normal Form Bisimulation for Parametric Polymorphism," 2008.
[49] Guilhem Jaber and Nikos Tzevelekos, "A Trace Semantics for System F Parametric Polymorphism," 2018.

Games and Open Composition Although category theory concepts are structuring our development behind the scenes, we have made the choice to minimize their amount. Besides making the manuscript slightly more accessible, it is a pragmatic implementation choice, as category theory mechanization in intensional type theory is notoriously a can of worms. However, Levy and Staton's games and strategies, presented in Ch. 2, enjoy a rich categorical theory, which we have mostly skipped over. As a consequence, we did not say anything about the properties of Ogs as a game, beyond the trivial fact that it is symmetric.

First of all, games are regularly used as models for (parts of) linear logic. It would be interesting to see how our notion of games fare in this respect. In fact our Ogs game is strongly reminiscent of "bang" games in typical such interpretations. In both cases, the game position is a list (or scope) of possible game copies to choose from, and moving *spawns* new possible positions (by concatenation on this list).

But linear logic is not the sole provider of structure on games, as Conway also studied quite similar objects. Slightly more precisely, it is not too difficult to see the following definition (which we saw in passing in §2.2.3) as creating a variant of the Conway sum of two games.

$$\_ +_h \_ : \mathrm{HGame}\ I\ I \to \mathrm{HGame}\ J\ J \to \mathrm{HGame}\ (I \times J)\ (I \times J)$$

$$A +_h B := \begin{bmatrix} \mathrm{Move}\ (i,j) & := A.\mathrm{client}.\mathrm{Move}\ i + B.\mathrm{client}.\mathrm{Move}\ j \\ \mathrm{next}\ \ (i,j)\ (\mathrm{inl}\ m) & := (A.\mathrm{client}.\mathrm{next}\ m, j) \\ \mathrm{next}\ \ (i,i)\ (\mathrm{inr}\ m) & := (i, B.\mathrm{client}.\mathrm{next}\ m) \end{bmatrix}$$

$$\_ + \_ : \mathrm{Game}\ I\ I \to \mathrm{Game}\ J\ J \to \mathrm{Game}\ (I \times J)\ (I \times J)$$

$$A + B := \begin{bmatrix} \mathrm{client} := A.\mathrm{client} +_h B.\mathrm{client} \\ \mathrm{server} := A.\mathrm{server} +_h B.\mathrm{server} \end{bmatrix}$$

Then, we conjecture that $\mathrm{Ogs}^{\mathrm{G}}_{O} + \mathrm{Ogs}^{\mathrm{G}}_{O} \approx \mathrm{Ogs}^{\mathrm{G}}_{O}$, where $\approx$ denotes a *game bisimulation*, i.e., an isomorphism between their respective sets of moves, mediated by a suitable relation between their respective sets of positions. We have preliminary results in this direction, but using a slightly different formulation of $\mathrm{Ogs}^{\mathrm{G}}_{O}$, taking the cleaner *absolute* point of view on the naive version presented in §4.1.1 (Definition 4.3, see Remark 4.4).

Like the linear logic $⅋$ connective, this Conway sum provides a candidate for game exponentials, as $A \Rightarrow B := A^{\dagger} + B$. This combinator enjoys a corresponding composition operation, taking a strategy on $A^{\dagger} + B$ and one on $B^{\dagger} + C$ to a strategy on $A^{\dagger} + C$. This could shed new light on our slightly

ad-hoc treatment of Ogs game strategy composition. Our hope is that we will then have the necessary scaffolding to define an *open* composition which does not merely return some final observation, but a whole Ogs strategy. We conjecture that at this point, by ditching this cumbersome final scope, we will be able to slightly strengthen the adequacy theorem. This would yield a stronger conclusion than correctness, namely that Ogs model equivalence is closed under substitution (i.e., substitutive).

Nf Strategies as a Language Machine We conjecture that with a slight tweak and a suitable definition of morphisms, Nf strategies can be exhibited as the *terminal* language machine. Let us unpack this! Define the family of configurations of this machine as active Nf strategies and the family of values as the "unary" passive Nf strategies presented in Remark 6.4. For this machine, the eval map simply seeks the first ⁿret or ⁿvis move of the given active Nf strategy. The observation application map apply is more problematic, but here comes the tweak. Recall from Remark 4.7 that there were two possible variants for the design of the application map. In our development we chose the *flex* one, which additionally takes a filling as argument. We now switch to the tighter one, which assumes the filling is given by fresh variables and thus simply extends the scope of the returned configuration. In this second version, the application map for our Nf strategies machine is just function application! A promising candidate for the terminal arrow is then given by the Nf interpretation $[\![\_]\!]_M^{\mathsf{Nf}}$.

A happy benefit of this construction, is that although the Nf language machine does not support substitutions, we conjecture it has a pointed renaming structure (extending Definition 6.8). As such, the Ogs machine strategy construction can apply, and it should computes exactly the Nf-to-Ogs embedding (Definition 6.10).

Taking a step back, we conjecture that what is happening, is that our axiomatization of language machines is exceedingly close to the definition of a big-step system over the Nf game. Since big-step systems consist essentially in a coalgebra on a functor associated to the game, it should not come as a surprise that Nf strategies—the final such coalgebra—are indeed the terminal language machine. This suggests taking a closer look at the various coalgebra presentations of game strategies (small-step systems and big-step systems). Although they are usually heavier to manipulate than their coinductive counterpart, their more intensional nature can be at times useful.

A Logic for Strategies Besides correctness w.r.t. observational equivalence, a common property to investigate is the reverse implication, i.e., completeness. When both correctness and completeness are true, the model is said to be *fully abstract*. Following game semantical insights, it is largely expected that our Ogs model of effect-free languages can only hope to be complete when restricted to

innocent strategies. Innocence is a property of a strategy, essentially meaning that it plays the same moves in any two observationally equivalent situations. Logically, it is a *safety property*, in the sense that it can be expressed as the inability to play certain moves, solely based on the past history: no bad moves are ever played.

More generally, other similarly structured predicates are of interest in game semantics, such as *well-bracketedness* (following a stack discipline for answering questions) or *visibility* (only observing variables which are in the causal past). As such it would be useful to design a logic for strategy properties, with temporal features.

Such a logic on the traces arising from coinductive automata has already been proposed in the case of non-indexed interaction trees [87][94]. There are even very expressive frameworks for reasoning with arbitrary monadic computations [65]. We conjecture that it is possible to follow their lead and adapt these techniques to the indexed setting.

[87] Lucas Silver and Steve Zdancewic, "Dijkstra monads forever: termination-sensitive specifications for interaction trees," 2021.

[94] Irene Yoon, Yannick Zakowski, and Steve Zdancewic, "Formal reasoning about layered monadic interpreters," 2022.

[65] Kenji Maillard, Danel Ahman, Robert Atkey, Guido Martínez, Catalin Hritcu, Exequiel Rivas, and Éric Tanter, "Dijkstra monads for all," 2019.

Indexing, however, unlocks even more possibilities by building upon the theory of *ornaments* [70][31]. Indeed, it is not too hard to enrich the positions of any game to keep track of the history of what has been played. Then, any safety predicate on strategies over a game can be baked into a *safe* game, by requiring any move to be played to be paired with a witness that it is safe with respect to the current history. Ordinary strategies for this *safe* game may then be proven equivalent to the subset of strategies of the original game for which the safety property holds: the fundamental theorem of correct-by-construction programming. Although ornaments are still only known to some circles, the unreasonable effectiveness of correct-by-construction programming in type theory is well established. We believe that adapting this toolkit to our indexed trees could provide novel reformulations and proof techniques for more advanced game semantical questions in type theory.

[70] Conor McBride, "Ornamental Algebras, Algebraic Ornaments," 2011.

[31] Pierre-Évariste Dagand, "The essence of ornaments," 2017.

Effectful Language Machines How do we scale our constructions and proofs to effectful languages? This is the big question, as these languages are the ones where Ogs shine the most. The natural starting point is to weaken the codomain of the evaluation map to

$$\mathrm{eval} : C\ \Gamma \to M\ (\mathrm{Nf}_O\ \Gamma)$$

for some arbitrary monad M, suitably modelling the language's effects. We can play this game of replacing every instance of Delay with M throughout our development. What we shall obtain, is that disregarding indexing for simplicity, our interaction tree monad has become the following *coinductive resumption monad* [78].

$$\mathrm{Res}_{M,\Sigma}\ X := \nu A.\ M\ (X + \Sigma\ A)$$

[78] Maciej Piróg and Jeremy Gibbons, "The Coinductive Resumption Monad," 2014.

Notice that in the above situation, we do not have any ℕtau node anymore, only ℕret and ℕvis. Instead, to recover unguarded recursion we require that M is (completely) ELGOT [78], intuitively that it behaves as Delay. But partiality is ubiquitous when manipulating coinductive game strategies, as for example it is entirely expected that under some circumstances, composition of two *total* strategies may fail to be total. It is a sometimes overlooked practical insight contributed by interaction trees, that for a whole lot of applications, partiality should be built into the notion of automata. We thus conjecture that it would be more fruitful to keep partiality under our control by reinstating the ℕtau nodes and avoid depending on some particular language's monad M for evaluation effects. We should then study the following generalization of interaction trees, where M is an arbitrary (of course strictly positive) monad.

[78] Maciej Piróg and Jeremy Gibbons, "The Coinductive Resumption Monad," 2014.

$$\mathrm{itreeT}_{M,\Sigma}\ X \coloneqq \nu A.\ M\ (X + A + \Sigma\ A)$$

We conjecture that with a suitable notion of weak bisimilarity, this can be precised to form the initial ELGOT monad with a monad morphism from M and a natural transformation from Σ. The hard part however, starts even before proving any property: a good notion of *weak* bisimilarity remains elusive! In other words, given a *relator* on M, we have some trouble defining a good relator on $\mathrm{itreeT}_{M,\Sigma}$ for weak bisimilarity.

We could go on for quite some time, but perhaps this is already enough open questions! Let us finish with more down-to-earth comments on our code artifact.

Proof Engineering Throughout this thesis we have said very little of the accompanying code artifact, but it leaves a lot to be desired as it is nowhere near a reusable software library. As one would imagine, a number of design mistakes have been made during the development and partially patched out, so that it would greatly benefit from a thorough re-architecturing. In fact, because we tried to remain faithful to the actual code, some of these oddities can at times be felt in the present manuscript.

A particularly central point is the absence of any definition of *setoid*. Reasoning up-to some equivalence relation is central throughout this thesis, but it has been worked out on a case-by-case basis (mostly for value assignments and for interaction trees). As such the artifact is left with some definitions which are too strict for some use cases (substitution structures and language machines). We have tried to make up for this in the manuscript by appealing to a slightly nebulous "extensional equality" written $\approx$, akin to a fictive type class. The clean solution is quite simple: truly parametrize by setoids and setoid families instead of types everywhere it is relevant.

Milder points include spinning off our theory of substitution into a more complete and separate library. Indeed Coq currently lacks such a library implementing a modern take on intrinsically typed and scoped syntax.

* https://github.com/lapin0t/ogs

Please do not hesitate to check the online repository*: who knows, maybe by the time you are reading these lines my compulsive tendency to shuffle code around will have made everything tidier! On a more general note, do not hesitate to reach out to me if any of the above ideas spark your curiosity.

Bibliography

[1] Michael G. Abbott, Thorsten Altenkirch, Neil Ghani, and Conor McBride, "Derivatives of Containers," in *TLCA*, 2003, Lect. Notes Comput. Sci., vol. 2701, pp. 16–30. doi: 10.1007/3-540-44904-3_2.

[2] Andreas Abel and Brigitte Pientka, "Well-founded recursion with copatterns and sized types," *J. Funct. Program.*, vol. 26, p. e2, 2016, doi: 10.1017/S0956796816000022.

[3] Andreas Abel, Brigitte Pientka, David Thibodeau, and Anton Setzer, "Copatterns: programming infinite structures by observations," in *POPL*, 2013, pp. 27–38. doi: 10.1145/2429069.2429075.

[4] Samson Abramsky, "The lazy lambda calculus," in *Res. Top. Funct. Program.*, 1990, pp. 65–116. doi: 10.5555/119830.119834.

[5] Peter Aczel, Jirí Adámek, Stefan Milius, and Jirí Velebil, "Infinite trees and completely iterative theories: a coalgebraic view," *Theor. Comput. Sci.*, vol. 300, no. 1–3, pp. 1–45, 2003, doi: 10.1016/S0304-3975(02)00728-4.

[6] Jirí Adámek, Stefan Milius, and Jirí Velebil, "From Iterative Algebras to Iterative Theories," in *CMCS*, 2004, Electron. Notes Theor. Comput. Sci., vol. 106, pp. 3–24. doi: 10.1016/j.entcs.2004.02.033.

[7] Jirí Adámek, Stefan Milius, and Jirí Velebil, "Elgot Algebras," *Log. Methods Comput. Sci.*, vol. 2, no. 5, 2006.

[8] Jirí Adámek, Stefan Milius, and Jirí Velebil, "Equational properties of iterative monads," *Inf. Comput.*, vol. 208, no. 12, pp. 1306–1348, 2010.

[9] Agda Developers, "Agda." https://agda.readthedocs.io/

[10] Benedikt Ahrens and Peter LeFanu Lumsdaine, "Displayed Categories," *Log. Methods Comput. Sci.*, vol. 15, no. 1, 2019, doi: 10.23638/LMCS-15(1:20)2019.

[11] Guillaume Allais, "Generic level polymorphic n-ary functions," in *TyDe*, 2019, pp. 14–26. doi: 10.1145/3331554.3342604.

[12] Guillaume Allais, "Builtin Types Viewed as Inductive Families," in *ESOP*, 2023, Lect. Notes Comput. Sci., vol. 13990, pp. 113–139. doi: 10.1007/978-3-031-30044-8_5.

[13] Guillaume Allais, Robert Atkey, James Chapman, Conor McBride, and James McKinna, "A type- and scope-safe universe of syntaxes with binding: their semantics and proofs," *J. Funct. Program.*, vol. 31, p. e22, 2021, doi: 10.1017/S0956796820000076.

[14] Thorsten Altenkirch, James Chapman, and Tarmo Uustalu, "Monads Need Not Be Endofunctors," in *FoSSaCS*, 2010, Lect. Notes Comput. Sci., vol. 6014, pp. 297–311. doi: 10.1007/978-3-642-12032-9_21.

[15] Thorsten Altenkirch, Neil Ghani, Peter G. Hancock, Conor McBride, and Peter Morris, "Indexed containers," *J. Funct. Program.*, vol. 25, 2015, doi: 10.1109/LICS.2009.33.

[16] Thorsten Altenkirch, Conor McBride, and James McKinna, "Why Dependent Types Matter," 2005. [Online]. Available: https://people.cs.nott.ac.uk/psztxa/publ/ydtm.pdf

[17] Robert Atkey, "Parameterised notions of computation," *J. Funct. Program.*, vol. 19, no. 3–4, pp. 335–376, 2009, doi: 10.1017/S095679680900728X.

[18] Brian E. Aydemir *et al.*, "Mechanized Metatheory for the Masses: The PoplMark Challenge," in *Theorem Proving in Higher Order Logics*, 2005, Lect. Notes Comput. Sci., vol. 3603, pp. 50–65. doi: 10.1007/11541868_4.

[19] Jean-Philippe Bernardy, Patrik Jansson, and Ross Paterson, "Proofs for free - Parametricity for dependent types," *J. Funct. Program.*, vol. 22, no. 2, pp. 107–152, 2012, doi: 10.1017/S0956796812000056.

[20] David Binder, Marco Tzschentke, Marius Müller, and Klaus Ostermann, "Grokking the Sequent Calculus," *Proc. ACM Program. Lang.*, vol. 8, no. ICFP, 2024, doi: 10.1145/3674639.

[21] Stephen L. Bloom and Zoltán Ésik, *Iteration Theories: The Equational Logic of Iterative Processes*, EATCS Monogr. Theor. Comp. Sci. 1993. doi: 10.1007/978-3-642-78034-9.

[22] Edwin Brady, *Type-Driven Development with Idris*. 2017. isbn: 978-1-61729-302-3.

[23] Venanzio Capretta, "General recursion via coinductive types," *Log. Methods Comput. Sci.*, vol. 1, no. 2, 2005, doi: 10.2168/LMCS-1(2:1)2005.

[24] James Chapman, Roman Kireev, Chad Nester, and Philip Wadler, "System F in Agda, for Fun and Profit," in *MPC*, 2019, Lect. Notes Comput. Sci., vol. 11825, pp. 255–297. doi: 10.1007/978-3-030-33636-3_10.

[25] Alonzo Church, "An Unsolvable Problem of Elementary Number Theory," *Am. J. Math.*, vol. 58, no. 2, pp. 345–363, 1936, doi: 10.2307/2371045.

[26] Jesper Cockx, "Dependent Pattern Matching and Proof-Relevant Unification," 2017. [Online]. Available: https://lirias.kuleuven.be/1656778

[27] John H. Conway, *On Numbers and Games*. 1976. isbn: 0-12-186350-6.

[28] The Coq Development Team, "The Coq Proof Assistant." 2024. doi: 10.5281/zenodo.11551307.

[29] Pierre-Louis Curien, *Categorical Combinators, Sequential Algorithms, and Functional Programming*, Prog. Theor. Comput. Sci. 1993. doi: 10.1007/978-1-4612-0317-9.

[30] Pierre-Louis Curien and Hugo Herbelin, "The duality of computation," in *ICFP*, 2000, pp. 233–243. doi: 10.1145/351240.351262.

[31] Pierre-Évariste Dagand, "The essence of ornaments," *J. Funct. Program.*, vol. 27, p. e9, 2017, doi: 10.1017/S0956796816000356.

[32] Pierre-Évariste Dagand and Conor McBride, "A Categorical Treatment of Ornaments," in *LICS*, 2013, pp. 530–539. doi: 10.5555/2591370.2591396.

[33] Paul Downen and Zena M. Ariola, "Compiling With Classical Connectives," *Log. Methods Comput. Sci.*, vol. 16, no. 3, 2020, doi: 10.23638/LMCS-16(3:13)2020.

[34] Derek Dreyer, Georg Neis, and Lars Birkedal, "The impact of higher-order state and control effects on local relational reasoning," *J. Funct. Program.*, vol. 22, no. 4–5, pp. 477–528, 2012, doi: 10.1017/S095679681200024X.

[35] Jana Dunfield and Neel Krishnaswami, "Bidirectional Typing," *ACM Comput. Surv.*, vol. 54, no. 5, pp. 1–38, 2022, doi: 10.1145/3450952.

[36] Calvin C. Elgot, "Monadic Computation And Iterative Algebraic Theories," in *Logic Colloquium '73*, 1975, Stud. Log. Found. Math., vol. 80, pp. 175–230. doi: 10.1016/S0049-237X(08)71949-9.

[37] Marcelo Fiore and Dmitrij Szamozvancev, "Formal metatheory of second-order abstract syntax," *Proc. ACM Program. Lang.*, vol. 6, no. POPL, pp. 1–29, 2022, doi: 10.1145/3498715.

[38] Jean-Yves Girard, "Geometry of Interaction 1: Interpretation of System F," *Logic Colloquium '88*, Stud. Log. Found. Math., vol. 127. pp. 221–260, 1989. doi: 10.1016/S0049-237X(08)70271-4.

[39] Sergey Goncharov, Christoph Rauch, and Lutz Schröder, "Unguarded Recursion on Coinductive Resumptions," in *MFPS*, 2015, Electron. Notes Theor. Comput. Sci., vol. 319, pp. 183–198. doi: j.entcs.2015.12.012.

[40] Sergey Goncharov, Lutz Schröder, Christoph Rauch, and Maciej Piróg, "Unifying Guarded and Unguarded Iteration," in *FoSSaCS*, 2017, Lect. Notes Comput. Sci., vol. 10203, pp. 517–533. doi: 10.1007/978-3-662-54458-7_30.

[41] WebAssembly Working Group, "WebAssembly Specification." https://webassembly.org/specs/

[42] Kurt Gödel, "Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I," *Monatsh. f. Mathematik und Physik*, vol. 38, pp. 173–198, 1931, doi: 10.1007/BF01700692.

[43] Peter Hancock and Pierre Hyvernat, "Programming interfaces and basic topology," *Ann. Pure Appl. Log.*, vol. 137, no. 1–3, pp. 189–239, 2006, doi: 10.1016/j.apal.2005.05.022.

[44] André Hirschowitz and Marco Maggesi, "Modules over monads and initial semantics," *Inf. Comput.*, vol. 208, no. 5, pp. 545–564, 2010, doi: 10.1016/J.IC.2009.07.003.

[45] Tom Hirschowitz and Ambroise Lafont, "A unified treatment of structural definitions on syntax for capture-avoiding substitution, context application, named substitution, partial differentiation, and so on," *CoRR*, 2022, doi: 10.48550/ARXIV.2204.03870.

[46] Furio Honsell and Marina Lenisa, "Conway games, algebraically and coalgebraically," *Log. Methods Comput. Sci.*, vol. 7, no. 3, 2011, doi: 10.2168/LMCS-7(3:8)2011.

[47] Chung-Kil Hur, Georg Neis, Derek Dreyer, and Viktor Vafeiadis, "The power of parameterization in coinductive proof," in *POPL*, 2013, pp. 193–206. doi: 10.1145/2429069.2429093.

[48] Guilhem Jaber and Andrzej S. Murawski, "Compositional relational reasoning via operational game semantics," in *LICS*, 2021, pp. 1–13. doi: 10.1109/LICS52264.2021.9470524.

[49] Guilhem Jaber and Nikos Tzevelekos, "A Trace Semantics for System F Parametric Polymorphism," in *FoSSaCS*, 2018, Lect. Notes Comput. Sci., vol. 10803, pp. 20–38. doi: 10.1007/978-3-319-89366-2_2.

[50] André Joyal, "Remarques sur la théorie des jeux à deux personnes," *Gazette des Sciences Mathematiques du Québec*, vol. 1, no. 4, pp. 46–52, 1977.

[51] Stephen C. Kleene, "On the interpretation of intuitionistic number theory," *J. Symb. Log.*, vol. 10, no. 4, pp. 109–124, 1945, doi: 10.2307/2269016.

[52] Georg Kreisel, "Interpretation of Analysis by Means of Constructive Functionals of Finite Types," in *Constructivity in Math.*, 1959, pp. 101–128.

[53] Ugo Dal Lago, Francesco Gavazzo, and Paul Blain Levy, "Effectful applicative bisimilarity: Monads, relators, and Howe's method," in *LICS*, 2017, pp. 1–12. doi: 10.1109/LICS.2017.8005117.

[54] James Laird, "A Fully Abstract Trace Semantics for General References," in *ICALP*, 2007, Lecture Notes in Computer Science, vol. 4596, pp. 667–679. doi: 10.1007/978-3-540-73420-8_58.

[55] Søren B. Lassen, "Bisimulation in Untyped Lambda Calculus: Böhm Trees and Bisimulation up to Context," in *MFPS*, 1999, Electron. Notes Theor. Comput. Sci., vol. 20, pp. 346–374. doi: 10.1016/S1571-0661(04)80083-5.

[56] Søren B. Lassen, "Eager Normal Form Bisimulation," in *LICS*, 2005, pp. 345–354. doi: 10.1109/LICS.2005.15.

[57] Søren B. Lassen, "Normal Form Simulation for McCarthy's Amb," in *MFPS*, 2005, Electron. Notes Theor. Comput. Sci., vol. 155, pp. 445–465. doi: 10.1016/J.ENTCS.2005.11.068.

[58] Søren B. Lassen and Paul Blain Levy, "Typed Normal Form Bisimulation," in *CSL*, 2007, Lect. Notes Comput. Sci., vol. 4646, pp. 283–297. doi: 10.1007/978-3-540-74915-8_23.

[59] Søren B. Lassen and Paul Blain Levy, "Typed Normal Form Bisimulation for Parametric Polymorphism," in *LICS*, 2008, pp. 341–352. doi: 10.1109/LICS.2008.26.

[60] Paul Blain Levy, *Call-By-Push-Value: A Functional/Imperative Synthesis*, Semantics Structures in Computation, vol. 2. 2004. isbn: 1-4020-1730-8.

[61] Paul Blain Levy, "Similarity Quotients as Final Coalgebras," in *FoSSaCS*, 2011, Lect. Notes Comput. Sci., vol. 6604, pp. 27–41. doi: 10.1007/978-3-642-19805-2_3.

[62] Paul Blain Levy and Sam Staton, "Transition systems over games," in *CSL-LICS*, 2014, pp. 1–10. doi: 10.1145/2603088.2603150.

[63] Giuseppe Longo, "Set-theoretical models of λ-calculus: theories, expansions, isomorphisms," *Ann. Pure Appl. Log.*, vol. 24, no. 2, pp. 153–188, 1983, doi: 10.1016/0168-0072(83)90030-1.

[64] Jean-Jacques Lévy, "An algebraic interpretation of the λβ-calculus and a labeled λ-calculus," in *Lambda-Calculus Comput. Sci. Theor.*, 1975, Lect. Notes Comput. Sci., vol. 37, pp. 147–165. doi: 10.1007/BFB0029523.

[65] Kenji Maillard, Danel Ahman, Robert Atkey, Guido Martínez, Catalin Hritcu, Exequiel Rivas, and Éric Tanter, "Dijkstra monads for all," *Proc. ACM Program. Lang.*, vol. 3, no. ICFP, pp. 1–29, 2019, doi: 10.1145/3341708.

[66] Ian A. Mason and Carolyn L. Talcott, "Equivalence in Functional Languages with Effects," *J. Funct. Program.*, vol. 1, no. 3, pp. 287–327, 1991, doi: 10.1017/S0956796800000125.

[67] Conor McBride, "Clowns to the Left of me, Jokers to the Right," in *POPL*, 2008, pp. 287–295. doi: 10.1145/1328438.1328474.

[68] Conor McBride, "Let's See How Things Unfold," in *CALCO*, 2009, Lect. Notes Comput. Sci., vol. 5728, pp. 113–126. doi: 10.1007/978-3-642-03741-2_9.

[69] Conor McBride, "Kleisli Arrows of Outrageous Fortune," 2011. [Online]. Available: https://personal.cis.strath.ac.uk/conor.mcbride/Kleisli.pdf

[70] Conor McBride, "Ornamental Algebras, Algebraic Ornaments," 2011. [Online]. Available: https://personal.cis.strath.ac.uk/conor.mcbride/pub/OAAO/LitOrn.pdf

[71] Conor McBride and James McKinna, "The view from the left," *J. Funct. Program.*, vol. 14, no. 1, pp. 69–111, 2004, doi: 10.1017/S0956796803004829.

[72] The MetaCoq Development Team, "Metacoq," 2024. https://metacoq.github.io/

[73] Stefan Milius and Tadeusz Litak, "Guard Your Daggers and Traces: Properties of Guarded (Co-)recursion," *Fundam. Informaticae*, vol. 150, no. 3–4, pp. 407–449, 2017, doi: 10.3233/FI-2017-1475.

[74] James H. Morris, "Lambda-calculus models of programming languages," 1968. [Online]. Available: http://hdl.handle.net/1721.1/64850

[75] Leonardo de Moura and Sebastian Ullrich, "The Lean 4 Theorem Prover and Programming Language," in *CADE*, 2021, Lect. Notes Comput. Sci., vol. 12699, pp. 625–635. doi: 10.1007/978-3-030-79876-5_37.

[76] Evelyn Nelson, "Iterative Algebras," *Theor. Comput. Sci.*, vol. 25, pp. 67–94, 1983, doi: 10.1016/0304-3975(83)90014-2.

[77] Benjamin C. Pierce and David N. Turner, "Local Type Inference," in *POPL*, 1998, pp. 252–265. doi: 10.1145/268946.268967.

[78] Maciej Piróg and Jeremy Gibbons, "The Coinductive Resumption Monad," in *MFPS*, 2014, Electron. Notes Theor. Comput. Sci., vol. 308, pp. 273–288. doi: 10.1016/J.ENTCS.2014.10.015.

[79] Gordon D. Plotkin, "A structural approach to operational semantics," *J. Log. Algebraic Methods Program.*, pp. 17–139, 2004, doi: 10.1016/j.jlap.2004.05.001.

[80] Nicolas Pouillard, "Nameless, painless," in *ICFP*, 2011, pp. 320–332. doi: 10.1145/2034773.2034817.

[81] Damien Pous, "Coinduction All the Way Up," in *LICS*, 2016, pp. 307–316. doi: 10.1145/2933575.293456.

[82] Damien Pous and Davide Sangiorgi, "Enhancements of the bisimulation proof method," *Adv. Top. Bisimulation Coinduction*, Cambridge Tracts Theor. Comput. Sci., vol. 52. pp. 233–289, 2011. doi: 10.1017/CBO9780511792588.007.

[83] The Univalent Foundations Program, *Homotopy Type Theory: Univalent Foundations of Mathematics*. Institute for Advanced Study, 2013.

[84] David MacQueen, Mads Tofte Robin Milner Robert Harper, *The Definition of Standard ML*. 1997. isbn: 0-262-63181-4.

[85] Davide Sangiorgi, "A Theory of Bisimulation for the pi-Calculus," in *CONCUR*, 1993, Lect. Notes Comput. Sci., vol. 715, pp. 127–142. doi: 10.1007/3-540-57208-2_10.

[86] Steven Schäfer and Gert Smolka, "Tower Induction and Up-to Techniques for CCS with Fixed Points," in *RAMiCS*, 2017, Lect. Notes Comput. Sci., vol. 10226, pp. 274–289. doi: 10.1007/978-3-319-57418-9_17.

[87] Lucas Silver and Steve Zdancewic, "Dijkstra monads forever: termination-sensitive specifications for interaction trees," *Proc. ACM Program. Lang.*, vol. 5, no. POPL, pp. 1–28, 2021, doi: 10.1145/3434307.

[88] Matthieu Sozeau, "A New Look at Generalized Rewriting in Type Theory," *J. Formaliz. Reason.*, vol. 2, no. 1, pp. 41–62, 2009, doi: 10.6092/ISSN.1972-5787/1574.

[89] Kornel Szlachányi, "Skew-monoidal categories and bialgebroids," *Adv. Math.*, vol. 231, no. 3–4, pp. 1694–1730, 2012, doi: 10.1016/j.aim.2012.06.027.

[90] William W. Tait, "Intensional Interpretations of Functionals of Finite Type I," *J. Symb. Log.*, vol. 32, no. 2, pp. 198–212, 1967, doi: 10.2307/2271658.

[91] Alfred Tarski, "A lattice-theoretical fixpoint theorem and its applications," *Pac. J. Math.*, vol. 5, no. 2, pp. 285–309, 1955, doi: 10.2140/pjm.1955.5.285.

[92] Alan M. Turing, "On Computable Numbers, with an Application to the Entscheidungsproblem," *Proc. Lond. Math. Society*, no. 1, pp. 230–265, 1937, doi: 10.1112/plms/s2-42.1.230.

[93] Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic, "Interaction trees: representing recursive and impure programs in Coq," *Proc. ACM Program. Lang.*, vol. 4, no. POPL, pp. 1–32, 2020, doi: 10.1145/3371119.

[94] Irene Yoon, Yannick Zakowski, and Steve Zdancewic, "Formal reasoning about layered monadic interpreters," *Proc. ACM Program. Lang.*, vol. 6, no. ICFP, pp. 254–282, 2022, doi: 10.1145/3547630.